Polyhedral Scheduling and Relaxation of Synchronous Reactive Systems

Guillaume Iooss
Inria
France

Albert Cohen
Google[1]
France

Dumitru Potop-Butucaru
Inria
France

Marc Pouzet
École normale Supérieure
France

Vincent Bregeon
Jean Souyris
Airbus
France

Philippe Baufreton
Safran
France

Abstract

The design and implementation of reactive, hard real-time systems involves modeling and generating efficient code for the integration of harmonic multi-periodic tasks. The simple principles of synchronous reactive programming met great scientific and engineering success in the area. A synchronous program orchestrates concurrent computations. It does so while maintaining composability, modularity, functional determinism and real-time execution guarantees. In the case of hard real-time systems, a reactive control program is composed of multi-periodic tasks related through integral ratios. This paper presents a language and optimizing compiler to implement large reactive control systems composed of multi-periodic, synchronous reactive tasks. The same language is used to program a complete control system, from the finest-grained computations to task-level integration, while generating efficient code satisfying real-time constraints. It is evaluated on two real-world applications.

1 Introduction

Dataflow synchronous languages—such as Lustre [7], Signal [2] and others—provide a simple and rigorous model to implement and validate concurrent computations and state machines on streams of data. These computations and reactions are defined by dataflow equations and organized according to their associated clock, which encodes their activation condition. Operationally, synchronous clocks are streams of booleans that determine if a computation or reaction has to be performed at a given logical time step. At compilation time, clocks define a type system to reason about and to generate efficient code. Clocks implement a logical model of time, but they can also be associated with real-time behavior for applications where the temporal aspect is vital.

In the most general setting, clocks define a very abstract type system carrying little information besides the name of the parent clock they derive from, the name of the associated boolean condition, and mutual exclusion with other clocks deriving from the same parent [7]. Many works considered other classes of clocks with richer properties, and used these to improve code generation or to enable more advanced verification schemes. For example, the N-synchronous model [8] considers ultimately periodic clocks and uses these to relax the synchronous composition hypothesis while determining the maximal size of communication buffers implementing such relaxed compositions.

We study the programming of large applications certified at the highest levels of safety-criticality. In this context, a so-called *integration program* orchestrates the communication and activation of a large number of tasks, implemented as stateful nodes. We study the case in which these nodes are scheduled over multiple harmonic periods (i.e., periods that are integral multiples from each other) and are activated only once over each period.

Synchronous languages are well suited to the implementation of individual tasks, but they are not generally adopted for implementing the integration program. Indeed, there is a mismatch between what an engineer wishes to specify and what synchronous languages require. In particular, a programmer might not know when a task should be executed relatively to another, or how two tasks on different periods should communicate. Forcing the programmer to fix these details might induce bad decisions, which can degrade the program performance or violate a real-time constraint. Clearly, there is room for a more declarative approach, with the compiler in charge of filling an incomplete schedule.

Contributions: We introduce 1-synchronous clocks which activate periodically and exactly once during their period. A single program may compose nodes with multiple periodic activation rates. Such clocks model the integration of a real-time system from individual synchronous nodes: the integration program orchestrates the whole system from the top-level, following harmonic rates. Individual nodes may trigger finer-grained computations and reactions by subsampling their parent clocks with arbitrary boolean conditions (i.e. not necessarily periodic). Our language extension abstracts away the verbose constructions needed to express

[1]Most of the work was done while at Inria and ENS.

IMPACT 2022, January 2022, Budapest, Hungary
2022.

such integration code in Lustre, while also enabling the generation of efficient code (essentially free of control flow).

However, this extension alone does not suffice to cope with large applications. Writing a program with 1-synchronous clocks involves explicit phase computations, scheduling short-period tasks over the hyperperiod (i.e. the least common multiplier of all periods of the program). For the clock inference and verification to succeed on the integration program, one needs to set the phase of all local variables and to verbosely specify clock conversion operations when communicating data across phases. This may be tedious on real-world systems integrating thousands of tasks and real-time load-balancing across phases.

To address this challenge, our second contribution offers to partially specify when local variables are being evaluated, by only specifying the period and not the phase of the computation. This program does not have a well-defined synchronous semantics but still implements a Kahn network with a well-defined functional semantics. We present a compilation flow to determine the value of the under-specified phases, resulting in a "classical" Lustre program with fully-specified 1-synchronous clocks.

This flow is composed of a clock inference algorithm which gathers the constraints on the unknown phases, a solver which figures out a valid set of phases, and a pass to reinject these phases inside the integration program. We evaluate our approach on a large real-world avionic application. The solver has to pick a single solution out of the many valid solutions. In order to orient this selection, we also consider two optional variants of a load-balancing cost function, in order to balance the computational work across all phases.

To further align with the needs of control automation engineers, our third contribution adds *underspecified temporal operators*, i.e. delay operators where the values themselves are relaxed and left to the compiler to decide. This "approximate computing" extension is useful when the user does not care about which precise data is operated upon, e.g. in physical measurements with robustness of the temporal variability, as long as some real-time constraints are enforced on the time range of the data being sampled. We can use this extra information to relax the constraints on phases. These extensions are integrated in the open-source synchronous compiler Heptagon [1].

Outline: Section 2 introduces background material about synchronous languages. Section 3 presents our 1-synchronous language extension. Section 4 presents the supporting code generation algorithms. Section 5 further extends the language and compiler with underspecified operators. Section 6 evaluates the language expressiveness and code generation algorithms on two large applications. We review related work in Section 7, before concluding in Section 8.

[1] https://gitlab.inria.fr/synchrone/heptagon/tree/assume/onesync

2 Background and Motivation

Let us recall concepts and syntax about synchronous languages, and further detail our motivations and contributions.

2.1 Syntax

We consider a simplified version of a synchronous language (inspired by [3]), with the following syntax illustrated on the example in Figure 1.

$$
\begin{array}{lcl}
node & ::= & \texttt{node}\ f(lvardecl)\ \texttt{returns}\ (lvardecl) \\
 & & \quad \texttt{var}\ lvardecl\ \texttt{let}\ leq \\
lvardecl & ::= & vardecl;\ lvardecl \mid \\
vardecl & ::= & id : type(::ck)? \\
ck & ::= & . \mid ck\ \texttt{on}\ x \\
leq & ::= & eq \mid eq; leq \\
eq & ::= & pat = exp \\
exp & ::= & cst \mid id \mid op(exp, \ldots, exp) \\
 & & \mid \texttt{if}\ id\ \texttt{then}\ exp\ \texttt{else}\ exp \mid c\ \texttt{fby}\ exp \\
 & & \mid exp\ \texttt{when}\ id \mid \texttt{merge}\ id\ exp\ exp
\end{array}
$$

```
node dummy(i : int) returns (o : int)
var
     half1 : bool :: .; half2 : bool :: .;
     temp : int :: . on half1;
let
     half1 = true fby false fby half1;
     half2 = false fby true fby half2;
     temp = 2 * (i when half1);
     o = merge half1 temp ((0 fby o) when half2);
```

Figure 1. Example of synchronous program.

Each time the dummy node activates, it reads from a variable i and writes to a variable o. These variables produce (resp. consume) one new integral data per tick of the clock of the node. In order to compute the value of o for a given tick, we declare 3 local variables: half1 and half2 which are boolean variables on the base clock (.), and temp which is an integer whose clock is . on half1, i.e., which is present only when the value of the variable half1 is true.

The local variable half1 is computed by the first definition: its value at the very first tick is true, then it is the value of (false fby half1) from the previous tick. The value of this fby expression is false at the first tick, then the value of half1 of the previous tick (it is a delay with initialization). So, the two first values of half1 are true, then false, then the value of itself 2 ticks in the past. Therefore, half1's value is true for all even ticks, and false for all odd ticks. Likewise, half2 is computed by the second definition and has the opposite boolean values from half1. The local variable temp is computed by the third definition: its value is twice the value of i. Because i is always present and temp is only present during the even ticks, we need to subsample i to keep only the values from the even ticks (i when half1). Finally, the output variable o is computed by the last definition. Because o is present more often than temp, we

need to oversample `temp` by distinguishing the cases when `temp` is present or not (depending on the value of the first argument `half1`). Its value is either the value of `temp` when it is present (`half1`=true), or its previous value (`0 fby o`) when it is not the case (`half1`=false, i.e., `half2`=true).

In the following and as is customary in the field, we refer to definitions as *equations* to emphasize the equational, value semantics of the language.

Clocks and periodicity In a general Lustre program, a clock can be either the base clock (`.`), or a sub-clock of another clock (`ck on x`), according to a sampling defined by a boolean variable `x`. The *clocking analysis* associates a clock to every sub-expression, and ensure that they match on both side of an equation, or on all the argument of a function. We can accelerate (resp. decelerate) the clock of an expression by using the operator `when` (resp. `merge`).

Notice that boolean variables occuring in clocks can have values which may not be statically predictable (e.g., they may be an input). This is not the case in our example: we have built the variable `half1` to define a clock which activates once every 2 ticks. Its values form a pattern which repeats itself indefinitely, with a period of 2. Because of this repetition, we says that the clock `. on half1` is periodic, and we note its activation value (10). Likewise, we can note the value of `half1` as (TF), where T is true and F is false.

This notation is coming from the N-synchronous model [8, 19, 21]. In general, infinite binary words ($w := 0w|1w$) can be used to describe the behavior of a clock, where a 1 corresponds to an activation of the clock. An *ultimately periodic binary word* $u(v)$, where u and v are two finite binary words, is a word built of a prefix u before infinitely repeating v. An *ultimately periodic clock* is a clock associated to a ultimately periodic binary word. In this paper, we will only consider *periodic clocks* (v), which are clocks associated to a periodic binary word, i.e. an ultimately periodic binary word with no prefix.

The `current` operator In our example, we produce `o` by oversampling `temp` and by using the `merge` and the `fby` operators. As a syntactic sugar, we introduce an operator `current(x, c, exp)` which oversamples `exp` according to the boolean variable `x`, with the initial value `c`. This initial value is only used if `x` is false at its first tick. During compilation, we can remove the `current` expression by substituting it with a fresh variable `v`, itself defined by a new `merge` expression. The substitution is:

```
current(x, c, e) ⇝ v
   where v = merge x e ((c fby v) whenot x);
```

Semantics We consider two semantics for synchronous programs. The first one is the Kahn semantics [12], keeping track of the sequence of values produced by any stream operator; any synchronization or timing aspect is ignored (i.e., when such values are produced). The second alternative is the synchronous semantics [14, 22], for which both the values and the tick when these values are produced are considered. The later semantics is more strict than the former.

2.2 The need for 1-synchronous constructions

The program in Figure 1 performs a multiplication once every two time ticks, resorting to subsampling and oversampling to fit the input and output rates.

The clocks used in this example are simple to predict due to their periodicity. However, their definition can become heavy to write manually, in particular when the period is large (16 on one of our production use cases). Many clocks with different phases coexist for a given activation period, each one defined by its own boolean equation and conditions. Therefore, it is interesting to provide constructions exploiting the specificity of these clocks and to automate their construction. This is the purpose of our first language extension (Section 3, illustrated on our example in Figure 2).

```
node dummy(i : int) returns (o : int)
var
     temp : int :: [0,2];
let
     temp = 2 * (i when [0,2]);
     o = current([0,2], 0, temp);
```

Figure 2. Example of Fig. 1, using the extension of Section 3.

Second, notice that a clock encodes when a computation should be performed. It can be viewed as scheduling metadata that needs to be provided by the programmer. For example, the computation defining `temp` takes place every even clock ticks. Managing these clocks by hand becomes verbose and combinatorial in presence of real-time constraints, with real-world applications scaling to thousands of equations, half a dozen harmonic periods, and hundreds of non-periodic activation conditions. This motivates the decoupling of this scheduling aspect from the equational specification of the computation, relying on the compiler to determine valid clocks satisfying timing requirements. This is the main goal of our second extension (Section 4).

3 1-synchronous clocks

This section presents our first language extension with constructs for 1-synchronous clocks.

3.1 1-synchronous clocks and language extension

Definition 1. *A* 1-synchronous clock *is a periodic clock with only one activation per period. It is of the form* $(0^k 10^{n-k-1})$*, or alternatively* $0^k(10^{n-1})$*, where* $0 \leq k < n$*; n is called the* period *and k the* phase.

We create a special kind of node, called *model node*, in which the clock calculus is restricted to 1-synchronous clocks

using *harmonic periods* (i.e., periods which are integral multiple of each other). Even if the expressiveness power is reduced on these nodes, this is enough to express integration programs at the top level of real-time system design and implementation. Notice that a model node can use other sub-nodes, corresponding to tasks of the system and which may still use more general Lustre clocks. This restriction also allows us to simplify greatly the key concepts introduced by the N-synchronous formalism. We add the following constructions to the syntax defined in Section 2:

$$
\begin{array}{ll}
model ::= & \texttt{model } f(lvd\_m) \texttt{ returns } (lvd\_m) \\
 & \quad \texttt{var } lvd\_m \texttt{ let } leq\_m \\
lvd\_m ::= & vd\_m\texttt{;}\ lvd\_m \mid \\
vd\_m ::= & x : type :: oneck \\
oneck ::= & [ph, per] \\
leq\_m ::= & eq\_m \mid eq\_m; leq\_m \\
eq\_m ::= & pat = exp\_m \\
exp\_m ::= & c \mid x \mid op(exp\_m, \ldots, exp\_m) \mid c \texttt{ fby } exp\_m \\
 & \mid exp\_m \texttt{ when } [k, ratio] \\
 & \mid \texttt{current}([k, ratio], c, exp\_m) \\
 & \mid \texttt{delay}(d)\ exp\_m \mid c \texttt{ delayfby}(d)\ exp\_m
\end{array}
$$

There are some differences, marked in red, compared to the syntax of a classical Lustre node. First, all variable declarations must have a declared one-synchronous clock (*oneck*). The first argument *ph* is the phase and the second argument *per* is the period (where $0 \leq ph < per$). We also impose that all inputs and outputs are on the base clock `[0,1]`.

The other difference occurs when defining a model node (*exp_m*). First, we specialize the `when` and the `current` expressions to one-synchronous clocks (Figure 3). Instead of specifying a boolean value, we provide two integers k and *ratio*. The latter is the integral ratio between the two periods (which must be harmonic). For the `when` expression, because the fast period executes *ratio* times during one slow period, k specifies which one of these instance is sub-sampled ($0 \leq k < ratio$). For the `current` expression, the fast period repeats the last slow value and k specifies for which of these fast values the update occurs ($0 \leq k < ratio$).

Second, the `merge` expression is not available for these nodes. Indeed, if we try to use it to link two periods whose ratio is more than 2, then the last branch of the merge is activated more than once per period, thus is not 1-synchronous.

Finally, we have added the `delay` and `delayfby` expressions. These expressions do not change the period of an expression, but increases the phase by a given integral value d. The `delay` expression cannot increase the phase of an expression above its associated period, because we would enter into the next period with no value for the current one, thus we would not have a strictly periodic clock. Instead, we can use the `delayfby` with an initial value provided on its left and which must cross a period. Also, notice that a `fby` does not change the clock of an expression, thus it can be seen as a particular case of a `delayfby`.

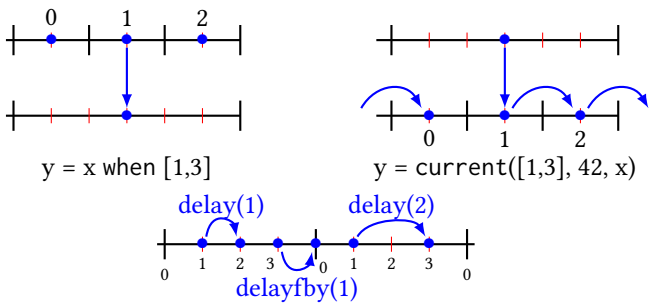
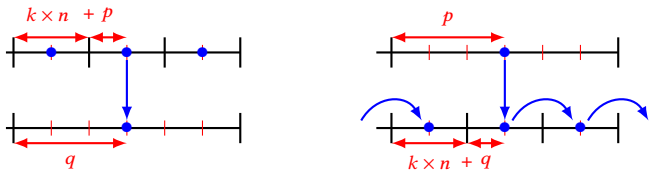


Figure 3. Graphical representation of `when`, `current`, `delay`, `delayfby`, for 1-synchronous clocks. The left and right parts have periods 2 and 6 respectively; bullets are labeled by the value of k to select this instance. In the bottom part, the period is 4 and the ticks are labeled by their phase.



(a) y = x when [k,r] (b) y = current([k,r], 0, x)

Figure 4. Typing rule of when and current operator specialized to 1-synchronous clocks.

```
model main() returns ()
var
   a :: [0,1];   b :: [1,2];
   c :: [2,6];   d :: [0,2];
   e :: [0,1];
let
   b = f1(a when [1,2]);
   c = delay(1) f2(b when [1,3]);
   d = f3(current([2,3], 0,
            delay(2) c) );
   e = f4(current([0,2], 0, d));
```

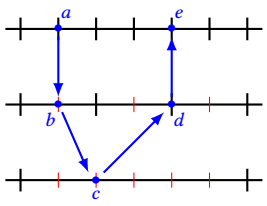


Figure 5. Fully specified 1-synchronous program (all phases are known). Black lines mark period boundaries (and a phase value of 0). Red lines materialize phases within a period. Blue arrows highlight sub/over-sampling chains across variables.

3.2 Clocking analysis for 1-synchronous operators

Same period operators Let us consider first the `delay` and the `delayfby` expressions. As described previously, these operators delay the phase of the clock by a constant value d. Their clocking rules are:

$$\frac{H \vdash x :: [k, n] \quad 0 \leq d < n - k}{H \vdash \text{delay(d) } x :: [k + d, n]}$$

$$\frac{H \vdash x :: [k, n] \quad H \vdash i :: [k + d - n, n] \quad 0 \leq k + d - n < n}{H \vdash \text{i delayfby(d) x} :: [k + d - n, n]}$$

Fast to slow period Let us consider a `when` expression of period m, whose subexpression is of period n. Because of the harmonicity hypothesis, we have an integer r such that $m = n.r$. The clocking rule for this operator is:

$$\frac{H \vdash x :: [p, n] \quad m = n.r}{H \vdash x \text{ when } [k, r] :: [k.n + p, m]}$$

Figure 4a relates the phases of the two clocks. The phase $q = k.n+p$ of the `when` expression is the sum of (i) the number of phases from the periods of the previous non-sampled instances ($k.n$), and (ii) of the phase of the subexpression (p). Notice the relation between p and q is linear.

Slow to fast period Let us consider a `current` expression of period n, whose subexpression is of period m. Again, because of the harmonicity hypothesis, we have a ratio r such that $m = n.r$. The clocking rule for this operator is:

$$\frac{H \vdash x :: [p, m] \quad H \vdash i :: [q, n] \quad m = n.r \quad q = p - k.n}{H \vdash \text{current}([k, r], i, x) :: [q, n]}$$

Figure 4b explains the relation between the phases of the two clocks. The phase p of the subexpression is the sum of (i) the number of phases from the periods of the previous repeated instances ($k.n$), and (ii) of the phase of the `current` expression (q). Again, the relation between p and q is linear.

Example Figure 5 presents an example of a model node, spawning over 3 periods (1, 2 and 6).

3.3 Code generation

We have several options to generate imperative code from a 1-synchronous program. We can either translate the model nodes back into classical Lustre node, and plug ourselves into the classical Lustre compilation flow. Alternatively, we can have a specialized code generator strategy, using the exposed properties of the clocks.

Given a 1-synchronous node, we can lower it into a classical Lustre node. This requires several preprocessing steps, including: (i) converting `delay` expressions into an equivalent combination of nested `when` and `current` expressions; (ii) logging all the different clock used; (iii) building a corresponding variable (similar to `half1` variable in Figure 1); and (iv) replacing the 1-synchronous `when` and `current` expressions by classical ones, using these new variables.

Another option is to have a code generation pass that supports 1-synchronous constructs. The classical Lustre code generation scheme [3] produces a step and a reset function for the compiled node, and can be adapted to our case. However, we have other possibilities. For example, instead of producing a single step function for any phase of the computation, we can produce one specialized step function per phase. This removes all the if statement in the generated code, but increases the size of the generated code. A third option is to unroll the computation and to produce a single step function corresponding to the smallest common multiplier of all periods in the program (called the *hyperperiod*).

4 Phase inference

In this section, we study the possibility of under-specifying phases, to dramatically improve the practicality of the language. We present a language extension and show how to extract constraints on the under-specified phases. This processus is similar in spirit to the step that gather constraints from the dependences, in order to derive an affine schedule, in polyhedral compilation [9]. We study how to solve these constraints in reasonable amount of time for realistic cost functions and scaling to large real-time applications. Once a solution is found, we can transform the program back to a classical 1-synchronous Lustre program.

4.1 Omitting phases of 1-synchronous clocks

In a 1-synchronous program, the phases of the clocks of a program can also be viewed as a coarse-grain schedule. Their values directly affect the quality of the generated code. Moreover, finding a good set of values is a complex problem, which generally depends on other non-functional factors, such as processor architecture and microarchitectural details. However, experimenting with and changing the phase of a node is a heavy operation. Indeed, the phase of a variable also impacts the equation which defines it and all of those that use it. This is a direct consequence of the clocking algorithm enforcing the synchronous composition hypothesis on the program. Also, changing the phase of a variable might introduce, remove or retime a delay operation in many other equations. Therefore, asking a programmer to decide and fix the phases of a program manually is error-prone and unreasonable, especially in the case of large applications and in safety-critical ones.

In order to solve this problem, we introduce the possibility to associate a 1-synchronous clock with an unknown phase with a phase variable. This implies the following modification to the syntax presented in Section 3.1:

$$\begin{array}{lcl} \mathit{oneck} & ::= & [\mathit{ph}, \mathit{per}] \mid [\,.\,., \mathit{per}] \\ \mathit{exp\_m} & ::= & \ldots \mid \texttt{buffer}\ \mathit{exp\_m} \mid c\ \texttt{bufferfby}\ \mathit{exp\_m} \end{array}$$

This new syntax for one-synchronous clocks leaves the phase of a clock unspecified and provides only the period. In terms of dataflow (Kahn) semantics, this is still valid and functionally deterministic: the same sequence of values is computed. However, the synchronous semantics is no longer applicable: the tick of the global clock on which the computation of a given value occurs might change, according to a decision taken internally by the compiler.

Because we have variables and expressions with no precise phase, we cannot set a constant length to a *delay* operator inside a given dataflow equation. Thus, we need to use a

more relaxed *buffer* operator. Thus, we introduce `buffer` and `bufferfby` which are variants of the `delay` and `delayfby` operators where the delay is not specified.

4.2 Clocking analysis and constraint extraction

Symbolic phases Because the clocking rules manipulate phases, which might be unknown, we need to introduce some fresh variables (called *phase variable*) representing the phases of the application. These phases can be manipulated symbolically through the clocking rules, as linear expressions ($var+const$), and might generate equality constraints during unification of clocks (in order to ensure that both sides of an equation or the arguments of an operator have the same clock). In the rest of the paper, we will use the same notation $[p, n]$ for a 1-synchronous clock, whatever the nature of p is (either a constant or a expression of phase variables).

Clocking rule of the new expressions Because the `buffer` operator only delays the clock of an expression of an unknown quantity, (i) both clocks must have the same period and (ii) the phase of the clock of the subexpression must be smaller or equal to the phase of the clock of the `buffer` expression. Its clocking rule is:

$$\frac{H \vdash x :: [p, n] \quad 0 \leq p \leq q < n}{H \vdash \text{buffer } x :: [q, n]}$$

Notice that we now have a constraint $p \leq q$ between the phase p of the subexpression of the buffer and the phase q of the buffer expression. This is the classical dataflow constraint: the producer must happen before the consumer. During the clocking analysis, we gather these constraints in order to check the coherence of the phases or to solve them if there are unknown phases involved.

The clocking rule of a `bufferfby` expression is:

$$\frac{H \vdash x :: [p, n] \quad H \vdash i :: [q, n]}{H \vdash i \text{ bufferfby } x :: [q, n]}$$

Notice that there is no constraint linking p and q: indeed the value of x used is coming from the previous instance of the period, so the `bufferfby` expression does not need to wait for anything. If we wish to share the memory of the old value of x and the new one, we have to ensure that the old value of x is used before the new one is computed. This adds the (optional) constraint $q \leq p$.

Example We consider again the example of 1-synchronous program given in Figure 5, and now assume that all variables do not have a phase specified. The resulting program and extracted constraints are shown in Figure 6. The program uses unspecified clock phase for all of its variables. When deriving the constraints, we associate a new phase variable p_x to each variable x. This phase must be valid, which gives us the first set of constraints (bounds from variable declarations). The equations are changed in the following way: a buffer is added as the top most operator of each equation, allowing to adjust the clocks of the left and right sides of the equation, depending on the choice made for their phase. Each buffer creates a constraint, and all of them are gathered to form the second set of constraints (in our case, one constraint per equation).

```
model main() returns ()
var
  a :: [..,1]; b :: [..,2];
  c :: [..,6]; d :: [..,2]; e :: [..,1];
let
  b = buffer f1(a when [1,2]);
  c = buffer f2(b when [0,3]);
  d = f3(current([2,3], 0, buffer c));
  e = f4(current([0,2], 0, buffer d));
```

Extracted constraints:

- Bounds from variable declarations:
  $0 \leq p_a, p_e < 1$, $0 \leq p_b, p_d < 2$ and $0 \leq p_c < 6$
- Constraints from `buffer` clocking rule:
  $p_a + 1 \leq p_b$, $p_b \leq p_c$, $p_c - 4 \leq p_d$ and $p_d \leq p_e$

Figure 6. Example of 1-synchronous program where all phases are unknown. p_x is the phase of the variable x.

Non-functional constraints We can enrich the syntax of the language defined in Section 4.1 with annotations that encode additional constraints on the phases. In particular, our compilation flow is able to manage (i) minimal or maximal value constraints ($p \leq M$ or $m \leq p$), (ii) precedence constraints ($p_1 \leq p_2$) and (iii) latency constraints over a chain of dependence ($e \leq L$, bounding the number of phases delayed because of the operators of the chain of dependences).

4.3 Constraint resolution and cost function

Form of the constraints The constraints on phases are exposed through clocking analysis. Note that the phase of any expression in the program is either a constant (if the phase is known) or the sum of a phase variable and a constant. This invariant can be proved by induction, using the different clocking rules. We consider 4 kinds of constraints:

- The *boundary conditions* on the phases of the underspecified clocks, which can be extracted from the variable declarations. An annotation imposing a minimal/-maximal value can refine it. These constraints have the form $m \leq p$ or $p \leq M$, p being a phase variable and m, M constants.
- The *dependence constraints* coming from the buffer clocking rules, or precedence annotations. Due to the form of the phases of all sub-expression of the program, there are at most 2 phase variables in these constraints, on opposite side of the inequality and with coefficients of 1. So, these constraints have the form $p_1 + C \leq p_2$, or the form of a boundary condition if one of the compared phase is a constant.

- The *latency constraints* coming from dependence annotations. These constraints can be optionally specified besides the program, and is linked to a dependence chain in the program. It imposes that the number of ticks spent along this dependence chain must be smaller than a constant. If the considered dependence chain has only one edge, the constraint has the same form than a dependence constraint. Else, an arbitrary number of phase variable might occur in these constraints. For example, if we specify the dependency chain of Figure 5, we can impose that the number of ticks between the update of the value of c and the fresh value of a it uses is less or equal to 2 ($p_c - p_a - 1 \leq 2$).
- The *unification constraints*, which are equalities coming from the equations and the application. Because we will always have a phase variable with a coefficient of 1 in this equalities, we can use them as substitution to reduce the number of phase variables of the system of constraints. Therefore, we can safely ignore them.

Resolution of the constraints In the absence of latency constraints, the system is made of difference-bound constraints only. Thus, we can use a Floyd-Warshall algorithm [10] to solve them. We pick by default the solution composed of the minimal feasible value for each phase variables, which corresponds to scheduling every computation at the earliest. If the constraints are more complicated we rely on an Integer Linear Programming (ILP) solver.

Cost functions The selected phase values is a coarse-grain schedule of the application, so selecting an arbitrary solution might have a negative impact on performance. Therefore, we wish to consider other criteria, such as memory allocation or load balancing across the phases.

These criteria are encoded in our problem as a cost function, which we try to optimize through an ILP. In many cases, getting an optimal solution is much more than needed, and just finding a solution which balances the workload well enough to fit a real-time constraint, or save enough memory to fit in a hardware memory, is enough.

WCET load balancing cost function An interesting cost function when mapping the application into a sequential architecture is the WCET load balancing. Each node is associated with a weight (corresponding to its WCET) and the goal is to balance it across all phases of the application, so that it fits in the allocated time for every phase.

The phase p_T of a node T is the phase shared by its output variables. W_T is the weight of the node T (WCET in our case). In order to encode the sum of the weights of the nodes scheduled on a given phase, we need to introduce a dual binary encoding of the integral phase variables. We introduce, for all node T and possible phase k, a binary variable $\delta_{k,T}$ which encodes if a node T is scheduled at phase k. We also introduce a final integral variable W_{max} which is also our cost function to minimize. The list of additional constraints introduced by the load-balancing is:

$$(\forall T) \quad \sum_k \delta_{k,T} = 1 \qquad \text{(unicity of the 1 value)}$$
$$(\forall T) \quad \sum_k k.\delta_{k,T} = p_T \qquad \text{(link integral-binary)}$$
$$(\forall k) \quad \sum_T (\delta_{k \ mod \ period(T),T} \times W_T) \leq W_{max} \qquad \text{(max weight)}$$

Binary WCET load balancing We have a variant of this cost function, called *binary WCET load balancing*, where we only use the binary variables in place of the integral phase variables. This avoids having two sets of variables represent the same information, and it can be solved faster in some situations. The main issue is to translate the constraints on the integral variables (dependencies or latency) into an equivalent set of constraints on binary variables.

When dealing with difference-bound constraints (e.g., all dependence constraints), the translation is straightforward:

$$p \leq q + C \quad \rightsquigarrow \quad (\forall k)\ \delta_{p,k} \leq \sum_{l \geq k+C} \delta_{q,l}$$

In the case of latency constraints with more than 2 phase variables (or of some dependence constraints as presented in the next section), we have at least a phase variable q whose coefficient is 1 that we can isolate from the rest of the constraint:

$$\sum_k c_k.p_k \leq q$$

where the c_k are the integral coefficients and p_k/q are phase variables. We introduce a new integral temporary variable $e = \sum_k c_k.p_k$, and we introduce its binary variables $(\delta_{e,i})_i$ (i.e., $\delta_{e,i} = 1 \Leftrightarrow e = i$). The constraint becomes $e \leq q$, which is translatable using the straight-forward case. We still need to translate the equation $e = \sum_k c_k.p_k$. We consider all combination of values $(v_k)_k$ of the p_ks, and by grouping the combination according to the value V of e it corresponds, we obtain the following constraint using the binary variables:

$$(\forall V)\ \delta_{e,V} = \sum_{\substack{(v_k)_k \\ \sum_k c_k.v_k = V}} \prod_k \delta_{p_k,v_k}$$

These constraints are polynomial, however it is possible to make them affine as follows:

$$(\forall (v_k)_k)\ min_k(\delta_{p_k,v_k}) \leq \delta_{e,V} \text{ , where } V = \sum_k c_k v_k$$

This translation works because we have a single $\delta_{e,k}$ that is equal to 1 (unicity constraint), and there is exactly one combination v_k such that all δ_{p_k,v_k} are also 1. For this combination, the minimum over k will be equal to 1, thus $\delta_{e,V}$ will be forced to be 1. For all the other combinations, their minimum is 0, thus their corresponding $\delta_{e,V}$ is not constrained. This expression can also be writen without the minimum:

$$(\forall (v_k)_k)\ \sum_k \delta_{p_k,v_k} - (card(v_k) - 1) \leq \delta_{e,V} \text{ , where } V = \sum_k c_k v_k$$

For example, let us consider the constraint $p_1 + p_2 \leq p_3$, where $0 \leq p_1 < 2$, $0 \leq p_2 < 2$ and $0 \leq p_3 < 4$. We introduce the new temporary variable $e = p_1 + p_2$. The constraint becomes $e \leq p_3$ which can be translated in the straightforward manner. We still have to translate the equalities, and the constraints on it are:

$$\begin{cases} \delta_{p_1,0}.\delta_{p_2,0} = \delta_{e,0} & (V = 0) \\ \delta_{p_1,1}.\delta_{p_2,0} + \delta_{p_1,0}.\delta_{p_2,1} = \delta_{e,1} & (V = 1) \\ \delta_{p_1,1}.\delta_{p_2,1} = \delta_{e,2} & (V = 2) \end{cases}$$

After transformation, we have the following set of linear constraints on binary variables:

$$\begin{cases} \delta_{p_1,0} + \delta_{p_2,0} - 1 \leq \delta_{e,0} & (V = 0) \\ \delta_{p_1,1} + \delta_{p_2,0} - 1 \leq \delta_{e,1} & (V = 1) \\ \delta_{p_1,0} + \delta_{p_2,1} - 1 \leq \delta_{e,1} & (V = 1) \\ \delta_{p_1,1} + \delta_{p_2,1} - 1 \leq \delta_{e,2} & (V = 2) \end{cases}$$

These constraints are affine and amenable to an ILP solver.

4.4 Substitution of the phase solution

Once a set of valid values for the phases of the program is found, we use these values to obtain a classical 1-synchronous program, in which all phases are known. This is done by completing the clock of all variable declarations and by transforming all `buffer` into `delay` (resp. `bufferfby` into `delayfby`). Given a phase solution $\mathcal{S}$, which is a mapping from a phase variable p_x (of variable x) to a valid value v, the transformation rules of this transformation are:

`x:type::[..,n]` $\rightsquigarrow$ `x : ty :: [v,n]`, where $\mathcal{S}(p_x) = v$

`e::[f(`$p_1, \ldots, p_r$`),n]` $\rightsquigarrow$ `e::[f(`$\mathcal{S}(p_1), \ldots, \mathcal{S}(p_r)$`),n]`, for every expression `e`

`buffer e` $\rightsquigarrow$ `delay(d) e`, where $d = q - p > 0$, (`buffer e`)::[q,n] and e::[p,n]

`buffer e` $\rightsquigarrow$ `e`, where $q = p$, (`buffer e`)::[q,n] and e::[p,n]

`i bufferfby e` $\rightsquigarrow$ `i delayfby(d) e`, where $d = n + q - p$ (`buffer e`)::[q,n] and e::[p,n]

We can verify the validity of the solution while performing this replacement. For example, if the parameter of a `delay` is found to be a negative number, its corresponding dependence constraint was violated by the solution.

5 Underspecified operators

The previous section proposed an extension where the phases of some clocks are not given, but inferred later by the compiler. However, such a formalization is still over-constrained for a wide range of real-time control scenarii where it does not matter which value is being consumed as long as this value is "fresh" enough and the computation takes place early enough. The typical example is that of a sensor sampling a physical value with low temporal variability. Fixing the dataflow arbitrarly restricts the set of valid schedules to be explored, and might invalidate potentially interesting schedules. The goal of this section is to capture this tolerance with "approximate computing" operators whose incoming dependence is underspecified. Then, we use this information to relax the constraints on the phases, and when the phases are set, to derive a fully deterministic 1-synchronous program.

5.1 Underspecified one-synchronous operator

We introduce the underspecified one-synchronous operator through an extension of the syntax first presented in Section 3.1, and extended in Section 4.1:

$$\begin{aligned} exp_m \quad ::= \quad & \ldots \mid c \texttt{ fby? } exp_m \mid c \texttt{ bufferfby? } exp_m \\ & \mid exp_m \texttt{ when? } r \mid \texttt{current?}(r, c, exp_m) \end{aligned}$$

We propose 4 underspecified operators relaxing their one-synchronous counterparts:

- The value of (`c fby? e`) can be either (`c fby e`) or `e`.
- The value of (`c bufferfby? e`) can be either (`c bufferfby e`) or (`buffer e`).
- The value of (`e when? r`) is (`e when [k,r]`), for a $0 \leq k < r$.
- The value of `current?(r, c, e)` is `current([k,r], c, e)`, for a $0 \leq k < r$.

The `when?` operator corresponds to a 1-synchronous `when` operator for which we do not know which value is sampled. Likewise, we do not know when the update of the fast value of a `current?` operator happens. Technically, we could build the operators `when?` (resp. `current?`) as syntactic sugar from the operator `when` (resp. `current`) and a succession of `fby?` operators on the faster period. However, it is more convenient and meaningful to define them as part of the language. The value of all underspecified operators are taken by the compiler, and fixed during the clocking analysis as part of the solution found for the phases.

Notice that, *the very computation performed by the program changes according to this decision.* By using these operators, *the programmer assures the compiler that all possible versions of the program are correct* and lets the compiler pick whichever it prefers.

Figure 7 shows two examples using non-deterministic operators, and one of their corresponding determinized version.The first example is a loop between the variables `x` and `y`, with potentially a delay between them (`fby`), which accesses the value of the previous period, instead of the current one. The proposed determinized version (one of the 3 possibilities) introduces a delay when computing `x`, and use its current value when computing `y`. The second example shows a loop between two variables of different period, without specifying which value is sub/over-sampled. In the determinized version, we chose to sub-sample the first value of `x`, then to over-sample the second value of `y`.

5.2 Clocking analysis and constraint extraction

Symbolic decision variables To represent which value is selected for an underspecified operator, we introduce new

```
model loop1()
   returns (y:int)
let
   x = 0 fby? y;
   y = 1 fby? x;

        ↓

model loop1_det()
   returns (y:int)
let
   x = 0 fby y;
   y = x;
```

(a) First example.

```
model loop2(x:int)
   returns (z:int)
var y:int::[..,2];
let
 y = x when? 2;
 z = current?(2,0,buffer y);
          ↓
model loop2_det(x:int)
   returns (z:int)
var y:int::[0,2];
let
 y = x when [0,2];
 z = current([1,2], 0,
             delay(1) y);
```

(b) Second example.

Figure 7. Examples of underspecified 1-synchronous programs. Notice that the determinized versions are not unique.

integral symbolic variables d , called *decision variables*. Due to these new variables, unknown phases become general affine expressions instead of just linear expressions.

Clocking rule of the* `fby?` *and* `bufferfby?` *expressions Like a `fby` expression, the `fby?` expression has the same clock than its subexpression. We still associate a decision variable $d \in [0, 1]$ to this operator, such that $d = 1$ corresponds to a `fby` and $d = 0$ corresponds to a simple copy.

The clocking rule of a `bufferfby?` expression is:

$$\frac{H \vdash x :: [p, n] \quad H \vdash i :: [q, n] \quad p \leq q + n.d \quad 0 \leq d \leq 1}{H \vdash i \text{ bufferfby? } x :: [q, n]}$$

When the decision variable of `bufferfby?` d is 1, we have a `bufferfby` and when it is 0, we have a `buffer`. Thus, we have to activate the constraint of the *buffer* only when $d = 0$. This is done by using the period n of the expressions, which will always be bigger than p.

Clocking rule of the* `when?` *and* `current?` *expressions The clocking rule of a `when?` expression is:

$$\frac{H \vdash x :: [p, n] \quad m = n.r \quad q = d.n + p \quad 0 \leq d < r}{H \vdash x \text{ when? } r :: [q, m]}$$

This is simply the clocking rule of a 1-deterministic `when` expression, where the index of the sampled value (which was k in Section 3.2) is the new decision variable d.

Likewise, the clocking rule of a `current?` expression is:

$$\frac{H \vdash x :: [p, m] \quad H \vdash i :: [q, n] \quad \begin{array}{c} m = n.r \\ q = p - d.n \end{array} \quad 0 \leq d < r}{H \vdash \text{current?}(r, i, x) :: [q, n]}$$

Again, this is simply the clocking rule of a 1-deterministic `current` expression, where the index of the sampled value (which was k in Section 3.2) is the new decision variable d.

Causality constraints The causality analysis ensures that no variable depends on itself, instead of depending on the values from the previous periods. Because the `fby?` and `bufferfby?` operators might either use a value of the current instance of period or from the previous instance of the period, they interact with this analysis. If we consider the first example of Figure 7, if none of the two `fby?` are transformed into a `fby`, then the resulting program has a causality loop and is not valid. Thus, we wish to add a constraint on the decision variables of the `fby?` and `bufferfby?` expressions to reject these programs. We consider all the cycles in the dependence graph of a model nodes:

- If they have a `fby` or a `bufferfby` expression, then we are sure to use a value from a past period. No constraint is needed for this loop.
- If there is no `fby`/`bufferfby`/`fby?`/`bufferfby?`, we are still on the same period and, because there is only one clock activation per period, the value used is the value produced: we have to reject the program.
- If there is no `fby`/`bufferfby` but there is at least a `fby?`/`bufferfby?`, we have to ensure that at least one of their decision variables is equal to 1. Thus, we generate the constraint $\sum_k d_k \geq 1$, where the d_k are the decision variables of the encountered `fby?`/`bufferfby?`

In the first example of Figure 7, the extracted constraint is $d_1+d_2 \geq 1$, where d_1 is the decision variable of the expression (`0 fby? y`) and d_2 the one of the expression (`1 fby? x`). We can indeed verify that $d_1 = d_2 = 0$, i.e., choosing to place no `fby` operator in the determinized program, introduces a cyclic dependence: a value of `x` requires the current value of `y`, which also requires the current value of `x`. So, this constraint is needed for the validity of the determinized program.

5.3 Solving the constraints

The constraints are slightly more complex compared to the ones encountered in Section 4.3: the coefficient in front of a decision variable might not be 1. However, this does not impact the solving method, and the solution produced contains additional values for the decision variables.

5.4 Determinizing the program

As shown in Section 4.4, once a valid solution is found for the phases, one may use it to return to a classical 1-synchronous Lustre program. Before substituting the values of the phases, we use the decision variables to substitute the underspecified expressions by their chosen value. We assume that the chosen solution $\mathcal{D}$ for the decision variables is a mapping associating a underspecified expression to the value of the decision variable. The rules of this transformation are:

```
          c fby? e ⇝   e   if 𝒟(c fby? e) = 0
          c fby? e ⇝   c fby e   if 𝒟(c fby? e) = 1
  c bufferfby? e ⇝   c buffer e
                       if 𝒟(c bufferfby? e) = 0
  c bufferfby? e ⇝   c bufferfby e
                       if 𝒟(c bufferfby? e) = 1
       x when? r ⇝   x when [k,r]
                       if 𝒟(x when? r) = k
current?(r, c, x) ⇝   current([k,r], c, x)
                       if 𝒟(current?(r, c, x)) = k
```

The program is now a deterministic 1-synchronous program and one may carry on with the rest of the compilation.

6 Experimental evaluation

Description of the use cases We consider two large scale use cases from the avionics domain. Both are production applications, deployed on commercial aircraft and DAL-A certified—the highest level of mission-critical safety—according to the DO-178C standard (implemented by the Federal Aviation Administration and other authorities worldwide). More detail is left out to preserve anonymity.

The first use case (called *UC1*) is the specification of the control commands of an Airbus airplane and is composed of 5124 Lustre nodes (equations) and 32006 Lustre variables. This application has 4 harmonic periods (1, 2, 4 and 12, corresponding to 10, 20, 40 and 120 ms) and each node is activated exactly once per period. The second use case (called *UC2*) is a motor regulation application from Safran and is composed of 142 Lustre nodes. This application has 5 harmonic periods (1, 2, 4, 8 and 16, corresponding to 15, 30, 60, 120 and 240 ms). Each node is activated exactly once per period.

Each node comes with a Worst-Case Execution Time (WCET) evaluated on the processor of reference for the application (e.g. a time-predictable P4-series Freescale processor). There are no latency constraints.

The use cases go through a series of normalization steps to be converted, automatically, into our extension of Lustre.

- First, we recursively inline the application to expose its dependence graph, bringing all equations under the same node. The depth of the recursion controls the granularity of the schedule. We choose to inline all integration nodes made of function calls and data structure manipulation only; computation and sensing/actuation take place in nested Lustre nodes, hidden from the scheduling algorithm.
- Then, to remove false dependences, we lower all temporary arrays into individual scalar elements. For each node performing I/O, we create a corresponding *scalarified node* which internally builds the array, calls the original node, and wraps it with array-to-scalar or scalar-to-array conversion.
- *UC1* requires additional care to encode its periodic activations into 1-synchronous clocks. We create a *sequencer* node from the ad-hoc integration specification used in the production application; it generates 41 boolean Lustre variables controlling the rest of the system and is scheduled on the base clock/period.
- Finally, we perform various normalization steps on the program, such as the removal of copy equations, deadcode elimination or constant propagation.

Scalability of the ILP solving Given a dependence graph, we generate the set of constraints on the phases. There is 2 boundary constraints per phase variable, and one dependence constraint per dependence in the program. Figure 8 shows the time spent by the Cplex solver to schedule the 3 variants of the ILP formulation of the 2 use cases. The reported number of lines roughtly corresponds to the number of dependence constraints, plus the number of variables in the ILP problem.

We can draw two conclusions: (i) it is possible to find a solution in a reasonable time (less than a minute) for a large real-world application; (ii) the variants for the load balancing function are in the same ballpark. About the precision of the solution found by the ILP, the solution found by the rational relaxation is reached for UC2. For UC1, the integral solution found is about 0.01% above the optimal found by the rational relaxation, with a gap of about 400 cycles (over about 5.10^6 cycles).

ILP problem		# of lines	# of vars	Solving time
UC1	No cost function	28742	5124	0.05 s
	Load balancing integral	79128	45249	17.75 s
	Load balancing binary	200746	40125	9.05 s
UC2	No cost function	3908	143	0.01 s
	Load balancing integral	5106	1039	0.03 s
	Load balancing binary	17689	897	0.05 s

Figure 8. Time spent to find a valid phase for all nodes on the 2 use cases and 3 cost functions.

7 Related work

The problem we are dealing with belongs to a long tradition of resource allocation and scheduling approaches [6]. The originality of our problem and solution reside on the programming language and compiler side. Optimizing harmonic multi-periodic schedules for load balancing purposes is also reminiscent of energy minimization problems [1], the main difference being that the period is taken as a constraint rather than an objective in our context. We focus on the programming language side of the related work.

1-synchronous clocks: Smarandache et al. [25] studied affine clocks which are similar to our 1-synchronous clocks, in the context of the Signal language. The formalization does not set bounds on the phase, which means that clocks may have no activations during the first periods at the start of the computation. The authors investigated the clock unification problem, across different periods and phases.

Plateau and Mandel [18, 21] studied the possibility of not specifying the phase of some clocks, in the context of the N-synchronous model [8]. In [18], the authors extract the constraints on the activation of a general N-synchronous clock in the Lucy-n language [19]. While their formalization generalizes our work, they did not study the scalability of the extracted constraints on large applications, nor did they model real-time load balancing or efficient code generation.

Forget, Pagetti et al. [11, 20] introduced Prelude, an equational language aimed at multi-periodic integration program. It proposes three operators to manipulate clocks: `/.k` (corresponding to a `when` which samples the first occurrence), `*.k` (corresponding to a `current` which updates on the first occurrence) and $\rightarrow$ (corresponding to a `delay` and used to orchestrate phases). Some real-time information are encoded in the language, and the nodes are associated with a WCET.

In term of scheduling, they adopt the relaxed synchronous hypothesis: the execution of a node must end before the next tick of its clock, and not the next tick of the base clock. Thus, the classical step function code generation strategy, where a step function corresponds to a tick of the base clock, is not applicable. Therefore, the earliest paper on Prelude assumed a preemptive operating systems. However, following papers [5] considered the generation of parallel code, allowing bare metal implementations.

Henzinger et al. [15] introduced Giotto, a time-triggered language for multi-periodic integration programs. In their model, drivers are instantaneous computations which are used to communicate between periodic tasks. The periods of those tasks are not required to be harmonic. A driver can be guarded and transmit the last computed value when it is not activated. They support modes, which can periodically replace the set of tasks of the application with another.

Underspecified operators: In Simulink, the rate transition block [24] allows communication between different periods. It can communicate from a fast period to a slow one, or the opposite, which correspond to the `when` and `current` operators. It also has the choice between being deterministic while introducing extra communication delay, or nondeterministic but with a better latency. Wyss [29] introduced the "don't care" operator (`dc`), which corresponds to our `fby?` operator. He defines the semantic of this operator and studies its impact on the latency requirements, and on the causality analysis and generates a set of pseudo-boolean constraints which are similar to our causality constraints in Section 5.2. Maraninchi and Graillat [13] have considered bounded nondeterministic delays to deal with the variable communication latency over computation clusters of the Kalray MPPA. They describe a fundamentally non-deterministic behavior, while our usage of the operator represents a collection of functionally deterministic programs among which the compiler can pick to optimize the load of a real-time implementation.

Synchronous Data-Flow (SDF) and Cyclo-Static Data-Flow (CSDF) graphs [4, 17] are abstract models of periodic behavior in systems. They are closely related to real-time scheduling theory and deployed in design, synthesis and simulation tools [16, 23, 26]. [28] has been the prominent programming language incarnation of the model, and led to a characterization of its expressiveness for computational and signal-processing applications [27]. Yet SDF and derivatives do not enable modular composition and have limited ability to model non-periodic control systems. Also, while bounded scheduling problems are very naturally expressed on SDF, we are not aware of any experiment with a real-world application scaling to the size of our use cases.

8 Conclusion

Multi-periodic clocks with a single activation allow to relax the synchronous composition hypothesis by not specifying the phase of some clocks. Dataflow and real-time constraints on these phases are collected and solved at the top level of the integration program. We also introduced underspecified operators whose execution may be delayed to later cycles, further relaxing the constraints on activation phases. This increases the space of valid schedules while exploiting the slack naturally occurring in control automation. The program is fully deterministic once the phases are computed. Finally, we presented cost functions and algorithms to solve these constraints, and evaluated their scalability on two large, safety-critical avionic applications.

Acknowledgments: This work has been realized as part of the project ITEA ASSUME, then as part of the collaboration "All-In-Lustre" between Inria and Airbus. We would also like to thank Paul Feautrier for very insightful discussions about the solvers used in this work and about the ILP encoding of the problem.

References

[1] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. *ACM Trans. Embed. Comput. Syst.* 15, 1, Article 7 (Jan. 2016), 34 pages. https://doi.org/10.1145/2808231

[2] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming* 16, 2 (1991), 103–149. https://doi.org/10.1016/0167-6423(91)90001-E

[3] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. 2008. Clock-directed Modular Code Generation of Synchronous Dataflow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, Tucson, Arizona.

[4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. 1995. Cyclo-static data flow. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, Vol. 5. 3255–3258 vol.5. https://doi.org/10.1109/ICASSP.1995.479579

[5] Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. 2012. Deterministic Execution Model on COTS Hardware. In *Architecture of Computing Systems (ARCS)*, Andreas Herkersdorf, Kay Römer, and Uwe Brinkschulte (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 98–110.

[6] Peter Brucker and Thomas Kampmeyer. 2008. A general model for cyclic machine scheduling problems. *Discrete Applied Mathematics* 156, 13 (2008), 2561–2572. https://doi.org/10.1016/j.dam.2008.03.029 Fifth International Conference on Graphs and Optimization.

[7] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'87)*. ACM, Munich, West Germany, 178–188. https://doi.org/10.1145/41625.41641

[8] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. 2006. N-synchronous Kahn Networks: A Relaxed Model of Synchrony for Real-time Systems. *SIGPLAN Notices* 41, 1 (Jan. 2006), 180–193. https://doi.org/10.1145/1111320.1111054

[9] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem: I. One-dimensional Time. *International Journal of Parallel Programming* 21, 5 (Oct. 1992), 313–348. https://doi.org/10.1007/BF01407835

[10] Robert W. Floyd. 1962. Algorithm 97: Shortest Path. *Communication of ACM* 5, 6 (June 1962), 345. https://doi.org/10.1145/367766.368168

[11] Julien Forget. 2009. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. Ph.D. Dissertation. Institut Supérieur de l'Aéronautique et de l'Espace. https://hal.archives-ouvertes.fr/tel-01942421

[12] Kahn Gilles. 1974. The semantics of a simple language for parallel programming. *Information Processing* (Aug 1974), 471–475.

[13] Amaury Graillat. 2018. *Code Generation for Multi-Core Processor with Hard Real-Time Constraints*. Ph.D. Dissertation. Université Grenoble Alpes. https://tel.archives-ouvertes.fr/tel-02069346

[14] Grégoire Hamon. 2002. *Calcul d'horloges et structures de contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML*. Ph.D. Dissertation. Informatique Paris 6.

[15] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. 2001. Giotto: A Time-triggered Language for Embedded Programming. In *Embedded Software*, Thomas A. Henzinger and Christoph M. Kirsch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–184.

[16] A. Honorat, H. N. Tran, L. Besnard, T. Gautier, J.-P. Talpin, and A. Bouakaz. 2017. ADFG: A Scheduling Synthesis Tool for Dataflow Graphs in Real-Time Systems. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems (RTNS '17)*. Association for Computing Machinery, New York, NY, USA, 158–167. https://doi.org/10.1145/3139258.3139267

[17] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245. https://doi.org/10.1109/PROC.1987.13876

[18] Louis Mandel and Florence Plateau. 2012. Scheduling and Buffer Sizing of n-Synchronous Systems. In *Mathematics of Program Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 74–101.

[19] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. *Lucy-n: a n-Synchronous Extension of Lustre*. Springer Berlin Heidelberg, Berlin, Heidelberg, 288–309. https://doi.org/10.1007/978-3-642-13321-3_17

[20] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. 2011. Multi-task Implementation of Multi-periodic Synchronous Programs. *Discrete Event Dynamic Systems* 21, 3 (Sept. 2011), 307–338. https://doi.org/10.1007/s10626-011-0107-x

[21] Florence Plateau. 2010. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. Ph.D. Dissertation. Université Paris XI. http://www.theses.fr/2010PA112080

[22] Marc Pouzet. 2002. *Lucid Synchrone: un langage synchrone d'ordre supérieur*. Ph.D. Dissertation. Université Pierre et Marie Curie, Paris, France.

[23] Claudius Ptolemaeus (Ed.). 2014. *System Design, Modeling, and Simulation Using Ptolemy II*. https://ptolemy.berkeley.edu/books/Systems.

[24] rate [n.d.]. Rate Transition Simulink Documentation. https://www.mathworks.com/help/simulink/slref/ratetransition.html. Accessed: 2020-02-07.

[25] Irina Madalina Smarandache. 1998. *Transformations affines d'horloges : application au codesign de systemes temps-reel en utilisant les langages Signal et Alpha*. Ph.D. Dissertation. Université Rennes 1. http://www.theses.fr/1998REN10085

[26] S. Stuijk, M.C.W. Geilen, and T. Basten. 2006. SDF^3: SDF For Free. In *Application of Concurrency to System Design, 6th International Conference, ACSD 2006, Proceedings*. IEEE Computer Society Press, Los Alamitos, CA, USA, 276–278. https://doi.org/10.1109/ACSD.2006.23

[27] William Thies and Saman Amarasinghe. 2010. An Empirical Characterization of Stream Programs and its Implications for Language and Compiler Design. In *International Conference on Parallel Architectures and Compilation Techniques*. Vienna, Austria. http://groups.csail.mit.edu/commit/papers/2010/thies-pact10.pdf

[28] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*. Grenoble, France. http://groups.csail.mit.edu/commit/papers/02/streamit-cc.pdf

[29] Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. 2012. A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming. In *10th Asian Symposium on Programming Languages and Systems*. Kyoto, Japan, 223–238. https://hal.archives-ouvertes.fr/hal-00800975