Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints

Timothy Bourke, Vincent Bregeon, Marc Pouzet

Scheduling and Compiling Rate-Synchronous Programs with End-To-End Latency Constraints

Timothy Bourke
Inria Paris, France
Ecole normale supérieure, PSL University, CNRS, Paris, France

Vincent Bregeon
Airbus Operations S.A.S., Toulouse, France

Marc Pouzet
Ecole normale supérieure, PSL University, CNRS, Paris, France
Inria Paris, France

Abstract

We present an extension of the synchronous-reactive model for specifying multi-rate systems. A set of periodically executed components and their communication dependencies are expressed in a Lustre-like programming language with features for load balancing, resource limiting, and specifying end-to-end latencies. The language abstracts from execution time and phase offsets. This permits simple clock typing rules and a stream-based semantics, but requires each component to execute within an overall base period. A program is compiled to a single periodic task in two stages. First, Integer Linear Programming is used to determine phase offsets using standard encodings for dependencies and load balancing, and a novel encoding for end-to-end latency. Second, a code generation scheme is adapted to produce step functions. As a result, components are synchronous relative to their respective rates, but not necessarily simultaneous relative to the base period. This approach has been implemented in a prototype compiler and validated on an industrial application.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Computer systems organization → Embedded software

Keywords and phrases synchronous-reactive, integer linear programming, code generation

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2023.1

1 Introduction

Embedded control software is often designed as a set of components that each repeatedly sample inputs, compute a transition function, and update outputs. Such components must be scheduled so as to share processor resources while respecting timing and communication requirements. Scheduling determines how data propagates along chains of components from sensor acquisitions, through successive computations, to corresponding actuator emissions. The end-to-end latencies of such chains are crucial to overall system performance.

We characterize and extend an approach for developing avionics software based on the synchronous-reactive languages Lustre [29] and Scade [13]. Our application model comprises (i) a set of components whose execution rates are specified as unit fractions ($1/n$) of a base rate, and (ii) a graph of data flow between components. The Worst-Case Execution Time (WCET) of each component must be less than the base period. This is a significant restriction, but one that is acceptable for safety-critical avionics applications. The implementation target is one or more sequential step functions called cyclically to, in turn, call individual component step functions. Data is exchanged by reading and writing static variables.

Besides providing a way to specify real-time behavior, execution rates allow implementations to balance requirements and resources. For example, in the absence of other constraints, a component at rate $1/3$ can be scheduled to run every 3 cycles with any of the phases 0, 1,

or 2. Such choices are made to distribute total computation load over successive cycles, to implement the dataflow specification by ordering variable reads and writes, and to respect resource bounds such as, for example, the capacity of an avionics bus. We show how it is also possible to choose component phases so as to satisfy overall end-to-end latency requirements.

In our approach, scheduling occurs in two stages. The first assigns components to phases and the second orders the components within a cycle. We realize the first stage by generating and solving an Integer Linear Programming (ILP) problem and the second by adapting an algorithm [4] used to compile Lustre and Scade. Using the ILP encoding presented in this article, offline scheduling is restricted by end-to-end latency constraints declared in source programs. As a result, if scheduling succeeds, these source-level constraints are respected by the generated code. We have implemented the presented techniques in a prototype compiler and validated them on a large flight control program.

Industrial context

This article characterizes an industrial approach to developing avionics software. We focus on a flight control and guidance system that is developed as follows. The control laws and monitoring functions are specified in the Scade language. The resulting design comprises approximately 5000 individual components communicating over 120 000 named signals. A component is a *block diagram* comprising blocks and lines: blocks represent basic arithmetic operations, unit delays, filters, etcetera; lines connect block outputs to block inputs. A code generator transforms each Scade component into a C function that reads and writes static variables corresponding to its input signals, output signals, and internal states. Preemption and dynamic scheduling are rigorously avoided to simplify reasoning and testing. The resulting code is then implemented on an embedded platform.

Scade programs are compiled following the synchronous paradigm: code generation produces *step functions* for cyclic execution. Since it is not feasible to execute all 5000 components in a single cycle of the platform, each is executed at an integer multiple (2, 4, 8, 24, or 48) of the base period of 5 ms and scheduled to distribute the computational load. It is crucial that (a) the aggregate of computations executed within a cycle does not exceed the base period; and (b) the final system strictly respects end-to-end latency constraints on servo control loops. The first constraint is taken into account during scheduling and validated on the generated executable using WCET analysis. End-to-end latencies are currently specified indirectly by application engineers who assign execution orders and bounds to certain function sequences. These indirect constraints, and platform limitations related to the avionics bus, are encoded by software engineers into an ILP problem that is solved to give a schedule.

Our work systematizes and streamlines the process described above. The idea is to specify the system as a single program that instantiates the 5000 individual functions together with constraints on resources and end-to-end latencies. In this way, we clarify the overall semantics, allow direct specification of end-to-end latencies, and automate compilation. We have successfully applied our approach to the avionics system, but industrial constraints prevent us reporting the details, so we instead focus on the ROSACE case study [51]. It has only 11 components but is representative of the domain and makes for a good example.

Although our work is guided by a specific application, we believe it is applicable more generally. For instance, many companies develop control applications as Simulink block diagrams and either manually reprogram or automatically generate code for *single-tasking execution* [46, §4-6]. Other applications are developed using a similar non-preemptive tasking model even though they are not specified explicitly as block diagrams. Examples include the open-source ArduPilot [1] and Paparazzi UAV [5] projects.

2 A rate-synchronous model

The first part of this section presents a variant of Lustre [29] with unit-fraction clocks. Unlike in similar programming languages [10, 15, 33, 45, 55], our clocks specify a rate without a phase. This is natural for real-time scheduling where release times may implement data dependencies [9 | 6, §3.5.2]. Our proposition is inspired by Prelude [22, 24] but restricts communication primitives and generates sequential code rather than real-time tasks.

The last part of this section presents our version of ROSACE and the results of the scheduling and code generation techniques which are detailed in the remainder of the article.

2.1 Syntax

As in any Lustre-like language, a function is defined by a set of mutually recursive equations.

$$eq ::= x \texttt{ = } e \mid x^* \texttt{ = } f\texttt{(}e^*\texttt{)}$$

A *basic equation* defines a variable using an expression. An *instantiation* of a function f defines the function inputs using a list of expressions and associates each output to a variable.

An *expression* is a constant, a variable, an application of a unary or binary operator, a conditional expression, the previous value of a variable, a sampling of a faster variable, a sampling of the previous value of a faster variable, or a buffering of a slower variable.

$$\begin{aligned} e ::= {} & c \mid x \mid \diamond\ e \mid e \oplus e \mid \texttt{if } e \texttt{ then } e \texttt{ else } e \mid \texttt{last } x \\ \mid {} & x \texttt{ when } s \mid \texttt{(last } x\texttt{) when } s \mid \texttt{current(}x\texttt{, } s\texttt{)} \end{aligned}$$

$$s ::= \texttt{(}c \texttt{ \% } c\texttt{)} \mid \texttt{(? \% } c\texttt{)}$$

The `last` operator [12, 53] can only be applied to a variable, but otherwise has the same meaning as Lustre's `pre` operator: it delays a signal by one cycle relative to its rate. Unlike `pre`, the `last` operator can be directly translated into flow graphs, as will be seen when they are introduced in Section 3.1, and directly implemented by a shared variable. It may only be applied to a variable declared with an initial last value.

For similar reasons, our `when` operator only applies to a variable or `last` expression. The *sample choice* argument s defines how to sample incoming values. For instance, $1 \,\%\, 4$ selects the second, numbered from zero, of every four values and $? \,\%\, 3$ lets offline scheduling determine which of every three values to sample. This choice is then fixed throughout all executions of the generated code. For $m \,\%\, n$, it must be that $0 \leq m < n$ and $1 < n$.

Our `current` operator buffers the value of a variable x, which must have an initial last value. The sample choice s determines how to hold incoming values. For instance, $2 \,\%\, 4$ specifies to repeat the initial value twice before repeating each input value four times and $? \,\%\, 4$ lets scheduling determine how many times to repeat the initial value.

A program comprises a list of declarations of resources, external functions, and functions.

$$\begin{aligned} p ::= {} & (d\ \texttt{;})^* \\ d ::= {} & \texttt{resource } x \texttt{ : } ty \\ \mid {} & \texttt{node } f\texttt{(}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty\ \texttt{;})^*\texttt{)} \texttt{ requires (}\ (x \texttt{ = } c\ \texttt{;})^*\texttt{)} \\ \mid {} & \texttt{node } f\texttt{(}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \texttt{ returns (}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^*\texttt{)} \\ & \texttt{var}\ (x \texttt{ : } ty \texttt{ :: } ck\ [\texttt{last = } c]\ \texttt{;})^* \texttt{ let}\ (((pragmas\ eq) \mid cst)\ \texttt{;})^+ \texttt{ tel} \end{aligned}$$

$$pragmas ::= [\texttt{label(}x\texttt{)}]\ [\texttt{phase(}c \texttt{ \% } c\texttt{)}]$$

A *resource declaration* introduces a name x for an integer or floating-point quantity that the scheduler must take into account. For example, `busout` for the number of digital outputs to be sent on an avionics bus, or `cpu` for a measure of processor load.

An *external function declaration* specifies the name f of an external function together with its input/output interface and its resource requirements. The inputs and outputs are each specified by a list of variable names and their types. The resource requirements are a list of resource names, each paired with a constant quantity.

A *function definition* specifies the name f and input/output interface and defines its implementation as a list of local variables and a list of equations and constraints. Each variable x has a declared type ty, clock type ck, and, optionally, an initial last value c (denoted $x\text{-}_1 = c$). We only consider primitive types, namely ty ::= `bool` | `int` | `float`, and unit-fraction clocks ck ::= `1/`c, where `1/1`, normally written as `1`, represents the base rate of a function and `1/`c represents a fraction of the base rate. Expressions may refer to input, local, and output variables. Equations define local and output variables only. Each local and output variable must appear at left of exactly one equation. An equation may be preceded by pragmas: `label` specifies a unique identifier and `phase` fixes the schedule. The label of an instantiation x^* `=` f `(` e^* `)` defaults to f when not ambiguous. In addition to equations, the definition may also include resource and timing constraints.

cst ::= `resource balance` x
| `resource` x rel c
| `latency` (`exists` | `forward` | `backward`) rel c `(` x `,` x (`,` x)* `)`

rel ::= `<=` | `<` | `=` | `>` | `>=`

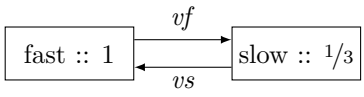
A *balance constraint* directs the scheduler to minimize, across cycles, differences in the sum of a given resource, like `cpu`. A *resource constraint* places a constant bound c on the sum of a resource, like `busout`, in a single cycle. A *latency constraint* sets a constant bound `c` on the end-to-end latency of one instance, or all forward or backward instances, of a chain. A *chain* is a sequence of equations, eq_0, eq_1, ..., eq_{n-1} where at least one of the variables defined at left of eq_i appears in an expression at right of equation eq_{i+1}.

2.2 Semantics

The focus here is not on programming languages, so we only outline the main principles. In a dataflow semantics [34], expressions are associated with sequences of values, equations associate variables to sequences, and functions map sequences to sequences. In a synchronous dataflow semantics [7], infinite sequences, or streams, are aligned. The idea is that they are calculated together over successive rounds. Streams are often presented in grids with rows for expressions and columns for rounds. Alignment may be shown in a grid by leaving gaps. In our model, slower streams are shown by placing their values in wider columns: they are synchronous at their rate and "simultaneous" with multiple values of faster streams.

Consider a simple example adapted from Forget [21, Figure 5.1] and sketched in Figure 1a. There is a fast component that executes every cycle and a slow one that executes every three cycles. We instantiate the two components below.

```
node eg1() returns ()
var vf : int :: 1;
    vs : int :: 1/3 last = 0;
     n : int :: 1 last = 0;
let
  n = (last n) + 1;
  vf = n + current(vs, (2 % 3));
  vs = (vf when (1 % 3)) + 5;
tel
```



(a) Fast and slow components.

vf	1	2	10	11	12	23	24	25	39	$\cdots$
n	1	2	3	4	5	6	7	8	9	$\cdots$
vs	7			17			30			$\cdots$

(b) Synchronized streams in the rate-synchronous model.

Figure 1 Structure and trace of `node eg1()`.

$$[\![e_1 \oplus e_2]\!](i) = [\![e_1]\!](i) \oplus [\![e_2]\!](i)$$

$$[\![\texttt{last}\ x]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i = 0 \\ [\![x]\!](i-1) & \text{otherwise} \end{cases}$$

$$[\![x\ \texttt{when}\ (s\,\%\,n)]\!](i) = [\![x]\!](n \cdot i + s)$$

$$[\![\texttt{current}(x, (s\,\%\,n))]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i < s \\ [\![x]\!]\left(\left\lfloor \frac{i-s}{n} \right\rfloor\right) & \text{otherwise} \end{cases}$$

$$[\![(\texttt{last}\ x)\ \texttt{when}\ (s\,\%\,n)]\!](i) = \begin{cases} x_{\text{-}1} & \text{if } i + s = 0 \\ x(n \cdot i + s - 1) & \text{otherwise} \end{cases}$$

(a) Stream-based semantics (x_{-1} is a declared initial last value).

$$\frac{e_1 :: 1/n \qquad e_2 :: 1/n}{e_1 \oplus e_2 :: 1/n}$$

$$\frac{x :: 1/n}{\texttt{last}\ x :: 1/n}$$

$$\frac{x :: 1/m}{x\ \texttt{when}\ (\cdot\,\%\,n) :: 1/mn}$$

$$\frac{x :: 1/mn}{\texttt{current}(x, (\cdot\,\%\,n)) :: 1/m}$$

(b) Clock typing rules.

Figure 2 Key formal definitions for expressions.

The `vf` and `vs` signals are declared with their execution rates. The local variable `n` is a counter used by the fast component. Both `vs` and `n` are declared with initial last values, so that they can be used with the `last` and `current` operators. Rate transitions are expressed with `current` (slow-to-fast) and `when` (fast-to-slow) operators. The fast value, `vf`, sums the counter with the initial last value of `vs` repeated twice and then each of its values repeated three times. The slow value, `vs`, adds five to the second of every three fast values. Figure 1b shows the resulting streams.

The grid shown in Figure 1b gives the meaning of the program. Each row in the grid associates a variable with a stream. This intuitive idea is made precise in Figure 2a by a semantic function $[\![\cdot]\!]$ that maps each syntactic element to a stream ($\mathbb{N}_0 \to \mathbb{V}$); that is, to a function from a cycle number i to the value of the expression in that cycle. The semantic function distributes over expressions to return a constant function for literals c or the value of an input or equation for variables x (not shown). A binary operator $\oplus$ is applied pointwise to its input streams. The `last` operator returns the initial last or preceding value of the named stream. The `when` operator selects one of every n input values. The `current` operator repeats values from the input stream with a special case for the first s values. The rule for (`last` x) `when` $(s\,\%\,n)$ can be derived by substitution from the rules for `when` and `last`. If a program contains $?\,\%\,n$, the s in the corresponding stream equation can take any value in $[0, n)$ and the semantic rules may admit more than one solution.

An expression like $x + (x\ \texttt{when}\ (0\,\%\,3))$ has a well-defined value according to the semantic rules but cannot be implemented without an unbounded buffer. Programs that require unbounded memory are unsuited to embedded control applications. In Lustre-like languages, a *clock type system* prescribes how streams may be combined [7] and thereby which programs are accepted for compilation. Every expression is associated with a rate by syntax-directed rules that define acceptable expressions. Figure 2b shows the main rules for the presented

language. For binary operators, the two input expressions (above the line) and the output expression (below the line) must all have the same rate ($1/n$). A similar rule, not shown, applies to function instances. The `last` operator does not change the rate of its argument, and the rate transition operators respectively divide or multiply the rate of an input argument.

Communications between non-harmonic rates require an intermediate equation at a common multiple of both rates. For example, between a writer `w :: 1/6` and a reader `r :: 1/8`, one could add an explicit buffer `b :: 1/2`.

```
b = current(w, (? % 3));
r = b when (? % 4);
```

Finally, there are well-clocked programs that have no semantics. For instance, in the `eg1` node, replacing `last n` by `n` or exchanging the $2 \% 3$ and $1 \% 3$ sample choices results in cyclic definitions for which there are no solutions. There are also correct programs that our current code generator cannot handle. For instance, changing the definition of `vs` to `vf when` $(0 \% 2) +$ `vf when` $(1 \% 2)$ gives a valid program that cannot be implemented using a single shared variable for `vf`. We do not pursue these issues here: scheduling simply fails for such programs.

2.3 Compilation

The source language allows specifying and reasoning about programs in terms of streams of values. Generating code for such programs and defining end-to-end latencies requires a change of perspective. We now want to consider a program as a set of components that repeatedly read and write shared variables. Each iteration of the program is termed a *cycle*. We will also refer to the *hyperperiod* of a set of equations *hp*, which is the Least Common Multiple (LCM) of their periods, that is, of the denominators of their rates. In the source semantics, each variable is associated with a single value for the duration of its *round* (that is, during one period / within one dataflow grid column), but the corresponding shared variable is updated in a specific cycle when the code generated for its defining equation executes. Execution order now becomes paramount: each equation must be assigned a phase relative to its execution rate and ordered relative to other equations for execution within a cycle.

The following C code was generated for `eg1`.

```
static int c = 0;

void step() {
  static int vs = 0, n = 0;
  int vf;

  n = n + 1;
  vf = n + vs;
  if (c == 1) { vs = vf + 5; }
  c = (c + 1) % 3;
}
```

A static variable `c` is introduced to count off successive phases of the hypercycle ($hp = 3$). Static variables are declared for `c` and `vs`. Their values persist across cycles. A local variable is declared for `vf` since its value is only needed within a cycle. In every cycle, `n` is updated first, then `vf`, and then, but only in the second of every three cycles, `vs`. The new value of `vs` is not used until the subsequent cycle.

The assignment of equations to phases, termed *scheduling*, and the ordering of equations within a cycle, termed *microscheduling*, are central to the compilation scheme presented in the following sections on constraint and code generation. First, though, we apply the specification language to an existing case study which will serve as a running example.

2.4 Example: ROSACE

The ROSACE case study [51] considers the development process of a longitudinal flight controller. Figure 3a shows our reimplementation of the original Prelude program [51, Figure 3 and §III.B]. Since the fastest components run at 200 Hz, that is, with a period of 5 ms, we set the cycle period to 2.5 ms to allow load balancing. Inputs are declared for the desired altitude `h_c` and speed `va_c`, both at 10 Hz. Outputs are declared for the throttle `d_th_c` and elevator deflection `d_e_c` commands, both at 50 Hz. Local signals are declared with explicit rates. The node body contains three groupings of components. At 200 Hz are the `elevator`, `engine`, and `dynamics` components that provide a discretized model of the environment. Such components would not normally be included in a controller, but we maintain them for the sake of the example. At 100 Hz are five filters on altitude `h`, vertical acceleration `az`, pitch rate `q`, vertical speed `vz`, and true airspeed `va`. At 50 Hz are the control laws for tracking the requested altitude `alt_hold`, vertical speed `vz_control`, and airspeed `va_control`. We use free sample choices (?) in `when` and `current` rate transitions to give greater scheduling freedom at the cost of underspecification.

Figure 3c graphs the data flows between components. There is a vertex for each equation. An arc indicates that a signal defined by the "tail" equation is used in the "head" equation. For example, there is an arc from `az_filter` to `vz_control` due to the `az_f` signal.

Strictly speaking, either the clock types of the local variables or the sample rates of the `when` and `current` operators could be inferred by the compiler; as in Prelude or Simulink. In this work, we prefer to state them explicitly. The compiler detects and signals any discrepancies when it checks clock types after parsing a source file.

The node body also contains two constraints. A given task chain must execute within 5 ms every 20 ms [51, §III.C], that is, with an end-to-end latency ≤ 2 cycles. The `exists` keyword specifies that only one instance of the chain per hypercycle need satisfy the bound. A resource called `ops` must be balanced across cycles. We calculated `ops` values in a simple way from the Prelude definitions: +10 for each libm function, node instance, or if/then/else; +3 for each multiplication or division; +1 for other operators and each equation. The weights, see Figure 3d, are added to external function declarations for each component, for example,

```
node alt_hold (h_c, h_f : float)
returns (vz_c : float)
requires (ops = 201);
```

Figure 3d also shows the scheduled phases. Since the `dynamics` node requires many more operations than the others, it has been scheduled with `elevator` in odd cycles. These two components are scheduled in the same cycle due to the tight latency constraint. The other components are scheduled in even cycles such that dependency and latency constraints are satisfied. Better load balancing could be obtained by inlining the `dynamics` and `alt_hold` nodes, and halving the cycle period while doubling all the rates.

The schedule is realized in the generated code, Figure 3b, via the guards of `if` and `switch` statements. The optimization of conditionals can be reduced so that `if (c % 4 == 2) {···}` is after, rather than within, `if (c % 2 == 0) {···}`, and the grouping by period is retained. Microscheduling determines the order of function calls for a given value of the counter `c`. Notably, `elevator` runs before `dynamics`; the `*_filter` run before `va_control`, `vz_control`, and `alt_hold`; and `alt_hold` runs before `vz_control`. In these cases, an output calculated by one component is propagated in a cycle to become the input for another. Conversely, the outputs calculated by `dynamics` are sampled less often by the `*_filter` and with a delay of one cycle. In this code, signals are implemented by reading and writing static variables inside components. The compiler can also generate code that passes values on the stack.

```
node assemblage(
   h_c : float :: 1/40 last = 0.;
  va_c : float :: 1/40 last = 0.)
returns (
  d_th_c : float :: 1/8 last = 1.6402;
  d_e_c  : float :: 1/8 last = 0.0186)
var
  vz_c : float :: 1/8;
  d_e, th, h, az, va, q, vz : float :: 1/2;
  vz_f, va_f, h_f, az_f, q_f : float :: 1/4;
let
  (* 200Hz = 1/2 *)
  d_e = elevator(current(d_e_c, (? % 4)));
  th = engine(current(d_th_c, (? % 4)));
  (va, az, q, vz, h) = dynamics(th, d_e);
  (* 100Hz = 1/4 *)
   h_f =  h_filter( h when (? % 2));
  az_f = az_filter(az when (? % 2)); ...
  (* 50Hz = 1/8 *)
  vz_c = alt_hold(current(h_c, (? % 5)),
                  h_f when (? % 2));
  d_e_c = vz_control(vz_c,
              vz_f when (? % 2),
              q_f when (? % 2),
              az_f when (? % 2));
  d_th_c = va_control(
              current(va_c, (? % 5)),
              va_f when (? % 2),
              q_f when (? % 2),
              vz_f when (? % 2));

  latency exists <= 2 (dynamics, h_filter,
              alt_hold, vz_control, elevator);
  resource balance ops;
tel
```

(a) Source program.

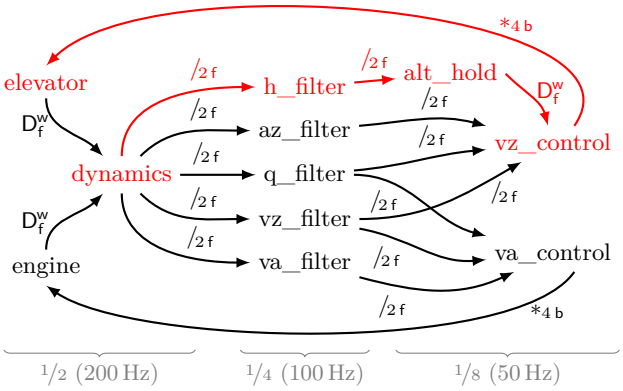
```
static int c = 0;

static float h_c = 0,
    va_c = 0;
    d_th_c = 1.6402,
    d_e_c = 0.0186
    vz_c, ..., q_f;

void step0()
{
  if (c % 2 == 0) {
    engine();
    if (c % 4 == 2) {
      vz_filter();
      h_filter();
      va_filter();
      q_filter();
      az_filter();
    }
  } else {
    elevator();
    dynamics();
  }
  switch (c) {
  case 2:
    va_control();
    break;
  case 6:
    alt_hold();
    vz_control();
    break;
  }

  c = (c + 1) % 8;
}
```

(b) Generated code.



(c) Flow graph.

	ops	*phase*	
elevator	98	1 % 2	(p_e)
engine	82	0 % 2	
dynamics	1174	1 % 2	(p_d)
h_filter	38	2 % 4	(p_h)
az_filter	37	2 % 4	
q_filter	37	2 % 4	
vz_filter	37	2 % 4	
va_filter	38	2 % 4	
alt_hold	201	6 % 8	(p_a)
vz_control	88	6 % 8	(p_v)
va_control	90	2 % 8	

(d) Schedule.

Figure 3 Main ROSACE [51, Figure 3] node.

3 Constraint generation

A source program is transformed into sequential code in three steps. First, the compiler is invoked to generate an ILP encoding of the constraints on equation phases. Second, the encoding is passed to an external solver like the IBM ILOG CPLEX Optimizer [32]. Third, if the solver finds a solution, it is returned to the compiler which then generates code. Both compiler invocations manipulate a flow graph constructed from the program source. We first describe the flow graphs before presenting the ILP encodings of data dependencies, resource constraints, and latency chains. The second compiler invocation is addressed in Section 4.

3.1 Flow graphs

As an intermediate step in the passage from dataflow programs to sequential code, we adapt the standard definition of flow graphs by labeling edges with sampling and microscheduling information. Subsequent definitions and reasoning are in terms of flow graphs, which thus provide a way to apply the results independently of the source language.

A *flow graph* is a directed graph (V, A) with labeled arcs $A \subseteq V \times S \times C \times V$. The first label specifies the *sampling*: $S = \{\mathsf{D}^{\mathsf{w}}, \mathsf{D}^{\mathsf{r}}, \mathsf{D}^{?}, /_n, /_n^{\mathsf{L}}, *_n, \ldots\}$ with $n \in \{2, 3, \ldots\}$; and the second one specifies the *concomitance*: $C = \{\mathsf{f}, \mathsf{b}\}$. Vertexes V represent equation labels, which for simplicity we conflate with the equations themselves. An arc from eq_w to eq_r indicates that a variable defined by eq_w appears in the defining expression of eq_r. That is, values flow from eq_w to eq_r, or, in implementation terms, eq_w writes shared variables that eq_r reads. The flow graph for the ROSACE example is shown in Figure 3c.

There are three types of sampling. *Direct sampling* indicates equations at the same rate: D^{w} is the standard case where writing must occur before reading; D^{r} is used to encode the `last` operator, where variables must be read before they are written; and $\mathsf{D}^{?}$ indicates that writing and reading may be scheduled in any order, but that the concomitance is important, otherwise the arc could simply be omitted. *Fast-to-slow sampling*, $/_n$, indicates that the writer has a higher rate than the reader, the subscript, n, is the relative ratio. The $/_n^{\mathsf{L}}$ form additionally specifies sampling of the `last` operator. *Slow-to-fast sampling*, $*_n$, indicates that the writer is slower than the reader. The symbols are adapted from Prelude [21, §4.2.2] and recall the clocking rules of Figure 2b. A `when` divides the writer rate by sampling a subset of its values. A `current` multiplies the writer rate by duplicating (buffering) input values.

For scheduling and microscheduling to work coherently they must agree on the order of related equations within a single phase. The concomitance labels C solve this problem by fixing the intraphase ordering prior to scheduling. They specify how two equations will be ordered if ever they execute in the same cycle: f, *forward concomitance*, constrains the write to occur before the read, immediately communicating a value; and b, *backward concomitance*, constrains the write to occur after the read, delaying the communication by one period.

For our source language, the generation of a flow graph from a function definition is immediate. For each equation eq_r, we simply descend into the defining expression $(x = e)$ or argument expressions $(x^* = f(e^*))$, ignoring constants and continuing recursively through operators and conditionals. The remaining cases all involve a variable x. If the variable is an input, or `last` y for y defined by eq_r, nothing is done, otherwise its defining equation eq_w is identified and an arc is added to the flow graph as per the cases:

x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} eq_r$	(`last` x) `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/_n^{\mathsf{L}\,\mathsf{b}}} eq_r$	
`last` x	$eq_w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} eq_r$	`current`$(x, (\cdot \,\%\, n))$	$eq_w \xrightarrow{*_n\,\mathsf{f}} eq_r$	(by default)
x `when` $(\cdot \,\%\, n)$	$eq_w \xrightarrow{/_n\,\mathsf{f}} eq_r$		$eq_w \xrightarrow{*_n\,\mathsf{b}} eq_r$	(if "fast-first")

The `current` operator normally has forward concomitance, but a "fast-first" option that makes it backward permits equations to be ordered within a cycle from fastest to slowest.

Clock typing almost guarantees the absence of arcs with different labels between two equations, but it is still necessary to reject equations that read both x and `last` x, for example, $y = x + \texttt{last}\ x$. Their compilation requires extra buffering: $lx = \texttt{last}\ x$, $y = x + lx$.

3.1.1 Circular dependencies

Equations with circular dependencies, or "algebraic loops" [45, §3–39], are normally rejected since a unique solution does not exist or cannot be found without iteration. For example, no streams satisfy both $x = y + 1$ and $y = x + 1$, and all pairs of identical streams satisfy $x = y$ and $y = x$. Typically, circular dependencies are broken by manually adding `last` operators. For example, the definitions $x = \texttt{last}\ y + 1$ and $y = x + 1$ are valid.

A Lustre program is analyzed by checking a graph of its static dependencies for cycles [29, §III.A] [4, §3.1]. A *dependency graph* is obtained from a flow graph by reversing all edges with backward concomitance. There are two kinds of cycles in such graphs. A *direct cycle* only contains arcs with direct sampling labels. All the equations in such a cycle have the same rate, will necessarily be scheduled in the same phase, and cannot be microscheduled. An *inter-rate* cycle contains at least one slow-to-fast and one fast-to-slow arc.

3.1.1.1 Direct cycles

In the industrial context that motivates our work, the standard solution of breaking direct cycles by manually adding `last` operators is untenable. There are simply too many cycles, mostly due to feedback from monitoring and maintenance features, and too many variables. Furthermore, it does not usually matter whether the most recent or last value is read as the extra delay usually has no consequence on overall system behaviour. Many variables carry samples of signals that change slowly, or their contribution to feedback and output calculations is minimal. Manually breaking cycles complicates development and overconstrains scheduling.

As a solution to this problem, Wyss et al. [57] propose a *don't-care* operator `dc x` that the compiler resolves to `x` or `last x`. Iooss et al. [33, §5.1] adopt a similar solution with their `last`? x operator. We considered adding such an operator to our source language, but for the flight control system described in the introduction, we found that programmers would simply add this operator to all direct reads. For large programs, this complicates the source text without providing any real advantages.

Instead, we provide three compiler options that transform a flow graph prior to scheduling: (i) *relax same period*, (ii) *relax same period cycles*, and (iii) *cut same period cycles*. The first drops all direct arcs that are not needed for latency chains and relabels those in latency chains with $\mathsf{D}^?$. It has the same effect as replacing all variables x in expressions e with `last`? x. The second is similar but only applies to arcs within a strongly connected component (SCC) of the dependency graph. The third calculates a *minimum feedback arc set* of the dependency graph and inverses the concomitance of those arcs in the flow graph to eliminate all cycles. That is, it replaces some variable instances x with `last`? x. Finding the smallest such set is NP-hard for general graphs [35], so we adapt a heuristic [18, 19] that executes in time $O(|V| \cdot |A|)$ and produces a minimum feedback arc set that is at worst twice the size of a minimal one. In limited experiments, neither of the three transformations proved better than the others in terms of scheduling time or result quality. The compiler options provide a pragmatic solution to a practical problem but require deforming the source-level specification. It remains to be seen whether such applications can be specified in a more principled manner.

3.1.1.2 Inter-rate cycles

Identifying cyclic dependencies between equations at multiple rates is challenging. For example, the equations $x = y$ `when` $(1 \,\%\, 2)$ and $y = \texttt{current}(x, (0 \,\%\, 2))$ have no solution since y must buffer the value of x in the first of every two rounds, but x may only sample y in the second one. Dependencies may even change over the course of a hypercycle. For now, we simply accept that constraint solving will fail for such programs.

When the generated constraints have a solution, however, we must ensure that microscheduling will succeed. We do so by transforming the original flow graph, prior to scheduling, to inverse the concomitance of all forward slow-to-fast arcs ($*_{n\,\mathsf{f}}$) between vertexes in the same SCC of the dependency graph. Then, even if interdependent equations are scheduled in the same phase, they can still be microscheduled. The semantics of the program is unchanged. This treatment occurs in Figure 3c where the arcs from `vz_control` to `elevator`, and from `va_control` to `engine` have backward concomitance. According to the schedule of Figure 3d, only `va_control` and `engine` may execute together in the same cycle, and `engine` then goes first, as in Figure 3b.

3.2 Dependency constraints

The flow graph produced from a program and modified as described above is translated into a set of ILP constraints. The encoding is unsurprising. The *phase* of an equation eq is represented by an integer variable $0 \leq \mathtt{p}_{eq} < \text{period}(eq)$, where $\text{period}(eq) = n$ for $eq :: {}^1\!/_n$. Any solution found for the phase variables is a valid schedule. A variable is unnecessary if $\text{period}(eq) = 1$. The substitution $\mathtt{p}_{eq} = 0$ is then applied to constraints and solutions.

In the following, we consider arbitrary pairs of equations eq_w and eq_r, where eq_w defines a variable w that eq_r uses to define a variable r. We will reason in terms of w and r, conflating variables and equations and ignoring the details of expressions. The generalization is trivial.

For equations of the form $r = w$, writing must occur before reading. For those of the form $r =$ `last` w, reading must occur before writing. Whether reading and writing may occur in the same phase or not, and thus the strictness of inequalities, depends on the concomitance. For example, for $\mathsf{D}^{\mathsf{w}}_{\mathsf{b}}$, w and r must not be scheduled in the same phase, since backward concomitance requires that microscheduling place the computation of r before that of w. Flow-graph arcs are thus translated to phase constraints as follows.

$$w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r \quad \text{becomes} \quad \mathtt{p}_w \leq \mathtt{p}_r \qquad\qquad w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{f}}} r \quad \text{becomes} \quad \mathtt{p}_r < \mathtt{p}_w$$
$$w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{b}}} r \quad \text{becomes} \quad \mathtt{p}_w < \mathtt{p}_r \qquad\qquad w \xrightarrow{\mathsf{D}^{\mathsf{r}}_{\mathsf{b}}} r \quad \text{becomes} \quad \mathtt{p}_r \leq \mathtt{p}_w$$

For an equation $r = w$ `when` $(i \,\%\, n)$, where $0 \leq i < n$, with forward concomitance, the read must occur with or after the ith write and strictly before any subsequent write:

$$i \cdot \text{period}(w) + \mathtt{p}_w \quad \leq \quad \mathtt{p}_r \quad < \quad (i+1) \cdot \text{period}(w) + \mathtt{p}_w \quad \text{for } w \xrightarrow{/_{n\,\mathsf{f}}} r.$$

For backward concomitance, $w \xrightarrow{/_{n\,\mathsf{b}}} r$, the strictnesses are reversed: $\cdots < \mathtt{p}_r \leq \cdots$.

For free sample choices, $r = w$ `when` $(? \,\%\, n)$, the rule is *write before first read*. Only the lower bound is needed and $i = 0$, giving $\mathtt{p}_w \leq \mathtt{p}_r$ for $w \xrightarrow{/_{n\,\mathsf{f}}} r$, and $\mathtt{p}_w < \mathtt{p}_r$ for $w \xrightarrow{/_{n\,\mathsf{b}}} r$.

For an equation $r =$ (`last` w) `when` $(i \,\%\, n)$, with backward concomitance, the read must occur strictly after any $(i-1)$th write and before or with the subsequent write:

$$(i-1) \cdot \text{period}(w) + \mathtt{p}_w \quad < \quad \mathtt{p}_r \quad \leq \quad i \cdot \text{period}(w) + \mathtt{p}_w \quad \text{for } w \xrightarrow{/^{\mathsf{L}}_{n\,\mathsf{b}}} r.$$

For $r =$ (`last` w) `when` $(? \,\%\, n)$, the rule is *read before last write*. Only the upper bound is needed and $i = n - 1 = \frac{\text{period}(r)}{\text{period}(w)} - 1$, giving $\mathtt{p}_r \leq \text{period}(r) - \text{period}(w) + \mathtt{p}_w$.

For an equation $r = \texttt{current}(w, (i \,\%\, n))$ with forward concomitance, the write must occur with or before the ith read but strictly after the preceding one:

$$(i-1)\cdot \text{period}(r) + \mathtt{p}_r \quad < \quad \mathtt{p}_w \quad \leq \quad i \cdot \text{period}(r) + \mathtt{p}_r \quad \text{for } w \xrightarrow{*_{n\,\mathsf{f}}} r.$$

For $r = \texttt{current}(w, (? \,\%\, n))$, the rule is *write before last read*. Only the upper bound is needed and $i = n - 1 = \frac{\text{period}(w)}{\text{period}(r)} - 1$, giving $\mathtt{p}_w \leq \text{period}(w) - \text{period}(r) + \mathtt{p}_r$.

3.3 Resource constraints

The resource constraint encoding is relatively standard [33, §4.3]. For each equation eq with $period(eq) > 1$, we introduce a set of *phase weight* variables: $\mathtt{pw}_{eq,i} \in \{0, 1\}$ indicates whether eq is scheduled in phase $0 \leq i < \text{period}(eq)$. Two constraints relate them to the corresponding phase variable:

$$\sum_{i=0}^{\text{period}(eq)-1} \mathtt{pw}_{eq,i} = 1 \qquad\qquad \sum_{i=0}^{\text{period}(eq)-1} i \cdot \mathtt{pw}_{eq,i} = \mathtt{p}_{eq}.$$

For example, for $\text{period}(eq) = 3$, if $\mathtt{p}_{eq} = 2$ then $\mathtt{pw}_{eq,0} = 0$, $\mathtt{pw}_{eq,1} = 0$, and $\mathtt{pw}_{eq,2} = 1$.

For a declared resource res, let $w(res, eq)$ represent the amount of res required by the equation eq. For instantiations, $w(res, x^* = f(e^*))$ refers to the constant given in the declaration of f and defaults to 0. Otherwise $w(res, x = e) = 0$.

A constraint of the form "`resource balance` *res*" is encoded by introducing an integer variable $\mathtt{rmax}_{res}$, and a constraint for each phase of the hypercycle $0 \leq k < hp$:

$$0 \leq \mathtt{rmax}_{res} - \sum_{\{eq \,|\, \text{period}(eq) > 1\}} w(res, eq) \cdot \mathtt{pw}_{eq,(k \bmod \text{period}(eq))}.$$

The objective is then to minimize each $\mathtt{rmax}_{res}$ variable, which means choosing equation phases that equalize as much as possible the sums of weights across all phases of the hypercycle. The $k \bmod \text{period}(eq)$ term accounts for the repetition of an equation across the hypercycle.

A source constraint of the form "`resource` *res* $\sim c$" is encoded by constraining the sum of resources in each phase $0 \leq k < hp$:

$$\sum_{\{eq \,|\, \text{period}(eq) > 1\}} w(res, eq) \cdot \mathtt{pw}_{eq,(k \bmod \text{period}(eq))} \sim c - \sum_{\{eq \,|\, \text{period}(eq) = 1\}} w(res, eq).$$

The equations with $\text{period}(eq) = 1$ are included in every phase by subtracting their weights from the constant c.

3.4 Latency constraints

Latency refers to the number of cycles from an initial read forwards to eventual related writes, or from a final write backwards to earlier related reads. Compared to the ILP encodings of dependency and resource constraints, the treatment of latency constraints is less obvious. A constraint `latency` $m \sim b\ (eq_0, eq_1, \ldots, eq_{n-1})$, where $m \in \{\texttt{exists}, \texttt{forward}, \texttt{backward}\}$ and b is a constant bound, is valid only if the chain describes a path through the source program's flow graph, $eq_0 \xrightarrow{s_0, c_0} eq_1 \xrightarrow{s_1, c_1} \cdots \xrightarrow{s_{n-2}, c_{n-2}} eq_{n-1}$. A schedule associates phases to the equations in a chain and thereby induces forward and backward end-to-end latencies.

For the ROSACE example, the latency path is $d \xrightarrow{/_{2\,\mathsf{f}}} h \xrightarrow{/_{2\,\mathsf{f}}} a \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} v \xrightarrow{*_{4\,\mathsf{b}}} e$, written here with truncated equation labels and traced in red in the flow graph of Figure 3c. The left half of Figure 4 plots the elements of this chain over a hypercycle, $hp = \mathsf{lcm}(2, 4, 8, 8, 2) = 8$, for the schedule of Figure 3d. There is a row for each equation: the downward pointing arrows indicate the equation's scheduled phase repeated across the hypercycle.

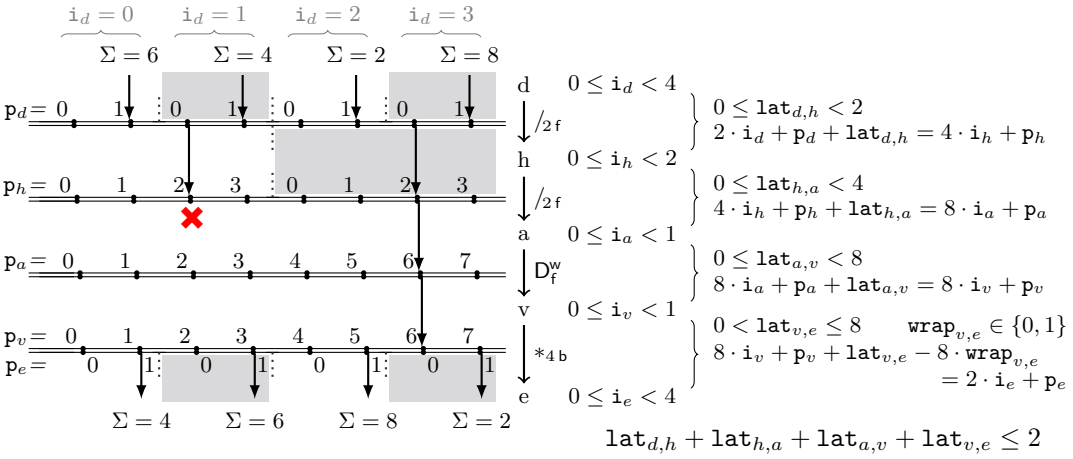


Figure 4 Unsimplified constraints for `latency exists <= 2 (d, h, a, v, e)`.

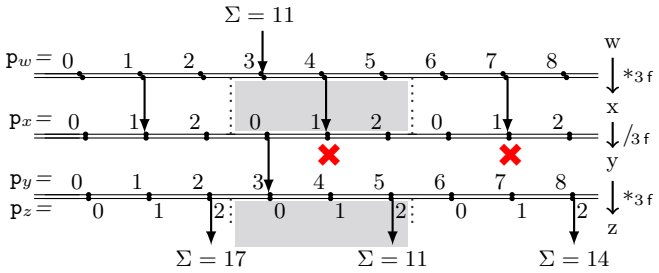


Figure 5 Latency chain: branching then selection.

The `forward` latencies, or *reaction times* (*first-to-first* [20, Figure 7]), are determined by working from the first equation in the chain through to the last. Consider the first of the four instances of d: the next closest instances are those of h after 1 cycle, the only instance of a after 4 cycles, that of v after 0 cycles, and finally the fourth instance of e after 1 cycle. The end-to-end latency is thus $1 + 4 + 0 + 1 = 6$. This is not the actual first-to-first path, which rather passes via the second instance of h, but it does not matter since, in terms of the sum of latencies, the paths commute. By always taking the next closest instance, we arrive at the correct corresponding last instance. The other forward end-to-end latencies are calculated similarly and shown at the top of the figure. Note that the fourth instance of d is sampled in a subsequent hypercycle and that the latency calculation thus "wraps around" through the first instance of h. The `backward` latencies, or *data ages* (*last-to-last* [20, Figure 7]), are determined by working from the last equation in the chain back to the first one. Consider the fourth instance of e: the closest preceding instance of v is 1 cycle before, those of a and h are immediate, and the third instance of d is 1 cycle before. The end-to-end latency is thus $1 + 0 + 0 + 1 = 2$. The backward end-to-end latencies are shown at the bottom of the figure. The `forward` and `backward` constraints require that all end-to-end latencies satisfy the bound, the `exists` constraint only requires that one backward latency does.

The goal now is to define an ILP encoding that characterizes end-to-end latencies in terms of phase variables. Bounding the resulting latency formulas constrains their phase variables and thus restricts the set of valid schedules.

3.4.1 Minimum pairwise latencies

A first idea is to calculate the end-to-end latency as the sum of minimum pairwise latencies. The minimum pairwise latency of a link with forward concomitance from w to r is

$$\texttt{lat}_{w,r} = (\texttt{p}_r - \texttt{p}_w + \text{period}(w)) \bmod \text{period}(w) \quad \text{with } 0 \leq \texttt{lat}_{w,r} < \text{period}(w).$$

Adding period(w) avoids a negative modulo. For a link with backward concomitance, the minimum pairwise latency is period(w) when $\texttt{p}_w = \texttt{p}_r \bmod \text{period}(w)$, giving

$$\texttt{lat}_{w,r} = ((\texttt{p}_r - \texttt{p}_w + \text{period}(w) - 1) \bmod \text{period}(w)) + 1 \quad \text{with } 0 < \texttt{lat}_{w,r} \leq \text{period}(w).$$

Unfortunately this idea does not work for chains that contain both slow-to-fast links, which introduce branching, and fast-to-slow links, which filter branches. For example, consider the scheduled chain in Figure 5. The sequence of flow-graph arcs are shown at right. Taking the sum of minimum pairwise latencies incorrectly gives $1 + 2 + 2 = 5$. The correct forward latency, 11, is show at top, and the backward latencies, 11, 14, and 17, are shown at bottom. As explained by Feiertag et al. [20, Figure 7], the problem is that scheduling choices induce different propagation paths between a write to a chain's first variable and a corresponding read of its last one.

3.4.2 Closest instances

A better idea for encoding end-to-end latency is to characterize an arbitrary propagation path by introducing variables to identify the *instances* of each closest reader/writer relative to the hypercycle. Each such variable is constrained by the preceding write and subsequent read, and the sum of their pairwise latencies gives the end-to-end latency. We explain the idea on the ROSACE example before presenting the formal definition.

Returning to Figure 4, we introduce an instance variable for each equation in the chain. Since there are four instances of d that could participate in a path through the hypercycle, its instance variable is $0 \leq \texttt{i}_d < 4$. Similarly, the instance variable for h is $0 \leq \texttt{i}_h < 2$. We represent the latency between these two arbitrary instances by introducing a variable $0 \leq \texttt{lat}_{d,h} < L$. The value of L is crucial. Since the example has an `exists` constraint, which applies to backward latencies, each reader instance must be associated to the immediately preceding writer instance by setting L to the period of the writer. Thus, here, $L = \text{period}(d) = 2$. For forward latencies, each writer instance must be associated to the immediately succeeding reader instance by setting L to the period of the reader. A constraint is introduced to match writer and reader instances: $2 \cdot \texttt{i}_d + \texttt{p}_d + \texttt{lat}_{d,h} = 4 \cdot \texttt{i}_h + \texttt{p}_h$. For each equation, we multiply the instance by the period and add the phase. The difference gives the pairwise latency. The right side of Figure 4 shows the result of applying this idea along the chain from d through to v. The instance variables $\texttt{i}_a$ and $\texttt{i}_v$ are not strictly necessary since they always equal zero. In the implementation, such variables are removed in a separate constant propagation pass.

For the $v \xrightarrow{*4\,\texttt{b}} e$ link, the value of v may be read from the previous hypercycle. We permit such wraparounds by adding a variable $\texttt{wrap}_{v,e} \in \{0, 1\}$ and subtracting $hp \cdot \texttt{wrap}_{v,e}$ from the writer expression: $8 \cdot \texttt{i}_v + \texttt{p}_v + \texttt{lat}_{v,e} - 8 \cdot \texttt{wrap}_{v,e} = 2 \cdot \texttt{i}_e + \texttt{p}_e$. This encodes a modulo allowing instances to match within ($\texttt{wrap}_{v,e} = 0$) and across ($\texttt{wrap}_{v,e} = 1$) hypercycles. The strictnesses of the bounds $0 < \texttt{lat}_{v,e} \leq 8$ account for the backward concomitance.

Finally, the sum of pairwise latencies is constrained: $\texttt{lat}_{d,h} + \texttt{lat}_{h,a} + \texttt{lat}_{a,v} + \texttt{lat}_{v,e} \leq 2$.

The scheme sketched above is implemented as a function from a list of flow-graph arcs to a set of ILP constraints. The function first calculates hp, the LCM of the periods in the chain's equations, and adds an instance variable for the initial writer $0 \leq \mathtt{i}_w < {}^{hp}/_{\text{period}(w)}$. Then, for each arc $w \xrightarrow{s,c} r$, it adds three fresh variables, with their bounds,

$$0 \leq \mathtt{i}_r < {}^{hp}/_{\text{period}(r)}$$
$$0 \leq \mathtt{lat}_{w,r} < L, \text{ for } c = \mathsf{f} \quad / \quad 0 < \mathtt{lat}_{w,r} \leq L, \text{ for } c = \mathsf{b}$$
$$\text{where } L = \text{period}(r) \text{ if } \mathtt{forward} \text{ and } L = \text{period}(w) \text{ if } \mathtt{backward},$$
$$0 \leq \mathtt{wrap}_{w,r} \leq 1, \text{ if } \mathtt{forward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, *_n\} \text{ or if } \mathtt{backward} \text{ and } s \notin \{\mathsf{D}^{\mathsf{w}}, /_n\},$$

and an equality constraint,

$$\text{period}(w) \cdot \mathtt{i}_w + \mathtt{p}_w + \mathtt{lat}_{w,r} - hp \cdot \mathtt{wrap}_{w,r} = \text{period}(r) \cdot \mathtt{i}_r + \mathtt{p}_r.$$

A $\mathtt{wrap}_{w,r}$ term is not needed if, within a hypercycle, the dependency constraints guarantee that, if `forward`, a read follows the last write; or if `backward`, a write precedes the first read. The function compresses maximal sequences $w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} \cdots \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r$ to $w \xrightarrow{\mathsf{D}^{\mathsf{w}}_{\mathsf{f}}} r$. Finally, it adds the requested constraint on the sum of pairwise latencies: $\mathtt{lat}_{eq_0,eq_1} + \cdots + \mathtt{lat}_{eq_{n-2},eq_{n-1}} \sim b$.

For an `exists` constraint, the solver need only find a single propagation path that satisfies the constraints. For `forward` constraints, however, all propagation paths from the first equation in the chain must be considered. This is done by invoking the constraint generation function for each instance i of the first equation eq_0, and by anchoring the resulting, fresh first instance variable by an equality $\mathtt{i}_{eq_0} = i$. Similarly, for `backward` constraints, all propagation paths to the last equation in the chain must be considered. The function is invoked for each instance i of the last equation eq_{n-1}, and the resulting, fresh last instance variable is anchored by an equality $\mathtt{i}_{eq_{n-1}} = i$.

4 Code generation

The code generator takes as input (i) the original source program, (ii) a solution to the constraints from Section 3 that assigns a value to every $\mathtt{p}_{eq}$, and (iii) a parameter n_{s}, the number of step functions to generate, which must evenly divide the hyperperiod of all equations, that is, $n_{\mathsf{s}} \mid hp$. It recreates the flow graph (V, A) used to generate the constraints and then produces code comprising a static variable `c`, initialized to zero and incremented modulo hp at the end of every cycle, and n_{s} step functions, called over successive cycles in a repeating sequence. The code in Figure 3b shows the typical case where $n_{\mathsf{s}} = 1$.

A step function $step_i$ is generated for every $0 \leq i < n_{\mathsf{s}}$ and called in cycles where $\mathtt{c} \bmod n_{\mathsf{s}} = i$. The equations are filtered to determine which to include in $step_i$:

$$\mathsf{include}_i(eq) = \begin{cases} \mathtt{p}_{eq} \bmod n_{\mathsf{s}} = i & \text{if } n_{\mathsf{s}} \mid \text{period}(eq) \\ i \bmod \text{period}(eq) = \mathtt{p}_{eq} & \text{if period}(eq) \mid n_{\mathsf{s}} \\ \mathit{true} & \text{otherwise} \end{cases}$$

Consider, for example, $n_{\mathsf{s}} = 4$: for $\text{period}(a) = 8$ and $\mathtt{p}_a = 5$, the first case applies, and the equation a need only be included in $step_1$; for $\text{period}(b) = 2$ and $\mathtt{p}_b = 1$, the second case applies, and b need only be included in $step_1$ and $step_3$; for $\text{period}(c) = 3$, c must be included in all step functions regardless of its phase.

The equations within a $step_i$ are microscheduled according to their *instantaneous dependency graph* (V_i', A_i'), which is derived from the flow graph (V, A):

$$V_i' = \{w \mid w \in V \wedge \mathsf{include}_i(w)\} \quad A_i' = \{w \longrightarrow r \mid w \xrightarrow{s,\mathsf{f}} r \in A \wedge \mathsf{stogether}_i(w,r)\} \cup \{r \longrightarrow w \mid w \xrightarrow{s,\mathsf{b}} r \in A \wedge \mathsf{stogether}_i(w,r)\}\,.$$

That is, it only has vertexes for equations in $step_i$ and arcs for equations that may execute in the same instant; and arcs with backward concomitance become reversed dependencies. The predicate $\mathsf{stogether}_i(w,r)$ defines if a pair of communicating equations w and r sometimes execute together in $step_i$, recalling that either $\mathrm{period}(w) \mid \mathrm{period}(r)$ or $\mathrm{period}(r) \mid \mathrm{period}(w)$:

$$\mathsf{stogether}_i(w,r) = \mathsf{include}_i(w) \wedge \mathsf{include}_i(r) \wedge (\mathtt{p}_r \bmod \mathrm{period}(w) = \mathtt{p}_w \vee \mathtt{p}_w \bmod \mathrm{period}(r) = \mathtt{p}_r)\,.$$

For example, if $\mathrm{period}(w) = 3$, $\mathtt{p}_w = 2$, $\mathrm{period}(r) = 6$, and $\mathtt{p}_r = 3$, then even if the two equations are in the same step function, they will never be executed together in the same cycle. However, if $\mathtt{p}_r = 5$, then r executes together with every second execution of w.

A standard algorithm [4, §5] is adapted to generate code for each $step_i$. It orders the equations according to (V_i', A_i') and translates each eq into a guarded assignment statement:

`switch (c %` $\mathrm{period}(eq)$ `) {case` $\mathtt{p}_{eq}$`:` $\mathsf{TEq}(eq)$`;}`,

where TEq translates the equation. No guard is added if $\mathrm{period}(eq) \mid n_{\mathsf{s}}$, since it would always be true. As usual, a heuristic apposes equations with similar guards so that a subsequent *join* optimization can merge them to reduce branching. In our case, this means grouping equations with the same period where possible, and otherwise preferring equations with a greater harmonic period. The microscheduler is implemented so that equations are ordered by increasing period when the "fast-first" option is used.

5 Related work

5.1 Programming languages

Lustre [29] is a specification language with expressive activation conditions: arbitrary boolean expressions. These conditions are formalized in Lucid synchrone [7] as clock types, where a clock type is a sequence of variable names that abstract from the underlying boolean conditions. Various proposals have been made to restrict clock types to allow for specialized code generation. We present them in chronological order.

Periodic computations were specified in Signal [41] using *affine clocks* [55]. Abusing our notation, `base on` $(1\,\%\,4)$ is an affine clock since it is activated at instants $\{\frac{n}{4} + 1 \mid n \in \mathbb{N}_0\}$ relative to a base clock. The clock calculus is enriched with equational rules, for example, `base on` $1\,\%\,4$ `on` $0\,\%\,2$ = `base on` $1\,\%\,8$. More programs can then be accepted, since two signals that do not have the same clock expression may still be composed synchronously.

The *n-synchronous model* [10] also uses clocks to specify periodic computations, this time as ultimately periodic binary words, like 00(10100). Such clocks are more expressive than affine periods but deciding equality is costly since they must be expanded to their LCM. Clock abstraction [11] solves this problem by considering envelopes (sets of clocks) characterized by a rational slope (the period) and an interval (possibly of length zero) for the initial phase. The slope of the `on` operation is the product of its argument's slopes as in our clock rules. This proposal was implemented, without code generation, in Lucy-n [43, 52].

Prelude is a multi-periodic synchronous language [21, 23, 24] whose clock types combine a rational phase and rational period. Following [7], the clock calculus is a type system that conservatively extends the ML-like clock calculus [14] of Lucid synchrone [53]. Prelude's periodic clocks correspond to clock envelopes [11], but the absence of subtyping gives a simpler, more efficient, but also more restrictive calculus. For example, the expression $f(x \texttt{ when } (1 \,\%\, 4)) + g(x \texttt{ when } (0 \,\%\, 4))$ is accepted by Lucy-n but rejected by Prelude because the clocks $1 \,\%\, 4$ and $0 \,\%\, 4$ have different phases and thus are not equal. Like Prelude, we adopt a "relaxed synchrony hypothesis" [15]: each stream may have its own notion of instant [21, §3.1.2]. The rules in Figure 2b are essentially the same as Prelude's periodic clock transformations [24, §3.3]. Prelude's semantics [21, §4.6] is defined using the tagged-signal model [42] with rational timestamps. In our setting, integer indexes suffice. While our components must execute within a cycle, those of Prelude execute as multiple tasks [50] with fixed [22] or dynamic [26] priorities. Compared to an earlier proposition [33], we discard phases in clock types, reduce the number of operators, and generate sequential code directly.

Other work treats the compilation of component-based languages into real-time tasks. Caspi et al. [8] propose a dynamic buffering protocol for multi-rate synchronous programs, and Giotto [31] introduces Logical Execution Time (LET) [37] to decouple execution and communication. Hamann et al. [30] describe AUTOSAR, where *runnables*, which correspond to our equations, are grouped into periodic tasks and communicate via *direct*, *implicit*, or LET conventions. The static ordering within a task determines which communications are "forward" or "backward" [30, §4.3]. Ignoring execution time and preemption allows us to propose a simpler model, constrain end-to-end latency, and generate sequential code.

The discrete-time subset of Simulink is a multi-periodic programming language. *Sample times* [45, §7] are period/phase pairs that are subject to rules like those for our clock types. Rate transition blocks play the same role as our `when` and `current` operators. When both data integrity and determinism are required, the writer and reader rates must be integer multiples of one another [44, §1-1482]. While Simulink models can be compiled to Lustre [56], Simulink Coder [46] is normally used. Code generation flattens a model, groups blocks by sample rate giving priority to faster ones, and produces one or more tasks for scheduling.

5.2 Real-time scheduling and end-to-end Latency

End-to-end latency continues to be studied in the context of real-time systems. Gerber et al. [27] and Davare et al. [16] apply constraint solving to assign task periods to ensure that latency bounds are met. Gerber et al. consider chains with direct and fast-to-slow links, for which sums of pairwise latencies suffice (Section 3.4.1). Davare et al. define a coarse upper bound: the sum of task periods and response times (§2.1: $l_p = \sum_{k \in p} t_k + r_k$), with response times considering preemptive scheduling for an ECU and non-preemptive scheduling for a CAN bus. The linearity of the bound permits its use in constraint-based scheduling. A tighter upper bound is possible when task priority decreases from producers to consumers [17, 39] since this reduces interference from preemption and thus worst-case response times. Task periods can be excluded from the sum if each task in a chain activates its successor. These "trigger" [49], "functional" [28], or "active" [17] chains, where activation and data dependencies coincide, are studied, for example, by Schlatow and Ernst [54] and Girault et al. [28]. In contrast, our constraint-based scheduling treats sample-based activation, assigns offsets not periods, requires slicing a source program into small tasks to avoid preemption, and uses an encoding that precisely characterizes end-to-end latencies (Section 3.4.2).

For the flight control system described in the introduction, and, doubtless, other systems, upper bounds on end-to-end latency can be too pessimistic. Doing better requires considering how chains are instantiated in a concrete schedule. Feiertag et al. [20, Figure 7] clearly describe the importance of propagation paths through communicating job instances and considering the "branching and filtering" effect of mixing slow-to-fast and fast-to-slow communications.

Which jobs actually communicate depends on the read/write discipline. It can be formalized using read and data intervals [3] or classified into *direct*, *read-execute-write*, or *LET* [30] communications. Direct access to shared variables complicates analysis and may cause inconsistencies. For read-execute-write, a task samples inputs when it begins and writes outputs when it ends. For LET, such atomic reads and writes occur at fixed times, isolating them from the effects of varying response times, and giving predictable but greater end-to-end latencies [47]. These issues do not arise in our source language and compilation scheme, but we do need to consider the timing of communications within a step function ("microscheduling"). To analyze non-harmonic communications, Hamann et al. [30, §5.2.2.3] insert virtual copy operations, much as we require programmers to insert explicit ones.

Calculating the exact worst-case end-to-end latency of a set of tasks, as opposed to an upper bound, requires exploring all data propagation paths in all possible schedules. Mohalik et al. [48] apply model-checking techniques. Lauer et al. [40] apply Mixed Integer Linear Programming (MILP) and use optimization to calculate the maximum end-to-end latency; our equality constraints (Section 3.4.2) resemble their constraints on production and consumption [40, §3.1, (2) & (5)] but we avoid real-valued time stamps, handle propagation paths across multiple hypercycles, and constrain latency during scheduling. Khatib et al. [36] propose an algorithm based on translating multi-periodic real-time task sets into Synchronous Dataflow (SDF) graphs. The algorithms of Kloda et al. [39, §IV.C] and Becker et al. [3, §V] iterate over a hypercycle through the jobs of the first task in a chain and recursively explore data propagation paths. In our approach, a solver combines (offline) scheduling with end-to-end latency calculation, allowing the latter to constrain the former. The solution of Becker et al. [3, §VI] is to prune data propagation paths that would exceed a given latency bound by adding job-level dependencies. This approach is implemented in a tool [2], which Klaus et al. [38] have incorporated into a compilation chain. They note that cyclic dependencies can be problematic [38, §6.2]. For Prelude, Wyss et al. [58] calculate worst-case latencies at the source level by defining propagation paths as *data dependency words*. Forget et al. [25] generalize this approach in a formal language for expressing end-to-end timing properties.

6 Conclusion

We present a language for expressing execution rates and rate transitions in the synchronous-reactive model. It is a special case of Lustre where programmers allow scheduling to reconcile resource requirements and end-to-end latencies. Our flow graphs facilitate the treatment of cycles and the interaction of two scheduling passes, one using an ILP solver to assign tasks to phases and the other ordering tasks within a phase. Our novel encoding of end-to-end latency constraints, allowing unconstricted rate transitions, improves on the current practice of manually scheduling critical sequences. Finally, we generalize a standard compilation scheme to produce sequential code.

Acknowledgements. We thank Matthieu Boitrel, Michel Angot, and Jean Souyris for their collaboration, and Guillaume Iooss for his earlier work. We are grateful to our RTSS 2022 reviewers for rightly rejecting an earlier, over-complicated latency encoding and for their encouraging comments.

References

1 Ardupilot website. URL: `https://ardupilot.org`.

2 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. MECH-AniSer - a timing analysis and synthesis tool for multi-rate effect chains with job-level dependencies. In *7th Int. Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2016)*, July 2016.

3 Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. Synthesizing job-level dependencies for automotive multi-rate effect chains. In *Proc. of the 22nd IEEE Int. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA 2016)*, pages 159–169. IEEE, August 2016.

4 Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. of the 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130. ACM Press, June 2008.

5 Pascal Brisset, Antoine Drouin, Michel Gorraz, Pierre-Selim Huard, and Jeremy Tyler. The Paparazzi solution. In *2nd US-European Competition and Workshop on Micro Air Vehicles (MAV 2006)*, October 2006.

6 Giorgio C. Buttazzo. *Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, 3ed. edition, 2011.

7 Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proc. of the 1996 ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'96)*, pages 226–238. ACM Press, May 1996.

8 Paul Caspi, Norman Scaife, Christos Sofronis, and Stavros Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Transactions on Embedded Computing Systems*, 7(2):15:1–15:40, January 2009.

9 Houssine Chetto, Maryline Silly, and Bouchentouf Toumi. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2:181–194, September 1990.

10 Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous Kahn networks: a relaxed model of synchrony for real-time systems. In *Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006)*, pages 180–193. ACM Press, January 2006.

11 Albert Cohen, Louis Mandel, Florence Plateau, and Marc Pouzet. Abstraction of clocks in synchronous data-flow systems. In *Proc. of the 6th Asian Symposium on Programming Languages and Systems (APLAS 2008)*, volume 5356 of *Lecture Notes in Computer Science*, pages 237–254. Springer, December 2008.

12 Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proc. of the 5th ACM Int. Conf. on Embedded Software (EMSOFT 2005)*, pages 173–182. ACM Press, September 2005.

13 Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *Proc. of the 11th Int. Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15. IEEE Computer Society, September 2017.

14 Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Proc. of the 3rd Int. Workshop on Embedded Software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Computer Science*, pages 134–155. Springer, October 2003.

15 Adrian Curic. *Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints*. PhD thesis, Université Joseph Fourier, September 2005.

16 Abhijit Davare, Qi Zhu, Marco Di Natale, Claudio Pinello, Sri Kanajan, and Alberto Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *44th Design Automation Conf.*, pages 278–283. ACM/IEEE, June 2008.

17 Marco Dürr, Georg von der Brüggen, Kuan-Hsun Chen, and Jian-Jia Chen. End-to-end timing analysis of sporadic cause-effect chains in distributed systems. *ACM Transactions on Embedded Computing Systems*, 18(5s):article no. 58, October 2019.

18 Peter Eades and Xuemin Lin. A heuristic for the feedback arc set problem. *Australasian Journal of Combinatorics*, 12:15–25, September 1995.

19 Peter Eades, Xuemin Lin, and W.F. Smyth. A fast & effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, October 1993.

20 Nico Feiertag, Kai Richter, Johan Nordlander, and Jan Jonsson. A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2008, co-located with RTSS 2008)*, November 2008.

21 Julien Forget. *Un Langage Synchrone pour les Systèmes Embarqués Critiques Soumis à des Contraintes Temps Réel Multiples*. PhD thesis, Université de Toulouse, November 2009.

22 Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*, pages 301–310. IEEE, April 2010.

23 Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *Proc. of the 11th IEEE High Assurance Systems Engineering Symposium (HASE 2008)*, pages 251–260. IEEE, December 2008.

24 Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *Proc. of the 25th ACM Symposium on Applied Computing (SAC'10)*, pages 527–534. ACM, March 2010.

25 Julien Forget, Frédéric Boniol, and Claire Pagetti. Verifying end-to-end real-time constraints on multi-periodic models. In *Proc. of the IEEE 22nd Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2017)*. IEEE Press, September 2017.

26 Julien Forget, Emmanuel Grolleau, Claire Pagetti, and Pascal Richard. Dynamic priority scheduling of periodic tasks with extended precedences. In *Proc. of the IEEE 16th Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2011)*. IEEE Press, September 2011.

27 Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. *IEEE Transactions on Software Engineering*, 21(7):579–592, July 1995.

28 Alain Girault, Christophe Prévot, Sophie Quinton, Rafik Henia, and Nicolas Sordon. Improving and estimating the precision of bounds on the worst-case latency of task chains. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2578–2589, November 2018.

29 Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

30 Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, and Falk Wurst. Communication centric design in complex automotive embedded systems. In *29th Euromicro Conf. on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 10:1–10:20. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, June 2017.

31 Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. of the IEEE*, 91(1):84–99, January 2003.

32 IBM Corp. *IBM ILOG CPLEX 20.1 User's Manual*, 2020.

33 Guillaume Iooss, Albert Cohen, Dumitru Potop-Butucaru, Marc Pouzet, Vincent Bregeon, Jean Souyris, and Philippe Baufreton. Polyhedral scheduling and relaxation of synchronous reactive systems. In *12th Int. Workshop on Polyhedral Compilation Techniques (IMPACT 2022)*. HiPEAC, June 2022.

34 Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. of the Int. Federation for Information Processing (IFIP) Congress 1974*, pages 471–475. North-Holland, August 1974.

35 Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series (IRSS), pages 85–103. Springer, 1972.

36 Jad Khatib, Alix Munier-Kordon, Enagnon Cedric Klikpo, and Kods Trabelsi-Colibet. Computing latency of a real-time system modeled by Synchronous Dataflow Graph. In *Proc. of the 24th Int. Conf. on Real-Time Networks and Systems (RTNS'16)*, pages 87–96. ACM Press, October 2016.

37 Christoph M. Kirsch and Ana Sokolova. The Logical Execution Time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

38 Tobias Klaus, Florian Franzmann, Matthias Becker, and Peter Ulbrich. Data propagation delay constraints in multi-rate systems — deadlines vs. job-level dependencies. In *Proc. of the 26th Int. Conf. on Real-Time Networks and Systems (RTNS'18)*, pages 93–103. ACM Press, October 2018.

39 Tomasz Kloda, Antoine Bertout, and Yves Sorel. Latency analysis for data chains of real-time periodic tasks. In *Proc. of the IEEE 23rd Int. Conf. on Emerging Technologies & Factory Automation (ETFA 2018)*, pages 360–367. IEEE Press, September 2018.

40 Michaël Lauer, Frédéric Boniol, Claire Pagetti, and Jérôme Ermont. End-to-end latency and temporal consistency analysis in networked real-time systems. *Int. Journal of Critical Computer-Based Systems*, 5(3/4):172–196, 2014.

41 Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proc. of the IEEE*, 79(9):1321–1336, September 1991.

42 Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on CAD*, 17(12):1217–1229, December 1998.

43 Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n: a n-Synchronous extension of Lustre. In *Proc. of the 10th Int. Conf. on Mathematics of Program Construction (MPC 2010)*, volume 6120 of *Lecture Notes in Computer Science*, pages 288–309. Springer, June 2010.

44 The MathWorks. *Simulink® Reference*, March 2022. Release 2022a.

45 The MathWorks. *Simulink: User's Guide*, March 2022. Release 2022a.

46 The MathWorks. *Simulink® Coder™ User's Guide*, March 2022. Release 2022a.

47 Slobodan Matic and Thomas A. Henzinger. Trading end-to-end latency for composability. In *Proc. of the 26th IEEE Real-Time Systems Symposium (RTSS 2005)*, pages 110–119. IEEE Computer Society, December 2005.

48 Swarup Mohalik, Devesh B. Chokshi, Manoj G. Dixit, A.C. Rajeev, and S. Ramesh. Scalable model-checking for precise end-to-end latency computation. In *Proc. of the 2013 IEEE Conf. on Computer Aided Control System Design (CACSD)*, pages 19–24. IEEE, August 2013.

49 Saad Mubeen, Jukka Mäki-Turja, and Mikael Sjödin. Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study. *Computer Science and Information Systems*, 10(1):453–482, January 2013.

50 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, September 2011.

51 Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The ROSACE case study: From Simulink specification to multi/many-core execution. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014)*, pages 309–318. IEEE, April 2014.

52 Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire borné*. PhD thesis, Université Paris XI, January 2010.

53 Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, April 2006.

54 Johannes Schlatow and Rolf Ernst. Response-time analysis for task chains in communicating threads. In *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016)*. IEEE, April 2016.

55 Irina Mădălina Smarandache, Thierry Gautier, and Paul Le Guernic. Validation of mixed Signal-Alpha real-time systems through affine calculus on clock synchronisation constraints. In *Proc. of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1364–1383. Springer, September 1999.

56 Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 4(4):779–818, November 2005.

57 Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. A synchronous language with partial delay specification for real-time systems programming. In *Proc. of the 10th Asian Symposium on Programming Languages and Systems (APLAS 2012)*, volume 7705 of *Lecture Notes in Computer Science*, pages 223–238. Springer, December 2012.

58 Rémy Wyss, Frédéric Boniol, Julien Forget, and Claire Pagetti. End-to-end latency computation in a multi-periodic design. In *Proc. of the 28th ACM Symposium on Applied Computing (SAC'13)*, pages 1682–1687. ACM, March 2013.