A Multi-Rate Precision Timed Programming Language for Multi-Cores

Alain Girault, Nicolas Hili, Éric Jenn, Eugene Yip

A Multi-Rate Precision Timed Programming Language for Multi-Cores

Alain Girault*, Nicolas Hili†, Eric Jenn†‡, and Eugene Yip§
* Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France alain.girault@inria.fr
† IRT Saint-Exupéry, 31432, Toulouse, France, first.last@irt-saintexupery.com
‡ Thales AVS, 105 Avenue du Général Eisenhower F31036 Toulouse, France, eric.jenn@fr.thalesgroup.com
§ Software Technologies Research Group, University of Bamberg, 96045, Germany, eugene.yip@uni-bamberg.de

***Abstract*—PREcision Timed (PRET) is a conceptual solution proposed in 2007 to address the ever increasing unpredictability of embedded processors, which results from features such as multi-level caches or deep pipelines. For many real-time systems, it is mandatory to compute a strict bound on the program's execution time. Yet, in general, computing a tight bound is extremely difficult. The rationale of PRET is to simplify both the programming language and the execution platform to allow precise execution times to be easily computed.**

ForeC is a PRET programming language. It is a multi-threaded variant of C with a synchronous execution semantics. ForeC programs are designed to be executed on multi-core processors, built around either PRET cores or classical cores. A drawback of ForeC is that programs are *single rate*, i.e., all reactions must be implemented to run at the fastest rate imposed by the environment. This represents a high overhead, both at design time and at run-time.

In this paper, we propose a multi-rate version of ForeC to improve its practicality and usability for industrial acceptance. We detail the syntax and semantics of the ForeC language in the context of multi-rate applications and present an implementation on a PRET multi-core architecture. Both the languages and its implementation are illustrated over a robotic application.

***Index Terms*—Multi-core, Precision timed, Synchronous Languages, Multi-rate, Real-time**

I. Introduction

PREcision Timed (PRET) programming languages and architectures have been proposed in 2007 with the seminal paper from Edwards and Lee [1], to alleviate the overwhelming efforts required in computing the Worst-Case Reaction Time (WCRT) of a program. The key idea of PRET is to remove all sources of uncertainties from the architecture (e.g., cache, branch prediction, and out-of-order execution) and to provide a dedicated programming language in order to achieve *predictable* execution [2]. By predictable, we mean that not only the computed WCRT will be more accurate (i.e., the difference between the computed WCRT and the actual WCRT will be small), but also that the computed WCRT will be close to the Average-Case Reaction Time (ACRT). With general purpose processors, achieving this is close to impossible, due to the over-approximations necessary to provide a guaranteed WCRT [3] in presence of the architecture features mentioned above (caches and so on), which all share the common goal of improving the ACRT.

Since 2007, several other programming languages and architectures have been proposed, notably ARPRET [4], Flex-PRET [5] for the architectures, and PRET-C [6], [7] and ForeC [8]–[10] for the programming languages. In short, PRET-C and ForeC are an extension of C with high-level concurrency and timing instructions inspired from Esterel [11], [12]. Esterel, PRET-C, and ForeC are examples of synchronous languages, which provide an abstract, ideal, and (mathematically) tractable view of time that allows programmers to focus on the logical timing of computations, rather than their physical timing due to implementation choices. With this notion of logical time, the programming, compilation, debugging, and verification of systems is greatly simplified.

Esterel and PRET-C are multi-threaded, but their parallelism is "compiled away" by the compiler, which computes an interleaving of the threads compatible with the control dependencies expressed in the program, and finally produces sequential code meant to be executed on single-core processors (just like Esterel's compiler). In this sense, the multi-threading offered by PRET-C is a parallelism of expression. In contrast, ForeC programs are truly multi-threaded and are meant to be executed on multi-core processors.

ForeC is missing the ability to design applications that run at different *rates*, a feature needed to address most non-trivial control systems that mix frequent—but short—computations with infrequent—but long—executions. Multi-rate applications could be emulated by splitting long computations into smaller chunks that fit within the period of the unique rate. However, this solution is impractical because it tightly couples the structural and temporal aspects of the design. In practice, this means that a structural modification implies a temporal modification, and vice-versa, and should clearly be avoided. Moreover, the computation's logical behaviour is altered because its inputs and outputs are updated too early at the boundaries of each chunk. The solution is to integrate the ability to handle multiple rates into the programming language.

Several synchronous programming languages do offer this possibility, either as high-level language constructs, e.g., Lustre++ [13], Prelude [14], Multi-Clock Esterel [15], Simulink [16], or Giotto [17] (the last two are not strictly synchronous languages, but still relevant), or by automatic parallelization tools, e.g., [18] or [19]. The Berkeley PRET

This work has been partially supported by the French Research Agency (ANR) and by the industrial partners of IRT Saint-Exupéry Scientific Cooperation Foundation (FCS). We wish to thank the anonymous reviewers for their valuable feedback.

language also offers this possibility but with low-level instructions (the deadline instruction, noted `deadi`). The goal of this paper is to bring multi-rate support to ForeC, such that the semantics of Multi-Rate ForeC is backwards compatible with classical ForeC, i.e., a classical ForeC program is simply a Multi-Rate ForeC program with all rates being equal. An in-depth description of classical ForeC is given in [10].

The outline of the paper is as follows. In Section II we introduce shortly the ForeC predictable programming language. In Section III we discuss in detail the support for multiple rates in ForeC programs–our core contribution of the paper–and Section IV presents an implementation. Finally, we conclude and propose future work directions in Section V.

II. ForeC in a Nutshell

A ForeC program is composed of a set of threads that are forked during the execution of `par` statements. The thread that executes the `par` statement is the *parent* of the *child* threads created by the statement. Threads communicate with each other via variables. Variables can be of four kinds. Input (resp. output) variables are variables sampled from (resp. emitted to) the environment. They are declared at the top-level of the program and are accessible by all threads. Variables declared with the `shared` keyword are shared between a parent thread and its (nested) children. Regular C variables can also be used. A ForeC program is composed of two files, a `forec` file that contains the program (e.g., declaration of the global variables and definition of the threads) and a `foreh` file that contains deployment data (e.g., declaration of the hardware platform and mapping of the threads to the platform's cores).

A ForeC program executes at the cadence of the *global tick*. At the beginning of the global tick, all input variables are sampled from the environment. Then, each thread executes its *local tick*, which consists in: (1) creating local copies of shared variables it will access, (2) performing its operations, and (3) propagating its copies of shared variables to its parent thread. Operations are performed on local copies of shared variables to avoid concurrent accesses and race conditions. If multiple threads update the same shared variable, a *combination* function must be specified to combine the values computed by the threads. A combination function must be associative and commutative, so that the combination is order-independent (this mechanism is inspired from Esterel's combine functions). The `par` statement is used to fork-join threads. A thread's local tick ends when it reaches a `pause` statement. A global tick ends when all local ticks end, at which point the output variables are emitted to the environment.

Fig. 1 depicts a sequence of global ticks in the case of two ForeC threads `t1` and `t2`. Each blue part is the *body* of a tick;

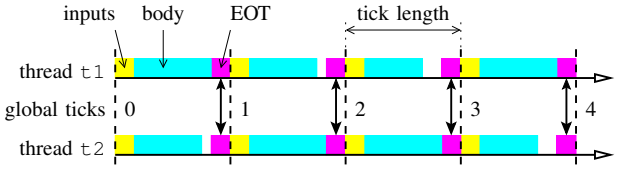

Fig. 1: The sequence of global ticks.

the computations of each thread. Each yellow part samples the *inputs* from the environment and updates the input global variables of the thread accordingly. Finally, each magenta part is the *end of tick (EOT)*, where the shared variables are resolved by calling their *combine function* and propagating their resolution back to the threads that use them, and the output global variables are emitted to the environment.

Any implementation of a ForeC program that conforms to the ForeC semantics [10] ensures that all observable input/output behaviours are invariant to implementation choices, e.g., thread scheduling or thread-to-core mapping. The way ForeC threads are actually executed in parallel depends on the chosen *hardware platform*. ForeC currently supports three architectures: two bare metal platforms (Xilinx multi-core MicroBlaze [20] and PTARM single-core multi-threaded PRET [21]), and x86 platforms implementing POSIX threads. When static timing analysis is of interest, the programmer can provide a static *thread-to-core* mapping to specify how ForeC threads are deployed on the architecture. The term *core* can designate a physical core (e.g., of the MicroBlaze architecture), or physical and logical threads (for PTARM and x86).

Running Example: Autonomous Robot

We demonstrate the concepts of multi-rate on a robotic demonstrator whose primary function is to guide visitors from the reception desk to the office of their host or to a meeting room. Given the map of the building and a mission specified as a set of waypoints, the robot's main objective is to move autonomously from its initial location to its target location.

Fig. 2 shows the main functions of the robot. Green circles denote sensors. Green rectangles denote internal functions and have no direct interface with the external world. Purple rectangles denote functions that interface directly with the external world via sensors and actuators, and have short reaction times of 10 μs. Each function output is annotated with the rate at which new data should be produced. The dashed rectangles delimit the functions that were considered for this paper.

For simplicity, we assume that the robot moves around on a plane. A localization function (LOC) is responsible for collecting information about the pose of the robot from various sources (odometry, IMU sensor, ultra-wideband (UWB) radio-localization device, marker-based positioning device), and to compute the actual pose (x,y,θ) of the robot. The odometry function provides odometry data. From the robot's position, the next target position (called *set point*) is computed by the target generation (TG) function, and the linear and rotational speed command (v, w) of the robot is elaborated by the tracking function (TRA). A speed limiter function (SPL) limits the robot's speed based on the safety monitoring of the robot's position (POM), attitude (ATM), acceleration (ACM) for collision detection, and speed (SPM). The robot can also adapt its speed when an obstacle is detected (functions ATG, LIM, OBD, SPC, and TM). Finally, the motor control (MOC) function translates the speed command $(v, w)_{SPL}$ from the speed limiter function into duty cycles (dc_L, dc_R) for the

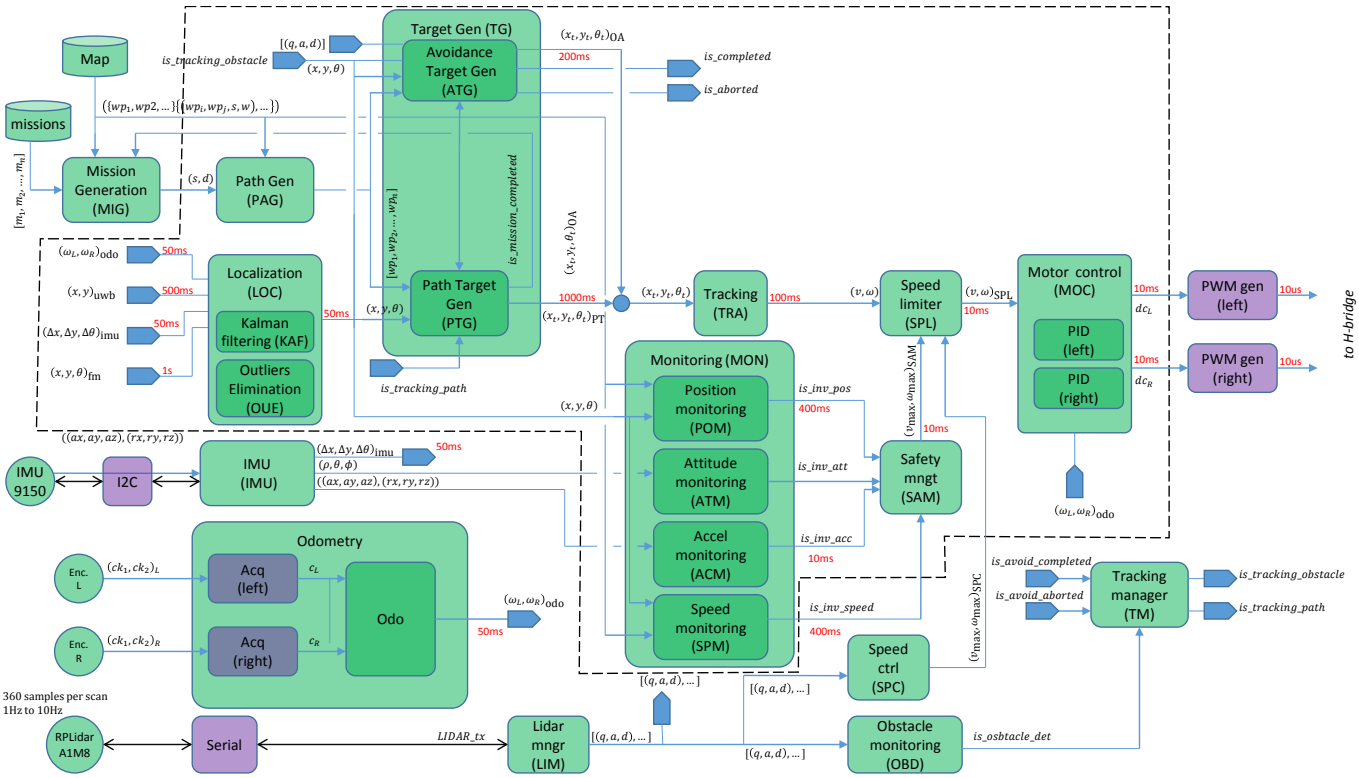

Fig. 2: Robot case study's main functions.

Pulse-Width Modulation (PWM) signals that will drive the wheel's two motors.

This example shows different cases where multi-rate is needed. First, the period of each function may depend on the actual computation of the function, or on timing constraints from the sensors and actuators the system uses to monitor its surrounding environment. For example, the period of a function collecting data from a sensor depends on the update rates of the sensor and not on the actual computation time the function requires. Various sensors and actuators may have very different timing constraints (e.g., the marker-based positioning device has an update rate of $1s$ while a PWM signal is updated every $10\,\mu s$). Multi-rate threads is a solution to deal with these strict and inherently heterogeneous timing constraints.

Second, the rate at which a function needs to update its outputs actually depends on functional and safety constraints. For example, the attitude computed by the ATM function (to detect falls) must be updated faster than the position monitoring issued by the POM function (in order to detect that the robot does not deviate from its trajectory).

To illustrate the problem of having a mono-rate ForeC program, consider the `forec` file in Listing 1 that has the top-level thread `main` with two operating modes. Under normal mode, the robot adjusts its linear and angular speed to follow a path. When a safety monitoring flag is raised, the `abort` statement is triggered, forcing the child threads to terminate and join. The robot enters the *safety mode* and the motors are stopped ($\mathrm{dc_L}$=0; $\mathrm{dc_R}$=0;) until the two safety monitoring flags are cleared.

In mono-rate ForeC, the top-level thread `main` and all threads forked by a `par` statement must execute at the same rate, i.e., the rate of the slowest or fastest thread. Two problems arise if the slowest rate (e.g., thread TG) is chosen: first, when the robot is tracking an obstacle, any new avoidance set point produced by the avoidance target gen (ATG) thread would only be read by the tracking (TRA) thread every $1,000\,ms$ instead of $200\,ms$. Second, when a safety monitoring flag is raised, the time needed to abort the execution and to stop the motors is incompatible with the update rate of each motor's PWM signal. If the fastest rate is chosen, the slower threads would contribute a higher execution load. In the sequel, we discuss the formalization and implementation of Multi-Rate ForeC.

III. Multi-Rate Support

Multi-rate ForeC extends ForeC [9] with *periodic* clocks that we call *rates* (so as not to make the confusion with the richer notion of clocks in dataflow synchronous programming languages). In order to separate design details from implementation details, *logical rates* and *concrete rates* are defined in two different files. In the `forec` file, rates are purely logical and only upsampling and downsampling relationships can be expressed. Whenever a thread is forked (by a `par` statement), its default rate is that of its parent, but a different logical rate may be specified. The top-level `main` thread is a special case because it is not forked by a `par` statement; it must therefore be declared with a logical rate.

In the `foreh` file, the programmer assigns a *concrete value* for the *period* of the `main` thread's rate. Based on

this concrete rate and on the upsampling and downsampling relationships, a concrete value can then be deduced for all the rates in the program.

A. Declaration of Logical Rates

Logical rates are declared in the `forec` file as C constants, with the keyword `rate`. For instance:

```
const rate r0;
```

Upsampling and downsampling relationships are then declared in the following way:

```
const rate r1=r0/2,r2=r0,r3=r1*4,r4=r2/3;
```

```
// Environment variables
env in Speed s_spc;       // the speed control point
env in int w_L, w_R;      // angular speed (left/right)
env in Position p_fm,     // marker-based pose
                dp_imu,   // displacement (dx,dy)
                p_uwb;    // UWB positionning
env in Acc acc_imu;       // IMU acceleration
env in Altitude alt_imu;  // Altitude of the robot
env in int is_tracking_obstacle;

env out int dc_L, dc_R; // duty cycle command

Path path; // path to follow
Map map;   // map of the building

void main (void) {

  shared Position p,    // robot's pose (x,y,θ)
                  p_t; // setpoint to track (x_t,y_t,θ_t)

  shared Speed s,     // speed setpoint (v,w)
               s_spl,  // bounded speed setpoint (v,w)
               s_max;  // maximum speed setpoint (v,w)

  // non-safety monitoring flags (position/speed)
  shared int is_inv_pos = 0, is_inv_speed = 0
  // safety monitoring flags (acceleration/altitude)
  shared int is_inv_acc = 0, is_inv_alt = 0;

  while (1) {
    // normal mode
    weak abort {
      par (
        loc: LOC(w_L, w_R, p_uwb, dp_imu, p_fm, p),
        tg: TG(q, a, d, p, p_t, path),
        tra: TRA(p_t, s),
        mon: MON(p, rot_imu, acc_imu, is_inv_pos,
                 is_inv_alt, is_inv_acc,
                 is_inv_speed, Map map),
        spl: SPL(s, s_max, s_spc, s_spl),
        sam: SAM(is_inv_pos, is_inv_alt, is_inv_acc,
                 is_inv_speed, s_max),
        moc: MOC(s_spl, w_L, w_R, dc_L, dc_R)
      );
    } when immediate(is_inv_acc | is_inv_alt);
    // safety mode
    weak abort {
      while (1) {
        dc_L = 0; dc_R = 0;
        pause;
      }
    } when immediate(!(is_inv_acc & is_inv_alt));
  }
}
```

Listing 1: Mono-rate ForeC program of the autonomous robot.

The expression attached to each logical rate declaration is built using the following grammar:

```
ckexp ::= id ([*|/] numexp)?
numexp ::= num ([*|/] num)?
```

where `id` must be the identifier of a previously declared logical rate, and `num` is a strictly positive integer. In the above example, `r1` is two times *faster* than `r0`, `r2` is *identical* to `r0`, `r3` is four times *slower* than `r1`, and `r4` is three times *faster* than `r2`. In the abstract syntax tree we keep the symbolic arithmetic expression of each rate (e.g., `r2/3` for `r4`).

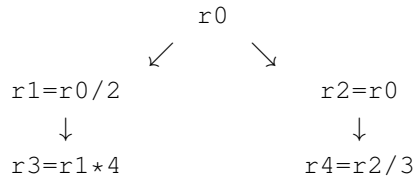

Fig. 3: The tree of logical rates for `r0` to `r4`.

It follows that the structure induced by these rate relationships is a *tree*, as shown in Fig. 3 for the above rate declarations. Two remarks are in order:

1) The structure linking the logical rates is inferred from the rate declarations and not from the call tree of the ForeC program (i.e., not from the `par` statements).
2) In theory, the logical rate structure could be a *forest* with at least two disjoint trees; for instance if the `forec` file contained more than one declaration like that of `r0` above. We argue that this does not make sense from a programming point of view. Indeed, if the logical rate structure is a forest, then it is as if the ForeC program was actually composed of at least two *disjoint* ForeC programs, which communicate only via environmental variables. For this reason, we assume here on that the rate structure is a tree. This property is enforced by the ForeC compiler.

B. Assignment of Concrete Rates to Threads

Each thread either inherits the logical rate of its parent thread, or is given an explicit logical rate when it is forked, prefixed with the special character `@`. Since the `main` thread is never forked, it must be given a logical rate in its preamble, prefixed with the special character `@`. Listing 2 provides an example. `t3` inherits the rate of `main` when it is forked.

```
const rate r0;
const rate r1=r0/2, r2=r0, r3=r1*4, r4=r2/3;

void main ()@r0 {
  ...
  par(t1()@r1, t2()@r2, t3(), t4()@r4);
  ...
}
```

Listing 2: A simple multi-rate ForeC program.

In a well-defined ForeC program, the logical rate of any thread can therefore be expressed as a linear function of the

logical rate of any other arbitrary thread. This property is a direct consequence of the logical rate structure: it is a *tree* so there is a unique path between any two nodes.

For a ForeC program to be executable, a *concrete* rate must be derivable for *all* threads. Thanks to the previous property, assigning a physical time measured in micro-seconds (μs) to an arbitrarily selected rate (actually to its period) will allow the period of all the other rates to be easily derived. The programmer must therefore provide a concrete value to one rate in the `foreh` file. For instance:

```
const rate r0=100;
```

The tree of rates of Fig. 3 can therefore be decorated with the concrete values and with the threads, as shown in Fig. 4 for the example of Listing 2. For each rate, if the computed concrete value is an integer, then we attach this value to this rate (e.g., `50` for `r1`), otherwise we keep the symbolic expression (e.g., `100/3` for `r4`).

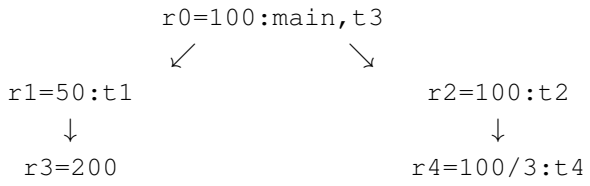

Fig. 4: The decorated tree of logical and concrete rates for Listing 2.

C. Thread Synchronization

In classical ForeC, all threads have the same rate, so their synchronization is straightforward: the end of a global tick (EOT) occurs when *all* threads end of their local tick by executing a `pause`. Synchronization is more complex in Multi-Rate ForeC because only the threads that end their local ticks at the same *absolute time* will synchronize together.

First, we distinguish between *total* and *partial* global EOT depending on whether *all* or only a *subset* of the threads participate in this global EOT. We define the counter n_g that denotes the number of the current global tick. Second, each parallel active thread keeps track of the absolute time, in micro-seconds (μs), which will be used to compute the time that the next (total or partial) global EOT n_g must take place.

The time of the next global EOT, denoted $\Delta(n_g)$, is computed in the following way (items ① to ⑤).

① Let $\mathcal{AT} = \{t_i\}_{1 \leq i \leq k}$ be the subset of currently *active* parallel threads. A thread is active from the time it is forked by the `par` statement until the corresponding join. For each $t_i \in \mathcal{AT}$, let r_i denote its concrete rate in μs, and let s_i denote the instant of its first local tick. According to the synchrony hypothesis all computations take *zero time*, so in the case of a child thread, s_i is the start of the global tick during which the `par` statement that forked it is executed, even if this `par` statement was not the first instruction of the tick. Finally, let n_i be the counter that denotes the number of the current tick of t_i. Because the rates are different, each active thread t_i may have a different tick count n_i.

② Each active thread t_i keeps track of the time of its next local EOT n_i, denoted $\delta_i(n_i)$ and computed *symbolically* by Eq. (1). This equation relies on the fact that the ForeC compiler makes sure that the start time of the $n_i + 1$ tick is equal to the end of the n_i EOT (see Fig. 1).

$$\delta_i(n_i) = s_i + n_i * r_i \tag{1}$$

Computing δ_i symbolically avoids rounding errors. For instance, for $r_4 = 100/3$ we would obtain the rounded value 33. As a result, the third local EOT of t_4 (that is, $\delta_4(3) = 99$) would never be equal to the first EOT of t_2 (that is, $\delta_2(1) = 100$).

③ The instant $\Delta(n_g)$ of the next global EOT n_g is computed by Eq. (2):

$$\Delta(n_g) = \min_{t_i \in \mathcal{AT}} \{\delta_i(n_i)\} \tag{2}$$

④ The subset $\mathcal{PT}(n_g)$ of participating threads in this global EOT n_g is computed by Eq. (3):

$$\mathcal{PT}(n_g) = \operatorname*{argmin}_{t_i \in \mathcal{AT}} \{\delta_i(n_i)\} \tag{3}$$

Once this global EOT n_g has been resolved, for each thread $t_i \in \mathcal{PT}(n_g)$ we increment the tick counter n_i and we update the value of the next local tick δ_i. Finally, we increment the global tick counter n_g. These computations are generated by the ForeC compiler and take place in the magenta part of each tick in Fig. 1.

⑤ Whenever some parent thread t_p executes a `par` statement, we keep track of t_p's ticks by incrementing its tick counter n_p and by computing the sequence of EOT instants $\delta_p(n_p)$ based on Eq. (1), as if t_p was still active. These EOTs when t_p is not active are called *phantom ticks*. When the join occurs (i.e., when all the threads forked by this particular `par` statement terminate), t_p resumes its execution at its rate r_p, as if it had been active during the `par`. Thus, the next local EOT for t_p would be n_p, which is due to occur at $\delta_p(n_p)$.

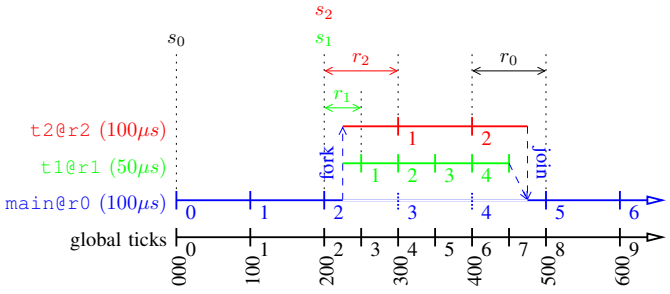

Fig. 5: A trace for a multi-rate ForeC program with three threads.

This behavior is illustrated in Fig. 5 for the example of Listing 2 by considering only threads `main`, `t1`, and `t2`. Since $r_0 = 100$, we have $\Delta(2) = \delta_{main}(2) = 200$ (all units are in μs). In this trace, the `par` occurs during the global tick numbered 2, say at time 225. From both `t1` and `t2`'s perspective, their local tick begins at $s_1 = s_2 = 200$ (see Item ① above). Since $r_1 = 50$, we thus have $\delta_1(1) = 250$, and since $r_2 = 100$, we thus have $\delta_2(1) = 300$. As a consequence, the next global EOT is partial and occurs at $\Delta(3) = 250$. This

process repeats, with the following global EOTs occurring at $\Delta(4) = 300$ (total) and at $\Delta(5) = 350$ (partial). In the meantime, the `main` thread (which is the `par`'s parent thread) has two phantom EOTs at $\delta_{main}(3) = 300$ and $\delta_{main}(4) = 400$, depicted in dotted style. Then, suppose that `t1` terminates its execution with a `pause` statement, just after its EOT at 450. Suppose also that `t2` terminates its execution in the middle of its tick at 475. It follows that the join takes place at 475, so at that time the `main` thread resumes its execution and its next EOT occurs at $\Delta(8) = 500$.

This example highlights the benefits of Multi-Rate ForeC: `t1` and `t2` do not have the same rate, which allows them to sample inputs and emit outputs at *different frequencies*. This is very useful to control devices that have different dynamics and/or real-time requirements. From the point of view of the overall program (the global ticks), we see an acceleration taking place during the global ticks 3 to 8, the rate being $50\,\mu s$, before resuming with the former rate of $100\,\mu s$.

We mention two points concerning the timing assumptions of our model. First, to ensure that threads are woken up at the required times, our bare-metal implementation (see Section IV) relies on the timing instructions offered by PRET processors [5], which allow a thread to be woken up at an absolute time. Second, compliance of a thread's execution time to the duration of its local tick is ensured by worst-case execution time (WCET) analysis [8].

Several other aspects need to be studied, e.g., the multi-rate semantics of `abort` statements, which, for lack of space, will be explained in a future research report.

D. Thread Communication

We now examine how two parallel threads having different rates communicate with each other via a shared variable. Consider Fig. 5 and the following shared variable declaration:

```
env out shared int x=0 combine mod with +;
```

where only the copies of `x` that have been written to (i.e., `mod`) are combined with mathematical addition (i.e., `+`). See [9], [10] for further combine policies and combine functions.

Suppose that `t1` and `t2` both increment `x` during each of their local ticks. At $n_g = 3$ only `t1` has incremented `x`, so the value emitted to the environment is 1; this value is not propagated to `t2` because `t2` does not participate in this partial global EOT. At $n_g = 4$, both threads have incremented `x` so the emitted value is $2 + 1 = 3$, which is propagated to both threads. The partial global EOT at $n_g = 5$ is similar to the one at $n_g = 3$ and the emitted value is 4. And so on.

In Simulink, it is also possible to program communicating tasks with different rates [18]. When the sender (e.g., task `A` at $20\,ms$) is faster than the receiver (e.g., task `B` at $80\,ms$), a *zero oracle* block `Z` has to be inserted between them, which inherits the rate of `B` and the priority of `A` (tasks in Simulink are executed by a Real-Time Operating System with a Fixed-Priority Preemptive scheduler). In this particular case, `A` executes four times for every execution of `B`. As a result of the zero oracle, the value received by `B` is the one produced by the *first* instance of `A`, as illustrated in Fig.6. Similarly, a *unit delay* block has to be inserted for communication from a slow to a fast task.

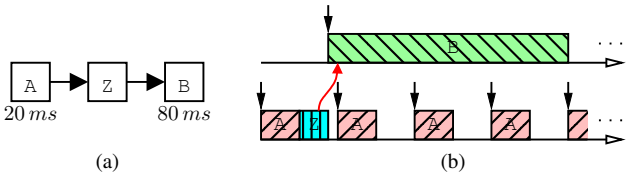

Fig. 6: Two communicating Simulink tasks.

Compared to Simulink, the advantage of ForeC's combine functions is to offer versatility in handling shared variables. Indeed, depending on the combine function chosen for the global variable shared between `A` and `B`, the equivalent of the behavior of Fig. 6 can be achieved (if the combine function keeps the first value and ignores the three other values), but also any other behavior, be it sending the fourth value produced, the sum of the four values, their average, etc.

Finally, in Multi-Clock Esterel [15], communications between different clock zones are handled by two special devices, called the *sampler* and the *reclocker*. However, defining the semantics for sampling the presence/absence of valued signals proved difficult, so the multi-clocked emission of valued signals was left undecided. This issue does not arise with Multi-Rate ForeC because shared variables are always present throughout a program's execution. Code generation of multi-clock Esterel towards multi-cores is also unsupported.

E. Glue-code for Global EOTs

In the multi-rate case, the glue code that the ForeC compiler has to generate for the global EOTs varies slightly compared to the mono-rate case. When the global EOT is total, the generated code is identical to the mono-rate case. In contrast, when the global EOT is partial, only the copies of shared variables from the threads that participate in this EOT (i.e., those in the subset $\mathcal{PT}$) are combined.

In addition, the compiler must generate code to keep up-to-date the various counters (n_g and, if within a `par` statement, n_p and all the n_i), the subsets of threads $\mathcal{AT}$ and $\mathcal{PT}$, the time of each thread's next local EOT (the values of δ_p, δ_g, and of all the δ_i), and finally the time of the next global EOT Δ. All these computations are generated by the ForeC compiler and take place within each magenta part in Fig. 1.

Regarding time predictability, the compiler checks that, for each rate r_i, the WCET between sampling the inputs on r_i and producing the corresponding outputs is less than the concrete value of r_i. Except from the fact that this is now performed for each rate, it proceeds as in the classical ForeC compiler.

IV. Bare-metal Multi-Core PRET Implementation

This section presents our implementation of Multi-Rate ForeC programs on a multi-core PRET architecture called MultiPRET [22], which consists of n FlexPRET [5] cores connected via a full cross-bar for inter-core communication (see Fig. 7). FlexPRET is itself a multi-threaded single-core PRET architecture. More efficient and less resource consuming inter-core communication could be provided by a time-deterministic

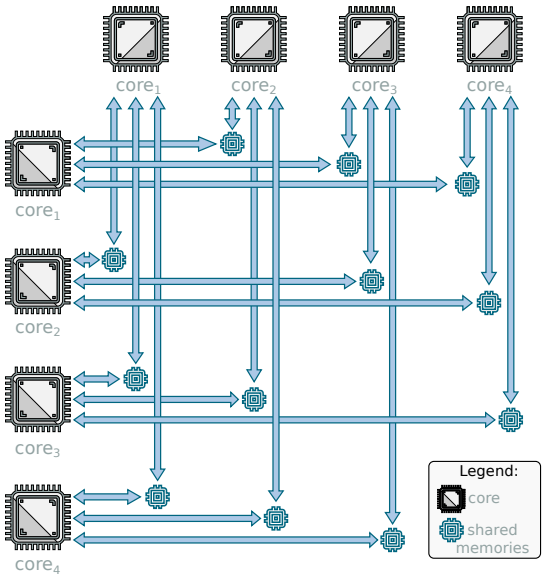

Fig. 7: MultiPRET n-core topology (with $n = 4$).

bus, but for this paper we use a full cross-bar to avoid inter-core interference that would be induced by bus contentions.

We adopted a full time-triggered model of execution and communication where threads can sleep or wake up at a specific (absolute) timestamp. This is guaranteed by special timing instructions provided by MultiPRET. We also assume that all environment variables are mapped to specific locations in the shared memories of the different cores. The rates at which environment variables are read or written by external processes are outside the scope of this paper. More information can be found in [22].

Fig. 8 outlines the translation of a ForeC program into a deterministic subset of C code that is executable on MultiPRET. The inputs of the translation is a ForeC program (a `forec` file) and the deployment details (a `foreh` file) such as the mapping of ForeC threads to the hardware threads of MultiPRET cores.

Listing 3 is an example `foreh` file for the ForeC program of Listing 1. It first defines the target architecture. Then, the rate of the `main` thread is set, which bridges the logical rates of all the ForeC threads with the physical time (in μs) used by the MultiPRET architecture. Finally, the deployment of each ForeC thread to a hardware thread on a core of the MultiPRET platform is defined.

Each core is uniquely identified by its ID and each ForeC

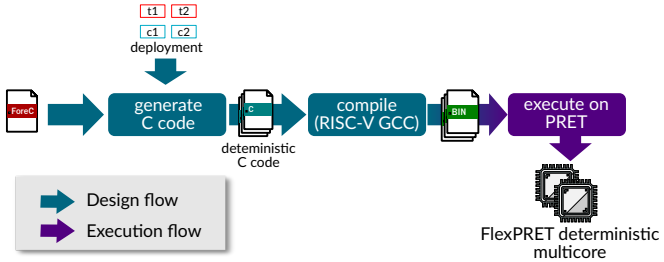

Fig. 8: Translating ForeC to C.

```
1  architecture: multipret
2  const rate r1: 1000
3
4  0: // core 0
5    main    // main thread
6    loc     // localization thread
7    tg      // target generation thread
8    tg.ptg  // path target generation forked by tg
9
10 1: // core 1
11   tg.atg  // avoidance target generation forked by tg
12   tra     // tracking thread
13   spl     // speed limitation thread
14
15 2: // core 2
16   mon     // monitoring thread
17   sam     // safety management thread
18   moc     // motor control thread
```

Listing 3: Example of the `foreh` deployment file.

thread is uniquely identified by a fully qualified name. The qualified name of the `main` thread is *main*. Direct children of the `main` thread are identified by their instance name (e.g., *loc*). Indirect children are identified by their instance name prefixed by their parent's instance name relative to `main`. For example, `tg.ptg` and `tg.atg`, respectively, refer to the *Path target generation* (`PTG`) thread instance and to the *Avoidance target generation* (`ATG`) thread instance forked by the *Target generation* (`TG`) thread instance. Note that thread's instance name is used rather than the thread's name because a ForeC program may execute several instances of the same thread in parallel. For example, the `PID` thread has two instances *left* and *right* (cf. Fig. 2). When creating the `foreh` file, the developer has to ensure that all thread instances are correctly mapped to the cores of the platform.

In MultiPRET, thread-specific control instructions are used to fork threads. These instructions only affect the threads deployed on the same core as their parent. Thus, a thread deployed on core j cannot fork threads or force the threads on core k to join. To alleviate this problem, a technical solution was to duplicate parts of the parent thread onto the cores of their children. For example, according to Listing 3, parts of the `main` thread are duplicated over three cores, while parts of the `tg` thread are duplicated on $core_0$ and $core_1$ so that it can fork `atg` and `ptg` on their deployed core.

Listing 4 gives a excerpt of the C code of the localization thread `LOC` (as shown in Fig. 1) to be executed on $core_0$. Lines 1–9 define the physical addresses of all shared variables. Given the memory mapping we adopted, the two safety variables at lines 8–9 are available in the shared memory of $core_0$ and $core_2$, and thus their addresses are set to a value above `0x50000000`. The `LOC` thread executes in a periodic loop where: i) the shared variables are copied into the local memory of the core (lines 21–29), ii) the thread performs its operations (line 31), iii) the thread waits for its period to elapse (line 33), and iv) the thread's outputs are written to the shared memory (line 34). Based on the relations between the different thread rates, the values of some shared variables may be refreshed less often than others (this is the case for

```
1  // Shared variables
2  Pos volatile* const p_addr = (Pos*) 0x40000024;
3  Pos volatile* const p_uwb_addr = (Pos*) 0x40000034;
4  Pos volatile* const dp_imu_addr = (Pos*) 0x40000044;
5  Pos volatile* const p_fm_addr = (Pos*) 0x40000054;
6  volatile int* w_L_addr = (int*) 0x40000064;
7  volatile int* w_R_addr = (int*) 0x40000068;
8  volatile int* is_inv_acc_addr = (int*) 0x50000024;
9  volatile int* is_inv_alt_addr = (int*) 0x50000028;
10 // Threads main, TG, TRA, MON, SPL, SAM, MOC, ...
11
12 /** localization thread (rate: 50ms) */
13 void LOC(int* w_L_addr, int* w_R_addr, ...) {
14   /* ... */
15   unsigned int i = 0, time = 0;
16   // local variables
17   int w_L = 0, w_R = 0;
18   Position p_uwb, dp_imu, p_fm, p;
19   while (1) {
20     do {
21       w_L = *w_L_addr;
22       w_R = *w_R_addr;
23       dp_imu = *dp_imu_addr;
24       is_inv_acc = *is_inv_acc_addr;
25       is_inv_alt = *is_inv_alt_addr;
26       // update every 500ms
27       if (i % 10 == 0) { p_uwb = *p_uwb_addr; }
28       // update every 1s
29       if (i % 20 == 0) { p_fm = *p_fm_addr; }
30
31       kalman(w_L, w_R, p_uwb, dp_imu, p_fm, &p);
32
33       delay_until_periodic(&time, 50000000); //50 ms
34       *p_addr = p;
35       i = (i+1) % 20;
36     } while(!(is_inv_acc | is_inv_alt)); // abort
37   }
38 }
```

Listing 4: Localization thread deployed on $core_0$. Thread `main` and two of its child threads have been omitted.

`p_uwb` and `p_fm`).

If an `abort` statement encloses a `par` statement, then the semantics of the `abort` is delegated to the `par`'s child threads. For example, the condition of the `weak abort` statement in Listing 1 (lines 32–45) is evaluated in Listing 4 by the localization thread `LOC` (lines 20–36).

V. Conclusion

We have presented a multi-rate extension to the ForeC multi-threaded precision timed language, and have proposed elements of its implementation. By providing the capability to handle directly threads with multiple rates, Multi-Rate ForeC offers the possibility to design embedded software with multiple temporal dynamics, while guaranteeing time predictability. As a result, it reduces both the development time and execution time of actual applications, and finally, enhances the industrial acceptance of ForeC. This new capability has been illustrated by a robotic application deployed on the MultiPRET deterministic PRET processor.

Future work will include the generalisation of rates with offsets, and the support for a forest of clocks. Integration with a tool is on-going.

References

[1] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in *Design Automation Conference (DAC)*. ACM, 2007, pp. 264–265.

[2] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. von Hanxleden, R. Wilhelm, and W. Yi, "Building timing predictable embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, p. 82, Feb. 2014.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[4] P. S. Roop, "Predictable Reactive Processors for Next Generation Computing: A Proposal," *School of Engineering Report*, vol. 662, 2008.

[5] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee, "FlexPRET: A Processor Platform for Mixed-Criticality Systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 101–110.

[6] S. Andalam, P. Roop, and A. Girault, "Predictable Multithreading of Embedded Applications Using PRET-C," in *International Conference on Formal Methods and Models for Codesign*. IEEE, Feb. 2010.

[7] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen, "A Predictable Framework for Safety-Critical Embedded Systems," *IEEE Transactions on Computers*, vol. 63, no. 7, pp. 1600–1612, July 2014.

[8] E. Yip, P. S. Roop, M. Biglari-Abhari, and A. Girault, "Programming and Timing Analysis of Parallel Programs on Multicores," in *International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2013, pp. 160–169.

[9] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari, "The ForeC synchronous deterministic parallel programming language for multicores," in *International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. Lyon, France: IEEE, Oct. 2016.

[10] E. Yip, P. Roop, A. Girault, and M. Biglari-Abhari, "Synchronous deterministic parallel programming for multicores with ForeC," Inria, Research Report RR-8943, Aug. 2016. [Online]. Available: https://hal.inria.fr/hal-01351552

[11] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[12] F. Boussinot and R. de Simone, "The Esterel language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, Sep. 1991.

[13] M. Alras, P. Caspi, A. Girault, and P. Raymond, "Model-based design of embeded control systems by means of a synchronous intermediate model," in *International Conference on Embedded Systems and Software (ICESS)*. IEEE, May 2009, pp. 3–10.

[14] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A multi-periodic synchronous data-flow language," in *High Assurance Systems Engineering Symposium (HASE)*. IEEE, 2008.

[15] G. Berry and E. Sentovich, "Multiclock Esterel," in *IFIP Conference on Correct Hardware Design and Verification Methods, CHARME'01*, ser. LNCS, vol. 2144. Springer-Verlag, Sep. 2001, pp. 110–125.

[16] The MathWorks, Inc., *Real-Time Workshop User's Guide, v3*, 1999.

[17] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," in *International Workshop on Embedded Software (EMSOFT)*, ser. LNCS, vol. 2211. Springer, Oct. 2001.

[18] A. Girault, X. Nicollin, and M. Pouzet, "Automatic rate desynchronization of embedded reactive programs," *ACM Transactions on Embedded Computing Systems*, vol. 5, no. 3, pp. 687–717, Aug. 2006.

[19] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, Feb. 2008.

[20] Xilinx, *MicroBlaze Processor Reference Guide*, 2012, accessed 28 Aug. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/mb_ref_guide.pdf

[21] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee, "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance," in *International Conference on Computer Design (ICCD)*. IEEE, Sep. 2012, pp. 87–93.

[22] N. Hili, A. Girault, and E. Jenn, "Worst-Case Reaction Time Optimization on Deterministic Multi-Core Architectures with Synchronous Languages," in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, Aug. 2019.