HAL open science

Synchronous semantics of multi-mode multi-periodic systems

Frédéric Fort, Julien Forget

▶ To cite this version:

Frédéric Fort, Julien Forget. Synchronous semantics of multi-mode multi-periodic systems. 2022. hal-03817684v1

HAL Id: hal-03817684

https://hal.science/hal-03817684v1

Preprint submitted on 18 Jan 2022 (v1), last revised 17 Oct 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Synchronous semantics of multi-mode multi-periodic systems

(Prepublication - Final version to appear in the 37th ACM/SIGAPP Symposium On Applied Computing)

Frédéric Fort, Julien Forget
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
firstname.lastname@univ-lille.fr

***Abstract*—This paper tackles the problem of designing and programming a real-time system with multiple modes of execution, where each mode executes a different set of periodic tasks. The main problem to tackle is that the period of Mode Change Requests (MCR) and the period of tasks are not all the same. Thus, not all tasks perceive MCRs in the same way. When programming such a system with traditional languages without mechanisms dedicated to mode changes (e.g. C), it is difficult to ensure a system is sound and deterministic.**

We propose an extension to synchronous dataflow languages to support mode changes. The semantics of the resulting language is defined formally, which prevents ambiguous programs. The language is flexible enough to support different types of mode changes. The compiler of the language includes a static analysis that rejects programs whose semantics is ill-defined.

The extension consists in transposing Synchronous State Machines to the PRELUDE language. This requires to extend the semantics of PRELUDE, and to define a new clock calculus, based on refinement typing.

I. Introduction

In this paper, we are interested in the programming of critical real-time applications that exhibit a multi-mode behaviour. Real-time systems are typically programmed as a set of tasks executing periodically. In a *multi-periodic* system, tasks may have different periods. In a *multi-mode* system, each mode implements a different behaviour, characterised by a different set of tasks to execute. An aircraft control system is a typical example of multi-mode system with modes such as take-off, cruise, and landing.

Our goal is to propose a language to program this kind of multi-periodic multi-mode system. The language has sound formal semantics, it is sufficiently flexible to express different mode change protocols, and it abstracts from the underlying platform (OS scheduler and hardware platform).

A. Motivation

To motivate our work, let us detail the challenges that arise when programming a multi-periodic multi-mode real-time system.

First, since we consider multi-periodic systems, the period of Mode Change Requests (MCR) and the period of the different tasks are not all the same. Thus, not all tasks perceive MCRs in the same way. When using general-purpose languages, such as C, that do not contain constructs dedicated to the specification of mode changes, it is difficult to ensure that the behaviour of the system is unambiguous and deterministic. This leads us to the requirement:

Requirement 1: The language shall provide a sound formal semantics for multi-periodic mode change protocols.

In the real-time literature, different mode change protocols have been studied, each offering different advantages and disadvantages. For instance, a protocol might feature better reaction promptness to MCRs, but require a more complex schedulability analysis and vice-versa. However, no protocol clearly dominates the others. Thus, the choice of the protocol depends on the specificities of the system under design. This leads us to the following requirement:

Requirement 2: The language shall allow the system designer to choose their mode change protocols.

Because real-time systems are often critical, it is preferable to provide automated analyses that reject programs whose semantics is ill-defined. This avoids manual testing or, even worse, run-time errors. This leads us to the last requirement:

Requirement 3: The language compiler must reject programs whose soundness cannot be guaranteed.

B. Contribution

Our work consists in extending a synchronous language to support multi-mode multi-periodic systems. Synchronous languages [1] are well-adapted to the programming of critical real-time systems thanks to their clean formal semantics and to their formally defined compilation process. Synchronous State Machines have been proposed in [6] for LUSTRE [11] and LUCID SYNCHRONE [4], as a way to program multi-mode systems (each state corresponds to a mode). However, these languages do not support the specification of explicit real-time constraints (eg. periodicity). On the other hand, the language PRELUDE [16] provides constructs dedicated to the specification of such constraints, but does not support state machines.

With the state machines of [6], all flows within a state must have the same period, i.e. only mono-periodic states are allowed. Our objective in this paper is to transpose state machines to PRELUDE, and in doing so to extend them to support programs with multi-periodic states.

This extension relies on the notion of *clock views*, which allow us to decouple the period of a task from the period at which it perceives mode change requests. We provide a

formal semantics for this extended language, which allows us to satisfy Requirement 1. The resulting multi-mode support is generic and allows programmers to choose the kind of protocol they need for their application, satisfying Requirement 2. Periods and clock views are inferred and checked for consistency by the compiler during the *clock calculus*. The modified semantics requires us to completely change the type system and base it on refinement typing [10], [20]. The clock calculus rejects programs whose semantics is ill-defined, which satisfies Requirement 3. Overall, we believe that our approach prevents misinterpretations and ambiguities about when and how mode changes actually occur. To summarize, the contributions are:

- A formal semantics for an extension of PRELUDE with state machines (Section III);
- A clock calculus supporting these extensions (Section IV);
- An illustration of the capabilities of the extended language, showing the implementation of different Mode Change Protocols (Section V).

II. RELATED WORKS

Mode change protocols have been studied extensively by the real-time scheduling community (see [19] for a survey). However, these works focus on the timing analysis of the system, and do not consider the semantics of the corresponding program. A *Mode Change Protocol* defines how the transition from one mode to another is handled. A *Mode Change Request* (MCR) is an event that triggers a mode change. Mode change protocols surveyed in [19] can be classified according to three criteria:

- *Overlapping*[1]: when do the new-mode tasks start executing?
- *Periodicity*: are *unchanged tasks* impacted by mode changes?
- *Retirement*: what happens to *old-mode tasks* during a mode change?

In a *non-overlapping* protocol, the new-mode tasks are only released at the end of the mode transition phase. In an *overlapping* protocol, new-mode tasks and old-mode tasks can both execute during the transition phase. Overlapping protocols tend to have shorter transition times at the expense of requiring a more complicated schedulability analysis. *Periodic* protocols allow unchanged tasks, i.e. tasks which are present both in the old-mode and in the new-mode, to continue their execution uninterrupted. In *aperiodic* protocols, unchanged tasks are interrupted for the duration of the transition phase. *Late-retirement* protocols allow old-mode tasks to continue their execution for a given amount of time (for instance until they complete their current activation). *Early-retirement* protocols abort them as soon as the MCR is triggered.

Synchronous state machines [6] enable to specify multi-mode systems in a synchronous data-flow language. The original publication requires tasks within the same mode to have the same clock and implements a non-overlapping, periodic protocol with late retirement (tasks finish the current execution when a MCR occurs). The restriction on clocks was later lifted in [21]. However, the language does not support the specification of explicit real-time constraints such as periodicity constraints. PRELUDE [9] introduces mechanisms to explicitly specify real-time constraints in a synchronous program. Other languages require the programmer to express such constraints using boolean conditions, which do not allow to automatise real-time scheduling and schedulability analysis [3]. However, PRELUDE does not tackle the problem of combining real-time constraints with state machines.

Outside the synchronous languages community, several other languages dedicated to real-time systems have addressed the implementation of multi-mode systems. The mode change protocol available in AADL [2] allows tasks of different periods in the same mode, but suffers from long transition delays between modes. First, the system must wait for a duration equal to the least common multiple of the periods (the *hyperperiod*) of the old-mode task. Then, the mode transition begins and takes up to one hyperperiod of the new-mode tasks to complete. This is thus a non-overlapping, periodic protocol with late retirement (tasks execute until the synchronisation point). The mode change protocol of GIOTTO [12] requires MCRs to occur at multiples of the hyperperiod of the tasks affected by the mode change. This mode change protocol is also non-overlapping, periodic with late-retirement (tasks finish the current execution). Multi-mode systems are also at the core of the Statechart model. However, they suffer from poor formalisation, which leads to many different interpretations of program semantics [23]. All these languages opt for one specific mode change protocol. Instead, our model is more generic and allows to control the kind of protocol to use in a program.

The clock calculus of the present paper relies on refinement typing [10], [20], a typing discipline which extends classic Hindley-Milner type inference with type predicates. Predicates *refine* types allowing to more precisely specify which values an expression may contain. Refinement typing has been applied to the clock calculus of the SIGNAL language in [22]. However, the clock types used in this context do not enable the specification of real-time constraints.

III. LANGUAGE DEFINITION

In this section, we present an overview of our extension of the PRELUDE language. In particular, we detail how to extend the existing **`when`** and **`merge`** operators to support multi-periodic states. First, we present clocks, which define task rates, in Section III-A. Section III-B presents the *surface language* the programmer interacts with. Programs of the surface language are transpiled into the *core language*, presented in Section III-C. The semantics and the clock calculus of the core language are presented in Sections III-D and IV respectively.

[1] In [19], this property is called synchronicity. We renamed it to avoid confusion.

A. Clocks

1) Reminder: In the PRELUDE language, time is represented as a sequence of *instants*. *Clocks* specify the instants at which flows produce values. Clocks are modelled using the tagged-signal model [15], and tags represent the dates associated to instants. The operator . denotes concatenation on sequences. The $\prod$ operator extends the . operator over ranges, for instance $4.7.10.13 = \left(\prod_{i=0}^{2}(4+3*i)\right).13$.

A *flow* s is an infinite sequence of tagged values. The sequence of tagged values produced by s is denoted $s^{\#} = (v,t).s'^{\#}$. Its head (v,t) is composed of the value v produced at tag t and tail is $s'^{\#}$. Intuitively, the value v is the value carried by the flow until its next tag. Let $\widehat{s}$ denote the clock of flow s, i.e. its sequence of tags. Because tags provide a total ordering over value-tag pairs, we can use the (unordered) set-theoretic notation of a flow $s = \{(v,t) \mid (v,t) \in s^{\#}\}$. We denote (v_n, t_n) the n-th value (according to the tag ordering relation) v_n of a flow produced at its n-th tag t_n.

Timing constraints are specified using a specific class of clocks called *strictly periodic clocks*, defined below.

Definition 1 (Strictly Periodic Clock): A strictly periodic clock is denoted as a pair (n,p), with n, p in $\mathbb{N}$, and:

- The infinite sequence of tags generated by (n,p), denoted $(n,p)^{\#}$, is defined as follows: $(n,p)^{\#} = \{i*n+p \mid i \in \mathbb{N}\}$.
- $\pi((n,p)) = n$ is the *period* and $\varphi((n,p)) = p$ is the *offset* of (n,p).

The acceleration ($*.$), deceleration ($/.$), and delay ($\rightarrow .$) operators are defined below. These clock definitions are illustrated in Figure 1a (see end of Section III-A for a more detailed description).

Definition 2 (Periodic Clock Operators):

$$\pi(ck *. k) = \pi(ck)/k \qquad \varphi(ck *. k) = \varphi(ck)$$
$$\pi(ck /. k) = \pi(ck) * k \qquad \varphi(ck /. k) = \varphi(ck)$$
$$\pi(ck \rightarrow . k) = \pi(ck) \qquad \varphi(ck \rightarrow . k) = \varphi(ck) + k$$

In existing synchronous languages, and also in PRELUDE without our proposed extension, a clock ck **on** $C(c)$ denotes a clock ck that is sub-sampled on condition dataflow c. It produces a tag t, iff ck produces that tag t and dataflow c produces at t the value C. It is defined as follows:

Definition 3 (Mono-periodic conditional sub-sampling):

$$(ck \textbf{ on } C(c))^{\#} = \{t \mid t \in ck^{\#} \wedge (C,t) \in c^{\#}\}$$

Throughout this paper, we will denote divisibility constraints using the relation $x \textbf{ div } y \Leftrightarrow y \bmod x = 0$, which reads "$x$ divides y".

2) Our extension: Clock views: In a multi-periodic context such as the present paper, we want to extend the definition of ck **on** $C(c)$ to allow ck and c to have different periods. To this intent, we introduce the concept of *clock views*. The clock ck **on** $C(c,w)$ denotes the clock ck sub-sampled on condition c, such that it produces tags only if c, perceived according to view w, produces a value C. A view is a clock that specifies the rate at which the condition is observed. The semantics of the **on** operator with views is defined as follows.

Definition 4 (Sub-sampling with clock views): A multi-periodic conditional clock is denoted ck **on** $C(c,w)$ where ck is a clock, C is a constant, c is a condition dataflow, and w is a strictly periodic clock called *view*.

- The infinite sequence of tags generated by ck **on** $C(c,w)$, denoted $(ck \textbf{ on } C(c,w))^{\#}$, is defined as follows:
$$(ck \textbf{ on } C(c,(n,p)))^{\#} = \{t \mid t \in ck^{\#}, \exists (C,t'') \in c^{\#}, t' \in (n,p)^{\#} \wedge t' \leq t < t' + n \wedge t'' = t' + \varphi(\widehat{c}) - p\}$$
- A view is valid iff $\pi(ck) \textbf{ div } \pi(w) \wedge \pi(\widehat{c}) \textbf{ div } \pi(w)$
- Extending the previous notation, we have $\pi(ck \textbf{ on } C(c,w)) = \pi(ck)$ and $\varphi(ck \textbf{ on } C(c,w)) = \varphi(ck)$. A conditionally sub-sampled clock has the same period and offset as its base clock (ck), but in addition filters tags according to a condition.

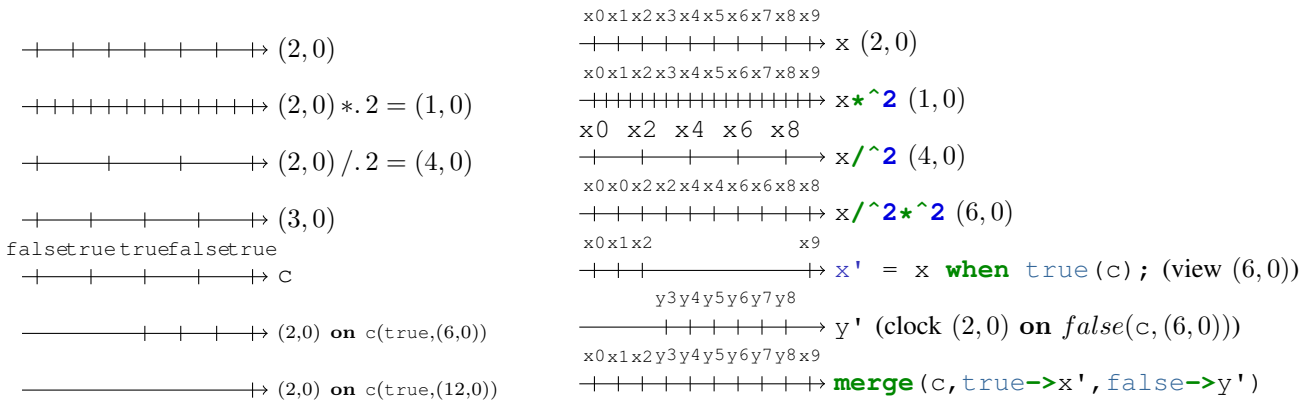
Intuitively, the view w delimits intervals where only a single value produced by c is responsible for all tags produced by ck within that interval. If that value equals C, then ck **on** $C(c,w)$ produces all tags within that interval. Otherwise, it produces no tag within that interval. The requirement on the view period is needed so that applying a rate-transition operator (Section III-D) always results in a dataflow with a view that is a strictly periodic clock.

Figure 1a illustrates the clocks described in this section. The first 4 timelines represent the instants at which different clocks produce tags. Below, we have the timeline for a condition dataflow `c` with clock $(3,0)$. The two timelines below show different sub-samplings of $(2,0)$ by condition `c`. They differ only in their views. The first one observes `c` according to view $(6,0)$ and the second one according to view $(12,0)$. Note that the first one produces tags at instants 6, 8 and 10 because `c` is `true` at instant 6. The fact that `c` is `false` at instant 9 is ignored because the view $(6,0)$ considers `c` only at tags that are multiples of 6. For the same reason, the `true` value of `c` at instant 3 is ignored. The figure shows that changing the view $((2,0) \textbf{ on } true(c,(12,0)))$ produces a different set of tags.

B. The surface language

As in [6], we distinguish between the surface language the programmer uses and the core language for which a formal semantics is defined. During the compilation process, the compiler transpiles the program in the surface language to an equivalent program in the core language. Being smaller than the surface language, formalizing the core language is simpler.

Figure 2 details the extended PRELUDE syntax considered in this paper. Figure 4a illustrates a program written according to this syntax and we briefly summarize those rules below. A program is structured in nodes, the synchronous languages equivalent to functions. An imported node lifts a function over scalar values from the target language (e.g. C) to flows by performing a point-wise application of the function. User-specified nodes define via definitions how local and output variables are computed from input variables. A definition is



(a) Clocks

(b) Dataflows

Fig. 1: Clocks and dataflows

either an equation or an automaton (composed of further definitions). An equation of the form `x = e;` defines variable `x` as equal to expression `e`. Note that equations are unordered. An expression can be a variable (`x`), a constant (`42`), a node application (`f(a,b)`) or the result of the application of one of the following built-in operators (assuming `i` has clock ck):

- `i/^k` keeps one out of k successive values of `i` and has clock $ck\,/.\,k$;
- `i*^k` repeats each value of `i`, k times, and has clock $ck *. k$;
- `i~>k` delays each value of `i` by k and has clock $ck \rightarrow .\,k$;
- `k fby i` produces value `k` followed by the values of `i` and has clock ck, effectively delaying values of `i` by $\pi(ck)$
- `i when true(c)` sub-samples `i` such that it produces values only if `c` produces `true`. It has clock ck **on** $true(c, w)$. Note that the view is not specified by the program, but instead inferred by the compiler;
- `merge(c, true->i_true, false->j_false)` combines the complementary flows `i_true` and `i_false` with respective clocks ck **on** $true(c, w)$ and ck **on** $false(c, w)$ and has clock ck.

We assume that all node inputs have rate annotations (e.g. `rate (10,0)`), i.e. their periods and offsets are always specified in the program. We believe that this restriction has little impact on the ability to express realistic real-time systems, even though it prevents users from defining intermediate polymorphic nodes. Lifting this restriction is left for future work.

C. The core language

The link between the surface and core language is done via a translation semantics. Compared to the surface language, the core language features two important distinctions:

- Automata are replaced by equations with the same semantics;

$$
\begin{array}{rcl}
\langle prog\rangle & ::= & \langle decl\rangle * \\
\langle decl\rangle & ::= & \langle nd\rangle \mid \langle ind\rangle \\
\langle ind\rangle & ::= & \texttt{imported node}\ \langle id\rangle\,(\langle var\rangle +)\ \texttt{returns}\ (\langle var\rangle +) \\
\langle nd\rangle & ::= & \texttt{node}\ \langle id\rangle\,(\langle var\rangle +)\ \texttt{returns}\ (\langle var\rangle +) \\
 & & (\texttt{var}\ \langle var\rangle +\,\texttt{;})?\ \texttt{let}\ \langle def\rangle +\,\texttt{tel} \\
\langle var\rangle & ::= & \langle id\rangle\,\texttt{:}\,\langle ty\rangle ?\langle ck\rangle ? \\
\langle ty\rangle & ::= & \texttt{int} \mid \texttt{real} \mid \texttt{bool} \\
\langle ck\rangle & ::= & \texttt{rate}\ (\langle int\rangle,\ \langle int\rangle) \mid \langle ck\rangle\ \texttt{on}\ \langle id\rangle\,(\langle id\rangle) \\
\langle def\rangle & ::= & \langle auto\rangle \mid \langle eq\rangle \\
\langle auto\rangle & ::= & \texttt{automaton}\ \langle state\rangle + \\
\langle state\rangle & ::= & \mid\ \langle id\rangle\ \texttt{->}\ (\texttt{var}\ \langle var\rangle +)?\ \langle st\rangle\ \langle def\rangle +\ \langle wt\rangle \\
\langle st\rangle & ::= & \texttt{unless}\ \langle expr\rangle\ \texttt{then}\ \langle id\rangle\texttt{;} \\
\langle wt\rangle & ::= & \texttt{until}\ \langle expr\rangle\ \texttt{then}\ \langle id\rangle\texttt{;} \\
\langle eq\rangle & ::= & \langle id\rangle(,\ \langle id\rangle) * = \langle expr\rangle \\
\langle expr\rangle & ::= & \langle id\rangle \mid \langle const\rangle \mid \langle id\rangle\ \ (\langle expr\rangle(,\ \langle expr\rangle) *\,) \\
 & & \mid \langle expr\rangle\ \ \texttt{/\^{}}\ \ \langle int\rangle \mid \langle expr\rangle\ \ \texttt{*\^{}}\ \ \langle int\rangle \mid \langle expr\rangle\ \ \texttt{\~{}>}\ \ \langle int\rangle \\
 & & \mid \langle const\rangle\ \ \texttt{fby}\ \ \langle expr\rangle \mid \langle expr\rangle\ \ \texttt{when}\ \ \langle id\rangle\,(\langle id\rangle) \\
 & & \mid \texttt{merge}\,(\langle id\rangle(,\ \langle id\rangle\texttt{->}\langle expr\rangle) +\,)
\end{array}
$$

Fig. 2: The surface language syntax

- Expressions are in Administrative Normal Form (ANF), i.e. arguments of node and operator applications are either variables or constants.

The automata flattening process is detailed in [6]. Our translation process remains the same because we simply extend the semantics of the `when` operator. We provide an intuitive overview of [6] below to make the paper more self-contained.

Within an automaton, we distinguish local from non-local variables. A local variable is defined within a specific state, while a non-local variable is available within all states of an automaton. In Figure 4, variables `pos` and `GPS` are non-local, while `controls` is local to state `Actuate`. The main steps of the translation are:

- A new type is introduced, the variants of which are the (potentially mangled) names of the automaton states. For an automaton with states `StateA` and `StateB`, this would introduce a type `auto0_state` with variants `StateA` and `StateB`;
- We introduce state variables, which represent the current automaton state. Transitions update these variables as appropriate;
- For each state, we *project* equations into that automaton state, i.e. we apply a **when** to non-local variables and mangle the left-hand side variables to state-specific ones. For instance, an equation `x = f(i);` in state `StateA` becomes `StateA_x = f(i when StateA(state))` where `state` is the current-state variable;
- Non-local equations are merged. For instance, we would find the following equation `x = merge(state, StateA-> StateA_x, StateB->StateB_x);`

Once all automata are flattened, we transform equations into ANF. This transformation is straightforward. If the argument of a node or operator application is not an atom (a variable or a constant), we introduce a new equation with that argument. We substitute the original argument by the left-hand side of our fresh equation. Then, we recursively call that procedure on the right-hand side of our fresh equation. For instance, when transpiling the equation `x = (i when true(c))*^2/^3;` we obtain the following equations:

```
x = v0/^3;  v0 = v1*^2;  v1 = i when true(c);
```

Once the program has been transpiled to the core language and all static analyses have been performed, the compiler proceeds to translating the program into a low-level language (e.g. C). This compilation step is out of the scope of this paper. In [16], a complete compilation process for PRELUDE without our extended **when** and **merge** is proposed. In future work we will adapt the proposed node communication protocols to take into account clock views.

D. Synchronous Kahn semantics

In this section, we present the semantics of the core language based on synchronous Kahn networks [13], [5]. The term $\diamond^{\#}(s_0, \ldots, s_n)$ denotes the flow resulting from the application of the operator $\diamond$ on flows $s_0, \ldots, s_n$. Operators fall into three categories: imported operators, rate-transition operators and conditional operators. The semantics for the first two categories remains as previously defined in [9] and is recalled in Figure 3. The figure also details the semantics of our extended conditional operators (op_f denotes an operator over scalars from the compiler target language). The $\texttt{when}^{\#}$ operator is a direct transposition of the **on** operator on flows. The $\texttt{merge}^{\#}$ combines flows that have complementary clocks. Examples are provided in Figure 1b.

Note that $\texttt{merge}^{\#}$ is deterministic iff the merged flows are *complementary*, i.e. only one is present at any given instant. Similarly, the imported node application requires arguments that are *synchronous*, i.e. that have the same clock. Also, $*^{\wedge\#}(s, k)$ is defined iff k **div** $\pi(\widehat{s})$. Finally, the semantics

$$
\begin{aligned}
op^{\#}(s_0, \ldots, s_n) &= \{(op_f(v_0, \ldots, v_n), t) \mid \\
&\qquad (v_0, t) \in s_0^{\#}, \ldots, (v_n, t) \in s_n^{\#}\} \\
*^{\wedge\#}(s, k) &= \{(v, t + i * \pi(\widehat{s})/k) \mid (v, t) \in s^{\#}, i \in [\![0..k[\![\ \} \\
/^{\wedge\#}(s, k) &= \{(v, t) \mid (v, t) \in s^{\#} \wedge t \in (\widehat{s}/.\,k)^{\#}\} \\
\sim>^{\#}(s, k) &= \{(v, t + k) \mid (v, t) \in s^{\#}\} \\
\texttt{fby}^{\#}(v, s) &= \{(v, t_0)\} \cup \{(v_i, t_{i+1}) \mid \\
&\qquad (v_i, t_i), (v_{i+1}, t_{i+1}) \in s^{\#}\} \\
\texttt{when}^{\#}(s, c, C, w) &= \{(v, t) \mid (v, t) \in s, t \in (\widehat{s}\ \mathbf{on}\ C(c, w))^{\#}\} \\
\texttt{merge}^{\#}(c, s_0, \ldots, s_n) &= \bigcup_{i=0}^{n} s_i^{\#}
\end{aligned}
$$

Fig. 3: Kahn semantics of operators

for $\texttt{when}^{\#}$ and $\texttt{merge}^{\#}$ require clock views, which are not specified by the program. Instead, they are inferred by the compiler. The *clock calculus*, defined in the next section, is responsible for checking clock constraints, and inferring the clocks and views of the program. Thus, the semantics of a program is well-defined only if the clock calculus succeeds.

E. Illustrative example

In this section, we illustrate the presented language. Figure 4 presents an implementation of the control software of a large sized Unmanned Aerial Vehicle (UAV) from [14], [12]. Figure 4a shows the corresponding program written with our extended version of PRELUDE using Synchronous State Machines. The system perceives its environment via a GPS and an Inertial Navigation System (INS). In addition, it receives via a wireless communication an enabling signal specifying in which mode it shall execute (`isEnabled`) and a destination point (`waypoint`). The system actuates via servo motors. We perform amplitude saturation on the computed instructions for the servo motors. This ceils the difference between two consecutive instructions, protecting the servo motors from extreme variations.

The application has two modes. In the `Estimate` mode, the UAV preserves its previous course and measurements serve only to update the UAV position. In the `Actuate` mode, the UAV computes orders for the servo motors so as to reach the current waypoint. The automaton switches from mode `Estimate` to mode `Actuate` when expression `isEnabled` is true, and from `Actuate` to `Estimate` when **not** `(isEnabled)` is true. As a consequence, Mode Change Requests are emitted with clock $(10, 0)$.

Due to the dataflow nature of the language, the period of nodes (e.g. `h_f`, `control`, `servo_driver`) is determined by the period of their inputs. For instance `control` has clock $(10, 0)$ **on** `Actuate(state,` $(10, 0)$`)`, while `servo_driver` has clock $(20, 0)$ **on** `Actuate (state,` $(20, 0)$`)`, even though both nodes are executed within the mode `Actuate`. This difference in views means that task `control` will respond to a change of `state` immediately, while `servo_driver` can in some cases respond with a

delay of 10 time units. Thus, this automaton implements an *overlapping mode change protocol*, i.e. nodes are not all impacted simultaneously by a mode change.

Figure 4b shows an excerpt of the transpilation into the core language. To improve readability, we renamed the identifiers generated by the compiler. The automaton state is defined by the variable `state`, which has clock $(10, 0)$. For a state S (either `Estimate` or `Actuate`), MCRs are emitted by the equation `s_S` which has clock $(10, 0)$ **on** $S(\texttt{state}, (10, 0))$. An MCR instantly updates the value of the variable `state`. However this state change is observed by tasks depending on their view as seen above.

Let us detail how the equations of servos are translated. For each state equation (e.g. `srvs = servos_driver(controls/^2);` in state `Actuate`), we replace it by a state-specific equation, (e.g. the equation for `Actuate_srvs`). State-specific dataflows (such as `Actuate_srvs`, `Estimate_srvs`) are then merged together (e.g. `srvs = merge(state, Estimate->Estimate_srvs, Actuate->Actuate_srvs);`), producing a flow with clock $(20, 0)$. To respect the dataflow-nature of the language, state non-local variables (e.g. `GPS`) are projected on the automaton state they are used in (e.g. `GPS when Actuate(state)`).

IV. Clock Calculus

The previous implementation of the Prelude clock calculus proceeds by Hindley-Milner type inference extended with subtyping [9]. In order to support our extended **on** clock operator, we rely on a clock calculus based on refinement typing [10], [20].

Let us first briefly illustrate a refinement typing system. In such a type system, types may be ascribed with predicates. For instance the expression `4` would have type $\{\nu{:}\texttt{int} \mid \nu = 4\}$, meaning "an `int` whose value is equal to 4". The type `int` is called the *base type*, the variable ν represents the value of the typed expression and "$\nu = 4$" is called the refinement. The refined function `(/)` would have type $a{:}\texttt{int} \to b{:}\{\nu{:}\texttt{int} \mid \nu \neq 0\} \to \{\nu{:}\texttt{int} \mid \nu = a/b\}$, meaning "a function taking an argument a of type int and an argument b of type int whose value is not equal to 0, returning an int whose value is equal to a/b". In our clock calculus, we use linear integer arithmetic predicates and rely on the Z3 SMT solver [7] to check the satisfiability of these predicates.

Throughout this section, we will illustrate our clock calculus via the running example in Figure 5. Node `main` has two inputs, `i` and `c` with different periods. The local variable `iwc` holds the dataflow of conditionally sub-sampling `i` to only produce values when `c` produces `true`. The output `o` is the result of delaying `iwc` by 50 time units.

A. Clock language

The goal of our clock language is to describe the clocks of Section III-A using refinement types. A full definition can be found in Figure 6. For instance, the clock $(3, 0)$ is described by the clock type $\{\nu{:}\texttt{pck} \mid \pi(\nu) = 3 \wedge \varphi(\nu) = 0\}$. The base type here is `pck`, i.e. a strictly periodic clock. The refinement states that the clock period is 3 and the clock offset is 0. For brevity's sake, we will write such types as $\{\nu{:}\texttt{pck} \mid \langle 3, 0\rangle\}$.

Definition 5: The clock type $\{\nu{:}ck_b \mid \langle n, p\rangle\}$ is a shorthand for the clock type $\{\nu{:}ck_b \mid r_n \wedge r_p\}$ where

$$r_n \equiv \pi(\nu) = n \qquad\qquad r_p \equiv \varphi(\nu) = p$$

The clock $(3, 0)$ **on** $true(c, (15, 0))$ is described by the clock type $\{\nu{:}\texttt{pck}$ **on** $true(c, \{\nu{:}\texttt{pck} \mid \langle 15, 0\rangle\}) \mid \langle 3, 0\rangle\}$. Again, the refinement $(\langle 3, 0\rangle)$ specifies the clock period and offset. The condition is found inside the base type (**on** $true(c, \ldots)$). The view of the condition is itself specified as a periodic (refined) clock $(\{\nu{:}\texttt{pck} \mid \langle 15, 0\rangle\})$.

In a functional clock $x{:}ck_r \to ck_e$, x is the called the *input binder*, ck_r is the *input type* and ck_e is the *output type*.

B. Overview

In this section, we provide on overview of the different passes of the clock calculus, which are:

1) Structural Clock Calculus
2) Refinement Clock Calculus
   a) Refinement Instantiation
   b) Refinement Checking
3) View Closing

In the *Structural Clock Calculus*, only the structure of clocks is inferred. This pass is very similar to Hindley-Milner typing, except that clocks are annotated with *refinement holes*, i.e. refinement placeholders. In the second pass, the *Refinement Clock Calculus*, we verify the actual refinements. We divide this pass into two steps: *Refinement Instantiation* and *Refinement Inference*. In the Refinement Instantiation step, we bridge the gap between the structural clocks and the refinement clocks. In the Refinement Inference step, the actual typing decisions are performed. Finally, we delay view computations until the last point, the *View Closing* pass. Before that, views only collect constraints without checking them.

Inference rules use *bi-directional typing* [18], [8]. *Synthesis judgements* $H \vdash e \Rightarrow t$ signify that in environment H, the type t is associated to expression e. *Checking judgements* $H \vdash e \Leftarrow t'$ signify that in environment H, the type t' is valid for expression e (even though e might be associated to a different type t). The link between these two judgements is provided in Definition 6 below. To check an expression e against a type t, first a type t' has to be synthesised and t' has to be a subtype of t. The subtyping relation $t' <: t$ is defined for refinement types as follows: a type t' is a subtype of type t, iff the base type of t' is a subtype of the base type of t and the refinement of t' implies the refinement of t. For instance, if we want to check that $\{\nu{:}\texttt{int} \mid \nu = 4\} <: \{\nu{:}\texttt{int} \mid \nu \neq 0\}$, we have to prove $\forall \nu{:}\texttt{int}.\ (\nu = 4) \implies (\nu \neq 0)$.

Definition 6 (Core refinement typing rules):

$$\textsc{Chk-Syn}\ \frac{H \vdash e \Rightarrow t' \qquad t' <: t}{H \vdash e \Leftarrow t} \qquad \textsc{Sub-Ref}\ \frac{t <: t' \qquad \forall \nu{:}t.\ r_0 \implies r_1}{\{\nu{:}t \mid r_0\} <: \{\nu{:}t' \mid r_1\}}$$

```
node main(GPS: GPSMessage rate(10,0); INS : INSMessage rate(10,0);
          isEnabled: bool rate(10,0); waypoint: real[4] rate(10,0))
returns(servos: ServoMessage)
var pos, srvs;
let
  servos = saturate(srvs, 0 fby servos);
  automaton
  | Estimate ->
    unless isEnabled then Actuate;
    var GPS_f, INS_f, pos_f;
    GPS_f,INS_f,pos_f = h_f(GPS, INS, init_pos fby* pos);
    pos = filter(GPS_f, INS_f, pos_f);
    srvs = init_servos fby* srvs;

  | Actuate ->
    unless not(isEnabled) then Estimate;
    var GPS_c, INS_c, pos_c,
        waypoint_c, controls;
    GPS_c,INS_c,pos_c,waypoint_c =
      h_c(GPS, INS, init_pos fby* pos, waypoint);
    pos,controls = control(GPS_c, INS_c, pos_c, waypoint_c);
    srvs = servo_driver(controls/^2);
  end
tel
```

(a) The case-study in the surface language

```
node main(GPS: GPSMessage rate(10,0); INS : INSMessage rate(10,0);
          isEnabled: bool rate(10,0); waypoint: real[4] rate(10,0))
returns(servos: ServoMessage)
var state, previous_state, s_Estimate, s_Actuate, ...;
let
  previous_state = Estimate fby state;
  state = merge(previous_state, Estimate->s_Estimate, Actuate->s_Actuate);

  s_Estimate = if Estimate_isEnabled then Actuate else Estimate;
  s_Actuate = if Actuate_not_isEnabled then Estimate else Actuate;

  srvs = merge(state, Estimate->Estimate_srvs, Actuate->Actuate_srvs);

  Estimate_isEnabled = isEnabled when Estimate(previous_next_state);
  Actuate_not_isEnabled = not(Actuate_isEnabled);
  Actuate_isEnabled = isEnabled when Actuate(previous_next_state);

  Estimate_srvs = srvs_fby when Estimate(state);
  srvs_fby = init_servos fby srvs;

  GPS_c, INS_c, pos_c, waypoint_c =
    h_c(Actuate_GPS, Actuate_INS, Actuate_waypoint_fby);
  Actuate_GPS = GPS when Actuate(state);
  Actuate_INS = INS when Actuate(state);
  Actuate_waypoint_fby = waypoint_fby when Actuate(state);
  waypoint_fby = init_pos fby waypoint;
  Actuate_pos, controls = control(GPS_c, INS_c, pos_c, waypoint_c);
  controls_div_2 = controls/^2;
  Actuate_srvs = servo_driver(controls_div2);
  ...
tel
```

(b) The case-study in the core language (excerpt)

Fig. 4: A multi-mode multi-rate synchronous program

```
node main(i: rate(100,0); c: rate(200,0)) returns (o)
var iwc;
let
  iwc = i when true(c);
  o = iwc ~>50;
tel
```

Fig. 5: The running example

$$
\begin{array}{lcl}
\sigma & ::= & \forall \alpha.\sigma \mid ck_e \\
ck_e & ::= & x{:}ck_r \to ck_e \mid ck_e \times ck_e \mid ck_r \\
ck_r & ::= & \{\nu{:}ck_b \mid r\} \\
ck_b & ::= & \texttt{pck} \mid \alpha \mid ck_b \ \mathbf{on}\ C(c, ck_r) \\
r & ::= & r \wedge r \mid p = a \mid a \ \mathbf{div}\ p \mid p \geq a \mid true \\
p & ::= & \pi(\nu) \mid \varphi(\nu) \\
a & ::= & k \mid \pi(x) \mid \varphi(x) \mid a + a \mid a - a \mid a * k \mid a/k \\
H & ::= & \emptyset \mid H; x{:}\sigma
\end{array}
$$

x : Variable $\quad k$: Constant

Fig. 6: Clock system

C. Structural Clock Calculus

The goal of this pass is to infer the structure of clocks, that is to say clock types where refinements are left unknown and represented by variables. This means in particular that clock conditions are inferred during this pass, while periods and offsets are inferred during the following pass, i.e. the refinement clock calculus. To differentiate between judgements of both clock calculi, we denote judgements of the Structural Clock Calculus with an S, e.g. $H \vdash e \overset{S}{\Rightarrow} ck$ and $H \vdash e \overset{S}{\Leftarrow} ck$. We detail the main principles of the structural clock calculus on our running example:

- Concerning node variables, clock annotations dictate clock structures and refinements are represented by fresh refinement holes. So for instance, `i` has clock $\{\nu{:}\texttt{pck} \mid \star_0\}$, `c` has clock $\{\nu{:}\texttt{pck} \mid \star_1\}$ (both variables are declared with strictly periodic clocks), while `iwc` has clock $\{\nu{:}\alpha \mid \star_2\}$;
- When typing an operator or a node application, first we instantiate its type with fresh refinement holes. So for instance, for operator **when** we get clock $e{:}\{\nu{:}\epsilon \mid \star_4\} \to c{:}\{\nu{:}\epsilon \mid \star_5\} \to \{\nu{:}\epsilon \ \mathbf{on}\ true(c, w'') \mid \star_6\}$ where $w'' = \{\nu{:}\texttt{pck} \mid \star_7\}$. Note that since all refinements are different, this implies that the operands of a **when** can have different periodic clocks;
- The second step of the application consists in checking arguments against input types. In our example, this implies that ϵ is substituted by `pck`, so expression `i` **when** `true(c)` gets clock $\{\nu{:}\texttt{pck} \ \mathbf{on}\ true(c, w) \mid \star_8\}$;
- Concerning equations, left and right-hand sides must synthesise to the same clock, so `iwc` and `i` **when** `true(c)` both get clock $\{\nu{:}\texttt{pck} \ \mathbf{on}\ true(c, w) \mid \star_2\}$;

The complete results for our running example are detailed in Table I. A full definition of structural inference rules can be found in Appendix A.

D. Refinement Clock Calculus

In the *Refinement Clock Calculus* pass, clock refinements are inferred in place of refinement holes. Types structures are considered fixed, only refinements may change. A full definition of inference rules can be found in Appendix A.

1) Refinement Instantiation: In this step, we bridge the gap between structural clocks and refinement clocks. Some refinements are initially known: refinements corresponding to clock annotations, refinements of nodes typed previously, and refinements of predefined operators. Refinement Instantiation

Item	After
i	$\{\nu$:pck $\mid \star_0\}$
c	$\{\nu$:pck $\mid \star_1\}$
iwc i **when** true(c)	$\{\nu$:pck **on** $true(c,w) \mid \star_2\}$
o iwc ~>50	$\{\nu$:pck **on** $true(c,w') \mid \star_3\}$
w	$\{\nu$:pck $\mid \star_9\}$
w'	$\{\nu$:pck $\mid \star_{10}\}$

TABLE I: The typing environment after the structural clock calculus

consists in injecting these refinements into structural types. We detail the main principles of this step on our running example:

- Concerning variables, we inject refinements related to clock annotations declared in the program (if any). For instance, for i and c we instantiate their refinements to $\{\nu$:pck $\mid \langle 100, 0\rangle\}$ and $\{\nu$:pck $\mid \langle 200, 0\rangle\}$;
- Concerning operator or node applications, in our example the structural clock of the **when** is e:$\{\nu$:pck $\mid \star_4\} \rightarrow c$:$\{\nu$:pck $\mid \star_5\} \rightarrow \{\nu$:pck **on** $true(c, w'') \mid \star_6\}$ where $w'' = \{\nu$:pck $\mid \star_7\}$. After instantiation, we obtain the following refinements: e:$\{\nu$:pck $\mid true\} \rightarrow c$:$\{\nu$:pck $\mid \langle n, \varphi(e)\rangle\} \rightarrow$ $\{\nu$:pck **on** $true(c, w'') \mid \langle \pi(e), \varphi(e)\rangle\}$ where $w'' = \{\nu$:pck $\mid \langle n, \varphi(e)\rangle \wedge \pi(e)$ **div** $\pi(\nu) \wedge \pi(c)$ **div** $\pi(\nu)\}$. These refinements impose that: 1) inputs and outputs all have the same offset ($\varphi(e)$); 2) e and c can have different periods; 3) the output has the same period as e; 4) the period of the view must be divisible both by the periods of e and c.

2) Expression Refinement: With our operators refined, we can proceed to type expressions and equations. The objective of this pass is to check the consistency of the refinements of all expressions inside a node. The calculus infers the constraints that the refinements must satisfy. The set of constraints is then submitted to the SMT solver for resolution. We detail the main principles of this pass on the **when** application in our running example:

- First, arguments are type checked against input types. Type checking for i succeeds trivially and substitutes i for e in the clock of **when**;
- Type checking for c must then check the following judgement: $H \vdash$ c$\Leftarrow\{\nu$:pck $\mid \langle n, \varphi(i)\rangle\}$. This is verified trivially by the solver because both variables have offset 0;
- Finally, the type of the result of the application is the output type where input binders are substituted by the actuals: $\{\nu$:pck **on** $true(c, w) \mid \langle 100, 0\rangle\}$ where $w = \{\nu$:pck $\mid \langle n, \varphi(i)\rangle \wedge \pi(i)$ **div** $\pi(\nu) \wedge \pi(c)$ **div** $\pi(\nu)\}$. Note that the period and offset of the view are still unknown at this point.

E. View Closing

The final pass, the view closing pass, computes the period and offset of views. Constraints of the form $\pi(i)$ **div** $\pi(\nu)$ are nonlinear, i.e. they belong to an undecidable fragment of integer arithmetic. However, since we postponed view resolution, the periods and offsets of flows are all solved (constants). Furthermore, we perform a constant propagation step before sending constraints to the solver. This yields a system of constraints with few variables, which can rather easily be solved by the solver heuristics (despite the undecidability of the theoretical problem). For instance, on our running example:

- Before closing, the views are

$$w = \{\nu\text{:pck} \mid \langle n, \varphi(i)\rangle \wedge \pi(i) \textbf{ div } \pi(\nu) \wedge \pi(c) \textbf{ div } \pi(\nu)\}$$
$$w' = \{\nu\text{:pck} \mid \langle \pi(w), \varphi(w) + 50\rangle\}$$

- Performing constant propagation yields:

$$w = \{\nu\text{:pck} \mid \langle n, 0\rangle \wedge 100 \textbf{ div } \pi(\nu) \wedge 200 \textbf{ div } \pi(\nu)\}$$
$$w' = \{\nu\text{:pck} \mid \langle \pi(w), \varphi(w) + 50\rangle\}$$

- This results in the following request to the SMT solver:

$$100 \textbf{ div } w_{period} \wedge 200 \textbf{ div } w_{period} \wedge w_{offset} = 0 \wedge$$
$$w'_{period} = w_{period} \wedge w'_{offset} = w_{offset} + 50$$
$$minimize(w_{period}, w'_{period})$$

- The solution returned by the solver is:

$$w = \{\nu\text{:pck} \mid \langle 200, 0\rangle\}$$
$$w' = \{\nu\text{:pck} \mid \langle 200, 50\rangle\}$$

V. Evaluation

Comparing our implementation of the clock calculus with the implementation of [9], our clock calculus requires around 800 additional lines of OCaml code. In addition, we observe a noticeable but still reasonable overhead in compilation time. For instance, the compilation time of the ROSACE case study [17] increases from $10ms$ to $50ms$ (including the constraints resolution time of Z3).

In the rest of this section, we illustrate the capabilities of the extended language through examples showing the implementation of different mode change protocols. Recall the criteria of [19] by which a mode change protocol can be classified:

- *Overlapping*: when do the *new-mode tasks* start executing?
- *Periodicity*: are *unchanged tasks* impacted by mode changes?
- *Retirement*: what happens to *old-mode tasks* during a mode change?

a) Unchanged tasks: In our work, unchanged tasks correspond to dataflows computed outside the automaton. For instance, in Figure 4, saturate is an unchanged task.

b) Retirement: In our work, old-mode tasks continue their execution until their views perceive the state change. Since the period of the view cannot be shorter that the period of the task, this implies that we cannot implement early-retirement protocols. Supporting early-retirement would require to interrupt a task during its execution, which raises serious semantics concerns in a dataflow context.

Item	Refinement Clock Calculus	View closing
`i`	$\{\nu{:}\texttt{pck} \mid \langle 100, 0\rangle\}$	
`c`	$\{\nu{:}\texttt{pck} \mid \langle 200, 0\rangle\}$	
`iwc`	$\{\nu{:}\texttt{pck}\ \textbf{on}\ true(c, w) \mid \langle 100, 0\rangle\}$	
`i when true(c)`		
`o`	$\{\nu{:}\texttt{pck}\ \textbf{on}\ true(c, w') \mid \langle 100, 50\rangle\}$	
`iwc ~>50`		
w	$\{\nu{:}\texttt{pck} \mid \langle n, \varphi(i)\rangle \wedge \pi(i)\ \textbf{div}\ \pi(\nu) \wedge \pi(c)\ \textbf{div}\ \pi(\nu)\}$	$\{\nu{:}\texttt{pck} \mid \langle 200, 0\rangle\}$
w'	$\{\nu{:}\texttt{pck} \mid \langle \pi(w), \varphi(w) + 50\rangle\}$	$\{\nu{:}\texttt{pck} \mid \langle 200, 50\rangle\}$

TABLE II: The typing environment during the refinement clock calculus

```
@@ -4,8 +4,9 @@
-var pos;
+var pos, isEnabled_slow;
  let
   servos = saturate(srvs, 0 fby servos)
+  isEnabled_slow = isEnabled/^2;
   automaton
   | Estimate ->
-    unless isEnabled then Actuate;
+    unless isEnabled_slow then Actuate;
@@ -14,15 +15,16 @@
   | Actuate ->
-    unless isEnabled then Estimate;
+    unless isEnabled_slow then Actuate;
```

Fig. 7: Changes for a non-overlapping

c) Periodicity: Our work only supports periodic protocols. Indeed, the execution of a node is triggered by the arrival of its inputs, it cannot be interrupted once it starts processing its inputs. As for early-retirement, supporting aperiodicity is antagonistic with dataflow semantics.

d) Overlapping: We can implement both overlapping and non-overlapping protocols. For instance, Figure 4 implements an overlapping protocol. Nodes `filter` and `control`, which compute `pos`, have the same view, $(10, 0)$. Thus, during a transition from one state to another, there is no overlap between nodes `filter` and `control`. However, in case of a transition from state `Actuate` to state `Estimate`, there can be an overlap, since `filter` (from the new mode) and `servo_driver` (from the old mode) may co-exist (because `servo_driver` has view $(20, 0)$).

To change the automaton protocol into a non-overlapping protocol, one must change the program such that all nodes share the same view. One possibility is to slow down the period of transitions. Figure 7 shows the changes required. We define a new dataflow `isEnabled_slow`, a down-sampled version of `isEnabled`, and replace it everywhere inside the automaton. Now, `state` has clock $(20, 0)$ and the views of all expressions inside the automaton become $(20, 0)$. Thus, the automaton implements a non-overlapping mode change protocol.

These examples illustrate the benefit of separating the execution rate of a flow (its strictly periodic clock) from the rate at which it perceives mode change requests (its state transition view). This allows us to reason about mode change protocols, and avoids misinterpretations and ambiguities (Requirement 1). Examples also demonstrate the flexibility of the language (Requirement 2).

VI. Conclusion

We defined an extension for synchronous dataflow languages to support applications with multiple modes of execution and tasks of different periods within the same mode. The extension allows the programmer to implement custom mode change protocols best adapted to the application. We defined a formal semantics for these extensions and a clock calculus based on refinement typing. In future works, we plan to extend the clock calculus to support nodes with polymorphic clocks.

References

[1] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] Dominique Bertrand, Anne-Marie Déplanche, Sébastien Faucou, and Olivier H Roux. A study of the aadl mode change protocol. In *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 288–293. IEEE, 2008.

[3] Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

[4] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Synchronous functional programming: The lucid synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools. Hermes*, pages 28–41, 2008.

[5] Paul Caspi and Marc Pouzet. Synchronous kahn networks. *ACM SIGPLAN Notices*, 31(6):226–238, 1996.

[6] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, page 173182, New York, NY, USA, 2005. Association for Computing Machinery.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5), May 2021.

[9] Julien Forget, Frdric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *2008 11th IEEE High Assurance Systems Engineering Symposium*, pages 251–260, 2008.

[10] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 268277, New York, NY, USA, 1991. Association for Computing Machinery.

[11] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

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

[13] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[14] H.Jin Kim and David H. Shim. A flight control system for aerial robots: algorithms and experiments. *Control Engineering Practice*, 11(12):1389–1400, 2003. Award winning applications-2002 IFAC World Congress.

[15] Edward A Lee and Alberto Sangiovanni-Vincentelli. Comparing models of computation. In *Proceedings of International Conference on Computer Aided Design*, pages 234–241. IEEE, 1996.

[16] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete event dynamic systems*, 21(3):307–338, 2011.

[17] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. The rosace case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 309–318. IEEE, 2014.

[18] Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[19] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-time systems*, 26(2):161–197, 2004.

[20] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159169, June 2008.

[21] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier, and Abdoulaye Gamatié. Polychronous mode automata. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 83–92, 2006.

[22] Jean-Pierre Talpin, Pierre Jouvelot, and Sandeep Kumar Shukla. Towards refinement types for time-dependent data-flow networks. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 36–41. IEEE, 2015.

[23] Michael Von der Beeck. A comparison of statecharts variants. In *Formal techniques in real-time and fault-tolerant systems*, pages 128–148. Springer, 1994.

Appendix

VAR
$$H; x{:}ck \vdash x \overset{S}{\Rightarrow} ck$$

CST
$$H \vdash c \overset{S}{\Rightarrow} ck$$

APPL
$$\frac{H \vdash N \overset{S}{\Rightarrow} x{:}ck_b \to ck'_e \qquad H \vdash e \overset{S}{\Leftarrow} ck_b \qquad ck_e = fresh(ck'_e)}{H \vdash N(e) \overset{S}{\Rightarrow} ck_e}$$

$$\frac{H \vdash x \overset{S}{\Rightarrow} ck \qquad H \vdash e \overset{S}{\Rightarrow} ck}{H \vdash x = e} \qquad \texttt{pck} \overset{S}{<:} \texttt{pck} \qquad \alpha \overset{S}{<:} \alpha$$

$$\frac{ck_b \overset{S}{<:} ck_b \qquad w \overset{S}{<:} w'}{ck_b \textbf{ on } C(c,w) \overset{S}{<:} ck'_b \textbf{ on } C(c,w')}$$

$$\frac{ck_b \overset{S}{<:} ck'_b}{\{\nu{:}ck_b \mid \star\} \overset{S}{<:} \{\nu{:}ck'_b \mid \star'\}}$$

$$\begin{aligned}
fresh(x{:}ck_r \to ck_e) &= x{:}fresh(ck_r) \to fresh(ck_e)\\
fresh(ck_e^0 \times ck_e^1) &= fresh(ck_e^0) \times fresh(ck_e^1)\\
fresh(\{\nu{:}ck_b \mid \star_n\} &= \{\nu{:}fresh(ck_b) \mid \texttt{next()}\}\\
fresh(ck_b \textbf{ on } C(c,w)) &= fresh(ck_b) \textbf{ on } C(c, \{\nu{:}\texttt{pck} \mid fresh(w)\})\\
fresh(\texttt{pck}) &= \texttt{pck}\\
fresh(\alpha) &= \alpha\\
\texttt{next()} &= \star_i \text{ where } i \text{ is unused previously}
\end{aligned}$$

$$\begin{aligned}
\texttt{e } \textbf{when } \texttt{C(c)} &: \forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to c{:}\{\nu{:}\alpha \mid \star_1\} \to \{\nu{:}\alpha \textbf{ on } C(c, \{\nu{:}\texttt{pck} \mid \star_2\}) \mid \star_3\}\\
\textbf{merge}\texttt{(c, C0->e0, C1->e1)} &: \forall \alpha.c{:}\{\nu{:}\alpha \mid \star_0\} \to e_0{:}\{\nu{:}\alpha \textbf{ on } C0(c,w) \mid \star_1\} \to e_1{:}\{\nu{:}\alpha \textbf{ on } C1(c,w) \mid \star_2\} \to \{\nu{:}\alpha \mid \star_3\}\\
\texttt{e } \textbf{*^ k} &: \forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\alpha \mid \star_1\}\\
\texttt{e } \textbf{/^ k} &: \forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\alpha \mid \star_1\}\\
\textbf{c fby } \texttt{e} &: \forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\alpha \mid \star_1\}\\
\texttt{e } \textbf{~> k} &: \forall \alpha.e{:}\{\nu{:}\alpha \mid \star_0\} \to \{\nu{:}\alpha \mid \star_1\}
\end{aligned}$$

VAR
$$H; x{:}\sigma \vdash x \Rightarrow \sigma$$

CST
$$H \vdash c \Rightarrow \sigma$$

APPL
$$\frac{H \vdash f \overset{S}{\Rightarrow} \sigma \qquad x{:}ck_r \to ck_e = inst(f,\sigma) \qquad H \vdash a \Leftarrow ck_r}{H \vdash f(a) \Rightarrow ck_e[x := a]}$$

CHK-SYN
$$\frac{H \vdash e \Rightarrow \sigma' \qquad \sigma' <: \sigma}{H \vdash e \Leftarrow \sigma}$$

SUB-REF
$$\frac{ck_b <: ck'_b \qquad \forall \nu{:}t.\ r_0 \implies r_1}{\{\nu{:}ck_b \mid r_0\} <: \{\nu{:}ck'_b \mid r_1\}}$$

SUB-PCK
$$\texttt{pck} <: \texttt{pck}$$

SUB-VAR
$$\alpha <: \alpha$$

SUB-ON
$$\frac{ck_b <: ck'_b \qquad w <: w'}{ck_b \textbf{ on } C(c,w) <: ck'_b \textbf{ on } C(c,w')}$$

$$\begin{aligned}
&inst(\star\texttt{^}k, e{:}\{\nu{:}ck_b \mid \star_0\} \to \{\nu{:}ck'_b \mid \star_1\}) =\\
&e{:}\{\nu{:}ck''_b \mid k \textbf{ div } \pi(\nu)\} \to \{\nu{:}ck'''_b \mid \langle \pi(e)/k, \varphi(e)\rangle\}\\
&\text{where } ck''_b, ck'''_b = inst^2_b(ck_b, ck'_b, true, \lambda x.\langle \pi(x), \varphi(x)\rangle)
\end{aligned}$$

$$\begin{aligned}
&inst(/\texttt{^}k, e{:}\{\nu{:}ck_b \mid \star_0\} \to \{\nu{:}ck'_b \mid \star_1\}) =\\
&e{:}\{\nu{:}ck''_b \mid true\} \to \{\nu{:}ck'''_b \mid \langle \pi(e) * k, \varphi(e)\rangle\}\\
&\text{where } ck''_b, ck'''_b = inst^2_b(ck_b, ck'_b, \pi(e) * k \textbf{ div } \pi(\nu),\\
&\lambda x.\pi(x) \textbf{ div } \pi(\nu) \wedge \pi(e) * k \textbf{ div } \pi(\nu) \wedge \varphi(\nu) = \varphi(x))
\end{aligned}$$

$$\begin{aligned}
&inst(\texttt{fby}, e{:}\{\nu{:}ck_b \mid \star_0\} \to \{\nu{:}ck'_b \mid \star_1\}) =\\
&e{:}\{\nu{:}ck''_b \mid true\} \to \{\nu{:}ck'''_b \mid \langle \pi(e), \varphi(e)\rangle\}\\
&\text{where } ck''_b, ck'''_b = inst^2_b(ck_b, ck'_b, true, \lambda x.\langle \pi(x), \varphi(x)\rangle)
\end{aligned}$$

$$
\begin{gathered}
inst(\tilde{}{>}k, e{:}\{\nu{:}ck_b \mid \star_0\} \to \{\nu{:}ck'_b \mid \star_1\}) = \\
e{:}\{\nu{:}ck''_b \mid k \ \mathbf{div}\ \pi(\nu)\} \to \{\nu{:}ck'''_b \mid \langle \pi(e), \varphi(e) + k\rangle\} \\
\text{where } ck''_b, ck'''_b = inst^2_b(ck_b, ck'_b, true, \langle \pi(x), \varphi(x) + k\rangle)
\end{gathered}
$$

$$
\begin{gathered}
inst(\texttt{when}, e{:}\{\nu{:}ck_b \mid \star_0\} \to c{:}\{\nu{:}ck_b \mid \star_1\} \to \\
\{\nu{:}ck_b \ \mathbf{on}\ C(c, w) \mid \star_2\}) = \\
e{:}\{\nu{:}ck'_b \mid true\} \to c{:}\{\nu{:}ck'_b \mid \langle n, \varphi(e)\rangle\} \to \\
\{\nu{:}ck'_b \ \mathbf{on}\ C(c, w') \mid \langle \pi(e), \varphi(e)\rangle\} \\
\text{where } ck'_b = inst_b(ck_b, true) \\
w = \{\nu{:}\texttt{pck} \mid \langle n, \varphi(e)\rangle \wedge \pi(e)\ \mathbf{div}\ \pi(\nu) \wedge \pi(c)\ \mathbf{div}\ \pi(\nu)\}
\end{gathered}
$$

$$
\begin{gathered}
inst(\texttt{merge}, c{:}\{\nu{:}ck_b \mid \star_0\} \to e0{:}\{\nu{:}ck_b \ \mathbf{on}\ C_0(c, w) \mid \star_1\} \to \\
e1{:}\{\nu{:}ck_b \ \mathbf{on}\ C_1(c, w) \mid \star_2\} \to \{\nu{:}ck_b \mid \star_3\}) = \\
c{:}\{\nu{:}ck'_b \mid true\} \to e0{:}\{\nu{:}ck'_b \ \mathbf{on}\ C_0(c, w') \mid true\} \to \\
e1{:}\{\nu{:}ck'_b \ \mathbf{on}\ C_1(c, w') \mid \langle \pi(e_0), \varphi(e_0)\rangle\} \to \{\nu{:}ck'_b \mid \langle \pi(e_0), \varphi(e_0)\rangle\} \\
\text{where } ck'_b = inst_b(ck_b, true) \\
w' = \{\nu{:}\texttt{pck} \mid true\}
\end{gathered}
$$

$$
\begin{aligned}
inst_b(\texttt{pck}, _) &= \texttt{pck} \\
inst_b(\alpha, _) &= \alpha \\
inst_b(ck_b \ \mathbf{on}\ C(c, w), r) &= inst_b(ck_b, r) \ \mathbf{on}\ C(c, \{\nu{:}\texttt{pck} \mid r\})
\end{aligned}
$$

$$
\begin{aligned}
inst^2_b(\texttt{pck}, \texttt{pck}, _, _) &= \texttt{pck}, \texttt{pck} \\
inst^2_b(\alpha, \alpha, _, _) &= \alpha, \alpha \\
inst^2_b(ck_b \ \mathbf{on}\ C(c, w), ck'_b \ \mathbf{on}\ C(c, w'), r, \lambda x.r') &= \\
ck''_b \ \mathbf{on}\ C(c, w''),\ ck'''_b \ \mathbf{on}\ C(c, w''') & \\
\text{where } ck''_b, ck'''_b = inst_b(ck_b, ck'_b, r, \lambda x.r') & \\
w &= \{\nu{:}\texttt{pck} \mid \star_0\} \\
w' &= \{\nu{:}\texttt{pck} \mid \star_1\} \\
w'' &= \{\nu{:}\texttt{pck} \mid r\} \\
w''' &= \{\nu{:}\texttt{pck} \mid (\lambda x.r')w''\}
\end{aligned}
$$