TESL: a Model with Metric Time for Modeling and Simulation

Hai Nguyen Van
Université Paris-Saclay, CNRS, LRI, 91405, Orsay, France
perso.crans.org/nguyen-van

Frédéric Boulanger
Université Paris-Saclay, CNRS, LRI, CentraleSupélec, 91405, Orsay, France
frederic.boulanger@lri.fr

Burkhart Wolff
Université Paris-Saclay, CNRS, LRI, 91405, Orsay, France
burkhart.wolff@lri.fr

Abstract

Real-time and distributed systems are increasingly finding their way into critical embedded systems. On one side, computations need to be achieved within specific time constraints. On the other side, computations may be spread among various units which are not necessarily sharing a global clock. Our study is focused on a specification language – named TESL – used for coordinating concurrent models with timed constraints. We explore various questions related to time when modeling systems, and aim at showing that TESL can be introduced as a reasonable balance of expressiveness and decidability to tackle issues in complex systems. This paper introduces (1) an overview of the TESL language and its main properties (polychrony, stutter-invariance, coinduction for simulation), (2) extensions to the language and their applications.

2012 ACM Subject Classification Theory of computation → Timed and hybrid models

Keywords and phrases Timed Systems, Semantics, Models, Simulation

Digital Object Identifier 10.4230/LIPIcs.TIME.2020.12

Supplementary Material Artifacts and source code available at `github.com/heron-solver/heron`.

1 Introduction

Designing and modeling systems nowadays still raise open problems. A very expressive language or framework can be useful to model a complex system where events are not trivially interleaved. On the opposite, an excessively expressive language is the reason for prohibitive slow-downs or even undecidability. As such, a reasonable balance between expressiveness and decidability needs to be found. In the current industrial trend for critical embedded systems, grows an increasing need for two kinds of systems:

- *Real-Time Systems* where an external input is followed by an output delivered within a specified time, named *deadline*. The correct behavior of such systems must be ensured at both logical and temporal levels.
- *Distributed Systems* where autonomous nodes communicate and cooperate to perform a common computation.

A distributed real-time system (DRTS) [34, 14] belongs to both categories and consists in autonomous computing nodes where specific timing constraints must be met. DRTS are essential as they describe more closely common real-time applications by providing fault tolerance and load sharing [35, 34, 14]. An example of a DTRS is a modern car using CAN buses [14]. In such a setting, a middle gateway connects two CAN buses. One of them is high-speed and connects the engine, the suspension and the gearbox control. The other one

is low-speed and connects the lights, seat and door control units. The aviation industry also exhibits an increasing need for DTRS as shown by recent developments in interoperable gateways ED-247 [21].

On the side of formal modeling, various environments have emerged to tackle the issue of modeling and verifying complex systems. Some are industrial products, such as Matlab/Simulink [15], Wolfram SystemModeler [33], SCADE [7]. Some others are academic experiments, such as Ptolemy II [13], TimeSquare [12], ModHel'X [20]. Our study is centered around the inner formalisms that drive these environments, and in particular the TESL language. The main question this paper addresses is: *Can we provide a uniform framework to model distributed and real-time systems?*. The paper is organized as follows: Section 2 introduces the TESL language which we believe can answer the main problem. Section 3 introduces its main properties, in terms of polychronous clocks, stutter-invariance and coinductive unfolding. Finally, in Section 4 we present some extensions and aim at showing their relevance in the scope we address.

2 The TESL language

The Tagged Events Specification Language (TESL) [8] originates from the idea of coordinating the execution of heterogeneous inner-parts of a model as components of the ModHel'X modeling and simulation environment. The language is inspired by CCSL [16, 26], the Tagged Signal Model [25] and from the constructive semantics of Esterel [6, 5] for the original simulation solver. In this setting, an event is modeled by a *clock*, with an associated time scale. Considering a continuous system, its behavior is discretized into a sequence of observation instants. At each instant, a clock admits a *timestamp* (also called *tag*), that stands for the metric time measured on this clock. Besides, a clock also admits a *tick* which indicates an occurrence of the event at this instant. The domain for timestamps can possibly be any totally ordered set. We emphasize the fact that the language handles chronometric time constraints, which are different from logical time constraints. Chronometric time constraints are given on durations measured between timestamps. Two forms of constraints may be specified in TESL:

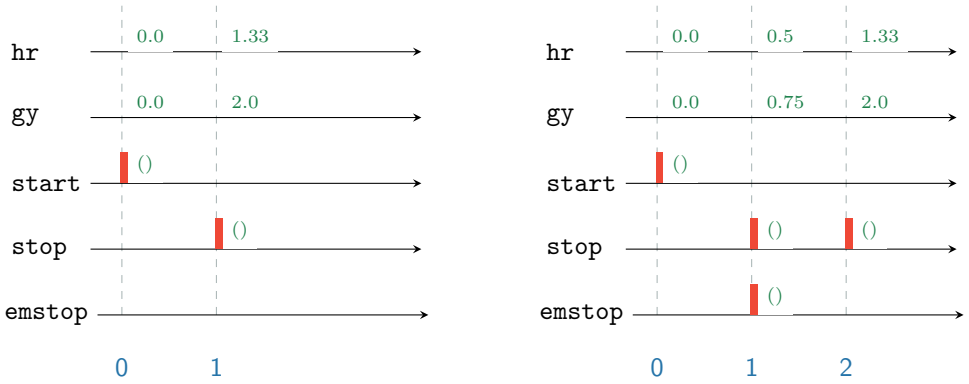
- *Event-triggered causality.* Events may occur due to the occurrence of other events. For instance "I have a coffee because my office mate prepares some coffee".
- *Time-triggered causality.* Events may occur because a time threshold has been reached. For instance "I have a coffee because it is 9am".

2.1 Illustrating the Language

Let us model in TESL the simple behavior of a radiotherapy machine used in cancer treatment. The patient has a prescription of 2 Gy of radiation in low-dose-rate of 1.5 Gy.h^{-1}.

Listing 1 Radiotherapy machine

```
1 rational-clock hr                  // Time unit in hours
2 rational-clock gy                  // Radiation unit in Gray
3 unit-clock     start sporadic ()   // Start emitting rays
4 unit-clock     stop                // Stop emitting rays
5 unit-clock     emstop              // Emergency stop
6 time relation gy = 1.5 * hr
7 start time delayed by 2.0 on gy implies stop
8 emstop implies stop
```



(a) Normal situation **(b)** Emergency stop

Figure 1 Two partially satisfying runs

Lines 1 to 5 declare clocks `hr` and `gy` with rational timestamps, and clocks `start`, `stop` and `emstop` with the unit timestamp (so there is no chronometric scale associated to them). The constraint `sporadic` enforces the occurrence of a tick on `start`. Line 6 specifies that time on `hr` flows 1.5 times as fast as on `gy`. Line 7 specifies that each time clock `start` ticks, clock `stop` will tick after a delay of `2.0` measured on the time scale of clock `gy`. Line 8 requires that each time the `emstop` clock ticks, the `stop` clock instantaneously ticks as well. The syntax of such expressions is detailed in Subsection 2.3.

Two behaviors are illustrated in Figure 1. They show possible execution traces or *runs* satisfying the TESL specification. A run consists in a sequence of synchronization *instants* (vertical dashed line with blue numbers). Each of them contains *ticks* (in red) along with *timestamps* (in green) on the time-scales of the clocks `hr`, `gy`, `start`, `stop` and `emstop`.

2.2 Clocks, runs and timestamps

▶ **Definition 1.** *Let $\mathbb{K}$ be the set of clocks, $\mathbb{B}$ the set of booleans and $\mathbb{T}$ the ordered domain of timestamps. The set of runs is denoted $\Sigma^{\propto}$ and defined by*

$$\Sigma^{\propto} \quad = \quad \mathbb{N} \to \mathbb{K} \to (\mathbb{B} \times \mathbb{T})$$

Additionally, we define two projections that extract the components of an event occurrence:

$\text{ticks}(\rho\ n\ K)$ *ticking predicate of clock K in run ρ at instant n (first projection)*

$\text{time}(\rho\ n\ K)$ *time value on clock K in run ρ at instant n (second projection)*

▶ **Example 2.** Let $\rho_{\text{Fig.1a}}$ be the run shown in Figure 1a, we have $\text{ticks}(\rho_{\text{Fig.1a}}\ 0\ \texttt{start}) = \text{true}$ and $\text{time}(\rho_{\text{Fig.1a}}\ 1\ \texttt{gy}) = 2.0$.

2.3 Quick overview of the syntax

We briefly introduce some expressions of the language which serve the purpose of this paper. The reader may refer to the official website of TESL[1] for an exhaustive description of all the features of the language. A TESL specification Φ is described by the following grammar:

[1] `https://wdi.centralesupelec.fr/software/TESL/`

$$
\begin{array}{lcl}
\Phi & ::= & \langle atom \rangle \wedge \ldots \wedge \langle atom \rangle \\
\langle atom \rangle & ::= & \langle clock \rangle \texttt{ sporadic } \langle timestamp \rangle \texttt{ on } \langle clock \rangle \\
& | & \langle clock \rangle \texttt{ implies } \langle clock \rangle \\
& | & \texttt{time relation } (\langle clock \rangle, \langle clock \rangle) \in \langle relation \rangle \\
& | & \langle clock \rangle \texttt{ time delayed by } \langle duration \rangle \texttt{ on } \langle clock \rangle \texttt{ implies } \langle clock \rangle
\end{array}
$$

where $\langle clock \rangle \in \mathbb{K}$, $\langle timestamp \rangle \in \mathbb{T}$, $\langle duration \rangle \in \mathbb{T}$ and $\langle relation \rangle \subseteq \mathbb{T} \times \mathbb{T}$.

To provide a quick understanding, we briefly and informally explain the semantics:

- K `sporadic` τ `on` K_{meas} requires a tick on clock K at an instant where the timestamp on K_{meas} is τ ;
- K_{master} `implies` K_{slave} models instantaneous causality by specifying that at each instant where K_{master} ticks, K_{slave} ticks as well ;
- `time relation` $(K_1, K_2) \in R$ relates the time frames of clocks K_1 and K_2 by specifying that at each instant, the timestamps on K_1 and K_2 have to be in relation R ;
- K_{master} `time delayed by` $\delta\tau$ `on` K_{meas} `implies` K_{slave} stands for delayed causality by duration. At each instant k where K_{master} ticks, it requires a tick on K_{slave} at an instant where the timestamp on K_{meas} is τ', with τ' the sum of $\delta\tau$ and the timestamp on K_{meas} at instant k. In other words, it states that each tick on K_{master} must be followed by a tick on K_{slave} after a delay $\delta\tau$ measured on the time scale of K_{meas}.

3 Properties of the language

3.1 Polychronous clocks and time islands

One of the most prominent properties of the TESL language lies in *polychronous clocks* [23], a *global clock* does not necessarily drive the system. In the context of distributed systems, there exists as many clocks as there are computing nodes: all run at different rates and their clocks may possibly drift along. This is why, an additional mechanism of *synchronization* is necessary to coordinate these subworkers to achieve a common desired computation.

- *Metric level.* There are similarities with time dilation as in *special relativity* [19] where time seems to flow more slowly for a stationary observer than for a moving observer. The drift increases with the speed of the moving observer. For instance, GPS satellites suffer from time drifting and it is necessary to take into account these effects.
- *Temporal level.* Modern computing also exhibits this idea where temporal cycles may speed up or slow down. Current predominant processors adjust their clock speed with respect to environmental variables (energy, heat, noise), this is called *throttling*. Today's multicore processors consist of multiple computing units which may run faster or slower for these reasons, while possibly being used to achieve a distributed computation.

We illustrate this statement with the running example by adding an independent computing unit used for auxiliary computation needs. Whenever its computation is finished, it will trigger an event to indicate that it is ready. Let us simply declare a clock `aux` whenever this computing unit yields its signal. Besides, we can also create a scenario where we require this to occur at timestamp 0.5. The following line can be added to the specification in Listing 1:

```
rational-clock aux sporadic 0.5
```

In this setting, clocks `hr` and `gy` are said to belong to the same *time island* as their timeframes are arithmetically related. On the other hand, clock `aux` belongs to another independent time island. There may also be other clocks living around as the specification is permissive and allows other clocks to exist even though they were not specified.

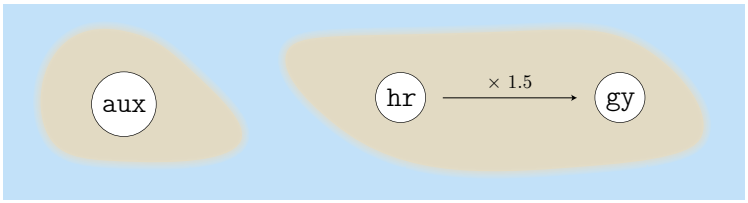


Figure 2 Graphic representation of time islands

Let us consider the specification in Listing 1, with the additional `aux` clock as declared above. Figure 3 depicts three runs which satisfy this specification. For presentation purposes, only three clocks `hr`, `gy` and `aux` are displayed. On the leftmost figure, we observe that `aux` ticks at 0.5 when it is 0.0 on `hr`. On the center figure, `aux` ticks at 0.5 when it is between 0.0 and 0.5 on `hr`. On the rightmost figure, `aux` ticks at 0.5 when it is 0.5 on `hr`. We see therefore that there exists an infinite number of satisfying runs as the timeframe on clock `aux` is left completely unrelated to the other time frames. However, we developed a simulation solver for TESL that supports symbolic runs, and hence captures this infinity of runs in a finite number of symbolic runs using symbolic timestamps.

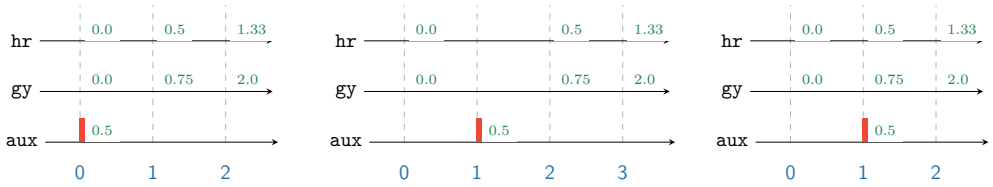


Figure 3 Examples of satisfying runs with additional clock `aux` in an independent time island

3.2 Stutter Invariance

A fundamental concept of concurrent and distributed systems is *stutter invariance*. In finite-state model checking, it is an essential requirement for partial-order reduction techniques. When composing automata, the addition of stutter, or silent instants, allows the accommodation for their different alphabets. From a point of view in language theory, the membership of any word in a language shall be preserved even if a letter is duplicated. In our setting to model and compose submodels, we need stutter invariance in order to provide *compositionality*. For instance, when composing two specifications, we may have to add observation instants to a run that satisfies a specification in order to observe events on clocks that belong to the other specification. In other words, stuttering in necessary to refine specifications [22]. Stutter invariance also allows one to observe a model more often than necessary without changing its behavior.

In TESL, composing specifications is simply performed by the conjunction of TESL-formulae. To illustrate the idea of stutter-invariance with the running example, let us assume that we require the system to trigger some refresh mechanism every 10 minutes. We would add the following lines to the specification:

```
refresh sporadic 0.0 on hr
refresh time delayed by <10/60> on hr implies refresh
```

If we consider the run from Figure 1b and wish to compose it with this refreshing mechanism, a satisfying run is shown in Figure 4. The top of the figure shows the original run as in Figure 1b, whereas the bottom depicts a run where new instants have been added. A one-to-one correspondence is observed between run instants in the top and the bottom figure. Both runs exhibit the same first instant where `start` is triggered, with `refresh` additionally ticking in the second run. However, the second instant of the second run exists due to the refreshing requirement at 0.166 on clock `hr`, which is not present on top.

Stutter-invariance is illustrated by the fact that a run may be dilated and new instants added while still satisfying the specification.

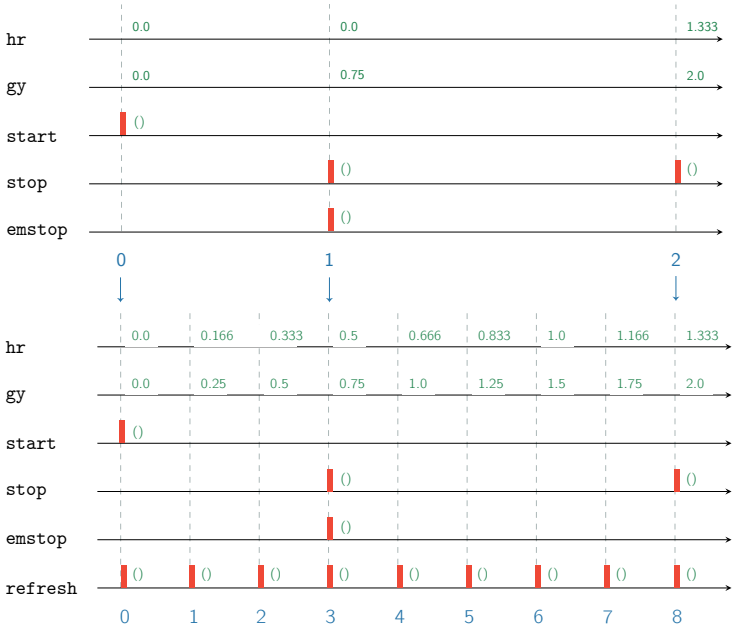


Figure 4 The example of radiotherapy run dilated

3.3 Unfolding Specifications

The language allows the specification of runs that can be constructed and described by operational rules. In [29], we introduced an operational semantics of the language whose main ideas are summarized in Figure 5. The general concept of the operational semantics revolves around a 3-component pattern *past-present-future*. The past component contains the run we are constructing (which we also call the *run context*), the present component contains TESL-formulae to consume for the construction of the current instant, while the future component contains TESL-formulae to consume for future instants. The system considers each TESL formula as a consumable resource, and its consumption produces a "smaller" resource, which allows to constructively build the past component. Finally, the past component is a symbolic run and contains logical primitives which are sent to a SMT-solver in order to decide the satisfiability of the constructed run. Put differently, we reduced the problem of solving a TESL specification to a simpler constraint solving problem.

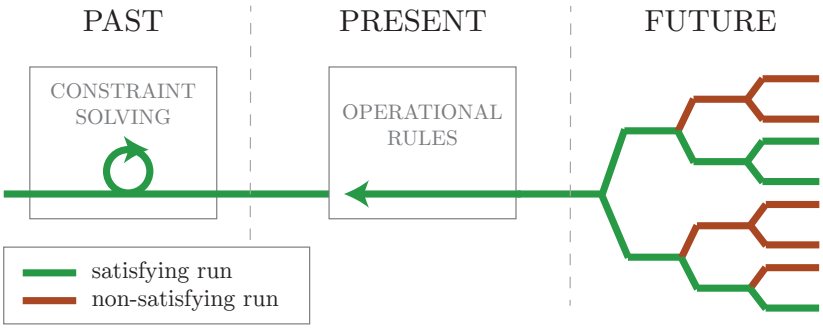


Figure 5 Usage of the operational semantics

4 Extensions

In this section, we propose two extensions of the language. From the original implementation of TESL, we have experimentally broadened its scope by adding two features on formulae and clocks. The addition of such has increased the language expressiveness without compromising constraint solving. To provide an insight, we illustrate them with an application example. We designed and experimented their semantics by implementing them into an experimental solver, named Heron[2] [29]. This implementation is a path-exhaustive multicore simulation solver built with MLton/MPL [36, 37]. It directly implements the operational semantics and the presented extensions. It can also be used for system testing and monitoring.

4.1 Precedence formula (and timed automata)

The first extension we propose is built around the precedence operator as found in CCSL. A appreciable motivation lies in modeling Synchronous Dataflows [24, 26]. In this model, each component provides an interface with inputs and outputs, and respectively a number of input tokens (to be read) and another of output tokens (to be written). When wiring two components, it is necessary that the n-th output writing event will precede the n-th input reading event. Precedence allows to specify this kind of indexed requirement over the order of event occurrence.

We extend the syntax of TESL as shown in Subsection 2.3 with

$$\begin{array}{lcl} \langle atom \rangle & ::= & \ldots \\ & | & \langle clock \rangle \texttt{ weakly precedes } \langle clock \rangle \\ & | & \langle clock \rangle \texttt{ strictly precedes } \langle clock \rangle \end{array}$$

Informally, K_1 `weakly precedes` K_2 means that each tick on clock K_2 may be uniquely mapped to a tick on K_1 in the past or current instants (as a one-to-one correspondence). K_1 `strictly precedes` K_2 is analogous but maps to instants that are strictly in the past.

▶ Remark 3. Mallet *et al.* showed that the decidability of this type of formula could be handled with counter automata [27]. In our framework, we modeled this formula in a similar way by embedding run contexts with arithmetic constraints containing counter expressions. Again, we reduced this problem to a constraint solving problem.

[2] `https://github.com/heron-solver/heron`

To illustrate our interest in this operator, we consider timed automata [2, 1] as introduced by Alur and Dill. An additional and distinct mechanism made of clocks (also referred as *chronometers*) is used to store and specify metric timing constraints. On the implementation side, they extend classical finite-state automata with timing constraints. This formalism allows time to progress inside states while transitions are instantaneous, meaning that transitioning from one state to another is fast enough to be abstracted. In this subsection, we describe how this model of computation can be encoded with TESL extended with precedence. Let us give in Figure 6 a simple timed automaton (extracted from [4]) which models a system in which an alarm is triggered whenever the delay between receiving two messages is less than 5 seconds.

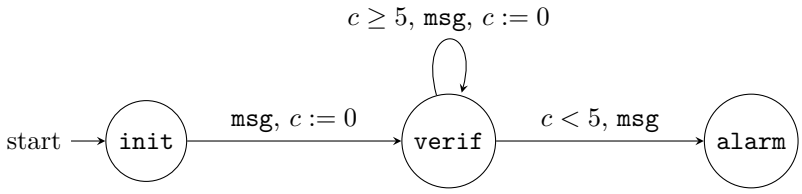


Figure 6 An example of timed automata from [4]

To model the timed automaton in Figure 6, we declare TESL-clocks that will simulate the events occurring at a lower level (suffixed by `_enter` and `_leave`). Other clocks are also declared for transitions.

```
// Set of states: {init, verif, alarm}
unit-clock state_init_enter
unit-clock state_init_leave
unit-clock state_verif_enter
unit-clock state_verif_leave
unit-clock state_alarm_enter
unit-clock state_alarm_leave
```

We also need to declare TESL-clocks related to the behavior of TA-clocks, in particular when resetting them.

```
// Set of clocks: {c}
unit-clock c_reset
rational-clock c sporadic 0.0
```

Likewise, we need a TESL-clock to model the reading of a symbol (so-called *action*).

```
// Set of actions: {msg}
unit-clock read_msg
```

We proceed by encoding in TESL each transition of the timed automaton. We model the first transition from `init` to `verif`, which must read symbol `msg` and reset clock c, as:

```
// Transition t1 = init -> verif: msg, c:= 0
state_init_leave when read_msg implies trigger_t1
trigger_t1 implies state_verif_enter
trigger_t1 implies c_reset
```

The second transition from `verif` to itself can be triggered when reading `msg` if time on clock c is greater than or equal to 5, which will eventually lead to resetting c. This means that the transition can be triggered if more than 5.0 units of time have elapsed on c since

the last time c has been reset. When using this transition, one will remain in state `verif` while resetting c to 0 each time a message has been read.

```
// Transition t2 = verif -> verif: c>=5, msg, c:= 0
c_reset time delayed by 5.0 on c with reset on trigger_t3
        implies trigger_t2_min
trigger_t2_min weakly precedes trigger_t2
state_verif_leave ∧ read_msg implies trigger_t2 ∨ trigger_t3
trigger_t2 implies state_verif_enter
trigger_t2 implies c_reset
```

The third transition from `verif` to `alarm` is triggered when a new message has been received before 5.0 units of time have elapsed. We model this as:

```
// Transition t3 = verif -> alarm: c<5, msg
c_reset time delayed by 5.0 on c with reset on trigger_t2
        implies trigger_t3_max
trigger_t3 strictly precedes trigger_t3_max
state_verif_leave ∧ read_msg implies trigger_t2 ∨ trigger_t3
trigger_t3 implies state_alarm_enter
```

Figure 7 shows a run prefix exhibiting the behavior of our encoding of the timed automaton. At instant 0, time on clock `c` is 0.0 and we enter in state `init`. At instant 1, 5.0 units of time have elapsed. At instant 2, 5.0 additional units of time have elapsed and `read_msg` has been triggered, thus the transition is triggered (`trigger_t1`). The TA-clock `c` is reset and leaves state `init` to enter `verif`. Also, a minimum limit has been set on triggering transition t_2 as it can only be fired after elapsing at least 5.0 units of time (as depicted by `trigger_t2_min` at instant 4). At instant 4, symbol `msg` is read and transition t_2 is triggered to re-enter in the same state `verif`. Finally, at instant 5, the symbol `msg` is read again and transition t_3 is triggered to enter `alarm`. A tick on `trigger_t3` is possible as it precedes `trigger_t3_max`. Likewise, `trigger_t3_max` defines a maximum limit to ensure any t_3-transition triggering only before.

4.2 Previous operator (and PID controllers)

Another useful operator is `pre` with similar syntax and semantics as in Lustre [18]. This operator simply allows to refer to the previous timestamp on a clock. Hence, a substantial part of feedback systems can be modeled accurately as they require registers to store previous values. The power of computation is significantly augmented and allows us to model more complex systems, such as mathematical sequences and series (*e.g.*, Fibonacci), differential calculus (derivatives, Euler's integrator), or digital filters.

Since this operator refers to the value of a signal at a previous instant, we generalized TESL clocks as *flows*. A flow is a clock where timestamps are no longer required to be monotonic. As a matter of fact, these "timestamps" are simply called *values*.

We extend the syntax of TESL as shown in Subsection 2.3, with:

$$\begin{array}{lcl} \langle clock \rangle & ::= & K \in \mathbb{K} \\ & | & \texttt{pre}\ \langle clock \rangle \end{array}$$

This extension is useful at modeling feedback systems. Let us illustrate this with the ubiquitous algorithm of automatic control theory: the Proportional-integral-derivative (PID) controller [39]. In this theory, a PID controller delivers a control signal to a process in order

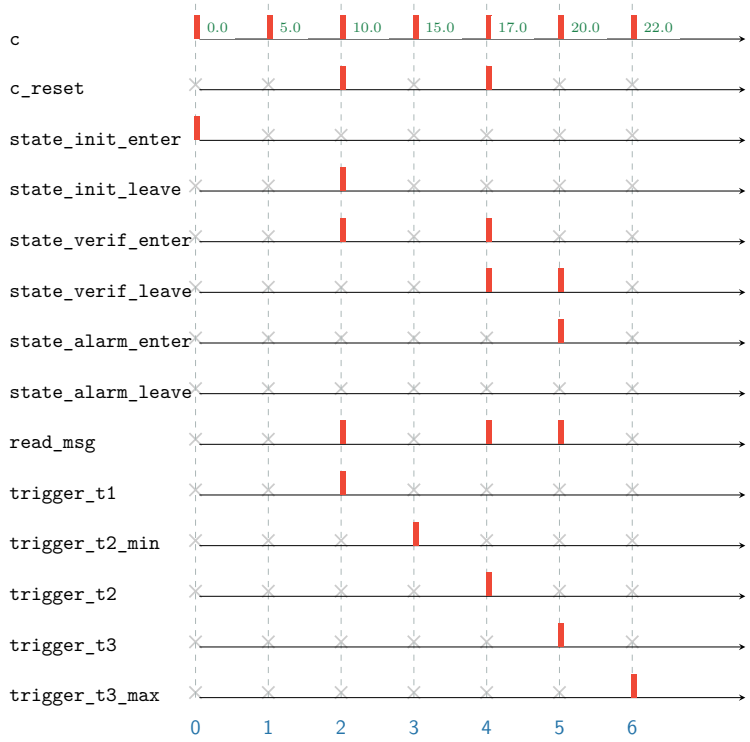


Figure 7 A satisfying run prefix to encode a timed automaton

to bring a process output closer to a reference setpoint (*e.g.*, cruise control in cars, autopilots in airplanes).

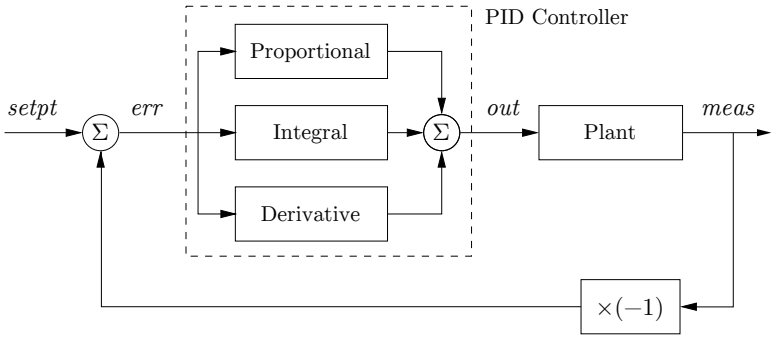


Figure 8 General diagram of a process using a PID controller

The block diagram in Figure 8 shows the structure of the controller. Basically, the system receives as input the error signal `err`, *i.e.* the difference between the reference setpoint `setpt` and the process output `out`, and computes a control signal based on the sum of a term proportional to the error, an integral term and a derivative term. Each of the three terms is parameterized by a multiplying factor, respectively `Kp`, `Ki` and `Kd`, which are commonly called *gains*. Thereafter, the controller output enters a *transfer function* which translates the control signal `out` into the process output `meas`. For instance in automotive control theory, this occurs when converting the position of the gas pedal into the generated car velocity. This new output will be used to feed the error back at the next computing cycle. It

is possible to describe this system straightforwardly in TESL as in Listing 2.

Listing 2 The PID controller

```
// Time
time relation dt = 1.0
time relation t = [0.0] -> (pre t) + dt
// Gain
time relation Kp = 0.1
time relation Ki = 0.2
time relation Kd = 0.2
// Setpoint
time relation setpt = 40.0
// Control signal
time relation err      = setpt - meas
time relation integr   = [0.0] -> (pre integr) + (err * dt)
time relation derivat  = [0.0] -> (err - (pre err)) / dt
time relation out      = (Kp * err) + (Ki * integr) + (Kd * derivat)
// Simple actuation
time relation meas     = [0.0] -> (pre meas) + (pre out)
```

When running this example, the solver yields the output shown by the extract in Listing 3.

Listing 3 An extract of the satisfying run found by Heron of the PID controller

```
### Solver has successfully returned 1 model
## Simulation result [0x1ADAB]:
             meas         err          integr       derivat      out
[1]          0.0          40.0         0.0          0.0          4.0
[2]          4.0          36.0         36.0         -4.0         10.0
[3]          14.0         26.0         62.0         -10.0        13.0
[4]          27.0         13.0         75.0         -13.0        13.0
[5]          40.0         -1.0         74.0         -14.0        12.0
...
```

Additionally, the values of the flows `meas`, `err` and `out` are plotted in Figure 9. As expected, we observe that the process output `meas` is brought closer to the reference setpoint `setpt` = 40.0. Besides, the error signal and the control signal `out` gradually decrease to 0.0 as the need to damp out oscillations progressively decreases.

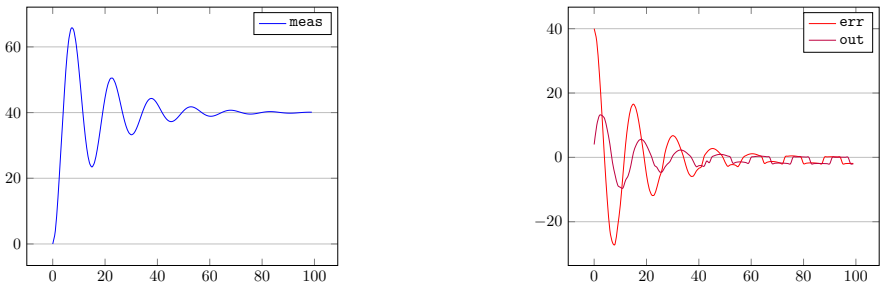


Figure 9 Plotting values for `meas`, `err` and `out`

5 Related Work

In the family of synchronous programming languages [3], Lustre [18], Esterel [6, 5] and Signal [17] are known to provide polymorphic time (time domains of various type). However,

their time model is purely logical, which is not suited when dealing with modeling non-discretizable systems. Prelude [32] and Zélus [9] overcome this with continuous dynamics.

All these previous models derive clocks from a global root clock, which constrains models to flow from a single reaction clock. Polychrony (clocks possibly living in various independent timeframes) overcomes this restriction by allowing specifications with more relaxed and concurrent execution of systems. This feature can be observed in the Signal language or polychronous automata [23]. Compared to TESL, they do not allow metric time constraints.

TESL is also inspired by CCSL which supports asynchronous constraints on events. It admits an executable [38] and denotational semantics [11, 28]. However, time in CCSL is purely logical and durations are counted as a number of ticks on a clock.

On a more theoretical-side, timed automata [2, 1] support both discrete events and metric time. However, clocks are global and uniform, they necessarily progress at the same rate.

All in all, TESL attempts to overcome these limitations and provides a general-purpose specification language of synchronous and asynchronous constraints with clocks over polymorphic time while supporting polychrony, and mixing logical and metric time.

6 Future work

The outcome of our study leads us to various future opportunities:

- An effort is currently running towards a machine-checkable formalization of the operational and denotational semantics into the Isabelle/HOL proof assistant [31, 30]. We successfully proved that the operational semantics was correct and complete with respect to the denotational semantics. Proving both extensions of the paper is a future direction.
- Numerous questions about model-checking remain unanswered. In our experiments, we have observed that unfolded specifications could be refolded with abstract interpretation techniques. This would offer a finite-representation of these infinite-state systems, thereby providing means to decide safety and liveness properties of such systems.
- In addition, the TESL language seems to be suited for modeling and simulation of systems with time of various kind. With the new extensions we propose and their implementation in an existing efficient solver, we believe TESL can become a relevant asset as a simulation engine for simulation platforms, such as the GEMOC Studio [10].

7 Conclusion

This study introduces a language – named TESL – suited for the modeling and simulation of complex systems with multi-level time considerations. For this purpose, we illustrated how the language is suited for various applications of time in models. We first illustrated the main properties of the language (absence of a global root clock, stutter invariance). Then, we introduced two extensions of the language along with two applications depicted by (1) an encoding of timed automata, and (2) an implementation of a PID controller.

Most of the widely used formalisms suffer from restrictions in their model of time, which we attempt to address. Some consider time as purely logical and may not be suited for real-time systems as computing cycles may not necessarily flow at a fixed rate. Some other consider time as global which is restrictive towards distributed systems where time does not flow at the same rate in the different components, and may not be synchronized. We believe our approach is complementary to state-of-the-art environments and may help to circumvent their drawbacks by considering time in its whole nature.

References

1. Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification*, pages 8–22, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994. URL: http://www.sciencedirect.com/science/article/pii/0304397594900108, doi:https://doi.org/10.1016/0304-3975(94)90010-8.
3. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
4. Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification*. Springer Berlin Heidelberg, 2001. URL: https://doi.org/10.1007%2F978-3-662-04558-9, doi:10.1007/978-3-662-04558-9.
5. G. Berry. The constructive semantics of pure Esterel, 1996.
6. Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.
7. Gérard Berry. SCADE: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.
8. Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Iuliana Prodan. TESL: a language for reconciling heterogeneous execution traces. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2014)*, pages 114–123, Lausanne, Switzerland, Oct 2014. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6961849, doi:10.1109/MEMCOD.2014.6961849.
9. Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with odes. In *Proceedings of the 16th International Conference on Hybrid Systems: Computation and Control*, HSCC '13, page 113–118, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2461328.2461348.
10. Benoit Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jezequel, and Bernhard Rumpe. *Globalizing Domain-Specific Languages*, volume 9400 of *LNCS, Programming and Software Engineering*. Springer International Publishing, 2015.
11. Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, Inria, November 2014. URL: https://hal.inria.fr/hal-01082274.
12. Julien DeAntoni and Frédéric Mallet. Timesquare: Treat your models with logical time. In Carlo A. Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, volume 7304 of *Lecture Notes in Computer Science*, pages 34–41. Springer, 2012. doi:10.1007/978-3-642-30561-0_4.
13. J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.
14. K. Erciyes. *Distributed Real-Time Systems*. Springer International Publishing, 2019. URL: https://doi.org/10.1007%2F978-3-030-22570-4, doi:10.1007/978-3-030-22570-4.
15. Sulaymon Eshkabilov. *MATLAB®/Simulink® Essentials: MATLAB®/Simulink® for Engineering Problem Solving and Numerical Analysis*. Lulu Publishing Services, 11 2016.
16. Kelly Garcés, Julien Deantoni, and Frédéric Mallet. A Model-Based Approach for Reconciliation of Polychronous Execution Traces. In *SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, Oulu, Finland, August 2011. IEEE. URL: https://hal.inria.fr/inria-00597981.
17. P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Synchronous data flow programming with the language SIGNAL. *IFAC Proceedings Volumes*, 20(2):359 – 364, 1987. 2nd IFAC

Workshop on Adaptive Systems in Control and Signal Processing 1986, Lund, Sweden, 30 June-2 July 1986.

18 N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

19 Michael J W Hall. Concepts in special relativity. In *General Relativity: An Introduction to Black Holes, Gravitational Waves, and Cosmology*, 2053-2571, pages 1–1 to 1–11. Morgan & Claypool Publishers, 2018. URL: http://dx.doi.org/10.1088/978-1-6817-4885-6ch1, doi:10.1088/978-1-6817-4885-6ch1.

20 Cécile Hardebolle and Frédéric Boulanger. Exploring multi-paradigm modeling techniques. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 85(11/12):688–708, November/December 2009. URL: /software/downloads/ModHelX/2009MPMSimulation.pdf, doi:http://dx.doi.org/10.1177/0037549709105240.

21 Yannick Hildenbrand. Ed-247 (vistas) gateway for hybrid test systems. In *Aerospace Systems and Technology Conference*. SAE International, oct 2018. doi:10.4271/2018-01-1949.

22 Leslie Lamport. What good is temporal logic? *Information Processing 83, R. E. A. Mason, ed., Elsevier Publishers*, 83:657–668, May 1983. URL: https://www.microsoft.com/en-us/research/publication/good-temporal-logic/.

23 Paul Le Guernic, Thierry Gautier, Jean-Pierre Talpin, and Loïc Besnard. Polychronous automata. In *TASE 2015, 9th International Symposium on Theoretical Aspects of Software Engineering*, pages 95–102, Nanjing, China, September 2015. IEEE Computer Society.

24 E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

25 Edward A. Lee and Alberto Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. CAD*, 17(12), 1998.

26 Frédéric Mallet, Julien Deantoni, Charles André, and Robert De Simone. The Clock Constraint Specification Language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1-2):99–106, March 2010.

27 Frédéric Mallet and Robert de Simone. Correctness issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106:78 – 92, 2015. Special Issue: Architecture-Driven Semantic Analysis of Embedded Systems. URL: http://www.sciencedirect.com/science/article/pii/S0167642315000519, doi:https://doi.org/10.1016/j.scico.2015.03.001.

28 Mathieu Montin and Marc Pantel. Mechanizing the denotational semantics of the clock constraint specification language. In El Hassan Abdelwahed, Ladjel Bellatreche, Mattéo Golfarelli, Dominique Méry, and Carlos Ordonez, editors, *Model and Data Engineering*, pages 385–400, Cham, 2018. Springer International Publishing.

29 Hai Nguyen Van, Thibaut Balabonski, Frédéric Boulanger, Chantal Keller, Benoît Valiron, and Burkhart Wolff. A symbolic operational semantics for TESL. In Alessandro Abate and Gilles Geeraerts, editors, *Formal Modeling and Analysis of Timed Systems*, pages 318–334, Cham, 2017. Springer International Publishing.

30 Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

31 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

32 Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011. URL: https://hal.inria.fr/inria-00638936.

33 Kirill Rozhdestvensky, Vladimir Ryzhov, Tatiana Fedorova, Kirill Safronov, Nikita Tryaskin, Shaharin Anwar Sulaiman, Mark Ovinis, and Suhaimi Hassan. *Description of the Wolfram SystemModeler*, pages 23–87. Springer Singapore, Singapore, 2020. doi:10.1007/978-981-15-2803-3_2.

34 Werner Schutz. *The Testability of Distributed Real-Time Systems*. Kluwer Academic Publishers, USA, 1993.

35 J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

36 Stephen Weeks. Whole-program compilation in mlton. In *Proceedings of the 2006 Workshop on ML*, ML '06, page 1, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1159876.1159877.

37 Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371115.

38 M. Zhang and F. Mallet. An executable semantics of Clock Constraint Specification Language and its applications. In *Formal Techniques for Safety-Critical Systems: 4th International Workshop, FTSCS 2015*, pages 37–51, Cham, 2016. Springer.

39 Karl Johan Åström and Richard M. Murray. *Feedback Systems*. Princeton University Press, Princeton, 2010. URL: https://www.degruyter.com/view/title/563028.