The synchronous Logical Execution Time paradigm

Fabien Siron, Dumitru Potop-Butucaru, Robert de Simone, Damien Chabrol, Amira Methni

The synchronous Logical Execution Time paradigm

Fabien Siron*†, Dumitru Potop-Butucaru†, Robert de Simone†, Damien Chabrol* and Amira Methni*
*Krono-Safe, Massy, France
†Université Côte d'Azur, Inria, France
*firstname.lastname@krono-safe.com
†firstname.lastname@inria.fr

***Abstract*—Real-Time industrial systems are not so much of those that have to perform tasks incredibly fast, but in a time-predictable manner; they rather focus on meeting previously specified timing requirements in a provable way. Consequently, time must be taken into account from the very start of the design. However, exact timing constants may not be available yet in early design stages as they may depend on the target. In answer, formalisms based on the Multiform Logical Time have been introduced to abstract real-time durations. The Synchronous-Reactive (SR) approach introduced a discretized abstraction of time on which computations happen logically instantaneously. Contrary to SR, Logical Execution Time (LET) mandates to specify the actual logical duration a task has to fulfill. This allows a more efficient compilation, at the price of a lower expressiveness. Classical LET (i.e. as introduced in *Giotto/TDL*) sticks to uniform pseudo-physical time, i.e. based on one logical clock mapped to the real-time. In this paper, we introduce a new paradigm called *synchronous Logical Execution Time* (sLET) that builds upon both SR and LET paradigms. It keeps the idea of *logical durations* coming from the LET paradigm, while having logical instants based on logical clocks. This extends the expressivity of LET, as time is totally abstracted as sequences of events. The various schedulings provide physically timed versions that, while having distinct non-functional properties (in terms of performance mostly), remain mutually functionally equivalent (in the logical time realm). A particular instance, where computations are executed "in a single instant", and then time is advanced (as in classical event-driven simulation), can lead to a direct translation into synchronous formalisms (in our case *Esterel*). We started inquiring how this could open new ways of verification and analysis on PsyC programs.**

I. Introduction

The design of embedded control software calls for stringent real-time constraints due to their permanent interaction with the physical environment. Such real-time systems are usually safety-critical because of the context in which they are used (avionic, automotive ...). Therefore, their designs should be verified with the highest possible level of confidence. Various formalisms based on the concept of *Multiform Logical Time* have been introduced to abstract real-time durations which are usually not known at the design phase, as they might be target-dependent. Then, specific analysis, called schedulability analysis (or time safety analysis) ensure that physical computations satisfy their logical time constraints.

Among those formalisms, the *Logical Execution Time* (LET) paradigm [1] brings a compromise between the strong expressiveness of the synchronous-reactive approach (SR) [2] and the efficiency of traditional task scheduling. Contrary to the SR model, computation takes time and is bounded by a fixed logical duration usually known before the computation happens. However, in SR model, because computations are considered instantaneous, time can be refined using multiple logical clocks. LET is usually based on a unique clock representing the progress of time. Consider the following temporal interval where the bounds are described by `sync`:

```
sync 1 ms; f(); sync 1 s;
```

Depending on the semantics of the language, the interval might last one second or up to the next second tick. The former semantics, usually used by LET based languages, would mean that both ms and s refer to the same clock (i.e. $1\,s$ being just a short for $1000\,ms$) while the latter semantics, corresponding to synchronous languages, mean that both ms and s are logical clocks related with an affine relation.

In industrial projects, while classical synchronous languages such as Lustre or Scade fit perfectly the need to describe the functional behavior of the system, Synchronous LET, which is implemented by the industrial language *PsyC* [3], actually focus on a different level of the system life-cycle, namely, the software integration level. This approach is quite classical (see [4], [5] and[6]) and divides the functional design of the system from its non-functional design, that is, how different functional components (either designed with synchronous languages or manually) are integrated. This multilayer approach is depicted in figure 1. This is, however, still a challenge today to verify that properties specified at system-level (i.e. in the specification) still holds at integration level.

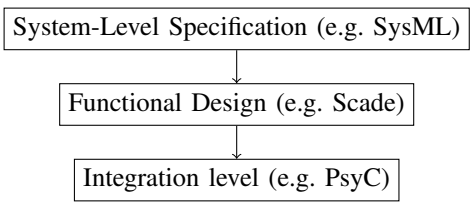


Fig. 1: Typical Software Life-Cycle of industrial real-time systems

In this paper, we introduce a new paradigm called *synchronous Logical Execution Time* (sLET) that builds upon both

the SR model and LET. It keeps the idea of *logical durations* coming from the LET paradigm while having logical instants based on logical clocks. This extends the expressivity of LET as time is totally abstracted as sequences of events. Thus, it inherits 1) the simple compilation and fine-grained schedulability analysis coming from LET and 2) the synchronous semantics that is well-suited for formal verification.

After an overview of existing Multiform Logical Time models, we introduce sLET as an extension of LET in section II. To illustrate sLET, we then give an overview of the formal semantics of a subset of the industrial language *PsyC* [3] in section III, developed by the company Krono-Safe, and then, we give translation rules to the synchronous language Esterel in section IV. While *PsyC* is compiled using techniques similar to the ones of LET, formal verification techniques coming from synchronous languages such as *Esterel* could then be re-used. Such approach is illustrated in section V where *PsyC* and Scade interact in a classical use-case showing a flight control system.

II. Multiform Logical Time Models

Synchronous languages introduce the notion of Multiform Logical Time, through logical clocks, so they could better be called Polychronous time. A logical clock counts the successive ticks of any relevant event. Still, their operational semantics expands the behaviors in terms of a grandmother clock, the discrete reaction step measuring the instant. On the other hand, initial LET formalisms rely on such a totally ordered discrete time, and introduce quantitative durations measured in it; although they may use pseudo-physical unit names (milliseconds, nanoseconds,...) for it, the compliance with actual physical implementation is only a wish (or a requirement), that will have to be checked later when the latter becomes available. We now extend on these notions, and the abstraction they allow for fast and human-legible logical design, cleanly split from the later physical implementation (and without back and further adjustements hopefully, a main purpose of these formalisms). The purpose is to position the sLET formalism as combining multiform logical time and the ability of user-defined logical clocks, together with quantitative durations in which a certain amount of computation can be spread in any instants of a given interval, providing it does not overflow it.

A. The Synchronous-Reactive Model (SR)

The Synchronous-Reactive model, implemented by synchronous languages [2], totally abstracts execution time to focus on logical instants, allowing both determinism and concurrency. For that, computations react simultaneously and instantaneously with the tick of a global common logical clock as shown in figure 2a, which can be further refined to give multiple logical clocks. The synchronous hypothesis, then, ensures that if every computation is bounded by the next reaction, then physical time can be safely ignored as shown in figure 2b. Combining synchrony and concurrency allows a very expressive formal model while keeping it very simple [2]. Hence, SR model is well adapted for formal verification.

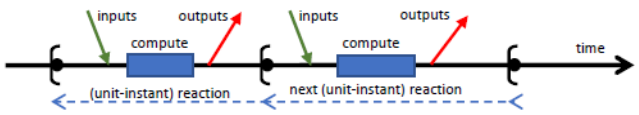
SR model has been introduced in languages such as *Esterel* [7] which is mainly imperative and *Lustre* [8] which is declarative. The compilation of synchronous languages is, however, quite complex. First, instantaneous communication makes the compilation for parallel platforms (i.e. multicore, distributed) quite hard and can produce causality issues. Secondly, execution time is usually limited to the reaction time of the system. Thus, applications with non negligeable execution time (i.e. also called the long period problem) can be rejected during the schedulability analysis. Nonetheless, languages based on the synchronous model such as *Prelude* [6], can adress this problem, but are usually limited to specific patterns such as a multi-periodic synchronous model.

B. The Logical Execution Time Model (LET)

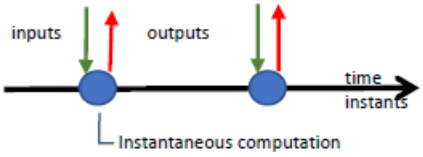
The Logical Execution Time paradigm abstracts physical time through a sequence of instants, similarly to the synchronous approach. However, contrary to the latter, computation takes time and is not considered instantaneous. For that, each computation must fit in a logical interval, called LET interval [1]. 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 termination (see Figure 2c). As communication is only done on predefined instants, computations behave like they always take the same time, which is the LET interval duration (see Figure 2d). As such, the determinism property is ensured.

The LET paradigm has been initially introduced in the *Giotto* [1] language in which each task correspond to a periodic LET interval. The language can also express global modes that allows to change from a given set of task frequency to an other. *Timing Definition Language* (TDL) [9] extends *Giotto* via a decomposition in concurrent modules. Each of them defines a set of task and their frequency, similarly to *Giotto*, but a module can also change its local mode independently of other modules. *Giotto* has also been extended with events in the *xGiotto* language [10] through the mechanism of event scoping. In all these approaches, the bounds of LET intervals are either defined using chronometric pseudo-physical durations or events (i.e. in *xGiotto*). This paper proposes a more abstract and homogeneous model of time, that is, the use of multiple logical clocks.

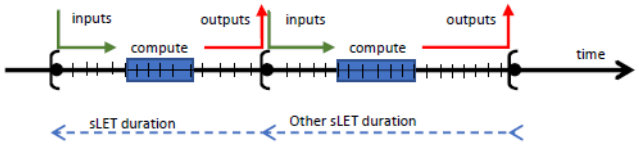
In a way, LET extends logical time with the concept of logical durations. This allows a precise schedulability analysis due to an increased execution time variability. In turn, this also makes compilation to parallel platforms easy and causality issues of the SR model are avoided. However, as LET forbids



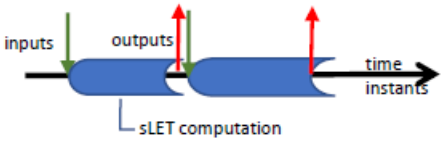
(a) Physical view of the Synchronous-Reactive model



(b) Logical view of the Synchronous-Reactive model



(c) Physical view of the (synchronous) Logical Execution Time model



(d) Logical view of the (synchronous) Logical Execution Time model

Fig. 2: Multiform Logical Time models: physical vs logical views

instantaneous communications, its theoretical expressivity is reduced compared to the SR model.

C. sLET as a synchronous extension of LET

Synchronous Logical Execution Time (sLET) extends the classical LET paradigm with the concept of logical clocks. As such, interval bounds can be triggered by clock ticks while in classical LET, interval duration is usually defined by a constant fixed duration. However, as in LET, communication can only happen on interval bounds, which ensure temporal determinism. Being closer to the SR model, sLET is actually fully compatible with the synchronous hypothesis, that is, computations can happen instantaneously on the activation date of the interval as long as outputs are delayed to the end of the interval. This model is actually a generalization of the *Psy* model introduced in the OASIS framework since the 90's [3].

A logical clock, in the sense of Lamport [11], abstracts time through a series of events called *clock ticks*. The sLET paradigm adds to LET a finite set of logical clocks $\mathbb{C}$ on which LET interval boundaries can be based. We will call *clock event* an event of the form $n \times c$ where $n \in \mathbb{N}^*$ and $c \in \mathbb{C}$. This event basically means that we have to wait n ticks of the clock c. An interval termination is then defined with a *clock event* which may also define the activation of a next interval. As an example, if an interval termination is defined with $n_t \times c_t$, then after its activation, it should wait for n_t ticks of the clock c_t. Note that if all intervals use the same clock, then clock events actually express fixed constant durations with respect to that clock which is similar to the classical LET model.

sLET keeps with the LET idea of imposing at an early phase of logico-functional design some fixed interval durations at which boundaries the I/O external behaviors will be supposed to occur. Then there is flexibility on when exactly computations are scheduled, as long as they remain inside these bounds. Next, we will introduce the PsyC languages foundations, bringing syntax to the sLET view. In particular, PsyC allows conditional durations, in alternative "if-then-else" behaviors. Note however that classical LET, as defined in *Giotto*, can actually be expressed easily with sLET, that is, using only one logical clock mapped to real-time. The converse is not true. Indeed, even considering that all logical clocks are derived using affine relation from a unique global clock, sLET interval might have different duration depending on the current global state.

III. The PsyC language: abstract syntax and formal semantics

A. Language description

In this section, we give a brief description of *PsyC*, a language developed by the company Krono-Safe, as an illustration of the sLET paradigm. This language is implemented in the ASTERIOS product which provides a set of tools to design safety-critical real-time software. Such applications can then be certified at the highest level of criticity for the avionic domain (DAL-A, DO-178C). ASTERIOS and *PsyC* are inherited from the OASIS and PharOS projects coming from the CEA. Initially, *PsyC* (for *Parallel SYnchronous*), in OASIS, was presented as a timed-triggered approach for safety-critical real-time software under a model called *Psy* [3]. This model is actually very close to LET, and synchronous LET generalize both of them. The modern version of *PsyC* is actually better characterized with synchronous LET due to the existence of multiple logical clocks.

A *PsyC* application is composed of a fixed set of tasks called *agents* that are formed of sLET intervals. The content of *agents* is composed of C code that is extended with special instructions like `advance` n `with` c which specifies the boundaries of sLET interval. Informally, its semantics is to advance the logical time of n ticks of a clock c. Such clocks have to be defined in the application as an affine relation with

respect to another clock. They are actually two kinds of logical clocks in *PsyC*, *sources*, which are defined externally, and *clocks*, which basically refine *sources*. In practice, most of the applications define only one *source* called *realtime* which is mapped on real time. The Figure 3 shows an example of an agent using multiple clocks and a conditional computation in the *PsyC* language. A possible timeline is shown in figure 4. Such pattern with multiple clocks allowed by sLET couldn't be expressed, as is, in the classical LET model.

```
source realtime;
clock c2 = 2 * realtime;
clock c3 = 3 * realtime;

agent Ag(uses realtime, starttime 1 with c2)
{
  body start /* infinite loop */
  {
    /* computation A */
    advance 1 with c3;
    if (/* condition */) {
      /* another computation B */
      advance 1 with c2;
    }
  }
}
```

Fig. 3: Example of a PsyC agent

Inter-agent communication is performed through a dedicated channel called *Temporal Variable*. Multiple agents can read a *temporal variable* but only one can write into it. According to sLET, inputs and outputs in a sLET interval can only be made on sLET interval bounds. Hence, data emitted by an agent is made visible (i.e. the temporal variable is updated) on the instant described by the next `advance` instruction. Futhermore, *Temporal Variables* also have another layer of sampling (defined with a clock) and values are read with the expression $\$[n]var$ which denotes the n^{th} last sampled value.

The Figure 5 describes the abstract syntax of a subset of *PsyC* which is necessary to introduce the formal semantics described in the next section. Only specific *PsyC* constructions are described and the syntax of C expressions is omitted for simplicity. Also, we add a specific statement skip that does nothing (i.e. in the C syntax, this could represented as a semi-

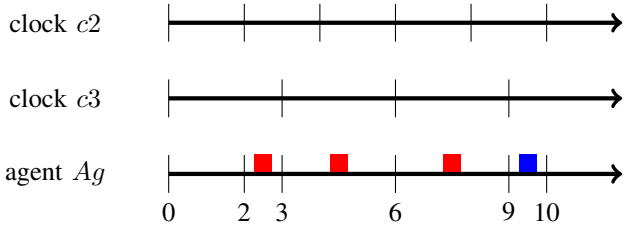


Fig. 4: Possible timeline of example in fig 3. A Red frame represent computation A while a blue frame represent computation B.

$$
\begin{array}{lcl}
\textit{application} & ::= & \textit{decl}^* \\
\textit{decl} & ::= & \mathsf{source}\ c \\
 & | & \mathsf{clock}\ c_1 = n_1 * c_2 + n_2 \\
 & | & \mathsf{temporal}\ id\ =\ value\ \mathsf{with}\ c \\
 & | & \textit{agent} \\
\textit{agent} & ::= & \mathsf{agent}\ id\ \textit{body}^* \\
\textit{body} & ::= & \mathsf{body}\ id\ \textit{stmt} \\
\textit{stmt} & ::= & id\ :=\ exp \\
 & | & \mathsf{skip} \\
 & | & \mathsf{advance}\ n\ \mathsf{with}\ c \\
 & | & stmt_1\ ;\ stmt_2 \\
 & | & \mathsf{if}\ (exp)\ stmt_2\ \mathsf{else}\ stmt_3 \\
 & | & \ldots
\end{array}
$$

Where $n, n_1 \in \mathbb{N}^*$, $n_2 \in \mathbb{N}$, v is the initial value of the temporal variable and c_i are clock identifiers.

Fig. 5: Abstract syntax of our *PsyC* subset

colon). This will be helpful in the semantics to express that a statement doesn't have any work left to do. Futhermore, both the `starttime` and the source parameter of the agent are omitted because the former could be defined as an `advance` and the latter could be deduced from the clocks. However, we assume that an agent could only use clocks that are derived from a unique source. Nonetheless, different agents might use different sources.

B. Formal Semantics

In this section, we introduce the formal semantics of a subset of the *PsyC* agents thanks to term rewriting. Basically, a term can be rewritten sucessively, with respect to a given set of formal rules, to form a symbolic execution that serves as an oracle. The rules are given using the Structural Operational Semantics (S.O.S.) approach introduced by Plotkin [12] and later adapted to synchronous languages [7]. The idea behind S.O.S. is to give the rewriting relation of a term with respect to the rewriting of its parts. This is usually defined through a set of inference rules. The behavioral semantics of Esterel extends S.O.S. through the use of an integer encoding the type of the transition, i.e. whether the transition takes time or is instantaneous [7] (i.e. fits entirely inside an instant reaction).

As described above, the semantics of *PsyC* is described in this paper using two relations, one that makes time progress, and one that is not explicitely timed, and should be considered as conceptually guaranteed to fit in the time interval closed by the next time-progress behavior. This vision can be considered as borrowed from discrete-event simulation models. While the semantics allows many later schedules for the internal behaviors before next time advance (a property exploited by the *PsyC* compiler to optimize their time allocations according to other criteria), one may also consider the (valid) interpretation where all computations are made in the initial instant, than

will allow the translation to synchronous languages of the next section.

Consider that the configuration of an agent (i.e. its state) can be described as a pair $\langle E, T\rangle$ where E is the assignements of its local variables and T the assignments of its displayed temporal variables (i.e. its outputs). Based on this, we define an instantaneous transition as following:

$$\langle E, T\rangle \vdash s \longrightarrow < E', T' > \vdash s'$$

This transition denotes that a statement s can be rewritten in s' instantaneously while the configuration $< E, T >$ is updated to $< E', T' >$ with respect to s. Based on this definition, we can define the semantics of the first basic statement, the assignation:

$$\langle E, T\rangle \vdash x \ := \ exp \longrightarrow \langle E', T\rangle \vdash \mathsf{skip} \qquad \text{(assign)}$$

where $E' = Update(E, v, exp)$.

The $assign$ rule evaluates its expression and updates its local variables accordingly using a function that we call $Update()$. This statement is logically instantaneous because it is executed at the beginning of a sLET interval to read the correct input values. Also, it is reduced to skip as there is no work left to do. The $advance$ rule is a little bit more complicated as this statement makes time progress. Recall however that clocks used in an agent by advance statements are all derivated from a unique source. Thus, we first instantaneously rewrite the $advance$ statement with respect to its parameters (i.e. number of ticks and clock) to a counter of source ticks that we define with the pseudo statement *Remains*. This pseudo statement takes an absolute duration in parameter (with respect to the agent source) that is computed with the function $Duration$ with respect to the $advance$ parameters, and the current source date.

$$\langle E, T\rangle \vdash \mathsf{advance}\ n\ \mathsf{with}\ c \longrightarrow \langle E, T\rangle \vdash \textit{Remains}(N) \qquad \text{(advance)}$$

where $N = Duration(n, c, date)$

The *Remains* pseudo statement defined above can then be used to make time progress. For that, we define a new transition that takes one time unit of the agent source (symbolized by the double arrow shape):

$$\langle E, T\rangle \vdash s \Longrightarrow < E', T' > \vdash s'$$

Remains is then defined with two different rules. The first one, *Remains-1*, decreases the counter when its greater than 1. When the counter is equal to 1, *Remains* is rewritten to skip as there is no work left to do and temporal variables are updated. As other agents can only read the latter (and not the agent local environment), this actually means that outputs can only be made visible at the end of a sLET interval, which is actually the semantics of the sLET paradigm.

$$\langle E, T\rangle \vdash \textit{Remains}(N) \Longrightarrow \langle E, T\rangle \vdash \textit{Remains}(N-1) \qquad \text{(Remains-1)}$$

if $N > 1$

$$\langle E, T\rangle \vdash \textit{Remains}(1) \Longrightarrow \langle E, T'\rangle \vdash \mathsf{skip} \qquad \text{(Remains-2)}$$

where $T' = UpdateOutputs(T, E)$.

The rules defined above can be composed quite easily. First, when there are executed in sequence, multiple instantaneous transitions can be represented with one big instantaneous transition and when they are followed by a non instantaneous transition, they can be represented with a big non instantaneous transition. In other words, the semantics of an agent can be represented with only big non instantaneous temporal transitions, which is interesting to show the expected temporal behavior of the agent. Second, the rules defined above only show simple statements, but actually, this can be extended very easily to control flow statement (e.g. condition statement) given that the type of transition is propagated correctly.

To illustrate a little bit the semantics above, we show a very basic example of a sLET interval with an output variable called tv. Let's consider the following PsyC statements: $tv \ := \ 2$; advance 2 with $c2$ Let's assume that $c2$ is a strictly periodic clock with period 2 and offset 0, and the current date of the agent is odd. Then, with respect to the semantics rules, we have the following execution:

$$\begin{aligned}
&\langle tv =?, tv =?\rangle \vdash tv \ := \ 2 \ ; \ \mathsf{advance}\ 2\ \mathsf{with}\ c2 \\
&\longrightarrow \langle tv = 2, tv =?\rangle \vdash \mathsf{advance}\ 2\ \mathsf{with}\ c2 \\
&\longrightarrow \langle tv = 2, tv =?\rangle \vdash \textit{Remains}(3) \\
&\Longrightarrow \langle tv = 2, tv =?\rangle \vdash \textit{Remains}(2) \\
&\Longrightarrow \langle tv = 2, tv =?\rangle \vdash \textit{Remains}(1) \\
&\Longrightarrow \langle tv = 2, tv = 2\rangle \vdash \mathsf{skip}
\end{aligned}$$

This execution shows that the communication is correctly updated at the end of the interval and its duration is actually 3 as it depends on the current date (e.g. if the date was even, then the interval should have lasted 4 ticks). Also, one can notice that such pattern is not possible in classical LET languages (e.g. *Giotto*, *TDL*) as they usually can only represent duration on a single clock.

Construction	PsyC syntax	Esterel Translation
Clock	$clock\ c = n_1 * c_p + n_2$	$await\ immediate\ n_2;$ $loop$ $\quad emit\ c$ $each\ n_1\ c_p$
Temporal Variable	$temporal\ tvar\ =\ init\ with\ c$	$signal\ tvar = init\ in$ $\quad run\ Sampler[tvar/In, tvar_0/Out, c/Clk]\ \|\|$ $\quad ...$ $endsignal$
Local Variable	$int\ var\ = init;$	$var\ private\_var\ :=\ init\ in\ ...$
Assignation/expression	$tvar = exp;$	$private\_tvar\ :=\ exp;$
Temporal variable reading	$\$[0]tvar$	$tvar_0$
Condition	$if\ (exp)\ s1\ else\ s2$	$if\ (exp)\ s1\ else\ s2$
Advance	$advance\ n\ with\ c$	$await\ n\ c; emit\ tvar(private\_tvar)$
Body (cyclic case)	$body\ p$	$loop\ p\ end$
Agent composition	$agent\ ag1\ ...agent\ ag2\ ...$	$run\ ag1[...]\ \|\|\ run\ ag2[...]\ \|\|\ ...$

Fig. 6: Some Esterel translation patterns (var is a private variable and $tvar$ is a (shared) temporal variable)

IV. Consequences

A. Synchronous semantics of sLET via Esterel translation

To show the direct relation between sLET and the SR model, we can encode the semantics of *PsyC* into a synchronous language like *Esterel*. We choose *Esterel* because its syntax is quite close to *PsyC*, that is, both languages are control-flow and imperative. Basically, the approach is to focus on the logical behavior of synchronous LET and to totally abstract actual executions, i.e. to consider them (logically) instantaneous. Thus, of course, this kind of simulation is not representative of the actual operational behavior where each computation takes time. Nonetheless, sLET (similarly to classical LET) allows multiple valid schedules that are mutually equivalent, that is, that form an equivalence class with respect to logical clock ticks. The synchronous semantics is then actually a particular instance where in a sLET interval:

1) all computations are executed during the first instant, thus inputs are read synchronously to the start of the interval;
2) time is advanced to the end of the interval;
3) outputs are produced synchronously to the end of the interval.

This abstraction should be sufficiant to express properties at the model level as actual computations should not impact temporal properties of a sLET design. While this approach focus on synchronous LET, as the latter is a generalization of the classical LET model, this approach could be adapted easily to other LET languages.

Esterel is an imperative synchronous language control-flow based. In particular, it allows imperative concurrency, that is, handling multiple instruction pointers at a same instant and preemption where a computation can be aborted when a given signal is present. The only communication means available in *Esterel* is through instantaneous signals which can have a status, *present* or *absent*, and a value. A signal which only has a status is said to be *pure*, otherwise, it is said to be *valued*. The instructions are basically divided into two categories similarly to *PsyC*:

- the instantaneous instructions, like $emit\ s$ which emits a given signal s;
- and temporal instructions, like $pause$ or $await\ s$ which respectively wait for the next instant and for the next instant where signal s is present.

For a full definition of the Esterel language, we invite the reader to consult [7].

In a *PsyC* application, different elements are actually composed in a synchronous parallel fashion. Assuming that *PsyC* clock ticks are represented as *Esterel* pure signals (i.e. either the clock ticks or not), we have to represent clock generation, temporal variable sampling and agent behavior. As explained at the beginning of section III, a *PsyC source* clock is actually an external logical clock, and is thus represented as an *Esterel* pure signal input. Clock generation basically refines a clock tick, i.e. a pure signal, based on an affine relation. A temporal variable is represented as a persistent valued signal (i.e. which is always present) and is sampled on the signal corresponding to its clock. And finally, an agent is almost translated as is. The control-flow is translated in its equivalent form in *Esterel* and basic *body* patterns can be translated either to sequence or to infinite loop depending on the next body to be executed. We do not cover advanced *body* patterns in this paper for simplicity but this could be adapted with some more advanced *Esterel* control-flow patterns. All variables of an agent, either local to the agent or temporal variables, are translated to *Esterel*

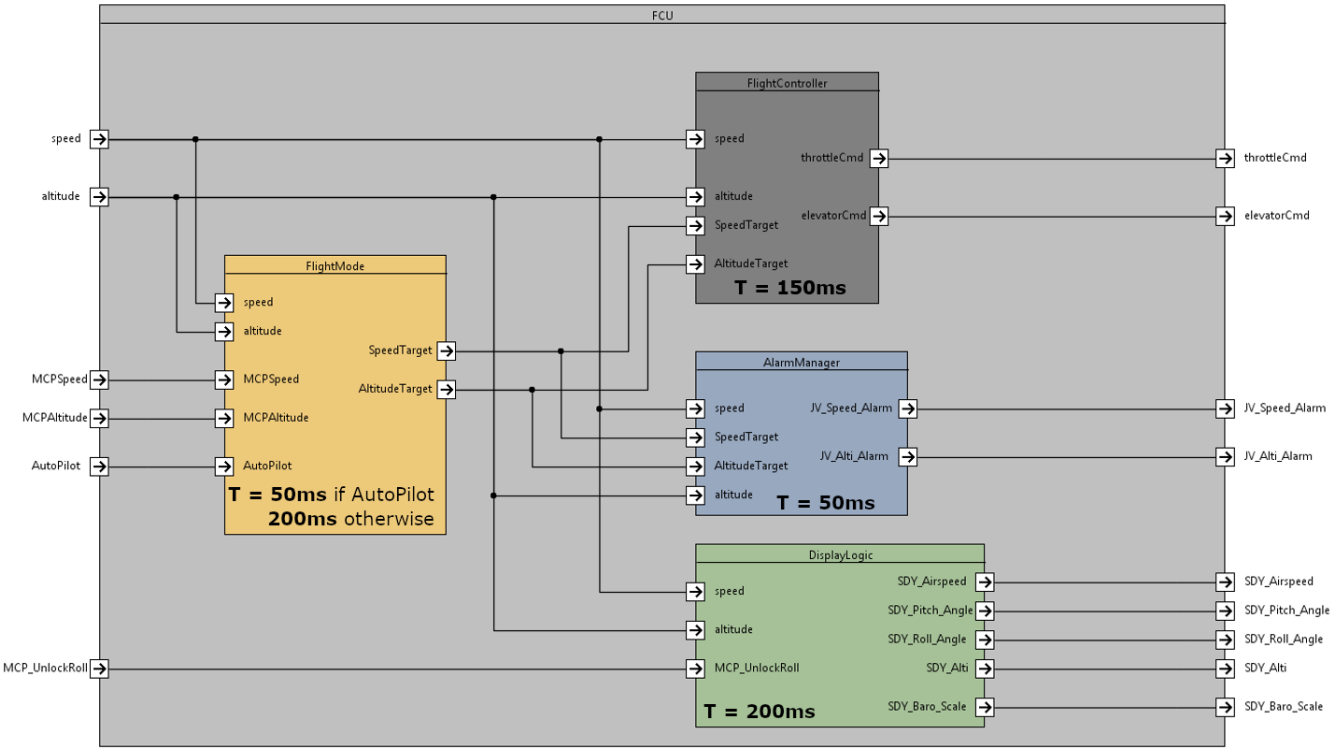


Fig. 7: The Flight Control System use-case

variables. The *advance* statement is translated to the *await* statement followed immediately by an update of the temporal variables, that is, the valued signal of each temporal variable in output of the agent is updated with its local variable value. This is actually the main difference from the synchronous model, communication cannot be updated instantaneously. All the translation patterns described above are sketched in the table 6.

B. Synchronous Observers and Formal Verification

The main interesting consequence of the *Esterel* translation is to apply techniques and tools coming from the synchronous community. Among them, formal verification has been successfully applied on the synchronous approach due to two main things: 1) the synchronous composition semantics, the state space is usually quite reduced comparing to asynchronous approaches and 2) symbolic model-checking allows to scale verification techniques even more, using BDD or SMT techniques. Futhermore, in synchronous languages, properties that are usually modeled using temporal logics, can also be modeled directly in the language using the so-called synchronous observer approach [13]. Observers only observe the state of the application and defines which state is acceptable or not. Classicaly, a signal *Error* is emitted if an error is detected and verification is reduced to a reachability analysis on this signal.

In *PsyC*, the typical kind of property that we might want to check applies on the different temporal variables rhythms. Typically, due to the sampled buffered semantics of temporal variable, if the producer and the consumer are not at the same rhythm, produced values can be lost or consulted values might be duplicated. While the second scenario is usually not a problem, the first one might be one. Of course, this kind of property is quite simple, but this opens the possibility to more evolved properties like freshness constraints of temporal variables or end-to-end latencies.

V. Use-Case

A. Description

As highlighted in the introduction, languages based on LET, such as *PsyC*, focus on the architecture description at integration level of real-time systems. In other words, they allow to express how multiple functional components connect and what is their real-time behavior. In particular, it is possible to express multi-rate behavior. Typically, in industrial applications, a synchronous language such as Scade, is used to design the functional components of the system and a language like *PsyC* can be used to specify the real-time software integration of these components.

To reflect this approach, the following use-case is based on an example found in the Scade Suite software. Consider a basic

Flight Control Software in which you expect the following behavior:

- given an altitude and a speed command, a flight controller implements control laws for altitude and speed with respect to altitude and speed sensors;
- depending on the flight mode, i.e. AutoPilot or Manual, the commands given to the flight controller is either the input commands or the commands of the previous instant;
- regulation alarm is raised if the altitude sensor (respectively the speed sensor) is too far from the altitude command (respectively the speed command).

The system architecture is shown in the figure 7. Basically, each component correspond to a function detailed above. Each functional component can either be implemented manually or with a *Scade* sheet. To illustrate a little bit more this approach we show in figure 8 the *Scade* sheet corresponding to the mode handling. The mode handling is modeled with a finite state machine with only two states, the *AutoPilot* state and the *Manual* state corresponding to the *AutoPilot* mode and *Manual* mode described at the beginning of the section. The control laws are based on PID controllers but we will not dive into the details as it is beyond the scope of this paper.

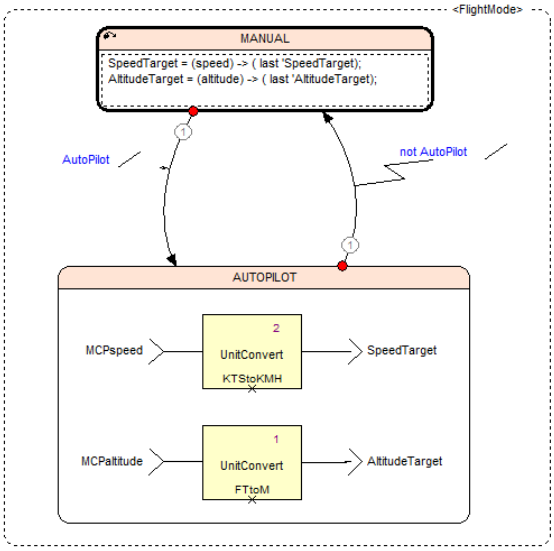


Fig. 8: Mode Controller of FCS

For simulation purpose, the example also contains a plane simulation that takes throttle and elevation commands (from the controller), simulates the actuator response as well as the aerodynamics behavior and yield updated speed and altitude sensors values (i.e. to the controller). Running such a simulation in *Scade* typically ignores real-time constraints and every components run at the same rate. While this is totally justified for simulation, it's is not necessarily the case for compilation. Indeed, in practice, different components might have different real-time behaviors, depending on available resource, as well as different mapping constraints (typically for multicore target). Thus, in the next section we will show how *PsyC* can be used to integrate seamlessly multi-rate functional components.

B. Architecture Description in PsyC

We choose a very simple mapping where each components described above are directly mapped to an individual agent. Futhermore, each communication signal is mapped to an individual temporal variable. Of course, while only one agent can write a temporal variable, multiple agents can read it. Thus, a temporal variable is not duplicated for each reader component. The temporal behavior of a component in an agent is the following:

- The reset function of the component generated from Scade is executed once with a given phase constraint (not specified here) and;
- The cycle function of the component generated from Scade is executed indefinitely *after* the reset function, on a given rate.

As an illustration, the code in 9 describes the temporal behavior of the controller component. The inputs and outputs of the cycle function are transfered with the help of global variables which is the usual calling convention with code generated from Scade.

```
agent FlightController(starttime 1 with clk_starttime)
{
  body start {
    next cycle;
    FlightController_reset_FlightControl();

    advance 1 with clk_10000_0 ;
  }
  body cycle /* infinite loop */ {
    altSensor = $[0]temporal_altitude;
    speedSensor = $[0]temporal_speed;
    speedSTarget= $[0]temporal_speedTarget ;
    altTarget= $[0]temporal_altitudeTarget ;

    FlightController_FlightControl();

    temporal_throttleCmd = throttleCmd;
    temporal_elevatorCmd = elevatorCmd;

    advance 1 with clk_150000_0 ;
  }
}
```

Fig. 9: Agent corresponding to the *FlightController* component

Chosing the right clocks is primary in the integration flow. First, the clocks should be sufficiently fast to ensure the stability of the different functional behaviors (typically control laws). Second, the clocks should be not too fast to ensure that the platform has enough resource to execute the system.

The expected clock periods are described in the figure 7. We consider that *FlightController* is a heavier task and hence, has a period three times bigger (e.g. $150ms$) than the other tasks that run with a period of $50ms$.

As explained in section III, *PsyC* also allows more advanced control flow patterns. As an example, the agent *FlightMode* may need to relax the deadline when the *AutoPilot* mode is disabled as specified in 7. This is illustrated in the listing 10. This kind of pattern doesn't have any impact on the functional behavior (i.e. commands are unchanged in manual mode), except during mode transition, and this allows to relax the cpu load.

```
agent FlightMode(starttime 1 with clk_starttime)
{
  body start {
    next cycle;
    FlightMode_reset_FlighControl();

    advance 1 with clk_10000_0;
  }
  body cycle /* infinite loop */ {
    /* From sensors */
    altitude = $[0]temporal_altitude;
    speed = $[0]temporal_speed;
    /* From user commands */
    AutoPilot = $[0]temporal_AutoPilot;
    MCPspeed = $[0]temporal_MCPspeed;
    MCPaltitude = $[0]temporal_MCPaltitude;

    FlightMode_FlightControl();

    temporal_speedTarget = SpeedTarget ;
    temporal_altitudeTarget = AltitudeTarget;

    if (AutoPilot) {
      advance 1 with clk_50000_0 ;
    } else {
      advance 1 with clk_200000_0 ;
    }
  }
}
```

Fig. 10: Agent corresponding to the *FlightMode* component with conditional deadline

C. Esterel Translation

The *PsyC* architecture given above allows efficiant compilation. However, as illustrated, the right clocks has to be used to respect the temporal behavior. For that, one can use the synchronous semantics of *PsyC*, that is, its translation to Esterel to analyze the application's behaviors. Of course, theorically, such analyze could be made in the Scade application simulating the multi-rate behavior. However, we think that such approach should be made at the integration level. Futhermore, the whole application might not be available in Scade, i.e. either coming from different teams or written manually.

The listing 11 shows the translation of the *Mode* agent. We remove the function calls for simplicity but not the communication mechanism. In the main loop, *altitude* is read instantaneously then, some computation is made and the *AltitudeTarget* is set to a private variable. Finally, after the `advance`, that is, the `await` here, this private variable is made visible through the `emit` statement. Special communication events are also added to show when communication is read and when communication is produced in this module. This will be needed later for verification.

```
module FlightMode:
[
  await clk_starttime;
  /* FlightMode_reset_FlightControl(); */
  await 1 clk_10000_0;

  loop
    emit FlightMode_Read;
    altitude := ?temporal_altitude_0;
    speed := ?temporal_speed_0;
    AutoPilot := temporal_AutoPilot_0;
    MCPspeed := temporal_MCPspeed_0;
    MCPaltitude := temporal_MCPaltitude_0;

    /* FlightMode_FlightControl(); */

    private_altitudeTarget := AltitudeTarget;
    private_speedTarget := SpeedTarget;

    if autoPilot then
      await 1 clk_50000_0;
    else
      await 1 clk_200000_0;
    emit temporal_altitudeTarget(
      private_altitudeTarget);
    emit temporal_speedTarget(
      private_speedTarget);
    emit FlightMode_Write;
  end loop
]
```

Fig. 11: Translation of *FlightMode* agent in Esterel

D. Synchronous Observers

The typical properties we might want to check with synchronous observers on such system are clock tick event ordering and clock tick event synchronizations. Typically, in *PsyC*, if the ticks of two tasks are alternating, and assuming temporal variable is at the speed of the producer, then, no value is lost. In a clock algebra such as CCSL, the predicate $c_1 \sim c_2$ states that the ticks of the logical clocks c_1 and c_2 alternate. Such constraints can be easily implemented in Esterel observers as shown in [14]. Thus, verification of such properties is then reduced to a reachability analysis on a signal representing an error state that we will call *Error* in this section.

In our example, let's consider that we want to ensure that all commands values are taken into account once by the *Alarm* agent. This functional chain is thus composed of the *FlightMode* agent, the *SpeedTarget* and the *AltitudeTarget* temporal variables, and the *Alarm* agent. Considering that the corresponding tick events have the same name, we have the following constraints to check to ensure that no values are lost:

$$\begin{aligned} \textit{FlightMode\_Write} &\sim \textit{SpeedTarget} \wedge \\ \textit{FlightMode\_Write} &\sim \textit{AltitudeTarget} \wedge \\ \textit{SpeedTarget} &\sim \textit{Alarm\_Read} \wedge \\ \textit{AltitudeTarget} &\sim \textit{Alarm\_Read} \end{aligned}$$

Based on [14], an *Esterel* observer for *Alternate* can be derivated. To fit our purpose, alternation strictness (whether two synchronous tick events could be synchronous or not) should be adapted to our communication mode. That is, the transition from event A to B in $A \sim B$ can be instantaneous but not the transition from B to A. The verification of the constraints described above can be written in instanciating the `Alternate` observer for each of the constraint with the help of parallel composition:

```
run Alternate[signal FlightMode_Write / A,
              SpeedTarget / B,
              Error/Error]; ||
run Alternate[signal SpeedTarget / A,
              Alarm_Read / B,
              Error/Error]; ||
run Alternate[signal FlightMode_Write / A,
              AltitudeTarget / B,
              Error/Error]; ||
run Alternate[signal AltitudeTarget / A,
              Alarm_Read / B,
              Error/Error];
```

In our scenario, this property is verified only when the *AutoPilot* mode is set as the *Alarm* agent might read multiple times the same values of the *FlightMode* agent when *AutoPilot* is disabled. The observer could be adapted easily to enforce alternation only when the mode is set. Also, note that this property does not enforce that all clocks should be synchronous (i.e. at the same rate). In fact, the agent *Alarm* might have some offset with *FlightMode* still without losing any value.

VI. Conclusion

This article presents a new paradigm called sLET that builds upon both the synchronous and the LET paradigm. Such unification allows to re-use formal analysis techniques coming from synchronous languages while keeping the simple compilation and efficient schedulability analysis of LET. sLET (as the classical LET paradigm) allows multiple valid schedulings that are mutually equivalent (in the logical time realm), as long as all of the computations fits in their sLET intervals (the *PsyC* compiler actually using one of these solutions also satisfying actual physical computation durations). The synchronous semantics is then actually a particular instance where computations are executed "in a single instant" and outputs are delayed. Consequently, this shows that one can use analysis based on the synchronous approach to reason on languages based on LET (e.g. *Giotto*, *xGiotto*, *TDL*) at the logical level; in particular, it unveils the possibility to re-use synchronous formal verification techniques for these languages. The sLET paradigm is implemented by the *PsyC* language which is part of the ASTERIOS framework, produced by KRONO-SAFE.

This work opens twofold perspectives: firstly, express more advanced properties, such as end-to-end latencies or data freshness, in a simpler way; secondly, synchronous verification techniques should be adapted more thoroughly to be efficient. In particular, (s)LET intervals are composed of instants that only make time progress and thus, can be optimized for verification. Finally, while our main focus is formal verification, other well known techniques can be adapted to languages based on (s)LET, such as model-in-the loop simulation, automatic test generation or test coverage.

References

[1] C. Kirsch and A. Sokolova, "The logical execution time paradigm," in *Advances in Real-Time Systems*, 2012.

[2] A. Benveniste and al, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, pp. 64 – 83, 2003.

[3] V. David and al, "Safety properties ensured by the oasis model for safety critical real-time systems," in *SAFECOMP*, 1998.

[4] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From simulink to scade/lustre to tta: a layered approach for distributed embedded applications," *ACM Sigplan Notices*, vol. 38, no. 7, pp. 153–162, 2003.

[5] S. Bliudze, X. Fornari, and M. Jan, "From model-based to real-time execution of safety-critical applications: Coupling scade with oasis," in *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[6] J. Forget and al, "A Multi-Periodic Synchronous Data-Flow Language," in *11th IEEE High Assurance Systems Engineering Symposium*, 2008.

[7] G. Berry, *The Constructive Semantics of Pure Esterel*, 1996.

[8] D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *14th Annual ACM Symposium on Principles of Programming Languages*, vol. 178, 1987, p. 188.

[9] W. Pree and J. Templ, "Modeling with the timing definition language (tdl)," in *Automotive Software Workshop*. Springer, 2006, pp. 133–144.

[10] A. Ghosal, T. Henzinger, C. Kirsch, and M. Sanvido, "Event-driven programming with logical execution times," vol. 2993, 2004.

[11] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[12] G. Plotkin, "A structural approach to operational semantics," *J. Log. Algebraic Methods Program.*, 2004.

[13] N. Halbwachs and P. Raymond, "Validation of synchronous reactive systems: From formal verification to automatic testing," in *5th Asian Computing Science Conference on Advances in Computing Science*, 1999.

[14] C. André, "Verification of clock constraints: CCSL Observers in Esterel," INRIA, Research Report, 2010. [Online]. Available: https://hal.inria.fr/inria-00458847