Polychronous Controller Synthesis from MARTE CCSL Timing Specifications

Huafeng Yu, Jean-Pierre Talpin, Loïc Besnard, Thierry Gautier, Hervé Marchand, Paul Le Guernic

Polychronous controller synthesis from MARTE CCSL timing specifications

Huafeng Yu Jean-Pierre Talpin Loïc Besnard
Thierry Gautier Hervé Marchand Paul Le Guernic
INRIA Rennes/IRISA/CNRS
263, avenue du Général Leclerc, Rennes, France
Email: {firstname}.{lastname}@inria.fr, Loic.Besnard@irisa.fr

***Abstract*—**
The UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) defines a mathematically expressive model of time, the Clock Constraint Specification Language (CCSL), to specify timed annotations on UML diagrams and thus provides them with formally defined timed interpretations. Thanks to its expressive capability, the CCSL allows for the specification of static and dynamic properties, of deterministic and non-deterministic behaviors, or of systems with multiple clock domains. Code generation from such multi-clocked specifications (for the purpose of synthesizing a simulator, for instance) is known to be a difficult issue. We address it by using the approach of controller synthesis. In our framework, a timed CCSL specification is regarded as a property whose satisfaction should be enforced for any UML diagram carrying it as annotation. To do so, CCSL statements are first translated into dynamical polynomial systems. Such systems can be manipulated using the model-checker Sigali to synthesize an executable property (a controller) which enforces the satisfaction of the specified timing constraints on the UML diagram with which it is executed.

***Index Terms*—MARTE; Polychrony; CCSL; GALS; controller synthesis;**

I. Introduction

Over the past decade, system-level design has been widely advocated as a way to overcome rising technological complexity of embedded system design and the aggravating factors of always stronger time-to-market constraints. To this end, high-level modeling languages, tools, and frameworks have been proposed to design, simulate and validate embedded systems with the claims of allowing engineers to gain comprehension and productivity thanks to the raised design-abstraction level offered by these models and tools. For instance, the Unified Modeling Language (UML) [29] has been widely used as a general purpose modeling and specification language. Its extension for modeling real-time and embedded (RT/E) systems, the UML Profile MARTE (for Modeling and Analysis of Real-Time and Embedded systems) [28], was adopted as an Object Management Group (OMG) specification. MARTE consists of necessary extensions to UML with modeling concepts and semantic disambiguations which make it usable for the formal specification of RT/E systems.

As importantly as the role time plays in RT/E system design, MARTE provides a formal model of time to accurately annotate UML objects with timing information. Inspired by the Tagged Signal Model [22] and synchronous languages [5], the CCSL (Clock Constraint Specification Language) [28], [2] is a specification formalism to express events, clocks and their relations in a way that supports both synchronous composition, asynchronous composition, as well as non-determinism, and in a progressive and compositional manner.

The design, simulation and validation of safety-critical RT/E systems not only require a high-level of abstraction to grasp system complexity, but also need to be grounded on a rigorously defined mathematical framework. Abstraction and rigor is how synchronous languages [12], [13] were designed and this is what brought them broad acceptance in the field.

Polychrony [18] is one such framework. It differs in its yet more abstract model of computation, which is based on timing relations (rather than timing functions as SCADE defines) and still comprises a so-called clock calculus to refine such relational specifications into deterministic, executable ones. SIGNAL [7] is the kernel design language at the origin of the Polychrony toolset. SIGNAL is a data-flow language in which equations describe abstract (timing) and concrete (causal) relations between discrete input and output streams of values (signals).

The relational framework of SIGNAL provides the unique capability to describe systems with multiple clocks (polychronous systems) as relational specifications, including non-deterministic devices (e.g., a non-deterministic bus) and external processes (e.g., an unsafe car driver). Deterministic specifications are obtained through a refinement process, aided by the clock calculus, to generate code for simulation, analysis, validation and synthesis. The application domains of Polychrony include software architectures of safety-critical systems, as found in automotive and avionics.

Nonetheless, it is notorious that code generation from a timed system with multiple clocks (a polychronous system) is far from obvious. For instance, SCADE always uses a *reference* or master clock (the fastest); all clocks and all conditions are defined as a functional sampling of this master clock, from the highest specification down to the lowest generated code. This is why it is called synchronous.

SIGNAL, on the contrary, enables the specification of systems with partially related or independent clock domains. A formally defined refinement process yields to the generation of (sequential or concurrent) code by the addition of control vari-

ables to get a deterministic behavior satisfying the constraints and allowing the desired amount of concurrency. The main advantage of this approach is the construction of deterministic temporal behavior that can be efficiently implemented on statically scheduled mono-processors or quasi-synchronous architectures (such as loosely Time-Triggered Architectures).

The main disadvantage of both approaches is that they are ad-hoc solutions for generating uni-processor code, making the exploitation of concurrency and distributed code generation both difficult and limited [9], [3].

The motivation of our work is still to take advantage of the formal framework of Polychrony in the context of a high-level specification formalism, MARTE CCSL. Yet, our work considers a totally different approach: to generate executable specifications by considering discrete controller synthesis (DCS) [32], [23], [25]. Distributed clock constraint resolution is addressed by DCS, which does not necessarily require a master clock to address polychronous clocks. In our approach, multi-clock (CCSL) specifications are translated into polynomial dynamical systems (PDSs). A PDS represents the transition system of a specification as well as the constraints (invariants) it must satisfy. Finally, the generated controller is synthesized into the original system to complete code generation via Polychrony. In conclusion, the temporal semantics of CCSL is mapped onto a polychronous model of computation, on which effective synthesis is carried out to meet constraint requirements. This approach provides both a useful mapping in theory and a flow, which is practical in the generation of reactive controllers.

DCS is one of the automated techniques that can exploit PDSs. It consists in constraining or controlling the system behavior against a given control objective. Using DCS, the synthesized controller preserves all possible behaviors of the initially specified system, with respect to the control objectives, instead of choosing one, ad-hoc, solution, as synchronous language compilers do. As a PDS can express polychronous systems, the computed result does not need a master-clock. To implement our DCS approach, we actually use Sigali, the model-checker and controller synthesis tool of Polychrony.

Outline. Section II presents some background on modeling time in reactive systems, using Polychrony and MARTE's CCSL. Section III gives a brief introduction to discrete controller synthesis and its implementation in Sigali. Section IV introduces our approach of synthesizing controllers from CCSL constraints, by describing a translation of CCSL into Signal and exemplifying it with a case study. Some related works are addressed in Section V. And the conclusion is drawn in Section VI.

II. Reactive systems and time modeling

Reactive systems [16] are generally embedded systems, which continuously interact with their environment under strict timing constraints required by the environment. These systems are generally safety-critical systems. The synchronous approach [15], based on the synchronous timing modeling, is proposed for the design of reactive systems. The evolution of system is considered as a discrete sequence of non-interleaving reactions. In each reaction, the input data reading, data computing, internal state updating, and output data writing are carried out in a short period of time, which is supposed to be compatible with the timing requirement of the interacting environment. All operations in a reaction run in parallel while all data dependency is respected. Reactions are abstracted by the notion of *instant* from a temporal view point. A logical clock, composed of a chain of instants, is then used to express when reactions occur. The time modeling in the synchronous approach is very close to synchronous circuits. This modeling makes it possible to provide a formal behavioral specification of systems, thus correctness proving of the behavior is possible and efficient. Synchronous programming languages [15], [5], compilers [7], formal verification and software synthesis tools [25] have been developed in the framework of synchronous approach. In this section, we give, based on synchronous modeling, a brief introduction to Polychrony and CCSL, and their corresponding time modeling for reactive systems.

A. The SIGNAL language and Polychrony

SIGNAL, based on the polychronous model of computation [21], is a synchronous language that allows the specification of *multi-clock/polychronous* systems, in which a process can be deactivated while other processes are still activated. In addition, the SIGNAL formal model allows partial and non-deterministic specifications. Thus, SIGNAL enables the globally asynchronous locally synchronous (GALS) design for distributed embedded systems.

In SIGNAL, variables are called *signals*[1]. Each signal (e.g., x) represents an infinite typed sequence, which is mapped onto the *logical time* indexed by natural numbers, i.e., x is actually $(x_\tau)_{\tau\in\mathbb{N}}$. The symbol $\perp$, which represents the absence of the signal at certain instant on the logical time, expands the domain of signal. Each signal is associated with a logical clock indicating the set of instants when the signal is present in the reactions. SIGNAL programs are mainly composed of equations over signals. These equations specify the relations, including clock and/or value relations, between signals. An elementary process is defined by an equation that associates an expression built on operators over signals with a signal. The arguments of operators can be expressions and signals. Several main elementary processes are listed in the following:

- **Stepwise extensions**. Let f be a symbol denoting a n-ary function $[\![f]\!]$ on values (e.g., Boolean, arithmetic or array operation). Then, the SIGNAL expression `y := f(x1,..., xn)` defines the process equal to the set of executions that satisfy: the signals $y, x_1, ..., x_n$ are synchronous and they are pure flows [7] that have same length l and satisfy $\forall t \leq l, y_t = [\![f]\!](x1_t, ..., xn_t)$. In addition, $\hat{}x \overset{def}{\Longleftrightarrow} (x = x)$ returns the clock of x, where $=$ denotes the stepwise extension of usual equality operator.

[1]Signal in capital, i.e., SIGNAL, indicates the language.

- **Delay**. This operator defines the signal whose t-th element is the (t-1)-th element of its (pure flow) input, at any instant but the first one, where it takes an initialization value. Then, the SIGNAL expression `y := x $ 1 init c` defines the process equal to the set of executions that satisfy: y, x are synchronous and they are pure flows, which have same length l and satisfy $\forall t \leq l, t > 1 \Rightarrow y_t = x_{t-1} \wedge t = 1 \Rightarrow y_t = c$.
- **Sampling**. This operator has a data input and a Boolean *control* input. When one of the inputs is absent, the output is also absent; at any logical instant where both input signals are defined, the output is present and equals to data input iff the control input holds the value true. Then, the SIGNAL expression `y := x when b` implies: y, x, b are extended to the same infinite domain T, respectively as y^T, x^T, b^T; and $\forall t \in T, (b_t^T = true \Rightarrow y_t^T = x_t^T) \wedge (b_t^T \neq true \Rightarrow y_t^T = \perp)$.
- **Deterministic merging**. The unique output provided by this operator is defined (i.e., with a value different from $\perp$) at any logical instant where at least one of its two inputs is defined (and non-defined otherwise); a priority defined on two inputs makes the output deterministic. The SIGNAL expression is `z := x default y`. The time domain T of z is the union of the time domains of x and y; z, x, y are extended to the same infinite domain $TT \supseteq T$, resp. as zz, xx, yy; synchronized flows satisfy $\forall t \in TT, (xx_t \neq \perp \Rightarrow zz_t = xx_t) \wedge (xx_t = \perp \Rightarrow zz_t = yy_t)$. There exists derived operators, such as `x1 ^+ x2` $\stackrel{def}{\Longleftrightarrow}$ `^x1 default ^x2` (clock union) returns an event signal that is present iff x_1 or x_2 is present; `x1 ^- x2` $\stackrel{def}{\Longleftrightarrow}$ `when ((not ^x2) default ^x1)` returns an event signal that is present iff x_1 is present and x_2 is absent.

Polychrony [18] is an integrated development environment based on SIGNAL. It is composed of the SIGNAL batch compiler, a Graphical User Interface (GUI), and the Sigali tool, etc. The compiler [7] provides a set of functionalities, which include program transformations, optimizations, code generation, etc. The Sigali tool is used to build associated formal systems for formal verification and controller synthesis [25], which will be detailed in Section III.

B. Time model in MARTE

MARTE, as a profile of UML, presents a time model in a more precise and clear manner than UML for the design of RTE systems. Both discrete and dense times are considered in MARTE. Clocks, which can be chronometric or logical, are used to access time structure. A chronometric clock implicitly refers to physical time, whereas a logical clock mainly addresses concrete instant ordering. The MARTE time model allows multiform/polychronous time modeling, which is inspired by synchronous languages. We only address logical clocks in this paper, which may be polychronous clocks.

A *clock* is a finite or infinite set of *instants*. A clock may represent a timed event and instants are its occurrences. A clock has a unit and the instants can have a label. These instants in a clock are totally ordered for discrete time clocks, thus they can be indexed by natural numbers. A time structure is composed of a set of clocks with the *precedence* relation between them. *Precedence* is a binary relation on clocks [1], and from this relation, we can derive the following new relations: *strict precedence*, *coincidence*, *independence*, and *exclusion*. We can directly find the last three relations in SIGNAL, whereas *strict precedence* implies non-deterministic ordering of instants, thus cannot be directly expressed by primitive SIGNAL operators. In order to avoid the confusion of concepts from different languages, we use *static* relation to indicate coincidence-based, sampling-based, and (^=, or ^−)-based relations in CCSL and SIGNAL. All other relations are called *dynamical* relations, including delay, independence, exclusion, and precedence-based relations.

MARTE introduces a new stereotype of UML Constraint, through which a MARTE timed system can be specified. CCSL is used to express the clock constraints based on these constraint stereotypes. It is a non normative language annexed to MARTE (Annex C), and it is independent of any existing language. A comprehensive informal description of CCSL has previously been presented in [2] and a formal semantics for a kernel of CCSL can be found in [1]. A CCSL specification consists of clock relations and clock expressions. *Subclock*, *alternatesWith*, etc. are examples of clock relations, while *delayedFor* and *s_sample* are clock expressions. In this paper, only main and frequently used constraints are presented, illustrated, and addressed (details can be found in Section IV).

III. Discrete controller synthesis

DCS [32], [23], [25] was originally defined in the framework of language theory, often called supervisory control of discrete event systems, and is related to game theory. It has been formulated in terms of labeled transition systems, and involves algorithms that explore symbolically and automatically the state space similar to model-checking, with complexity issues and capacities of the same order.

A. An informal introduction to DCS

DCS consists of computing constraints on controllable events, with regard to current system state and all the possibility of uncontrollable events, so that control objectives (specified by some properties) are always satisfied. To achieve DCS, we need the following preparations at least: make the system controllable and specify control objectives. The first preparation is involved in the partition of system behavior according to its controllability. For instance, the involved events in the system are partitioned into uncontrollable (I_u) and controllable ones (I_c), as illustrated in Fig. 1. The uncontrollable events are generally inputs that come from the system's environment, while controllable events correspond to locally produced events in the system. The second preparation concerns the specification of control objectives: properties to be always ensured, for instance, *invariance* and *reachability* of some given states of the system, i.e., behaviors are kept

within the safe states, or certain (e.g., termination) states of the application are always reachable. DCS synthesizes the resulting constraints, called *controllers* into the original system. The controllers are maximally permissive when the constraints on controllable events are minimal, i.e., system behaviors are constrained as least as possible while ensuring control objectives.

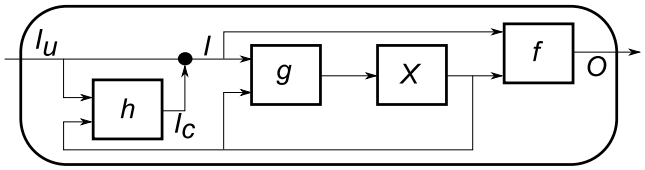

Fig. 1. An equational view of a reactive system controlled by h (obtained by DCS).

In Fig. 1, the function h represents the controller to synthesize. It is automatically computed from the reactive system (X, g, f), X being the state of the system, g the transition function, and f the output function. In a given state X and given any uncontrollable input I_u, the controller h computes values of controllable I_c so that the resulting behavior satisfies the control objectives.

B. Sigali and a symbolic representation

Sigali [11] is a formal verification and controller synthesis tool associated with Polychrony. It enables to prove the correctness of the dynamical behavior of a system. Sigali is based on polynomial dynamical equation systems (PDSs) over $\mathbb{Z}/3\mathbb{Z}$ [20], i.e., integers modulo 3: {0,1,-1} = {0,1,2}, as a formal model to describe the program behavior. Sigali manipulates the system of equations instead of the sets of solutions, which avoids the problem of enumerating state spaces, i.e., a set of states and/or events is represented by a unique polynomial. Thus, operations are performed on sets, while still remaining in the domain of polynomial functions, and avoiding to enumerate them. Sigali relies on an implementation of polynomials by Ternary Decision Diagram (TDD) (for three valued logic) in the same spirit of Binary Decision Diagram (BDD) [8]. However, the paths in the data structures of TDD are labeled by values in $\{1, 0, -1\}$.

C. Translating SIGNAL into PDS

SIGNAL processes are translated into a system of polynomial equations over $\mathbb{Z}/3\mathbb{Z}$ [20]. The three possible states of a Boolean signal X are coded as: *present* and *true* $\rightarrow$ 1, *present* and *false* $\rightarrow$ -1, and *absent* $\rightarrow$ 0). Non-Boolean signals are only considered by two states: *present* $\rightarrow$ 1 and *absent* $\rightarrow$ 0. The square of *present* is 1, i.e., it is present whatever its value. Hence, the clock of a signal X can be coded by x^2 (x is the corresponding variable of X in PDS). Similarly, two synchronous signals X and Y satisfy the constraint equation: $x^2 = y^2$.

Each primitive process of SIGNAL can be expressed in a polynomial equation. For instance, `C := A when B` can be translated as $c = a(-b - b^2)$: the solutions of this equation represent the set of behaviors of the primitive process `when`. The following table shows the translation of all the primitive SIGNAL operators into polynomial equations. For the non Boolean expressions, we only translate the synchronization between the signals. For instance, `x := y when B` is translated into $x^2 = y^2(-b - b^2)$ and `B := U > V` into $b^2 = u^2 = v^2$.

Event constraints	
`A ^= B`	$a^2 = b^2$
`C := A ^+ B`	$c = a^2 + b^2 - a^2b^2$
`C := A ^* B`	$c = a^2b^2$
Boolean instructions	
`B := not A`	$b = -a$
`C := A and B`	$c = ab(ab - a - b - 1)$ $a^2 = b^2$
`C := A or B`	$c = ab(1 - a - b - ab)$ $a^2 = b^2$
`C := A default B`	$c = a + (1 - a^2)b$
`C := A when B`	$c = a(-b - b^2)$
`B := A $1 (init` b_0 `)`	$x' = a + (1 - a^2)x$ $b = a^2x$ $x_0 = b_0$

TABLE I
TRANSLATION OF SIGNAL PRIMITIVE OPERATORS.

Any complete SIGNAL specification can be translated into a set of equations of PDS through the composition of equations that represent SIGNAL primitive processes. A PDS can be considered to have three sub-systems of polynomial equations in the form:

$$S = \begin{cases} X' = P(X, Y, U) \\ Q(X, Y, U) = 0 \\ Q_0(X) = 0 \end{cases}$$

where X and X' are vectors of *state* variables in $\mathbb{Z}/3\mathbb{Z}$. Y is a vector of *uncontrollable* variables, whereas U is a vector of *controllable* variables. The first equation, composed of all the equations over state variables, is a *state transition equation*. It captures the dynamical aspects of the system. The second equation, called *constraint equation*, specifies which event may occur in a given state. The last equation defines the initial states.

D. Control objective and synthesis

Given a PDS S, a controller is defined by a system of two equations: $C(X, Y, U) = 0$ and $C_0(X) = 0$, where the second equation determines the initial states that satisfy the control objectives and the first equation decides the instantaneous controls, i.e., when the controlled system is in state $x \in X$, and when an event $y \in Y$ occurs, any value $u \in U$ such that $Q(x, y, u) = 0$ and $C(x, y, u) = 0$ can be chosen. The behavior of the system S composed with the controller is modeled by the system S_c:

$$S_c = \begin{cases} X' = P(X,Y,U) \\ Q(X,Y,U) = 0 \qquad C(X,Y,U) = 0 \\ \quad Q_0(X_0) = 0 \qquad C_0(X_0) = 0 \end{cases}$$

The frequently-used (but not limited to) control objectives for which we are able to synthesize a controller include: the invariance of a set of states, i.e., a set of states E is invariant if every trajectory initialized in E remains in E; the (global) reachability of a set of states, i.e., a set E is (globally) reachable, if starting from any possible state, there exists a trajectory that reaches E.

More information about Sigali, including examples and synthesis algorithm, can be found in [26], [14] and [19].

IV. Synthesis considering CCSL constraints

Using Polychrony for the validation and synthesis of embedded systems, specified by MARTE CCSL, is not a simple and direct job due to the following reasons. First, code generation in synchronous languages is always based on the *reference* clock, which is the fastest clock existing in the system or synthesized by the compiler into the system. This solution to code generation is efficient for mono-task systems. However, it is not an adequate solution for multi-task systems or distributed systems, i.e., the systems that may have independent clocks. The second reason is related to the deterministic temporal behavior required by the code generation. Deterministic behavior is very useful for the design and verification of safety-critical systems. However, obtaining deterministic behavior may be a difficult task for large and complex systems. SIGNAL code generation may synthesize arbitrary reference clocks and control scheme in order to obtain deterministic behavior. Thus it may demand large amount of synchronization between clocks, which makes the system over-loaded.

Our proposition to this issue, compared to the code generation for synchronous languages, is based on DCS. The advantages of using DCS include:

- symbolic representation of system for an efficient system exploration, including the dynamical behavior, and the system exploration, similar to model-checking, enables one-hundred percent coverage;
- seamless connection to synchronous languages that simplifies the translation;
- ensuring constraints and properties, such as safety, reachability, for the purpose of reliable execution of system;
- automatic synthesis of controllers that preserves all possible solutions while ensuring control objectives.

The main synthesis process of this work is illustrated in Fig. 2. CCSL clock constraints are first partitioned according to the nature of constraints: *static* or *dynamical*. Static constraints can be well handled by compilers of synchronous languages, while dynamical constraints may lead to non-deterministic instant ordering in the system, and some of them could not be solved by synchronous compilers. Hence, we need a systematic method, DCS in our approach, to analyze and process these constraints as a complementary technique to synchronous compiling. We consider these constraints as control objective in our approach. The generated deterministic controller ensures the satisfaction of these constraints.

The next step is to translate these constraints into SIGNAL programs. All static constraints and part of dynamical constraints are translated into SIGNAL programs in a direct way as we can find similar operators in SIGNAL. However, some of dynamical constraints are translated into specific SIGNAL processes that meet the constraints. The translations of these constraints are detailed in Section IV-A. Then all the translated SIGNAL programs are compiled into dynamical polynomial systems (Section IV-B). The Sigali tool is used here to carry out the computing over PDS, and generate new dynamical polynomial systems in which all the constraints are satisfied.

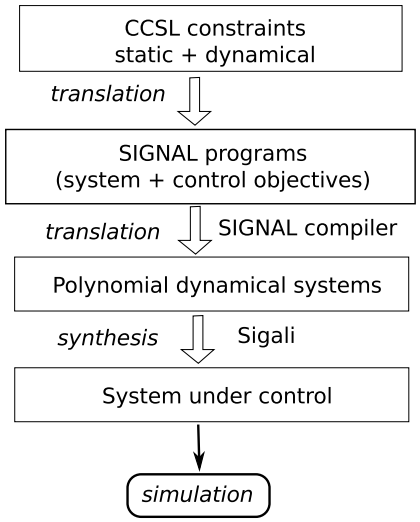

Fig. 2. The general schema of the polychronous controller synthesis.

A. Translation of CCSL clock constraints into SIGNAL

This work partly relies on the translation of typical CCSL relations into SIGNAL [24]. We developed a SIGNAL library for most of the CCSL constraints. Only main constraints are presented in this paper, which include *subclock*, *equality*, *exclusion*, *union*, *intersection*, *difference*, *alternatesWith*, *s_sample*, and *delayedFor*. These constraints are frequently used and typical enough to illustrate our approach. In the following, we give the definition of these constraints, then simple examples, followed by their translations into SIGNAL.

Before the presentation of the CCSL translation, several notations are first introduced for the definition of clock constraints. These notations are mainly based on the *instant* and *clock* notion:

- $\sim$ and $\nsim$ indicate that two instants are synchronous (coincidence in CCSL, and similar to $\hat{}=$ in SIGNAL for just one instant) and not synchronous (exclusive) respectively, for instance, $a \sim b$ implies instants a and b are synchronous.
- $<$ and $\leq$ are used to specify the precedence relation between two instants. For example, $a < b$ means instant a precedes b, while $c \leq d$ signifies $(c < d) \vee (c \sim d)$.

- $c[k],\ k \in \mathbb{N}$ indicates the k-th instant of clock c.
- $f_n(i, c, k)$ is a function that returns a k-th instant counted from the instant of clock c that appears after the instant i. Note that i can be an instant of clock c, or it can be an instant of another clock.
- $f_u(c_1, c_2)$ is a function that returns a clock that ticks whenever c_1 or c_2 ticks.

In the following, CCSL clock constraints and their translations into SIGNAL are presented. Please refer to [6] for the SIGNAL operators that are not presented in this paper.

1) Clock relation subclock, equality and exclusion: the definitions of these relations are:

Subclock: for two clocks c_1 and c_2,

$$c_1 \ isSubClockOf \ c_2 \text{ iff } \forall i \in c_1\ \exists j \in c_2,\ i \sim j.$$

Equality, for two clocks c_1 and c_2,

$$c_1 = c_2 \text{ iff } (c_1 \ isSubClockOf \ c_2) \bigwedge (c_2 \ isSubClockOf \ c_1)$$

Exclusion: for two clocks c_1 and c_2,

$$c_1 \ \# \ c_2 \text{ iff } \forall i \in c_1 \forall j \in c_2,\ i \nsim j.$$

Fig. 3 shows a simple example of *subclock* and *equality*.

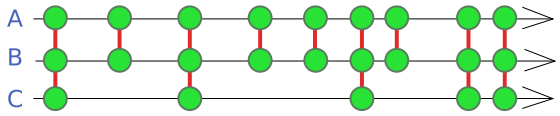

Fig. 3. An example of clock relation *subclock* and *equality*: $A = B\ ;\ C\ isSubClockOf\ A$.

The translation of these clock relations into SIGNAL is illustrated here:

- *isSubClockOf*: `C1 ^< C2;`
- *equality (=)*: `C1 ^= C2;`
- *exclusion (#)*: `C1 ^# C2;`

2) Clock expression union, intersection, difference: the definitions of these expressions are:

Union: for two clocks c_1 and c_2, c_1 *clockUnion* c_2 leads to the third clock c_3, where

$$(\forall i \in c_3\ \exists j \in f_u(c_1, c_2), (i \sim j)) \bigwedge$$
$$(\forall k \in f_u(c_1, c_2)\ \exists n \in c_3,\ (k \sim n)).$$

Intersection: for two clocks c_1 and c_2, c_1 *clockInter* c_2 results in another clock c_3, where

$$(\forall i \in c_3\ \exists j \in c_1\ \exists k \in c_2,\ (i \sim j) \wedge (i \sim k)) \bigwedge$$
$$(\forall l \in c_1\ \forall m \in c_2\ \exists n \in c_3,\ (l \sim m) \Rightarrow (n \sim l) \wedge (n \sim m)).$$

Difference: for two clocks c_1 and c_2, c_1 *clockDiff* c_2 leads to another clock c_3, where

$$(\forall i \in c_3\ \exists j \in c_1,\ i \sim j) \bigwedge (\forall k \in c_2\ \forall l \in c_3,\ k \nsim l) \bigwedge$$
$$(\forall m \in c_1\ \forall n \in c_2\ \exists o \in c_3,\ (m \nsim n) \Rightarrow (m \sim o).$$

Fig. 4 shows a simple example of clock expression *union*, *intersection*, *difference*:

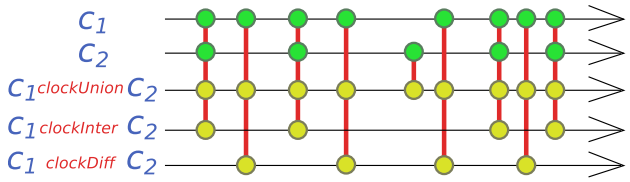

Fig. 4. An example of *union*, *intersection*, *difference*.

The following is the translation of these three clock expressions into SIGNAL processes:

- *clockUnion*: `C1 ^+ C2;`
- *clockInter*: `C1 ^* C2;`
- *clockDiff*: `C1 ^- C2;`

3) Clock relation alternatesWith: it indicates the alternate occurrences of instants from two clocks. More precisely, $C1\ alternatesWith\ C2$ signifies the occurrence of first instant c_1 from $C1$ precedes all occurrences of any instant from $C2$, and then $C2\ alternatesWith\ C1'$ where $C1'$ is tail of $C1$ from the instant c_2. In this case, the two clocks $C1$ and $C2$ are asynchronous and constrained by the alternatesWith relationship. Fig. 5 shows an example of *alternatesWith* between the two clocks $C1$ and $C2$. There are two forms of *alternatesWith*: weak form and strict form. Only strict form is discussed here.

Definition of **alternatesWith**: for two clocks c_1 and c_2,

$$c_1 \ alternatesWith \ c_2 \text{ iff}$$
$$\forall k \in \mathbb{N},\ (c_1[k] < c_2[k]) \bigwedge (c_2[k] < c_1[k+1]).$$

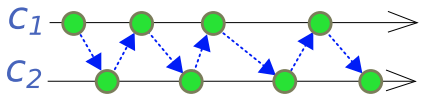

Fig. 5. An example of *alternatesWith*: $C_1\ alternatesWith\ C_2$.

The translation of *alternatesWith* into SIGNAL is shown here:

```
process alternatesWith =
    ( ? event C1, C2 ;)
    (| altern := not (altern$ init false)
     | C1 ^= when altern
     | C2 ^= when not altern
     |) where
          boolean altern;
        end;
```

4) Clock expression s_sample: this expression is based on sampling. There are two forms of sampling: weak and strict form, we only consider the strict form (s_sample) here. $C3 \overset{def}{\Longleftrightarrow} C2\ s\_sample\ C1$ indicates $C3$ is a subclock of $C1$ and each instant of $C3$ corresponds to an instant of $C1$ that comes directly after an instant of $C2$. i.e., between these two instants of $C1$ and $C2$, there is no other instant of $C1$. Fig. 6 presents an example of the *s_sample* relation: $C3$ is a subclock of $C1$, and $C1$ and $C2$ are polychronous.

Definition of **s_sample**: for two clocks c_1 and c_2, c_1 *s_sample* c_2 defines the clock c_3, where

$$(\forall i \in c_3\ \exists j \in c_1,\ i \sim j) \bigwedge$$

$$(\forall k \in c_2\ \exists l \in c_3,\ f_n(k, c_1, 1) \sim l) \bigwedge$$

$$(\forall m \in c_3\ \exists n \in c_2,\ f_n(n, c_1, 1) \sim m)$$

.

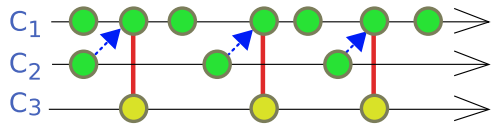

Fig. 6. An example of s_sample: $C3 \overset{def}{\Longleftrightarrow} C2\ s\_sample\ C1$.

This is the translation of *s_sample* into SIGNAL process:

```
process s_sample =
  ( ? event C2, C1; ! event C3; )
  (| c  := C2 default false
   | zc := c $ init false
   | c  ^= C1 ^+ C2
   | C3 := (C1 when C2 when zc)
           default (when not c when zc)
   |)
  where
    boolean c, zc;
  end;
```

*5) Clock expression **delayedFor**:* this expression is used to obtain delayed signals according to a faster clock. For instance, $C3 \overset{def}{\Longleftrightarrow} C2\ delayedFor\ n\ on\ C1$ implies, between each occurrence of an instant from $C2$ and its delayed occurrence in $C3$ (this instant of $C3$ is synchronous with an instant of $C1$), there are $n-1$ instants of $C1$. $C2$ is not synchronous with $C1$, and clock $C3$ is a subclock of $C1$ but not $C2$. Fig. 7 illustrates this relation with an example.

Definition of **delayedFor**: for two clocks c_1 and c_2, c_1 *delayedFor* n *on* c_2 results in another clock c_3, where

$$(\forall i \in c_3\ \exists j \in c_1,\ i \sim j) \bigwedge$$

$$(\forall k \in c_2\ \exists l \in c_3,\ f_n(k, c_1, n) \sim l) \bigwedge$$

$$(\forall m \in c_3\ \exists h \in c_2,\ f_n(h, c_1, n) \sim m) \bigwedge$$

$$(\forall q \in c_2\ \exists p \in c_1,\ (q < p)\ \wedge\ (p < f_n(q, c_2, 1)))$$

.

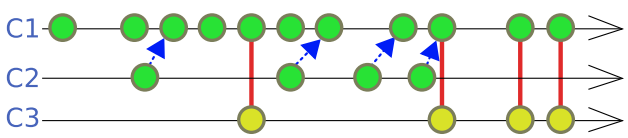

Fig. 7. An example of *delayedFor*: $C_3 \overset{def}{\Longleftrightarrow} C_2\ delayedFor\ 3\ on\ C_1$.

This is the translation of *delayedFor* into SIGNAL process:

```
process delayedFor =
  { integer n; }
  ( ? event C2, C1; ! event C3; )
  (| countC2 := C2 count n
   | array i to n-1 of
     (| countC1[i]:=C1 after (when countC2 = i)
      | fullDelay:=(countC1[i]=n) or fullDelay[?]
      |)
     with fullDelay := false
     end
   | C3 := when fullDelay
   |) where
        boolean fullDelay; integer countC2;
        [n] integer countC1;
      end;
```

B. Polychronous controller synthesis

All the CCSL constraints, including static and dynamical relations, can be considered as control objective to be ensured by the controller to synthesize. In addition, we also need to specify the controllability of the system, i.e., distinguish controllable clocks and uncontrollable clocks. In our approach, we mainly consider the following cases in the control of controllable clocks: *synchronization* of clocks and *value* of Boolean clocks. How to choose the right clock controllability can be found in [25]. A concrete example is used for illustration at the end of this section.

In the compilation process of SIGNAL programs, we can use the `z3z` option in order to automatically obtain PDS from SIGNAL programs. Compared to SIGNAL code generation, we generate z3z files (PDS) instead of executable files, such as C or Java files. A big difference between z3z and executable files is the solution synthesis. The generated executable files are integrated with an arbitrary solution to the equation system if it is polychronous, i.e., a deterministic solution is chosen if multiple (non-deterministic) solutions exist. However, in our approach, we avoid to exclude possible and valid solutions under condition that these solutions (non-deterministic) are not harmful to system safety (specified by control objective). Actually, no pre-defined solution is chosen and synthesized in the z3z file. Sigali will be used to analyze the system in z3z and synthesize controllers according to expected control objectives.

C. Code generation for simulation within Polychrony

Polychrony is used for the code generation and simulation. Fig. 8 illustrates the process carried out in Polychrony in order to perform the simulation:

- Once we obtain the SIGNAL programs P_s that are translated from CCSL, we need to specify the control objective CO_s.
- SIGNAL compiler is then used to obtain z3z files: `P.z3z` and `P_CMD.z3z`. `P.z3z` contains the polynomial equations, and synthesis commands used in Sigali are generated in `P_CMD.z3z`.
- Sigali is used for synthesis with imported `P.z3z` and `P_CMD.z3z`, and the controller is generated and saved in two files `Vt.sim` and `Vt.res`.

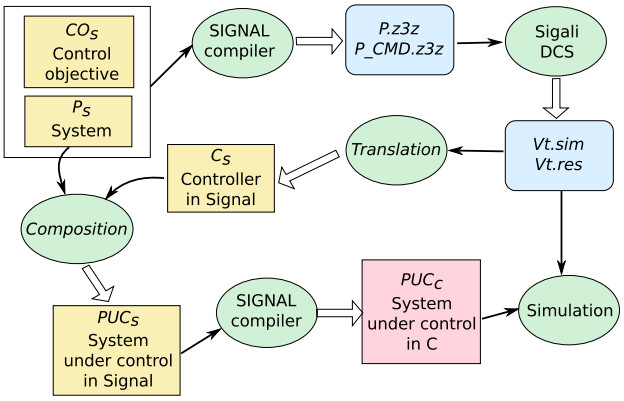

Fig. 8. Simulation for the controller synthesis in the framework of Polychrony.

- `Vt.sim` and `Vt.res` are translated to obtain controller C_s expressed in SIGNAL. These two files will also be used in the final step for simulation.
- P_s and C_s are composed together to get PUC_s. PUC_s represents the SIGNAL program with controller integrated.
- PUC_s is compiled by the SIGNAL compiler so that PUC_c is obtained. PUC_c is a C code file that corresponds to the system integrated with the controller.
- PUC_c is then compiled to obtain executable code, which is used for simulation, together with `Vt.res`.

In the following, an example is taken to illustrate our approach to address asynchronous clock relations with polychronous controller synthesis. Although this example is not a big one but it is adequate to illustrate the approach. The controller synthesis we used here adopts the same technique to calculate fixed-point as model checking, both of them have the same problem of state space explosion. Modular synthesis [10] has been proposed to alleviate this problem.

D. Simulation with an example

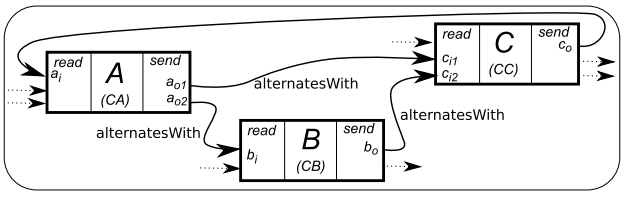

Fig. 9. An example of three synchronous components and a partial specification of the asynchronous communication between them.

1) An informal description of the example: This example is mainly involved in the asynchronous communication between three components. The three components, A, B, and C, have their own activation clocks: CA, CB and CC. These components, together with a partial specification of their communication, are illustrated in Fig. 9. Each of these components is considered as a synchronous component with *reading* and *sending* data operations. However, each operation can have its own running clock, which is a subclock of its component's activation clock. Clock constraints, including *isSubClockOf*, *equality* and *alternatesWith*, are specified between the data I/O operations of these components so that the coherent data reading and sending is ensured. These constraints are only partial specifications as the components can have other data I/O operations running on different clocks, described with dotted lines.

In this example, the components are expected to execute in a coherent way according to our partial specification: first, a_i of A reads data from c_o of C (for the first time, a_i reads the default value); after computing, A sends data to B and C via b_i and c_{i1} respectively; then B computes and sends data to C through b_o; when C receives all data from A and B through c_{i1} and c_{i2}, it computes and sends the results back to A. Without any specified constraints on data I/O operations of these components, data reading and sending may be stochastic. e.g., if CC is faster than CA, c_{i1} may read several times for just one sending by a_{o1}.

In the code generation of synchronous languages, a simple and possible solution is to synthesize a reference clock so that all the clocks in the system are directly or indirectly synchronized. However, the reference clock is a harsh requirement in a distributed environment. Furthermore, as this is only a partial specification, this reference clock may lead to conflicts with other specifications to add.

2) CCSL specifications and their translation into PDSs: Three clocks CA, CB, and CC are associated with the three components A, B, and C respectively. Each data I/O operation has its own clock, such as $a_i, a_{o1}, a_{o2}, b_i, b_o, c_i, c_{o1}, c_{o2}$, etc. These clocks are constrained by the following CCSL relations:

$$
\begin{gathered}
a_i \ isSubClockOf \ CA \\
a_i = a_{o1} \quad a_i = a_{o2} \\
b_i \ isSubClockOf \ CB \\
b_i = b_o \\
c_o \ isSubClockOf \ CC \\
c_{i1} = c_o \quad c_{i2} = c_o \\
a_{o2} \ alternatesWith \ bi \\
a_{o1} \ alternatesWith \ c_{i1} \\
b_o \ alternatesWith \ c_{i2}
\end{gathered}
$$

If we directly translate these expressions into SIGNAL programs according to Section IV-A and compile them, the SIGNAL compiler will prompt information related to *clock constraints*[2], and code will not be generated. The main reason of this problem comes from the fact that the compiler failed to synthesize a reference clock that satisfies all the given constraints. However, this does not signify there is no solution for the system. We use DCS to address this issue.

In our synthesis process, the three *alternatesWith* constraints are considered as control objective, which will be modified according to the requirement of Sigali. The result of this modification and all other constraints are then translated into

[2]Clock constraints in SIGNAL are generally related to non-deterministic aspects in the clock relations, sometimes the SIGNAL compiler cannot find a solution to solve these constraints.

SIGNAL, followed by the compilation into z3z. The clocks of components, such as CA, CB and CC, are specified as uncontrollable, and the clocks of reading and sending, such as a_i, a_{o1}, a_{o2}, b_i, b_o, c_{i1}, c_{i2} and c_o, are specified as controllable.

3) Simulation demonstration: After the synthesis by Sigali, Polychrony is used to carry out the simulation. Fig. 10 illustrates an example of the graphical interface for simulation, where the evolution of the controllable and uncontrollable clocks are shown.

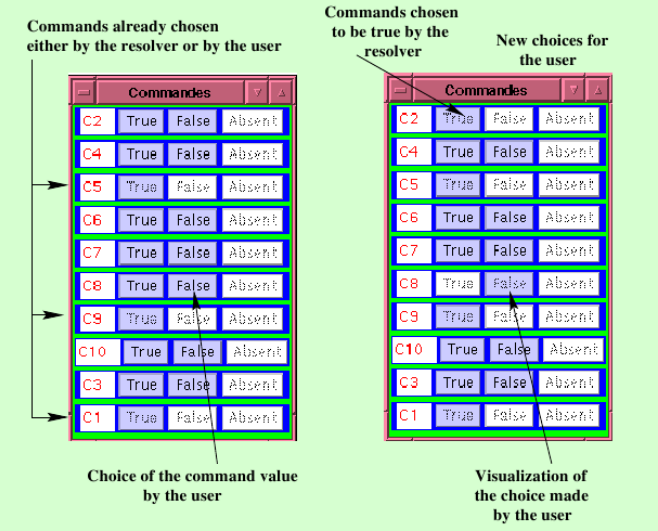

Fig. 10. The graphical interface for the simulation of DCS [25].

Fig. 11 illustrates a comparison of the uncontrolled system and controlled system from a view point of possible execution trace. The left part of this figure shows the original system without control, where CA, CB, and CC have their own clocks. Meanwhile, a_i, a_{o1}, a_{o2}, b_i, b_o, c_{i1}, c_{i2}, and c_o are not constrained. In the right part of the figure, CA, CB and CC still have their own clocks, however, a_i, a_{o1}, a_{o2}, b_i, b_o, c_{i1}, c_{i2}, and c_o can only occur considering the satisfaction of the constraints.

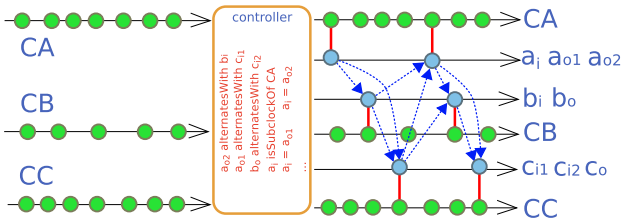

Fig. 11. Illustration of the controller from the view point of execution trace.

V. Related work

Our work is a major contribution to the code generation techniques for non-deterministic specifications as used in the SIGNAL compiler: to synthesize the control of a process, the compiler takes into account all static invariants; it may thus reject programs that could avoid states in which those invariants are violated. It is precisely the improvement brought by the Sigali DCS to allow a larger and more natural expression of control objectives to get executable C code. In general, a reference clock is needed in the code generation for synchronous languages. For instance, clock calculus [7] is used to analyze clock relations and to determine the *endochrony* property of a timed system specified in SIGNAL. Endochrony is an important property for the compilation of SIGNAL programs. An endochronous SIGNAL process can reconstruct synchronous clock relations from untimed yet ordered input signals with the help of signal relations and states defined in the process. [34] introduces an extension to address CCSL specifications. However, when the system is polychronous, code generation for SIGNAL requires to endochronize the system, for instance, integrate a reference clock for synchronization. In our work, we avoid to endochronize the system when it is polychronous.

As endochrony is not well situated to address issues of compositionality [4], asynchronous clock relations [3], etc., weakly endochronous systems [31] have been proposed for GALS design. They aim at meeting the requirements of building deterministic asynchronous implementations from polychronous specifications. Weak endochrony enables identical execution of synchronous specifications in any asynchronous environment. Weak endochrony enables to address asynchronous composition while preserving deterministic system behavior. In comparison, our work accepts non-deterministic behavior under condition that the expected properties are always ensured.

[25] presents the general purpose of controller synthesis for Polychrony. It is a little different from [33], [32] in the definition and partition of controllable and uncontrollable events. Based on [25], our approach is mainly dedicated to addressing asynchronous and/or independent clock relations. [10] presents the same idea of controller synthesis dedicated to code generation. The main difference is that it is based on the endochronous programs and behavior contracts are integrated.

TimeSquare [17] is a software environment dedicated to the resolution of CCSL constraints and computation of partial solutions. The simulation of CCSL constraints is based on local constraint satisfaction, i.e., the constraints are calculated at each simulation step [1]. In case of multiple solutions in a step, a pre-defined policy determines the solution to choose. Thus, TimeSquare provides an active and non-deterministic constraint solution. Whereas, our controller synthesis is rather a passive one and all possible solutions are preserved.

From another point of view, our work is similar to partial order based scheduling, such as [30]. However, in one hand, we need to extend poset with the notion of synchronization relation; on the other hand our synchronous reaction simplifies the execution time computing. Another similar work based on poset is *synchronous structure*, presented in [27]. Synchronous structure provides a theoretical point of view of event structure with synchronous relations and partial order notion. So it is interesting to analyze the asynchronous clock relations with the help of synchronous structure in our approach.

VI. Conclusion

We presented an original and effective approach to generating executable specifications from MARTE's CCSL timing constraints by using the controller synthesis framework of the Polychrony toolset. This approach which, to our knowledge, was never tried before, is neither based on code generation techniques for synchronous languages nor on the interpreted and non-deterministic constraint resolution techniques present in TimeSquare.

We show that clock constraints can be considered as the control objectives and that they can be enforced by the synthesized controller. The computation of the controller itself is carried out using polynomial dynamical systems (PDS). CCSL constraints are first translated into PDS using Signal and Sigali is then used to compute the controller.

Polychronous controller synthesis offers the additional advantage to enable the simulation of sporadic or asynchronous clocks. A perspective for future work is to use our technique to complement code generation techniques presented in the SIGNAL compiler with one based on controller synthesis, for the purpose of concurrent simulation, rather than sequential code generation.

Another perspective would be to extend the present framework with the capability of handling numerical constraints (as found with the number n used in the delayedFor clock expression). As a PDS is based on logical (symbolic) equations, one would need to translate these numbers into Boolean variables so that they could be processed by Sigali or extend it with SMT solving capabilities (satisfiability modulo theory).

References

[1] C. André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009. 37 pages.

[2] C. André, F. Mallet, and R. de Simone. Modeling Time(s). In *ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS/UML'07)*, LNCS 4735, pages 559–573, TN, USA, October 2007. Springer.

[3] A. Benveniste, B. Caillaud, and P. Le Guernic. From Synchrony to Asynchrony. In J. Baeten and S. Mauw, editors, *10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *LNCS*, pages 162–177. Springer-Verlag, 1999.

[4] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in Dataflow Synchronous Languages: Specification and Distributed Code Generation. 163:125–171, 2000.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.

[6] L. Besnard, T. Gautier, and P. Le Guernic. SIGNAL V4 Reference Manual. http://www.irisa.fr/espresso/Polychrony/document/V4_def.pdf, March 2010.

[7] L. Besnard, T. Gautier, P. Le Guernic, and J.-P. Talpin. Compilation of Polychronous Data Flow Equations. In S. Shukla and J.-P. Talpin, editors, *Correct-by-Construction Embedded Software Synthesis: Formal Frameworks, Methodologies, and Tools*. Springer, 2010.

[8] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulations. *IEEE Transaction on Computers*, C-45(8):677–691, 1992.

[9] P. Caspi, A. Girault, and D. Pilaud. Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors. *IEEE Trans. Software Engin.*, 25(3):416–427, May 1999.

[10] G. Delaval, H. Marchand, and E. Rutten. Contracts for Modular Discrete Controller Synthesis. In *Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'10)*, 2010.

[11] B. Dutertre. *Spécification et Preuve de Systèmes Dynamiques*. PhD thesis, Universite de Rennes I, IFSIC, 1992. In French.

[12] Esterel technologies. ESTEREL. http://www.esterel-technologies.com/.

[13] Esterel Technologies. SCADE suite. http://www.esterel-technologies.com/products/scade-suite/.

[14] M. L. B. H. Marchand. Partial order control and optimal control of discrete event systems modeled as polynomial dynamical systems over galois fields. Research Report IRISA, No 1125, October 1997.

[15] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[16] D. Harel and A. Pnueli. *On the Development of Reactive Systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[17] INRIA AOSTE team. TimeSquare. http://www-sop.inria.fr/aoste/dev/time_square, 2009.

[18] INRIA ESPRESSO team. Polychrony. http://www.irisa.fr/espresso/Polychrony, December 2010. Polychrony V4.16.

[19] INRIA Vertecs and Espresso team. Sigali. http://www.irisa.fr/vertecs/Softwares/sigali.html.

[20] M. Le Borgne, A. Benveniste, and P. Le Guernic. Polynomial Dynamical Systems Over Finite Fields. *Algebraic Computing in Control*, 165:212–222, 1991.

[21] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3):261–304, 2003.

[22] E. Lee and A. Sangiovanni-Vincentelli. A Framework for Comparing Models of Computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, December 1998.

[23] O. Maler, A. Pnueli, and J. Sifakis. On the Synthesis of Discrete Controllers for timed Systems. In *Proceedings STACS'95*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242, 1995.

[24] F. Mallet and C. André. On the Semantics of UML/MARTE Clock Constraints. In *12th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2009)*, pages 305–312, Tokyo, Japan, March 2009. IEEE Computer Society.

[25] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the SIGNAL Environment. *Discrete Event Dynamic System: Theory and Applications*, 10(4):325–346, October 2000.

[26] H. Marchand and M. Samaan. Incremental Design of a Power Transformer Station Controller using Controller Synthesis Methodology. *IEEE Transaction on Software Engineering*, 26(8):729–741, August 2000.

[27] D. Nowak, J.-P. Talpin, and P. Le Guernic. Synchronous Structures. In *International Conference on Concurrency Theory (CONCUR'99)*. Springer Verlag, August 1999.

[28] Object Management Group (OMG). Modeling and Analysis of Real-time and Embedded systems (MARTE), v1.0. http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf, November 2009. Document number: formal/2009-11-02.

[29] Object Management Group (OMG). UML 2.2 Superstructure and Infrastructure. http://www.omg.org/spec/UML/2.2, February 2009. formal/2009-02-04.

[30] N. Policella. Scheduling with Uncertainty: A Proactive Approach Using Partial Order Schedules: Thesis. *AI Commun.*, 18:165–167, April 2005.

[31] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in Synchronous Systems. In *Formal Methods in System Design*, volume 28, pages 111–130, March 2006.

[32] P. J. Ramadge and W. M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE, Special issue on Dynamics of Discrete Event Systems*, 77(1):81–98, 1989.

[33] W. M. Wonham and P. J. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM J. Control Optim.*, 25(3):637–659, 1987.

[34] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, F. Mallet, C. André, and R. de Simone. Polychronous Analysis of Timing Constraints in UML MARTE. In *IEEE International Workshop on Model-Based Engineering for Real-Time Embedded Systems Design (hosted by ISORC 2010)*, Parador of Carmona, Spain, May 2010.