A Symbolic Operational Semantics for TESL with an Application to Heterogeneous System Testing

Hai Nguyen Van, Thibaut Balabonski, Frédéric Boulanger, Chantal Keller, Benoît Valiron, Burkhart Wolff

A Symbolic Operational Semantics for TESL
with an Application to Heterogeneous System Testing

Hai Nguyen Van[1], Thibaut Balabonski[1], Frédéric Boulanger[2], Chantal Keller[1], Benoît Valiron[2], and Burkhart Wolff[1]

[1] LRI, Université Paris Sud, CNRS, Université Paris-Saclay, France
[2] LRI, CentraleSupélec, Université Paris-Sud, Université Paris-Saclay, France
Firstname.Lastname@lri.fr

Abstract. TESL addresses the specification of the temporal aspects of an architectural composition language that allows the composition of timed subsystems. TESL specifies the synchronization points between events and time scales. Methodologically, subsystems having potentially different models of execution are abstracted to their interfaces expressed in terms of timed events.
In this paper, we present an operational semantics of TESL for constructing symbolic traces that can be used in an online-test scenario: the symbolic trace containing a set of constraints over time-stamps and occurrences of events is matched against concrete runs of the system.
We present the operational rules for building symbolic traces and illustrate them with examples. Finally, we show a prototype implementation that generates symbolic traces, and its use for testing.

Keywords: Heterogeneity, Synchronicity, Timed Behaviors

1 Introduction

The design of complex systems involves different formalisms for modeling their different parts or aspects. The global model of a system may therefore consist of a coordination of sub-models that use differential equations, state machines, synchronous data-flow networks, discrete event models and so on. This raises the interest in *architectural composition languages* that allow for "bolting the respective sub-models together", along their various interfaces, and specifying the various ways of collaboration and coordination. Figure 1 shows a conceptual diagram of such a heterogeneous system model.

The Ptolemy project [10] was one of the first to provide support for mixing heterogeneous models. More recently, the GEMOC initiative [9] has been putting the focus on the development of techniques, frameworks and environments to facilitate the creation, integration, and automated processing of heterogeneous modeling languages. While Ptolemy follows a generic approach to architectural composition, the BCOoL language [20] is more specifically targeted at coordination patterns for Domain Specific Events, which define the interface of a domain specific modeling language.

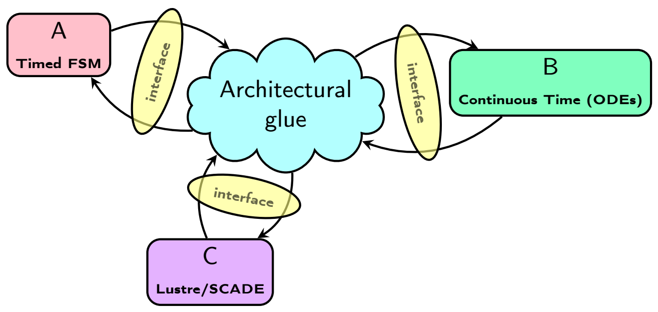


Fig. 1. A Heterogeneous Timed System

Our interest in architectural composition has a particular emphasis on subsystems involving *time* and *timed behavior*. In contrast to BCOoL, which translates its coordination patterns into CCSL (Clock Constraints Specification Language, see [11,15]), we target TESL (Tagged Events Specification Language, see [6]), a language that we designed to allow the specification of durations as differences between tags, and not only as a number of occurrences of an event. This model of time is close to the time in the MARTE [1] profile, or in the Tagged Signal Model [14]. This allows us to coordinate systems with different forms of time that flow at different rates.

The TESL language, which was developed in the ModHel'X [12] heterogeneous simulation platform, was originally targeted only at the timed coordination of sub-models during the simulation of heterogeneous models. It allows sub-systems to live in different "time islands" by supporting the notion of time scale and of relations between the speed at which time elapses for different clocks. TESL is totally synchronous and focuses on *causality* between events and *synchronization* on time scales. Causality is expressed in statements such as: "event X should occur now because event Y occurs now". Synchronization is expressed in statements such as: "event X should occur because time reaches t on the time scale of a clock". This can be used to coordinate the execution of models that have different notions of time (physical time, angular position, distance) that flow at different rates, which are in the most general case only loosely coupled and can even accelerate.

In this paper, we extend our simulation framework to a verification framework: we present a novel *test method* establishing that the time coordination of some sub-models, as it is actually implemented in a given system, conforms to the specification modeled in TESL. Since an enumerative model-checking approach is impossible for real-time systems and infeasible for practical discrete time systems, we develop a novel operational semantics geared to the symbolic execution of TESL specifications. If the latter is run in parallel to a system under test (SUT), symbolic traces containing variables for instants of time, constraints over time scale relations, and causal conditions, can be instantiated following the reactions of the SUT, refining the current constraints to produce a new specification conforming to input stimulus. The approach has been implemented in a novel prototype tool, for which we will present early experimental results.

2 An Introduction to TESL

The Tagged-Event Specification Language (TESL) [6] is a declarative language designed for the specification of the timed behavior of discrete events and their synchronization. Event occurrences (aka *ticks*) are grouped in *clocks*, which give them a time-stamp (aka a *tag*) on their own *time scale*. Tags represent the occurrence of the event at a specific time. The tag domains used for time must be totally ordered; typically, they are reals, rationals, integers, as well as the singleton `Unit`, which is used for purely logical clocks where time does not progress.

TESL allows for specifying causality and time scales between clocks, basically by three main classes of constraints.

Event-triggered implications. The occurrence of an event on one clock might trigger another one: “Whenever clock `a` ticks, clock `b` will tick under conditions”. For instance, to model the fact that the minutes hand of a watch moves every minute, we will say that the `min` clock implies the `move` clock.

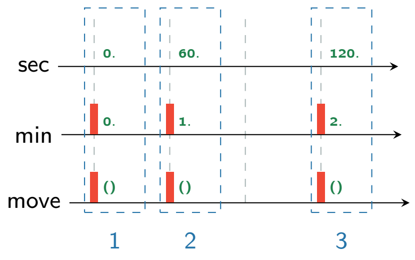
Time-triggered implications. This kind of causality enforces the progression of time. The occurrence of an event triggers another one after a chronometric delay measured on the time scale of a clock. For instance, in order to specify that the `min` clock ticks every minute, we can require that clock `min` *implies* itself with a time delay of 1.0 measured on its time scale. It is important to note that this delay is a duration (a difference between two tags) and not a number of ticks.

Tag relations. When all clocks are combined in a specification, each of them lives in its own “time island”, with a potentially independent time scale. The purpose of tag relations is to link these different time scales. For instance, time runs 60 times as fast on clock `sec` as on clock `min`. This does not mean that the faster clock has more ticks, it only means that in any given instant, the tags of these clocks are in a ratio of 60. In general, TESL allows for fairly general tag relations (permitting even acceleration or slow-down); for the sake of simplicity, we will present only *affine tag relations* throughout this paper; this reduces the complexity of constraint-solving to handling linear equation systems.

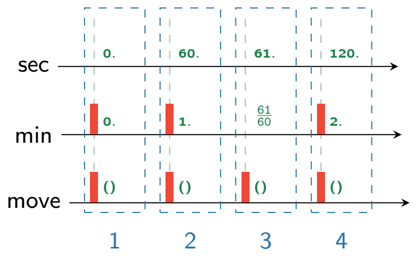
Here is a TESL specification for the examples above:

```
1  rational-clock sec
2  rational-clock min sporadic 0.0
3  unit-clock move
4  tag relation sec = 60.0 * min
5  min implies move
6  min time delayed by 1.0 on min implies min
```

Lines 1 to 3 declare clocks `sec` and `min` with rational tags, and clock `move` with the unit tag. The constraint `sporadic` enforces a tick on `min` with tag 0. Line 4 specifies that time on `sec` flows 60 times as fast as on `min`. Line 5 requires that each time the `min` clock ticks, the `move` clock ticks as well. Line 6 forces clock `min` to be periodic with period 1.0, specifying that it ticks every minute. The grammar of such expressions is detailed in subsection 3.2.



(a) Returned by simulation engine



(b) Alternative possibility at instant 3

Fig. 2. Two partially satisfying runs

TESL is a specification language that defines the *set* of possible execution traces or *runs* of a global system. In Figure 2 we present two of them; runs are presented by ticks (solid rectangles) timestamped with tags (small numbers) on the time-scales of the clocks `sec`, `min` and `move`; additionally, they are grouped in a sequence of synchronization *instants* (dashed rectangles).

Note that an infinity of other runs satisfy this specification, both from an architectural point of view (runs with additional clocks) and from a behavioral point of view (runs with additional ticks or instants). For instance, Figure 2(b) shows a run with an additional tick on `move`, which may correspond to a movement of the minute hand caused by setting the time on the watch.

The original TESL simulator only computes "minimal" runs, as shown in Figure 2(a), which makes its interpretation deterministic. Since our objective is to turn TESL into a specification language for timed behaviors, we consider not only minimal runs of the system, but *any* run of a given specification.

For more information about TESL and more application-oriented examples involving multiform time and heterogeneous time scales, see the TESL gallery; its engine ignition example[3] may be the most illustrative one.

3 Operational Semantics

In this section, we define an operational semantics of TESL for deriving all possible runs satisfying a given TESL specification. The operational semantics works with a specification of the future of the run, and instantiates it in the present instant, which incrementally builds a set of constraints on the runs. This process also extends the specification of the future when a choice for the present has consequences in the future because of time delayed implications.

3.1 Runs

We describe the execution of a model as a sequence of instants, each instant being a map from *clocks* to *event occurrences*. The latter are represented by a

[3] `http://wdi.supelec.fr/software/TESL/` `http://wdi.supelec.fr/software/TESL/GalleryEngine`

boolean indicating the occurrence and a time tag which gives the date of the occurrence. Such a sequence of instants is called a *run*. More formally, we define:

$\mathbb{K}$	set of clocks $K_1, K_2, \ldots$
$\mathbb{B}$	booleans
$\mathbb{T} = \biguplus_{K \in \mathbb{K}} \mathbb{T}_K$	universe of tags, with $\mathbb{T}_K$ the domain of tags of clock K
$\Sigma = \mathbb{K} \to (\mathbb{B} \times \mathbb{T})$	set of instants
$\Sigma^{\propto} = \mathbb{N}^+ \to \Sigma$	set of runs
ρ_n	n^{th} position (instant) in the run $\rho \in \Sigma^{\propto}$

where $\biguplus$ is the disjoint union operator. Informally, some tag type conditions apply: for a given instant $\sigma \in \Sigma$, a clock K maps to an event occurrence with a fixed tag domain $\mathbb{T}_K$.

Additionally, we define two projections to extract the components of an instant for a given clock:

$\mathsf{ticks}(\sigma(K))$	ticking predicate of clock K at instant $\sigma \in \Sigma$ (first projection)
$\mathsf{tag}(\sigma(K))$	tag value of clock K at instant $\sigma \in \Sigma$ (second projection)

For instance, if we write ρ as the run in Figure 2(a), we have

$$\begin{array}{cc} \mathsf{ticks}(\rho_1(\mathsf{sec})) = \mathsf{false} & \mathsf{ticks}(\rho_1(\mathsf{min})) = \mathsf{true} \\ \mathsf{tag}(\rho_1(\mathsf{min})) = 0.0 & \mathsf{tag}(\rho_1(\mathsf{move})) = () \end{array}$$

3.2 TESL Specifications

A TESL specification φ is a set of atomic constraints that must all be satisfied by a conforming run. To simplify notations, we write them as conjunctions, and we ignore clock types and some operators of the full TESL. Here is a grammar:

$$\begin{array}{lcl} \varphi & ::= & \langle \mathit{atom} \rangle \wedge \cdots \wedge \langle \mathit{atom} \rangle \\ \langle \mathit{atom} \rangle & ::= & \langle \mathit{clock} \rangle\ \texttt{sporadic}\ \langle \mathit{tag} \rangle \\ & | & \langle \mathit{clock} \rangle\ \texttt{sporadic}\ \langle \mathit{tag} \rangle\ \texttt{on}\ \langle \mathit{clock} \rangle \\ & | & \texttt{tag relation}\ \langle \mathit{clock} \rangle = \langle \mathit{tag} \rangle \times \langle \mathit{clock} \rangle + \langle \mathit{tag} \rangle \\ & | & \langle \mathit{clock} \rangle\ \texttt{implies}\ \langle \mathit{clock} \rangle \\ & | & \langle \mathit{clock} \rangle\ \texttt{time delayed by}\ \langle \mathit{tag} \rangle\ \texttt{on}\ \langle \mathit{clock} \rangle\ \texttt{implies}\ \langle \mathit{clock} \rangle \\ \langle \mathit{clock} \rangle & \in & \mathbb{K} \\ \langle \mathit{tag} \rangle & \in & \mathbb{T} \end{array}$$

We also define the subset of *sporadic specifications* as follows:

$$\mathsf{Sporadic}(\varphi) \quad = \quad \{\varphi_{\mathsf{atom}} \in \varphi \mid \varphi_{\mathsf{atom}} \text{ is a } \texttt{sporadic} \text{ atom}\}$$

The expression c_1 `sporadic` τ `on` c_2 is a generalization of the sporadic statement. It means that clock c_1 has to tick in an instant where time is τ on the time scale of c_2. Therefore c_1 `sporadic` τ is the same as c_1 `sporadic` τ `on` c_1.

3.3 Primitives for Run Contexts

Symbolic runs are defined by *run contexts* constructed from a set of constraint primitives introduced below. Run contexts may contain variables that can be arbitrarily instantiated; instances of symbolic runs with ground terms are called *concrete runs*.

Definition 1 (Run context). *A run context Γ is a set containing constraint primitives of the following kind:*

$K \Uparrow_n$	clock K is ticking at instant index n
$K \not\Uparrow_n$	clock K is not ticking (idle) at instant index n
$K \Downarrow_n x$	clock K has timestamp (tag) x at instant index n
$x_1 = \alpha \times x_2 + \beta$	affine relation between x_1 and x_2 with constants α, β

where symbols x, x_1, x_2 can be variables or tag constants in $\mathbb{T}$.

Note that a symbolic run can be instantiated as an infinite number of concrete runs. We give below the interpretation of symbolic runs as concrete runs:

$$\begin{aligned}
[\![\Gamma]\!] &= \bigcap_{\gamma \in \Gamma} [\![\gamma]\!] \\
[\![K \Uparrow_n]\!] &= \left\{\rho \in \Sigma^\infty \mid \mathsf{ticks}(\rho_n(K)) \text{ is true}\right\} \\
[\![K \not\Uparrow_n]\!] &= \left\{\rho \in \Sigma^\infty \mid \mathsf{ticks}(\rho_n(K)) \text{ is false}\right\} \\
[\![K \Downarrow_n \tau]\!] &= \left\{\rho \in \Sigma^\infty \mid \mathsf{tag}(\rho_n(K)) = \tau\right\} \\
[\![\tau_1 = \alpha \times \tau_2 + \beta]\!] &= \left\{\rho \in \Sigma^\infty \mid \tau_1 = \alpha \times \tau_2 + \beta\right\}
\end{aligned}$$

It is possible to construct run contexts that contain contradictory primitive constraints. They are interpreted as the empty set reflecting the fact that they do not denote any concrete run. We observe the following:

Lemma 1. *The consistency of a context Γ –* i.e. *whether $[\![\Gamma]\!] \neq \varnothing$ – is decidable.*

Proof sketch. The affine relations described above belong to the class of linear arithmetic problems which are known to be decidable for integers and rationals, using Fourier-Motzkin elimination. The propositional part is a SAT problem and their combination remain decidable.

3.4 Configurations of the Execution Process

We now define the machinery for constructing symbolic runs. We chose to treat TESL as a logic of resources, where some TESL formulae (such as `sporadic`, which denotes a single event occurrence) are consumed, while others (such as `implies`, which denotes a permanent constraint) are persistent. Processing these formulae produces additional constraint primitives, which refine the shape of satisfying symbolic runs.

The *rules* of our operational semantics relate *configurations* of our symbolic execution process, similarly to triples in a Hoare logic. Configurations consist of:

n	current simulation step index
Γ	run context containing primitives, describing the "past"
ψ	TESL-formula to satisfy in the "present"
φ	TESL-formula to satisfy in the "future" of the process

and are formally introduced in:

Definition 2 (Configuration). *A* configuration *is a tuple* $(\Gamma, n, \psi, \varphi)$ *that we write as* $\Gamma \models_n \psi \triangleright \varphi$

The operational semantics can be seen as an abstract machine, in which a configuration corresponds to an abstract state comprising the past (Γ), present (ψ) and future (φ) of the symbolic run under construction. Intuitively, the abstract machine constructs a symbolic run by refining the current configuration via the actions:

1. moving or duplicating parts from the future to the present (introduction)
2. then, consuming the present to produce the past (elimination)

3.5 Execution Rules

The execution rules of our abstract machine are defined by the $\rightarrow$ relation, which we decompose into $\rightarrow_i$ and $\rightarrow_e$ to identify introduction and elimination rules.

Introduction Rule for Instant Initialization We build a run by adding instants to it. Initializing an instant makes time progress by copying constraints (defined in subsection 3.2) from the future to the present. Sporadic constraints are moved (consumed) rather than copied. Initializing an instant consists of:

– checking that the present constraints of the previous instant have been consumed (*i.e.* $\psi = \varnothing$)
– copying permanent constraints from φ to ψ
– moving sporadic constraints from φ to ψ

This is defined by rule $\mathsf{instant}_i$, whose goal is to initialize a new instant.

$$\Gamma \models_n \varnothing \triangleright \varphi \quad \rightarrow_i \quad \Gamma \models_{n+1} \varphi \triangleright \big(\varphi - \mathsf{Sporadic}(\varphi)\big) \qquad (\mathsf{instant}_i)$$

As an instant has been created, ψ contains instantaneous constraints that are pending to be instantiated into Γ. We now give reduction rules to eliminate formulae from ψ, adding constraints in Γ for the current instant n.

Elimination Rules for `sporadic-on` Formula K_1 `sporadic` τ `on` K_2 constrains K_1 to tick when the time on K_2 is τ. K `sporadic` τ is syntactic sugar for K `sporadic` τ `on` K. Such a constraint can be satisfied in the current instant, or postponed to a future instant. We therefore have two elimination rules for it:

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \models_n \psi \triangleright \varphi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2) \qquad (\mathsf{sporadic-on}_{e1})$$

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ \begin{matrix} K_1 \Uparrow_n \\ K_2 \Downarrow_n \tau \end{matrix} \right\} \models_n \psi \triangleright \varphi \qquad (\mathsf{sporadic-on}_{e2})$$

Elimination Rule for `tag relation` An affine tag relation has to be satisfied at every instant by adding a constraint on the tags of the corresponding clocks:

$$\Gamma \models_n \psi \wedge (\texttt{tag relation } K_1 = \alpha \times K_2 + \beta) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ \begin{matrix} K_1 \Downarrow_n \mathsf{tag}^n_{K_1} \\ K_2 \Downarrow_n \mathsf{tag}^n_{K_2} \\ \mathsf{tag}^n_{K_1} = \alpha \times \mathsf{tag}^n_{K_2} + \beta \end{matrix} \right\} \models_n \psi \triangleright \varphi \qquad (\mathsf{tagrel}_e)$$

where $\mathsf{tag}^n_{K_1}$ and $\mathsf{tag}^n_{K_2}$ are symbolic values, which will be instantiated with ground values in concrete runs.

Elimination Rules for `implies` An implication is satisfied at every instant, either by forbidding a tick on the master clock (rule $\mathsf{implies}_{e1}$), or by making the slave clock tick also (rule $\mathsf{implies}_{e2}$):

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ implies } K_2) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ K_1 \not\Uparrow_n \right\} \models_n \psi \triangleright \varphi \qquad (\mathsf{implies}_{e1})$$

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ implies } K_2) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ \begin{matrix} K_1 \Uparrow_n \\ K_2 \Uparrow_n \end{matrix} \right\} \models_n \psi \triangleright \varphi \qquad (\mathsf{implies}_{e2})$$

Elimination Rules for `time delayed` implication
K_1 `time delayed by` δt `on` K_2 `implies` K_3 means that whenever K_1 ticks, K_3 will tick after a delay of δt measured on the time scale of K_2. It can be satisfied either by forbidding a tick on K_1 (with primitive $K_1 \not\Uparrow_n$), or by making K_1 tick and adding the corresponding `sporadic-on` constraint to the future formula φ:

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ time delayed by } \delta t \texttt{ on } K_2 \texttt{ implies } K_3) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ K_1 \not\Uparrow_n \right\} \models_n \psi \triangleright \varphi \qquad (\mathsf{time-delayed}_{e1})$$

$$\Gamma \models_n \psi \wedge (K_1 \texttt{ time delayed by } \delta t \texttt{ on } K_2 \texttt{ implies } K_3) \triangleright \varphi \quad \rightarrow_e \quad \Gamma \cup \left\{ \begin{matrix} K_1 \Uparrow_n \\ K_2 \Downarrow_n \mathsf{tag}^n_{K_2} \end{matrix} \right\} \models_n \psi \triangleright \varphi \wedge \left(K_3 \texttt{ sporadic } (\mathsf{tag}^n_{K_2} + \delta t) \texttt{ on } K_2 \right) \qquad (\mathsf{time-delayed}_{e2})$$

3.6 Termination of a Simulation Step

A *simulation step* consists in building the next instant of the symbolic run by:

1. initializing an instant with reduction $\rightarrow_i$ (uniquely defined by rule $\mathsf{instant}_i$);
2. eliminating all ψ-subformulae using $\rightarrow_e$ elimination rules until $\psi = \varnothing$.

A simulation step is more formally defined as a reduction rule, with $\cdot$ the composition of relations, and $\rightarrow_e^*$ the reflexive transitive closure of $\rightarrow_e$:

$$\rightarrow_{\not{t}} := \{(\Gamma_1 \models_n \varnothing \triangleright \varphi_1) \rightarrow_i \cdot \rightarrow_e^* (\Gamma_2 \models_n \varnothing \triangleright \varphi_2) \mid \Gamma_1 \text{ and } \Gamma_2 \text{ are consistent}\} \qquad (\mathsf{simulation})$$

Note that we add a consistency constraint on Γ-contexts as we are interested in symbolic runs that have concrete instances. Indeed, reductions given by $\rightarrow$ are purely syntactical and do not take into account the constraints in Γ. For instance, $\rightarrow_e$ allows adding $K_1 \Uparrow_n$ to a context that already contains $K_1 \not\Uparrow_n$.

The termination of the computation of one simulation step $\rightarrow_{\not{t}}$ is ensured by the termination of $\rightarrow_e$, because the number of formulae in ψ strictly decreases when a rule is applied. Moreover, whenever ψ is not empty, there is at least one applicable elimination rule, so when $\rightarrow_e^*$ terminates, ψ is necessarily empty and we can proceed with the next simulation step.

Following the specification given as an example in section 2 (denoted as φ_0), we illustrate the use of our operational rules in Figure 3. We start with an empty symbolic run and show the two first simulation steps on the left hand-side. Then we focus on the first step and provide the underlying reduction details on the right-hand side. This step is decomposed into the application of the introduction rule $\mathsf{instant}_i$, then a sequence of elimination reductions ($\mathsf{sporadic}-\mathsf{on}_{e2}$, tagrel_e, $\mathsf{implies}_{e2}$, $\mathsf{time}-\mathsf{delayed}_{e1}$), until irreducibility.

4 Heron: a Solver for TESL Specifications

Since the operational semantics can be seen as an abstract execution machine, its implementation is conceptually straightforward. The resulting prototype solver, called Heron[4], is more general than the original deterministic TESL solver since it is not restricted to "minimal" runs. It consists of approximately 2500 lines of Standard ML code, and is compiled with MLton [22]. Heron is a standalone command-line interpreter, which takes a TESL specification as input and produces prefixes of satisfying symbolic runs, written in the Value-Change Dump format [3]. The solver is complete in the sense discussed in subsection 3.6, *i.e.* it produces all satisfying runs up to a fixed step index. Assuming that the 'future' formula contains no contradiction, this means that the satisfying symbolic runs have instances which are *exactly* the prefixes of *all* satisfying concrete runs.

Heron can be used in four modes:

[4] Heron is distributed as free software at `https://github.com/heron-solver/heron`

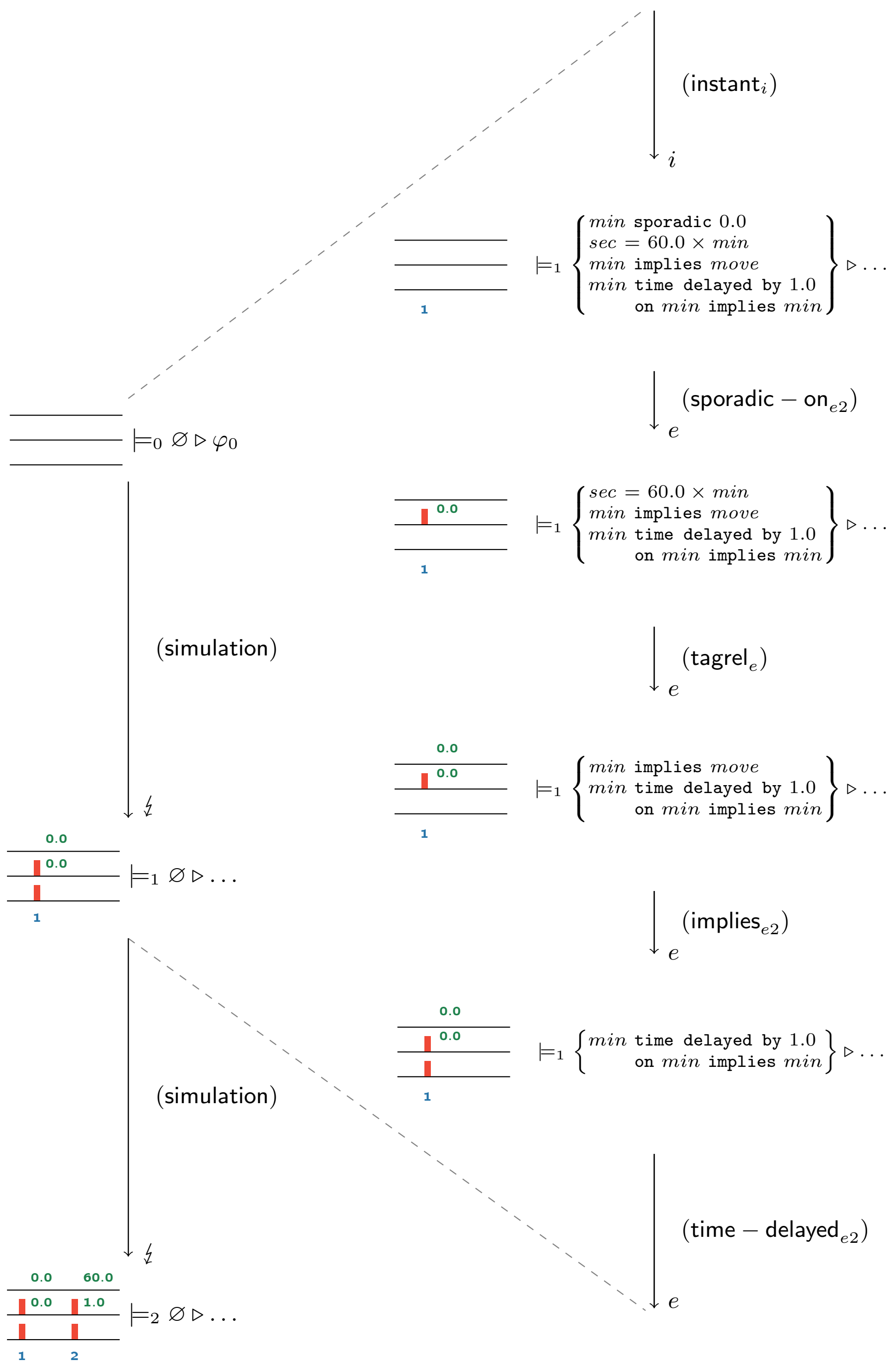


Fig. 3. Detail of the reduction steps of the operational semantics

Exhaustive exploration. The non-deterministic nature of our semantics allows multiple choices for deriving runs. By default, they are all explored when no specific simulation policy is given. In this mode, state-space explosion emerges quickly.

Minimal fast simulation. Several heuristic policies are provided to restrict the state-space, among them, the "minimal run strategy" mimics the original TESL simulator by making events occur as early as possible, and only when mandatory (a clock does not tick unless an implication or a sporadic constraint forces it to tick). These policies turn Heron into an execution engine targeted at specific kinds of runs.

Scenario monitoring. The state-space can also be restricted by the behavior of a concrete system under test (SUT) observed at its interfaces (see Figure 1). The observed behavior — both from the interface of system components and from the architectural glue — is checked against the TESL specification.

Scenario testing. For testing, scenario monitoring is extended with the concept of distinguished *driving-clocks*, for which Heron can produce tagged event instances that are consistent with the current constraint-set (it essentially picks an instance at each instant among the consistent instances). These event-instances can be converted into suitable stimuli for the SUT (however, we have currently not yet implemented a driver for this).

In the following, we discuss the monitoring scenario in more detail and then refine it into a kind of input-output conformance [19] test scenario.

4.1 Conformance Monitoring and Error Detection

The Heron solver can be used as an online monitoring tool, permitting to tackle the infinite number of possibilities for concrete test-runs at all possible instants. The conformance monitoring scenario makes the following assumptions:

1. we assume the monitor has an access to the SUT interfaces (see Figure 1) via a driver that abstracts observations into tagged events on clocks;
2. we assume that the computing time of the driver and of Heron can be neglected with regard to the execution time of the SUT, and
3. we assume that the system is output deterministic; *i.e* after an initialization of the SUT by the tester, it is possible to track the state of the SUT by only observing its inputs *and* outputs [8].

The idea for the monitoring scenario is to filter out the branches in the set of runs maintained by Heron that are no longer compatible with the behavior of the system, as observed through the interfaces. If the SUT produces a behavior that does not conform to the specification, the solver will fail to produce a satisfying configuration and abort.

A monitoring sequence is illustrated in Figure 4. The solver first starts by generating all satisfying states (circled $\models$). It then keeps the states that are compatible with the observed behavior of the SUT (plain circles), while dropping the other ones (dashed grey circles). When the SUT produces a bad behavior (circled $\not\models$), the solver drops all of its states and finds none that match the behavior of the SUT. No further simulation is possible.

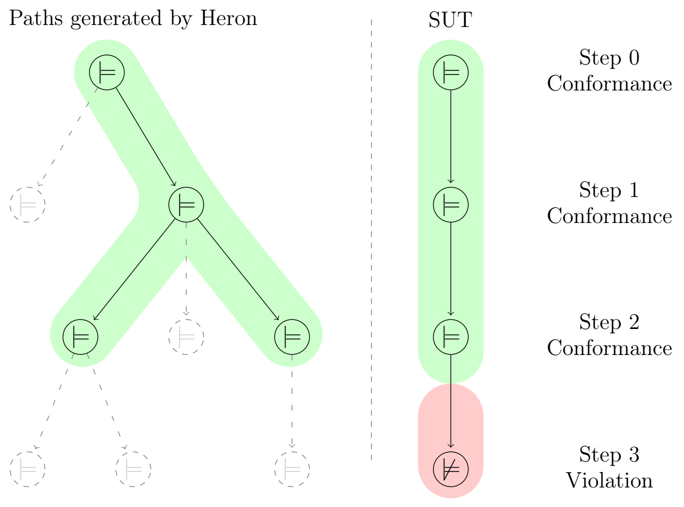


Fig. 4. Executing Heron and the SUT in parallel

Example: based on the specification shown in Listing 2 on page 3, we use the `@scenario` directive to feed Heron with the observed behavior, and the `@step` directive to take this behavior into account and update the reachable states:

```
 7  @scenario strict 1 min move
 8  @step
 9  @scenario strict 2 min move
10  @step
11  @scenario strict 3 move
12  @step
13  @scenario strict 4 min move
14  @step
```

For instance in Line 7, we tell Heron that we observed that clocks `min` and `move` tick at instant 1. The `strict` option indicates that only the given clocks tick, all the others remain idle in that instant. Alternatively, we could use:

```
 9  @scenario strict 2 (min-> 1.0) move
```

to indicate that the tag on clock `min` at this instant is `1.0`. This instantiates the symbolic tag variable in the symbolic run with a concrete tag for clock `min`. Thus, the observations on the concrete run of the SUT can be used to prune execution branches that are not relevant for the future of the run.

In the above example, the solver finds 24 symbolic runs, among them the one shown in Figure 2(b):

```
@print
## Simulation result:
              sec         min         move
[1]           ⊘           ↑ 0.0       ↑
[2]           ⊘           ↑ 1.0       ↑
[3]           ⊘           ⊘           ↑
[4]           ⊘           ↑ 2.0       ↑
```

The output shows a run containing four instants, with a timeline for each of the specified clocks (`sec`, `min`, `move`). A ticking clock is depicted by the upwards arrow (↑) with the associated time tag on the right. An idle clock is depicted by the circled slash (⊘). If nothing is specified for a clock, it can either tick or not.

Property violation. As long as the SUT produces behaviors for which the solver does not detect a contradiction, the observed run "potentially conforms" to the TESL specification. However, if a non-conforming behavior occurs, the solver detects a contradiction in its constraint set. For instance, if in step 3, clock `min` ticks but clock `move` does not, we have:

```
@scenario strict 1 min move
@step
@scenario strict 2 min move
@step
@scenario strict 3 min
@step
```

In this case, the solver detects the violation of the `min implies move` formula.

4.2 Input/output Conformance Testing

We consider online testing as an extension of online monitoring with a policy for generating input stimuli on the fly. This policy explores the state space with respect to a particular coverage criterion.

In order to use Heron as an online testing tool, the clocks that are considered as *inputs* must be declared as *driving-clocks*:

```
@driving-clock move
```

After this declaration, Heron may be instrumented by:

```
@event-solve 2
```

which leads to the invocation of a constraint solver (lemma 1) for step 2, which by default choses for the driving clocks, an input that satisfies the constraints. More sophisticated generation policies could be implemented.

Conformance: if the future of a configuration (see subsection 3.4) becomes empty or *stable*, the observed run "fully conforms" to the TESL specification. A (future) specification is stable, if it represents a Buchi-automaton producing an infinite behaviour such as:

```
min time delayed by 1.0 on min implies min
```

which represents an infinite stream of event occurrences, each separated from the previous one by a 1.0 time delay measured on the time scale of clock `min`. For the moment, we only have an incomplete set of patterns to characterize stable specifications. Moreover, we cannot conclude if we do not reach such a configuration during the test, which corresponds to the classical *inconclusive* situation in conformance testing.

4.3 Performance

We give some benchmarks that were made on a conventional laptop computer with an Intel Core™ i5-2520M CPU @ 2.50GHz and 8 GB of RAM. They are based on examples provided by the official gallery of TESL and fully logged in Table 1. They highlight the state-space explosion for exhaustive paths, while depicting the feasibility of scenario monitoring of a SUT.

Example \ Policy and steps		Exhaustive				Minimal Run				SUT Monitoring			
		1	2	3	4	1	2	3	4	1	2	3	4
HandWatch	Time	0.02	0.00	0.01	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02
	Memory	2412	3124	6464	10264	2592	2512	3220	3220	2496	3236	3892	5768
LightSwitch	Time	0.00	0.06	3.20	10:02.81	0.00	0.00	0.01	0.02	0.00	0.02	0.04	0.11
	Memory	3132	9872	288120	4029676	3172	5300	7088	7064	3180	7080	8140	12444
ConcurrentComp	Time	0.00	1.77	10:26.32	Timeout	0.02	0.06	0.08	0.06	0.02	0.23	1.19	3.27
	Memory	7064	145208	4029688		7120	7916	7856	7860	7136	15956	68864	121884
LeapYears	Time	0.01	3.24	15:12.41	Timeout	0.05	0.06	0.07	0.08	0.01	0.52	1.12	1.53
	Memory	8320	217688	4029792		8356	8384	8260	8360	8332	39832	39820	39884
Engine	Time	0.00	0.03	0.32	8.34	0.00	0.01	0.01	0.01	0.00	0.02	0.04	0.08
	Memory	3212	7752	20728	342240	3300	4780	6628	7196	3252	7384	8044	8460

Table 1. Time (in sec or min:sec) and memory usage (in kB) with respect to a given policy and a fixed number of simulation steps for several examples of the TESL gallery

5 Conclusion

We have presented the Tagged Event Specification Language (TESL) to specify the timing behavior at the interfaces of components of an heterogeneous system. We have defined its operational semantics by a set of symbolic evaluation rules, permitting the construction of symbolic representations of infinite sets of timed behaviors (runs). We have shown how our semantics leads naturally to an implementation of a solver that can be used to monitor and test the architectural glue of heterogeneous systems with timed behavior.

The introduction of *driving-clocks* (see subsection 4.2) paves the way for the distinction between mere observations of the SUT (and their relative check of conformance) and the stimulation by timed inputs consistent with the constraint set that is monitored in a particular symbolic run. This gives TESL the flavor of an input-output automaton or labelled transition system for which a well-known theoretic testing framework exists [19] which also has been extended to timed behavior [17,13]. Due to their proximity to model-checking, these frameworks are usually restricted to discrete time and cannot treat causality. To overcome the former limitation, an entire research community emerged under the label *online testing* [2] which discusses techniques based on symbolic execution in parallel to test execution. Our work can be seen as a form of online testing for heterogeneous timed systems with arbitrary linear relations between time scales.

Related work. The TESL language is sourced from different ideas. It originally started as a complementary approach to the CCSL specification language, by keeping purely synchronous logical clocks, while adding support for time tags and time scale relations as described in the Tagged Signal Model [14], which allows specifying the occurrence of events after a chronometric delay. The original solving algorithm relies on a constructive semantics in the style of the Esterel synchronous language [5]. Compared to CCSL, the restriction to purely synchronous constraints in TESL comes from the necessity to compute time tags, which is not possible when asynchronous relations give only precedence constraints on event occurrences. The style of executable semantics we give in this article is similar to [23] but we abstract time with symbols while preserving the bounded computability of the run state-space.

The idea of a timed architectural composition language is conceptually similar to orchestration languages for web-services, for example BPEL [4] and more formal treatments thereof such as [21]. BPEL is designed to organize and synchronize a set of communication threads, called conversations. In contrast to TESL, BPEL-like languages allow for dynamic thread creation and therefore a dynamic evolution of channels and interfaces; however, they are not designed to treat time, duration, and causality of possibly periodic events.

Future Work. A strengthening of both foundational as well as practical aspects of the TESL language is desirable. Although the operational semantics has been carefully designed, there is no formal proof of the inherent logical consistency of the rule set: to this end, a denotational version is currently under development (which assures consistency by construction) in Isabelle/HOL [16] which could serve as a reference in a validity proof of these rules (which thus become *derived*). This would allow also pave the way to describe the exchange of *data* between sub-components, either process-oriented [18] or program-oriented [7]. Furthermore, the conformity of a SUT to a spec S can only be established when the possible futures becomes either trivial or stable. For now, this can only by decided for certain patterns based on an automata-based reasoning. The denotational semantics may help to find a less ad-hoc characterization of “stability” based on co-induction. On the practical side, we wish to explore more refined heuristics to monitor and test heterogeneous systems with Heron.

References

1. UML profile for MARTE™: Modeling and analysis of real-time embedded systems™, `http://www.omg.org/spec/MARTE/1.1/`
2. International online testing symposion (1995-2017), `http://tima.imag.fr/conferences/iolts/`
3. IEEE standard verilog hardware description language. IEEE Std 1364-2001 (2001), `https://doi.org/10.1109/IEEESTD.2001.93352`
4. Specification: Business process execution language for web services version 1.1 (2003), `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/`
5. Berry, G.: The constructive semantics of pure Esterel (1996)
6. Boulanger, F., Jacquet, C., Hardebolle, C., Prodan, I.: TESL: a language for reconciling heterogeneous execution traces. In: Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2014). pp. 114–123. Lausanne, Switzerland (Oct 2014), `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6961849`
7. Brucker, A.D., Wolff, B.: An extensible encoding of object-oriented data models in HOL. J. Autom. Reasoning 41(3-4), 219–249 (2008), `https://doi.org/10.1007/s10817-008-9108-3`
8. Brucker, A.D., Wolff, B.: Monadic sequence testing and explicit test-refinements. In: Tests and Proofs - 10th International Conference, TAP 2016, Held as Part of STAF 2016, Vienna, Austria, July 5-7, 2016, Proceedings. pp. 17–36 (2016), `http://dx.doi.org/10.1007/978-3-319-41135-4_2`
9. Combemale, B., Cheng, B.H., France, R.B., Jezequel, J.M., Rumpe, B.: Globalizing Domain-Specific Languages, LNCS, Programming and Software Engineering, vol. 9400. Springer International Publishing (2015), `https://hal.inria.fr/hal-01224096`
10. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. In: Proceedings of the IEEE. pp. 127–144 (2003)
11. Garcés, K., Deantoni, J., Mallet, F.: A Model-Based Approach for Reconciliation of Polychronous Execution Traces. In: SEAA 2011 - 37th EUROMICRO Conference on Software Engineering and Advanced Applications. IEEE, Oulu, Finland (Aug 2011), `https://hal.inria.fr/inria-00597981`
12. Hardebolle, C., Boulanger, F.: Exploring multi-paradigm modeling techniques. SIMULATION: Transactions of The Society for Modeling and Simulation International 85(11/12), 688–708 (November/December 2009), `/software/downloads/ModHelX/2009MPMSimulation.pdf`
13. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Form. Methods Syst. Des. 34(3), 238–304 (Jun 2009), `http://dx.doi.org/10.1007/s10703-009-0065-1`
14. Lee, E.A., Sangiovanni-Vincentelli, A.: A framework for comparing models of computation. IEEE Trans. CAD 17(12) (1998)
15. Mallet, F., Deantoni, J., André, C., De Simone, R.: The Clock Constraint Specification Language for building timed causality models. Innovations in Systems and Software Engineering 6(1-2), 99–106 (Mar 2010), `https://hal.inria.fr/inria-00464894`
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

17. Schmaltz, J., Tretmans, J.: On Conformance Testing for Timed Systems, pp. 250–264. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-85778-5_18
18. Tej, H., Wolff, B.: A corrected failure divergence model for CSP in Isabelle/HOL. In: FME '97: Industrial Applications and Strengthened Foundations of Formal Methods, 4th International Symposium of Formal Methods Europe, Graz, Austria, September 15-19, 1997, Proceedings. pp. 318–337 (1997), https://doi.org/10.1007/3-540-63533-5_17
19. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
20. Vara Larsen, M.E., Deantoni, J., Combemale, B., Mallet, F.: A Behavioral Coordination Operator Language (BCOoL). In: 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015) (Aug 2015), https://hal.inria.fr/hal-01182773
21. Viroli, M.: A core calculus for correlation in orchestration languages. The Journal of Logic and Algebraic Programming 70(1), 74 – 95 (2007), http://www.sciencedirect.com/science/article/pii/S1567832606000300
22. Weeks, S.: Whole-program Compilation in MLton. In: Proceedings of the 2006 Workshop on ML. pp. 1–1. ML '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1159876.1159877
23. Zhang, M., Mallet, F.: An Executable Semantics of Clock Constraint Specification Language and Its Applications, pp. 37–51. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-29510-7_2