A synchronous language with partial delay specification for real-time systems programming

Rémy Wyss, Frédéric Boniol, Julien Forget, Claire Pagetti

A synchronous language with partial delay specification for real-time systems programming

Rémy Wyss[1], Frédéric Boniol[1], Julien Forget[2], and Claire Pagetti[1]

[1]ONERA–Toulouse, France, [2]LIFL/USTL–Lille, France

Abstract. High-level formal programming languages require system designers to provide a very precise description of the system during early development phases, which may in some cases lead to arbitrary choices (i.e. the designer "overspecifies" the system). In this paper, we propose an extension of synchronous dataflow languages where the designer can specify that he does not care whether some communication is immediate or delayed. It is then up to the compiler to choose where to introduce delays, in a way that breaks causality cycles and satisfies latency requirements imposed on the system.

1 Introduction

Context. Implementing real-time critical systems is an increasingly complex process that calls for high-level formal programming languages. In this paper, we focus on synchronous languages [1], such as Lustre [6] and its commercial version Scade [4], which have successfully been adopted for the formal specification of control systems. These languages are based on the *synchronous paradigm*, where the behaviour of a program is seen as a sequence of reactions: (1) each reaction consists in reading the current inputs, computing the current outputs and updating the internal state of the system; (2) each reaction occurs in zero *logical time* (we do not care about when computations occur during a reaction). Relying on such a language imposes very early in the development process to specify a completely deterministic system, while in some specific cases the designer might want to leave some degree of liberty in the specification. This paper details how to avoid such overspecification.

Non deterministic communication specification. Let us consider the simplified mono-periodic flight control system depicted in the Figure 1. It consists of a set of avionics functions, which acquire information on the state of the aircraft and on the pilot orders, and which objective is to control the position, speed and attitude of the vehicle thanks to its control surfaces. The right part of the figure depicts the control of the ailerons while the left part depicts the control of the elevators. Each vertex depicts a function. Edges depict data-communications between functions and are of two different kinds. Plain arrows stand for immediate communications, which induce a precedence constraint from the producer to the consumer. Dashed arrows stand for less constrained communications that do not induce precedence constraints. We illustrate this distinction below.

The variable `a_angle` stands for the current angle of the aileron and is acquired by the function AAF (Aileron angle filter). This function consolidates

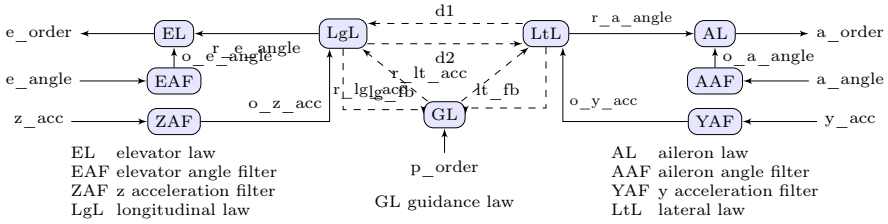


Fig. 1. A simplified flight control system

the data and sends the variable `o_a_angle` (the observed angle) to the function AL (Aileron law). AL controls the aileron and maintains the required angle `r_a_angle`. According to the observed angle and the required angle, it sends an order `a_order` that enables to reach safely the required angle. Thus, the differential equations that command the aileron surface are implemented by the data-flow L_1 = `a_angle` → `o_a_angle` → `a_order`. Such a computation path, from a sensor acquisition to an actuator order is called a *functional chain*. In this case, L_1 corresponds to the discretisation of a command law and functions must be applied strictly sequentially for the computation to be correct. The elevator law behaves similarly.

The control laws of the ailerons and of the elevators communicate through the functions `LtL` and `LgL`, in order to verify that the orders sent to the different actuators (ailerons and elevators) are consistent. This consolidation is however less stringent than the command described in L_1, therefore longer latencies in the communication can be supported. As a result, communications between such functions do not impose strict precedence constraints and a function is allowed to compute using data produced by another function during the previous reaction instead of the current reaction (delayed communication). Still, the comparison and consolidation must be done on sufficiently "fresh" data. Thus, the number of delayed communications that the functional chain L_2 = `z_acc` → `o_z_acc` → `d2` → `r_a_angle` → `a_order` can tolerate is bounded by a maximum latency constraint.

The last part of the system, the guidance law (`GL`), computes a series of accelerations to apply in order to reach a position ordered by the pilot. This is described by the chain L_3 = `p_order` → `r_lt_acc` → `r_a_angle` → `a_order`. This function is temporally less constrained since the required position is reached only after several execution steps (several reactions). The function could even be implemented as a function executed less often than every logical instant. Thus, communications between `GL` and other functions are flexible.

This illustrates two cases where the communication specification should remain non deterministic (unspecified): (1) when loosely coupled control laws communicate, (2) when there is a global latency imposed over a functional chain; local delays between each component of the chain are not explicitly required and may remain unspecified by the designer, provided that at run time the global end-to-end latency is satisfied.

Allowing such a liberty in the way functions communicate eases the design of large systems that are typically made up of several thousand components (for instance, a flight control system is made up of about 5.000 functional nodes). In such a system, specifying the temporal semantics of all the communications (more than 20.000 in a flight control system) is not humanly reasonable. A better solution would be to let designers specify only for critical communications and to leave the rest unspecified, meaning that he "does not care" whether the communications are immediate or delayed. The second advantage of this liberty is to ease the scheduling as it allows for more parallelism, in particular when the system is implemented on a distributed platform.

Contribution. With existing synchronous languages, the designer must explicitely choose between immediate and delayed communications, even for flexible communications. In this paper we detail how to transform this manual choice into an automated choice included in the compilation process. Notice that we still allow a system to contain explicitly immediate or delayed communications (for instance to memorize the current state of the system requires a delay). The new type of flexible communications comes as an additionnal mechanism, not as a replacement for either immediate or delayed communications.

We define a synchronous language with partial delay specification that provides a higher degree of flexibility during the design phase. We introduce a new operator called “don’t care” to specify communications that are allowed to be either immediate or delayed. The semantics of the language is defined formally with a Kahn’s semantics in section 2. A program with “don’t care” operators is partially non deterministic as it accepts several different behaviours. The language also enables to impose latency constraints expressed as bounds on the number of delays involved in functional chains.

Though the initial specification is non deterministic, we want to generate completely deterministic code in the end, as we are interested in highly critical systems. We propose to translate a program with “don’t care” operators into a standard synchronous program that respects the latency requirements imposed on the functional chains and that is causal (meaning that there is no cycle without delay). We have chosen a deterministic syntactic translation (section 3), which either simply drops a “don’t care” operator or replaces it by a classic delay operator (the `pre`/`fby` of Lustre/Lucid Synchrone [6,17]). We show that finding a valid translation (choosing where to keep delays) is equivalent to solving a set of pseudo-Boolean constraints and can thus be solved using existing techniques for this widely studied problem.

Related work. Real-time embedded code is often implemented as a set of threads scheduled concurrently by a scheduling policy [3]. These policies focus on ensuring the respect of temporal constraints and do not consider functional determinism as tasks are assumed to be completely independent. We might say that the system behaves as if all data-flows were tagged with a “don’t care”. There is however an important difference with our language: two different executions of

the same system may have different functional behaviours due to timing variations (if a task executes for less than its worst-case execution time for instance). Precedence constraints can be imposed on tasks imposing direct communication and thus ensuring an order between tasks, however we lose the flexibility of the "don't care".

Architecture Description Languages such AADL (*Architecture and Analysis Design Language*) [5], MARTE (*Modeling and Analysis of Real-time and Embedded systems*) [13] or SYSML (*System Modeling Language*) [14] enable to specify detailled communication patterns between functional components. In AADL, data communications are either immediate or delayed. Immediate connections impose a precedence constraint from the producer to the consumer of the data, while delayed communications do not. End-to-end latency constraints can also be imposed on flows (a flow is a path through sub-components of a component). However, the patterns are always fully deterministic and no mechanism allows to handle "don't care" communications. The CCSL language [11] used in MARTE is closer to our work. It enables to specify clock constraints, that is to say constraints on the relation between the activation rates of the different operations of the system. This can yield several different instanciated systems that satisfy the constraints.

In [15] and [10], the authors introduce a methodology to optimise the computation time of synchronous circuits. For a given deterministic specification, the method rewrites the specification into a functionally equivalent one by modifying the equations. To this intent, a synchronous program is represented as a graph where the vertices are functional units and the edges represents the communication between two functional units. The edges are tagged with an integer which represents the number of delay registers required for the communication. The idea of the methodology, called retiming, is to apply graph transformations which insert and remove delays.

In [12], the authors define a non deterministic language based on Input/Output Boolean Automata. The semantics of a non deterministic program is defined as the set of deterministic programs which behaviour is included in the non deterministic ones. This language has a higher expressive power than our data-flow language. The authors then define the product of non deterministic specifications. To convert a non deterministic specification into a deterministic one, the designer needs to define an "oracle" and synchronise the program with this oracle. How to define an oracle is out of the scope of the paper.

2 A synchronous language with partial delay specification

In this section, we extend synchronous data-flow languages with the ability to have partial delay specifications using a new *"don't care"* operator.

2.1 Syntax

We define the syntax of a very simple first-order synchronous data-flow language. The language is meant as a proof-of-concept and those concepts should later be included in a larger, more realistic language. The grammar is the following:

$$
\begin{array}{lll}
\mathit{syst} & ::= & \mathit{n\_list}; \\
 & & \texttt{node}\ N(\mathit{io})\ \texttt{returns}\ (\mathit{io})\ [\ \texttt{var}\ \mathit{io}; \mathit{req\_list};]\ \texttt{let}\ \mathit{eqs}\ \texttt{tel} \\
\mathit{n\_list} & ::= & \mathit{imported\ node\ N(io)\ returns\ (io)} \\
 & & \mid \mathit{imported\ node\ N(io)\ returns\ (io)}; \mathit{n\_list} \\
\mathit{io} & ::= & \mathit{id} \mid \mathit{io}, \mathit{id} \\
\mathit{eqs} & ::= & \mathit{eq} \mid \mathit{eqs}\ \mathit{eq} \\
\mathit{eq} & ::= & \mathit{io} = \mathit{exp}; \\
\mathit{exp} & ::= & \mathit{id} \mid \mathit{cst} \mid \mathit{dop}(\mathit{id}, \ldots, \mathit{id}) \mid \mathit{cst}\ \texttt{fby}\ \mathit{id} \mid \mathit{cst}\ \texttt{dc}\ \mathit{id}
\end{array}
$$

A program (*syst*) is made up of a list of imported nodes (*n_list*) and a unique main node which assembles the set of imported nodes. Imported nodes are black boxes (programmed with other existing languages) that have no side effects on the rest of the program. The program takes a set of inputs, produces a set of outputs and can use local variables. All of these are described by *io*. It is possible to specify a set of requirements (*req_list*). This will be explained in section 2.3. The assembly of imported nodes is described as a set of equations (*eqs*). An equation computes a list of local or output flows using a flow expression (*expr*). A flow expression can be a flow variable (*id*), a constant (*cst*), the application of an imported node (*dop*) on flows, the application of the `fby` operator (the delay operator of Lucid Synchrone [17]) on a flow. This *"followed by"* operator introduces a delay of one reaction between its input and its output flow (parameter `cst` is the initialization value for the first reaction, as it has no previous value). Finally an expression can be the application of the new *"partial delay"* operator `dc` (called "don't care") on a flow. The equation $x = cst\ \texttt{dc}\ x'$ can correspond to either $x = x'$ or $x = cst\ \texttt{fby}\ x'$ and thus introduces some non determinism. In the first case, the communication between x' and x is immediate and in the second it is delayed. Again, the operator `dc` can only be applied to identifiers and not to expressions in order to simplify our presentation. Notice that programs contain no hierarchy (no nodes decomposed into other non-imported nodes). Modular compilation/analysis is out of the scope of this paper.

Example 1. Let us illustrate our syntax with the textual version of the flight control system of Figure 1.

```
imported node AL(i1, i2) returns (o);
imported node EL(i1, i2) returns (o);
imported node GL(i1, i2, i3) returns (o1, o2);
...
node FCS(a_angle, y_acc, e_angle, z_acc, p_order)
returns (a_order,  e_order)
  var o_e_angle, o_a_angle, r_a_angle, o_y_acc, r_e_angle, o_z_acc,
      lg_fb, r_lg_acc, d1, d2, r_lt_acc, lt_fb,  dc1, dc2, dc3, dc4, dc5, dc6;
let
  o_a_angle = AAF(a_angle);            o_y_acc = YAF(y_acc);
  o_z_acc = ZAF(z_acc);                o_e_angle = EAF(e_angle);
  o_z_acc = ZAF(z_acc);                o_e_angle = EAF(e_angle);
  (lg_fb, d2, r_e_angle) = LgL(dc1, o_z_acc, dc4);
  (r_a_angle, lt_fb, d1) = LtL(dc2, o_y_acc, dc6);
  (r_lt_acc, r_lg_acc) = GL(dc3, dc5, p_order);
  a_order = AL(o_a_angle,r_a_angle);
  e_order = EL(r_e_angle, o_e_angle);
```

```
  dc1 = 0 dc d1;                    dc2 = 0 dc d2;
  dc3 = 1 dc lt_fb;                 dc4 = 0 dc r_lg_acc;
  dc5 = 1 dc lg_fb;                 dc6 = 0 dc r_lt_acc;
tel
```

At the beginning of the code we declare 9 imported nodes, one for each functional block (we only show 3 declarations here for more conciseness). After that we define the node FCS which takes 4 inputs (a_angle, y_acc, e_angle and z_acc) and returns 2 outputs (a_order and e_order). We also declare a set of local variables which correspond to the variables exchanged between the nodes. The set of equations between **let** and **tel** defines how outputs and local variables are computed from inputs and how imported nodes communicate. To specify the "don't care" communications accordingly to the syntax, we add 6 auxiliary variables ($dc1$ to $dc6$).

Since the language does not include over-sampling or sub-sampling operators (such as `when` or `current`), all the flows are on the same clock (the base logical time). Therefore we do not provide any clock calculus. Mixing sampling operators and *"don't care"* will be studied in future work. All the flows are well initialised because the delay operator (`fby`) includes an initialisation value.

2.2 Kahn's semantics

We provide a Kahn's semantics [8] for our language, which details the sequence of values corresponding to flow expressions. We introduce the following grammar to represent such sequences:

$$E ::= \{x_1, \ldots, x_n\} \qquad s ::= E.s$$

E is a set of values. $E.s$ denotes the sequence whose head value is non deterministically chosen among the values in E and whose tail is s. We abusively write $v.s$ instead of $\{v\}.s$ to denote a flow whose first value is deterministic. Finally, we define:

$$< E_1.s_1 | \ldots | E_n.s_n >= \cup_{1 \leq i \leq n} E_i. < s_1 | \ldots | s_n >$$

For any operator $\diamond$, $\diamond^{\#}(p_1, ..., p_n) = s$ means that the operator $\diamond$ applied to parameters $p_1, ..., p_n$ produces the sequence s. Let $\prod_{i=1}^{n} E_i = E_1 \times \ldots \times E_n$ denote the cartesian product of the sets E_i. The semantics of flow expressions is given with:

$$cst^{\#} = cst.cst^{\#}$$

$$imp^{\#}(E_1.s_1, \ldots, E_m.s_m) = \cup_{t \in \prod_{i=1}^{m} E_i} \{imp(t)\}.imp^{\#}(s_1, \ldots, s_m)$$

$$\texttt{fby}^{\#}(cst, s) = cst.s^{\#}$$

$$\texttt{dc}^{\#}(cst, s) = < \texttt{fby}^{\#}(cst, s) | s^{\#} >$$

1. The first rule describes the constant: it simply produces a flow that always has the same value.

2. The second rule is the call of an imported node. At each step of the sequence, we may take as input for the node any combination of the non deterministic values of its inputs (any tuple in the cartesian product of the possible head values of each input) and then we apply the imported node to each combination. In other words, we simply apply an imported node point-by-point on the sequence of values of its inputs.
3. The third rule is for the `fby` operator. It simply concatenates the initialisation value to the flow, thus delaying each value of the input flow by one reaction.
4. The last rule is for the `dc` operator. It introduces non-determinism by stating that the `dc` operator can either be replaced by a `fby` or by the identity.

Example 2. We illustrate the semantics in an example where the imported node *plus* simply computes the addition of the two inputs.

```
imported node plus (i1,i2) returns (o);
node ex (i) returns (o)
var v1, v2;
let
  v1 = 0 dc i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

i	5	3	7	...
v_1	$\{0,5\}$	$\{5,3\}$	$\{3,7\}$	
v_2	$\{1\}$	$\{0,5\}$	$\{5,3\}$	
o	$\{1,6\}$	$\{3,5,8,10\}$	$\{6,8,10,12\}$	

2.3 Latency requirements

Let $x \xrightarrow{0} x'$ denote that the variable x' immediately depends on the variable x. Similarly, $x \xrightarrow{1} x'$ denotes a delayed dependency and $x \xrightarrow{?} x'$ a "don't care" dependency. Let $x \to x' \overset{def}{\Leftrightarrow} x \xrightarrow{0} x' \vee x \xrightarrow{1} x' \vee x \xrightarrow{?} x'$. Let $Var_0(e)$ denote the variables that appear free in e and not as an argument of a `fby` or a `dc`, $Var_1(e)$ those that are an argument of a `fby` and $Var_?(e)$ those argument of a `dc`. Given a program p and $c \in \{0, 1, ?\}$, we have $x \xrightarrow{c} x'$ in p iff there is an equation $io = e$ in p such that $x' \in io$ and $x \in Var_c(e)$.

Definition 1 (Functional chain). *A functional chain $(x_1, \ldots, x_n)$ is a list of flow variables such that $\forall 1 \leq i < n$, $x_i \to x_{i+1}$.*

Example 3. Let us consider again the program `ex` of example 2. There are two chains from the input i to the output o: $L = (i, v_1, v_2, o)$ and $L' = (i, v_1, o)$.

The latency of a functional chain is the number of delays of the chain. Latency constraints impose a bound on this number and are in most cases maximum latency constraints, i.e. constraints of the form $Lat(x_1 \ldots x_n) < k$, where $k \in \mathbb{N}^\star$ and $(x_1 \ldots x_n)$ is a functional chain. Our approach can however support constraints where $<$ is replaced by any other comparison operator. In a program, latency constraints are specified using the keyword `req`.

Example 4. The flight control system specified in example 1 contains several chains. The requirements on those mentioned in the introduction are:

```
req (z_acc, o_z_acc, d2, dc2, r_a_angle, a_order) < 1; --L2
req (p_order, r_lt_acc, dc6, r_a_angle, a_order) < 4; --L3
```

3 Concretisation of an abstract program

We propose a compilation scheme in two steps: a) Translate the specification into a standard deterministic synchronous data-flow program; b) Apply a synchronous compiler. The second step is well studied [7] and is out of the scope of this paper. This section describes the first step, which consists in choosing one program among the different deterministic synchronous programs which semantics is included in the semantics of the original non deterministic program. The compiler performs a *purely syntactic* translation that replaces each "don't care" operator either by a direct communication or by a delayed communication.

3.1 Instance space

Non deterministic programs are called *abstract* programs, while deterministic ones are called *concrete* programs.

Definition 2. *A system (or a program) is said* concrete *if and only if it contains no* `dc` *operations, otherwise it is* abstract*.*

Example 5. The abstract program `ex` given in example 2 corresponds to several concrete versions. If the "don't care" operator is syntactically translated into either the identity or the `fby` operator, there are 2 solutions (`ex1` and `ex2`). A more complex solution (not supported in our approach) would be to interleave direct and delayed communications (for instance, `ex3`).

```
node ex1 (i) returns (o)
var v1, v2;
let
  v1 = i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

```
node ex2 (i) returns (o)
var v1, v2;
let
  v1 = 0 fby i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

```
node ex3 (i) returns (o)
var v1, v2, j;
let
  j = true fby (not j);
  v1 = if j then i
       else (0 fby i);
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

Let sys denote an abstract system. Let $\texttt{dc}(sys) = \{dc_1, \ldots, dc_n\}$ denote the ordered set of `dc` operators in sys (we take the chronological apparition order of the operators in the set of equations). Let $p = (dc_i \mapsto \texttt{fby})sys$ denote the program p resulting of the substitution of operator dc_i by a `fby` operator in sys. Similarly, $(dc_i \mapsto id)sys$ denotes the substitution of dc_i by the identity operator and $(dc_x \mapsto op, ..., dc_y \mapsto op)sys$ denotes the program resulting of the set of substitutions $dc_x \mapsto op, ..., dc_y \mapsto op$ (where op is either `fby` or the identity).

Definition 3 (Instance). *Let p be a concrete system and sys be an abstract program such that $\texttt{dc}(sys) = \{dc_1, \ldots, dc_n\}$. p is an instance of sys iff there exists a set of substitutions $dc_1 \mapsto op_1, ..., dc_n \mapsto op_n$ such that:*

$$p = (dc_1 \mapsto op_1, ..., dc_n \mapsto op_n)sys$$

In the following, $sys[b_1, \ldots, b_n]$ denotes the instance $p = (sub_1, \ldots, sub_n)sys$ such that:

$$\begin{cases} sub_i = dc_i \mapsto id & \text{if } b_i = 0 \\ sub_i = dc_i \mapsto \texttt{fby} & \text{if } b_i = 1 \end{cases}$$

Example 6. The program `ex` has two instances as shown in example 5, with `ex1=ex[0]` and `ex2=ex[1]`.

Let us denote by $\mathcal{I}(sys)$ the set of instances of the system $sys(dc_1, \ldots, dc_n)$. We define the relation $\sqsubseteq_{sys}$ between instances of sys as follows:

Definition 4. *Let $p = sys[n_1, \ldots, n_m]$ and $p' = sys[n'_1, \ldots, n'_m]$ be two instances of sys:*

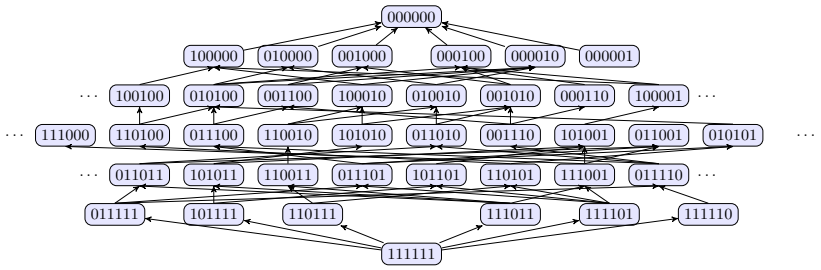
$$p \sqsubseteq_{sys} p' \overset{def}{\iff} \forall i \leq m, n'_i \leq n_i$$

In other words, `dc` *communications are "more immediate" in p' than in p.*

Proposition 1. *$(\mathcal{I}(sys), \sqsubseteq_{sys})$ is a finite lattice in which the top element is $sys[0, \ldots, 0]$ (all* `dc` *communications are immediate) and the bottom element is $sys[1, \ldots, 1]$ (all* `dc` *communications are delayed).*

This is a very classical example of a Boolean lattice. We use a Hasse diagram to represent the lattice $(\mathcal{I}(sys), \sqsubseteq_{sys})$. In such a diagram, each element of $\mathcal{I}(sys)$ is a vertex and there is an upward edge from s to s' whenever $s \sqsubseteq s'$ and there is no s'' such that $s \sqsubseteq s'' \sqsubseteq s'$. Notice that $(\mathcal{I}(sys), \sqsubseteq_{sys})$ is a n-dimensional hypercube.

Example 7. The instance space of the simplified flight control system $FCS(dc1, dc2, dc3, dc4, dc5, dc6)$ is described by the 6-dimensional hypercube:



3.2 Valid instance space

Only a subset of the instances of an abstract program are valid. We say that an instance p of sys is *valid* if and only if 1) p is causal and 2) p satisfies all the latency requirements of the system. Part 1) ensures that the instance is implementable (by checking that the instance does not contain an data-dependency cycle). However, not all implementable instances satisfy latency requirements, thus we also need to check 2).

Respecting latency requirements. Let us first define formally the latency of a functional chain.

Definition 5. *Let sys be a system and $(x_1, \ldots, x_n)$ be a functional chain of sys. The latency of the chain in sys is defined inductively as:*

$$Lat_{sys}(x_1,\ldots,x_n) = \begin{cases} 1 + Lat_{sys}(x_1,\ldots,x_{n-1}) & \text{if } x_{n-1} \xrightarrow{1} x_n \\ Lat_{sys}(x_1,\ldots,x_{n-1}) & \text{if } x_{n-1} \xrightarrow{0} x_n \text{ or } x_{n-1} \xrightarrow{?} x_n \end{cases}$$

with $Lat_{sys}() = 0$*, where* $()$ *is the empty functional chain.*

Example 8. For program `ex` and the chain $L = (i, v_1, v_2, o)$, $Lat_{ex}(L) = Lat_{ex1}(L) = 1$ and $Lat_{ex2}(L) = 2$.

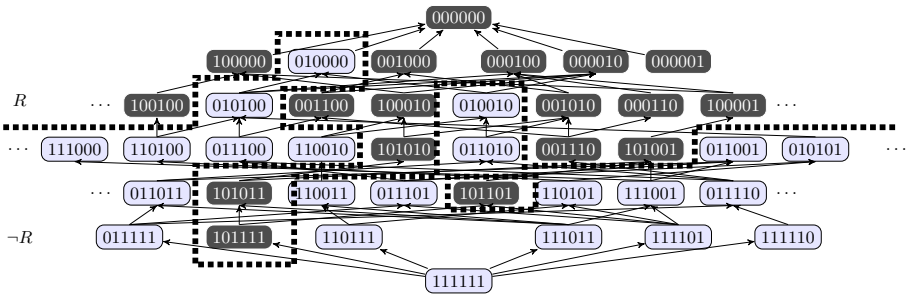
We say that dc_i *is involved in* $\mathcal{C}$ if there exists two variables x, x' in $\mathcal{C}$ such that $x \xrightarrow{?} x'$ and dc_i is the `dc` operator between x and x'. The following proposition details how the latency of a chain is computed for a given instance.

Proposition 2. *Let* $sys(dc_1,\ldots,dc_n)$ *be an abstract system, let* $\mathcal{C}$ *be a functional chain of sys and let* $p = sys[b_1,\ldots,b_n]$ *be an instance of sys. We have*

1. $Lat_p(\mathcal{C}) = \sum_{j=1}^{l} b_{i_j} + Lat_{sys}(\mathcal{C})$*, where the operators* dc_{i_j} *are those involved in* $\mathcal{C}$*,* $l \leq n$ *and for all* i_j*,* $1 \leq i_j \leq n$*.*
2. *If* p' *is another instance such that* $p \sqsubseteq p'$*:* $Lat_p(\mathcal{C}) < k \implies Lat_{p'}(\mathcal{C}) < k$

This proposition allows us to partition the instance space into two sets: the instances satisfying the requirements and those which do not.

Example 9. The flight control system Figure 1 has to satisfy the requirement $R = Lat(z\_acc, o\_z\_acc, d2, dc2, r\_a\_angle, a\_order) < 1$. The partitioning is depicted below.



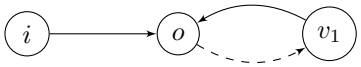
Causality. A program is causal if it does not contain instantaneous data-dependency cycles. A non-causal program must be rejected because we cannot find a computation order that satisfies all data-dependencies. As a consequence, in a concrete program data-dependency cycles are allowed only if they contain a `fby`.

Definition 6. *The functional chain* $\mathcal{C} = (x_1,\ldots,x_n,x_1)$ *is a* dc-cycle *iff:*

$$\exists i, 1 \leq i \leq n, x_i \xrightarrow{?} x_{(i \bmod n)+1}$$

Example 10. The following program contains a cycle where a `dc` is involved:

```
imported node n1 (i1,i2) returns (o);
node causal (i) returns (o)
var v1;
let
  o = n1(i,v1);
  v1 = 0 dc o;
tel
```



(o, v_1, o) is a dc-cycle
causal[0] is not causal,
while causal[1] is.

Definition 7. *Let $sys(dc_1, \ldots, dc_n)$ be an abstract system.*

1. *sys is* strongly causal *if all its instances are causal.*
2. *sys is* weakly causal *if sys admits at least one causal instance.*
3. *sys is* non-causal *if it admits no causal instance.*

Example 11. The program `ex` of example 2 is strongly causal since there is no cycle in the data dependency graph. `causal` of example 10 is weakly causal.

Proposition 3. *Let $sys(dc_1, \ldots, dc_n)$ be an abstract system,*

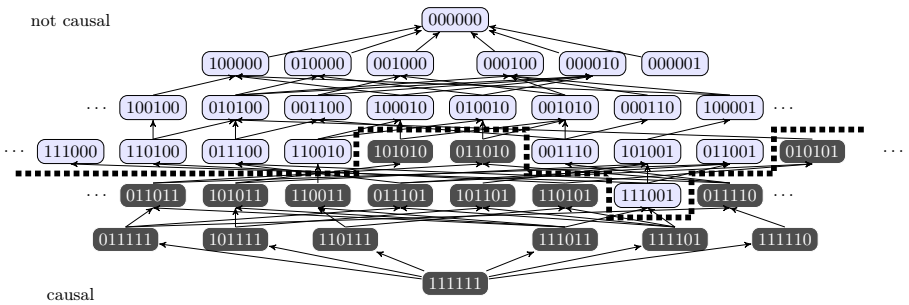
1. *If sys is strongly causal, then sys is also weakly causal.*
2. *sys is strongly causal if and only if $sys[0, \ldots, 0]$ is causal.*
3. *sys is weakly causal if and only if $sys[1, \ldots, 1]$ is causal.*

Proposition 4. *Let $sys(dc_1, \ldots, dc_n)$ be an abstract system. Let p and p' be two instances of sys such that $p \sqsubseteq p'$. Then if p' is causal, so is p.*

Proof. If $p \sqsubseteq p'$, for all equations $io = e$, we have $Var_1^{p'}(e) \subseteq Var_1^{p}(e)$. Thus, the dependency graph of p is a sub graph of the one of p'. If p' is causal, it means that the dependency graph of p' is acyclic and so is any subgraph.

As a consequence, the Hasse diagram of a system can be partitioned into two sets: a "south" partition which contains the causal instances of *sys* and a "north" partition which contains the non causal instances of *sys*.

Example 12. If we consider again the flight control system Figure 1, the graph of instances of this system can be partitioned as shown below. As a result, the set of valid instances of the abstract program is the intersection of the causal instances with the black instances in figure of example 9. There are 4 valid instances: 101111, 101011, 101010, 101101.



If a cycle is a dc-cycle, it is possible to find an instance p where this cycle is broken by replacing one of the `dc` operations in the cycle by a `fby`. If a cycle is not a dc-cycle the program is not causal. The following property shows that causality can be treated as additional latency requirements that must be satisfied by valid instances:

Proposition 5. *Let sys be an abstract system, p be an instance of sys, $\mathcal{C} = (x_1, \ldots, x_n, x_1)$ be a dc-cycle in sys:*

$$\mathcal{C} \text{ is a cycle in } p \Leftrightarrow Lat_p(\mathcal{C}) = 0.$$

3.3 Generation of valid instances

Finding a valid instance p of $sys(dc_1, \ldots, dc_n)$ consists in finding a valuation $sys[b_1, \ldots, b_n]$ which satisfies the latency constraints explicitly specified in the node definition and the constraints generated by the causality analysis. If there exists several solutions, we arbitrarily choose one among them (there is no notion of "better" solution). In this section, we show that we can solve this problem by translating it into a pseudo-Boolean problem, which enables us to reuse existing solvers, such as SAT4J [9] for instance.

Let sys be a system with several latency requirements of the form $L = Lat(\mathcal{C}) \sim k$ (where $\sim \in \{<, >\}$). According to proposition 2, an instance p of sys satisfies L if and only if:

$$\sum_{j=1}^{l} b_{i_j} \sim k - Lat_{sys}(\mathcal{C})$$

where dc_{i_j} are the `dc` operators involved in the chain $(x_{i_1}, \ldots, x_{i_l})$ and $b_{i_j} \in \{0, 1\}$. Thus, the set of latency constraints can be translated into a conjunction of linear pseudo-Boolean constraints of this form. The variables of the pseudo-Boolean problem are the b_{i_j} and $k - Lat_{sys}(\mathcal{C})$ can be considered as a constant as it does not depend on instance p.

The causality analysis can also be translated into pseudo-Boolean constraints. We enumerate the elementary cycles of the form $\mathcal{C} = (x_{i_1}, \ldots, x_{i_m}, x_{i_1})$ of the data-dependency graph of sys using a classic cycle detection algorithm such as that of Tarjan [18]. According to proposition 5, a cycle $\mathcal{C}$ will be avoided (broken) if and only if:

$$\sum_{j=1}^{l} b_{i_j} + Lat_{sys}(x_{i_1}, \ldots, x_{i_m}, x_{i_1}) > 0$$

Example 13. Let us consider the following program:

```
node trad (i) returns (o1,o2)
req (i,v1,v2,v3,o2) < 3;
var v1, v2, v3;
let
  o1 = n1(i,v1);
  v1 = 0 dc o1; --dc1
  v2 = 0 dc v1; --dc2
  v3 = 0 fby v2;
  o2 = n2(v2);
tel
```

The set of constraints is

$$\begin{cases} b1 + b2 + 1 < 3 \\ b1 > 0 \end{cases}$$

3.4 Implementation

To implement the valid instance search, we could proceed as follows:

1. Translate latency constraints into pseudo-Boolean (PB) constraints;
2. For each cycle, add a PB constraint to break the cycle;
3. Solve the whole set of PB constraints.

However, the enumeration of the cycles of a graph is a problem with exponential time-complexity (even though the complexity of Tarjan's algorithm is close to polynomial when the number of cycles is small). Thus, instead we use an iterative process, described in Algorithm 1: we ask the PB-solver for a solution, if the solution is non-causal we add a PB constraint to break a cycle and iterate. With this solution, we do not enumerate all cycles, because breaking one cycle often breaks several other cycles. Experiments suggest that we actually only perform a polynomial number of iterations (even in worst cases), but this remains a conjecture that needs to be proved in future works.

Algorithm 1 Implementation of the valid instance search

```
Translate latency constraints into pseudo-Boolean constraints.
Check the causality for non-dc cycles only[1].
while The PB-solver finds a solution to the constraints do
    if The solution is not-causal then
        Add a PB constraint to break the first cycle we find
    else
        We have found a valid instance, return the solution.
    end if
end while
```

We have implemented an OCaml prototype, which translates a specification with partial delays into a valid synchronous program. The compilation process is shown in Figure 2. The compiler first takes an abstract program and generates the corresponding pseudo-Boolean problem. The problem is then solved using the solver SAT4J [9]. There may be several iterations between the solver and the compiler so that the compiler checks the causality of the solution returned by the solver. Finally, the compiler produces the valid instance corresponding to the final solution returned by the solver.

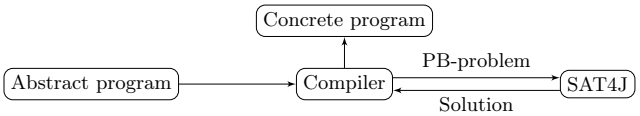
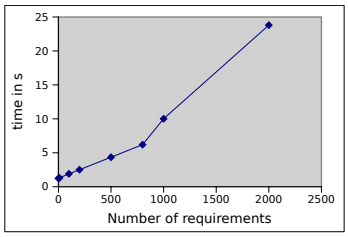


Fig. 2. The compilation process

We have experimented the compiler on several examples and in particular an avionic application similar to the one presented in [2]. It is made up of 3994 imported nodes and 16186 variables. We made the experiments by duplicating all the variables with a "don't care" variable. If we do not specify any latency requirements, the compiler produces the trivial solution where all `dc` are replaced by a `fby`. We then generate some latency requirements to have an overview of the performance of the compiler. The time needed to parse the file is about 1 second. The experiments were made on a Linux machine with 4 GB of memory. We make 10 different benchmarks on each point. These first results are very promising because they show that the method scales well on a representative case study.



4 Conclusion and perspectives

In this paper we have presented an extension of a synchronous data-flow language with a new operator (`dc`), which enables to specify communications that can either be immediate or delayed. We also introduced the possibility to specify latency requirements. We have detailled how a high level specification containing `dc` can be translated into a classic synchronous program without `dc`, taking latency and causality constraints into account.

There are several open questions and future works. First, experiments show that we should ease the specification of latency requirement. For instance we could introduce some kind of pattern matching, e.g. to compute all functional chains matching with `(i,*,v,*,o)`.

Second, we need to study the problem complexity when considering only latency requirements of the form $Lat(\mathcal{C}) \leq k$. In that case, we have a conjection of slighly less general constraints of the form $\Sigma a_j \leq k$ and $\Sigma a_j > 0$.

The third perspective concerns the extension with sampling operators. If we define the exact latency of a functionnal chain $(x_1, \ldots, x_m)$ as the maximum number of reactions before the value of x_m actually depends on x_1, then sampling operators have an impact on this latency. Indeed, applying sampling operators (such as the `when`/`current` operators of Lustre) on a delayed flow can increase the delay of the flow, thus increasing the latency of the chains it belongs to. The effect of general Boolean sampling operators on latencies is hard to analyse: latencies may fluctuate with the sampling condition and even be unpredictable if the condition is an input flow. Instead, we could focus on more restrictive sampling operators that can be analysed more efficiently, such as the *strictly*

periodic clock sampling operators of PRELUDE [16]. On the contrary to Boolean sampling operators, latencies should be constant and computable fully statically.

Acknowledgment

The authors would like to thank Jean-Louis Colaço for his careful analysis and help for improving this work.

References

1. Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
2. Frédéric Boniol, Pierre-Emmanuel Hladik, Claire Pagetti, Frédéric Aspro, and Victor Jégu. A framework for distributing real-time functions. In *6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2008)*, volume 5215 of *Lecture Notes in Computer Science*, pages 155–169, 2008.
3. Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in real-time systems* . John Wiley & Sons, octobre 2002.
4. Francois-Xavier Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference (ERTS'2008)*, 2008.
5. Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software Engineering Institute, Carnegie Mellon University, February 2006.
6. Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Software Eng.*, 18(9):785–793, 1992.
7. Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*, Passau, Germany, 1991.
8. Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.
9. D. Le Berre and A. Parrain. The sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
10. Charles.E. Leiserson and James .B. Saxe. Optimizing synchronous systems. In *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*, pages 23–36. IEEE, 1981.
11. Frédéric Mallet, Julien DeAntoni, Charles André, and Robert de Simone. The clock constraint specification language for building timed causality models. *Innovations in Systems and Software Engineering*, 6:99–106, 2010. 10.1007/s11334-009-0109-0.
12. Florence Maraninchi and Nicolas Halbwachs. Compositional semantics of non-deterministic synchronous languages. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 235–249, London, UK, 1996. Springer-Verlag.
13. OMG. A UML profile for MARTE. Technical report, Object Management Group, Inc, 2007.

14. OMG. Systems modeling language. Technical report, Object Management Group, Inc, 2010.
15. Timothy W. O'Neil, Tongsima Sisades, and Sha Edwin H.-M. Optimal scheduling of data-flow graphs using extended retiming. In *Proceedings of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*, pages 292–297, 1999.
16. Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.
17. Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, 2006.
18. Robert Endre Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973.