Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Polychronous Automata and Their use for Formal Validation of AADL Models

Thierry GAUTIER (✉)[1], Clément GUY [1], Alexandre HONORAT [1], Paul LE GUERNIC [1], Jean-Pierre TALPIN [1,*] Loïc BESNARD [2]

1 INRIA, Rennes-Bretagne-Atlantique Research Centre, 35042 Rennes cedex, France
2 CNRS, IRISA, 35042 Rennes cedex, France

Abstract This paper investigates how state diagrams can be best represented in the polychronous model of computation (MoC) and proposes to use this model for code validation of behavior specifications in Architecture Analysis & Design Language (AADL). In this relational MoC, the basic objects are signals, which are related through dataflow equations. Signals are associated with logical clocks, which provide the capability to describe systems in which components obey multiple clock rates. We propose a model of finite-state automata, called polychronous automata, which is based on clock relationships. A specificity of this model is that an automaton is submitted to clock constraints, which allows one to specify a wide range of control-related configurations, being either reactive or restrictive with respect to their control environment. A semantic model is defined for these polychronous automata, which relies on boolean algebra of clock signals. Based on a previously defined modeling method for AADL software architectures using the polychronous MoC, the proposed model is used as a formal model for the AADL behavior annex. This is illustrated with a case study involving an adaptive cruise control system.

Keywords Architecture modeling, Formal semantics, Finite-state automaton, Polychronous model, Synchronous concurrency, Code generation, AADL.

Received month dd, yyyy; accepted month dd, yyyy

E-mail: *Thierry.Gautier@inria.fr*

* Partially supported by Nankai University and by the National Science Foundation of China, under grant 61672074

1 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, among others. One way to help designers is to provide them with friendly usable tools supported by strong mathematical semantics. These formal models and methods allow designers to ensure the correctness of components used and/or to define them 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], and Lustre [4]), the polychronous model of computation implemented in the dataflow language called Signal [5, 6] and its environment Polychrony[1)], stands apart because of its capability to model multi-clocked systems. The synchronous paradigm consists of abstracting the non-functional implementation details of a system and allows one to benefit from a focused reasoning process on the logics behind the instants at which the system functionalities should be secured. The fundamental notion of *polychrony* is the capability to describe systems in which components obey multiple clock rates. In particular, the Signal language allows the opportunity to seamlessly model embedded systems at multiple levels of abstraction while

[1)] The toolset can be downloaded freely on the official website of Polychrony at Inria.

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 the co-modeling of the door management system of the A350 [7]. In this case study, door functionalities were modeled with synchronous Simulink[2)] and a system-level model of the mechanical equipment was specified in Architecture Analysis & Design Language (AADL) [8]. The Polychrony toolbox was then used to interpret the computations and communications specified in both models to synthesize schedulers for sequential and distributed simulation. The experiment was successful, and 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 dataflow oriented notation, thus there is sometimes some separation between an actual specification which may use for instance, state oriented descriptions, 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, first, we investigate the manner in which state diagrams can be best represented in the polychronous model of computation while 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) relationships, 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 manipulation of these automata without having to necessarily translate them into dataflow equations. In previous works [9, 10], with the aim of virtually prototyping embedded architectures, we defined a compositional semantic translation of AADL specifications into the polychronous model. We now refine this modeling by considering a polychronous automata model as a formal model for the AADL behavior annex.

Our work is motivated by practical reasons such as effective combination of heterogeneous programming notations (including dataflow and automata), and formal validation and virtual prototyping of timed software architectures. Our purpose is not to simply propose another extension of an existing programming language. Instead, we focus on the definition of a specific model of automata adapted to the polychronous model of computation, to be adopted as a common semantic model. Such automata have to relate events by expressing and specifying *clock relationships* (or clock *constraints*) among 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 the first sections of this paper, as they are sufficient to illustrate the basics of the model.

In the next section, we first consider some related work, concerning both the introduction of models of automata in synchronous languages, and the use of formal models to represent and validate AADL models. In Section 3, we review the main operators of the Signal language and their semantics. Then in Section 4 we define the boolean control algebra which is used to manipulate clock formulas. In Section 5, we describe the refinement of polychronous programs as automata. In Section 6, 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 7. Then, the principles of the semantic translation of AADL in the polychronous model are presented in Section 8. A concrete AADL case study, including behavior specifications as automata, is detailed in Section 9 to illustrate the formal validation of this modeling framework. Conclusions and future work are discussed in Section 10.

2 Related work

Automata were introduced several years ago in dataflow synchronous languages and are used every day in production tools such as SCADE [11]. 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 [12]. So-called modal models in particular are hierarchical models where the top level model comprises a finite-state machine, the states of which are refined into other models, possibly from different domains [13]. In our approach, heterogeneous designs are expressed in terms of common semantics, represented by the polychronous model of computation. In the software system Matlab/Simulink, which is largely

[2)] Refer to the MathWorks website.

accepted in the industry, the Stateflow notation [14] is used to describe modes in event-driven and continuous systems.

Mode-automata [15] were originally proposed to apply the advantages of declarative and imperative approaches to synchronous programming and to extend the functional dataflow paradigm of Lustre with the capability to model transition systems. Mode-automata have been combined with stream functions in Lucid Synchrone [16]. Related forms of hierarchical state machines include Statecharts [17] and their variants, including UML state machines [18], SyncCharts [19], associated in particular with Esterel and, more recently, SC-Charts [20], based on an improved definition of (sequential) constructivity [21]. DDFCharts [22], which compose finite-state machines and synchronous dataflow graphs, have multiple clocks; transitions can be driven by different clocks and the instants at which clocks synchronize are considered rendezvous communications.

Our approach may be distinguished from others by its capability to model multi-clocked systems and to express clock relationships through the automata. A first attempt was made a few years ago to define polychronous mode automata [23], but compared to our current proposal, it did not allow manipulation of automata as specific objects that can be used, for instance, to specify the dynamic properties of events. In this spirit, our approach, that is based on constraint specifications relating the occurrence of events, is similar to that taken by Lutin [24], but with a different purpose. In Lutin, statements describe sequences of non-deterministic atomic reactions expressing constraints on input/output values. The method is used mainly for test sequence specification and generation. In our proposal, constraints relate values and *clocks* of signals.

Concerning the second contribution of this article, i.e., the use of a formal model (in our case, polychronous automata) to represent and validate AADL behavior specifications, there have been many related works that have contributed to the formal specification, analysis, and verification of AADL models and its annexes. We limit ourselves to mentioning here the works which appear to be closest to our approach. In particular, RAMSES [25] presents the implementation of the AADL behavior annex. The behavior annex supports the specification of automata and sequences of actions to model the behavior of AADL programs and threads. Its implementation (OSATE) proceeds by model refinement and can be plugged in to Eclipse-compliant backend tools for analysis or verification. For instance, the RAMSES tool uses OSATE to generate C code for OSs complying with the ARINC-653 standard.

Synchronous modeling is central in [26], which presents a formal real-time rewriting logic semantics method for a behavioral subset of the AADL. This semantics rewriting can be directly executed in Real-Time Maude and provides a synchronous AADL simulator (as well as LTL model-checking). It is implemented by the tool AADL2MAUDE using OSATE.

Similarly, Yang et al. [27] define a formal semantics for an implicitly synchronous subset of the AADL, which includes periodic threads and data port communications. Its operational semantics is formalized as a timed transition system. This framework is used to prove semantics preservation through transformations from AADL models to the target verification formalism of a timed abstract state machine (TASM).

With respect to the related works mentioned here, we also annex the core AADL and its behavior annex with formal semantic frameworks to express executable behaviors and temporal properties, but we endeavor to structure and use them together within the framework of a more expressive multi-clocked synchronous model. Polychrony allows us to gain abstraction from the direct specification of executable, synchronous operations in the AADL, yet offers services to automate the synthesis of such locally synchronous, executable specification with global asynchrony, when or wherever needed.

3 The Signal language

We first introduce the Signal language and its semantics, before formalizing its boolean control algebra, which 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 [28] for an overview, or [5, 29] 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 $((\exists n \in \mathbb{N})(x = (x_t)_{t\in\mathbb{N},t\leqslant n})$ 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 $\mathbb{D}_x$, is *absent* and virtually holds an extra value denoted $\perp$, or is *completed* and never holds any actual or virtual value for all instants s such that $t \leqslant s$. The set of instants at which a

signal x is present is represented by its *clock* $\hat{x}$. Two signals are *synchronous* if and only if (iff) they have the same clock cycle. 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, 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 provide a description of its function using dataflow 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 the least domain fixed point. For a data domain $\mathbb{D}$, we consider a poset $(\mathbb{D}_\perp \cup \{\bullet, \#\}, \leqslant)$ such that $(\mathbb{D}_\perp, \leqslant)$ is flat, i.e., $x \leqslant y \Rightarrow x = y$, 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 noting s as a (possibly empty) signal in $\mathbb{D}^\infty$, v as a value in $\mathbb{D}$, and x as a value in $\mathbb{D}_\perp$. As usual, $|s|$ is the length of s, $s(i)$ is the i^{th} element of s, and $s_1.s_2$ 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 function $e : A \to \mathbb{D}_\perp$. For a behavior b on a set of signal variables A, and an integer $i \leqslant |b|$, $b(i)$ 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 the tuples of signals. Two behaviors, b_1 and b_2 are *stretch-equivalent* iff they only differ in non-final empty events (see [1] for more details).

Signal functions. A Signal function is an n-ary (with $n > 0$) function f that is total, strict, and continuous over domains [30] (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 an 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 as 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 v1`.

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: `s1 default s2`.

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: `s1 when s2`.

Process. An *equation* is a pair (x, E) denoted as `x := E`. An equation `x := E` associates the variable `x` with 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 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 (owing 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 is a strict continuous function or Kahn Process Network (KPN) [31], defined by the least upper bound satisfying the equations. It further satisfies the termination and stretching properties (it is closed for the stretching relationship). 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 [32]. The input-free equation `x := x $ init 0` is a typical example of a not timely deterministic process: `x` holds a sequence of constant values 0 separated by an undetermined number of silent transitions, characterized by an occurrence of $\perp$.

An example of a 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 both 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 the current state. The states being exclusive, the consistency property is satisfied. Partial definitions are also used to define *state variables*, the elements of which are present as frequently as necessary. When such a state variable is not explicitly defined, it retains 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* iff 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 `[:b]`), defined by `^b when b` (present, and *true*, when `b` is present and is *true*)
- *intersection* `x1 ^* x2`, defined by `^x1 when ^x2`
- *union* `x1 ^+ x2`, defined by `^x1 default ^x2`
- *difference* `x1 ^- x2`, defined by `[(not ^x2) default ^x1]`
- *synchronization* `x1 ^= x2`, defined as `(c := (^x1 = ^x2))/c`

Note that, in the syntax, the hat notation appears just before the symbol or variable and applies to: for instance, "`^*`" is a syntactic representation of "$\hat{*}$" (see below).

A *synchronized memory* `y := x cell c init x0` is defined by the composition of `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 when `c` is present and *true*. Finally, the Signal term `ll :: P` associates the label `ll` with the process `P`; a label `ll` 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`).

4 Boolean control algebra

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

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

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

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 $\forall x \in V \cup S$, $\widehat{x}, \widetilde{x} \in F_{V,S}$;
- 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, and $\widetilde{x}$ designates the clock at which the variable x is *true*. The formulas satisfy the boolean axioms: $(F_{V,S}, \hat{*}, \hat{+}, \neg, \mathbf{0}, \mathbf{1}_V)$ in 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 e.g., Shannon disjunctive forms (given an arbitrary total order of variables).

Note that, independently of the state variables which are distinguished here, this boolean control algebra is that of the standard polychronous model of computation. The formal tools based on this control algebra, such as the *clock calculus* (which builds a *clock hierarchy*), still apply in the exact manner when automata are considered. In both contexts, timing analysis mainly refers to analyzing clock relationships based on clock hierarchy.

Clock hierarchy. The clock hierarchy of a process is a component of its *Data Control Graph* (DCG). The DCG consists of a multigraph G and a *clock system* Σ. Refer to [28] 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; which is why it is called *clock hierarchy*. The clock hierarchy is defined by a relationship $\hat{\searrow}$ (*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, also contains the list of signals *signal*(n) whose clock shares 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* (i.e., 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 in Signal programs. We have mentioned a few properties such as allowing the rewrite of 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 considered as a sequence of morphisms allowing rewrite of programs into transformed Signal programs. The final steps (C code generation for instance) are simple morphisms of the transformed Signal programs. These transformation steps to sequential, clustered, or distributed code generation, are described in [28].

5 Refinement of processes as automata

So far, contrary to other synchronous languages, including dataflow ones such as Lustre (see for instance [33]), no explicit representation of automata was directly produced in the compilation of Signal programs, either for code generation purposes, 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 the clock.

A given Signal program may be seen as an automaton which contains a single state and a 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 activate 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 to obtain two states, and thus two new Signal programs. Each one of these two states is obtained using a rewritten version 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.

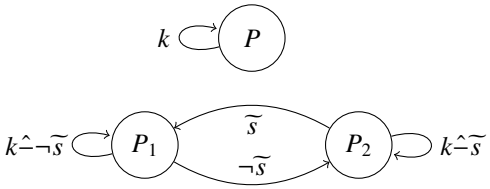


Figure 1 Refinement of state s in P by the automaton $A_1 = (P_1, P_2)$

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 begins with the valuation of a first state variable, s, in the program P, respectively with *true* and *false*, which produces 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 generally results 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 of 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 $\widetilde{\neg s}$. 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 (the same reasoning applies to 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 other 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 $0 < i \leqslant 2^{n-1}$ obtained from step $n-1$, by partitioning them according to an n^{th} state variable s_n and by repeating the same mechanism (see Figure 2).

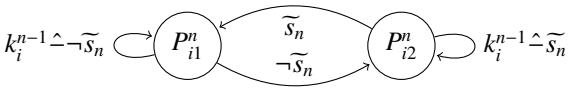


Figure 2 Refinement of state s_n in P_i^{n-1} by the automaton (P_{i1}^n, P_{i2}^n)

This would seem to be an expensive construction, at least in the worst case, as the size of the explicit automaton would be an exponential of its number of state variables. Fortunately, however, all Signal programs have a clock hierarchy, defined from the dominance relationship introduced in Section 4, which is used to represent the control flow of the program in a much more efficient manner (and actually optimal, as hierarchies can be normalized and allow a canonical representation). Concretely, most of the 2^n states are in practice inaccessible, because of dominance (of a state value by a clock).

Example. 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 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, because 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 manner, more frequent state variables are evaluated before less frequent ones. Note also that when some state variable is evaluated, the corresponding program is rewritten, using in particular constant propagation. This generally results in many simplifications, as a number of clocks may become null, thus eliminating their corresponding variables.

6 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 relationships. In the case of a reactive automaton, events from the environment are free to occur at their own rate. The automaton registers the occurrences of these events and causes its state to evolve according to these occurrences. In the case of a restrictive automaton, the automaton enforces constraints on the events that can occur. Events that are not explicitly allowed in some state are forbidden; this has an effect on the environment, which is in some way controlled by the automaton.

This can be illustrated by an automaton alternating two events, `a` and `b`. In Figure 3, events `a` and `b` are *constrained* to alternate by the clock relationship $a \hat{*} b = 0$, which imposes that they cannot occur simultaneously (the intersection of clocks `a` and `b` is never present). It should further be assumed that `b` cannot occur in S_1 and `a` cannot occur in S_2. Note that the occurrences of `a` and `b` are always controlled (or constrained), and the control is state dependent.

Figure 3 Restrictive behavior example

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`, see Figure 4, but it will not enforce them.

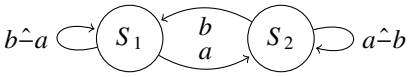


Figure 4 Reactive behavior example

In Signal, the observer will be implemented using a pair of equations that monitor alternation using a state variable `change` and stuttering using another variable `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 such as the famous `ABRO` is an object which lies between constrained and reactive

behaviors; it emits an output `o` immediately after receiving both inputs `a` and `b` and is reset when `r` occurs. So, signal `o` is controlled, while others are not.

```
module ABRO:
input a, b, r;
output o;
loop
  [ await a || await b ];
  emit o
each r
end module
```

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

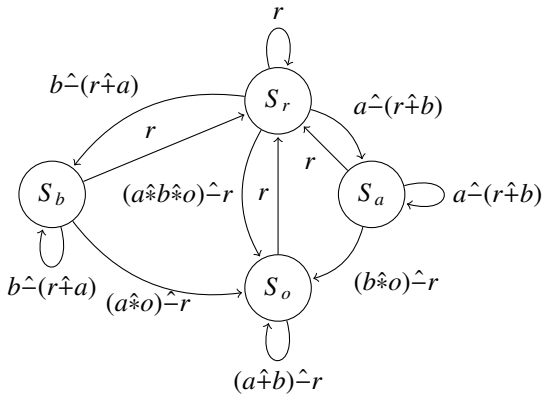


Figure 5 Automaton for ABRO

An equational definition of ABRO may be specified as follows in Signal, using a state variable to represent 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)
```

In the `ABRO` program, the inputs `a`, `b`, and `r` are never controlled, and the output `o` is obviously controlled. However, in Signal, it is worth noting that controlled variables are not limited to output signals.

7 Explicit structure of polychronous 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 preserve high-level semantic properties as well as the traceability of model transformations. We have hence chosen light-weight syntactic extensions to the Signal language 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. Standard equations can be used in these automaton processes to specify constraints or to define computations. Then the question arises as to whether these automata should be only a syntactic structure (in such a manner 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 formal manipulation of automata (which may or may not be, 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 making explicit some hidden boolean variables. It is therefore desirable to define a model of "polychronous automata" that allows a smooth integration within the polychronous model.

A basic statement for the definition of our automata is that *state change takes time.* This assumption is also made in [15, 16], and in SCADE 6. Consequently, there is a single state at each logical instant and there is no immediate transition. Such automata should be used to schedule steps, not the actions in a step. This drives toward 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 trigger(s). An automaton is *clocked*, that is, 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 view.

7.1 Notations for polychronous automata

To illustrate, let us write a syntactic representation of the automaton in Figure 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 must always be respected, can be expressed by a clock formula that is constrained to be null, here, `a ^* b ^= ^0` is expressed as `Never (a ^* b)`. Such constraints can always be expressed as a conjunction of `Never` formulas (which can also be specified as a single `Never` statement with several parameters) or of explicit synchronizations such as `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 demonstrate below that there are also implicit transitions.

First, we define 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`.

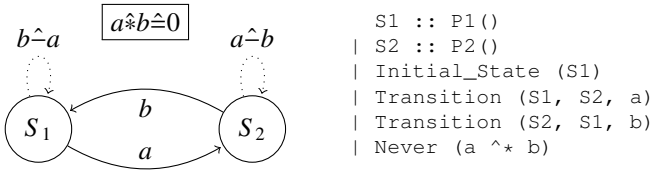


Figure 6 Polychronous automaton (with implicit stuttering loops)

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, the current state is `S1` and there is no enabled transition. All steps that are not explicitly forbidden are allowed. In the example of Figure 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.

7.2 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 4).

Definition 2 A *polychronous automaton A* is an epsilon-free automaton defined up to an 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 relationship;
- 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 and 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 $f \,\hat{\sharp}\, C_A = f$).

Remarks. A transition in T_A carries a boolean control formula 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), and 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. An 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 Figure 6 is defined by $A = (S_A, s_0, R_A, V_A, T_A, C_A)$ with

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

(a, b are events, and 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 $T_A((s_1, s_2)) = h)$.

7.3 Properties

Now the notions introduced previously can be formalized:

- The *control clock* of an automaton A is 1_A $(= \sum_{x \in V_A} \widehat{(x)})$, which represents 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 it is *a trigger* in s_1.
- The *trigger* of a state s, $trigger_A(s)$, is the upper bound of the triggers of $(s, *)$, where $(s, *)$ represents 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 $\tau(s) = 1_A \hat{-} (C_A(s) \hat{+} trigger_A(s))$. Hence the definition of implicit transitions is: when the stuttering clock $\tau(s)$ of a state s is not null, there is a silent implicit transition $\tau(s) : sR_As$ named a *step*. The standard 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 $(\exists s$ $(n-1)$-reachable in $A)$ $(\exists h)(h \hat{*} \widetilde{s}$ not null$)(h : sR_At)$.
- 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 $\tau(s) \hat{+} (\Sigma_{(s,t) \in R_A}(trigger_A((s,t)))) = 1_A$.
- 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).

7.4 Polychronous automata algebra

As in synchronous composition, the composition (or synchronous product) of polychronous automata corresponds to the conjunction of the behaviors specified by each automaton.

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

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

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

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

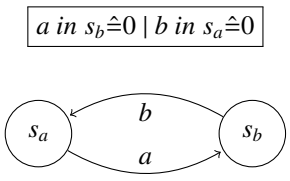
7.5 Discussion

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 dataflow language. They need not necessarily be translated by systems of equations on signals, although such a translation is, of course, possible. Note that using such a translation, the semantics of polychronous automata reduces to the semantics of standard polychronous programs. Comparable translations have been studied in previous work, such as [16] for example (it is not our purpose here to describe another similar translation). 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 article because it does not present new challenges with respect to previous studies.

Just as 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 triggered. 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. For example, let `t1` and `t2` be labels associated with two transitions in a given automaton, then the equation `o := t1 ^+ t2` expresses the action of emitting an event `o` as soon as one of these transitions is triggered.

A further remark can be made on permissive versus restrictive interpretation (recall that permissive is the default). The transformation of a given automaton from a permissive interpretation to a restrictive one is obtained as follows by *disabling its steps*. Given a (per-

missive) automaton A, the high-level operation "`strong A`" consists of 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 manner) "`h in S`" the clock `h ^* [S]`. Consider as example the `A` automaton represented in Figure 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 obtain the automaton represented in Figure 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

8 Polychronous AADL modeling

In previous sections, we have introduced a model of polychronous automata in the polychronous model of computation. This polychronous MoC has been used previously as a semantic model for systems described in the core AADL standard. The core AADL is extended with annexes, such as the behavior annex, which allows more precise specification of architectural behaviors. The translation from AADL specifications into the polychronous model should take into account these behavior specifications, which are based on descriptions of automata. In this section, we first introduce briefly AADL, then we describe the principles of a compositional semantic translation of AADL specifications into the polychronous model, and we discuss a global view of the AADL with respect to the Signal toolchain.

8.1 A short introduction to AADL

AADL is a Society of Automotive Engineers (SAE) standard, dedicated to modeling embedded real-time system architectures. It describes the structure of systems as an assembly of software components allocated on execution platform components along with timing constraints.

Architecture. Three families of components are provided in AADL:

- software application components, which include process, thread, thread group, subprogram, and data components;
- execution platform components, which include processor, virtual processor, memory, device, bus, and virtual bus components;
- composite components, which describe systems containing execution platform, application software, or other composite components.

Each component has a type, which represents the functional interface of the component and externally observable attributes. Each type may be associated with zero, one, or more implementation(s) that describe the contents of the component as well as the connections between components. The components communicate via data ports, event ports, and event data ports.

Properties. AADL properties provide information about model elements of an AADL specification. For example, the property `Dispatch_Protocol` is used to describe the dispatch type of a thread. Property associations in component declarations assign a particular property value (e.g., `Periodic`) to a particular property (e.g., `Dispatch_Protocol`) for a particular component. Timing properties associated with threads, such as `Input_Time` or `Output_Time` of ports, assure an input-compute-output model of thread execution. Binding properties assign hardware platform(s) to the execution of application components.

AADL timing execution model. Threads are dispatched either periodically, by the arrival of data or events on ports, or by the arrival of subprogram calls from other threads, depending on the thread type. Three event ports are pre-declared: *dispatch*, *complete*, and *error* (Figure 8). A thread is activated

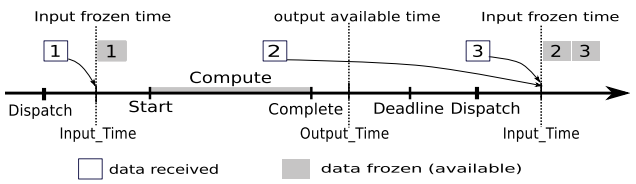


Figure 8 Execution time model for an AADL thread

to perform the computation at *start* time, and has to be finished before a *deadline*. A *complete* event is sent at the end of the execution. The received inputs are frozen at a specified point (*Input_Time*), by default the *dispatch* time, which means that the content of the port does not change during the execution of a dispatch, even though the sender may send new values. For example, the values 2 and 3 (Figure 8) arriving after the first *Input_Time* will not be processed until the next *Input_Time*. As a result, the performed computation is not

affected by a new input arrival until an explicit request for input is issued. Similarly, the output is made available to other components at a specified point of *Output_Time*, by default at the *complete* (resp., *deadline*) time if the associated port connection is an immediate (resp., delayed) communication.

Behavior. The behavior annex provides an extension to the AADL core standard so that complementary behavior specifications can be attached to AADL components. The behavior is described with a state transition system equipped with guards and actions.

8.2 AADL modeling

The compositional transformation from AADL to Signal is based on considering the AADL time model in the polychronous model.

8.2.1 AADL time model in Polychrony

The key idea for modeling the computing latency and communication delay in Signal is to preserve the ideal view of instantaneous computations and communications while delegating computing latency and communication delays to specific "memory" processes that introduce delays, and provide well-suited synchronizations to timed signals.

A "memory" process $o = f_m(i, b)$ is used to repeat the input signal i on the instants of a boolean signal b. The result o contains values of i when i is present and b is *true*, and the last value of i when i is absent and b is *true*:

$$o = f_m(i, b) \equiv$$

$$\forall t > 0: \quad o_t = \begin{cases} i_t & \text{if } i_t \neq \perp, \text{ and } b_t = true \\ i_{pred(t)} & \text{if } i_t = \perp, \text{ and } b_t = true, \\ & pred(t) = max\{k < t \mid o_k \neq \perp\} \\ \perp & \text{otherwise} \end{cases}$$

Input freezing. Let $f(x)$ represent the result of the behavior f of a given *in* port on its input signal x (e.g., f can be a FIFO to represent queued events or can be an event data port). A port $y = f(x)$ gives the available output y from the currently received input x:

$$y = f(x) \equiv \forall t > 0: \ (x_t \neq \perp \Leftrightarrow f(x_t) \neq \perp) \wedge (y_t = f(x_t)))$$

The freezing of x at t, denoted here by $x \blacktriangleright t$, is a function that takes an input x, a frozen time event t, and produces a new signal z at time t:

$$z = x \blacktriangleright t \equiv z = f_m(f(x), t)$$

Thread activation. We use $th(z_1, z_2, \dots)$ to represent the original computation of a thread *th* with frozen inputs $z_1, z_2, \dots$ An activation condition *start* is introduced so that the thread *th* is activated to perform computation at *start*. This is denoted as $th'(z_1, z_2, \dots, start)$, where the inputs $z_1, z_2, \dots$ are memorized at *start*:

$$th'(z_1, z_2, \dots, start) \equiv th(z'_1, z'_2, \dots) \text{ where } \forall i, z'_i = f_m(z_i, start)$$

Output sending. Similar to *in* ports, let $g(y)$ represent the behavior of an *out* port. The sending function, denoted here as $y \triangleright t$, is such that the generated output of $g(y)$ is held and sent out at time t:

$$w = y \triangleright t \equiv w = f_m(g(y), t)$$

8.2.2 Compositional transformation

The translation from AADL to Signal is recursive. A package, which represents the root of an AADL specification, is transformed into a Signal module, the root of a Signal program, allowing it to describe an application in modular fashion. The rest of the transformation proceeds modularly by an inductive translation of the AADL concepts of a given model or source text. Each AADL component implementation is translated into a Signal process composed of the following elements:

- an interface consisting of input/output signals translated from the features (ports) provided by the component type,
- additional control signals that may be added depending on the component category (*Dispatch* and *Deadline* for a thread),
- a body, itself composed of subcomponents, subprogram call sequences, connections, component properties, and a transition system specifying the functional behavior.

Threads. Let us sketch the principles of the translation of threads, which are the main executable and schedulable components. The six types of threads (*periodic, aperiodic, sporadic, timed, hybrid, and background*) are discriminated by the dispatch; a dispatch request is periodic, or is triggered by an event (or event data) arriving, etc. The different threads are implemented in the same manner, but the "dispatch" signal is generated differently.

An AADL thread component is implemented as a Signal process (Figure 9): it is composed of processes that represent its behavior, property, ports, and subcomponents. *Dispatch*, *Complete*, and *Error* are pre-declared ports in

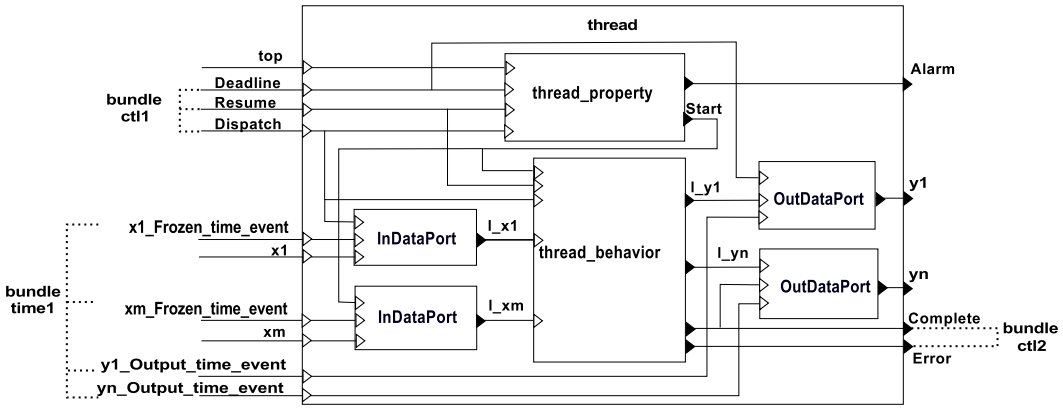


Figure 9 Thread modeling

AADL. They are represented as input/output signals (`Dispatch`, `Complete`, and `Error`). According to the AADL semantics, the signals `Resume` and `Deadline` are added as inputs, which are generated by the scheduler. `Start` is represented as the first `Resume` after a `Dispatch` signal. It is computed in the `xx_Thread_property` subprocess. The event signals (`x1_Frozen_time_event`, `y1_Output_time_event...`) are represented as input signals, which are produced by the scheduler.

```
process xx_thread =
  (? x1,...;
     event Dispatch, Resume, Deadline;
     event x1_Frozen_time_event,...,
           y1_Output_time_event,...;
! y1,...; event Complete, Error;)
     (| (...) := xx_thread_behavior
                 (..., Dispatch, Start, Resume)
      | Start := xx_thread_property
                 (Dispatch, Resume, Deadline)
      | x1_InDataPort{...}(...)
      |...
      | y1_OutDataPort{...}(...)
      |...
      | sxx1()
      |...
      |)
  where
    event Start;
    process xx_thread_behavior(...);
    process xx_thread_property
      (?event Dispatch,Resume,Deadline;!Start;);
    process x1_InDataPort{...}(...);
    process y1_OutDataPort{...}(...);
    process sxx1();
...
  end;
```

The translation (as polychronous automata) of the transition system that provides the functional behavior of the thread will be illustrated in the case study described in Section 9.

8.3 Complete toolchain

A global view of the toolchain for modeling, timing analysis, and verification of the AADL models in the polychronous MoC is given in Figure 10.

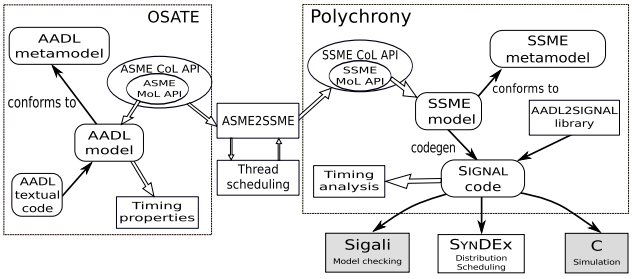


Figure 10 Global view of the AADL wrt the Signal toolchain

The AADL model, which conforms to the AADL metamodel, is captured as AADL textual code in the OSATE toolkit[3]. The timing properties provide detailed timing specifications related to the AADL model. A model transformation toolchain, ASME2SSME, performs analyses on the ASME models (AADL Syntax Model under Eclipse) and generates Signal models in the Signal Syntax Model under Eclipse (SSME). The SSME models can be transformed to Signal textual code within Polychrony.

An AADL2SIGNAL library provides common Signal processes, reducing significantly the transformation cost. The timing properties represented as Signal clocks are calculated and analyzed in the compilation of Signal programs. Then, executable code can be generated for simulation. Associated tools, such as Sigali [34] and SynDEx [35], can be used for further verification and validation. The global architecture, in the context of Polychrony, is presented in Figure 11.

[3] Refer to the OSATE website.

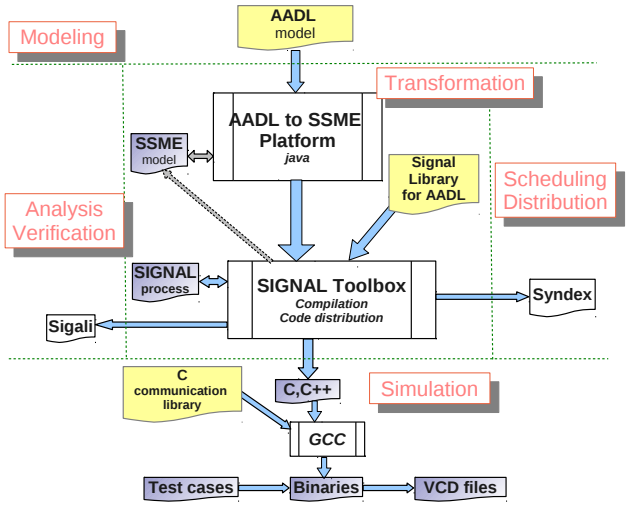


Figure 11 Global architecture

9 Case study: an Adaptive Cruise Control system

In this section, we present the AADL modeling of an adaptive cruise control (ACC) system, a highly safety-critical system embedded in modern cars, and we show how polychronous automata allow verification of properties in such a heterogeneous system.

9.1 Adaptive Cruise Control systems

Adaptive Cruise Control systems are embedded systems in cars. They receive information from different sensors and can act on the speed of the vehicle, notably in the case of the risk of a collision. As such, ACC systems are safety-critical systems, requiring careful design and verification.

> An Adaptive Cruise Control system is an optional cruise control system for road vehicles that automatically adjusts the vehicle speed to maintain a safe distance from vehicles ahead. [...] Control is imposed based on sensor information from on-board sensors [36].

ACC systems pursue two main goals: to automatically follow a preset speed (as do classic cruise control systems), improving comfort of the driver, reducing fatigue, and preventing subconscious violation of speed limits, and adapt the vehicle's speed to maintain a safe distance from a vehicle ahead to prevent collisions.

For this, an ACC system receives information from different sensors: speedometer, lidar/radar to detect vehicles or obstacles ahead, and wheel sensors to adjust the position of the lidar/radar. It also receives information from the driver through buttons used to set the preferred speed and to activate/deactivate the system. Depending on the situation (presence of an obstacle or not, activation of the cruise control or not), it computes the acceleration/deceleration for the vehicle to reach the needed speed, the preferred speed of the driver if there is no obstacle and the cruise control is on, or the speed of the vehicle ahead if one is detected. Finally, it acts on the vehicle through its brakes and throttle.

ACC systems are thus highly safety-critical systems which must satisfy multiple requirements linked to multiple perspectives:

- from the timing and scheduling perspective, all threads must meet their deadlines and the overall task of reacting to the presence or absence of an obstacle must meet a maximum reaction time;
- from the logical perspective, the system must be free of deadlocks and race conditions;
- from the security perspective, critical software components (processes or systems) must be protected from less critical components, and thus are executed on dedicated processors;
- from the consumption perspective, the system must draw minimal power from the car battery, thus processors must run on the minimal possible frequency;
- from the cost perspective, the overall cost of the system should be minimal, which means minimizing hardware component size and complexity.

All these requirements interact in many ways, and thus cannot be checked on only a subset of the system. Minimizing hardware components size and processor frequency to reduce costs and power consumption of the vehicle will affect execution time and scheduling of the different tasks (e.g., a less powerful processor will take more time to execute a task and smaller or slower buses will slow data communication). Similarly, changing a processor with another to reduce costs may lead to impossible bindings between processes and processors (e.g., some embedded processors may be unable to execute complex operations).

To collectively analyze these requirements and to ensure that choices made to satisfy one requirement do not break another (i.e., the set of requirements is compatible), the full system must be analyzed.

The AADL model of the ACC system can be transformed

into a Signal program, where behavior is described through polychronous automata and properties are used as constraints over the system. It is then possible to use the Polychrony framework to analyze and simulate the system. The Polychrony framework targets the two first perspectives presented above: time and scheduling, and logical perspectives. Requirements depending on other perspectives should be addressed using complementary analyzes.

9.2 AADL modeling of the Adaptive Cruise Control system

In the following, we present the modeling of a simplified ACC system using AADL. AADL provides a combination of visual and textual modeling: we use the visual representation for the overview of the system and details about connections, communications, and behavior automata, and we use the textual representation for details about requirements and properties such as types, numbered or enumerated properties, and about behavior actions.

9.2.1 Architecture of the Adaptive Cruise Control system

Figure 12 presents an overview of the system, consisting of:

- devices, such as sensors (speedometer, radar, wheel sensor), console with buttons and display, throttle, and brakes;
- buses allowing subsystems to communicate with each other and with devices;
- controller and console subsystems.

Each of the two subsystems consists itself of hardware components, such as processors, memories, and buses; and software components consisting of processes containing threads. Figure 13 presents the controller subsystem and its components: one processor, one memory, one bus connecting the processor and the memory, and one controller process. The controller process itself contains four threads, one for each sensor, and the *ComputeActionThread*, which is responsible for sending *speed up*, *slow down*, or *complete stop* signals to the throttle and brakes of the vehicle.

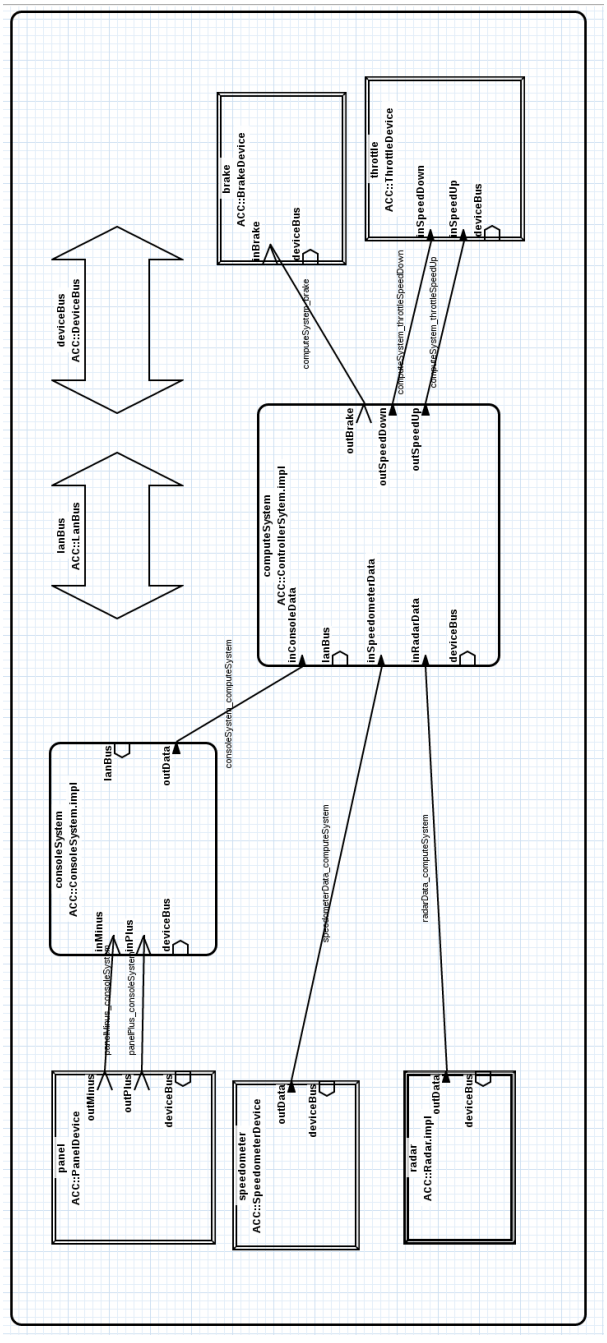


Figure 12 Overview of the Adaptive Cruise Control system modeled with AADL.
Double-lined rectangles represent devices, double-arrows are buses, and rectangles with rounded corners are systems and subsystems.

9.2.2 Behavior of the Adaptive Cruise Control system

In AADL, system behavior is specified through the so-called behavior annexes attached to components. Behavior annexes specify the behavior of AADL components (threads and subprograms) through state transition systems with guards and actions, which can be expressed as polychronous automata.

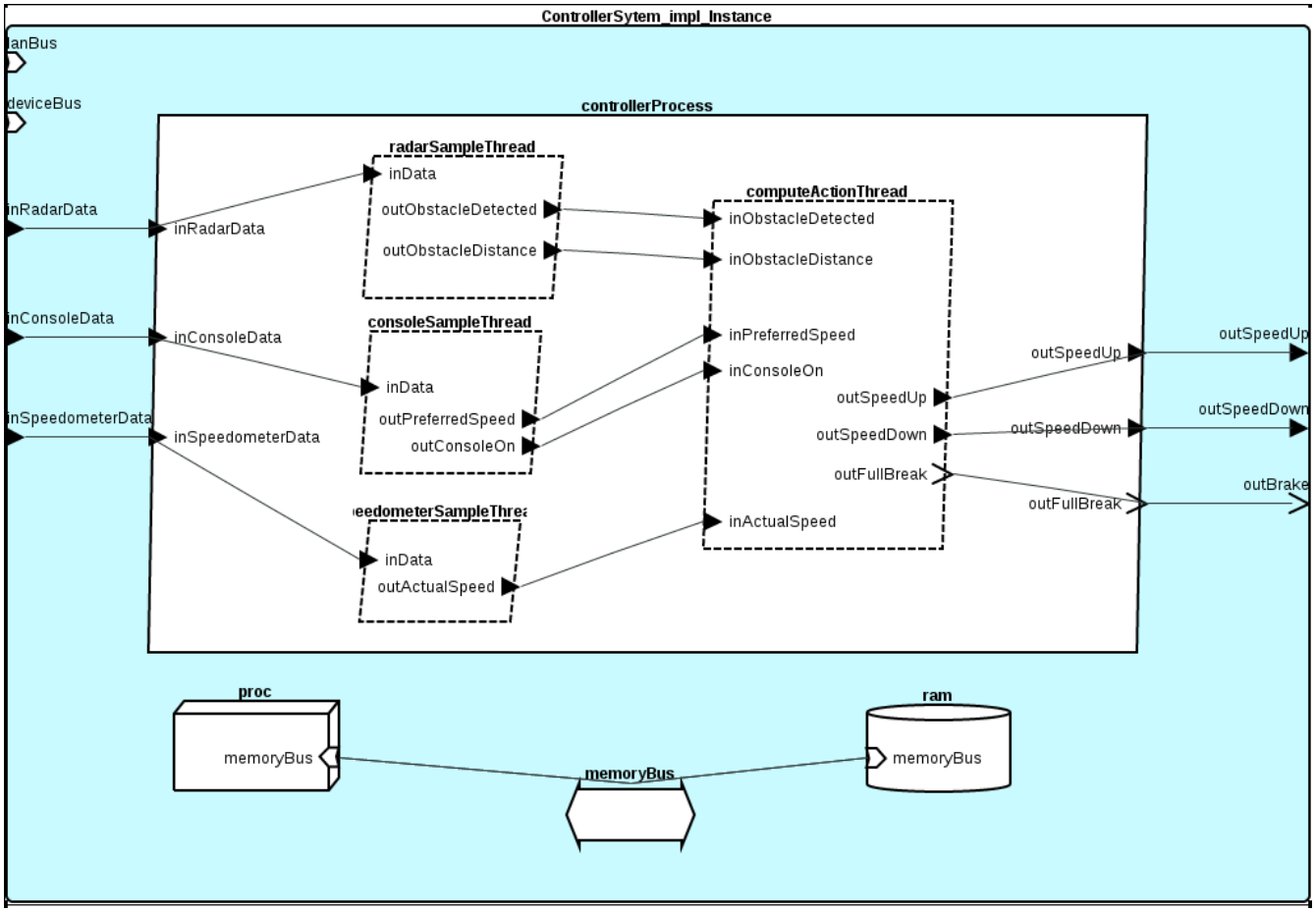


Figure 13 Controller subsystem of the Adaptive Cruise Control system modeled with AADL.
Rectangles represent processors, double-arrows are buses, cylinders are memories, and rhombuses are processes and threads.

Figure 14 shows the state transition system describing the behavior of the *ComputeActionThread* thread, which is responsible for processing the correct behavior the system should adopt (slow down, speed up, or keep the speed constant) depending on the situation.

For readability, guards and actions have been omitted. In the case of this state transition system, guards are tests performed on input signals (are they present or not, and value comparison if they are present), and actions are of two types: either the sending of a signal through one of the output ports of the thread, or the computation of an intermediate value, such as the vehicle speed relative to the obstacle, or the acceleration/deceleration needed to reach a given speed. For an example of such guards and actions, see Listing 1 which presents the complete transition from the *Started* state to the *Detected* one.

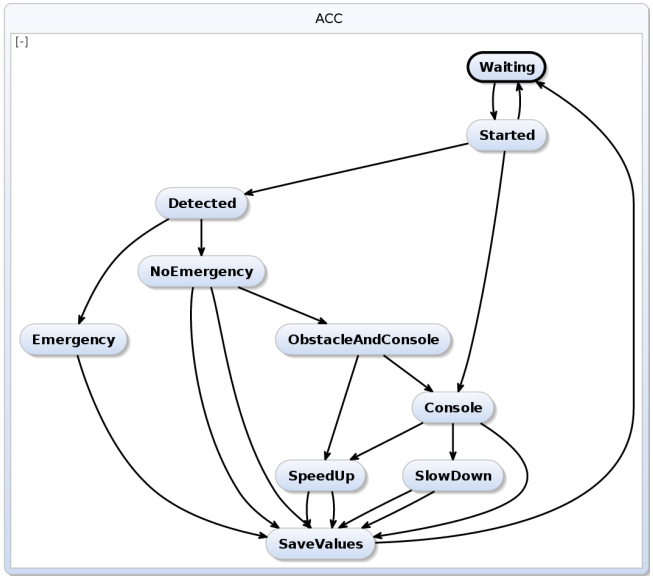


Figure 14 State transition system for the *ComputeActionThread* thread.

```
Started -[inObstacleDetected]-> Detected
    {obstacle_distance:=inObstacleDistance};
```

Listing 1 Transition between two states of the state transition system described by the behavior annex of the *ComputeActionThread* thread. The guard of the transition is indicated between brackets and the action between braces.

The state transition system starts in the *Waiting* state, waiting for its thread to be dispatched, and to pass to *Started* state. The *Waiting* state is a *complete* one, that is, a state in which a thread pauses its execution when entering, waiting for a new dispatch.

After entering the *Started* state, depending on the inputs, the state transition system can pass into the *Detected* state (the system detected an obstacle) or the *Console* state (the system did not detect an obstacle and the cruise control is on), or go back to the *Waiting* state (the system did not detect an obstacle and the cruise control is off).

In the *Detected* state, the system must decide the status of the situation: if the obstacle is in an unsafe range, the system goes into the *Emergency* state and its next transition will send a signal to the brakes to stop the vehicle; if the obstacle is outside this range, the system enters the *NoEmergency* state and then determines whether it should slow down to adapt its speed to the obstacle speed, speed up, or keep the speed constant (each transition sending the corresponding signal to the throttle after the computation of the needed acceleration/deceleration). The same occurs in the *Console* state depending on the current speed of the vehicle and the speed preset by the driver. After saving useful values (e.g., current speed, current obstacle speed, and obstacle distance in the *SaveValues* state, the state transition system returns to the *Waiting* state, waiting for the next dispatch of its thread.

9.2.3 Properties of the Adaptive Cruise Control system

Requirements, such as thread deadline, sensor period, memory and bus size, processor frequency, and scheduling policies, are modeled using AADL properties.

Timing and scheduling requirements on a thread can be expressed through AADL properties such as dispatch protocol (`Dispatch_Protocol`), period in case of a periodic dispatch (`Period`), execution deadline (`Deadline`), and worst-case execution time (`Compute_Execution_Time`). For example, Listing **??** presents these properties attached to the *ComputeActionThread* thread.

```
thread implementation ComputeActionThread.impl
   properties
      Dispatch_Protocol => Periodic;
      Period => 50 ms;
      Deadline => 40 ms;
      Compute_Execution_Time => 20 ms;
end ComputeActionThread;
```

Listing 2 Timing and scheduling properties of the *ComputeActionThread* thread implementation (double right arrows associate values to properties).

Logical requirements, such as the absence of deadlocks and race conditions, are not explicitly expressed through properties, but are automatically checked by the Polychrony framework.

9.3 Refinement and verification of system requirements using the polychronous model of computation

The AADL model of the ACC system can be transformed into a Signal program, where behavior is described through polychronous automata and properties are used as constraints over the system. It is possible to use the Polychrony framework to analyze the system owing to this transformation. For Synchronous Data Flow (SDF) applications [37], scheduling properties can be computed during the transformation.

9.3.1 Transformation of the behavior annex to Signal

An overview of the compositional transformation from AADL to Signal is presented in Section 8. We will now focus on the translation of a state transition system described through an AADL behavior annex to a polychronous automaton described in Signal. The rules formally describing the semantics of the behavior annex and its translation in transition systems represented as polychronous automata are detailed in [38]. Here, we simply provide the intuition of this translation as Signal automata through our case study.

Such an automaton is described using the Signal syntactic extensions presented in Section 7. Listing 3 shows the declaration and interface (input and output signals) of the Signal automaton process corresponding to the state transition system of the *ComputeActionThread* thread presented above.

States are simply declared as Signal labels. Listing **??** presents declarations of a few states from the Signal automaton obtained from the transition system of the *ComputeActionThread* thread of the Adaptive Cruise Control system.

Transitions are declared as a label attached to the special Signal process `Automaton_Transition` which takes as parameters the labels of the source and destination states, and the condition expression corresponding to the AADL guard of the transition. Moreover, transition processes declare which Signal equations to compute when triggering a transition. Listing 5 presents the Signal declaration corresponding to the AADL transition presented in Listing 1.

Once the AADL model of a system is transformed into a Signal program, one can analyze the program using the Polychrony framework to check if the logical requirements over the entire system are met. Note that timing and scheduling requirements are checked during the transformation itself.

9.3.2 Verification of timing and scheduling requirements

Polychrony provides a schedulability analysis for periodic programs obtained from AADL models. In particular, it can detect non-schedulable systems owing to the timing properties (mainly the period and the WCET—Worst Case Execution Time). Moreover, with some extra information (port rates), periods can be left undefined; they will be computed by calling a standalone plugin named ADFG (this plugin is based on the work of Bouakaz [39,40]).

The schedulability test is triggered during the translation of an AADL processor device into Signal, provided that all threads of the unique process bound to this processor are periodic and as a minimum have a defined `Compute_Execution_Time`. The test consists of the following two steps:

1. Polychrony checks that all threads have a defined period and tries to compute it if this is not the case (in our case study, all periods are defined so this step is skipped). The periods can be computed if all threads define connections between them (such that they form a connected graph) and have `Input_Rate` and `Output_Rate` properties set on their connection ports. The computation is performed by ADFG, which ensures that the deduced periods imply a schedulable system.
2. A simple utilization factor test is performed on the current processor (which is useful especially if the periods are user-provided). If the utilization factor is less than one, each thread period is converted into an affine

```
automaton ComputeActionThread_behavior =
     (? event Dispatch, Resume, Deadline;
        boolean inObstacleDetected;
        integer inObstacleDistance;
        integer inPreferredSpeed;
        real inActualSpeed;
        boolean inConsoleOn;
   ! event outFullBreak;
        real outSpeedUp;
        real outSpeedDown;
        event Complete;
        event Error;
     )
```

Listing 3 Signal declaration and interface of the automaton corresponding to the transition system of the *ComputeActionThread* thread.

```
label S_Waiting, S_Started, S_Detected [...]
```

Listing 4 Extracts of the Signal declaration of the states of the automaton corresponding to the transition system of the *ComputeActionThread* thread.

```
(| t1 :: Automaton_Transition(S_Started, S_Detected,
   [:inObstacleDetected])
 | on t1 :: (| obstacle_distance ::= inObstacleDistance |)
 |)
```

Listing 5 Signal translation of the AADL transition presented in Listing 1.

clock relationship between the fastest clock and each thread control signal (Dispatch, Resume, Deadline, and Initialize) with corresponding phases. Otherwise, the AADL to Signal transformation continues on the other AADL elements without generating the current processor's scheduling equations.

Note that this method of defining the periodic thread control signals implies that the generated Signal code does not take into account possible thread preemptions; it is oriented toward functional simulation only.

The current implementation has some limitations: e.g., all temporal properties (periods, etc.) must be expressed in the same time unit, and the scheduled threads must exist in the same unique process of a processor. Future work will be dedicated to release these limitations and to use an exact scheduling test in Step 2, instead of the utilization factor (which is only a necessary condition) for Earliest Deadline First (EDF) and Deadline Monotonic (DM) scheduling algorithms. Besides, Step 1 is based on ADFG, which can compute the best periods, deadlines, and communication buffer sizes given a cyclo-static dataflow graph to maximize the throughput. This is more than what is currently used, so a possible improvement would be to reuse all the ADFG results to refine more AADL properties related to scheduling.

9.3.3 Verification of logical requirements

Polychrony also allows detection of deadlocks in the AADL model of a given system. After the transformation of the AADL model to a Signal program (following the compositional transformation presented in Section 8), the Polychrony framework calculates the *Graph of Conditional Dependencies* of the Signal program and computes the product clock of each cycle in this graph (a cycle representing a potential deadlock) [41]. The *Graph of Conditional Dependencies* is a labeled, directed graph where:

- vertices are the signals and clock variables;
- edges indicate data dependencies among signals and clock variables;
- labels represent the conditions for which the dependencies are valid.

In this graph, cycles represent possible deadlocks e.g., cyclical data dependencies between signals and/or clock variables. However, because dependencies are conditioned, Polychrony only needs to consider instants where all dependencies are valid. To do so, it computes the product of the labels of each edge in the cycle. If this product is null, there is no instant where all the dependencies are valid at the same time, and thus there is no possible deadlock.

If the product clock of every cycle in the *Graph of Conditional Dependencies* of the Signal program is null, then the program (and the AADL model from which it has been obtained) is *deadlock-free*.

In the case study, a race condition due to an error in the state transition system of the *ComputeActionThread* was detected by the clock calculus of Polychrony.

In addition, model checking-based formal verification can be performed in the Polychrony framework using the associated Sigali tool [34]. Properties such as invariance, reachability, and attractivity can be checked by Sigali. Algorithms for computing state predicates are also available in the tool.

10 Conclusion

We have presented a model of finite-state automata, called polychronous automata, that integrates smoothly with dataflow equations in the polychronous model of computation. Automata 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 have been integrated in the open-source version of the Polychrony framework through lightweight syntactic extensions of the Signal language. They may be used to specify behaviors (and constraints) and to *abstract* behaviors, as the result of a formal calculus.

This formal model of automata supports the recommendations adopted by the SAE committee on the AADL to implement a timed and synchronous behavioral annex for the standard [42]. The model of polychronous automata has been provided as a semantic model for our proposal as an extension of the AADL behavior annex.

An experimental implementation of the semantic features of this "timing annex" enriches the already existing transformation of AADL models to Signal programs to consider the behaviors of AADL models. This transformation will be integrated in the POP environment for Eclipse[4)]. The implementation has been tested with an adaptive cruise control case study, developed with Toyota ITC. Adaptive Cruise

[4)] *Polychrony on PolarSys*, an Eclipse project in the PolarSys Industry Working Group (refer to the POP project on the PolarSys website.)

Control systems are highly safety-critical embedded vehicle systems which must satisfy multiple interdependent requirements. Polychronous automata allow us to define a complete semantic model for AADL specifications of such systems. We provide tools to this semantic model for verifying and analyzing properties (such as deadlock freedom and schedulability) over an entire system.

References

1. Le Guernic P, Talpin J P, Le Lann J C. Polychrony for system design. Journal of Circuits, Systems and Computers, 2003, 12(03). http://hal.inria.fr/docs/00/07/18/71/PDF/RR-4715.pdf
2. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages twelve years later. Proceedings of the IEEE, Special issue on Modeling and Design of Embedded Systems, 2003, 91(1). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.1117
3. Berry G, Gonthier G. The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming, 1992, 19(2): 87–152. http://dx.doi.org/10.1016/0167-6423(92)90005-V
4. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous dataflow programming language LUSTRE. Proceedings of the IEEE, 1991, 79(9): 1305–1320
5. Le Guernic P, Gautier T, Le Borgne M, Le Maire C. Programming real-time applications with Signal. Proceedings of the IEEE, 1991, 79(9): 1321–1336. http://hal.inria.fr/inria-00540460
6. Gamatié A. 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. Yu H, Ma Y, Glouche Y, Talpin J P, Besnard L, Gautier T, Le Guernic P, Toom A, Laurent O. System-level co-simulation of integrated avionics using Polychrony. In: ACM Symp. on Applied Computing. March 2011. http://hal.inria.fr/inria-00536907/en/
8. Aerospace Standard AS5506A: Architecture Analysis and Design Language (AADL), 2009
9. Yu H, Ma Y, Gautier T, Besnard L, Talpin J P, Le Guernic P, Sorel Y. Exploring system architectures in AADL via Polychrony and SynDEx. Frontiers of Computer Science, 2013, 7(5): 627–649
10. Yu H, Ma Y, Gautier T, Besnard L, Le Guernic P, Talpin J P. Polychronous modeling, analysis, verification and simulation for timed software architectures. Journal of Systems Architecture, 2013, 59(10): 1157–1170
11. Berry G. Scade: Synchronous design and validation of embedded control software. In: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems. Springer, 2007
12. Tripakis S, Stergiou C, Shaver C, Lee E A. A modular formal semantics for Ptolemy. Mathematical Structures in Computer Science, 2013, 23: 834–881. http://chess.eecs.berkeley.edu/pubs/877.html
13. Lee E A, Tripakis S. Modal models in Ptolemy. In: Proceedings of 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT 2010). October 2010, 1–11. http://chess.eecs.berkeley.edu/pubs/700.html
14. Hamon G, Rushby J. An operational semantics for Stateflow. In: Fundamental Approaches to Software Engineering: 7th International Conference (FASE), LNCS 2984. 2004, 229–243
15. Maraninchi F, Rémond Y. Mode-automata: a new domain-specific construct for the development of safe critical systems. Science of Computer Programming, 2003, 46(3): 219–254. http://dx.doi.org/10.1016/S0167-6423(02)00093-X
16. Colaço J L, Pagano B, Pouzet M. A conservative extension of synchronous data-flow with state machines. In: Proceedings of the 5th ACM international conference on Embedded software, EMSOFT '05. 2005, 173–182. http://doi.acm.org/10.1145/1086228.1086261
17. Harel D. Statecharts: A visual formalism for complex systems. Science of Computer Programming, 1987, 8(3): 231–274. http://dx.doi.org/10.1016/0167-6423(87)90035-9
18. Wang Y, Talpin J P, Benveniste A, Le Guernic P. 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, ISORC '00. 2000, 96–103. http://dl.acm.org/citation.cfm?id=850984.855510
19. André C. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003
20. von Hanxleden R, Duderstadt B, Motika C, Smyth S, Mendler M, Aguado J, Mercer S, O'Brien O. SCCharts: Sequentially constructive statecharts for safety-critical applications. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). June 2014
21. von Hanxleden R, Mendler M, Aguado J, Duderstadt B, Fuhrmann I, Motika C, Mercer S, O'Brien O. Sequentially constructive concurrency—A conservative extension of the synchronous model of computation. In: Design, Automation Test in Europe Conference Exhibition (DATE), 2013. March 2013
22. Radojevic I, Salcic Z, Roop P. Design of distributed heterogeneous embedded systems in DDFCharts. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(2): 296–308. http://dx.doi.org/10.1109/TPDS.2010.69
23. Talpin J P, Brunette C, Gautier T, Gamatié A. Polychronous mode automata. In: Proceedings of the 6th ACM & IEEE International conference on Embedded software, EMSOFT '06. 2006, 83–92. http://doi.acm.org/10.1145/1176887.1176900
24. Raymond P, Roux Y, Jahier E. Lutin: a language for specifying and executing reactive scenarios. EURASIP Journal on Embedded Systems, 2008
25. Cadoret F, Borde E, Gardoll S, Pautet L. Design patterns for rule-based refinement of safety critical embedded systems models. 2014

19th International Conference on Engineering of Complex Computer Systems, 2012, 0: 67–76
26. Ölveczky P C, Boronat A, Meseguer J. Formal semantics and analysis of behavioral aadl models in real-time maude. In: Proceedings of the 12th IFIP WG 6.1 International Conference and 30th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems, FMOODS'10/FORTE'10. 2010, 47–62
27. Yang Z, Hu K, Bodeveix J P, Pi L, Ma D, Talpin J P. Two formal semantics of a subset of the AADL. In: 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011. 2011, 344–349
28. Besnard L, Gautier T, Le Guernic P, Talpin J P. Compilation of polychronous data flow equations. In: Synthesis of Embedded Software. Springer, 2010. http://hal.inria.fr/inria-00540493
29. Besnard L, Gautier T, Le Guernic P. SIGNAL V4-INRIA version: Reference Manual, 2010. http://www.irisa.fr/espresso/Polychrony/documentation.php
30. Abramsky S, Jung A. Domain theory. In: Abramsky S, Gabbay D, Maibaum T, eds, Handbook of Logic in Computer Science, volume 3, 1–168. Oxford University Press, 1994
31. Kahn G. The semantics of a simple language for parallel programming. Proceedings of the IFIP Congress 74, Stockholm, Sweden, 1974, 471–475
32. Plotkin G D. A powerdomain construction. SIAM Journal on Computing, 1976, 5: 452–487
33. Halbwachs N, Raymond P, Ratel C. Generating efficient code from data-flow programs. In: Third International Symposium on Programming Language Implementation and Logic Programming. August 1991
34. Marchand H, Le Borgne M. Synthesis of discrete-event controllers based on the Signal environment. In: In Discrete Event Dynamic System: Theory and Applications. 2000, 325–346
35. Sorel Y. SynDEx: System-Level CAD Software for Optimizing Distributed Real-Time Embedded Systems. ERCIM News, 2004, 59: 68–69
36. Wikipedia . Autonomous cruise control system — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 26-November-2015]
37. Lee E A, Messerschmitt D G. Synchronous data flow. Proceedings of the IEEE, 1987, 75(9): 1235–1245
38. Besnard L, Gautier T, Le Guernic P, Guy C, Talpin J P, Larson B R, Borde E. Formal semantics of behavior specifications in the architecture analysis and design language standard. In: Nakajima S, Talpin J P, Toyoshima M, Yu H, eds. Cyber-Physical System Design from an Architecture Analysis Viewpoint. Springer, 2017, 53–79
39. Besnard L, Bouakaz A, Gautier T, Le Guernic P, Ma Y, Talpin J P, Yu H. Timed behavioural modelling and affine scheduling of embedded software architectures in the AADL using Polychrony. Science of Computer Programming, 2015, 54–77
40. Bouakaz A. Real-time scheduling of dataflow graphs. PhD thesis, Université de Rennes 1, November 2013
41. Ngo V C, Talpin J P, Gautier T. Precise deadlock detection for polychronous data-flow specifications. In: ESLsyn - DAC 2014. May 2014
42. Besnard L, Borde E, Dissaux P, Gautier T, Le Guernic P, Talpin J P. Logically timed specifications in the AADL: a synchronous model of computation and communication (recommendations to the SAE committee on AADL). Technical Report RT-0446, April 2014

Thierry Gautier is researcher with Inria. He received his graduate degree from INSA Rennes (France) and a PhD degree in Computer Science from the University of Rennes 1. He is one of the designers of the Signal language, the polychronous model of computation, and the Polychrony toolset. His main research interests focus on the safe design of complex embedded systems, including formal modeling, formal validation, transformations of models to target architectures, and synthesis of schedulers.

Clément Guy is a digital, educational, and science popularizer and facilitator. He obtained a PhD in Software Engineering at the University of Rennes 1. He then worked as a research engineer in different teams at INSA of Rennes and Inria. His work focused mainly on developing new methods and tools for better, safer, and more efficient embedded software and hardware. He then completed a Master's degree in techno-educational engineering. He works as a digital facilitator at the Mission locale du bassin d'emploi de Rennes, where he helps to deploy new communication tools and methods which better fit the needs of the employees. This includes audit and survey of user needs, meeting facilitating, tool deployment, and training.

After obtaining his Master's diploma in High Performance Computing from the ENSEIRB-MATMECA French engineering school in 2015, Alexandre Honorat began work with the TEA team at Inria Rennes to maintain the ADFG scheduling synthesizer. He also participated in linking the ADFG tool with the AADL analyzer in Polychrony, which generates Signal programs. Currently working as an engineer, he plans to study scheduling of real-time systems as a PhD student.

Paul Le Guernic graduated from Institut National des Sciences Appliquées de Rennes in 1974. He performed his Thèse de troisième cycle in Computer Science in 1976. From 1978 to 1984 he held a research position at Inria and served as the Directeur de Recherche in this institute since 1985. He has been head of the "Programming Environment for Real Time Applications" group, which has defined and developed the Signal language. He is one of the architects of the Polychrony toolset. Before he retired, his main research interests included the development of theories, tools, and methods for the design of real-time embedded heterogeneous systems.

Jean-Pierre Talpin is senior scientist with Inria and scientific leader of the Inria project-team TEA. Graduated in Applied Mathematics, he received a Master's degree in Theoretical Computer Science from University Paris 6 and comleted his Ph.D. thesis at Ecole des Mines de Paris. He then worked three years as research associate at the European Computer-Industry Research Centre in Munich. He joined Inria in 1995 and led project-team ESPRESSO from 2000 to 2012. He is an associate editor with the ACM's Transactions on Embedded Computer Systems. He edited three books, guest-edited a dozen scientific journal special issues with ACM and IEEE, and has authored more than one hundred journal articles, book chapters, and conference papers. He received the 2004 ACM Award for the most influential POPL paper and the 2012 ACM/IEEE LICS Test of Time Award.

Loïc Besnard is currently a senior engineer at CNRS, France. He received his Ph.D. degree in Computer Science from University of Rennes 1, France (1992). His research interests include software reliability for the design of embedded systems: modeling, temporal analysis, formal verification, simulation, and synthesis of embedded systems. He participates in the development of the Polychony toolset based on the synchronous language Signal, the POP platform, and with the import of formalisms to AADL. He is also involved in the development of Heptane, a static worst-case execution time estimation tool.