Formal Semantics of the PsyC language

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

Formal Semantics of the PsyC language

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

RESEARCH REPORT

N° 9506

March 2023

Project-Team Kairos

ISSN 0249-6399 ISRN INRIA/RR--9506--FR+ENG

Formal Semantics of the PsyC language

Fabien Siron[*† ‡], Dumitru Potop-Butucaru[‡], Robert de Simone[‡], Damien Chabrol[*], Amira Methni[*]

Project-Team Kairos

Research Report n° 9506 — March 2023 — 34 pages

Version du 20 Fevrier 2023

[*] Krono-Safe
[†] Université Côte d'Azur
[‡] Inria

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Abstract: Time must be taken into account from the very start of the design of real-time systems. Because exact computation durations are usually not available in early design phase, multiple formalisms based on the concept of Multiform Logical Time have been introduced. The synchronous approach introduced a discretized abstraction of time based on logical clocks, on which computation happens logically instantaneously. While being very expressive, the synchronous approach suffer from compilation issues when mapping logical time to non-negligible real-time computation durations. In answer, the *Logical Execution Time* (LET) paradigm has been introduced to give such a compromise between the strong expressiveness of the synchronous approach and traditional task scheduling, making LET a perfect fit for the temporal architecture and the software integration of real-time systems. In this report, we consider a variation of LET extended with logical clocks, namely synchronous LET (sLET), implemented in the industrial language PsyC. The sLET approach — and thus PsyC — allows to specify computation to be executed inside a logical interval *up to* the next occurrence of a logical clock expression (i.e. in opposition to a constant duration). As sLET is based on both LET and the synchronous approach, we give in this report two operational semantics of the PsyC language: a native semantics (or big-step) based on the Structural Operational Semantics approach (S.O.S.) and a synchronous semantics defined by translation to the Esterel language. The former gathers all valid computation schedulings as an equivalence class, of which the latter provides a specific representative (as-soon-as-possible execution). We show the semantics equivalence of those two semantics at interval boundaries. While the native semantics is mainly used for compilation, future work will show how logical timing properties can be verified using those two semantics.

Key-words: Real-Time Systems, Multiform Logical Time, Logical Execution Time, Formal Semantics

Semantique formelle du langage PsyC

Résumé : Le temps doit être pris en compte dès le début de la conception des systèmes temps-réel. Du fait que les durées exactes des calculs ne sont généralement pas disponibles au début de la phase de conception, plusieurs formalismes basés sur le concept de Temps Logique Multiforme ont été introduits dans la littérature. L'approche synchrone introduit une abstraction discrète du temps basée sur des horloges logiques, sur lesquels les calculs s'effectuent logiquement instantanément. Bien que très expressive, l'approche synchrone souffre de difficultés de compilation lors de l'association du temps logique à des durées de calculs non négligeables. En réponse, le paradigme *Temps d'Exécution Logique* (LET) a été introduit comme compromis entre l'expressivité de l'approche synchrone et l'ordonnancement temps-réel traditionnel, ce qui fait que le LET est parfaitement adapté à la modélisation d'architecture temporelle. Dans ce rapport, nous considérons une variation du LET enrichi via l'usage d'horloges logiques, que nous appelerons LET synchrone (sLET), implémenté par le langage industriel PsyC. L'approche sLET — et donc le PsyC — permet de spécifier un calcul comme étant exécuté au sein d'un intervalle de temps logique *jusqu'à* la prochaine occurrence d'une expression d'horloge (en opposition à une durée constante). Du fait que sLET est basée à la fois sur LET et sur l'approche synchrone, nous donnons dans ce rapport deux sémantiques opérationnelles du PsyC : une sémantique native (ou grand pas) basé sur une approche *à la* Plotkin (S.O.S.) et une sémantique synchrone définie par traduction vers le langage Esterel. La première regroupe tous les ordonnancement valides sous forme de classe d'équivalence, dans laquelle la seconde donne un représentant spécifique (dès que possible). Nous montrons l'équivalence sémantique de ces deux sémantiques sur les bornes des intervalles. La sémantique native est principalement utilisée pour la compilation, tandis que des travaux à venir montreront comment vérifier des propriétés de temporisation en utilisant essentiellement la sémantique synchrone.

Mots-clés : Système Temps-Réel, Temps Logique Multiforme, Temps d'Exécution Logique, Sémantique Formelle

Contents

1 Introduction

Safety-critical real-time systems have to handle time in a predictable manner. That is, specified temporal requirements from the system specification should still hold in the final system. It is clear that time must be taken into account from the very start of the design. However, exact timing constants, e.g. computation durations, are usually not available in early design phases as they depend on the target. In answer, various formalisms, based on the Multiform Logical Time concept have been introduced to abstract those real-time durations. Then, specific analysis, usually called schedulability analysis (or time safety analysis) fill the gap between logical and physical time, that is, they ensure that physical computations satisfy their logical time constraints.

The *Synchronous-Reactive* approach (SR) introduced a discretized abstraction of time based on logical clocks in which reactions happen in discrete atomic instants, and so, logically instantaneously [2]. While multiple logical clocks may be used for specification expressiveness, the traditional operational semantics of synchronous languages usually imposes to expand the behaviors on a unique parent clock. Consequently, compilation of synchronous languages may suffer from the so-called "Long Task Problem". When different tasks of different rhythms are compiled, execution time variability is usually limited to a common cycle rate. More recently, the *Logical Execution Time* (LET) paradigm has been introduced to give a compromise between the strong expressiveness of the synchronous approach and the efficiency of traditional task scheduling [6]. For that, LET mandates to specify the actual logical duration a task has to fulfill based on a uniform pseudo-physical time. This forms LET intervals in which communication can only happen on its bounds. That is, inputs can only be consulted at the start of the interval and outputs can only be made visible to other tasks at the end. Consequently, as communication can only be made at predefined instants, LET ensure the *temporal determinism* property. In this report, we call synchronous Logical Execution Time (sLET), a variation of the classical LET paradigm in which interval bounds are based on logical clock ticks, as in the synchronous approach. As a consequence, the duration of an interval is specified as being "up to the next n^{th} tick" of a logical clock. It should be noted that such duration could not be constant across all reactions as it is specified *relatively* to a clock. In the simple case where there is only one universal clock used, then all logical durations become constants as in the case of the original LET paradigm. This makes sLET supersedes original LET. sLET (as classical LET) allows multiple valid schedule taking into account physical time that are mutually equivalent with respect to logical ticks. The synchronous semantics we give in this report is actually one of those in which computation happens logically synchronously to the start of sLET intervals and outputs are delayed to their end.

The sLET paradigm is implemented in the PsyC language which is an industrial language, developed by the company Krono-Safe, dedicated to the design of real-time safety-critical software [1].This reports introduces two operational semantics for PsyC execution:

- a *native* semantics is expressed in the celebrated structural operational semantics (SOS) format due to Gordon Plotkin [10]. This semantics could be qualified as *big step* since, for individual agents, it builds a single transition for a whole duration interval between two successive interval boundaries, called *Temporal Synchronization Points (TSP)*. The case of composite networks (of agents) is slightly more complex, involving intermediate states, as time intervals in distinct agents may not terminate on the same instants. However,

the dispatch of exact execution times for elementary computations are irrelevant, as long as they do not exceed prescribed interval boundaries. This semantics consider a native communication means using a valued event (e.g. similar to ESTEREL valued signal) that can be extended to multiple communication channels : *temporal variable* (i.e. a sampled shared variable) and *stream* (i.e. a fifo channel).

- in contrast, the second semantics called *synchronous* semantics expands temporally all behaviors onto the baseline discrete synchronous step and can thus be qualified as *small step*). It is defined by translating programs in a structural fashion into Esterel synchronous expressions. In essence it acts *as if* computations were all performed in the initial interval instant (but with effects remaining local), then waiting for the prescribed interval duration to terminate and show results, much as in discrete-event simulation frameworks, and respecting the LET philosophy.

Because computation effects can only be visible at interval boundaries (i.e. on TSPs), the second semantics can be abstracted and formally proven observationally equivalent to the first at this level. Consequently, it allows to retrieve compilation and verification techniques that have been defined for "pure" synchronous languages.

The outline of this report is organized as follows. Section 2 defines the synchronous Logical Execution Time paradigm and discuss its properties. Then, section 3 introduces the syntax of the PSYC subset considered in this report as well as some examples. Its native (or big-step) semantics is given in section 4 and 5 describing respectively the semantics of an individual agent and the semantics of a network of agents along with communication means. Section 6 defines an the synchronous semantics by translation to the ESTEREL language. Section 7 shows an equivalence criteria between both semantics and section 8 illustrates that result using a simple avionics case-study. Finally, section 9 discuss some implementation issues, due to the semantics, of some PSYC patterns.

2 Synchronous Logical Execution Time

2.1 Multiform Logical Time

The essential assumption of Logical Time is that time can be discretized according to a notion of base step (or instant), which is a subdivision of any behavior, and form the support for atomic reactions. Multiform Logical Time design, then, relies on the fact that designers can express timing of events using those logical clocks.

In this report, we call *logical clock* a totally-ordered sequence of events or instants at which it ticks [7]. They are used to measure event occurence dates in a system by replacing physical dates by logical sequencing. As such, they can be defined by some monotonic function from integer to integer as well as some boolean sequence that express at which instant they tick. The former representation could then be used to define algebraic constraints that will seek a solution by an assignment of instants to clock ticks, corresponding in general to a particular schedule resulting from logical time design constraints. In synchronous languages, as well as in synchronous LET, logical clocks are usually related by some expressions or constraints. Thus, they are partially ordered.

Definition 1. *A logical clock c is defined by an infinite totally-ordered sequence* $(t_i)_{i\in\mathbb{N}}$*. A finite set of clock C can be constrained using precedence and simultaneity relations (e.g. periodicity).*

Logical clocks can be associated with any events. Consequently, synchronous languages like ESTEREL use them to represent any countable units like seconds, meters, laps ... As any units

(or sequence of events) can represent a unique logical time, the concept has been called *multiform logical time* in the litterature. As an automotive exemple, an engine can be modeled using two main logical clocks, one that describes a discrete physical clock and one that describes the angular position of a crankshaft. Note that many concrete engineering frameworks that use a the simple notion of *nano* or *milli*-second to break down atomic steps fall into this category, as long as these units are only used to express constraints/requirements put on specifications; in general, logical time step need not be associated to a pseudo-physical unit. Note, also, that the same abstract discretization assumption is reported in the notion of tagged-systems [8]. While classical LET approaches does not really build on *logical clocks* and *multiform logical time*, the sLET paradigm builds on both approaches in the same spirit as classical synchronous languages.

2.2 Functional and temporal determinism

It is highly desirable for industrial time-critical systems that they behave in a functionally deterministic (reproducible) and temporally deterministic (predictable) manner. Determinism can be sorted as internal; or external, depending on its source:

- internal non-determinism may rise from the fact that conditions on values guarding the control flow choices are uninterpreted. This is usually the case in early design phases, and at compilation time, when programs represent their whole sets of different executions. Internal functional non-determinism may lead to temporal non-determinism, as distinct execution branches may "cost" different durations;
- external non-determinism may come from uncertainly on input/clocks arrival times; if a sequential component is allowed to be sensitive to more than one input that may occur concurrently, then whichever arrives first may disable the dependance on others, together with their effects. Requesting the absence of such conflicts has been considered under many names in different frameworks (of *input guards* in CSP, *conflict-freeness* in Petri Nets, *monotonicity* in Kahn Process Networks, *latency-insensitity* in high-level hardware synthesis languages, *endochrony* in reactive synchronous formalisms . . .). It is important to note here that temporal external non-determinism can lead to functional non-determinism as well, with phenomena known as timing anomaly at microarchitecture level [9] or priority inversion in real-time scheduling theory [13].

Enforcing uniform time boundaries at early design time level, as a requirement that will have to be respected and enforced later by implementation, is the main purpose of so-called Logical Exceution Time (LET) formalisms. Since the timing values used at this level do not need to be expressed in terms of physical time (which at this level is just as arbitrary values as any other sporadic events), then synchronous reactive languages (that were meant originally to prescribes exact activation logical instants), can be interpreted to provide boundary interval instants. This leads to the combination of synchronous LET (or sLET) described below.

2.3 The synchronous Logical Execution Time paradigm

Classical Logical Execution Time, introduced in languages such as *Giotto* [6] or *TDL* [12], basically defines the logical duration a task have to fulfill. This forms logical intervals in which communication can only happen on its bounds. Inputs are read at the beginning of the LET intervals while outputs are made visible at the end of them. Consequently, exact computation duration is totally abstracted by the logical duration of the interval. This ensures that, assuming that computation fits in the interval, temporal behavior is completely deterministic.

Synchronous Logical Execution Time (sLET) extends the classical LET paradigm with the concept of logical clocks. A logical clock, in the sense of Lamport [7], abstracts time through a series of totally-ordered event called *clock ticks*. These logical clocks can be either related or not. While in classical LET, interval duration is specified using a constant fixed duration (i.e. usually chronometric), in synchronous LET, interval duration can be specified *relatively* to clock ticks. Let's define more precisely what we mean by logical interval:

Definition 2 (Elementary Action)**.** *A logical interval (or elementary action) is defined as a computation constrained by:*

- *an activation date d_{esd} defined on a logical clock c_{start}, on which input can be read;*
- *a termination date d_{ddl} defined by a number of ticks of a logical clock c_{ddl} on which outputs are made visible.*

They both define synchronous instants called Temporal Synchronization Points (TSPs).

As an illustration, assume we have two clocks, MS and S related with an periodic relation; S ticks each 1000 MS ticks. Let's define `advance n with c` to be a synchronization of n ticks of clock c (which will be detailed in the next sections). In our paradigm, `advance 1 with MS; advance 1 with S;` is not equivalent to `advance 1 with MS; advance 1000 with MS;`. The former one describes a logical interval which starts on MS and lasts *up to* the next S tick (i.e. the exact duration can vary depending on when the interval has started) while the latter one describes a logical interval lasting exactly 1000 ticks of MS (i.e. the exact duration is fixed). This shows that a LET semantics based on multiple logical clocks (i.e. based on the Multiform Logical Time concept) provide more expressivity. While less used in the industry, the constraints can also be multiform in the sense that logical clocks can represent any event sequence. However, note that, while TSPs inherit their definition from the synchronous approach, the communication model still behave as in the classical LET paradigm, allowing a more flexible compilation.

Let's define the schedule of an sLET interval to be its corresponding computations that can happen anywhere in its interval with any physical duration. A schedule is said to be valid if the computation fits in their sLET interval. sLET (as well as classical LET) actually admits a whole class of different valid schedule that are observationally equivalent, i.e. they have the same behaviors with respect to TSPs. It is the role of the compiler to find such a valid schedule given the execution time of the computations (usually a preemptive schedule). However, from the semantics perspective, all valid schedules are strictly equivalent as they give the same observable behavior. Consequently, the synchronous semantics is actually one of those valid schedule where computation is infinitely fast. The synchronous behavior of a synchronous LET interval is then the following:

1. *Inputs*: On the interval activation, the inputs are read and saved in an internal state synchronously;
2. *Compute*: The actual computation is done synchronously with the interval activation but the outputs are not updated synchronously. Instead, it is saved in an internal state;
3. *Wait*: We wait for the interval deadline with respect to some clock expression, similarly to the ESTEREL `await` statement;
4. *Display*: The saved outputs are displayed and made visible to other tasks synchronously.

Hence, synchronous LET (as well as classical LET) have both a native behavior in which computation can happen anywhere in their interval and a synchronous behavior that only models the beginning and the ending of each interval. This motivates our work to define two different semantics of the PSYC language, which is introduced in the next section.

3 Overview of PsyC

The industrial language PsyC is a language developed by the french company Krono-Safe [1], which provides a set of tools for the design and the integration of safety-critical real-time applications. Such applications can then be certified at the highest level of criticality for the avionic domain (DAL-A with the DO-178C standard). PsyC stands for *Parallel SYnchronous* and has been initially presented as a model based on the timed-triggered approach [5]. The modern version of PsyC can use multiple logical clocks and thus, naturally implements the synchronous LET paradigm.

Logical Clocks In the spirit of the Multiform Logical Time concept, time is defined through the use of logical clocks, that is, totally-ordered sequences of ticks. The PsyC language allows to describe two levels of logical clocks:

- *sources* are externally-defined logical clocks, they can be mapped to a timer (i.e. pseudo-physical time) or any events (e.g. the rotation of an engine crankshaft);
- *clocks* define a sub-sampling of sources; they are defined through an affine relation $p \times c + o$ with p and o being respectively the period and the offset with respect to another *clock* or *source* c.

```
source realtime; // assuming realtime is 1 ms
clock c2 = 2 * realtime; // c2 ticks each 2 ms
clock c4_2 = 2 * c2 + 1; // c4_1 ticks each 4 ms starting at 2
```

Tasks A PsyC application is composed of multiple concurrent components, called *agents*. Each agent defines an infinite sequential behavior using a syntax based on the C language. A special statement called *advance* allows to synchronize on a tick of a given PsyC clock. They specify both the deadline of preceding code, and the triggering of code following it. Moreover, following the LET semantics, the *advance* specifies the instants in which communication can happen. As an example, the following code describes an infinite loop (due to the body construction) of two functions $f()$ and $g()$ in which the former interval takes 2 ticks of clock A while the latter takes 3 ticks of clock B.

```
body start {
  f();
  advance 2 with A;
  g();
  advance 3 with B;
}
```

Communication *agents* can communicate with each other through dedicated deterministic communication channels. Those communication channels only synchronize on PsyC clocks similarly to the agents. In PsyC, there are two types of communication channels:

- *Temporal Variable* which is an implicit one-to-several real-time data-flow. The task owner of the temporal variable updates this flow at a predetermined rhythm. Moreover, its value is sampled with respect to a *clock* allowing an additional sampling level between different *agents*.

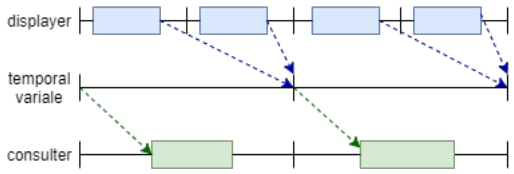


Figure 1: Visibility Principle. Here, the temporal variable has an harmonic relation with both the displayer and the consulter agent

- *Stream* which is a buffer shared between multiple producer and one consumer following the FIFO policy (*First-In First-Out*). Contrary to *Temporal Variables*, data are not always available (i.e. the buffer can be empty).

Determinism of communication is ensured by the, so-called, *Visibility Principle*. (see Figure 1). Considering that each exchanged data is timestamped:

1. a data can only be timestamped with a (logical) date greater or equal to its corresponding deadline;
2. a data can be consulted only if its timestamp is lower or equal to the current (logical) activation date;

Theoretically, more communication means can be defined, as long as they respect the visibility principle. However, in practice, *Temporal Variable* is the most commonly used. For that reason, the formal semantics defined in the next section will give the semantics of a generalized *Temporal Variable* channel that gives only the primitive to build any communication channel type (i.e. *Temporal Variable*, *Streams* or any new channel).

Also, the *Visibility Principle* can be partially relaxed to simulate synchronous communication. This mechanism, called *Null-Latency* communication is achieved using a intermediate tick defined on a *Fractional Clock*. However, as *Null-Latency* (NL) can be seen as an extension of PsyC, it is not in the scope of the present version of this report. The sLET formal semantics described in the next section is compatible with NL but the synchronous semantics needs more work.

Illustration As an illustration, the listing 2 shows a simple agent with a periodic filtering of an input which depends on a mode which states whether the filtering should be fast or slow. Depending on this, two implementation can be used, `FilterFast` and `FilterSlow`. The slow version can perform more heavy computation than the former but introduce an additional delay. This can be necessary, for example, if a system is set to a degraded mode because of a detected error, or if the system is not in a critical phase in which it needs quick reactiveness (e.g. take-off or landing of a plane). Additionnaly, this example shows an other feature of PsyC which is body handling. A body is basically an infinite loop except if one of the following statement is used : `next` which describes the next body to be executed (by default, itself), `endbody` which stop the current body execution to go to the next one and `jump` which immediately jumps to a specified body. In the example, the *start* `body` is the initialisation phase of the agent (and is executed only once at the beginning), while the *loop* `body` is the nominal phase (and is executed indefinitely).

Abstract Syntax To describe the formal semantics of PsyC in the next section, we first give an abstract syntax of the language. We choose to take a subset of PsyC excluding all advanced expressions and statements of the C language (in fact, most of the C constructions are

```
source source_ms;
clock c10ms = 10 * source_ms;
clock c100ms = 10 * c10ms;

temporal t_mode mode = SLOW with c10ms;
temporal t_in input with c10ms;
temporal t_out output with c10ms;

agent Filter(starttime 1 with c100ms) {
  display output : all;
  consult 1 $ input;
  consult 1 $ mode;
  body start {
    FilterInit();
    next loop;
    advance 1 with c100ms;
  }
  body loop {
    if ($[0]mode == SLOW) {
      output = FilterSlow($[0]input);
      advance 3 with c10ms;
      advance 1 with c100ms;
    } else {
      output = FilterFast($[0]input);
      advance 1 with c10ms;
    }
  }
}
```

Figure 2: PsyC implementation of task GNC

not interpreted by the PsyC compiler). The abstract syntax is described in Figure 3. The left column specifies the PsyC constructions defined at the application level (e.g. communication channels, clocks) while the right column describes the PsyC constructions at the agent level. Aside from the syntaxical constructs, the definition of a PsyC application that we consider should respect the following constraints:

- A temporal variable can be displayed by only one agent and can be consulted by multiple agents.
- We assume that the `starttime` value could be 0 while `advance` value could never be 0.
- The starting body of an agent is called "*start*" by convention.
- All execution paths inside a body should declare at least an `advance` statement.

application	::=	*decl+*	*agent*	::=	agent *id* (*n* with *c*) *body+*
decl	::=	*clock*	*body*	::=	body *id stmt*
	|	*source*	*stmt*	::=	$id := f(exp*)$
	|	*agent*		|	$stmt_1$; $stmt_2$
	|	*temporal*		|	if (*exp*) $stmt_2$ else $stmt_3$
source	::=	source *c*		|	while *exp* [*n*] do *stmt*
clock	::=	clock $c_1 = n_1 * c_2 + n_2$		|	while *exp* do *stmt*
				|	advance *n* with *c*
				|	endbody
				|	next *b*
				|	jump *b*
				|	nothing
			exp	::=	*id*
				|	$\$[n]id$

Figure 3: Abstract syntax of a PsyC subset

4 Native sLET semantics of one PsyC agent

We now provide the SOS big-step semantics [10] [3] for single agents; the case of agent networks will be dealt with in next section. The semantics amalgamate structurally the run of all elementary computations until (and including) the occurrence of the next `advance` statement, resulting in a single compound transition for a sLET behavior that lasts the prescribed interval.

As always with SOS rules, operational transitions can be collected in a transition system, which in the case of PsyC control-flow restrictions is actually a finite state machine (FSM). We first provide necesssary notations (something somehow tedious), then provide in turn SOS rules for the non-temporal control-flow features, then for temporal clocks advancements, and finally for the merge in a single form at the agent declaration level.

4.1 Notations

Operational semantics rules are usually given as a relation over a possible rewritting. Let's consider the relation $x \longrightarrow x'$ which basically means that a given state x can be rewritten in x' (also call the residual) after the transition. In a rule, either the relation is given alone, then it's said to be an *axiom* or it's given with respect to some assumptions. In this article, inference rules are represented as following:

$$\frac{h_1\ h_2 \ldots h_n}{x \longrightarrow x'} \tag{1}$$

This means that if all assumptions are true (i.e. $h_1, h_2 \ldots h_n$), then the transition relation can be used. Additionally, the program state has often a program environment that keeps track of the environment valuation. For a given environment E, all declared variables x can have their value accessed using the notation $E(x)$. $[\![\ exp\ ]\!]_E$ denotes a more general evaluation of *exp* with respect to the environment E and $E[x \leftarrow [\![\ exp\ ]\!]_E]$ denotes the update of the environment E with the evaluation of *exp* assigned to x.

4.2 Configuration syntax

In this paper, a configuration (i.e. a program state) of an agent is defined by the following tuple:

$$\langle E, T_{output}, b \rangle$$

where E is the private environment of the agent, b is the identifier of the next body to be executed and T_{ouput} is the public environment of the agent. This allows to distinguish values that can be access by other tasks (through the communication mechanism) and values that shouldn't be accessed.

4.3 Transition Syntax

The semantics of PsyC is described in this paper using two complementary relations, one for (logically) instantaneous behavior and one that represents time progress in term of logical instants. We shall call the former *non-temporal transitions* and the latter *temporal transitions*.

4.3.1 Non-temporal transitions

For *non-temporal* transitions, the following relation is used:

$$C \vdash t \longrightarrow C' \vdash t'$$

where C, C' denote the agent configuration respectively before and after the transition and t, t' denote the agent statements in the same way.

4.3.2 Temporal transitions

Temporal transition are due to *advance* statements making the time progress and is defined in a similar way:

$$C \vdash t \Longrightarrow_{n \times s} C' \vdash t'$$

where $n \in \mathbb{N}^*$ and s is a source denoting a temporal transition that has a duration of n ticks of the source s.

They can be composed with instantaneous statements as long as the sequence of instantaneous transitions terminates with a temporal transition:

$$\frac{C \vdash t \longrightarrow C_1 \vdash t_1 \longrightarrow \dots C_n \vdash t_n \Longrightarrow C' \vdash t'}{C \vdash t \Longrightarrow C' \vdash t'}$$

Finally, we note $\longrightarrow^*$ the sequence of zero, one or more rules $\longrightarrow$,

4.4 Sources and Clocks

Consider the following clock $c = n_1 \times c_p + n_2$ where $n_1 \in \mathbb{N}^*$ and $n_2 \in \mathbb{N}$. Its semantics is given by the following equations, which give respectively the source, the period and the offset of the clock:

$$Source(c) = \begin{cases} c, & \text{if } c_p \text{ is a source} \\ Source(c_p), & \text{if } c_p \text{ is another clock} \end{cases} \qquad \text{(clk-source)}$$

$$\Pi(c) = \begin{cases} 1, & \text{if } c_p \text{ is a source} \\ \Pi(c_p) \times n_1, & \text{if } c_p \text{ is another clock} \end{cases} \qquad \text{(clk-period)}$$

$$\Phi(c) = \begin{cases} 0, & \text{if } c_p \text{ is a source} \\ n_2 \times \Pi(C_p) + \Phi(n_2), & \text{if } c_p \text{ is another clock} \end{cases} \quad \text{(clk-offset)}$$

Additionally, to avoid keeping an unbounded date for each source, we assume that d_c gives the (current) date of c modulo its period.

Definition 3 (Clock State)**.** *A clock state d_c is defined as $d_c = (d_s - \Phi(c)) \bmod \Pi(c)$ with respect to its corresponding source date d_s.*

Note that, for each of these definitions, we assume that clock dependency cannot be cyclic and forms either a tree or a forest, depending on wether a unique source is used or not.

4.5 Semantics Rules

4.5.1 Basic statements

- nothing terminates instantaneously and can be rewritten to itself.

$$C \vdash \mathsf{nothing} \longrightarrow C \vdash \mathsf{nothing} \quad \text{(nothing)}$$

- $x := exp$ terminates instantaneously with an updated environment due to the evaluation of exp.

$$E, T, b \vdash x := exp \longrightarrow E[v \leftarrow [\![exp]\!]_E], T, b \vdash \mathsf{nothing} \quad \text{(assign)}$$

- next b statement change the target body instantaneously

$$E, T, b_1 \vdash \mathsf{next}\ b_2 \longrightarrow E, T, b_2 \vdash \mathsf{nothing} \quad \text{(next)}$$

- jump b statement change the target body instantaneously and exits the current body

$$E, T, b_1 \vdash \mathsf{jump}\ b_2 \longrightarrow E, T, b_2 \vdash \mathsf{endbody} \quad \text{(jump)}$$

- endbody statement exists the current body and starts the target body instantaneously

$$E, T, b \vdash \mathsf{endbody} \longrightarrow E, T, b \vdash \textit{abort current body and go to } B_{initial}(b) \quad \text{(endbody)}$$

with $B_{initial}$ the initial set of body of the current agent

4.5.2 If statements

- `if` statements terminate instantaneously and are rewritten with the branch selected by the condition.

$$\frac{[\![exp]\!]_E \neq 0}{E, T, b \vdash \mathsf{if}\ (exp)\ s_1\ \mathsf{else}\ s_2 \longrightarrow E, T, b \vdash s_1} \quad \text{(if-1)}$$

$$\frac{[\![exp]\!]_E = 0}{E, T, b \vdash \mathsf{if}\ (exp)\ s_1\ \mathsf{else}\ s_2 \longrightarrow E, T, b \vdash s_2} \quad \text{(if-2)}$$

4.5.3 While statements

- While statements are instantaneous and rewritten the sequence of its body followed by itself if the condition is evaluated to true

$$\frac{[\![\ exp\ ]\!]_E \neq 0}{E, T, b \vdash \mathsf{while}\ exp\ \mathsf{do}\ s \longrightarrow E, T, b \vdash s\ ;\ \mathsf{while}\ exp\ \mathsf{do}\ s} \quad \text{(while-1)}$$

- While statements are instantaneous and rewritten to nothing if the condition is evaluated to false.

$$\frac{[\![\ exp\ ]\!]_E = 0}{E, T, b \vdash \mathsf{while}\ exp\ \mathsf{do}\ s \longrightarrow E, T, b \vdash \mathsf{nothing}} \quad \text{(while-2)}$$

4.5.4 Bounded while statements

- Bounded while statements are instantaneous and behave like classical while except that they have a counter n representing the maximum iteration bound

$$\frac{[\![\ e\ ]\!]_E \neq 0 \quad n > 0}{E, T, b \vdash \mathsf{while}\ e\ [n]\ \mathsf{do}\ s \longrightarrow E, T, b \vdash s\ ;\ \mathsf{while}\ e\ [n-1]\ \mathsf{do}\ s} \quad \text{(bounded-while-1)}$$

$$\frac{[\![\ e\ ]\!]_E \neq 0}{E, T, b \vdash \mathsf{while}\ e\ [0]\ \mathsf{do}\ s \longrightarrow E, T, b \vdash \mathsf{nothing}} \quad \text{(bounded-while-2)}$$

$$\frac{[\![\ e\ ]\!]_E = 0}{E, T, b \vdash \mathsf{while}\ e\ [n]\ \mathsf{do}\ s \longrightarrow E, T, b \vdash \mathsf{nothing}} \quad \text{(bounded-while-3)}$$

4.5.5 Sequence statements

- The sequence statement is instantaneous if its first statement terminates instantaneously.

$$\frac{C \vdash s_1 \longrightarrow^* C' \vdash \mathsf{nothing}}{C \vdash s_1\ ;\ s_2 \longrightarrow C' \vdash s_2} \quad \text{(seq)}$$

We assume that this rule is extended to also propagate temporal transitions.

4.5.6 Advance statement

The advance n with c is not instantaneous. In fact, it is the only PsyC statement that takes time. Thus, it is rewritten to `nothing` and takes takes exactly the length specified by the advance statement (with respect to the source clock).

$$\frac{N = \mathit{Duration}(n, c, \mathit{date}) \quad T' = \mathrm{UpdateOutputs}(E) \quad E' = \mathrm{ResetOutputs}(E) \quad s = \mathit{Source}(c)}{E, T, b \vdash \mathsf{advance}\ n\ \mathsf{with}\ c \Longrightarrow_{N \times s} E', T', b \vdash \mathsf{nothing}} \quad \text{(advance-1)}$$

As written in the rule above, the *Duration* function need another date parameter (left undefined in this rule) to convert a deadline relative to a clock tick to an absolute deadline represented as a duration in source ticks.

Additionaly, the UpdateOutputs function extracts all the outputs from the internal environment to the external visible one and the ResetOutputs function removes all the outputs from the internal environment (alternatively, they can be defined to a special *non present* value like $\perp$). This will allow other agents or some communication mechanism to know when an output value has been produced or not.

Definition 4 (Logical Duration)**.** *Considering a date d_s assigned to each source s (and updated at each s source tick), Duration can be defined with respect to a deadline date:*

$$LastTick(c,d) = \left\lfloor \frac{d - \Phi(c)}{\Pi(c)} \right\rfloor$$

$$Deadline(n,c,d) = \Pi(c) \times (n + LastTick(c,d) + \Phi(c))$$

$$Duration(n,c,d) = Deadline(n,c,d) - d$$

However, this definition is not satisfying. First, the formula has unnecessary complexity due to the conversion between clock tick index (e.g. *LastTick*) and source dates (e.g. *Deadline*). Secondly, the use of a date require using an unbounded integer which adds a big complexity in the semantics. Indeed, most model-checking techniques require finite state models, and thus, we would need to convert back an infinite model to a finite one.

Let's define an other *Duration* function using the clock state d_c which which is finite:

Definition 5 (Finite State Logical Duration)**.**

$$Duration'(n,c,d_c) = n \times \Pi(c) - d_c$$

Theorem 1. *Let $d_c(d) = (d - \Phi(c))\ mod\ \Pi(c)$, then*
$Duration(n,c,d) = Duration'(n,c,d_c)$

Proof. By definition of *mod*:

$$d_c(d) = (d - \Phi(c))\ mod\ \Pi(c) = (d - \Phi(c)) - \Pi(c) \times \left\lfloor \frac{d - \Phi(c)}{\Pi(c)} \right\rfloor)$$

$$n \times \Pi(c) - d_c = n \times \Pi(c) - (d - \Phi(c)) + \Pi(c) \times \left\lfloor \frac{d - \Phi(c)}{\Pi(c)} \right\rfloor)$$

$$Duration_2(n,c,d_c) = \Pi(c) \left(n + \left\lfloor \frac{d - \Phi(c)}{\Pi(c)} \right\rfloor \right) + \Phi(c) - d$$

$$Duration_2(n,c,d_c) = Duration_1(n,c,d)$$

□ □

4.5.7 Body handling

For body handling, we assume that a special body, called *current* is used only to keep track of rewriting. That is, it's initially a copy of the *start* body and it is then updated at each source tick of the agent. When it's empty, it's content becomes a copy of the next body.

- A body takes time if its content takes time (i.e. typically due to an advance statement). Then, at the next step, the `current` body will be assigned to the residual statement of the latter.

$$\frac{C \vdash B(current) \Longrightarrow C' \vdash stmt' \quad B' = B[current \leftarrow stmt']}{C \vdash B \Longrightarrow C' \vdash B'} \quad \text{(body-1)}$$

- A body terminates instantaneously if it's content is composed of a sequence of instantaneous relations terminating by a `nothing` statement. Then, at the next step, the `current` body will be assigned to the last target body given by b'.

$$\frac{C \vdash B(current) \longrightarrow^* E', T', b' \vdash \mathsf{nothing} \quad B' = B[current \leftarrow B(b')]}{C \vdash B \longrightarrow E', T', b' \vdash B'} \quad \text{(body-2)}$$

Note that we have also, implicitely, a third rule for body in the case of *endbody* that aborts the current body to start the next one. However, we did not represent it for simplicity.

4.5.8 Agent handling

We then wrap everything in the rules of the PsyC agents.

- An agent first transforms its starttime expression into a one that depends only on source tick. Note that, given the specification, the target body b is initially equal to "start".

$$\frac{n > 0 \quad N = \text{Duration}(n, c, 0) - 1 \quad s = Source(c)}{C \vdash \mathsf{agent}\ id\ (n\ \mathsf{with}\ c)\ B \Longrightarrow_{N \times s} C \vdash \mathsf{agent}\ id\ (0\ \mathsf{with}\ \_)\ B} \quad \text{(agent-1)}$$

- An agent with a starttime expression equal to 0 starts the execution of its bodies.

$$\frac{C \vdash B \Longrightarrow_{N \times s} C' \vdash B'}{C \vdash \mathsf{agent}\ id\ (0\ \mathsf{with}\ \_)\ B \Longrightarrow_{N \times s} C' \vdash \mathsf{agent}\ id\ (0\ \mathsf{with}\ \_)\ B'} \quad \text{(agent-3)}$$

- An agent with a starttime expression equal to 0 with multiple instantaneous relations on its bodies can squash them, as long as they are followed by a non-instantaneous relation.

$$\frac{C \vdash B \longrightarrow^* C' \vdash B' \quad C' \vdash B' \Longrightarrow_{N \times s} C'' \vdash B''}{C \vdash \mathsf{agent}\ id\ (0\ \mathsf{with}\ \_)\ B \Longrightarrow_{N \times s} C'' \vdash \mathsf{agent}\ id\ (0\ \mathsf{with}\ \_)\ B''} \quad \text{(agent-4)}$$

A general observation is that PsyC agent are can only do temporal transitions, in other words, they always *advance* in time.

5 Native sLET semantics of a PsyC agent network

5.1 Global states vs intermediate states

The native sLET semantics of PsyC agent is defined quite classically by a transition expressing some rewriting of the agent. However, more interestingly, those transitions also express some (logical) duration with respect to some logical clock (or source clock) which is the foundation of synchronous LET; they allow to express sLET intervals (as well as classical LET intervals).

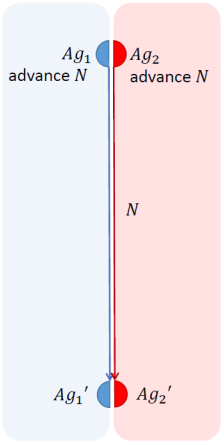
However, problems arise when building the network of agents. Indeed, in a classical synchronous language, the semantics naturally gives a synchronous composition as each instant have the same duration, that is, the cycle rate. In (synchronous) LET, multiple sLET intervals might have different durations. Consequently, in the native semantics, one cannot express directly the agent composition with only agent states.

To resolve this issue, we shall define two kinds of agent states in the context of an agent network:

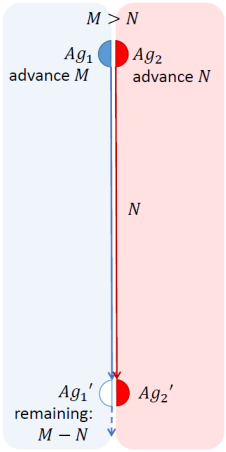
1. *Global states*, which corresponds to the actual agent states from the native semantics (i.e. the boundaries of an sLET interval)

2. *Intermediate states*, which corresponds to intermediate states in an sLET interval, typically resulting from the overlaping of the global state of another agent.

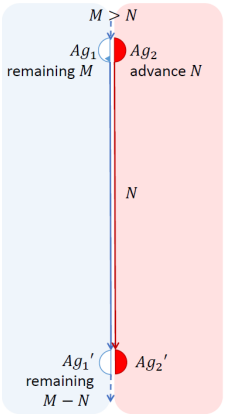
Futhermore, multiple agent transitions might have different durations based on different sources. Whereas the composition of multiple agents based on a unique source (but possibly different logical clocks based on the same source) express a total ordering, the composition of multiple agents shall also handles the case of multiple sources expressing a partial ordering coming from external non-determinism. The next section illustrates the different overlapping patterns between agents.



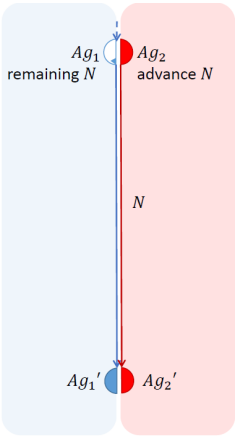
(a) Case 1: synchronized advance



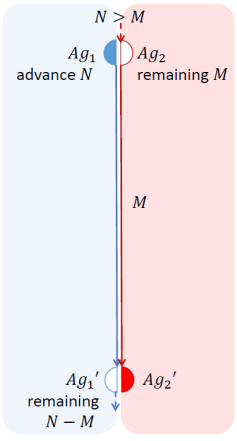
(b) Case 2: synchronized advance



(c) Case 3: synchronized advance



(d) Case 4: synchronized advance



(e) Case 5: synchronized advance

Figure 4: sLET Composition patterns

5.2 Overlapping patterns

The figure 4 illustrates all the different overlapping patterns between sLET intervals. All the cases have two agents, a blue and a red one executing both an sLET interval (but not necessarily synchronized). For each agent, a filled half circle represents a global state while an empty half circle represent an intermediate state. `advance` denotes the logical time of the whole interval while `remaining` denotes the logical time remaining in the current interval, due to overlapping.

- The case 4a shows two agents synchronized, they both start and end in a global state;
- The case 4b shows one agent (the blue one) longer than the other one, the longer interval shall then be split in sub interval, thus introducing an intermediate state and a remaining time;
- In the previous case, both agent still start at the same time. However, in the case 4d, the blue agent is already in the middle of an interval, thus starting in an intermediate state. Nonetheless, they both end synchronously as the remaining time of the blue agent and the logical time of the red one are equal.
- The case 4c extends the previous case in having a longer remaining time for the blue agent than the logical time of the red one. Thus, the blue agent starts and ends on intermediate states.
- The case 4e is another variation of the two previous cases in which the red agent has a remaining time smaller than the logical time of the blue one. Hence, the blue agent ends on an intermediate state while the red one start on an intermediate state but ends in a global state.

The reader might remark that the case where both agents end in an intermediate state is not covered. This should actually not happen for two agents as the one with the smaller logical time should ends with a global state.

The rules of the next sections formalize these patterns and extend them to a (non-empty) vector of agents. Similarly, at least on agent should end in a global state.

5.3 Notations

The agent semantics relation is defined on a set of pair configuration/agent. Each reaction is based on three steps. First, each agent advance of one step and potentially update their outputs if they terminate their interval. Then, all visible temporal variables are extracted. And finally, each configuration is updated with respect to the new visible temporal variables. Note that it's the local environment which is updated in the resulting configuration so that the temporal variables are available in the expressions. The source set $\mathcal{S}$ is deliberated left undefined as it is an input of our application, any source can tick on any instant.

In a parallel setting, an agent can be either in a global state, that is on the bound of one of its interval or in an intermediate state, which is somewhere during its interval. To convey this idea, we extend the notation of the state of an agent. We note:

- $C \vdash P@N \times s$ which denotes an intermediate state of an agent with a remaining of N ticks of source s. If only one source is used we may simplify to the following notation $C \vdash P@N$;
- $C \vdash P$ which denotes a global state, as in the previous sections.

Futhermore, as we consider multiple agent, we denote a vector of agents as follows: $[ag_1, ag_2, \ldots ag_n]$ with ag_i being an agent state as described above.

5.4 Semantics rules

This section gives the semantics rules to yields the native semantics of agent to a network of agents as explained in the two previous sections.

First of all, we have to define how to handle an intermediate state for a given agent. For that, we define an additional agent relation as following:

$$\frac{T' = Update(T, E)}{E, T \vdash ag@n \times s \Longrightarrow_{n \times s} E, T' \vdash ag} \quad \text{(agent-5)}$$

The rules should now handle two cases, *agent-network-1* which is used for a network of agent using one source and *agent-network-2* which is used for multiple sources.

$$\frac{\begin{array}{c} ag_1 \Longrightarrow_{k_1 \times s} ag'_1 \\ ag_2 \Longrightarrow_{k_2 \times s} ag'_2 \\ \dots \\ ag_n \Longrightarrow_{k_n \times s} ag'_n \\ k = min(k_1, k_2, \dots, k_n) \quad s \in S \\ T = \bigcup_{i=1}^{n} \begin{cases} Temporal(ag'_i) & \text{if } k = k_i \\ \emptyset & \text{otherwise} \end{cases} \end{array}}{[ag_1,\, ag_2 \dots ag_n] \overset{S}{\Longrightarrow}_{k \times s} [\Delta_{k/k_1}(ag_1,\, ag'_1,\, T),\, \Delta_{k/k_2}(ag_2,\, ag'_2,\, T), \dots \Delta_{k/k_n}(ag_n,\, ag'_n,\, T)]} \quad \text{(agent-network-1)}$$

$$\frac{\begin{array}{c} ag_1 \Longrightarrow_{k_1 \times s_1} ag'_1 \text{ if } s_1 \in S \text{ otherwise } k_1 = 0 \\ ag_2 \Longrightarrow_{k_2 \times s_2} ag'_2 \text{ if } s_2 \in S \text{ otherwise } k_2 = 0 \\ \dots \\ ag_n \Longrightarrow_{k_n \times s_n} ag'_n \text{ if } s_n \in S \text{ otherwise } k_n = 0 \\ T = \bigcup_{i=1}^{n} \begin{cases} Temporal(ag'_i) & \text{if } k_i = 1 \text{ and } s_i \in S \\ \emptyset & \text{otherwise} \end{cases} \end{array}}{[ag_1,\, ag_2 \dots ag_n] \overset{S}{\Longrightarrow}_{1 \times S} [\Delta_{1/k_1}(ag_1,\, ag'_1,\, T),\, \Delta_{1/k_2}(ag_2,\, ag'_2,\, T), \dots \Delta_{1/k_n}(ag_n,\, ag'_n,\, T)]} \quad \text{(agent-network-2)}$$

Finally, we define Δ_k which denotes an intermediate advance of k steps on a network of agents. This operator is defined as following:

$$\Delta_{k/0}(ag, \_, \_) = ag \quad \text{(delta-stuttering)}$$

$$\Delta_{k/k_{ag}}(\langle E,\, T\rangle \vdash P \,@\, k_{ag}, \langle E,\, T'\rangle \vdash P,\, T_{app}) = \begin{cases} \langle E \cup T_{app},\, T'\rangle \vdash P, & \text{if } k = k_{ag} \\ \langle E,\, T\rangle \vdash P \,@\, (k_{ag} - k), & \text{otherwise} \end{cases} \quad \text{(delta-intermediate)}$$

$$\Delta_{k/k_{ag}}(\langle E,\, T\rangle \vdash P,\, \langle E',\, T'\rangle \vdash P', T_{app}) = \begin{cases} E' \cup T_{app}\, T' \vdash P', & \text{if } k_{ag} = k \\ E',\, T \vdash P' \,@\, (k - k_{ag}) & \text{otherwise} \end{cases} \quad \text{(delta-global)}$$

In the equations above, all the patterns described in figure 4 are covered. First, for *multi-source* application *delta-0* handles the case of agent stuttering (the agent does nothing on this step), *delta-1* covers the case of figures 4c and 4d in which the blue agent starts in an intermediate state while *delta-2* covers the case of the figures in 4a, 4b and 4e in which the blue agent starts in a global state.

5.5 Communication channels

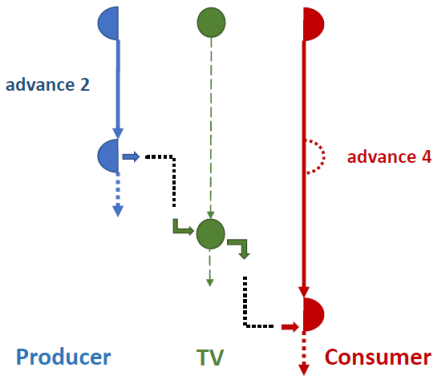


Figure 5: Two agents separated by a synchronous component TV defining a temporal variable

The section above described only communication between tasks through a generic temporal variable which is available only on producer ticks and transmitted directly to any consumer. Based on this mechanism, in all PsyC communication means, data can only be produced or consumed on visibility instants, which correspond to interval boundaries. This means that on the start instant of all agent intervals, the corresponding inputs are updated and at the end instant, the corresponding outputs are updated to be available to other agents. Consequently, in temporal variables as well as streams (detailed below), writing is effective only on the deadline of the current interval. The mechanism formalized in the last sections share similarities with ESTEREL valued signals. As such, more practical communication channels can be built using intermediate synchronous components (i.e. fifo, shared variable ...) as long as they only synchronize using PSYC clocks. The PSYC language define two of those, although more can be defined. We discuss now those two channel types and how they can be handled:

- The actual *Temporal Variable* which, in practice, is persistent and also features sampling. This can be defined with a function $tv_{consumer} = Temporal(tv_{producer}, clk)$ where $tv_{consumer}$ is a persistent $tv_{producer}$ sampled on clk. Figure 5 illustrates such behavior.
- The *Stream* which acts as a fifo between the producer and the consumer. Its definition is more complex as we have to deal both with the *push*() statements of the producer as well as the *pop*() statements of the consumer. Intuitively, all the *push*() statements of an sLET interval outputs an array from the producer and all the *pop*() statements outputs a pop counter from the consumer. Then, the communication function can be defines as $s_{consumer} = Stream(s_{producer}, nb_{consumer})$ where each time an $s_{producer}$ is sent by the producer, it is concatenated in the global fifo where $nb_{consumer}$ previously read elements deleted (in a fifo fashion).

The communication functions defined above are synchronous and thus their semantics is better defined using the synchronous approach. Basically, they can create a new temporal environment T' such that $T' = \{t' \mid \forall c \in \text{Channels}, f_c(T) = t'\}$ where Channels denotes the set of communication channels of the application and f_c its communication function.

Note that, even if communication channels only synchronize on PsyC clocks, like the agents, the logical instants on which they read or write values from agents is of prime importance. The good rhythm of data exchange is actually the key factor for the correction of the whole system. It is needed in properties such as end-to-end latencies or freshness. Verify those properties is one of the possible applications of this semantics as well as the synchronous one defined in the next section.

Although theoretically, those definition of communication channel also work in the case of agents based on different sources, we only consider mono-source communication in this document as:

1. It can be difficult to implement those communication channels (e.g. due to potential infinite buffers);
2. The PsyC compiler in its current state does not implement it.

The next section introduces the synchronous semantics and defines one communication function : the *Temporal Variable* with a sampling history which is the main communication channel of PsyC.

6 Synchronous Semantics of PsyC

While the sLET paradigm has an native real-time interpretation which is amenable to efficient compilation and scheduling analysis, the synchronous approach is more abstract and thus, more adapted for model-level analysis such as formal verification. The strong semantics of synchronous languages allows supporting foundations such as Mealy machines and digital circuits which are universally recognized formalisms used, among other, by formal verification tools. As stated in the description of the formalisms, the sLET and the synchronous model have a very close semantics as they both rely on the Multiform Logical Time concept.

6.1 Esterel translation principle

This section gives an other semantics of PsyC as a translation to a synchronous language, the Esterel language. We choosed this language as PsyC and Esterel are actually quite close syntaxically; they are both imperative and control-flow.

The translation principle is quite straightforward. All PsyC clock ticks are translated using Esterel signals, even intermediate ticks not used for agent synchronization. Those "synchronization" signals can only be emitted from the environment or from periodic clock components. Agents synchronize on these clock signals using the Esterel `await` statement, which has an equivalent semantics to the PsyC `statement`. The agent control-flow statements can be translated to the equivalent Esterel ones without difficulties.

The main difference lies in the communication handling. In PsyC, an elementary action starts on some clock tick and end on some other. Any emitted value in an elementary action has to wait for its ending to be displayed (i.e. made visible to others). Similarly, any read value in an elementary action has to be consulted at its start. In Esterel, all computations are virtually instantaneous with respect to some global instant. In our Esterel translation of PsyC, any computation is done synchronously to the start of the elementary action, thus reading input on

the same instant. However, outputs has to be kept internally (i.e. via a variable) until the end of the elementary action, on which it is then emitted.

All agents are composed in parallel as well as communication channels, like temporal variables. The latter is treated similarly to the agents as they can only synchronize on clock signals. However, contrary to the agents, they take input and outputs from agents that can be processed synchronously.

6.2 Sources and Clocks

Sources are just pure input signals whereas clocks are signals defined with respect to other signal with the following generic Clock module:

```
module Clock;
  -- input clock
  input clk_in;
  -- output clock
  output clk_out;
  -- constants
  constant Period, Offset : integer;

  await immediate clk_in;
  await [Offset mod Period] clk_in;
  loop
    emit clk_out;
  each Period clk_in;
end module
```

6.3 Temporals

Let's consider first a basic sampling module defined as follow:

```
module Sampler;
  type T;
  input In : T; output Out : T;

  loop
    emit Out(?In);
    pause;
    sustain Out(pre(?Out));
  each C
end module
```

As with clocks, a generic module can implements the mechanism of temporal variable. The depth is given as a static parameter and the types are parametric.

```
module Temporal;
  generic constant DEPTH : unsigned;
```

```
  type T;
  type T_Buffer = T[DEPTH];
  input C;
  input Temporal_In : T; output Temporal : T_Buffer;

  [
    run Sampler[Temporal_Input/In,Temporal[0]/Out,C/Clk] ||
    for i < N - 1 dopar
      run Sampler[signal pre(?Temporal[i])/In,
                         ?Temporal[i + 1]/Out, C/Clk]
    end for
  ]
end module
```

6.4 Agent declaration

The declaration of an agent is more complicated as it is no possible to implement a generic module agent as for clocks or temporals. Hence, each agent has to be translated to a dedicated *Esterel* module. Let's consider that our agent, named ID, has the body b_{start}, b_1, ...b_K and the clocks c_1, c_2, ..., c_J. The temporal variables consulted (i.e. in input) by the agent are $temporal\_in_1$, $temporal\_in_2$ ..., $temporal\_in_N$ and the displayed ones (i.e. in ouput) are $temporal\_out_1$, $temporal\_out_2$, ..., $temporal\_out_M$. Their respective types are T_IN_i for inputs and T_OUT_i for output. Futhermore, inputs also have its depth $DEPTH_i$, which is its array size, associated to its type. Finally, the starttime's value is `STARTTIME_VALUE` while its clock is `STARTTIME_CLOCK`. For simplicity, we present first the agent interface:

```
interface Agent_ID_Intf:
  -- clocks
  input c_1, c_2, ..., c_J;
  -- inputs
  input temporal_in_1 : T_IN_1[DEPTH_1];
  input temporal_in_2 : T_IN_2[DEPTH_2];
  ...
  input temporal_in_M : T_IN_M[DEPTH_M];
  -- outputs
  output temporal_out_1 : T_OUT_1;
  output temporal_out_2 : T_OUT_2;
  ...
  output temporal_out_N : T_OUT_N;
end interface
```

The main ideas of this translation are the following:

- initially, the agent waits for its starttime.
- then, after initializing its variables (i.e. its next body state, its private temporal variables and potentially its local variables) the agent enters an infinite loop in which the matching body is called, i.e. initially its `start` body;
- when a body is terminated, the loop handles the call to the next matching body;

- when a body is aborted, i.e. via a `jump` or `endbody` statement, the trap catches the body exit and the loop handles the call to the next body.

```
type Agt_ID_bodies =
  enum {BODY_B_START, BODY_B_1, ..., BODY_B_K};

module Agent_ID:
  extends Agent_ID_Intf;

  -- starttime
  await STARTTIME_VALUE STARTTIME_CLOCK;

  -- body handling
  var body_target : Agt_ID_bodies := BODY_B_START,
      private_temporal_out_1 : T_OUT_1 := ⟨init⟩,
      private_temporal_out_2 : T_OUT_2 := ⟨init⟩,
      ...
      private_temporal_out_M : T_OUT_N := ⟨init⟩ in
    loop
      trap body_exit in
        switch
          case body_target = BODY_B_START do T(b_start)
          case body_target = BODY_B_1 do T(b_1)
          ...
          case body_target = BODY_B_K do T(b_k)
        end switch
      end trap
    end loop
  end var
end module
```

Note that, all displayed temporal variables are duplicated with private copies that are variables. Body state is also handled by a variable because it may change multiple times during an instant. As the body content b_i is actually a PsyC statement, $T(s)$ actually denotes the translation of a statement described in the next section.

6.5 Agent Statements

PsyC nothing is just translated by:

```
nothing
```

PsyC assignation of temporal x, $x := exp$ is translated by:

```
private_temporal_out_x := T(exp)
```

PsyC assignation of variable x, $x := exp$ is translated by:

```
var_x := T(exp)
```

PSYC statement's advance, of the form advance n with c and assuming `var1`, `var2` ... `varN` are all the temporal variables in output of the agent, is translated by:

```
await n c;
emit temporal_var1(private_temporal_var1);
emit temporal_var2(private_temporal_var2);
...
emit temporal_varN(private_temporal_varN);
```

PSYC statement next, next b, is just translated by:

```
body_target := body_b;
```

PSYC statement endbody, is just translated by:

```
exit body_exit;
```

PSYC statement jump, jump b, is the sequence of a next and an endbody statement, thus it is translated by:

```
body_target := body_b; exit body_exit;
```

PSYC sequence statement is translated by:

```
T(stmt1); T(stmt2)
```

PSYC condition statement, if (exp) $stmt1$ else $stmt2$ is translated by:

```
if T(exp) then
  T(stmt1)
else
  T(stmt2)
end if
```

PSYC while statement, while exp do $stmt$, is translated by a combination of a trap/exit, handling the exit condition and a loop:

```
trap exit_while_id in
  loop
    if T(exp) else exit exit_while_id;
    T(stmt)
  end loop
end trap
```

PSYC bounded while statement, while exp $[n]$ do $stmt$, is similar to the while statement but with an additional bound which is decremented at each iteration and can also act as an exit condition. It is translated by:

```
var while_id_count := N in
  trap exit_while_id in
    loop
      if T(exp) and while_id_count > 0 else
        exit exit_while_id;
      while_id_count := while_id_count - 1;
```

```
      T(stmt)
    end loop
  end trap
end var
```

6.6 Agent Expressions

PsyC agent expressions can be evaluated into an equivalent form in *Esterel*. However, we will not dive into the details of expression translation as it is out of the scope of this report. Indeed, in PsyC, most complex C expressions are actually defined outside of the language, through function calls. We only give the translation of the temporal variable access.

A temporal variable access expression is expressed as $\$[n]$`temporal_in` which denotes the access `temporal_in` on the n^{th} last value sampled on its clock. It is translated by the following access on the input temporal variable array:

```
temporal_in[n]
```

6.7 Agent Application

The global PsyC application translation pattern is then simply the global parallel composition of all the modules defined above. Let's assume that we have an application with:

- the clocks being `C1, C2, ..., CJ`
- the temporal variables being `tv1, tv2, ..., tvN`, and the sampled temporal variables being `tv1_sampled, tv2_sampled, ..., tvN_sampled`
- and the agents being `ag1, ag2, ..., agM`

```
module Application:
  -- sources
  input s1, s2, ..., sn;

  signal tv1 : T1, tv2 : T2, ..., tvN : TN,
         tv1_sampled : T1[DEPTH1],
         tv2_sampled : T2[DEPTH2],
         ...,
         tvN_sampled : TN[DEPTHN],
         C1, C2, ... CJ in
  [
    run Clock[signal c1/c, ⟨c1 parameters⟩] ||
    run Clock[signal c2/c, ⟨c2 parameters⟩] ||
    ...
    run Clock[signal cJ/c, ⟨cJ parameters⟩] ||
    run Temporal[signal tv1/Temporal_In,
                        tv1_sampled/Temporal,
                 ⟨tv1parameters⟩] ||
```

```
    run Temporal[signal tv2/Temporal_In,
                        tv2_sampled/Temporal,
                 ⟨tv2parameters⟩] ||
    ...
    run Temporal[signal tvN/Temporal_In,
                        tvN_sampled/Temporal,
                 ⟨tvNparameters⟩] ||
    run Agent_ag1 ||
    run Agent_ag2 ||
    ...
    run Agent_agM
  ]
end module
```

In this translation pattern, agent parameters are given implicitely. This means that, in the agent modules, the exact parameter name should be used. Also, it should be surprising that there is no input nor output except the sources. But actually, in practice, such inputs or outputs might be dedicated tasks (potentially agents). For semantics purpose, some temporal variables could be however put in input or output of the application. Input temporal variables should be not yet sampled while output temporal variables should be already sampled (i.e. through the `Temporal` module). As an example, consider the following application module interface:

```
module Application:
  -- sources
  input s1, s2, ..., sn;
  -- temporal variables
  input tv1_input : T1_INPUT, tv2_input : T2_INPUT, ...
  output tv1_output_sampled : T1_OUTPUT[DEPTH1_OUTPUT],
         tv2_output_sampled : T2_OUTPUT[DEPTH2_OUTPUT]
         ...

  ...
end module
```

Of course, input and output temporal variables should not be redeclared as local signal in the application module.

7 Equivalence criteria between both semantics

Based on the introduced PsyC semantics and its Esterel translation, this section gives an observational equivalence between them for individual and network of agents.

To give an equivalence between the native semantics of PsyC and Esterel, we shall first have an operational semantics of both of them. The PsyC operational semantics has already been given in section 4 and various Esterel formal semantics have been given in the litterature. However our Esterel translation of PsyC heavily relies on data (e.g. private variables, body handling) and the original Esterel semantics does not consider data [3]. Hence, we will use the semantics described in [11] that extends the original one with data handling. This gives the following transition relation for Esterel:

Definition 6 (Esterel semantics)**.** *Given a constructive* Esterel *program P and a set of data, the semantics of its macro-step is given by the following relation:*

$$P, data \xrightarrow[E_i]{E_o}\!\!\!\!\!\rightarrow P', data'$$

where E_i and E_o are respectively the input and output signal set active on the current reaction

In our case, we only consider agent translation as Esterel program along with clock declaration, as illustrated in Figure 8b. Esterel data corresponds to agent private variable, temporal variable copies, next body value etc, as expressed in the agent translation. We shall then give a data equivalence between Esterel data representation and PsyC data representation, called configuration in the semantics.

Definition 7 (Data equivalence)**.** *Given a set of* Esterel *data and a* PsyC *agent configuration C, we note $C \approx data$ the equivalence relation between both representation*

The equivalence theorem yields naturally from both semantics (PsyC and Esterel) and the data equivalence relation. Considering an sLET interval, the theorem states that both the PsyC and the Esterel representation have an equivalent behavior on the boundaries of the interval. The PsyC semantics yields only one transition while the Esterel semantics yields a sequence of unitary transitions (i.e. with respect to the source). If both data representation are equivalent at the start of the interval, then, they are also equivalent at the end.

Theorem 2. *Forall* PsyC *agent p_{ag} and the* Esterel *translation T and for any agent transition of the form:*

$$C \vdash p_{ag} \Longrightarrow_{n \times s} C' \vdash p'_{ag}$$

Assuming, $C \approx data$ and $s \in E$, we have an equivalent sequence of Esterel *transitions:*

$$T(p_{ag}), data \xrightarrow[E]{}\!\!\!\!\!\rightarrow P^1, data^1 \ldots \xrightarrow[E]{}\!\!\!\!\!\rightarrow P^n, data^n$$

such that $T(p'_{ag}) = P^n$ and $C' \approx data^n$.

Proof. By structural induction on native rules structure □

Note that theorem 2 does not consider stuttering, that is, transition without source tick. However, this result extends easily to allow stuttering, Esterel transition does not change the program when no source tick is present.

Nonetheless, this equivalence result only adress a single agent while an application is actually a system of multiple agents, as described in both semantics. More precisely, the native semantics divides an agent interval into sub-intervals, introducing partial states, to allow the parallel product with other agents. This product operator thus models the beginning and the ending of all agents intervals in a synchronous fashion. Additionnaly, communication only synchronize on global logical clocks (i.e. sources or periodic). Consequently, because agents evolve independently in terms of synchronization, this equivalence result compose nicely and is thus preserved to the case of multiple agents. Full formalization is left for future work.

8 Case-Study

As an illustration, we shall consider a very simplified version of the control system of a spaceship introduced in [4], called a Guidance, Navigation and Control system (GNC). This use-case is

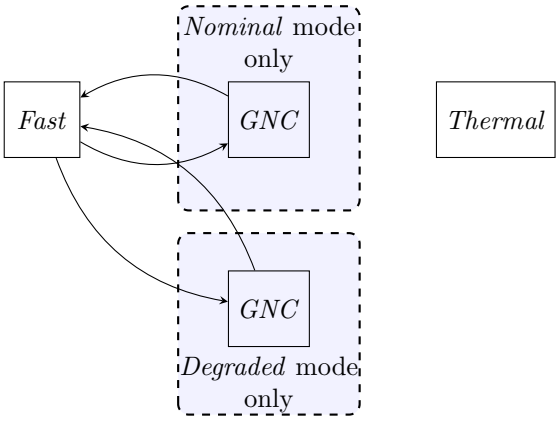


Figure 6: The GNC functional architecture

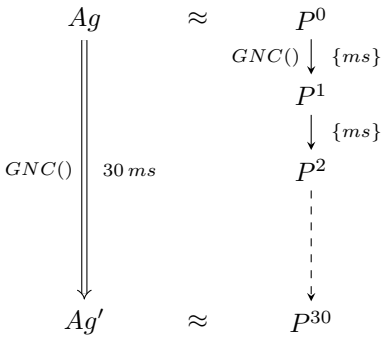
composed of three tasks: *Fast* which interacts with the sensors and actuators, *GNC* which is responsible of the control and guidance functions and *Thermal* which is responsible of the thermal regulation. *GNC* has two different modes, one nominal mode and one degraded mode that has different periods constraints. The functional architecture is described in Figure 6.

Based on the functional and temporal requirements, the *GNC* task could be implemented in PsyC as detailed in Figure 8a and based on the Esterel translation, it can be translated to Esterel as shown in Figure 8b.

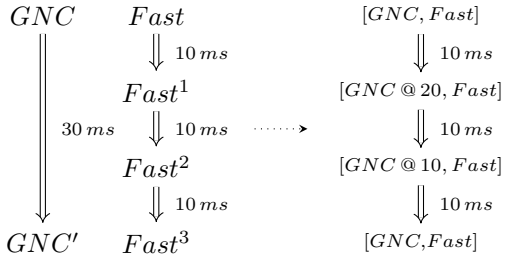
In nominal mode, the native semantics yields the following infinite trace for agent *GNC*:

$$C \vdash ag \Longrightarrow_{100} C^1 \vdash ag^1 \overset{GNC()}{\Longrightarrow}_{30} C^2 \vdash ag^2 \Longrightarrow_{70} C^3 \vdash ag^3 \overset{GNC()}{\Longrightarrow}_{30} C^4 \vdash ag^4 \dots$$

The source `source_ms` is omitted for clarity reasons and the function *GNC()* has be annotated on its corresponding transition. If we take this transition, the Esterel translation yields the



(a) GNC semantics simulation diagram



(b) GNC agent network semantics

Figure 7: GNC semantics diagrams

```
clock c10ms = 10 * source_ms;
clock c100ms = 10 * c10ms;
...

agent GNC(starttime 1 with c100ms) {
  display updateCommands : all;
  consult 1 $ filteredSensors;
  consult 1 $ mode;
  body start {
    if ($[0]mode == NOMINAL) {
      /* compute GNC */
      updateCommands =
        GNC($[0]filteredSensors);
      advance 3 with c10ms;
    }
    advance 1 with c100ms;
  }
}
```

(a) PsyC implementation of task GNC

```
run Clock[signal source_ms, c10ms,
          constant 10, 0] ||
run Clock[signal c10ms, c100ms,
          constant 10, 0] ||
...
module Agent_GNC:
  ...
  await 1 c100ms;
  var private_updateCommands : t_cmd in
    loop
      if mode[0] = NOMINAL then
        -- compute GNC
        run GNC[signal
          filteredSensors[0],
          private_updateCommands];
        await 3 c10ms;
        emit update_commands(
          private_updateCommands);
      end if;
      await 1 c100ms;
    end loop
  end var
end module
```

(b) Esterel translation of task GNC

Figure 8: Task GNC

following unit transitions:

$$P, data \xrightarrow[E^1]{GNC()} P^1, data^1 \xrightarrow[E^2]{} P^2, data^2 \ldots \xrightarrow[E^{30}]{} P^{30}, data^{30}$$

where *data* contains *filteredSensors* used for the computation in the first transition which outputs the updated *updateCommands* value only in $data^{30}$, thus having an equivalent behavior as the native semantics described above. The simulation diagram in Figure 7a illustrates this equivalence at the agent level while Figure 7b illustrates how the parallel operator from section 5.4 allows to build a unique automata from a network of agent automata.

9 Implementation Issues

Without going into details into the compilation process, it is important however to raise some issues due to the expressivity of PsyC. Depending on the application domain, PsyC can be used with multiple sources or with only one temporal source (each logical clock can reduce to one clock). In this section, we shall consider the latter sub-case of PsyC which is predominant in our industrial work. (Of course, those problems also arise with multiple-sources, but the solution are more complex.)

The "mono-source" compilation process of PsyC starts by reducing all the clocks to one source clock. This is called the *unfolding process*. For each task, one get one unfolded automata using only the source clock, as given by the native semantics. The second compilation process is

```
/* start: 0 */
advance 1 with c2;
f();
advance 1 with c3;
```

(a) PsyC code

```
/* start: 0 */
advance 2; // c2
f();
advance 1; // c3
advance 1; // c2
f();
advance 2; // c3
```

(b) Unfolded PsyC code

```
/* start: 0, constraint = 2 */
advance 2; // constraint = 1
f();
advance 1; // constraint = 1
advance 1; // constraint = 2
f();
advance 2; // constraint = 2
```

(c) Unfolded constrained PsyC code

Figure 9: Implementation

to *pull up* the temporal constraints on the start of a logical interval. This allows the execution platform to set up the correct timers *before* starting any computation. The Figure 9 sketches the different steps of the implementation process, constraints annotation then become *system calls* for the real-time platform which ensure that any computation following the constraint terminates (i.e. the next `advance` is reached) *before* the end of the allowed time.

However, the compilation is more complex when multiple `advance` statement with different values are reachable from the start of an interval. In that case, the compilation process could not predict which constraint is the correct one as illustrated in 10a. This is a consequence of the compilation model which could not evaluate conditional expression, thus having internal non-determinism when branching. (However, note that the operational semantics of PsyC is still deterministic.) The solution that have been implemented in the Krono-Safe compiler for years now, is a mechanism to release the constraint, after its initialization, during a computation in a logical interval. The initial constraint is thus the minimal constraint of all the reachable `advance` and when the execution chose a branch with a higher constraint that the one previously set, additional time is given to the constraint according to the new reachable `advance`. This mechanism is illustrated in 10b. However, in this example, one should note that $h()$ should systematically fit in a logical interval of duration 2, even if the chosen branch has a constraint of 3.

```
advance 1; // constraint = 2 or 3?
h();
if (/* some condition */) {
  f();
  advance 2;
} else {
  g();
  advance 3;
}
```

(a) PsyC code with conditional constraint

```
advance 1; // constraint = min(2, 3)
h();
if (/* some condition */) {
  f();
  advance 2;
} else {
  // constraint += 1
  g();
  advance 3;
}
```

(b) PsyC code with translated conditional constraint

Figure 10: Implementation with conditions

In the case of *multiple sources*, the problem is however harder to solve as one may not be

able to compute the *min* of multiple clocks related with partial order. This one of the reasons that *multi-source* handling is limited in the industrial compiler, in particular to one source by agent/task.

10 Related Work

Synchronous-Reactive languages Several models have been proposed to specify the timing behavior of real-time systems. The most classical approach is based on synchronous languages and the synchronous hypothesis. This includes the use of languages such as ESTEREL or ESTEREL. However, the compilation of synchronous languages may suffer from the so-called "Long Task Problem" when tasks of different rhythms are compiled; execution time variability is usually limited to a common cycle rate. More recently, synchronous languages, such as ESTEREL relax the synchronous hypothesis, and thus do not suffer from the same compilation problems, but are limited to the the use of strictly periodic clocks.

Logical Execution Time The closest model to our work is the LOGICAL EXECUTION TIME paradigm introduced in the GIOTTO language, later improved with the TDL language. These languages can express a set of periodic tasks with a succession of periodic constant LOGICAL EXECUTION TIME intervals as well as a limited mode semantics. Comparing to synchronous languages, they trade expressivity with a simpler model, allowing a simpler and less pessimistic compilation. In particular, they could not express logical clocks nor more complex behavior such as aperiodic behaviors. Our work try to reach a compromise between classical LOGICAL EXECUTION TIME and the synchronous approach. SYNCHRONOUS LOGICAL EXECUTION TIME, and in particular the PSYC language, can describe a set of task that have an automata expressivity as well as some logical clocks used to set logical deadlines for each computation. In the sub-case where each logical clocks can be reduced to one *pseudo-physical* logical clock (mapped to real-time), classical LOGICAL EXECUTION TIME compilation techniques could be re-used.

External Non-Determinism In the more general case in which logical clocks could be mapped to any event sequence, this raises concern, however, about external non-determinism. These problems have been heavily studied in the litterature in work such as CSP with the so-called "input guards", conflict freeness in Petri Nets, monotonocity in Kahn Process Networks, latency-insensitivity and endochrony of synchronous languages.

Logical Execution Time with events AS SYNCHRONOUS LOGICAL EXECUTION TIME (and PSYC) can also be used to describes events, its expressivity is also very close to the XGIOTTO language, extending LOGICAL EXECUTION TIME with events. Nonethess, besides compilation issues, it seems that the so-called "multi-source" version of PSYC does not have great industrial interest yet. Most Krono-Safe industrial partners are satisfied with the "mono-source" version of PSYC.

11 Conclusion and Future Work

Through a generalization of Logical Execution Time, called synchronous Logical Execution Time, this report gave two formal semantics of PSYC, a native operational semantics and a synchronous semantics defined by translation to a synchronous language. It also introduced an observational equivalence criteria between both semantics. As PSYC faithfully represents the synchronous

Logical Execution Time paradigm and more generally, the class of Logical Execution Time languages (e.g. GIOTTO, TDL, XGIOTTO), this equivalence result also applies to these languages. Verify timing properties is one of the possible applications of the native semantics as well as the synchronous one. Hence, future work will focus on developing a formal verification framework for the PSYC language based on this approach.

References

[1] Krono-Safe (2023), `https://www.krono-safe.com/`

[2] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P., Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE **91**, 64 – 83 (02 2003)

[3] Berry, G.: The Constructive Semantics of Pure Esterel (06 1996)

[4] Carle, T., Potop-Butucaru, D., Sorel, Y., Lesens, D.: From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation *. Leibniz Transactions on Embedded Systems (Nov 2015). https://doi.org/10.4230/LITES-v002-i002-a001

[5] Chabrol, D., Vidal-Naquet, G., David, V., Aussagues, C., Louise, S.: Oasis: A chain of development for safety-critical embedded real-time systems (01 2004)

[6] Kirsch, C., Sokolova, A.: The logical execution time paradigm. pp. 103–120 (10 2012)

[7] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (Jul 1978)

[8] Lee, E.A., Sangiovanni-Vincentelli, A.: The tagged signal model-a preliminary version of a denotational framework for comparing models of computation. Department of Eelectrical Engineering and Computer Science, University of California p. 7 (1996)

[9] Lundqvist, T., Stenstrom, P.: Timing anomalies in dynamically scheduled microprocessors. In: Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054). pp. 12–21 (1999). https://doi.org/10.1109/REAL.1999.818824

[10] Plotkin, G.: A structural approach to operational semantics. J. Log. Algebraic Methods Program. (2004)

[11] Potop-Butucaru, D., Edwards, S.A., Berry, G.: Constructive operational semantics. Compiling Esterel pp. 79–102 (2007)

[12] Pree, W., Templ, J.: Modeling with the timing definition language (tdl). In: Automotive Software Workshop. pp. 133–144. Springer (2006)

[13] Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., Becker, B.: A definition and classification of timing anomalies. vol. 4 (01 2006)

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399