# Programming and verifying real-time design using logical time

Fabien Siron*†, Dumitru Potop-Butucaru†, Robert de Simone†, Damien Chabrol* and Amira Methni*
*Krono-Safe - Massy, France
†INRIA - Paris/Sophia-Antipolis, France
*firstname.lastname@krono-safe.com
†firstname.lastname@inria.fr

***Abstract*—The design of embedded control software calls for stringent real-time constraints. For that, formalisms and theories based on the notion of logical time give abstraction of real-time durations that are usually not known at design level. Comparison between synchronous languages, *Logical Execution Time (LET)* and the *PsyC* language can be fruitful, in our case, with the goal of empowering the industrial language *PsyC*, which is close to *LET*, with (logical) time and functional verification methods inspired from synchronous languages.**

## I. Introduction

The design of embedded control software, extensively interacting with its physical environment, calls for stringent real-time constraints. But effective real-time durations are often not known of designers (either fluctuating or not yet set), while temporal system correctness must imperatively be established at early stage, for safety-critical reasons.

A number of formalisms and theories have introduced notions of logical time, so that design flow can be split in two: full design (specification, programming, verification) enforcing Logical Time assumptions on the one hand, validation of these assumptions regarding the physical implementation (time) on the other hand. Synchronous languages and Multiform Logical Time [1], Logical Execution Time [2], and the *PsyC* language [3] propose domain-specific languages (on top of general-purpose C) using Logical Time variations for their primitive constructs. Comparison can be fruitful, in our case, with the goal of empowering the industrial language *PsyC*, which is close to *LET*, with (logical) time and functional verification methods inspired from synchronous languages.

## II. Synchronous Languages

Synchronous languages target determinism and concurrency. For that, computations react simultaneously and instantaneously with the tick of a global common clock. Time cannot be used explicitly but through the use of a sequence of reactions. Hence, time is said to be logical. The synchronous hypothesis, then, ensures that if every computation is bounded by the next reaction, then physical time can be safely ignored [1].

The compilation of synchronous languages is however quite complex. Besides causality issues, it is also generally not possible to compile them to parallel code. Moreover, execution time is usually limited to the reaction time of the system through an analysis called *worst case reaction time* (WCRT). This makes the schedulability analysis of certain systems (i.e multi-rate systems) quite pessimistic.

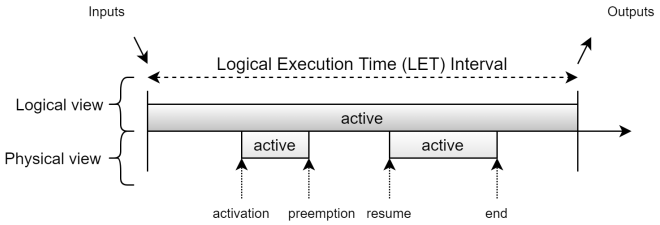


Fig. 1. Description of a *LET* interval

More recent work addresses this kind of issues through the use of a weaker form of synchrony. *CoReA* (by Boniol, [4]) is a synchronous process algebra in which reactions are synchronous but communication is always delayed to the next instant. More recently, Forget and al introduced *Prelude* [5] as a synchronous data-flow multi-rate language based on *Lustre*. Basically, the reaction time of each flow is bounded by its next activation. That is, each flow has its own reaction rate.

## III. Logical Execution Time

The *Logical Execution Time (LET)* [2] paradigm abstracts the physical time through a sequence of logical instants, similarly to the synchronous paradigm. However, contrary to the latter, computation takes time and is not considered instantaneous. For that, each computation must fit in a logical interval, called *LET* interval. Furthermore, communications are only made on the boundaries of *LET* intervals. Inputs are read at their beginning and outputs are made visible to other tasks at their end (see figure 1). Such abstraction is justified because the observable execution time of a given function does not really depends on the execution time of its computation. It actually depends on when communication is done.

In a way, *LET* extends logical time with the concept of duration. This allows a precise schedulability analysis due to an increased execution time variability. In turn, this also makes multi-rate systems easy to compile to parallel code (i.e. for multitasking platform, multicore platform ...). Moreover, determinism coming from the synchronous paradigm is also

```
clock S = 1000 * MS;
agent My_Agent(uses realtime, starttime 1 MS)
{
  body start
  {
    /* computation */
    advance 1 with S;
    /* infinite loop on start body */
  }
}
```

Listing 1. Example of a *PsyC* agent

preserved as all communications are made available on predefined dates.

Although *LET* is generally classified into a timed model of computation due to its proximity with the timed-triggered architecture (TTA) [6], all *LET* instants (i.e. *LET* interval boundaries) are relative to the same common global clock. Thus, like for the synchronous model, time is logical and is not made explicit. As an example, *Giotto*, a periodic timed-triggered *LET* language has been extended to handle events with the *xGiotto* [2] language. This is actually a generalization of the *LET* paradigm to handle non constant *LET* intervals (i.e. where their boundaries depend on external events). As both *LET* and the synchronous model are based on logical time, they share some big similarities. Even if *LET* executions are not considered instantaneous (i.e. they are only bounded by the end of the interval), it is actually a correct abstraction to consider execution to be instantaneous if and only if communication is delayed to the next activation date (i.e. the end of the interval). That is, the synchronous model, can be used to abstract the *LET* paradigm. This is actually very close to the model proposed by *CoReA* [4]. This abstraction allows to obtain a simpler model which can be used, in turn, for verification.

## IV. The PsyC language

The *PsyC* language is an implementation of a generalized form of the *LET* paradigm (i.e. similarly to *xGiotto*) which is dedicated to safety-critical real-time software [3]. It was actually made as an extension of the C language to ease the integration of applications written with the latter. *PsyC* is packaged in a software suite called *ASTERIOS* which is produced by the company Krono-Safe. This toolsuite is used to allow the integration of safety-critical applications in the avionic domain certified at the highest level of criticity (DAL-A, DO-178C).

A *PsyC* application is composed of tasks called *agents* that are sequential units of computation formed of *LET* intervals. The content of an agent is composed of C code in which a special instruction, `advance n with c`, specifies the boundaries of the *LET* intervals. Informally, its semantics is to advance the logical time of $n$ ticks of a clock $c$ derived from the temporal source. Determinism is a consequence of the visibility principle for inter-task communication. Data are timestamped at the dates where they are made visible, that is, the end of the *LET* interval. Moreover, for a given function, its external data (i.e. its inputs) are visible if and only if their timestamp is smaller or equal than the activation date of the *LET* interval. At the application level, *agents* are composed together in parallel in a synchronous way and communicate together with a sampled communication means called *temporal variable*.

The listing 1 shows the example of a *PsyC agent* that do some periodic processing on clock $S$. There is, however, a little difficulty as the first non empty *LET* interval of the *agent* is actually smaller than the period of $S$. This is a very classical issue coming from synchronous languages and logical clocks. A property we might want to check, is to ensure that the latency between the input and the output of this task is bounded by a minimum and a maximum value.

## V. Discussion et perspective

As the synchronous model is actually an abstraction of the *LET* paradigm, it can be used to express logical models of *LET* based languages. Those models can be used to express the formal semantics of such languages, or they can be used for a simulation purpose where execution time is not relevant. But the more interesting result is probably to abstract *LET* systems using the synchronous model to use synchronous verification techniques. For *PsyC*, it is possible, for example, to give translation rule to *Esterel* which is very close. This is actually a correct way to give a formal semantics of *PsyC* as *Esterel* is formally defined. Moreover, this allows simulation and formal verification using synchronous techniques.

## VI. Conclusion

The *LET* paradigm ensures determinism while allowing easy compilation and fine-grained schedulability analysis. However, today, there is almost no work on the formal verification of complex *LET* based languages such as *PsyC*. This work suggests that the similarity between *LET* and synchronous languages can be exploited to reuse the verification techniques of the latter on *LET* based languages. A more precise positioning of *PsyC* will be developed in futur work.

## References


[1] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, 10 1991.

[2] C. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*. Springer Berlin Heidelberg, 10 2012.

[3] D. Chabrol, G. Vidal-Naquet, V. David, C. Aussagues, and S. Louise, "Oasis: A chain of development for safety-critical embedded real-time systems," in *2nd European Congress Embedded Real Time Software. (ERTS 2004)*, 01 2004.

[4] F. Boniol, "CoReA: A synchronous calculus of parallel communicating reactive automata," in *PARLE'94 Parallel Architectures and Languages Europe*. Springer, 1994.

[5] J. Forget, F. Boniol, D. Lesens, and C. Pagetti, "A Multi-Periodic Synchronous Data-Flow Language," in *11th IEEE High Assurance Systems Engineering Symposium*, Dec. 2008.

[6] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer Publishing Company, 2011.