TESL: A language for reconciling heterogeneous execution traces

Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, Iuliana Prodan

TESL: a Language for Reconciling Heterogeneous Execution Traces

Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle and Iuliana Prodan
Supélec E3S
3 rue Joliot-Curie
91192 Gif-sur-Yvette
France
Email: Firstname.Name@supelec.fr

***Abstract*—Various formalisms deal with time, and each of them has its own notion of time. When designing a system, it is often desirable to combine several of these formalisms to model different parts. Therefore one has to reconcile execution traces that may use different kinds of time (discrete, continuous, periodic) and different time scales (e.g. minutes, microseconds or even angles in degrees). In this article, we present a deterministic model of time which allows the specification of the coincidence of events that occur on different time scales, as well as instantaneous causality between events. This model supports both event-driven and time-driven specifications.**

I. Introduction

Various modeling paradigms deal with time [1], for example timed finite state machines, timed Petri nets, discrete event formalisms, etc. These formalisms use different notions of time (discrete, continuous, periodic). When designing a system, it is often necessary to combine several of them: a state machine for the controller, a discrete events model for the system bus, a periodically sampled model for signal processing, a continuous time model for the dynamics of the environment, etc. To give semantics to the overall model, we need to reconcile the execution traces of the various formalisms used [2], that obey different semantics and may not share the same notions of time and control.

Therefore we need a model of time that allows us to express the relations between the control flows and time scales of several models of heterogeneous natures. Two aspects have to be handled in order to do this: (a) modeling causality between the occurrences of events, (b) modeling how time advances on different time scales. After examining several other frameworks for modeling time and events, we concluded that most of them are oriented toward verification or are specific to event-triggered models. The closest match to our needs was CCSL [3], [4] (Clock Constraints Specification Language), but it does not support the specification of arbitrary durations for delays.

Therefore, in this article we introduce TESL, the Tagged Events Specification Language, inspired from CCSL, but that addresses our more specific needs. TESL is both a library with a Java API to build models and solve them step by step, and a language with a textual concrete syntax (used in this article) from which the solver produces timing diagrams. Both are publicly available as an Eclipse feature[1] under the EPL license.

This work is part of the ANR INS Project GEMOC (ANR-12-INSE-0011).
[1]TESL home page: http://wwwdi.supelec.fr/software/TESL/

In the following, we start in section II by introducing several formalisms for modeling time, and their limitations with regard to our goal. Then, we present the TESL model and its concrete syntax in section III, and show how it can be used to model and solve problems through some examples. Section IV describes the algorithm used to solve TESL specifications. Complete examples are shown in section V to illustrate the capabilities of TESL. An overview of the use of TESL for reconciling heterogeneous execution traces is given in section VI, and perspectives for a formal semantics of TESL are discussed in section VII. Then, we discuss the main differences between TESL and CCSL in section VIII, and give some perspectives in section IX before concluding.

II. Related Work

This section reviews several frameworks for modeling time, and compares them with respect to the needs stated in introduction.

Timed Automata [5] and DEVS [6] are modeling formalisms which handle both discrete events and continuous time. In the domain of multi-paradigm modeling, DEVS is used for instance in MoTif for timing the model transformations used to define the semantics of DSLs in AtoMPM [7]. These formalisms allow both the formal analysis of models and efficient simulation of discrete events models with timed transitions. However, they rely on a global homogeneous time which flows at the same pace in every part of the model and they have no direct support for polymorphic time and the expression of relations between different time scales.

MARTE [8] is a UML profile for the Modeling and Analysis of Real-Time Embedded systems. It contains a sub-profile which is dedicated to the modeling of time. The Time sub-profile of MARTE is able to model the defects of real clocks compared to ideal clocks (for instance jitter). It relies on both synchronous and asynchronous relations between instants on different clocks (coincidence, precedence). It also supports polymorphic time: a clock may measure actual time as well as an angle of rotation, a distance, the number of persons entering a room, etc. Several schedulability analysis tools may be applied to MARTE models: RapidRMA [9], Cheddar, MAST, Xoncrete [10]. These tools focus on finding or checking an execution schedule for tasks that have deadlines and periods, which constitutes only a subset of MARTE. The only kind of dependency between tasks is the use of shared resources,

which leads to mutual exclusion. There is no way of handling more general implication relations, or asynchronous events.

CCSL [11] is a language for specifying constraints on clocks. It relies on a model of time similar to the model of MARTE, but clocks have no associated time scale. The only way to measure time in CCSL is by counting ticks. It is not possible to specify a place in time on a clock if there is no tick at this time on the clock. CCSL comes with the Time^2 (pronounced *Time square*) model-checker, which detects inconsistencies in a clock model, or builds a possible time-line if the specification is consistent. In previous work [12], we used CCSL to model the semantic adaptation of control between a discrete event model, a state machine and a synchronous dataflow model. However, without the possibility to specify the occurrence of an event at an arbitrary time, we could not model the occurrence of an event with an arbitrary delay after the occurrence of a triggering event. On the positive side, the possibility to use precedence constraints between clocks in CCSL (and not only coincidence constraints) makes it very suitable for specifying models of computation (MoC) [13]. An MoC only states the properties of the execution of a model, without forcing one particular execution. Therefore, CCSL can help validating a model independently of its execution platform.

Time in Synchronous Languages [14] such as Lustre, Esterel, Signal and Lucid Synchrone is abstract and polymorphic. This model of time allows the deterministic parallel composition of processes, efficient sequential code generation as well as the verification of safety properties by model-checking. However, the abstract nature of time in synchronous languages ignores durations and is purely event based, like in CCSL (although CCSL is not synchronous).

The Tagged Signal Model (TSM) [15] and **Tag Machines [16]** are frameworks for modeling heterogeneous notions of time for signals and systems. They model a signal as a series of samples, and attach a time tag to each sample. The tags for each signal belong to a time domain, and the properties of the time domain (being a metric space, being discrete, continuous, dense) give the properties of time. Morphisms between time domains can be used to model the semantic adaptation of time. These frameworks are very generic and can account for any existing notion of time. However, they only provide ways of integrating the notions into a theoretical framework: there is no associated toolset available to solve actual time constraints while taking inputs into account at runtime.

Conclusion: the MARTE metamodel of time appears to be quite complete, but all implementations are partial. CCSL, one of the most prominent of them, is purely event-based, like synchronous languages. In contrast, we need timestamps, tags like in TSM, as well as morphisms between time domains. Therefore we have created TESL, a new metamodel accompanied by a solving algorithm.

III. Introduction to TESL

TESL was inspired from CCSL. It retains only the synchronous relations between clocks, and adds support for tags (which are dates or timestamps) as in the Tagged Signal Model or the Time profile of MARTE. Removing asynchronous relations between clocks makes TESL deterministic, which is an advantage for simulation. Adding support for tags also improves the performance of the simulator which can jump directly to the next time tag without having to count events whose sole purpose is to make time advance. The design choices for TESL are oriented toward the execution of heterogeneous models and are therefore different from the choices made for CCSL or MARTE which are oriented toward specification and verification. The model of time used in TESL is similar to the one used in Timed Automata, except that it does not rely on a global dense clock and allows various tag domains for clocks.

A. Meta-Model

In TESL, a recurring event is modeled by a *clock*, and each occurrence of the event is represented by a *tick* of the clock. A clock c has a time domain $\mathrm{dom}(c)$, with a total order on it. Examples of time domains include $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ and $\mathbb{R} \times \mathbb{N}$ (superdense time [17]). Every tick t of a clock c has a *tag* $\mathrm{tag}(t) \in \mathrm{dom}(c)$. The special Unit domain, which contains a single, meaningless value, is used for clocks with no notion of time. Such clocks are used to model events which can occur simultaneously with any event, without an intrinsic time scale.

Ticks on a clock specify occurrences of the event modeled by the clock. In a specification, a tick on a clock c has a tag in $\mathrm{dom}(c) \cup \{\bot\}$, which gives the date at which the event should occur. The special $\bot$ value indicates a tick that should occur as soon as possible according to the relations between the clocks. Such a $\bot$-tick is called a *floating tick*, and its tag in $\mathrm{dom}(c)$ is determined when the specification is solved.

The solution to a TESL specification is a discrete series of instants, potentially infinite but countable. At each instant, each clock may have a tick which belongs to the instant, and if so, this tick has a tag in the time domain of the clock. For a given clock, ticks that belong to successive instants must have non-decreasing tags. The relations between the clocks must hold at each instant, and it is not allowed to put a tick in an instant if all the ticks with a lower tag on the same clock have not already been assigned to previous instants. This last condition means that causality must be respected on each clock, and that every tick (which specifies an occurrence of the event) must be assigned to exactly one instant of the series.

B. Implication relations

TESL allows one to specify *implication relations* between clocks. If a clock a implies a clock b, then each instant which contains a tick on a also contains a tick on b. There are several variants of this simple implication, that all amount to computing the presence of a tick on b (the *slave* clock) using a state machine which reacts to the presence of ticks on a set of *master clocks* at successive instants. Notable flavors of implication relations include[2]:

- `a` **`implies`** `b` b ticks each time a ticks (but there can be other ticks on b).
- `a` **`sustained from`** `start` **`to`** `stop` **`implies`** `b` the implication is initially *off*. It switches *on* when $start$ ticks and *off* when $stop$ ticks. A tick on a implies a tick on b only when the implication is *on*.

[2]The complete reference manual of TESL is available at http://wwwdi.supelec.fr/software/TESL/RefMan

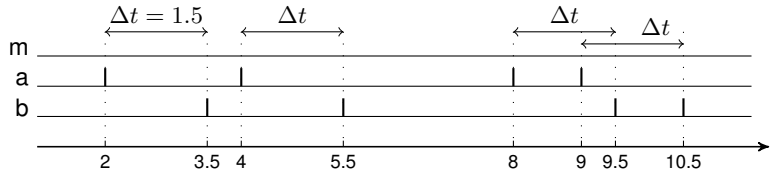

Fig. 1. Time delayed implication: `a` **`time delayed by`** `1.5` **`on`** `m` **`implies`** `b`.

- `a` **`next to`** `t` **`implies`** `b` puts a tick on b at an instant if a also has a tick at this instant and t has occurred since the last occurrence of a.
- `a` **`when`** `s` **`implies`** `b` if a and s have a tick at the same instant, b also has a tick at this instant.
- `a` **`filtered by`** `s,k (rs, rk)*` **`implies`** `b` filters a by skipping s ticks, then keeping k ticks, then repeatedly skipping rs and keeping rk ticks.
- `a` **`delayed by`** `count` **`on`** `c` **`implies`** `b` creates a tick on b when c ticks for the $count^{th}$ time after a has ticked.
- **`await`** `a b` **`implies`** `c` creates a tick on c each time both a and b have ticked at least once.

There are several variants of these relations, depending on whether the implication starts (resp. stops) as soon as the condition is met, or only at the next instant (which is the default). For instance, **`sustained immediately`** is instantaneously switched on when $start$ ticks. Likewise, **`immediately delayed`** starts counting on c at the very instant when a ticks. `a` **`next to`** `t` **`implies`** `b` is syntactic sugar for `a` **`sustained immediately from`** `t` **`to`** `a` **`implies`** `b`, and the non-immediate variant also exists.

All of the above implication relations create ticks on the slave clock at the current instant by reacting to the presence of ticks on other clocks, and could be modeled using the synchronous subset of CCSL. TESL also has relations which deal with time scales. Instead of waiting for a delay by counting ticks on a clock, it is possible in TESL to wait for a duration to elapse on a time scale. For instance `a` **`time delayed by`** `1.5` **`on`** `m` **`implies`** `b` creates a tick on b 1.5 time units of clock m after a tick on a (assuming the time domain of m is $\mathbb{Q}$ or $\mathbb{D}$), as illustrated on Figure 1. If the time on m is t_0 when a occurs, there will be a tick on b when time reaches $t_0 + 1.5$ on m. This does not require that m ticks, this clock is just used as a time scale.

The way time advances on different clocks is specified using *tag relations*, studied in the next paragraph. Tag relations and timed delays introduce *time-triggered behavior*, in contrast to implication relations that specify *event-triggered behavior* (ticks are created in response to the occurrence of other ticks).

C. Tag relations

Tag relations relate the tags of two clocks by means of a bi-directional mapping between their domains. For instance, if clock m counts minutes, and clock h counts hours, we can specify a bijective mapping between the tags of ticks on m and h that belong to the same instant (assuming their domain is $\mathbb{Q}$ or $\mathbb{R}$). If t_m is the tag of a tick on m and t_h the tag of a tick on h, we can choose $t_m = 60 \times t_h + 17$ for instance (here assuming an offset of 17 minutes between the two clocks).

Tag relations introduce a notion of coincidence of event occurrences by identity of their tags (modulo the relations). For instance, if a given instant contains a tick h_2 with tag 2 on clock h, and if m has a tick m_{137} with tag $137 = 60 \times 2 + 17$, m_{137} also belongs to this instant because the tags of h_2 and m_{137} are congruent modulo the tag relation. Conversely, if m has a *floating* tick, which specifies an occurrence as soon as possible, it is put in the current instant and its tag is set to 137 because this is the current time on m according to the current time on h and the tag relation.

The general form of a tag relation between clocks a and b is a pair of non-decreasing functions $d : \text{dom}(a) \to \text{dom}(b)$ and $r : \text{dom}(b) \to \text{dom}(a)$. Given two tags $\tau_a \in \text{dom}(a)$ and $\tau_b \in \text{dom}(b)$, τ_a and τ_b are in relation if either $\tau_b = d(\tau_a)$ or $\tau_a = r(\tau_b)$. For instance, with $\text{dom}(a) = \mathbb{Z}$ and $\text{dom}(b) = \mathbb{Q}$, we may define:

$$d : i \mapsto 2\frac{i}{1} + 3 \quad \text{and} \quad r : q \mapsto \left\lfloor \frac{q-3}{2} \right\rfloor$$

which will for instance make 1 be in relation with both $\frac{5}{1}$ and $\frac{6}{1}$ since $r(\frac{5}{1}) = r(\frac{6}{1}) = 1$. In order to form a tag relation, the functions d and r also have to be consistent, that is: $d \circ r \circ d = d$, and $r \circ d \circ r = r$, so that they do not make ticks with different tags on a given clock simultaneous.

Determining which ticks belong to an instant and computing their tags is at the heart of the solving algorithm (see Section IV). Tag relations allow time to advance on a clock even when it has no tick, and make it possible to specify *time-triggered behavior*.

D. Example: Light Switch

We consider a compact fluorescent light which is switched on by pushing a button and is automatically switched off 1 minute later. When switched on, the light takes 1 second to be fully lit. When switched off, it stops producing light after 50 milliseconds. In the following, **`rational-clock`**s are clocks with time domain $\mathbb{Q}$.

We model the physical time in this example using three clocks named `ms`, `s` and `min` which respectively measure milliseconds, seconds and minutes. Tag relations specify that time flows 1000 times as fast on the `ms` clock as on the `s` clock, and 60 times as fast on the `s` clock as on the `min` clock. We use a clock with the same time scale as `ms` for modeling the push on the button and use the **`sporadic`** qualifier to specify that the button event will occur at time 500. The switch on, switch off, light on and light off events are modeled using unit-clocks because they correspond to pure events without an intrinsic notion of time or duration.

```
rational-clock ms  // milliseconds
rational-clock s   // seconds
tag relation ms = 1000 * s + 0
rational-clock min // minutes
tag relation s = 60 * min + 0

// button is pushed at time 500
rational-clock button sporadic 500
// measured on the ms scale
tag relation button = ms

unit-clock switch_on  // switch the light on
unit-clock switch_off // switch the light off
unit-clock light_on   // the light becomes on
unit-clock light_off  // the light becomes off

// pushing the button switches the light on
button implies switch_on
// The light is switched off 1 minute later
switch_on time delayed by 1.0 on min
          implies switch_off

// The light is on 1s after being switched on
switch_on time delayed by 1.0 on s
          implies light_on
// and off 5ms after being switched off
switch_off time delayed by 50.0 on ms
           implies light_off
```

Solving this specification produces the timing diagram on Figure 2.

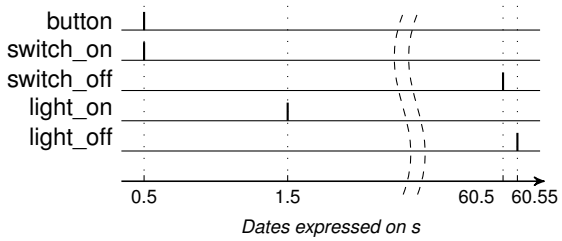

Fig. 2. Simulation of the light switch.

Without the possibility to post event occurrences at a given date on a clock, we would have to count occurrences of the smallest time step in this model, which is 50 ms. We would therefore have 1211 simulation steps instead of 4. This example shows that time-triggered events and tag relations are very efficient for the simulation of systems where the greatest common divisor of the durations between event occurrences is small. However, this comes at the price of introducing arithmetics on the clock domains, which makes verification much more complex.

Another benefit of using tags and tag relations is that it allows us to take inputs into account without knowing in advance when they will occur. For instance, the button clock event could be generated by a GUI during the simulation and be tagged with the current value of the millisecond clock of the computer. All the delays would be computed exactly (up to the resolution of the computer clock) using as few simulation steps as needed, while without tag relations and timed delays, these durations would be rounded down according to the period of the fastest clock in the model. The current implementation of the TESL language has no support for interactive simulation, but using the API of the TESL library, it is possible to create ticks at runtime in response to the simulation environment in order to make a specification run against a live scenario. The TESL library is used to execute models in the ModHel'X [18][3] heterogeneous modeling platform developed in our team.

E. Example: Concurrent Processes

In this example, we model two concurrent computations running each on a CPU. CPU 1 computes some value A in 0.5 units of time starting at time 1, then waits for value B to become available in order to compute $A + B$ in 1 unit of time. CPU 2 computes value B in 1.5 units of time starting at time 2. In this model, we do not specify any tag relation between the time domains of CPU 1 and CPU 2. Therefore, time flows independently on the two CPUs, but causality is preserved because $A + B$ can be computed only once A and B are both available.

```
// time scale on CPU 1
rational-clock CPU1_time
// start computing A at 1
rational-clock compute_A sporadic 1
tag relation compute_A = CPU1_time
unit-clock A_available
compute_A time delayed by 0.5 on CPU1_time
          implies A_available
// time scale on CPU 2
rational-clock CPU2_time
// start computing B at 2
rational-clock compute_B sporadic 2
tag relation compute_B = CPU2_time
unit-clock B_available
compute_B time delayed by 1.5 on CPU2_time
          implies B_available
// start computing A+B when
// both A and B are available
unit-clock compute_A_plus_B
await A_available B_available
      implies compute_A_plus_B

unit-clock A_plus_B_available
compute_A_plus_B time delayed
                 by 1.0 on CPU1_time
                 implies A_plus_B_available
```

The solution to this specification has three instants (see Figure 3).

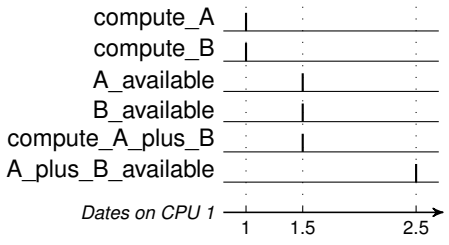

Fig. 3. Simulation of concurrent processes.

We find this solution because the TESL solver is designed for simulation and therefore makes time advance as fast as possible in each connected component of the tag relation graph.

[3] http://wwwdi.supelec.fr/software/ModHelX/

We call such connected components *time islands* because time can advance independently in each time island. When building the first instant of this example, there is a tick with tag 1 in the time island of CPU 1, and a tick with tag 2 in the time island of CPU 2. Since no tag relation allows to order these tags, both ticks can be put in the first instant. The default policy for a clock is to be *greedy*, which means that a given tick is put in an instant as soon as possible. Therefore in our example we put both ticks in the first instant, and thus we make time advance in both islands. The time delayed implications create ticks at dates 1.5 on CPU 1 and 3.5 on CPU 2. So the next instant contains these ticks and makes time advance to 1.5 in the island of CPU 1 and to 3.5 in the island of CPU 2. The await implication creates a tick on `compute_A_plus_B` at the same instant, and the last time delayed implication creates a tick on `A_plus_B_available` at date 2.5 in CPU 1 time. Therefore, the third instant contains this tick and makes time advance to 2.5 in the CPU 1 time island. No tag relation allows us to compute the current time in the CPU 2 time island, but this is not an issue since nothing happens at this time in this island. TESL also has support for *non greedy* clocks whose ticks are put in an instant only when necessary, but this is out of the scope of the article. This example shows how TESL builds a deterministic execution of a model which contains unrelated time scales by always putting the maximum number of ticks in a instant according to the relations between clocks.

F. Solving TESL Specifications: Problem Statement

Solving a TESL specification therefore consists in computing a series of instants which contains the maximum number of ticks on greedy clocks while satisfying the implication relations and the tag relations between clocks. Moreover, all ticks on a clock must be assigned to an instant: since a tick represents the occurrence of an event at a certain point in time, it is forbidden to discard a tick on a clock without making it "present" in an instant.

IV. Solving Algorithm

The solving algorithm for TESL specifications computes each instant as a fixed point of the implication relations and the tag relations. The fixed point is computed by iterating the addition of ticks to the instant until no tick can be added. Since there can be at most one tick on each clock in a given instant, and no tick can be removed from an instant once it has been added, the iteration is guaranteed to terminate because each step is a monotonous function in the number of ticks in the instant, which is bounded by the number of clocks. However, as we will see later, inconsistencies in the specification may prevent the existence of a fixed point. The deterministic and constructive (new facts are only built from known facts, without making hypotheses) nature of TESL guarantees the unicity of the fixed point if it exists, and allows us to compute it in polynomial time, without the need to backtrack in order to explore several possibilities. After the computation of each instant, the ticks of this instant are removed from the clocks, which amounts to moving to the strict future of this instant, and the next instant is computed. A simulation stops when all clocks in the specification have no tick (the future is empty), or, in the case of infinite behaviors, when some condition (maximum number of steps or the occurrence of a tick on a given clock) is met.

A. Implications

Implication relations are Mealy state machines which react to the presence of ticks in an instant by updating their state, which is preserved between instants, and by adding ticks to the current instant in reaction to some of their transitions. Because of the iterative nature of the computation of the fixed point, care must be taken when making these machines react so that they receive a consistent set of inputs. For instance, a machine which computes a delay by counting the ticks on a clock should count a tick only once in an instant, even if several iterations are needed to compute this instant. The first step in the computation of an instant is to make the implication relations react to the ticks that are already in the instant (considered as inputs) until no new tick is added to the instant.

B. Determination of the Tags

After applying the implication relations, the known tags of the ticks in the instant and the tag relations are used to compute the possibly missing tags of ticks in the instant, and to determine the current time on the clocks (the tag that a tick would have on this clock in this instant). Adding a tick to the current instant of a greedy clock can make time advance on other clocks, and we must choose which greedy clock should tick. For this, in each time island, we consider all greedy clocks which have ticks, and we compute a matrix T where T_{ij} contains the time it would be on clock j at this instant if the smallest tick on clock i were put in the instant. This value is obtained through the transitive closure of the tag relations. The T_{ii} element of the diagonal is the tag of the first tick of clock i in the island. We want to make time advance on as many clocks as possible, but by the least positive amount as possible so that the current time on each clock is kept less or equal to the tag of its earliest tick. Therefore, we look for a row k in T such that $\forall i, j \quad T_{kj} \leq T_{ij}$. If such a row exists, we add the first tick of clock k to the instant. If there is no such row, this means that at least one tag relation is decreasing and that time flows backwards on a clock. This is an inconsistent specification and the algorithm stops with an error.

Once the tags of the ticks have been computed and that ticks that have the same tag modulo the tag relations have been synchronized, if new ticks have been added to the current instant, the algorithm starts a new iteration, applying the implication relations again and then computing the tags and making time advance in the different time islands if necessary. It stops when no new tick has been added to the instant in the iteration, which means that the fixed point has been reached.

A last check is performed to verify that each tick in the computed fixed point has a known tag. If not, the model is underspecified because some information is missing to compute the current date on some clocks.

V. Complete Examples

A. Determining the Date of Easter

Determining the date of Easter is complex. A simplified model was used to demonstrate the capabilities of CCSL [19]. We use a similar model to implement the problem in TESL, and we compare both approaches.

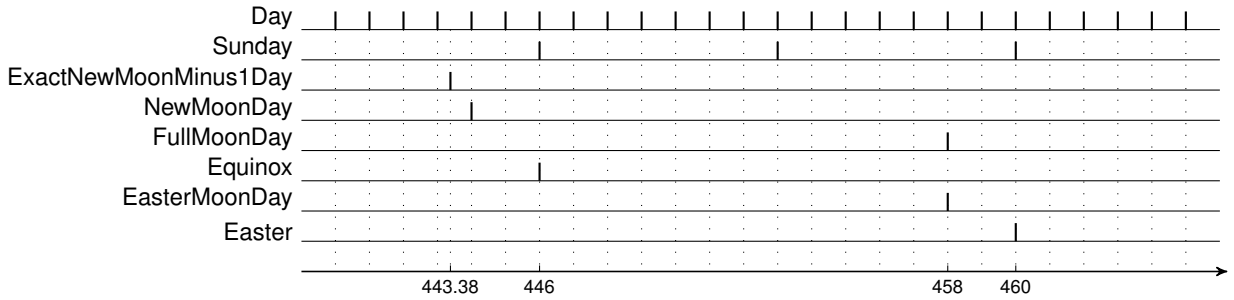

Fig. 4. Timing diagram depicting the computation of the date of Easter for 2015.

First we need to keep track of Sundays. We set the origin date to be January 1st, 2014 and we express tags in terms of days. We define:

```
rational-clock Day periodic 1 offset 1
// The first Sunday is on January 5, 2014
rational-clock Sunday periodic 7 offset 5
tag relation Sunday = Day
```

Easter is the Sunday just after the full moon that follows March 21st, the ecclesiastic equinox. The period of the moon is a non-integral number of days: 29.53059 days. Full moon is 14 days after new moon. To determine the day of the new moon, we need to sample the non-integral moon period onto days. To achieve this, we determine the exact time of the new moon *minus one day* and we use the *next to* implication:

```
// The first new moon of 2014 is on 1 Jan 2014
// at 10:15 UTC, which is day 1.42. One day
// before is day 0.42:
rational-clock ExactNewMoonMinus1Day
               periodic 29.53059 offset .42
tag relation Day = ExactNewMoonMinus1Day

unit-clock NewMoonDay
Day next to ExactNewMoonMinus1Day
    implies NewMoonDay

unit-clock FullMoonDay
NewMoonDay delayed by 14 on Day
           implies FullMoonDay
```

NewMoonDay ticks each day that sees a new moon. Despite the period being non-integral, each of the above clocks only ticks once per lunar cycle. In contrast, CCSL needs an extremely fine clock to account for the non-integral factor. [19] approximates the period to 29.53, therefore the authors had to create a clock called *"hundredth of a day"* that ticks 100 times a day.

Defining the equinox is straightforward (March 21st is day 81 in 2014), and the Sunday after the first full moon that follows the equinox is Easter:

```
rational-clock Equinox periodic 365 offset 81
tag relation Equinox = Day

unit-clock EasterMoonDay
FullMoonDay next to Equinox
            implies EasterMoonDay

unit-clock Easter
Sunday next to EasterMoonDay implies Easter
```

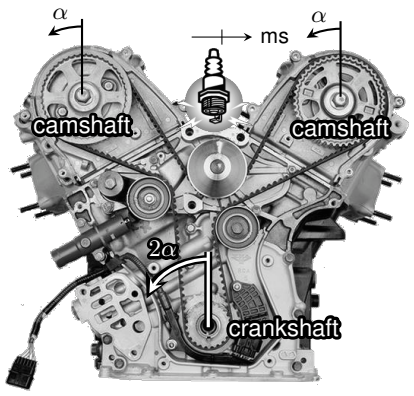

Fig. 5. Timing elements of a 4-stroke petrol engine

This model works correctly for 2014 and 2015. After that, we have to take leap years into account, which can be done in TESL (see the TESL web page). This specification can compute the answer in as few steps as there are days between the beginning of 2014 and the computed date (110 for Easter 2014, on April 20, 460 for Easter 2015, on April 5), as illustrated on Figure 4.

B. Multiform time

We consider a model of the ignition system of a four-stroke petrol engine as shown on Figure 5. The state of the engine can be measured by the angular position of its crankshaft. Because each cylinder produces thrust only every other revolution, the camshafts that drive the valves and the ignition turn twice as slow as the crankshaft, so the "time" scale of the camshafts is an angular position which changes twice as slow as the angular position of the crankshaft. The time at which the ignition spark must be produced depends on the ignition delay of the air-fuel mixture, which is measured on physical time. The relation between this delay and the angular "time" on the camshaft depends on the rotation speed of the engine. The following TESL model shows how multiform time and tag relations can be used to compute the angular ignition advance from the ignition delay:

```
rational-clock realtime    // in seconds
rational-clock crankshaft  // in degrees
rational-clock camshaft    // in degrees
// crankshaft turns twice as fast as camshaft
tag relation crankshaft = 2 * camshaft + 0

let int rpm = 2000      // in turns/minute
```

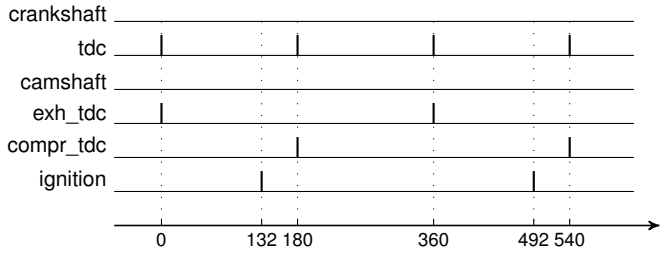

Fig. 6. Angular ignition advance on the camshaft time scale

```
let rational degrees_per_sec =
    [rational $rpm] * 360 / 60
tag relation crankshaft =
    $degrees_per_sec * realtime + 0
// Make the engine run
rational-clock tdc periodic 1 offset 0
// crankshaft advances by 360 degrees each TDC
tag relation crankshaft = 360 * tdc + 0
// Clocks for exhaust and compression TDCs
unit-clock exh_tdc
tdc every 2 implies exh_tdc
unit-clock compr_tdc
tdc every 2 starting at 1 implies compr_tdc
// Spark triggered 8ms before TDC
let decimal ignition_advance = 8E-3
// Convert advance into delay after TDC
let rational period = [rational 60] / $rpm
let rational ignition_delay =
    $period - $ignition_advance
// Trigger ignition
unit-clock ignition
exh_tdc time delayed by $ignition_delay
            on realtime implies ignition
```

In the above code, the notation `[rational $rpm]` is the type conversion of the integer value `$rpm` into a rational, and the **let** construct is used to define named constants. The result, shown on Figure 6, shows that at 2000 rpms, the ignition should be triggered 48° on the camshaft before each compression top dead center (132° after the exhaust top dead center) to have an ignition advance of 8 ms.

C. Timed Finite State Machines

Finite state machines (FSM) are commonly used to model controllers. In this section, we explain how to build a TESL specification that encodes a given FSM systematically. We consider FSMs with *timed* transitions, which are taken when some delay has elapsed since entering the current state.

Input events as well as events produced by output actions are modeled as clocks. To be able to determine at any time which transitions can fire, we need to keep track of the current state. Therefore, we associate a clock called `in_S` to every state S. `in_S` must tick at each instant after entering state S and until leaving state S. To systematize the design of the TESL specification, we associate two new clocks to state S: `enter_S`, which ticks when a transition *to* state S is fired, and `leave_S`, which ticks when a transition *from* state S is fired. Therefore we can have the following definitions:

```
// all events imply this clock
unit-clock allEvents
// State S:
unit-clock in_S
unit-clock enter_S
unit-clock leave_S
allEvents sustained from enter_S to leave_S
          implies in_S
```

When an event occurs that matches the guard of a transition leaving the current state, this transition fires, therefore several clocks must tick: `leave_S` (resp. `enter_S'`), to indicate that we are leaving the current state S (resp. entering a new state S'), and any action clock associated with the transition.

For example, let us consider the state machine of Figure 7 that models a very simple network protocol: when in the Ready state, the machine sends a reply upon receiving a request. Then it waits for an ack during 10 seconds. After that delay it timeouts and goes back to the Ready state.

The non-timed transitions from Ready to Waiting, and from Waiting to Ready are specified as:

```
unit-clock event_request
event_request when in_Ready
              implies enter_Waiting
event_request when in_Ready
              implies leave_Ready
event_request when in_Ready
              implies action_reply
event_request implies allEvents

unit-clock event_ack
event_ack when in_Waiting implies enter_Ready
event_ack when in_Waiting implies leave_Waiting
event_ack implies allEvents
```

To specify a timed transition, first we create an event corresponding to the expiration of the delay. With a *time delayed* implication, we imply a tick on the delay-expiration-event clock a given amount of time after entering the state (the **with reset** ensures that the timeout event will not be generated if we leave the state before the delay has elapsed):

```
unit-clock timeout_W_to_R
enter_Waiting time delayed by 10 on chrono
              with reset on leave_Waiting
              implies timeout_W_to_R
timeout_W_to_R implies allEvents
```

Here we use `chrono`, a clock used only to express the duration of delays, and defined as **rational-clock** `chrono`. There must be a tag relation between it and the clock that generates the events (a scenario player, or a real-time clock). The timeout event is used in the same manner as any event to define the transition:

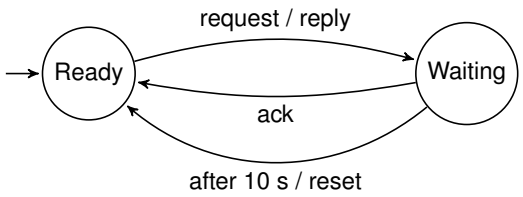

Fig. 7. A finite state machine modeling a very simple network protocol.

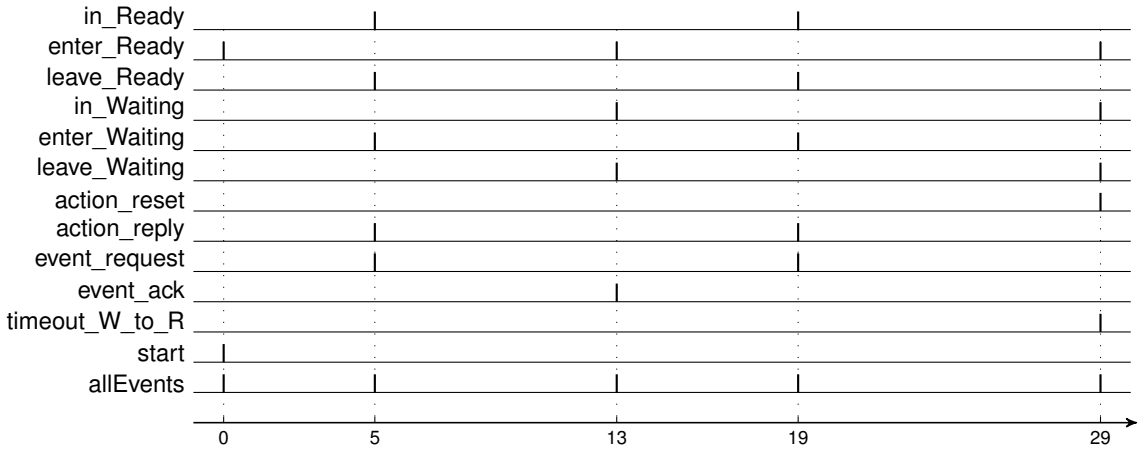

Fig. 8. Timing diagram depicting the behavior of the clocks associated with the state machine of Figure 7, with the following input scenario: request at $t = 5$, ack at $t = 13$ and request at $t = 19$.

```
timeout_W_to_R when in_Waiting
               implies enter_Ready
timeout_W_to_R when in_Waiting
               implies leave_Waiting
timeout_W_to_R when in_Waiting
               implies action_reset
```

Finally we need to generate an `enter_Ready` event at time 0 to put the FSM in the initial state `Ready` at the start:

```
rational-clock start sporadic 0
start implies enter_Ready
```

Let us consider the timing diagram of Figure 8. The FSM is in the Ready state initially and up to $t = 5$, at which point it receives a request. The `in_Ready` and `event_request` clocks tick, which fires the transition. The `action_reply` clock ticks, and so do the `leave_Ready` and `enter_Waiting` clocks. `in_Waiting` does not tick at this point, but only when the next event is received: at $t = 13$, `in_Waiting` and `event_ack` tick, so the transition back to Ready is taken.

We note that the clocks tick only when an event (input event or timeout) occurs. This is to be contrasted with the way of specifying an FSM in CCSL where all events have to be sampled on some clock, including the timed transitions. Therefore this clock in general has a relatively small period, for instance 1 ms. This means that in CCSL when we are in a given state S, `in_S` must tick every millisecond, even when nothing happens [12].

VI. Reconciliation of Execution Traces

When designing a system, different components and different aspects may call to different modeling paradigms. Each paradigm can be expressed using a Domain Specific Modeling Language (DSL). The approach used in the GEMOC[4] initiative [20] for defining the semantics of DSLs is to segregate the actions on the runtime model into Domain Specific Actions (DSAs) and to describe the control using Domain Specific Events (DSEs) (see [21] for details). For instance, when considering state machines, evaluating the guard of a transition, firing a transition and changing the current state would be DSAs which would be triggered by respective DSEs. The constraints on the occurrences of DSEs are defined by a Model of Computation (MoC) which gives the semantics of control (including concurrency) and time for a DSL. In addition to decoupling the control and time from the computations needed to update the state of a model, this way of defining DSLs provides an abstract interface to the execution trace of a model as a sequence of DSE occurrences.

We can rely on this interface to specify how heterogeneous components of a model interact to provide the global behavior of the model. By using clocks for modeling the DSEs provided by a DSL at the interface of the components of a model, we can specify relations between these clocks to describe how the execution trace of each component is reconciled with the execution traces of the other components. For instance, a data-flow language may have a DSE for the activation of a network of operators, and a periodic activation pattern of the network can be specified by making the corresponding clock periodic. Implication relations and sampling (using the **`next to`** operator) can also be used to specify when input data is provided to a component and when its outputs are taken into account. Exemples of such reconciliation patterns are given in [2]. This approach is currently being investigated in the ANR INS GEMOC project for combining the behavior of an arbitrary number of heterogeneous components, and it is already in use in the ModHel'X heterogeneous modeling and simulation framework for defining the hierarchical composition of pairs of heterogeneous components with TESL [22], [23]. The description of the semantic adaptation of time and control between heterogeneous components using relations between the clocks that are associated to DSEs has several advantages. It provides a uniform way of defining the composition operators which can be used to assemble components into a system. It also makes this definition explicit in the model, so that it can be taken into account in a consistent way by different tools such as a simulator, a code generator or an analysis tool. A limitation of this approach is that it does not model the transformation of data at the boundary between heterogeneous models, which has to be wrapped into DSAs. Another limitation is that it requires

[4]http://gemoc.org

that all the DSLs used in the model be defined according to this approach in order to provide an interface in terms of DSEs. However, when the control and timing part (the MoC) of every DSL is specified as relations between the clocks of its DSEs, and the semantic adaptation between heterogeneous components is also described in the same way, it becomes possible to extract a homogeneous description of the control and timing of a whole heterogeneous model, which opens perspectives to the precise definition of its behavior and to formal verification.

By modeling the execution environment of a model using clocks, it is also possible to specify how a model is executed. For instance, ModHel'X uses *driving clocks* which can be linked to the system clock of the computer or to clicks on a button in a GUI. By specifying relations between these driving clocks and the clocks of the model, we can describe precisely how the model should run with respect to its environment without hiding control inside opaque components of the model (for instance, components which wait for a delay on the system clock or implement a callback for a GUI button). By binding the driving clocks to events of the execution platform, we could reuse the same specification to describe how generated code would run on its target platform. From this point of view, controlling the execution of a model amounts to reconciling its execution trace with the trace of its environment, in a similar way to the binding of model time to real time discussed in [24].

VII. Semantics

The semantics of the purely synchronous part of TESL can be defined by translation to the Esterel language and relying on the constructive semantics of Esterel as described in [25]. It corresponds to the fixed point of the implication relations computed by the solving algorithm presented in section IV. The main difficulty in the formalization of the semantics of TESL is the processing of tag relations and the choice of the ticks that occur at the current instant in each time island.

A possible approach for TESL models where tag relations are static (the relation between the time scales of any two clocks does not depend on time) would be to translate each time island into a timed automaton [5], and to consider the whole specification as the synchronous product of the timed automaton of each time island. However, in a timed automaton, all clocks run at the same pace while in TESL, time may not advance on one clock until it has advanced by a given amount on another clock. Consider for example the following model:

```
int-clock a sporadic 0, 1, 2
int-clock b
tag relation a = 2 * b + 0
a implies b
```

time will not progress on clock b when time goes from 0 to 1 on a, and the first two occurrences of b will happen at time 0 on its time scale. This is because tag relations are bidirectional mappings, so `a = 2 * b + 0` means that a tag τ_a on a and a tag τ_b on b belong to the same instant if either $\tau_a = 2\tau_b$ or $\tau_b = \tau_a \div 2$. With integer clocks, these mappings are not bijective, and the mapping from a to b is not the reverse of the mapping from b to a. In the above model, there are three instants which contain occurrences of both a and b. On the time scale of a, they have tags 0, 1 and 2, but on the time scale of b, they have tags 0, 0 and 1, so the first two instants happen to have the same date on this time scale. It may however be possible to encode such a model into a timed automaton by resetting the b clock when it should not advance.

A second issue is to define a synchronous product on timed automata which lets time flow independently in each automaton while synchronizing event occurrences that are causally linked by an implication relation. The theory of tag structures, tag morphisms and fibered products of tag morphisms presented in [16], and used for simulation in [26] seems to be a suitable foundation for the formal specification of these issues. We currently have a long term project for defining the semantics of TESL in Isabelle/HOL [27], [28] in order to be able to prove properties of TESL models and to generate test cases using the HOL-TestGen framework [29].

VIII. Discussion

The model of time of TESL that we have just presented has several advantages over CCSL for the execution of heterogeneous models. Using tag relations and the specification of event occurrences anywhere on the time scale of a clock, it allows for an efficient specification of delays and of the synchronization of events on different time scales, as well as taking into account input events produced by the environment during a simulation. Similarly to CCSL, it does not assume the existence of a global root clock and allows time to advance independently on different clocks as long as relations between clocks are respected. TESL is deterministic because it does not support the asynchronous operators of CCSL, and constructive (each instant is built only by accumulating known facts about clocks). Its solver therefore runs in polynomial time, and the computations on tags allow it to compute delays starting at arbitrary times and without counting numerous event occurrences.

However, the deterministic nature of TESL is a drawback for the specification of behaviors at a high level of abstraction. It is not possible to use TESL for specifying a set of allowed behaviors and verifying properties that hold for all of them. These differences are the result of different design choices: CCSL was designed in the context of MARTE, with a focus on the verification of real-time systems, while TESL was designed in the context of the execution of heterogeneous models in a deterministic simulation environment. An interesting challenge would be to verify that the behavior of a TESL specification matches one of the behaviors allowed by a CCSL specification, which would allow us to consider TESL as a practical implementation language for CCSL models. This may be a byproduct of the formalization of the semantics of TESL.

IX. Perspectives

Although its semantics is currently captured by a natural-language description and a reference implementation, TESL allows the explicit reconciliation of the traces of execution of heterogenous behaviors. To be able to check properties on such composed behaviors, we plan to define this semantics in a formal tool such as Isabelle.

Currently, a TESL specification written in the concrete syntax describes *an instantiation of a problem*. For instance, we described in Section V-C an algorithm for generating a TESL specification for any timed finite state machine (FSM). We cannot express the generic structure of an FSM in TESL:

we had to write a Python script to implement the translation algorithm. A useful extension of the TESL language would be to add a syntax for expressing *patterns*, for instance a pattern for generating a set of TESL statements for any state of an FSM. With such a templating feature, it would no longer be necessary to resort to a translator written in another language. This would also allow us to define semantic adaptation patterns for reconciling execution traces that would generate the suitable TESL clocks and relations when instantiated in a model.

As mentioned before, we already use the TESL library to execute models in ModHel'X, a heterogeneous modeling platform. This involves adapting control and time between different models of computation. We plan to use it as well to describe the models of computation themselves. In this paper, we have shown that TESL can capture the semantics of timed FSMs: we will extend this to other models of computation, such as synchronous data flows. This has already been done using CCSL [30] so it should not be difficult. With both models of computations and semantic adaptation between heterogeneous components of a model described in TESL, we would have a uniform description of all control and timings aspects of a model, which would help us to verify its behavior.

X. Conclusion

TESL is a language for modeling (a) causality relations between discrete events (implications), and (b) arithmetic relations between time scales (tag relations). Therefore it supports both (a) event-triggered and (b) time-triggered behavior. A TESL model may be solved deterministically in polynomial time. Compared to similar approaches such as CCSL, tag relations allow us to calculate dates without the need for creating as many ticks as *possible* instants: we just consider *needed*, *meaningful* instants. Moreover, the constructive nature of the solving algorithm allows us to introduce ticks at run-time, in-between solving steps, namely in reaction to data coming from the environment, for instance a numerical solver, or an input device. This allows us to use TESL for powering the model execution platform ModHel'X. This tool also relies on TESL to specify how the trace of input events from sensors, the traces of the simulator events, and the trace of commands sent to actuators are reconciled when running a software simulation with hardware in the loop. The use of TESL makes this reconciliation explicit and avoids the use of glue code between the simulator and its environment.

References

[1] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling Time in Computing*, ser. Monographs in Th. Comp. Science. Springer, 2012.

[2] F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet, "Semantic adaptation for models of computations," in *Proceedings of the 11th International Conference on Application of Concurrency to System Design*. IEEE Computer Society, 2011, pp. 153–162.

[3] J. DeAntoni and F. Mallet, "TimeSquare: Treat your models with logical time," in *TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012*, ser. LNCS, vol. 7304. Springer, 2012, pp. 34–41.

[4] F. Mallet, "CCSL: specifying clock constraints with UML/MARTE," *Innovations in Systems and Soft. Eng.*, vol. 4, no. 3, pp. 309–314, 2008.

[5] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[6] B. P. Zeigler, T. G. Kim, and H. Praehofer, *Theory of Modeling and Simulation*, 2nd ed. Orlando, FL, USA: Academic Press, Inc., 2000.

[7] E. Syriani, J. Gray, and H. Vangheluwe, "Modeling a model transformation language," in *Domain Engineering*. Springer, 2013, pp. 211–237.

[8] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier, "First experiments using the UML profile for MARTE," in *11th IEEE Int. Symp. on Object Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 50–57.

[9] W. Godard, M.-L. Valentin, P. Kortmann, and M. Gerhardt, "Analysis of real-time systems scheduling using MARTE," NATO, Tech. Rep. STO-MP-IST-115, 2013.

[10] V. Brocal, M. Masmano, I. Ripoll, A. Crespo, P. Balbastre, and J.-J. Metge, "Xoncrete: a scheduling tool for partitioned real-time systems," in *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS2 2010)*, 2010.

[11] F. Mallet, "Clock constraint specification language: specifying clock constraints with UML/MARTE," *Innovations in Systems and Soft. Eng.*, vol. 4, no. 3, pp. 309–314, 2008.

[12] F. Boulanger, A. Dogui, C. Hardebolle, C. Jacquet, D. Marcadet, and I. Prodan, "Semantic adaptation using CCSL clock constraints," in *Post-proceedings of MODELS 2011 Workshops*, ser. LNCS, vol. 7167. Springer-Verlag, 2012, pp. 104–118.

[13] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity - the Ptolemy approach," in *Proceedings of the IEEE*, 2003, pp. 127–144.

[14] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The Synchronous Languages 12 Years Later," *Proc. of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[15] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Trans. CAD*, vol. 17, no. 12, 1998.

[16] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli, "Composing heterogeneous reactive systems," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 4, pp. 43:1–43:36, 2008.

[17] Z. Manna and A. Pnueli, "Verifying hybrid systems," in *Hybrid Systems*. Springer, 1993, pp. 4–35.

[18] C. Hardebolle and F. Boulanger, "Exploring multi-paradigm modeling techniques," *SIMULATION: Transactions of The Society for Modeling and Simulation International*, vol. 85, no. 11/12, pp. 688–708, 2009.

[19] C. André and F. Mallet, "Clock constraints in UML/MARTE CCSL," INRIA, Research Report RR-6540, 2008.

[20] B. Combemale, J. DeAntoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray, "Globalizing modeling languages," *Computer*, vol. 47, no. 6, pp. 68–71, June 2014.

[21] B. Combemale, J. DeAntoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France, "Reifying concurrency for executable metamodeling," in *6th International Conference on Software Language Engineering*, ser. LNCS, vol. 8225. Springer, 2013, pp. 365–384.

[22] F. Boulanger, C. Jacquet, C. Hardebolle, and A. Dogui, "Heterogeneous Model Composition in ModHel'X: the Power Window Case Study," in *Proceedings of Gemoc 2013, Workshop on the Globalization of Modeling Languages*, Sep. 2013, 10 pages.

[23] B. Meyers, J. Denil, F. Boulanger, C. Hardebolle, C. Jacquet, and H. Vangheluwe, "A DSL for Explicit Semantic Adaptation," in *Proceedings of MPM 2013*, Sep. 2013, pp. 47–56.

[24] Y. Zhao, J. Liu, and E. Lee, "A programming model for time-synchronized distributed real-time systems," in *Real Time and Embedded Technology and Applications Symposium, 2007*, 2007, pp. 259–268.

[25] G. Berry, "The Constructive Semantics of Pure Esterel," 1996.

[26] T. T. H. Le, R. Passerone, U. Fahrenberg, and A. Legay, "Tag machines for modeling heterogeneous systems," in *Application of Concurrency to System Design (ACSD 2013)*, July 2013, pp. 186–195.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[28] M. Wenzel, L. C. Paulson, and T. Nipkow, "The Isabelle Framework," in *Theorem Proving in Higher Order Logics*. Springer, 2008, pp. 33–38.

[29] A. D. Brucker and B. Wolff, "On theorem prover-based testing," *Formal Aspects of Computing*, vol. 25, no. 5, pp. 683–721, 2013.

[30] F. Mallet, J. DeAntoni, C. André, and R. de Simone, "The clock constraint specification language for building timed causality models," *Innovations in Systems and Soft. Eng.*, vol. 6, no. 1-2, pp. 99–106, 2010.