A Multi-Periodic Synchronous Data-Flow Language

Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti

A Multi-Periodic Synchronous Data-Flow Language

Julien Forget*, Frédéric Boniol*, David Lesens† and Claire Pagetti*

*ONERA, Toulouse, France, Email: firstname.lastname@onera.fr

†EADS Astrium Space Transportation, Les Mureaux, France

Abstract

Implementing real-time critical systems is an increasingly complex process that calls for high-level formal programming languages. Existing languages mainly focus on mono-periodic systems, implementing multi-periodic systems with these languages is possible but inefficient. As a result, current practice usually consists in writing one program for each different rate and then letting a real-time operating system handle the multi-rate aspects. This can be a source of non-determinism as communications between processes of different rates are not precisely defined. We propose a new language, built upon synchronous data-flow languages, to handle multi-rate systems properly. It has strong formal semantics, which prevents non-deterministic communications, and relies on real-time primitives that enable efficient use of existing multi-periodic schedulers.

I. Introduction

Programming reactive systems is a complex task since this does not only consist in implementing the functional aspects of the system. These systems are often critical, an error can lead to dramatic results, therefore the main concern for the program supervising a reactive system is an increased need for predictability. This requires of course strong *functional determinism*, which means that the program will always produce the same output sequence with respect to the same input sequence. However the program must be *temporally deterministic* as well, always having the same temporal behaviour and respecting hard real-time constraints. The system also needs to be optimized, in terms of latency, hardware cost, power consumption or weight for instance. The complexity of the development process and the criticity of the systems call for high-level formal programming languages, which cover the complete process from design to implementation. Automatic code generation process provides high confidence in the final program as every step of the generation is proven formally.

A. Motivation

1) Multi-Rate Reactive Systems: In [1] we studied the programming of an adapted version of the Mission Safing Unit (MSU) of the Automated Transfer Vehicle designed by Astrium Space Transportation for resupplying the International Space Station. This unit supervises the main computing unit of the vehicle in order to detect possible failures. As any reactive systems, it repeats indefinitely the same process (described in Fig. 1): acquire inputs on sensors, perform computations, produce outputs on actuators. It is a typical example of multi-rate reactive system, consisting in a set of communicating processes (`BASIC_OP`, `APPLY_CMD`, `UPSTREAM` and `DOWNSTREAM`) executing at two different rates (10Hz or 2Hz). Programming this system first requires to describe the functional behaviour of each process separately and then to assemble the different processes, specifying the rate of each process and how they communicate. Obviously, communications between processes executing at the same rate will not be performed in the same way as communications between processes of different rates (called multi-rate communications). Furthermore, a multi-rate communication will usually require to be delayed (the Z boxes) when it goes from a slow process to a fast process so as not to delay the fast process, while a multi-rate communication that goes from a fast process to a slow process can be performed immediately. Our study showed that existing languages lack simple primitives to precisely specify this assembly level. The current paper addresses this issue.

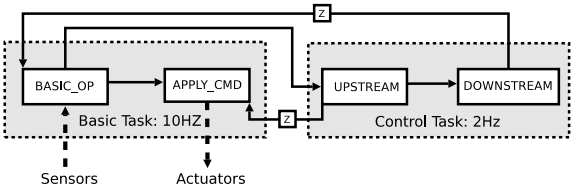

Fig. 1. The Mission Safing Unit

2) Formal Languages for Automated Code Generation: Reactive systems have traditionally been implemented using low level programming languages (ADA or C) at a level very close to the underlying operating system. However, verifying the correction of such low level programs is difficult, which led to the use of higher level languages. Matlab/Simulink [2] or AADL [3] can be used during the specification and prototyping phases but due to the lack of formal semantics and/or of an efficient formally defined compiler, the final program is implemented separately using low level languages. The Synchronous approach [4] is well adapted to implementing reactive systems and provides formally proven automatic code generation. Unfortunately, our study [1] showed that synchronous languages lack primitives to handle multi-rate systems. This paper proposes a new language, built upon synchronous languages, that addresses this issue.

3) Real-Time Constraints in the Synchronous Approach: Synchronous data-flow languages (LUSTRE/ SCADE [5], SIGNAL [6]) have successfully been applied to the implementation of reactive systems in the past. They are well adapted to describing precisely the data flow between communicating processes. A synchronous program captures the functional behaviour of a reactive system by describing the computations performed during one basic iteration of the system, called *instant* (this behaviour is then repeated indefinitely). Computations can be activated or deactivated at different instants using *clocks* (Boolean conditions). Clocks define the temporal behaviour of the program on a logical time scale: the sequence of instants. The synchronous approach enforces precise specification of communications between processes because only flows present at the same instants (i.e. flows that have the same clock) are allowed by the compiler. Flows that have different (but still related) clocks must be brought to the same clock, using sampling primitives, before they can be combined.

Unfortunately, classic synchronous data-flow languages are best adapted to mono-rate systems. Indeed, the activation conditions of processes are all specified with respect to the same base rate, the rate at which basic iterations are repeated. Multi-rate can be emulated with Boolean clocks, for instance a clock that is true once every n instant can stand for a rate n times slower than the base rate. However, this is tedious in practice [1]. Indeed, if a process executes once every n instant, it still needs to end before the end of the instant where it starts. If the process requires longer than one instant to execute, its computations must be split by the programmer over several instants. This amounts to calculating and implementing a multi-periodic schedule manually.

B. Contribution

We propose a new language for programming multi-rate reactive systems. It is built upon synchronous data-flow languages to inherit their strong formal properties. It adds high level primitives to assemble synchronous processes executing at different rates. The rates of the different processes are specified using a new class of clocks called *strictly periodic clocks*. Transformations on strictly periodic clocks are introduced to handle multi-rate communications. Strictly periodic clocks are defined uniquely by their period and by their phase. Transformations on strictly periodic clocks either increase or decrease their period or change their phase to produce new strictly periodic clocks. Strictly periodic clocks and Boolean clocks are complementary so our language uses both. Strictly periodic clocks define the real-time rate of a flow while Boolean clocks define activation conditions. The programmer can first specify the base frequency of a process using strictly periodic clocks and then specify that on this frequency the process is activated only if a certain condition holds, using Boolean clocks.

For the most part, strictly periodic clocks and their associated primitives could be programmed using Boolean clocks. However, providing specific primitives for this class of clocks has two important benefits. First, on the contrary to Boolean clocks, strictly periodic clocks and their associated transformations are statically evaluable. Thus the compiler does not need to overapproximate the set of activation dates of the tasks activated by such clocks. Strictly periodic clocks directly provide the real-time characteristics required by the scheduler, that is the period, release date and deadline of each process of the program. Second, a program written using strictly periodic clocks clearly identifies the real-time behaviour of the system by separating flow rates from flow activation conditions.

C. Paper Outline

This article is organized as follows. Section II presents the synchronous model adapted to real-time and strictly periodic clocks. The syntax of our language and its formal synchronous temporised semantics are then defined in Section III. The semantics of a synchronous program are well-defined only if the program always combines flows present at the same instants, that is only flows that have the same clock. This is checked by the clock-calculus detailed in Section IV. The clock-calculus uses a type system to generate clock constraints from a program. It then tries to solve the set of constraints to check that the program is correct. If the program is correct, the compiler can generate code according to the semantic rules previously defined. The code generation, that translates the synchronous program to lower level sequential code (for instance C code), is presented in Section V.

II. Strictly Periodic Clocks

A. Synchronous Real-Time

Our synchronous real-time model relies on the Tagged-Signal Model [7]. Given a set of *values* $\mathcal{V}$ and a countable set of *tags* $\mathcal{T}$, an *event* is an element of $\mathcal{V} \times \mathcal{T}$ and a *signal*, also called *flow* for our class of systems, is a set of events. We only consider functional signals, i.e. a signal cannot contain two events with the same tag. Two flows are *synchronous* if they have the same set of tags. The *clock* of a signal is its projection on $\mathcal{T}$. A *process* (more precisely a behaviour satisfying a process) is a set of signals. We focus on a subclass of processes called *synchronous temporised systems*. In such a system, the set of tags is completely ordered, which gives a chronological order on events.

Synchronous temporised systems are related to real-time by a *datation function*. The datation function associates a real-time date to each tag of the synchronous temporised system. It is a strictly increasing and bijective function, denoted $date$, of type $\mathcal{T} \rightarrow \mathbb{Q}$, which returns dates corresponding to the time elapsed since the beginning of the execution of the program. Synchronous temporised systems respect a *relaxed synchronous hypothesis* (similar to [8]) where each flow is required to be computed before its next activation, instead of before the next instant. When a flow is required to be activated at tag t_i and its next activation is t_{i+1}, the implementation will verify the relaxed synchronous hypothesis only if the corresponding value is produced during the time interval $[date(t_i), date(t_{i+1})[$. The deadline $date(t_{i+1})$ may actually have to be refined due to data dependencies, this point will be detailed in Sect. V-A. The datation function relates flow activations to real-time dates and enables us to abstract from the type of underlying real-time model (it can for instance be discrete as well as continuous).

B. Definitions

A clock is a sequence of tags, we define a particular class of real-time periodic clocks called strictly periodic clocks:

Definition 1. *(*Strictly periodic clock*). A clock* $h = (t_i)_{i \in \mathbb{N}}$, $t_i \in \mathcal{T}$, *is* strictly periodic *if and only if:*

$$\exists n \in \mathbb{Q}^{+*},\ \forall i \in \mathbb{N},\ date(t_{i+1}) - date(t_i) = n$$

n *is called the* period *of* h, *denoted* $\pi(h)$ *and* $date(t_0)$ *is called the* phase *of* h, *denoted* $\varphi(h)$.

A strictly periodic clock defines the real-time rate of a flow. As $date$ is bijective, a strictly periodic clock can be uniquely characterized by its phase and by its period. The notion of phase defined above is a little more general than usual as we do not impose that $\varphi(h) < \pi(h)$.

Definition 2. *The term* $(n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$ *denotes the strictly periodic clock* α *such that:*

$$\pi(\alpha) = n,\ \varphi(\alpha) = \pi(\alpha) * p$$

Definition 3. *(*Periodic clock division*). Let* α *be a strictly periodic clock and* $k \in \mathbb{Q}^{+*}$. *"*$\alpha /_. k$*" is a strictly periodic clock such that:*

$$\pi(\alpha /_. k) = k * \pi(\alpha),\ \varphi(\alpha /_. k) = \varphi(\alpha)$$

Definition 4. *(*Periodic clock multiplication*). Let* α *be a strictly periodic clock and* $k \in \mathbb{Q}^{+*}$. *"*$\alpha *_. k$*" is a strictly periodic clock such that:*

$$\pi(\alpha *_. k) = \pi(\alpha)/k,\ \varphi(\alpha *_. k) = \varphi(\alpha)$$

Definition 5. *(*Phase offset*). Let* α *be a strictly periodic clock and* $k \in \mathbb{Q}$. *"*$\alpha \rightarrow_. k$*" is a strictly periodic clock such that:*

$$\pi(\alpha \rightarrow_. k) = \pi(\alpha),\ \varphi(\alpha \rightarrow_. k) = \varphi(\alpha) + k * \pi(\alpha)$$

Divisions and multiplications respectively decrease or increase the frequency of the clock while phase offsets shift the phase of the clock. This is illustrated in Fig. 2. Again, we do not constrain that $\varphi(\alpha \rightarrow_. k) < \pi(\alpha)$.

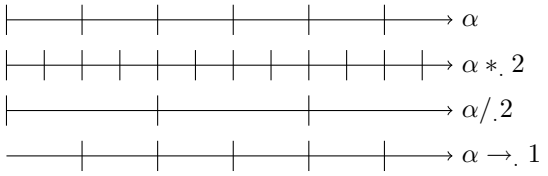

Fig. 2. Strictly periodic clocks.

Let $\mathcal{P}$ be the set of strictly periodic clocks. From previous definitions we directly obtain Properties 1, 2 and 3.

Property 1. $\forall \alpha, \beta \in \mathcal{P}$*:*

- $\forall k \in \mathbb{Q},\ \alpha = \beta \rightarrow_. k \Leftrightarrow \beta = \alpha \rightarrow_. -k$
- $\forall k \in \mathbb{Q}^{+*},\ \alpha = \beta /_. k \Leftrightarrow \beta = \alpha *_. k$

Property 2. $\forall (n, p) \in \mathbb{Q}^{+*} \times \mathbb{Q}$*:*

- $\forall k \in \mathbb{Q}^{+*},\ (n, p) *_. k = (\frac{n}{k}, kp)$
- $\forall k \in \mathbb{Q}^{+*},\ (n, p) /_. k = (nk, \frac{p}{k})$
- $\forall k \in \mathbb{Q},\ (n, p) \rightarrow_. k = (n, p + k)$

Property 3. *The set of strictly periodic clocks* $\mathcal{P}$ *is closed under operations* $*_.$, $/_.$ *and* $\rightarrow_.$.

C. Comparing Strictly Periodic Clocks

Let $\mathcal{E}^{\mathcal{P}}$ be the set of expressions on strictly periodic clocks, that is expressions obtained by applying a succession of periodic clock transformations on a strictly periodic clock. The following arithmetical properties, when read from left to right, form a rewriting system $\mathcal{R}$ to

simplify expressions in $\mathcal{E}^{\mathcal{P}}$. They can easily be derived from the definition of periodic clock transformations and from Property 2:

Property 4. *$\forall \alpha \in \mathcal{P}$:*

- $\forall k, k' \in \mathbb{Q}^{+*}, \alpha *_. k *_. k' = \alpha *_. kk'$
- $\forall k, k' \in \mathbb{Q}^{+*}, \alpha /_. k /_. k' = \alpha /_. kk'$
- $\forall k, k' \in \mathbb{Q}, \alpha \rightarrow_. k \rightarrow_. k' = \alpha \rightarrow_. (k + k')$
- $\forall k, k' \in \mathbb{Q}^{+*}, \alpha /_. k *_. k' = \alpha *_. k' /_. k$
- $\forall k \in \mathbb{Q}, \forall k' \in \mathbb{Q}^{+*}, \alpha \rightarrow_. k *_. k' = \alpha *_. k' \rightarrow_. kk'$
- $\forall k \in \mathbb{Q}, \forall k' \in \mathbb{Q}^{+*}, \alpha \rightarrow_. k /_. k' = \alpha /_. k' \rightarrow_. (k/k')$

Theorem 1. *The rewriting system $\mathcal{R}$ is convergent.*

The proof is fairly standard and not detailed here. As $\mathcal{R}$ is convergent, any expression e of $\mathcal{E}^{\mathcal{P}}$ has a unique normal form, denoted $NF(e)$ and:

$$\forall e_1, e_2 \in \mathcal{E}^{\mathcal{P}},\ e_1 = e_2 \Leftrightarrow NF(e_1) = NF(e_2)$$

Lemma 1. *$\forall e \in \mathcal{E}^{\mathcal{P}},\ \exists \alpha \in \mathcal{P}, k, k' \in \mathbb{Q}^{+*}, k'' \in \mathbb{Q}$:*

$$NF(e) = \alpha *_. k /_. k' \rightarrow_. k''$$

D. Integer Strictly Periodic Clocks

We defined the general model of strictly periodic clocks using dates in $\mathbb{Q}$. However, to define a compilable language, we need to restrain to dates in $\mathbb{N}^+$. Indeed operating systems rely on a discrete time model and there is always a lower bound on the level of granularity that can be used to describe time. For instance the date $1/3$ does not exist in a real system, or is approximated. For the same reasons, scheduling theory, needed to implement our systems, only applies to dates and durations in $\mathbb{N}$. We consequently constrain the type of the function $date$ to be $\mathcal{T} \rightarrow \mathbb{N}^+$ and the parameter k used in periodic clock divisions and periodic clock multiplications to be an element of $\mathbb{N}^*$. The parameter k used in phase offsets remains an element of $\mathbb{Q}$. In the rest of the document we will only consider such *integer strictly periodic clocks* and we will refer to them simply as strictly periodic clocks.

In the set of integer strictly periodic clocks, clock (n, p) is a valid strictly periodic clock only if: $n \in \mathbb{N}^{+*}, n * p \in \mathbb{N}^+$. Indeed, clocks that do not verify this property refer to dates that cannot be expressed with the restricted datation function (unless some approximations are performed). This restricted set is not closed under $*_.$ and $\rightarrow_.$ anymore (but is still closed under $/_.$). We use notation $k|k'$ to say that k divides k' (the rest of the integer division is 0). First, let (n, p) be a strictly periodic clock, if $k|n$ then $(n, p) *_. k$ is a valid strictly periodic clock, but otherwise it is not, as this clock refers to dates which value is not in $\mathbb{N}$. Second, if $(p + k)n \in \mathbb{N}^+$ then $(n, p) \rightarrow_. k$ is a valid strictly periodic clock, but otherwise it is not, as this clock refers to negative dates. The clock calculus, presented in Sect. IV, will check that no such invalid clock is used in a program.

III. A Synchronous Real-Time Language

We now define a language for programming multi-rate reactive systems using strictly periodic clocks. The language builds upon data-flow synchronous languages, adding high level real-time primitives based on strictly periodic clocks.

A. Syntax

The syntax is close to LUSTRE, however we do not impose to declare types and clocks, which are computed automatically, similarly to LUCID SYNCHRONE [9]. The main novelty is the introduction of expressions on strictly periodic clocks ($epck$). The language grammar is given below:

$$\begin{array}{lcl}
i & ::= & \texttt{true} \mid \texttt{false} \mid 0 \mid ... \\
var & ::= & x \mid var, var \\
e & ::= & i \mid x \mid (e, e) \mid e\ \texttt{fby}\ e \mid N(e) \\
 & & \mid e\ \texttt{when}\ e \mid \texttt{merge}\ e\ e\ e \mid epck \\
epck & ::= & e/\hat{}k \mid e *\hat{} k \mid e \sim\!> q \\
eq & ::= & var = e \mid eq; eq \\
io & ::= & x : (n, p) \mid x \mid io; io \\
nd & ::= & \texttt{node}\ N(io)\ \texttt{returns}\ (io) \\
 & & [\ \texttt{var}\ var;]\ \texttt{let}\ eq\ \texttt{tel} \\
 & & \mid \texttt{imported node}\ N(io)\ \texttt{returns}\ (io); \\
dr & ::= & \texttt{wcet}\ N = n;
\end{array}$$

A program consists in a list of node declarations (nd) and a list of node durations, more precisely an upper bound on worst case execution times (dr). Nodes can either be defined in the program (`node`) or implemented outside (`imported node`), for instance by a C function. Node durations must be provided for each imported node and predefined node of the language (for instance arithmetic operators). The clock of input/output parameters (io) of a node can be declared strictly periodic ($x : (n, p)$, n being the period and p being the phase of the clock) or unspecified (x). The body of a node consists in an optional list of local variables (var) and a list of equations (eq). Each equation defines the value of one or several variables using an expression on flows ($var = e$). Expressions may be immediate constants (i), variables (x), pairs ((e, e)), initialised delays (e `fby` e), applications ($N(e)$), Boolean sub-sampling (e `when` e), combination of flows on complementary Boolean clocks (`merge` $e\ e\ e$) or expressions using strictly periodic clocks ($epck$). $e/\hat{}k$ sub-samples e using a periodic clock division and $e *\hat{} k$ over-samples e using a periodic clock multiplication ($k \in \mathbb{N}^*$). $e \sim\!> q$ applies a phase offset of factor q ($q \in \mathbb{Q}^+$). Values k and q used in operations on strictly periodic clocks must be statically computable, furthermore they are not flows, they have no clock (they can be considered as always present with the same value).

$$i^{\#}((v,t).s) = (v,t).i^{\#}(s)$$
$$op^{\#}((v_1,t).s_1,(v_2,t).s_2) = (op(v_1,v_2),t).op^{\#}(s_1,s_2)$$
$$\texttt{fby}^{\#}((v_1,t).s_1,(v_2,t).s_2) = (v_1,t).\ \texttt{fby}'^{\#}(v_2,s_1,s_2)$$
$$\texttt{fby}'^{\#}(v,(v_1,t).s_1,(v_2,t).s_2) = (v,t).\ \texttt{fby}'^{\#}(v_2,s_1,s_2)$$
$$\texttt{when}^{\#}((v,t).s,(T,t).cs) = (v,t).\ \texttt{when}^{\#}(s,cs)$$
$$\texttt{when}^{\#}((v,t).s,(F,t).cs) = \ \texttt{when}^{\#}(s,cs)$$
$$\texttt{merge}^{\#}((T,t).s,(v,t).s_1,s_2)) = (v,t).\texttt{merge}^{\#}(s,s_1,s_2))$$
$$\texttt{merge}^{\#}((F,t).s,s_1,(v,t).s_2)) = (v,t).\texttt{merge}^{\#}(s,s_1,s_2)$$

Fig. 3. Synchronous semantics of classical operators.

B. Synchronous Temporised Semantics

We provide a semantics based on Kahn's semantics on sequences [10]. It is a direct adaptation of the synchronous semantics presented in [11] to the Tagged-Signal model. For any operator $\diamond$, $\diamond^{\#}(s_1,...,s_n) = s'$ means that the operator $\diamond$ applied to sequences s_1, ..., s_n produces the sequence s'. Term $(v,t).s$ denotes the flow whose head has value v and tag t and whose tail is sequence s. Operator semantics is defined inductively on the argument sequences. We omit rules on empty flows, which all return an empty flow. The semantics of some operators (for instance binary operators) is defined only for synchronous flows. The clock calculus will check that the program only combines synchronous flows and consequently that the semantics of the program is well-defined. In our model the absence of a flow is defined implicitly in its clocks: if a tag does not appear in a clock, then the corresponding flow is absent at this instant. As the sequence of tags of a clock is increasing, we can express the constraint that the arguments of an operator must be synchronous by requiring that at each step the tags of the heads of the arguments are the same. We start by redefining the semantics of classic synchronous operators in Fig. 3 (T stands for $true$ and F for $false$).

- A constant flow produces the same value every time it is present ($i^{\#}$).
- Binary operators are applied point-wisely ($op^{\#}$).
- x `fby` y concatenates the head of x to y, delaying the values of y by one tag.
- x `when` c produces a sub-sequence of x, only producing values of x when c is true.
- `merge` (c,x,y) combines flows x and y which are on complementary Boolean clocks: when c is true, x is present and y is absent. When c is false, x is absent and y is present. The result is present every time c is present (i.e. on the clock of c).

The synchronous semantics of operators on strictly periodic clocks is given in Fig. 4. These semantic rules are temporised in addition to being synchronous. Operator $*\hat{}$ produces values on tags that do not all appear in the flow on which they are applied. These new tags must respect some specific temporal properties relating them to the tags of the parameter flow. Function π is extended to flows, $\pi(f) = \pi(c)$ where c is the clock of flow f.

$$*\hat{}^{\#}((v,t).s,k) = \prod_{i=0}^{k-1}(v,t'_i).*\hat{}^{\#}(s,k)$$
$$\text{(where } \ t'_0 = t \ \text{ and } \ date(t'_{i+1}) - date(t'_i) = \pi(s)/k)$$
$$/\hat{}^{\#}(\prod_{i=0}^{k-1}(v_i,t_i).s,k) = (v_0,t_0)./\hat{}^{\#}(s,k)$$
$$\sim>^{\#}((v,t).s,q) = (v,t').\sim>^{\#}(s,q)$$
$$\text{(where } \ date(t') = date(t) + q\pi(s))$$

Fig. 4. Synchronous temporised semantics of operators on strictly periodic clocks.

- $x *\hat{} k$ produces a flow k times faster than x. Each value of x is duplicated k times in the result. The time interval between two successive values of the result (the period) is k times shorter than the interval between two successive values in x.
- $x/\hat{} k$ produces a flow k times slower than x. The result only keeps the first value of each k successive values of the argument.
- $x \sim> q$ delays each value of x by $q\pi(x)$ temporal units. The classic `pre` operator of LUSTRE is approximately equal to $\sim> 1$. However, on the contrary to `pre`, operation $\sim> 1$ does not require a specific analysis to check that the program does not access uninitialised values. Indeed, the clock calculus does not allow flows to be combined if they do not have the same phase (see Sect. IV).

Operation $\alpha \rightarrow_{.} q$ is defined for $q \in \mathbb{Q}$, because clocks unification (see Sect. IV) requires operation $\alpha \rightarrow_{.} q$ to be invertible ($\alpha \rightarrow_{.} -q$). However, operation $x \sim> q$ is defined only for $q \in \mathbb{Q}^{+}$. Indeed if q was negative, the values of $x \sim> q$ would have to be produced earlier than the values of x, effectively shortening the deadline for the computation of x. When trying to combine two flows that do not have the same phase, the user should prefer to delay the flow with the smallest phase instead of requiring the flow with the greatest phase to be produced earlier.

C. Example

Figure 5 shows the program for the MSU written with our new language. This case study is a simplified version of the real unit, in particular the details of nodes `upStream` and `downStream` do not correspond to the actual system. The MSU is duplicated for safety reasons, here we present only one of the duplicated processes. The activation condition c applied to the node `msu` is used to ignore the result when the system detects that the unit is faulty. The two units communicate through flows `otherMSU`

```
imported node basicOp(i,j,k) returns (o,p,q);
imported node A(i) returns (o); ...
wcet basicOp=40; wcet applyCmd=20; wcet A=30;
wcet B=10; wcet C=20; wcet D=40;wcet E=10; wcet F=30;

node upStream(i) returns (o1,o2)
let
  o1=A(B(i)); o2=C(i);
tel
node downStream(i) returns (o)
let
  o=D(E(F(i)));
tel
node msu(fromEnv,otherMSU)
  returns (toEnv,toOtherMSU)
var bop1,bop2,us1,us2,ds;
let
  bop1,bop2,toOtherMSU=
    basicOp(fromEnv,otherMSU,(0 fby ds)*^5);
  toEnv=applyCmd((0 fby us1)*^5,bop1);
  us1, us2=upstream(bop2/^5);
  ds=downStream(us2);
tel
node main(c,fromEnv:(100,0),otherMSU:(100,0))
  returns (toEnv,toOtherMSU)
let
  toEnv,toOtherMSU=(msu(fromEnv,otherMSU)) when c;
tel
```

Fig. 5. Program for the MSU.

and `toOtherMSU`. Operations `applyCmd` and `basicOp` execute on the fast rate (100ms period) while operations `upStream` and `downStream` execute on the slow rate (500ms period). Slow operations are further split into several sub-nodes. Inputs and outputs are respectively consumed and produced on the fast rate. Communications from fast to slow operations are performed instantaneously (from `basicOp` to `upStream`) while communications from slow to fast use a delay (**`fby`**, from `downStream` to `basicOp` for instance). The durations of imported nodes are provided at the beginning of the program (we did not list the complete declaration of imported nodes).

This system illustrates how multi-rate reactive systems can be programmed. The programmer first specifies the main operations of his system independently from each other in separated "functional nodes" (`basicOp`, `applyCmd`, `upStream` and `downStream`). Then he specifies the rates of each operation and how they communicate in an "assembly node" (`msu`). We added an optional third hierarchy level that instantiates strictly periodic clocks, by specifying their phase and period, and adds the activation condition. Node rates are determined by the strictly periodic clocks of their inputs and the synchronous semantics forces the programmer to precisely specify which data is exchanged during multi-rate communications through the use of periodic clock transformations. Here `upStream` consumes data produced by iterations 0, 5, 10, and so on, of `basicOp`. This is only one possible type of multi-rate communications that the programmer could specify with this language. He could as well have chosen to store five successive iterations of `basicOp` in an array and consume them all at each iteration of `upStream`. Still, the language enforces precise specification of communications. This does not leave room for non-deterministic communications that could change depending on the order in which operations are finally scheduled.

IV. Clock Calculus

Before translating a program to lower level code, the compiler performs static analysis to detect sources of run-time errors. Type-checking verifies that expressions only combine flows of the same types while the clock calculus checks that expressions only combine flows present at the same instants. The type-checking of our language is quite standard and is not detailed here. This section only details the clock calculus.

A. Clock Types

We say that an expression is *well-synchronized* if it does not access to undefined values (it only combines flows on the same clocks), *ill-synchronized* otherwise. The aim of the clock calculus is to verify that a program only uses well-synchronized expressions. If the program is well-synchronized then its semantics are well-defined (by the semantic rules of Sect. III-B). For each expression of the program, the clock calculus produces constraints that have to be met for the expression to be well-synchronized. As shown in [11], the clock calculus can use a type system to generate these constraints. The clock calculus produces judgements of the form $H \vdash e : cl$, meaning that "the expression e has clock cl in environment H". The grammar of clock types is given below:

$$
\begin{array}{lcl}
\sigma & ::= & \forall \alpha_1 <: C_1, ..., \alpha_m <: C_m.cl \\
cl & ::= & cl \rightarrow cl \mid cl \times cl \mid ck \mid c : ck \\
ck & ::= & ck \texttt{ on } c \mid ck \texttt{ on not } c \mid pck \\
pck & ::= & pck *_. k \mid pck /_. k \mid pck \rightarrow_. q \mid (n, p) \mid \alpha \\
c & ::= & nm \mid X \\
H & ::= & [x_1 : \sigma_1, ..., x_m : \sigma_m]
\end{array}
$$

Clock types can either be clock schemes (σ) quantified over a set of clock variables, which belong to different subtypes of clocks ($C_1, ..., C_m$), or unquantified clocks (cl). A clock can be a functional clock ($cl \rightarrow cl$), a clock product ($cl \times cl$), a clock sequence (ck) or a dependence ($c : ck$). A clock sequence can be a clock sampled using a Boolean condition (ck `on` c, ck `on not` c) or a strictly periodic clock (pck). The clock operator `on` is defined as follows (the opposite operator `on not` only keeps tags when c is false):

$$
\begin{array}{c}
\texttt{on}^{\#}(t.ck, (true, t).c) = t.\texttt{on}^{\#}(ck, c) \\
\texttt{on}^{\#}(t.ck, (false, t).c) = \texttt{on}^{\#}(ck, c)
\end{array}
$$

This requires c to be on clock ck ($c : ck$). A strictly periodic clock can be a strictly periodic clock over-sampled

periodically ($pck *_. k$), a strictly periodic clock undersampled periodically ($pck/_. k$), a strictly periodic clock with phase offset ($pck \rightarrow_. k$), a constant strictly periodic clock ((n, p)), or a clock variable (α). The values n and p are constant integer values. Integer values k and rational values q used in periodic clock transformations must be computable statically. A carrier (c) is either a name (nm) or a carrier variable (X). A strictly periodic clock α is a linear expression resulting of a succession of periodic clock transformations on another strictly periodic clock β. In the following, α will be called an *abstract strictly periodic clock* if β is a clock variable. It will be called a *concrete strictly periodic clock* if β is a strictly periodic clock constant.

We distinguish different subtypes among the global clock type set $\mathcal{C}$ in order to ensure that the periodic clock transformations used in the program will always produce valid strictly periodic clocks (see Sect. II-D). The two main clock subtypes of $\mathcal{C}$ are Boolean clocks $\mathcal{B}$ (clocks containing the operator `on`) and strictly periodic clocks $\mathcal{P}$. Boolean clocks and strictly periodic clocks can be mixed in the same program as they correspond to complementary notions. Strictly periodic clocks define the real-time frequency of a flow while Boolean clocks define activation condition of a flow. The clock grammar says that Boolean clocks can be derived from strictly periodic clocks but strictly periodic clocks cannot be derived from Boolean clocks. The programmer will first specify the base frequency of the flow and then specify that on this frequency the flow is present only if a certain condition is true. Clock type $\mathcal{P}$ is further split into subtypes $\mathcal{P}_{k,\frac{a}{b}}$, $k \in \mathbb{N}^*$, $\frac{a}{b} \in \mathbb{Q}^+$, $a \wedge b = 1$ (a and b are co-prime), $b|k$, where:

$$\mathcal{P}_{k,\frac{a}{b}} = \{(n, p)|(k|n) \wedge p \geq \frac{a}{b}\}$$

In other words, the set $\mathcal{P}_{k,q}$ contains all the strictly periodic clocks α for which transformations $\alpha *_. k$ and $\alpha \rightarrow_. -q$ produce valid strictly periodic clocks. Notice that $\mathcal{P} = \mathcal{P}_{1,0}$. The sub-typing relation $<:$ on clock types is defined as follows:

- $\mathcal{B} <: \mathcal{C}$, $\mathcal{P} <: \mathcal{C}$
- $\mathcal{P}_{k,q} <: \mathcal{P}_{k',q'} \Leftrightarrow k'|k \wedge q \geq q'$

The set of all subsets of $\mathcal{C}$ ordered by the subset inclusion $<:$ forms a lattice. We can apply classical results on subtyping and bounded types quantification [12].

Finally, clocks may be generalized (at a node definition) and instantiated (at a node call) as follows:

$$inst(\forall \alpha <: C.cl) = cl[(cl' \in C)/\alpha]$$
$$gen_H(cl) = \forall \alpha_1 <: C_1, ..., \alpha_m <: C_m.cl$$
$$\text{where } \alpha_1, ..., \alpha_m = FTV(cl) \backslash FTV(H)$$

This states that a clock scheme is instantiated by replacing clock variables by clocks belonging to the correct clock subtypes and that any clock variable can be generalized if it does not appear free in the environment (FTV stands for "free type variables").

$$\text{(SUB)}\ \frac{H \vdash e : ck \qquad H \vdash ck <: C}{H \vdash e : C}$$
$$(*\hat{}\,)\ \frac{H \vdash e : pck = \mathcal{P}_{k,0} \qquad k \neq 0}{H \vdash e *\hat{}\, k : pck *_. k}$$
$$(/\hat{}\,)\ \frac{H \vdash e : pck \qquad k \neq 0}{H \vdash e/\hat{}\, k : pck/_. k}$$
$$(\sim>)\ \frac{H \vdash e : pck = \mathcal{P}_{b,0} \qquad (q = \frac{a}{b}), a \wedge b = 1}{H \vdash e \sim> q : pck \rightarrow_. q}$$

Fig. 6. Strictly periodic clocks inference rules.

B. Clock Inference Rules

We give clock inference rules for operators on strictly periodic clocks in Fig. 6. The other inference rules remain the same as in [11]. A rule $\frac{A}{B}$ means that the type judgement B holds if and only if condition A holds.

- The *(SUB)* rule is the classical subsumption rule of sub-typing applied to clock types. It states that if ck is a clock subtype of C then any expression of clock ck also has clock type C.
- Operator $*\hat{}$ can only be applied to an expression the clock of which is a subtype of $\mathcal{P}_{k,0}$. This ensures that the period of the resulting clock is an integer.
- Operator $/\hat{}$ can only be applied to an expression the clock of which is strictly periodic.
- The parameter q of operator $\sim>$ denotes a rational $\frac{a}{b}$ for which $a \wedge b = 1$. The constraint $pck = \mathcal{P}_{b,0}$ ensures that the phase of $pck \rightarrow_. q$ is an integer.

To determine if the subsumption rule can be applied, we use the following properties:

Property 5. $\forall \alpha \in \mathcal{P}, \forall k, k' \in \mathbb{N}^{+*}, \forall q, q' \in \mathbb{Q}$:

- $\alpha *_. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k*k',q/k'}$
- $\alpha/_. k' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k/gcd(k,k'),q*k'}$
- $\alpha \rightarrow_. q' <: \mathcal{P}_{k,q} \Leftrightarrow \alpha <: \mathcal{P}_{k,max(0,(q-q'))}$
- $\alpha <: \mathcal{P}_{k,q} \wedge \alpha <: \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'} \Leftrightarrow \alpha <: \mathcal{P}_{lcm(k,k'),max(q,q')}$

These properties are derived from the definition of $\mathcal{P}_{k,q}$ and Property 2. When read from left to right, they form subsumption rules that can be applied to transfer subtyping constraints on a clock type to the clock variable appearing in it (for an abstract clock) or to the constant clock appearing in it (for a concrete clock). In the former case, the constraints are just added to the constraints already existing on this clock variable. In the latter case, we directly check that the constant clock verifies the constraints.

$$(\rightarrow)\ \frac{cl_1 = cl_1' \qquad cl_2 = cl_2'}{cl_1 \rightarrow cl_2 = cl_1' \rightarrow cl_2'} \qquad (\times)\ \frac{cl_1 = cl_1' \qquad cl_2 = cl_2'}{cl_1 \times cl_2 = cl_1' \times cl_2'}$$

$$(\text{ON})\ \frac{ck_1 = ck_2 \qquad c_1 = c_2}{ck_1\ \texttt{on}\ c_1 = ck_2\ \texttt{on}\ c_2}$$

$$(\text{VAR})\ \frac{\alpha <: \mathcal{P}_{k,q} \qquad pck <: \mathcal{P}_{k',q'} \qquad \alpha = NF(pck)}{\alpha = pck <: \mathcal{P}_{k,q} \cap \mathcal{P}_{k',q'}}$$

$$(\text{VAR'})\ \frac{\alpha \notin FTV(cl)}{\alpha = cl} \qquad (\text{CONST})\ \frac{n = n' \qquad p = p'}{(n,p) = (n',p')}$$

$$(*.)\ \frac{pck = pck'/.k}{pck *. k = pck'}$$

$$(/.)\ \frac{pck = pck' *. k \qquad pck' <: \mathcal{P}_{k,0}}{pck/.k = pck'}$$

$$(\rightarrow.)\ \frac{pck = pck' \rightarrow. (-\frac{a}{b}) \qquad pck' <: \mathcal{P}_{b,\frac{a}{b}}}{pck \rightarrow. \frac{a}{b} = pck'}$$

Fig. 7. Clock unification rules.

Computing a real-time schedule for the program is much more simple if we use concrete dates (concrete clocks) instead of dates parametrized by variables (abstract clocks). Therefore we constrain the clock type of the main node to be fully instantiated, no clock variable can appear in it. This means that the clocks of the main node inputs, from which all clocks of the program are derived, can only be concrete strictly periodic clocks and implies that any clock used in the program is derived from a strictly periodic clock.

On the contrary to classical synchronous languages, some expressions can now have a faster clock than the clock of the fastest input of the node in which they appear. This is an important feature which implies that the clocks hierarchy of a node does not form a tree anymore, the root of which is the base clock of the node (the clock of unsampled inputs). This enables greater flexibility in the specification of a system, as the programmer can choose any rate of its system as the base description rate of the system [1]. However, optimized compilation using loop instructions is for now an opened question.

C. Clocks Unification

The resolution of the set of constraints produced for a program follows [13] where typing judgements are treated as unificands, i.e. typing problems are reduced to unification problems. The resolution algorithm of the clock calculus consists in trying to find substitutions σ such that $\sigma(H) \vdash e : \sigma(cl)$, ie substitutions that unify $H \vdash e : cl$. Clock unification rules are given in Fig. 7. A rule $\frac{A}{B}$ means that to solve B we must solve A.

- We use structural unification for clocks $\rightarrow$ and $\times$.
- Boolean clocks (`on`) are unified syntactically: ck_1 `on` c_1 and ck_2 `on` c_2 can be unified if ck_1 and ck_2 can be unified and if c_1 and c_2 are syntactically equal. This simplification indeed implies that sometimes two equivalent clocks cannot be unified. Unification of Boolean clocks is performed syntactically while unification of strictly periodic clocks is performed semantically.
- In rule *(VAR)*, condition $\alpha = NF(pck)$ implicitly includes an occurrence check: if α appears in pck, then the normal form of pck must be α (or equivalently $\alpha *. k/.k \rightarrow. 0$). If α does not occur in pck, it can be substituted by pck. In both cases, pck inherits the subtyping constraints of α (in addition to its own constraints). The rule (*VAR'*) is used when a clock variable is unified with a clock that is not strictly periodic, $\alpha \notin FTV(cl)$ checks that α is not a free type variable of cl, i.e. that α does not occur in cl. (*VAR*) and (*VAR'*) are the only rules in which substitutions take place.
- Rule (*CONST*) says that two constant clocks can be unified if their periods and phases are equal.
- Rules $(*.)$, $(/.)$ and $(\rightarrow.)$ rely on Property 1 to transform the unification problem. Rules $(/.)$ and $(\rightarrow.)$ also introduce sub-typing constraints, the verification of which again relies on Property 5.

Strictly periodic clocks unification rules require priority rules, we check that two strictly periodic clock types pck_1 and pck_2 can be unified as follows:

- If pck_1 and pck_2 are concrete strictly periodic clocks, we can directly use Definitions 3, 4 and 5 to compute the period and phase of the clocks and that way reduce to rule (*CONST*).
- If pck_1 is an abstract strictly periodic clock and pck_2 is a concrete strictly periodic clock, we use rules $(*.), (/.)$ and $(\rightarrow.)$ recursively (with pck_1 appearing on the left of the rules) to reduce to rule (*VAR*). This leads to substituting the variable α appearing in pck_1 by a concrete strictly periodic clock.
- We use the same principle if pck_2 is an abstract strictly periodic clock and pck_2 is a concrete clock, this time with pck_2 appearing on the left of the rules.
- If pck_1 and pck_2 are both clock variables, we arbitrarily choose to apply the rules with pck_1 appearing on the left of the rules.

D. Example

We apply the clock calculus to the case study of the MSU (Fig. 1). We suppose that the clock of the inputs and outputs of an imported node are all the same. For instance the clock scheme of node `basicOp` is: $\forall \alpha <: \mathcal{C}.\alpha \times \alpha \times \alpha \rightarrow \alpha \times \alpha \times \alpha$. We get the same kind of clock schemes for the "functional nodes" as they do not introduce clock constraints on their inputs and outputs. Concerning node `msu`, let c_1, c_2, c_3 be respectively the clocks of variables `fromEnv`, `otherMSU` and `ds`:

- `(0 fby ds)*^5` has clock $c_3 * .5$ and $c_3 <: \mathcal{P}_5$.

- `basicOp(fromEnv,otherMSU,(0 fby ds) *^5)`: we need to unify c_1, c_2 and $c_3 * .5$, so we let $c_2 = c_1$ and $c_3 = c_1/.5$. We also get $c_1 <: \mathcal{P}$.
- `bop1,bop2,toOtherMSU=basicOp(...)`: variables `toOtherMSU`, `bop1` and `bop2` all have clock c_1.
- Similarly, `toEnv` has clock c_1. `us1, us2` and `ds` have clock $c_1/.5$.
- We can generalize clock c_1 and obtain the clock of node `msu`: $\forall \alpha <: \mathcal{P}.\alpha \times \alpha \rightarrow \alpha \times \alpha$.

Concerning node `main`:

- `msu(fromEnv,otherMSU)`: instantiates the clock of the node `msu` to clock $(100,0) \times (100,0) \rightarrow (100,0) \times (100,0)$
- `(msu(fromEnv,otherMSU)) when c`: changes the clock of the result to $(100,0)$ `on` $c \times (100,0)$ `on` c
- We finally get the clock type of node `main`: $(100,0) \times (100,0) \rightarrow (100,0)$ `on` $c \times (100,0)$ `on` c

V. Sequential Code Generation

Synchronous languages compilers do not directly generate assembly code, but instead intermediate level code (C code). The synchronous compiler translates the input program consisting in a list of equations into an output program consisting in a sequence of instructions. The expressions used in equations are first ordered (scheduled) and then translated into lower level code.

A. Scheduling

We use notations found in [14] to summarize our scheduling problem. The atomic computation steps of a synchronous program (ie the *tasks* the program is made up of) are the leaf nodes of the nodes hierarchy, that is calls to either predefined nodes or imported nodes. Node calls are partially ordered by data dependencies between variables. Strictly periodic clocks introduce periodicity constraints. If the clock α of a flow f is strictly periodic, then the activation dates of f can be computed statically using the phase and the period of α. If α is a Boolean clock, the activation dates of f cannot be computed statically, so we need to approximate α. As mentioned in Sect. IV-B, the clock of every flow is derived from a strictly periodic clock. Thus, for the scheduling phase we can over-approximate any clock to its strictly periodic clock parent defined as follows:

$$pparent(\alpha \texttt{ on } c) = pparent(\alpha)$$
$$pparent(pck) = pck$$

Each task τ_i is characterized by its period T_i, its initial release date s_i, its relative deadline D_i and its computation time C_i. For a task τ_i of clock α, $T_i = \pi(pparent(\alpha))$,

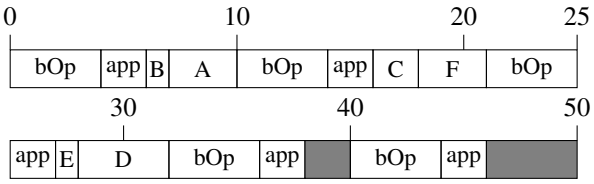

Fig. 8. Schedule for the MSU.

$s_i = \varphi(pparent(\alpha))$ and C_i is the wcet specified for the corresponding node. Initially, $D_i = T_i$, however due to data dependencies and periodic clock multiplications, the actual deadline may be earlier. For instance, let x be a flow produced on some clock (n,p). The deadline for its production is initially set to period n. If x is under-sampled by 2 (`x*^2`), the deadline for x becomes $n/2$. Still, if x is delayed before being under-sampled (`(0 fby x)*^2`), the under-sampling does not change the deadline for x.

A survey of scheduling problems and algorithms can be found in [14]. We want to schedule a set of tasks (the node calls), related by precedence constraints, and subject to periodicity constraints. Due to the criticity of the systems we consider, we choose off-line scheduling, where the schedule is computed at compilation-time, rather than on-line scheduling, where the schedule is computed at execution-time. The main benefit is that the temporal behaviour of the program is deterministic since every action is planned within the schedule. For the same reason we choose non-preemptive scheduling, which avoids hard to predict run-time overhead. The schedule is computed on time interval $[0, 2P + max\{r_i\}[$, where P is the least common multiple of the periods of the tasks [14]. The schedule becomes cyclical after time $P+max\{r_i\}$ (a cycle has duration P). The scheduling problem we want to solve is known to be NP-complete for the class of off-line non-preemptive algorithms [15]. However exact solutions that work fine in practice exist (i.e. that generally do not require exponential computation time with respect to the number of tasks), for instance [16], and can be reused.

Figure 8 gives a possible schedule for the MSU. For better readability, we did not detail the scheduling of predefined operators. In this particular example the schedule directly becomes cyclical because the slow period is a multiple of the fast period. Gray parts correspond to idle times. Notice that slow operations can spread over two successive fast cycles, which cannot happen when programming with classic synchronous languages on the fastest rate of the system.

B. Schedule Translation

The schedule can be translated into classic sequential code that requires only simple real-time primitives from the operating system to execute. The translation of each node into instructions of the target code, the variable allocations

and most optimizations can reuse standard synchronous languages compilation techniques. Operators on strictly periodic clocks mainly impact the scheduling phase. Once node calls are scheduled, they are simply translated into the corresponding function call. For instance, the code corresponding to the schedule of Fig. 8 first acquires inputs, then calls function corresponding to node `basicOp`, calls function for node `applyCmd`, produces outputs, calls function for node `B`, and so on. This sequence is repeated by a standard `while` loop. Idle times can be handled by a simple idling primitive $wait$. The instruction $wait(t)$ causes the execution to wait until date t and then to resume to the next instruction.

VI. Related Work

Statically computable clocks have also been studied in [17] (ultimately periodic clocks) and [18] (affine clock transformations). However, both of these models are not directly related to real-time, which means that the fact that a process A executes n times when a process B executes one time does not imply that the period of A is n times shorter than the period of B. This again prevents efficient multi-periodic scheduling. Real-time periodic clocks have previously been introduced in [8], however they lacked specific clock transformations to efficiently handle multi-rate communications. Multi-periodic synchronous data-flow is also the subject of [19] though this work focuses more on the scheduling problem instead of on the definition of a formal compilable language. Finally, strictly periodic clocks do not solve the more general problem of communicating processes that have unrelated clocks (as presented in [20]). Indeed, strictly periodic clocks are all related to the same real-time scale.

VII. Conclusion

We proposed a formal language for programming multi-rate reactive systems, its semantics and its compilation. This language builds upon data-flow synchronous languages and adds real-time primitives to handle the multi-rate aspects. These primitives are based real-time clocks called strictly periodic clocks. A prototype of the clock calculus has been implemented in OCaml. It is about 2500 code lines long, including syntax analysis, typing, clock calculus and proper error handling. It should be available shortly at `http://www.cert.fr/anglais/deri/jforget/`. Future work will take better advantage of the properties of strictly periodic clocks for the code generation process. Indeed, the current code might be long as it is directly generated from one cycle of the schedule. The formal semantics of operators on strictly periodic clocks suggests that strictly periodic clocks can be compiled more efficiently using loop instructions.

References

[1] J. Forget, F. Boniol, D. Lesens, C. Pagetti, and M. Pouzet, "Programming languages for hard real-time embedded systems," in *4th European Congress on Embedded Real-Time Software (ERTS'08)*, Jan. 2008.

[2] *Simulink: User's Guide*, The Mathworks.

[3] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," Carnegie Mellon University, Tech. Rep., 2006.

[4] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," in *Readings in hardware/-software co-design*. Kluwer Academic Publishers, 2001.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data-flow programming language LUSTRE," *Proc. IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991.

[6] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: the SIGNAL language and its semantics," *Sci. of Compu. Prog.*, vol. 16, no. 2, 1991.

[7] E. A. Lee and A. L. Sangiovanni-Vincentelli, "Comparing models of computation," in *International Conference on Computer Aided Design (ICCAD)*, 1996.

[8] A. Curic, "Implementing lustre programs on distributed platforms with real-time constraints," Ph.D. dissertation, Université Joseph Fourier, Grenoble, Jul. 2005.

[9] M. Pouzet, *Lucid Synchrone, version 3. Tutorial and reference manual*, Université Paris-Sud, LRI, Apr. 2006.

[10] G. Kahn, "The semantics of simple language for parallel programming," in *International Federation for Information Processing (IFIP) Congress*, 1974.

[11] J.-L. Colaço and M. Pouzet, "Clocks as first class abstract types," in *Third International Conference on Embedded Software (EMSOFT'03)*, S. Berlin/Heidelberg, Ed., 2003.

[12] B. C. Pierce, *Types and programming languages*. Cambridge, MA, USA: MIT Press, 2002.

[13] D. Rémy, "Extending ML type system with a sorted equational theory," INRIA, Rocquencourt, France, Tech. Rep. 1766, 1992.

[14] L. George, N. Rivierre, and M. Spuri, "Preemptive and non-preemptive real-time uniprocessor scheduling," INRIA, Tech. Rep. RR-2966, 1996.

[15] M. R. Garey and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Trans. Softw. Eng.*, vol. 16, no. 3, pp. 360–369, 1990.

[17] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-Synchronous Kahn Networks: a relaxed model of synchrony for real-time systems," in *ACM International Conference on Principles of Programming Languages (POPL'06)*, USA, 2006.

[18] I. Smarandache and P. Le Guernic, "A canonical form for affine relations in signal," INRIA, Tech. Rep. RR-3097, 1997.

[19] L. Cucu and Y. Sorel, "Real-time scheduling for systems with precedence, periodicity and latency constraints," in *10th International Conference on Real-Time Systems (RTS'02)*, 2002.

[20] G. Berry and E. Sentovich, "Multiclock esterel," in *11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'01)*. Springer-Verlag, 2001.