**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.2020.15

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
```
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 $K$ be the set of clocks, $B$ the set of booleans and $T$ the ordered domain of timestamps. The set of runs is denoted $\Sigma^\infty$ and defined by

$$
\Sigma^\infty = N \rightarrow K \rightarrow (B \times T)
$$

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

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

### 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 $\Phi$ is described by the following grammar:

\(^1\) https://wdi.centralesupelec.fr/software/TESL/
Φ ::= ⟨atom⟩ ∧ ... ∧ ⟨atom⟩
⟨atom⟩ ::= ⟨clock⟩ sporadic ⟨timestamp⟩ on ⟨clock⟩
| ⟨clock⟩ implies ⟨clock⟩
| time relation ((clock), (clock)) ∈ ⟨relation⟩
| ⟨clock⟩ time delayed by ⟨duration⟩ on ⟨clock⟩ implies ⟨clock⟩

where ⟨clock⟩ ∈ K, ⟨timestamp⟩ ∈ T, ⟨duration⟩ ∈ T and ⟨relation⟩ ⊆ T × T.

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

- K sporadic τ on Kmeas requires a tick on clock K at an instant where the timestamp on Kmeas is τ;
- Kmaster implies Kslave models instantaneous causality by specifying that at each instant where Kmaster ticks, Kslave ticks as well;
- time relation (K1, K2) ∈ R relates the time frames of clocks K1 and K2 by specifying that at each instant, the timestamps on K1 and K2 have to be in relation R;
- Kmaster time delayed by δτ on Kmeas implies Kslave stands for delayed causality by duration. At each instant k where Kmaster ticks, it requires a tick on Kslave at an instant where the timestamp on Kmeas is τ′, with τ′ the sum of δτ and the timestamp on Kmeas at instant k. In other words, it states that each tick on Kmaster must be followed by a tick on Kslave after a delay δτ measured on the time scale of Kmeas.

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 \textit{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.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{time_islands.png}
\caption{Graphic representation of time islands.}
\end{figure}

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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{satisfying_runs.png}
\caption{Examples of satisfying runs with additional clock \( aux \) in an independent time island.}
\end{figure}

### 3.2 Stutter Invariance

A fundamental concept of concurrent and distributed systems is \textit{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 \textit{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 \cite{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](image-url) 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.
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\textsuperscript{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

\[
\langle \text{atom} \rangle ::= \ldots \\
| \langle \text{clock} \rangle \text{ weakly precedes } \langle \text{clock} \rangle \\
| \langle \text{clock} \rangle \text{ strictly precedes } \langle \text{clock} \rangle
\]

Informally, $K_1$ \textit{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). $K_1$ \textit{strictly precedes} $K_2$ is analogous but maps to instants that are strictly in the past.

\begin{itemize}
  \item \textbf{Remark 3.} Mallet \textit{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.
\end{itemize}

\textsuperscript{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.

\[ c \geq 5 \text{, } \text{msg, } c := 0 \]

\[ c < 5 \text{, } \text{msg} \]

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 \( \text{verif} \) while resetting \( c \) to 0 each time a message has been read.

```plaintext
// Transition \( t_2 = \text{verif} \rightarrow \text{verif} : c \geq 5, \text{msg}, c := 0 \)
\begin{align*}
\text{c_reset time delayed by 5.0 on } c \text{ with reset on } & \text{trigger}_t3 \\
\text{implies } & \text{trigger}_t2\_\text{min} \\
\text{trigger}_t2\_\text{min weakly precedes } & \text{trigger}_t2 \\
\text{state}_\text{verif}_\text{leave} \land \text{read}_\text{msg} \implies & \text{trigger}_t2 \lor \text{trigger}_t3 \\
\text{trigger}_t2 \implies & \text{state}_\text{verif}_\text{enter} \\
\text{trigger}_t2 \implies & \text{c_reset}
\end{align*}
```

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

```plaintext
// Transition \( t_3 = \text{verif} \rightarrow \text{alarm} : c < 5, \text{msg} \)
\begin{align*}
\text{c_reset time delayed by 5.0 on } c \text{ with reset on } & \text{trigger}_t2 \\
\text{implies } & \text{trigger}_t3\_\text{max} \\
\text{trigger}_t3 \text{ strictly precedes } & \text{trigger}_t3\_\text{max} \\
\text{state}_\text{verif}_\text{leave} \land \text{read}_\text{msg} \implies & \text{trigger}_t2 \lor \text{trigger}_t3 \\
\text{trigger}_t3 \implies & \text{state}_\text{alarm}_\text{enter}
\end{align*}
```

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 \( \text{init} \). At instant 1, 5.0 units of time have elapsed. At instant 2, 5.0 additional units of time have elapsed and \text{read}_\text{msg} has been triggered, thus the transition is triggered (\( \text{trigger}_t1 \)). The TA-clock \( c \) is reset and leaves state \( \text{init} \) to enter \( \text{verif} \). 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 \( \text{trigger}_t2\_\text{min} \) at instant 4). At instant 4, symbol \text{msg} is read and transition \( t_2 \) is triggered to re-enter in the same state \( \text{verif} \). Finally, at instant 5, the symbol \text{msg} is read again and transition \( t_3 \) is triggered to enter \( \text{alarm} \). A tick on \( \text{trigger}_t3 \) is possible as it precedes \( \text{trigger}_t3\_\text{max} \). Likewise, \( \text{trigger}_t3\_\text{max} \) defines a maximum limit to ensure any \( t_3 \)-transition triggering only before.

### 4.2 Previous operator (and PID controllers)

Another useful operator is \text{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 (\textit{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 \textit{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 \textit{values}.

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

\[
\begin{align*}
\text{(clock)} & ::= K \in \mathbb{K} \\
& \mid \text{pre (clock)}
\end{align*}
\]

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 to bring a process output closer to a reference setpoint (\textit{e.g.}, cruise control in cars, autopilots in airplanes).
<table>
<thead>
<tr>
<th>c</th>
<th>0.0</th>
<th>5.0</th>
<th>10.0</th>
<th>15.0</th>
<th>17.0</th>
<th>20.0</th>
<th>22.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_reset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_init_enter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_init_leave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_verif_enter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_verif_leave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_alarm_enter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>state_alarm_leave</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_msg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_t1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_t2_min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_t3_max</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7** A satisfying run prefix to encode a timed automaton.

**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 $K_p$, $K_i$ and $K_d$, 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]:

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>mea</td>
<td>err</td>
<td>integ</td>
<td>deriv</td>
<td>out</td>
</tr>
<tr>
<td>[1]</td>
<td>0.0</td>
<td>40.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>[2]</td>
<td>4.0</td>
<td>36.0</td>
<td>36.0</td>
<td>-4.0</td>
</tr>
<tr>
<td>[3]</td>
<td>14.0</td>
<td>26.0</td>
<td>62.0</td>
<td>-10.0</td>
</tr>
<tr>
<td>[4]</td>
<td>27.0</td>
<td>13.0</td>
<td>75.0</td>
<td>-13.0</td>
</tr>
<tr>
<td>[5]</td>
<td>40.0</td>
<td>-1.0</td>
<td>74.0</td>
<td>-14.0</td>
</tr>
</tbody>
</table>

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


