# Formalizing Time and Causality in Polychronous Polytimed Models

Hai Nguyen Van

université PARIS-SACLAY

NNT : 2018SACLS282

Thèse de doctorat

CentraleSupélec | UNIVERSITÉ PARIS SUD

# Formalizing Time and Causality in Polychronous Polytimed Models

Thèse de doctorat de l'Université Paris-Saclay
préparée à l'Université Paris-Sud

École doctorale n°580
Sciences et Technologies de l'Information et de la Communication

Spécialité Informatique

en vue de l'obtention du grade de

**Docteur en Informatique**

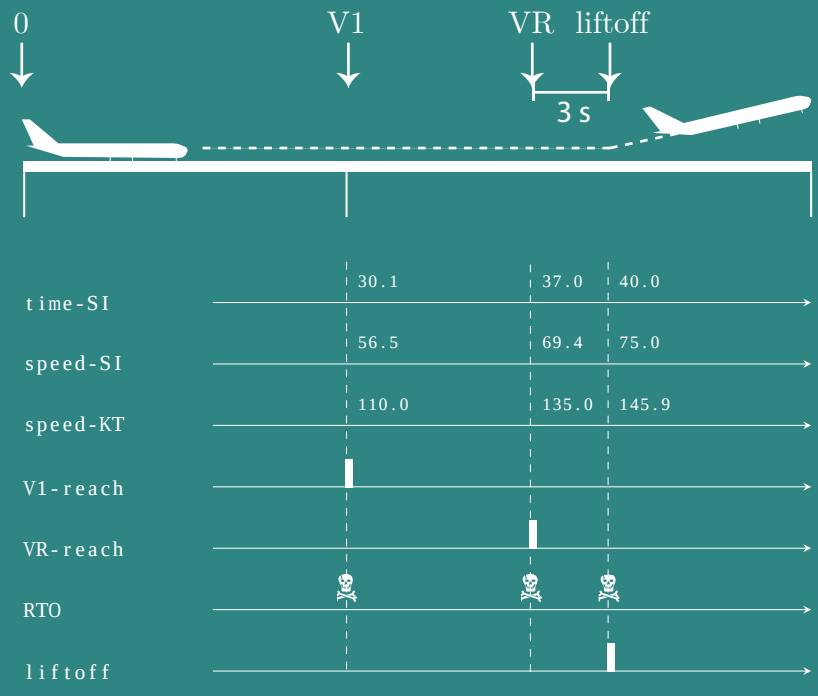
Thèse présentée et soutenue à Orsay le 27 septembre 2018, par


## Hai Nguyen Van


Composition du jury

| | |
|---|---|
| **Catherine Dubois**<br>Professeur, ENSIIE | Présidente |
| **Frédéric Mallet**<br>Professeur, Université Nice Sophia Antipolis | Rapporteur |
| **Stephan Merz**<br>Directeur de recherche, Inria Nancy | Rapporteur |
| **Timothy Bourke**<br>Chargé de recherche, Inria Paris | Examinateur |
| **Marc Pantel**<br>Maître de conférences, Université de Toulouse | Examinateur |
| **Mihaela Sighireanu**<br>Maître de conférences HDR, Université Paris Diderot | Examinatrice |
| **Frédéric Boulanger**<br>Professeur, CentraleSupélec | Directeur |
| **Burkhart Wolff**<br>Professeur, Université Paris-Sud | Directeur |

# Formalizing Time and Causality in Polychronous Polytimed Models




Hai Nguyen Van

# FORMALIZING TIME AND CAUSALITY IN POLYCHRONOUS POLYTIMED MODELS

DOCTORAL THESIS

Hai Nguyen Van

A thesis submitted in fulfillment of the requirements
for the degree of

**Doctor of Philosophy in Computer Science**



Committee in charge

| | |
|---|---|
| Frédéric Mallet | Referee |
| Stephan Merz | Referee |
| Catherine Dubois | Examiner |
| Timothy Bourke | Examiner |
| Marc Pantel | Examiner |
| Mihaela Sighireanu | Examiner |
| Frédéric Boulanger | Supervisor |
| Burkhart Wolff | Supervisor |

September 2018

This document was typeset using LaTeX and inspired from the typographical style `classicthesis` by André Miede and Ivo Pletikoić.

Hai Nguyen Van: *Formalizing Time and Causality in Polychronous Polytimed Models,* Doctoral Thesis, © September 2018

Ông, bà, mẹ thân yêu của con.

## ABSTRACT

Integrating components into systems turns out to be difficult when these components were designed according to different paradigms or when they rely on different time frames which must be synchronized. This synchronization may be event-driven (an event occurs because another event occurs) or time-driven (an event occurs because it is time for it to occur). Considering that each component admits its own time frame, and that they may not be related, a unique global time line may not exist.

We are interested in specifying synchronization patterns for such polychronous and polytimed systems. Our study had led us to design semantic models for a timed discrete-event language, called the TESL language developed by Boulanger et al. This language has been used for coordinating the simulation of composite models and testing system integration.

In this thesis, we present a denotational semantics providing an accurate and logic-consistent understanding of the language. Then we propose an operational semantics to derive satisfying runs from TESL specifications. It has been used for testing purposes, through the implementation of a solver, named Heron. To tackle the issue of the consistency and correctness of these semantic rules, we developed a co-inductive intermediate semantics that relates both the denotational and the operational semantics. Then we establish properties over the relation of our semantic models: soundness, completeness and progress, as well as local termination. Finally, our formalization and these proofs have been fully mechanized in the Isabelle/HOL proof assistant.

## RÉSUMÉ

L'intégration de composants dans un système peut s'avérer difficile lorsque ces composants ont été conçus selon différents paradigmes ou s'ils se basent sur différents cadres de temps devant être synchronisés. Cette synchronisation peut être dirigée par les évènements (un évènement est provoqué par un autre), ou dirigée par le temps (un évènement se produit parce qu'il en est l'heure). En considérant que chaque composant admet son propre cadre de temps et qu'ils peuvent ne pas être reliés, il est possible qu'une unique ligne de temps globale n'existe pas.

Nous nous intéressons à la spécification de schémas de synchronisation pour de tels systèmes polychrones et polytemporisés. Notre étude nous a mené à la conception de modèles sémantiques pour un langage temporisé à évènements discrets, appelé TESL et développé par Boulanger et al. Ce langage a été utilisé pour coordonner la simulation de modèles composites et pour tester l'intégration de systèmes.

Dans cette thèse, nous présentons une sémantique dénotationnelle fournissant une compréhension précise et logiquement cohérente du langage. Puis nous proposons une sémantique opérationnelle afin de dériver des traces d'exécutions satisfaisant une spécification TESL. Celui-ci a été utilisé pour les problématiques de test des systèmes, à travers l'implantation d'un solveur nommé Heron. Pour résoudre la question de cohérence et de correction de ces règles sémantiques, nous avons également développé une sémantique intermédiaire coinductive reliant les deux sémantiques dénotationnelles et opérationnelles. Nous établissons des propriétés sur la relation entre les deux sémantiques : correction, complétude, progrès ainsi que terminaison locale. Enfin, notre formalisation ainsi que les preuves associées ont été entièrement mécanisées dans l'assistant de preuve Isabelle/HOL.

# TÓM TẮT NỘI DUNG

Tổng hợp các bộ phận thành hệ thống tỏ ra là một vấn đề khó khi các bộ phận được thiết kế bằng các phương pháp hệ khác nhau hay là khi chúng hoạt động trong các khung thời gian khác nhau được đồng bộ hóa. Sự đồng bộ hoá này có thể do sự kiện dẫn định (một sự kiện xảy ra là do một sự kiện khác xảy ra), hay sự đồng bộ hoá này là do thời gian dẫn định (một sự kiện xảy ra là do đến lúc nó phải xảy ra). Trong trường hợp các bộ phận có khung thời gian riêng mà lại không liên đới với nhau, thời gian toàn cục duy nhất có thể không tồn tại.

Chúng tôi quan tâm đến vấn đề định rõ dạng mẫu đồng bộ hoá cho các hệ thống đa bộ (polychronous) và đa thời hoá (polytimed). Nghiên cứu của chúng tôi đưa đến việc thiết kế các mô hình ngữ nghĩa cho một ngôn ngữ thời hoá (timed) có các sự kiện rời rạc (discrete event). Ngôn ngữ này được gọi là TESL do Boulanger và đồng nghiệp thiết kế nên. Ngôn ngữ này đã được dùng để điều phối việc mô phỏng các mô hình hỗn ghép và để kiểm nghiệm (testing) sự tổng hợp hệ thống.

Trong luận án này chúng tôi trình bày một ngữ nghĩa biểu thị (denotational semantics) nhằm hiểu ngôn ngữ này chính xác và mạch lạc về mặt logic. Sau đó, chúng tôi đề xuất một ngữ nghĩa tác vụ (operational semantics) nhằm suy ra cách vận hành thoả mãn các đặc tính (specification) diễn đạt bằng ngôn ngữ TESL. Ngữ nghĩa tác vụ này đã được dùng cho mục đích kiểm nghiệm (testing) thông qua một công cụ giải (solver) do chúng tôi tạo ra, tên là Heron. Để đảm bảo tính mạch lạc và chuẩn xác của các qui tắc ngữ nghĩa (semantic rules) này, chúng tôi lập một ngữ nghĩa đồng-quy nạp trung gian (intermediate co-inductive semantics), cho phép liên kết được hai ngữ nghĩa biểu thị và tác vụ. Rồi chúng tôi xác minh các tính chất (property) của mối liên hệ giữa các mô hình ngữ nghĩa này: cụ thể là hợp lý (soundness), hoàn chỉnh (completeness), tiến triển (progress) và kết thúc (termination). Phương pháp hình thức hóa (formalization) mà chúng tôi đề xuất và các phép chứng minh đều được tự động hoá hoàn toàn trong công cụ hỗ trợ chứng minh Isabelle/HOL.

Many thanks to Thao Dang for correcting this translation.

# CONTENTS

# 1

# INTRODUCTION

*The software did exactly what it was told to do. In fact it did it perfectly. The reason it failed is that it was told to do the wrong thing.*

— Somers, J. (2017, Sept 26). The Coming Software Apocalypse. *The Atlantic*.

## 1.1 CONTEXT

Software is everywhere. In the past few years, the prevalence of software has altered the way industrial systems are now being designed. What used to be electromechanical has crucially become software, whether in an airplane or in a car. The ability of software to implement and automate complex tasks in a much more efficient way than mechanical and electrical devices [MSE04, MV04], has led to its widespread use in most complex systems. As portrayed in Figure 1, the volume of embedded software carried in Airbus aircrafts exhibits an exponential growth curve.

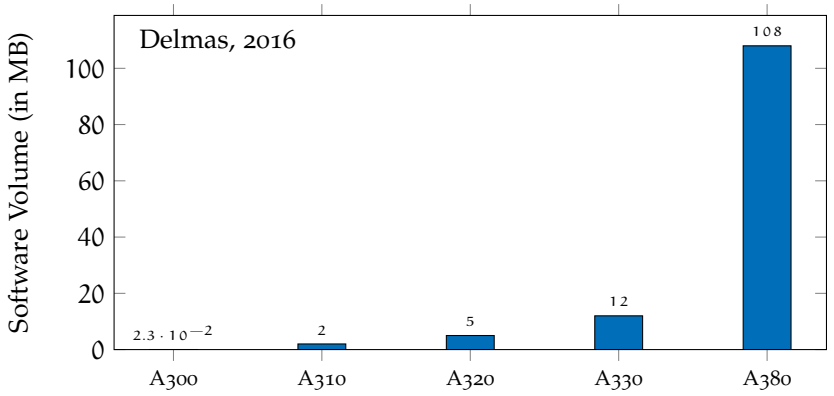



Figure 1: Volume of embedded software in Airbus transport-category aircrafts

However, designing software relies on human intelligence and confidence. Based on this assumption, software are naturally prone to errors and misunderstanding. In the best scenario, faults can eventually cause no harm or shall be prevented upstream. With a more complex system comes even greater risks of fault. With a more critical system comes a greater degree of gravity. To en-

sure safety of such systems which can involve human lives or large money investment, the demand for program verification has grown rapidly in the past decades. Especially when considering the complexity growth, it is unavoidable to automate verification procedures at the expense of manual analysts to ensure safety, security and maintainability of such large systems. These techniques are based on logic and mathematics: they are called *formal methods*.

Two complementary approaches intervene in this process: *modeling* and *verification*.

- On one side, the goal of modeling is to provide an interface for analysts to abstract systems or physical behaviors into simpler objects. The obvious outcome is the ability to conceptualize and classify real world problems to some extent. The very same way that a country map serves as a more understandable representation of geography and geometry. A *model* is nothing but a "map" of what we expect from the real world with respect to some degree of abstraction. It should not overapproximate, otherwise it will not demonstrate enough relevance.

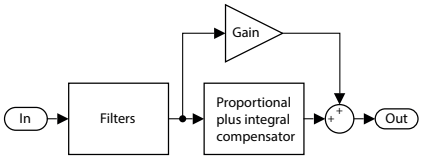
- On the other side, we are willing to use models to *verify and validate* required properties on the real-world system. We require that the associated semantics is understandable enough to assess and make decisions. In an approximation, we lose granularity but gain ability, and even automation of decisions and procedures. As opposed to a too low approximation that remains too complex to reason upon.

As a matter of fact, the game is all about looking for a *compromise* where the model is accurate enough to reflect real-world behaviors, while remaining simple enough to gain perspective and to assess properties.

Nonetheless, comparing physical systems to a country "map" would be too simplistic. A model itself is also a moving object, that can be executed in some cases to produce behaviors. This is where *simulation* opposes verification, and provides the ability to run the model as we run programs. Similarly to satisfiability in logical systems, simulation is done by means of a solver. Again, the game is about finding the best compromise to yield an accurate simulation, while keeping a very expressive modeling language.

In fact, models are not so different from programs, they can take inputs, then can be executed to produce an observable output. They serve as programs described in a more abstract shape, where programming details do not take part in the design process. In a relative degree of abstraction, they can even be translated into low-level programs. All the drawbacks related to programming details (bit-wise operations, pointers, registers...) do not take part in the process, and design engineers can produce code while designing models closer to physical systems. From this statement, the trend in industrial system modeling has evolved leading to the usage of domain-specific languages: this is the aim

of Model-Based Design (MBD). This sequence of transformations is exhibited by the model of a simple autopilot (using a Proportional–Integral–Derivative (PID) controller) in Figure 2a, then a generated program code in Figure 2b, then the compiled machine code in Figure 2c, and finally the hardware integration in Figure 2d.




(a) Controller Model

```
while(true) {
  error = desired_value - actual_value
  integral = integral + (err * iter_time)
  derivative = (err - err_prior) / iter_time
  output = KP*err + KI*integral + KD*derivative
  err_prior = err
  sleep (iter_time)
}
```

(b) Program

```
 97  0025 DD45        STD     AREGH
 98  0027 8601        LDAA    #$01
 99  0029 9749        STAA    ALUF
100  002B DC00        LDD     KPNUM
101  002D DD47        STD     BREGH
```

(c) Machine Code


(d) Hardware (a BendixKing autopilot)

Figure 2: From model to hardware

## 1.2 PROBLEM

Our problem is located around the design of complex systems, which involves different formalisms for modeling their different parts or aspects. The global model of a system may therefore consist of a coordination of concurrent 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 with their various interfaces, and specifying the various ways of collaboration and coordination.

### 1.2.1 *Architectural Composition of Systems*

The following diagram in Figure 3 of a heterogeneous system model may give an intuition over the application scenarios we have in mind, assuming different subsystems A, B and C described in different programming or specification formalisms, which are coordinated by some architectural glue in the center, having only access to the interfaces of the subsystems.

In order to tackle the heterogeneous nature of the subsystems, we abstract their behavior as *clocks*. Each clock models an event – something that can occur or not at a given time. From a system-centric perspective, an *event* corresponds to a change of some predicate over time-continuous variables, e.g., threshold

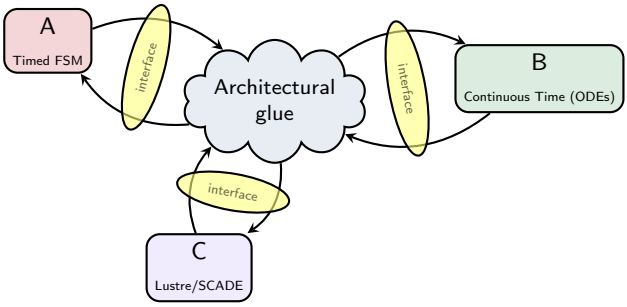



Figure 3: A heterogeneous timed system model

crossing. Then the time at which the event occurred is measured in a time frame associated with each clock, and the nature of time (integer, rational, real or any type with a linear order) is specific to each clock. When the event associated with a clock occurs, the clock *ticks*. In order to support any kind of behavior for the subsystems, we are only interested in specifying what we can *observe* at a series of discrete instants. There are two constraints on observations: a clock may tick only at an observation instant, and the time on any clock cannot decrease from an instant to the next one. However, it is always possible to add arbitrary observation instants, which allows for stuttering and modular composition of systems.

**Example 1** (The Car Power Window Model)**.** From another perspective of the issue of heterogeneous modeling in the real world, we cite the example of a power window system [BHJM11] as found in modern cars. The components of such a system are designed by people from different technical domains (electronics, mechanics, automation, control...). They are based on different languages and different paradigms: this is where *heterogeneity* comes into operation. This kind of system revolves around three main parts as shown in Figure 4 which can be modeled this way

1. The electromechanical window subsystem with synchronous dataflow graphs [LM87];
2. The controller with a timed finite state machine [AD94a];
3. The communication bus, which serves as a medium among system parts, with discrete events.

The abstraction we shape to connect heterogeneous parts is determined by the interface borders of each model. Each part works as a black box providing independent behaviors. Their relation and how they are interleaved is meant to become the behavior of the global system.

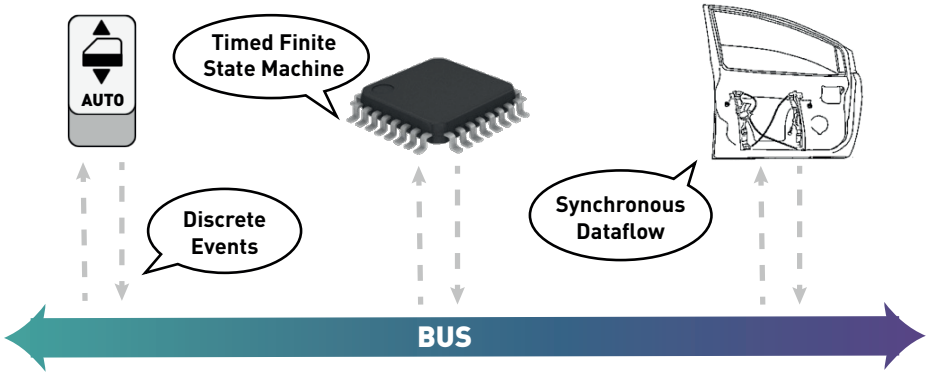



Figure 4: Interaction within heterogeneous parts in the power window case

### 1.2.2 *Independent Timeframes for Independent Models*

One of the key ingredient of heterogeneous modeling we also investigate is the ability for each independent model to live in a completely independent timeframe. These timeframes are not related unless specified. This belief of how time should be handled is not uncommon and can even also be required. In this approach, we want to eliminate the usual assumption of an absolute Newtonian time, as found in most timed formalisms including timed automata [AD94b]. Hence we consider specification languages with support for events that occur in independent time frames or in related time frames where time elapses according to some relation.

**Example 2** (Relativity between Systems)**.** An example of an application is the combined effects of relative speed (according to Special Relativity) and a weaker gravity (according to General Relativity) that make time run faster by 38 µs per day for a GPS satellite than for a stationary receiver on Earth.

The dates in seconds in their time frames are related by an affine relation:

$$t_{sat} = \left(1 + \frac{38.10^{-6}}{24 \times 3600}\right) t_{rec} + \text{offset}.$$

Time frames in which time elapses in a related way live in the same *time island*. Time frames which belong to different time islands have unrelated notions of time, which can flow independently. Although this representation of time is natural in heterogeneous models, it has seldom been studied in the literature. Such situations are observed when we consider time as measured by the processor clock in distributed systems. Each system performs computations in its own time frame where the clock speed varies with respect to processor load and power consumption. Even if for each run of the distributed system, it was possible to map dates from one system to dates in the other system, this mapping is not predetermined, and we should be able not to specify such a mapping. However, we need a time frame in each local system to specify that

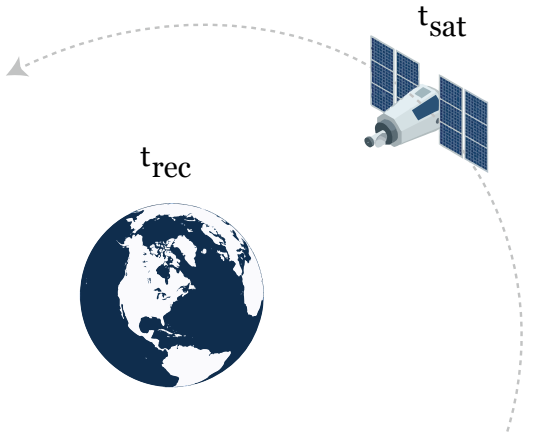



Figure 5: Effects of relativity on GPS satellite time measurements

some computation is eventually performed after a given duration. We therefore need time frames that live in different time islands.

## 1.3 GOALS AND ISSUES

We are interested in languages that allow for specifying timed coordination of subsystems by addressing the following conceptual issues:

- events may occur in different sub-systems at unrelated times, leading to *polychronous* systems, which do not necessarily have a common base clock,
- the behavior of the sub-systems is observed only at a series of discrete instants, and time coordination has to take this discretization into account,
- the instants at which a system is observed may be arbitrary and should not change its behavior (stuttering invariance),
- coordination between subsystems involves *causality*, so the occurrence of an event may enforce the occurrence of other events, possibly after a certain duration has elapsed or an event has occurred a given number of times,
- the domain of time (discrete, rational, continuous,...) may be different in the subsystems, leading to *polytimed* systems,
- the time frames of different sub-systems may be related (for instance, time in a GPS satellite and in a GPS receiver on Earth are related although they are not the same).

In this scheme, our problem is twofold:

1. How can we specify the occurrence of events in related or unrelated time frames?
2. How can we derive potentially infinite traces for such specifications that combine different paradigms?

## 1.4 RELATED WORK

The interest of our study was focused on the Tagged Events Specification Language (TESL), a specification language developed by Boulanger *et al.* [BJHP14] to coordinate heterogeneous models and behaviors inside the ModHel'X simulation framework [HB09]. It deals with discrete events (represented as *clocks*) and expresses time and causality constraints. They can be gathered in three main classes.

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".

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.

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.

TESL is a polychronous and polytimed language. Polymorphic time exists in the family of synchronous languages that were designed in the 1980's, such as Lustre [HCRP91], Esterel [Ber00] and Signal [GBBG87]. In these languages, time is purely logical (there are no dates nor chronometric durations), and can be used for modeling occurrences of any kind of events, hence the polymorphic nature of time.

Some synchronous models derive all clocks from a root clock, which defines the instants where the system reacts. On the contrary, polychronous models [TBG+13] do not constrain all clocks to derive from a single reaction clock, allowing a more relaxed and concurrent execution of systems. Polychrony is supported by the Signal language and in Polychronous automata [LGGTB15].

Another source of inspiration for TESL is the Clock Constraint Specification Language (CCSL) [MDADS10, And09], which supports asynchronous constraints on the occurrence of events. It has an executable semantics [ZM16] and a denotational semantics [DAG14]. However, all these approaches do not support chronometric clocks, with dates and durations. They measure time in numbers of ticks on a clock, not in elapsed durations on a time scale. In opposition, TESL supports chronometric time, and allows different clocks to live in different time frames, hence its *polytimed* nature.

Timed automata [AD94b] support both discrete events and measuring durations on a time scale, with several mechanization approaches of their semantics [GBFA13, PM01, HCOH93]. However, this time scale is global and uniform: all clocks in a timed automaton progress at the same rate.

The GEMOC initiative [CCF+15] has been putting the focus on the development of frameworks to facilitate the creation and integration of heterogeneous modeling languages. In particular, the BCOoL language [VLDCM15] is specifically targeted at coordination patterns for Domain Specific Events, which define the interface of a domain specific modeling language.

## 1.5 CONTRIBUTIONS

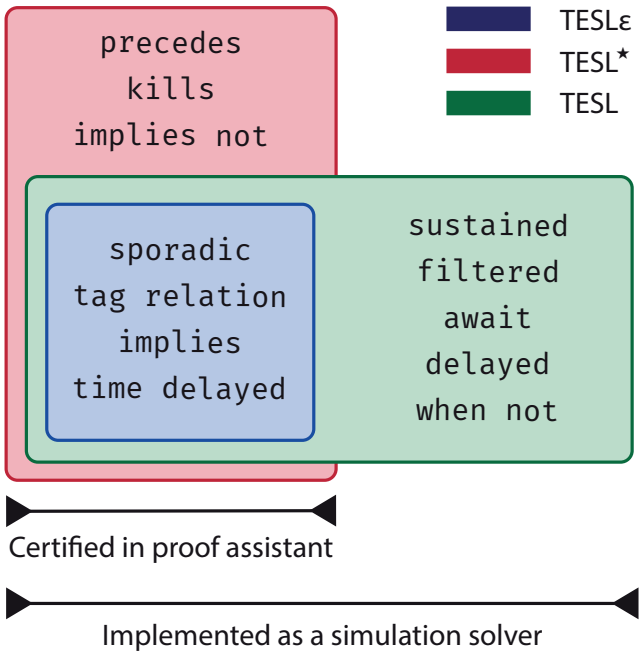



Figure 6: Overview of the contributions developed in this thesis

Our contributions are divided in five parts which are summarized in Figure 6.

1. In Chapter 3, we first isolate a fragment of the TESL language that we name TESLε and that is expressive enough to deal with event-triggered and time-triggered event synchronization. This minimalistic language consists of four constraint operators:
    - `sporadic`: An event will occur at a specific date;

- `tag relation`: The timeframes of clocks are synchronized through some arithmetic relation;
- `implies`: An event occurrence will instantaneously trigger another one ;
- `time delayed`: An event occurrence will trigger another one after a delay measured on some timeframe.

Our understanding of these mathematical objects first starts with defining how these operators shall describe *runs* (also called *execution traces*). This is done with a *denotational semantics* that gives a mathematical description of the run set for each operator. Then, we need another formalism to generate these runs in a constructive way. This is done with a *symbolic operational semantics* that incrementally generates runs by means of a rewriting system. Additionally, we prove the key property of *local termination* that ensures that computing one simulation instant will terminate.

2. In Chapter 4, we develop an extension named TESL$^\star$. Above the previous minimal fragment, we investigate on the addition of operators to increase the expressiveness of the language to the extent of compliance towards real-time modeling frameworks and standards. We notably add asynchrony to express events with constraints on past events, as well as some other operators.
   - `precedes`: An occurrence of an event is necessarily preceded by the occurrence of another event;
   - `implies not`: An event occurrence prevents an occurrence of another event at the same instant;
   - `kills`: An event occurrence prevents any further occurrence of another event.

   In the same way, we provide to these additional operators, a conservative extension to the denotational and the operational semantics.

3. In Chapter 5, we complete the minimal fragment given in TESL$\varepsilon$ with additional operators as originally found in the TESL language. They consist of five sequential symbols, i.e., they embed in their structure a state to be taken into account.
   - `filtered implies`: Implication between events holds within a rational pattern;
   - `delayed implies`: An event occurrence triggers another one after delaying by a number of event occurrences;
   - `sustained implies`: Implication between events holds within a range of start and stop control events;
   - `await implies`: An occurrence of an event is triggered when all the awaited events have occurred at least once;

- `when implies`: An occurrence of an event is triggered when two events occur simultaneously.

They are covered with an extension to the operational semantics for solving purposes.

4. The goal is to relate the operational and denotational semantics that were previously defined. This is done in Chapter 6 through a *coinductive characterization* of the language fragments, in the style of expansion laws as in modal logics. Doing so bridges the gap between both semantics, and allows us to state safety properties on our approaches. In particular, our study provides three main properties on our system:

   SOUNDNESS The operational semantics produces runs that are sound with respect to the denotational semantics;

   COMPLETENESS Any denoted run of the denotational semantics can be produced by the operational semantics;

   PROGRESS Also served for testing and simulation purposes, the operational semantics produces runs where time eventually progresses;

   The theory has been formalized into the Isabelle/HOL proof assistant and all mechanized contributions are available at

   `https://github.com/heron-solver/hygge`

5. Finally in Chapter 7, we explore possibilities given by the operational semantics we developed for TESL. We applied it for testing and monitoring concurrent models. This was done by implementing it into a solver for specifications, named Heron [NBB+17]. It solves TESL specifications and returns satisfying run prefixes. The solver and its source code are distributed at

   `https://github.com/heron-solver/heron`

# 2

# THEORETICAL AND TECHNICAL BACKGROUND

*When your tires are flat, you look at your tires, they are flat. When your software is broken, you look at your software, you see nothing.*

— Gérard Berry

This chapter is dedicated to the introduction of related tools and technologies that are studied in this thesis. Section 2.1 aims at presenting how some systems are nowadays modeled. Then we focus on a modeling and simulation platform named ModHel'X in Section 2.2 that has become the root of our study. Finally, we present an environment for formal proofs, named Isabelle/HOL in Section 2.3, that is the key tool to fully certify the well-foundedness of our approaches.

## 2.1 SYSTEMS, MODELS AND THEIR INTERACTIONS

### 2.1.1 *Synchronous and Reactive Systems*

Embedded systems are nothing new. However, their pervasive presence in everyday-life objects (cars, phones, home appliances) has drawn considerable attention in recent years. Among different models, languages and formalisms, the public interest has grown on *synchronous reactive* programming languages in the 1980's. This approach is central to the design of digital circuits such as processors, and hence applied to the field of embedded software.

They rely on the *synchronous hypothesis*, which assumes that computations and behaviors can be divided into instantaneous discrete computation steps, called *reactions* or *execution instants*. A program is assumed to react quickly enough to perceive all external events and in suitable order. A close analogy can be made with cycles in computation circuits. This *lockstep* computation paradigm is necessary to program circuits containing registers as found in most of computers. A cycle represents a logical step, not a physical time step. The paradigm is interesting in the sense that it allows to be compiled into executable code and guarantees deterministic execution and space and time bounds.

Similarly to short-circuits in electronics or deadlocks in parallel systems, the synchronous hypothesis may cause paradoxes (causality cycles). In an imperative language such as Esterel, these are preceived as

LACK OF BEHAVIOR A signal is emitted if and only if it is absent ;

MULTIPLE BEHAVIORS A signal must be present if and only if it is present.

#### 2.1.1.1 *Lustre*

Lustre is a declarative language based on the dataflow model as used by most control engineers: this for instance includes analog diagrams, block-diagrams, gates and flip-flops. It is a core language for the SCADE development environment [Ber07].

A Lustre program is a set of equations, defining sequences of values. It contains point-wise boolean and arithmetical operators, a unit-delay and sampling operators. It is directly equivalent to graphical representation by block diagrams. Moreover, it is *synchronous* in the sense that at every tick of a global clock, every operation does a step. A variable of common usage is in fact a stream ruled by equations. For instance, the equation $Z = X + Y$ means that for every instant $n \in \mathbb{N}$,

$$Z_n = X_n + Y_n$$

A satisfying stream for this equation would trivially be:

| X | 1 | 2 | 1 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|
| Y | 2 | 4 | 2 | 1 | 1 | 2 | ... |
| X + Y | 3 | 6 | 3 | 5 | 6 | 8 | ... |

Discretizing time and events with synchrony, allows the language to make use of the unit delay. This consists of referring to the value of a stream at the previous step of the computation. Let us consider the linear recurrence $(S_n)_{n\in\mathbb{N}}$ with $S_0 = X_0$ and $S_n = S_{n-1} + X_n$. It is written in Lustre as

```
S = X -> pre S + X
```

As depicted by the following stream, Lustre features two operators used in the value delay. One is the unary operator of precedence, written `pre`. It yields the value of some stream at the previous computation step. Then the binary operator "followed by", written `->`, yields a new stream where the first operand initializes the stream then is followed by the stream of the second operand. In particular, the latter is necessary to define initial values for delayed streams which face undefined values, as depicted below by the nil symbol.

| X | 1 | 2 | 1 | 4 | 5 | 6 | ... |
|---|---|---|---|---|---|---|---|
| `pre S` | nil | 1 | 3 | 4 | 8 | 13 | ... |
| `pre S + X` | nil | 3 | 4 | 8 | 13 | 19 | ... |
| `S = X -> pre S + X` | 1 | 3 | 4 | 8 | 13 | 19 | ... |

### 2.1.2 *Realtime Systems*

Finite-state machines – or automata [HMU01] – allow to elide programming details in order to reason over global behaviors. They enjoy interesting properties such as decidability, and enough expressiveness to encode logical problems such as Presburger arithmetic. However, they are limited by a too high level of abstraction. They deal with pure logical time: events occur and are ordered, but their distance is not measurable. Hence systems in which time constraints need to be considered, cannot be expressed by pure automata. Rajeev Alur and David Dill introduced *timed automata* [AD94a] in the early 1990s to overcome this issue. Their formalism is based on the addition of real-valued variables to measure time duration between transitions of the automation. Such variables can be reset along some transitions, or serve as transition guards, and progress at the same rate.

A simple example depicting a situation where timed constraints are necessary, would be the mouse double-click. A simple click selects an item on screen, while a double-click opens it. A double click is detected when the elapsed time, i.e., the duration, between two clicks is less than 500 ms. The timed automaton in Figure 7 captures this behavior (the 100 ms delay before returning to the `idle` state after detecting a click or a double click is arbitrary).

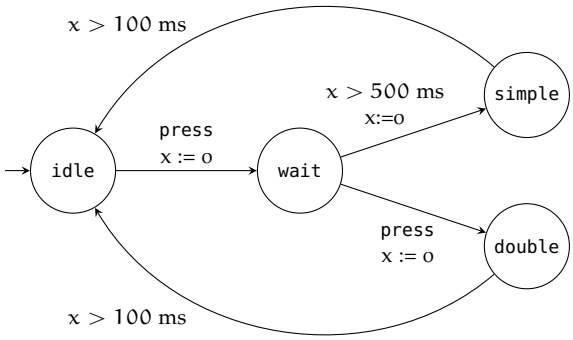



Figure 7: Capturing a mouse double-click with a timed automata

MODELING WITH TIMED AUTOMATA In the following paragraphs, we recall a few properties over timed automata as defined by [AD94a]. Note such clocks are slightly different from those we have in mind.

CLOCKS CONSTRAINTS. For a set of clock variables $\mathcal{C}$, the set $\Phi(\mathcal{C})$ is inductively defined by

$$\delta \quad := \quad x \leqslant \lambda | \lambda \leqslant x | \neg\delta | \delta \wedge \delta$$

where $x$ is a clock in $\mathcal{C}$ and $\lambda \in \mathbb{R}$

**Definition 3 (*Timed Automaton [AD94a]*)**

A *timed automaton* $\mathcal{A}$ is a tuple $\langle S, S_0, \mathcal{C}, \Sigma, I, E \rangle$, where

$S$ is a finite set of locations (or nodes),

$S_0 \subseteq S$ is a set of initial locations,

$\mathcal{C}$ is a finite set of real-valued variables standing for *clocks*,

$\Sigma$ is a finite alphabet called *actions*,

$I : S \to \Phi(\mathcal{C})$ maps a location to a set of clock constraints,

$E \subseteq S \times S \times \Sigma \times \mathcal{P}(\mathcal{C}) \times \Phi(\mathcal{C})$ gives the set of transitions.

An edge $\langle s_1, s_2, a, R, \delta \rangle$ represents a transition from location $s_1$ to $s_2$ on input symbol $a$. The set $R \subseteq \mathcal{C}$ denotes the clocks to be reset (to 0) with this transition, and $\delta$ is a clock constraint over $\mathcal{C}$ to be satisfied when the transition is taken.

**Example 4** (Double-click Automata)**.** The automata shown on Figure 7 is made of four locations $S = \{\texttt{idle}, \texttt{wait}, \texttt{simple}, \texttt{double}\}$, whose start is set at `idle`. They rely on a unique action $\Sigma = \{\texttt{press}\}$ and a unique clock $\mathcal{C} = \{x\}$. Transitions and clock constraints are given by:

$$\begin{aligned} E \;=\; \{\; & \langle \texttt{idle}, \texttt{wait}, \texttt{press}, \{x\}, \varnothing \rangle, \\ & \langle \texttt{wait}, \texttt{simple}, \bot, \{x\}, \{\neg x \leqslant 500\} \rangle, \\ & \langle \texttt{wait}, \texttt{double}, \texttt{press}, \{x\}, \varnothing \rangle, \\ & \langle \texttt{simple}, \texttt{idle}, \bot, \varnothing, \{\neg x \leqslant 100\} \rangle, \\ & \langle \texttt{double}, \texttt{idle}, \bot, \varnothing, \{\neg x \leqslant 100\} \rangle \quad \} \end{aligned}$$

Although timed automata enjoy polynomial-space reachability analysis, they suffer in terms of language-theoretic issues: they cannot be complemented nor determinized, and the problems of language inclusion and universality are undecidable [AD94a]. More importantly, the structure of global time does not allow to specify models evolving at different rates, which is one of the requirements of our study.

### 2.1.3 *Heterogeneous Systems*

The problem of heterogeneous modeling rises when different models of computation are involved in modeling subparts of the system, and how they should interact. We consider how to combine different paradigms such as: state machines, block diagrams, process networks, discrete systems, continuous systems.

PTOLEMY II The approach developed in Ptolemy [Pto14, EJL+03] relies on *hierarchical* composition. The idea is that each hierarchical level uses only one Model of Computation (MoC). Then their semantics are *adapted* along three axes: data, time, and control. Each paradigm is named a *semantic domain* provided with execution rules, called a MoC. Each model is defined by actors, that stand for components that are executed concurrently and sharing data and messages via ports. An example of model is shown on Figure 8 where three actors A, B and C interact through their ports. The director (green rectangle) describes the MoC of the current actor. Then each *composite* actor gets described by "smaller" actors.

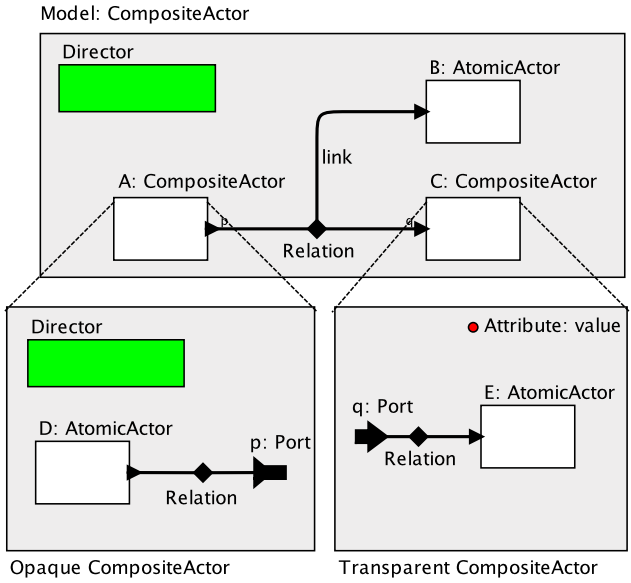



Figure 8: A hierarchical actor model of Ptolemy II (extracted from [Pto14])

Heterogeneity is featured by MoC that Ptolemy handles. Here are a few:

COMMUNICATING SEQUENTIAL PROCESSES Processes communicate by means of instantaneous rendezvous points. A token is put in a receiver and retrieved by another process during the rendezvous.

CONTINUOUS TIME An Ordinary Differential Equation (ODE) can be expressed with integrators and each port in this domain denotes a function on continuous time.

DISCRETE EVENT Actors communicate with events placed on a timeline (created by Lukito Muliadi [Mul99]), where each of them has a value and a timestamp.

PROCESS NETWORK Processes communicate by means of FIFO queues [Kah74] as implemented in receivers.

SYNCHRONOUS DATAFLOW A restriction of the latter is the Synchronous Data Flow (SDF) model [LM87]. The execution of an actor is done by consuming a fixed number of tokens from input and producing a fixed number of token on output. This allows to statically analyze deadlock and boundedness.

The interaction between components needs an interface for each subpart. The goal is to manage how sending and receiving information for each actor is handled in a generic way. [dAH01] propose a theoretical model in which the issue is tackled by using *interface automata*. Each actor is provided with invocation methods that describe how they are fired and how they receive input tokens. Such methods are directly put in correspondence with transitions of interface automata.

## 2.2 EXECUTION AND SIMULATION WITH MODHEL'X

ModHel'X [HB09] is a framework that similarly addresses the issue of modeling several formalisms into a single model. In this framework, heterogeneous components can be modeled by heterogeneous design paradigms, and the interaction among components and the environment refers to model composition. Compared to Ptolemy in which interface between components can be incompletely designed, ModHel'X adds a structure of *interface blocks* that prevents this lack of design.

### 2.2.1 *Initial Inspiration*

THE CLOCK-CONSTRAINT SPECIFICATION LANGUAGE CCSL [Mal08] is a synchronization language used in the TimeSquare modeling framework. It applies as an intermediate language for the Modeling and Analysis of Real-Time and Embedded systems (MARTE) profile [SG13] for the Unified Modeling Language (UML), and provides specification and simulation for timed models. Events are modeled by clocks that define the timeline of events. Logical time is the main paradigm where time is counted as a number of events. This language allows to write specifications for the coordination of models thanks to polychronous clauses:

SYNCHRONOUS CLAUSES. Events occur instantaneously because other events occur: coincidence, exclusion and subclocking.

ASYNCHRONOUS CLAUSES. Events occur due to other events in the past: precedence.

Dealing with pure logical time, as in temporal logics, can be useful nowadays from a computer-centric view, notably in the context of energy saving where processor clocks no longer have fixed frequencies, but are lowered (or

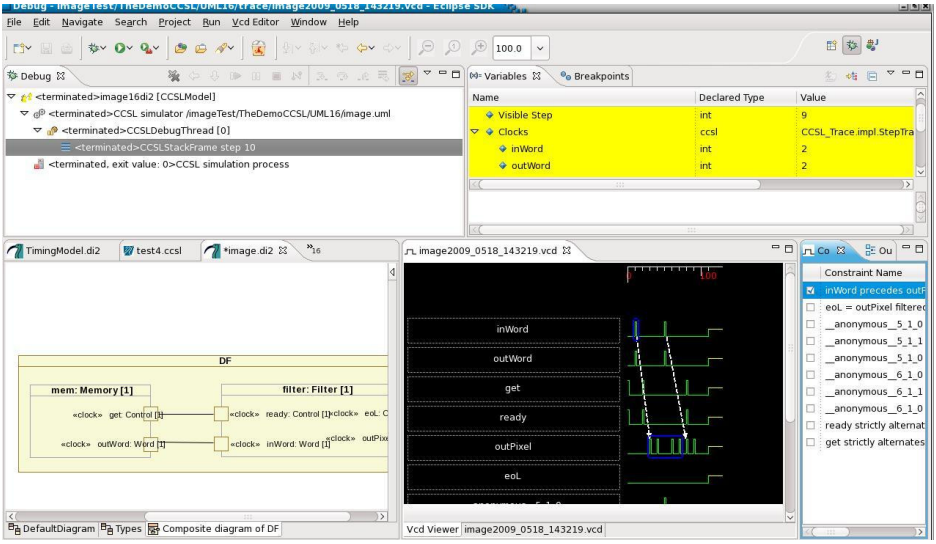



Figure 9: The TimeSquare framework

increased) depending on the current computation needs. Still, the restriction brings a too high level of abstraction from a system-centric view. Even if the attempt at modeling realtime problems [PFD11] can be in some cases reduced to time discretization, where a continuous time domain would be sampled and reasoned upon with discrete instants, this limitation however entails that occurring events would be observed at the limits of the sampling window. This approximation may indicate that an event occurs instantaneously with another, while reality shows that they are just very close.

THE TAGGED SIGNAL MODEL The Tagged Signal Model [LSV96] is a framework aiming at describing with a denotational semantics a MoC by means of signals and systems. A signal is modeled by a sequence of samples, each of them is attached a *time tag* ordered in a domain. Time between different domains can be adapted with morphisms. On the contrary, this framework is too mathematically-theoretical and does not integrate a constructive way to describe such objects. However, this idea is integrated in the TESL language, and aims at bridging the final gap to solve MARTE models, where time is not only discrete but also continuous.

Following this idea, Tag Machines [BCC+08] propose an algebra of tag structures and define parallel composition for heterogeneous models similarly. This setting precisely and mathematically captures the notions of logical time, physical time, causality, scheduling constraints... as a unifying mathematical framework to relate such paradigms. However, relations between time scales are not explicit and the semantic model lacks constructiveness, which prevents a machine-concretization of the ideas.

### 2.2.2 *From ModHel'X to TESL*

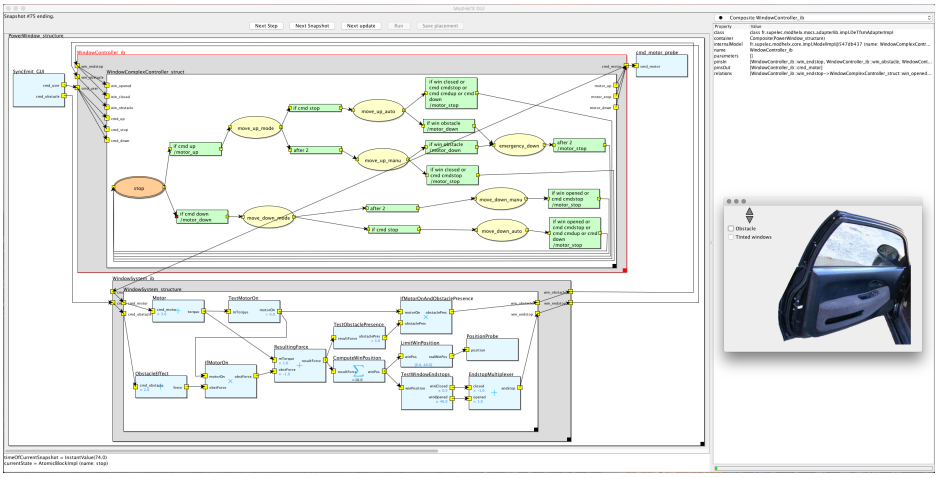
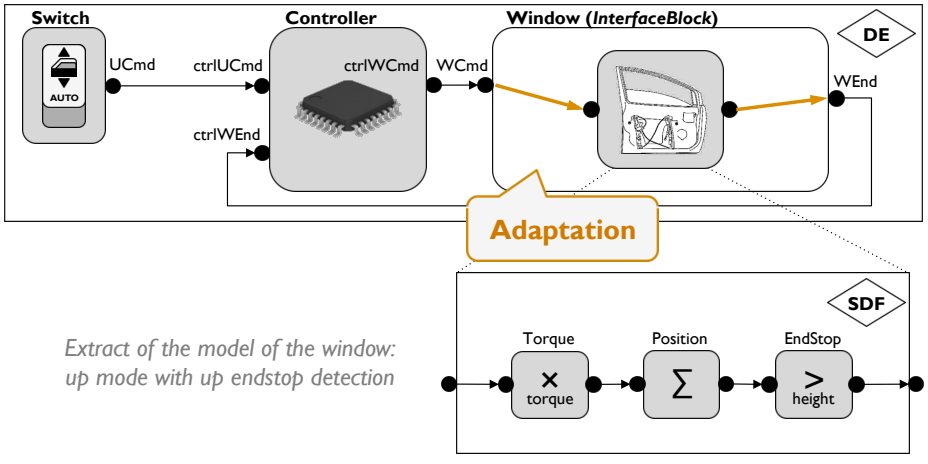


Figure 10: The ModHel'X framework with the car power window case study

The main issue of modeling and simulation revolves around determining the global behavior of such a system and in defining how the heterogeneous parts in question shall be composed to compute this global behavior. ModHel'X [BHJM11] is an experimental platform which provides an interface between these heterogeneous components. It is based on a block diagram description of models, encapsulated in interface blocks. Doing so allows to model the heterogeneous parts, but also how they are supposed to interact. It relies on TESL to specify the coordination of the heterogeneous parts of a model.




*Extract of the model of the window: up mode with up endstop detection*

Figure 11: Subsystem interface within the supermodel (extracted from [BHJM11])

The example of the power window is a well-known case study in heterogeneous modeling. It depicts the challenge of modeling a power window as widespread in cars nowadays. Each part of the system is modeled by a MoC, i.e., an abstract machine with semantics rules. Figure 4 illustrates this example in the context of the ModHel'X modeling framework as well. This system contains a position button, the controller and the electromechanical subsystem. The controller is modeled by a timed finite state machine, the electromechanical part by a synchronous dataflow model, and the multiplexed communication on the bus is described by discrete events.

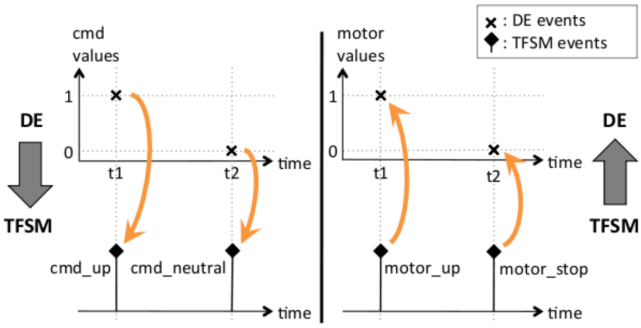



Figure 12: Adapting semantics between discrete events and timed finite state machines (extracted from [BHJM11])

While the execution of each of these small models is known and standard, heterogeneous modeling tackles the issue of executing these models in parallel. This supermodel should synchronize the inputs and outputs of each of its component part. ModHel'X abstracts models to consider their interface only. The execution flow runs through all the layers of the supermodel, in a way that *semantic adaptation* is ensured and performed at each border of models: they are based on their corresponding *interface blocks* as shown on Figure 11.

Figure 12 depicts the adaptation performed by the discrete event/timed FSM interface block on the running example. In this case, the translation is trivial. On the left hand-side, discrete events are translated into timed finite state machines (TFSM) events: an event on the `cmd` discrete event input pin is interpreted into a `cmd_up` or `cmd_neutral` TFSM event according to its boolean value (0 or 1). The other way of the translation is analogous.

To adapt semantics, ModHel'X uses interface blocks as in Figure 13 to describe how information is synchronized. A *block* is a black-box with an interface. It has *pins* for input and output; and a *structure*, that is a relation between pins and is defined by the semantics of the MoC. A model is the combination of the MoC and the structure.

Finally, the time constraints for the semantic adaptation in the whole supermodel are flattened into a TESL specification (given in Listing 4 on page 43) that is detailed and explained further in Section 3.2.2.

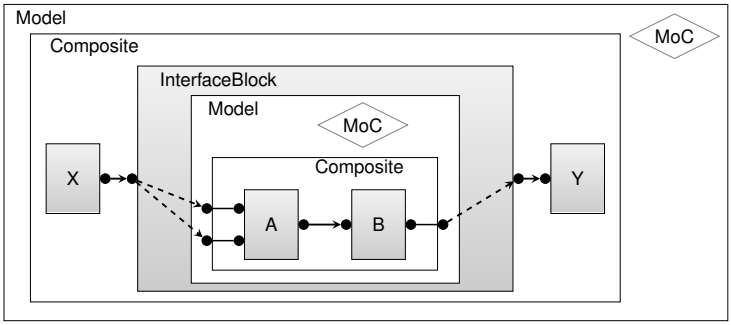



Figure 13: Interface blocks in ModHel'X (extracted from [BHJM11])

## 2.3 INTERACTIVE THEOREM PROVING IN FORMAL METHODS

To understand the foundations of mathematics, logicians from the past centuries investigated how to reason over numbers, and without loss of generality, over general structures. It is necessary to define a sound basis of computability, to ensure that all computations done nowadays and in various areas of engineering, are themselves sound, that they do not run on inconsistent foundations and shall not lead to mistakes and failures. To ensure that, *metamathematics*, now called *logic*, tackle the issue of defining such consistent foundations.

Likewise, defining semantics for specification or programming languages shall rely on similar accurate and sound basis for the verification and the simulation of models. Interactive theorem proving tames the issue of formalizing theories and proofs by mechanizing the logical processes. Hence, a semantic or logical theory is proved, the same way as the execution of a program. This section presents a brief overview of the historical foundations that have led to the actual Isabelle/HOL proof environment [NWP02, NK14] which is at the heart of our methodology.

### 2.3.1 *Mathematical Preliminaries*

We give hereafter some useful mathematical notations for the reader and shall assume that she or he is familiar with set and recursion theory. That is, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{D}$ will denote usual sets in mathematics. $\mathbb{U} = \{()\}$ is the singleton set which contains exactly one element called *unit*, and $\mathbb{B}$ is the set of booleans.

#### 2.3.1.1 *Sets, Products and Kleene Stars*

Let $S$ be a set. The operator card denotes set cardinality for finite sets, i.e., "the number of elements in a set". $\mathcal{P}(S)$ denotes the powerset of $S$ and $S^n$ as the $n$-time *cartesian product* of $S$. In particular, its elements will be called *tuples over* $S$ and denoted here as $(u_0, \ldots, u_{n-1})$, where each $u_0, \ldots, u_{n-1}$ is a *component* of the tuple. Moreover, a *sequence over* $S$ is an arrow $\mathbb{N} \to S$ and is denoted as

$(u_i)_{i\in\mathbb{N}}$. A *word over* $S$ is an object of the Kleene closure $S^\star$ which be can be defined as follows

$$S^\star \quad = \quad \bigcup_{i\in\mathbb{N}} S^i$$

We adopt the following and similar notations for tuples, sequences and words. The $k$-*component projection* of a sequence $u$ is written $u_k$ or $u[k]$. The *prefix of length* $k$ of the word $u$ is $u_{<k}$ or $(u_i)_{i<k}$. The *suffix starting at the* $k$-*th component* of $u$ is $u_{\geqslant k}$ or $(u_i)_{i\geqslant k}$.

Furthermore, a *subsequence* (or *subword*) of $u$ is a sequence $v$ such that there exists a non-decreasing arrow $\phi : N \to N$ satisfying $v_k = u_{\phi(k)}$ (where $N \subseteq \mathbb{N}$). This weakens the following idea: $y$ is a *factor of* $u$ whenever there exists $x, z$ such that $u = x \cdot y \cdot z$ where $\cdot$ denotes the *concatenation* of words.

#### 2.3.1.2 *Relations and Reductions*

Let $S$ be a set. A *binary relation* $R$ on $S$ is a subset of $S \times S$. Two elements $x$ and $y$ of $S$ are said to be *in relation with* $R$, also denoted $x\ R\ y$ if $(x, y) \in R$. Recall

- $R$ is *reflexive* iff for all $x \in S$, $(x, x) \in R$ ;
- $R$ is *symmetric* iff for all $x, y \in S$, $(x, y) \in R$ implies $(y, x) \in R$ ;
- $R$ is *transitive* iff for all $x, y, z \in S$, $(x, y) \in R$ and $(y, z) \in R$ imply $(x, z) \in R$.

Let $R$ be a binary relation and $S$ a set. We denote $R_{|S}$ as the restriction of $R$ over domain $S$, i.e., $R_{|S} = \{(x, y) : x \in S \text{ and } x\ R\ y\}$. Composing relations $R_1$ and $R_2$ is given by

$$R_1 \cdot R_2 \quad \stackrel{\text{def}}{=} \quad \{(x, y) \quad | \quad \exists z \quad (x, z) \in R_1 \text{ and } (z, y) \in R_2\}$$

Finally, we define

- $R^0$ as the reflexive closure,

$$R^0 \quad = \quad \{(a, a) \quad | \quad a \in S\}$$

- $R^n$ as the $n$-composition,

$$R^n \quad = \quad R \cdot R^{n-1}$$

- $R^+$ as the transitive closure,

$$R^+ \quad = \quad \bigcup_{n=1}^{\infty} R^n$$

- $R^*$ as the reflexive-transitive closure.

$$R^* \quad = \quad R^+ \cup R^0$$

#### 2.3.1.3 *Domains, Recursion and Fixpoints*

In recursion theory, let $S, S'$ be two sets. The set of *partial recursive functions* is $S \rightharpoonup S'$, while the set of *total functions* is $S \to S'$. Let $P$ be a predicate, Kleene's $\mu$-operator (also called *minimization operator*) is defined as

$$\mu x.\ \Big[P(x)\Big] \quad = \quad \inf\left\{x \in \mathbb{N} : P(x)\right\} \quad \text{if} \quad \exists x \quad P(x)$$

Finally, we will need some fixpoint theory. Let $(\mathbb{X}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a complete lattice. Recall the following notations

$$\begin{aligned} \mathsf{fp}(f) &= \left\{x \in \mathbb{X} : f(x) = x\right\} \\ \mathsf{lfp}_x\ f &= \mathsf{min}_{\sqsubseteq_{\mathbb{X}}}\left\{y \in \mathsf{fp}(f) : x \sqsubseteq_{\mathbb{X}} y\right\} \text{ if exists} \\ \mathsf{lfp}\ f &= \mathsf{lfp}_{\bot}\ f \end{aligned}$$

The usage of least fixpoint operators is restricted to specific algebraic structures called *lattices*. The following is a very fundamental fixpoint theorem which gives the existence and ability to *construct* the fixpoint of a function by *successive iterations* of the function

**Theorem 5 (*Fixpoint in a CPO (Tarski, Kleene)*)**

*If* $f : X \to X$ *is continuous in a complete partial order* $X$ *and* $\bot$ *is the least element of* $X$ *then* $\mathsf{lfp}\ f$ *exists and can be computed as*

$$\mathsf{lfp}\ f \quad = \quad \mathsf{sup}\left\{f^n(\bot) : n \in \mathbb{N}\right\}$$

□

### 2.3.2 *Programming with Lambda-calculi*

#### 2.3.2.1 *Untyped Lambda-calculus*

In the 1930s, Alonzo Church first introduced a model of computation called *lambda-calculus*. In this logical system where terms are called λ-terms, the goal is to follow the basic rules of functions,

VARIABLE A symbol which indistinguishably serves as parameter or function.

APPLICATION Applying a function to an argument.

ABSTRACTION Defining a function with an additional parameter. A variable is abstracted inside the body of a function.

This system is purely syntactical and only uses variable binding and substitution to serve its goal. All terms are undistinguished: the parameter of a function can be a function itself. For instance, the λ-term $\lambda x.x$ stands for a mathematical object that associates a λ-term $x$ with the same λ-term $x$. In mathematics, it can be interpreted as the trivial identity function $x \mapsto x$. In computer science, it can be a program function that outputs its input.

**Definition 6 (*Grammar of the untyped λ-calculus*)**

A λ-term $M$ of the untyped λ-calculus is given by

$$\begin{aligned} M \quad ::= \quad & x && \text{(variable)} \\ | \quad & \lambda x.\, M && \text{(term abstraction)} \\ | \quad & M\ M && \text{(term application)} \end{aligned}$$

However, the meaning of such a system needs to be properly defined. A simple operational semantics can be defined by reductions as given by the following rule.

**Definition 7 (*Reductions of the untyped λ-calculus*)**

$$(\lambda x.M)E \underset{\beta}{\rightarrow} M[x/E] \qquad (\beta\text{-reduction})$$

**Definition 8 (*Equivalence in the untyped λ-calculus*)**

$$\lambda x.M[x] \underset{\alpha}{\rightarrow} \lambda y.M[y] \qquad (\alpha\text{-conversion})$$

$$\lambda x.Mx \underset{\eta}{\rightarrow} M \qquad (\eta\text{-expansion})$$

A β-reduction explicits how some term is computed into a "smaller" one, e.g., computing the value returned by a function applied to a parameter. The α-conversion corresponds to renaming the parameter name of a function from $x$ to $y$. Note this does not change the behavior of the term if the newly renamed variable has not already occured before. The application of a function $\lambda x.M$ on a term $E$ directly corresponds to the term $M$ where free occurrences of symbol $x$ are substituted by $E$.

CONFLUENCE AND NON-TERMINATION The β-reduction we mentioned earlier can be applied at different stages of a term. The order of applying such a reduction was indeed left unspecified. We can ask ourselves: *If we apply some order of reduction instead of another order of reduction, will we get the same result in the end?* The following theorem gives the answer by stating that β-reduction is confluent. Two sequence of reductions will eventually reduce to the same term.

**Theorem 9 (*Confluence of λ-calculus (Church, Rosser)*)**

*Let $M, M_1, M_2$ be λ-terms such that $M_1 \underset{\beta}{\overset{*}{\leftarrow}} M \underset{\beta}{\overset{*}{\rightarrow}} M_2$. There exists a term $M'$, such that*

$$M_1 \underset{\beta}{\overset{*}{\rightarrow}} M' \underset{\beta}{\overset{*}{\leftarrow}} M_2.$$

□

Even if confluence property gives good guarantees, it is however not sufficient to provide a safe logical framework of computation. Indeed, this lambda-calculus is untyped and permits non termination: there are terms that can be infinitely reducing, they are said to be *non strongly normalizing*.

**Example 10** (Non-termination of $\Delta\Delta$)**.** For instance, let us consider

$$\Delta \quad = \quad \lambda x.(x\ x)$$

The term $\Delta$ is a function that takes as input a function $x$, and yields its application on itself. It is a $\beta$-normal form as there exists no further sequence of $\beta$-reductions starting from this term. However, we observe that

$$\begin{array}{rcll} \Delta\Delta & = & \big(\lambda x.(x\ x)\big)\big(\lambda x.(x\ x)\big) & \to_\beta \quad \big(x\ x\big)\big[^{x}/_{\lambda x.(x\ x)}\big] \\ & & & = \quad \big(\lambda x.(x\ x)\big)\big(\lambda x.(x\ x)\big) \\ & & & = \quad \Delta\Delta \end{array}$$

The term $\Delta\Delta$ is looping. Whenever we apply a $\beta$-reduction, the same term is being yielded.

The previous observation leads to the following: the lambda-calculus system is as expressive as Turing machines. Any Turing-computable problem can be expressed in lambda-calculus. It consequently serves as a universal model of computation given the Church-Turing thesis. This is exhibited by Curry's fixed-point combinator that serves to define recursion,

$$\mathsf{Y} \quad \stackrel{\text{def}}{=} \quad \lambda F.\ (\lambda x.\ F(x\ x))\ (\lambda x.\ F(x\ x))$$

that has the key property of unfolding the recursion of a *functional* $\mathsf{F}$ (again a $\lambda$-term),

$$\mathsf{Y}\ \mathsf{F}\ x \quad = \quad \mathsf{F}\ (\mathsf{Y}\ \mathsf{F})\ x$$

**Example 11** (Factorial function)**.** To define a factorial function fact, we first assume that we have a conditional if-then-else, the natural integers (e.g., Church encoding), as well as multiplication $\times$, predecessor $-1$ and a predicate $\mathsf{isZero}$ that decides whether an integer is zero. The functional $\mathsf{F}$ we define is

$$\begin{array}{rcl} \mathsf{F}\ f\ n & \stackrel{\text{def}}{=} & \mathsf{if} \quad \mathsf{isZero}(n) \\ & & \mathsf{then} \quad 1 \\ & & \mathsf{else} \quad n \times (f(n-1)) \end{array}$$

As a matter of fact, the factorial function is defined as

$$\mathsf{fact} \quad \stackrel{\text{def}}{=} \quad \mathsf{Y}\ \mathsf{F}$$

A consequence of these statements is that the problem of deciding whether two $\lambda$-terms are equivalent is undecidable.

#### 2.3.2.2 *Simply-typed Lambda-calculus*

While the untyped λ-calculus is fully adequate to represent computations, it is yet unsafe to form a basis for logic. In 1940, Church introduced a computationally weaker but consistent variant of the latter, named the *simply typed lambda-calculus*, and abbreviated $\lambda^{\rightarrow}$. The idea is to consider that terms live in specific domains or universes, and shall remain inside: this is called *typing*. A *typing judgment* provides a statement that enforces some term $t$ to be of type $A$ given a collection of other type assignments $\Gamma$. It is written $\Gamma \vdash t : A$, and reads "given the typing context $\Gamma$, $t$ has type $A$".

The issue of term typing is ensured by proving typing judgments. To do so, Church provided derivation rules, which serve as proof rules. A typing judgment is valid whenever there exists a derivation tree whose leaves are only made of axioms, under the following rules

**Definition 12 (*Typing Rules of* $\lambda^{\rightarrow}$)**

$$\frac{}{\Gamma + \{x : A\} \vdash x : A} \quad (ax)$$

$$\frac{\Gamma + \{x : A\} \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \quad (\to_i)$$

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B} \quad (\to_e)$$

The axiom rule $(ax)$ states if some context contains the type assignment $x : A$, then $x$ admits type $A$; no further derivation is required. The introduction rule $(\to_i)$ states that if a term $t$ has type $B$ with a context containing the information that $x$ has type $A$, then the abstraction of $t$ with $x$ is a term $\lambda x.t$ of type $A \to B$. Finally, the elimination rule $(\to_e)$ states that under some context $\Gamma$, if $t$ is of type $A \to B$ and $u$ of type $A$, the application of $t$ on $u$ is a term $t\ u$ of type $B$.

*Remark* 13. The pattern that we observe on Rule $(\to_e)$ is very similar to modus ponens, which is a rule of inference in propositional logic corresponding to implication elimination. The theorem of proof-program isomorphism by Curry, Howard and de Bruijn will explore this idea.

*Remark* 14. Type annotations in simply typed lambda-calculus resemble a lot of functions as known in elementary set theory. Indeed, they can be straightforwardly interpreted so: types are mapped to sets, and typing judgments to

functions. Assume that for any basic type $\iota$ there exists a non-empty set $S_\iota$[1], we can inductively associate a set $[\![A]\!]$ to each type $A$:

$$\begin{aligned} [\![\iota]\!] &= S_\iota \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \end{aligned}$$

where $B^A$ is the set of all functions from set $A$ to set $B$.

#### 2.3.2.3 *Polymorphism with ML-style*

A program can serve the same purpose while being applied to different types. In our logical system, we wish a term to have the ability to admit possibly several distinct types, this is called *polymorphism*. Let us consider the example of a list-sorting function: sorting elements is independent of the type of the elements, given a comparison function. The simply typed lambda-calculus is monomorphic, while we want to be able to specify this kind of polymorphic type:

$$\forall\alpha.\ (\alpha \to \alpha \to \mathsf{bool}) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$$

which can later be *instantiated* with concrete types, such as

$$\begin{aligned} (\mathsf{int} \to \mathsf{int} \to \mathsf{bool}) &\to \mathsf{int\ list} \to \mathsf{int\ list} \\ (\mathsf{float} \to \mathsf{float} \to \mathsf{bool}) &\to \mathsf{float\ list} \to \mathsf{float\ list} \end{aligned}$$

Compared to $\lambda^\to$, we internalize the type variables inside the type system. This way we are able to reason and manipulate elements in an abstract way: the variable $\alpha$ is said to be an *abstract type*.

DAMAS-MILNER SYSTEM. Girard (1972) and Reynolds (1974) independently introduced the polymorphic $\lambda$-calculus, so-called System F. It is a direct extension of $\lambda^\to$, which abstracts types the same way that terms are abstracted. They introduce a different level of abstraction and application for types. Unfortunately, such a type system faces too much impracticability. More precisely in the Curry-style variant without explicit annotations, *type checking* i.e., checking whether some $\lambda$-term admits a specific type, is not decidable. Moreover, if we write terms without fully expliciting type annotations, the goal of *type inference* is to compute the weakest type that the term admits. Both problems have been proven to be equivalent [Wel99], and hence undecidable.

Damas and Milner's type system (1978), also abbreviated ML-style, offer a restricted approach of polymorphism that overcomes such obstacles. It is currently at the heart of well-known functional programming languages, such as

1 That is type $\iota$ is inhabited

Standard ML, OCaml and Haskell. The core idea is based upon a small extension of $\lambda^{\to}$ with a primitive let-binding. It consists of factoring out a subexpression $e_1$ in a term $e_2[^x/_{e_1}]$ using a let-binding $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$. Here is an implicitly-typed version, where an ML-term M is defined by the following grammar

**Definition 15 (*Grammar of ML-style*)**

A $\lambda$-term M of $\lambda^{\to}$ is given by

$$\begin{array}{rcll} M & ::= & x & \text{(variable)} \\ & | & \lambda x.\ M & \text{(term abstraction)} \\ & | & M\ M & \text{(term application)} \\ & | & \mathsf{let}\ x = M\ \mathsf{in}\ M & \text{(let-binding)} \end{array}$$

where we recall in gray the grammar for $\lambda^{\to}$.

In this system, the let-binding no longer exists as the usual syntactic sugar $\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 := (\lambda x.M_2)\ M_1$. It is defined as a language primitive with a proper typing rule. In particular, the grammar of types has two levels: *types* on one side, *type schemes* on the other side. From now on, we have

**Definition 16 (*Type schemes in ML-style*)**

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{(types)} \\ & | & \tau \to \tau & \\ \sigma & ::= & \tau & \text{(type schemes)} \\ & | & \forall \alpha.\ \sigma & \end{array}$$

In this setting, the problem of deciding whether two $\lambda$-terms are equivalent is decidable, and provides good logical foundations.

### 2.3.3 *Theory of Demonstration in a Nutshell*

To increase confidence in theories and proofs, we need to formalize the logical reasoning that drives proofs as we usually know them. The biggest advances were brought in the 1930s by Gentzen and Prawitz. We present a short introduction to the topic of *demonstration* based on natural deduction. The goal is to define a logical system to allow proof-checking in finite time. Consequently, this allows to mechanize the task of checking whether a proof is logically *valid*. In propositional logic, e.g., SAT problems, proofs can be given with truth tables and values which is a semantic perspective. Yet, proofs as we usually have in mind are rather based on a syntactical perspective made of conclusions derived from premises.

The first deduction systems were brought by Hilbert but were not practical to carry out proofs. The biggest advance was brought by Gentzen with the introduction of *judgment rules* for Natural Deduction (ND) [Dal04].

#### 2.3.3.1 *Natural Deduction*

Let us restrict ourselves to the connectives $\Rightarrow$ (implication), $\wedge$ (conjunction) and $\bot$ (false). ND is a proof calculus that allows to prove statements by deriving a proof tree. Each connective admits an introduction and an elimination rule. To prove a term, the proof designer will interactively provide a tree leading to axiom leaves.

**Definition 17 (*Deduction System*)**

A *deduction system* is a set of judgment rules of the kind

$$\frac{A_0 \quad \dots \quad A_{n-1}}{A_n}.$$

This rule reads as "from the assumptions $A_0$ to $A_{n-1}$, we conclude $A_n$".

In the following, $\neg A$ is an abbreviation for $A \Rightarrow \bot$.

**Definition 18 (*Rules of Natural Deduction*)**

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B} \quad (\wedge_e) \qquad \frac{A \quad B}{A \wedge B} \quad (\wedge_i)$$

$$\frac{A \quad A \Rightarrow B}{B} \quad (\Rightarrow_e) \qquad \frac{\begin{array}{c}[A]\\ \dots \\ B\end{array}}{A \Rightarrow B} \quad (\Rightarrow_i)$$

$$\frac{\bot}{A} \quad (\bot_e) \qquad \frac{\begin{array}{c}[\neg A]\\ \dots \\ \bot\end{array}}{A} \quad (\text{RAA})$$

The rules for $\wedge$ are straightforward: (Rule $\wedge_e$) If we have $A \wedge B$, then we can have $A$; as well as we can have $B$. (Rule $\wedge_i$) If we simultaneously have $A$ and $B$, then we have $A \wedge B$. For the implication connective: (Rule $\Rightarrow_e$) If we have $A$ and $A \Rightarrow B$, then we have $B$. (Rule $\Rightarrow_i$) If we can derive $B$ from $A$ (as an hypothesis), then we have $A \Rightarrow B$. In the case of $\bot$: (Rule $\bot_e$) An absurdity can derive anything (ex falso quodlibet); (Rule RAA) on the side, the principle of *proof by contradiction* is similar, but states that if we can derive a contradiction from $\neg A$, then there exists a derivation of $A$.

The terms given in square brackets introduce the principle of *canceling* (or *discharging*) hypotheses. In the introduction rule of $\Rightarrow$, the hypothesis $A$ has disappeared to continue the proof with $B$, but we store the information of $A$ without necessarily using it, so that it will be useful at some other time in the upper derivations of the tree.

A well-known example of Rule RAA in analysis would be: if we want to prove that some $l$ is the unique limit of a sequence $(u_n)$, we prove that the sequence admits at most one limit: we suppose there are two distinct limits $l$ and $l'$, and finally derive a contradiction.

#### 2.3.3.2 *Proofs in Higher-Order Logic*

Although first-order logics, we have in mind to represent predicates and symbol functions, enjoy good properties (countable set of valid formulae), they are however insufficient to express higher-order properties as would be found in elementary mathematics. For instance,

$$P\ x \to P\ x$$

will remain true, no matter what value is assigned to $x$. In other words, first-order logic would express this as

$$\forall x \quad P\ x \to P\ x$$

Still, whatever the interpretation is given to the predicate $P$, the formula remains valid. This cannot be expressed in first-order logic but we want to write

$$\forall P \quad \forall x \quad P\ x \to P\ x$$

This is why higher-order logic overcomes this limitation by extending the power of expressiveness and allows quantification over predicates and functions. For instance, usual induction schemes apply for any property $P$ on inductive structures as on natural integers,

$$\frac{\begin{array}{c}\left[P(n)\right] \\ \cdots \\ P(0) \qquad P(n+1)\end{array}}{P(n)} \tag{ind}$$

The logic system of Isabelle/HOL is based on natural deduction of Higher Order Logic (HOL) [Chu40] where terms come from the $\lambda$-calculus with polymorphism in ML-style.

### 2.3.4 *The Isabelle/HOL Proof Environment*

Isabelle/HOL [NWP02, NK14] is the specialization for HOL of the Isabelle generic system for implementing logical formalisms. It is a combination of logic and functional programming. It uses the Damas-Milner type system extended with type classes and integrates ND, while providing support for the usual constructs of functional programming, such as pattern-matching, datatypes, recursive functions and definitions, and records.

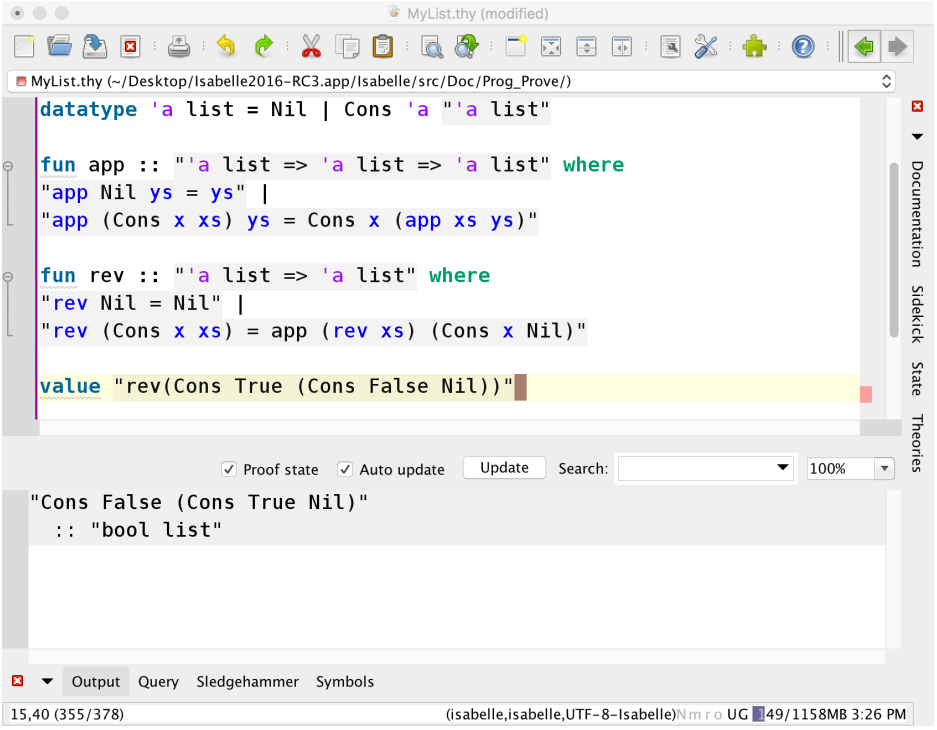



Figure 14: The jEdit IDE with datatypes in Isabelle/HOL

Figure 14 depicts the jEdit IDE that is used for developing theories in Isabelle/HOL. The keyword `datatype` creates a new sum type named `list` which has a type parameter `'a`, and two constructors `Nil` and `Cons`. They are similar to the usual definition of lists. `Nil` denotes the empty list, while `Cons` denotes the concatenation of a term of type `'a` with another list containing elements of the same type `'a`. Then two functions `app` and `rev` are inductively defined on the structure of the sum type `'a list`. The first function appends the elements of the right-component list to the left-component list by inductively unfolding the structure. The second function reverses the elements of the list. Finally, the keyword `value` executes a term by computing its normal form.

#### 2.3.4.1 *Brief Syntax*

Isabelle strongly relies on *conservative extensions*. This means that the addition of theories will not break provability of any term from the previous theory. Theories are decomposed into definitions and lemmas that are described by the following type system.

The type system of HOL consists of four sorts of types

- *base types*, also called constants, like `bool` for the type of truth values, `nat` for the type of natural integers, and `int` for the type of mathematical integers;
- *type constructors* such as `list` for the type of lists and `set` for the type of sets;
- *function types* written as $\Rightarrow$
- *types variables* written as `'a`, `'b`...

A term $t$ of type $\tau$ is written $t :: \tau$. Types can be left unspecified and Isabelle automatically infers when disambiguation is not necessary.

Similarly to functional programming *subject reduction* holds: if $f$ is a function (i.e., a $\lambda$-term) of type $\tau_1 \Rightarrow \tau_2$ and $t$ is a term of type $\tau_1$, then the application of both is a term $f\ t$ of type $\tau_2$. Likewise, the usual syntactic sugars of functional programming are available:

- *conditional value*, written as `if` $b$ `then` $t_1$ `else` $t_2$;
- *let-binding*, written as `let` $x = t$ `in` $u$;
- *pattern-matching* written as `case` $t$ `of` $\text{pat}_1 \Rightarrow t_1 \mid \ldots \mid \text{pat}_n \Rightarrow t_n$.

Logical formulae are terms of type `bool`. They contain the basic constants `True`, `False` and are closed under negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\longrightarrow$) and quantification ($\forall$, $\exists$). Above these constructs, Isabelle employs meta-implication ($\Longrightarrow$) to structure derivations. Let $A_0 \ldots A_n$ be logical formulae, a meta-term $A_0 \Longrightarrow \ldots \Longrightarrow A_{n-1} \Longrightarrow A_n$ is similar to the structure of ND for inference rules:

$$\frac{A_0 \quad \ldots \quad A_{n-1}}{A_n}$$

Finally, $\bigwedge$ denotes introduced variables in the proof context as found with discharged hypotheses in ND.

#### 2.3.4.2 *Statements and Proofs*

Proofs in Isabelle/HOL can be stated in two ways. To highlight the differences between both approaches, we first illustrate the classic `apply`-script style on the small theory shown on Figure 14.

`apply`-SCRIPT STYLE Proofs are sequences of *tactics* application. They consist of rule applications or proof methods. They are linear while proof trees are branching. Hence, they are applied on the proof tree on pre-order traversal. Due to these restrictions, they are unreadable and hardly maintainable.

The function `rev` entails the following theorem: reversing a reversed list yields the same list.

```
1 theorem rev_rev: "rev (rev l) = l"
```

Note that `l` appears here as a free variable, it is considered in Isabelle as universally quantified outside by default. The typical proof process for lists is structural induction (empty, then non-empty cases).

```
2 apply (induction l)
```

The current proof state now contains two goals to solve and the output window of Isabelle displays

```
proof (prove)
goal (2 subgoals):
1. rev (rev Nil) = Nil
2. ⋀ x1 l. rev (rev l) = l ⟹
              rev (rev (Cons x1 l)) = Cons x1 l
```

The first subgoal is the base case with the empty list `Nil`. The second one is the inductive case with the non-empty list and the induction hypothesis in assumption. Then the proof process continues by solving the first goal, potentially yielding even more subgoals. Finally, the proof ends whenever no subgoal is left. In this case, the subgoals are trivial enough to be solved by Isabelle's simplification engine and the proof ends with

```
3 apply (simp)                    solves the first subgoal
4 apply (simp)                    solves the second subgoal
5 done
```

ISAR STYLE To take advantage of the branching nature of proofs, the Isar language [Nip03] structures breakpoints to decompose a proof into manually defined goal statements, so that it is not necessary to "run" the proof to know the proof state. The proof designer adopts a forward reasoning and writes proofs as naturally as on a chalkboard.

In the style of Isar, subgoals have to be clearly stated by the proof designer. Compared to the Isabelle engine produced subgoal, a "manual" subgoal can be declared in a different order, or can contain renamed variables, as long as it can pass unification. If the proof is by induction, Isabelle recognizes the structure, and the proof directly reflects the idea of usual induction reasoning. A proof sketch of the previous theorem would be

```
proof (induction l)
  case Nil                                   corresponds to the base case
  then show ?case
    by simp
next
  case (Cons x1 l)                           corresponds to the inductive step
  then show ?case
    by simp
qed
```

Although Isar-style proofs may seem more verbose, they ensure a higher level of maintainability compared to other well-known proof assistants. By enjoying unification, structural changes in the proof engine that may lead to proof incompatibilities between prover versions that are more easily handled in this situation. Besides, any non-proficient Isabelle user can read and understand the underlying proof structure.

**Example 19** (Irrationality of $\sqrt{2}$)**.** To prove that $\sqrt{2}$ is not a rational number, we use a proof by contradiction. Assume it was rational, it would decompose as a fraction $\frac{m}{n}$ where $m$ and $n$ are coprime mathematical integers. This entails $m^2 = 2n^2$, and that 2 divides $m^2$, and consequently $m$. Thus, 2 divides $m$ and then again $n$, so both are not coprime, which conflicts with the assumption. Hence, $\sqrt{2}$ cannot be rational. This pattern is written in Isabelle/HOL as

Listing 1: Proof sketch of irrationality of $\sqrt{2}$ in Isabelle/HOL

```
theorem sqrt2_not_rational: "sqrt (real 2) ∉ ℚ"
proof
  let ?x = "sqrt (real 2)"
  assume "?x ∈ ℚ"
  then obtain m n :: nat where
       sqrt_rat: "|?x| = real m / real n"
    and lowest_terms: "coprime m n" by ...
  then have "real (m^2) = (?x)^2 * real (n^2)" by ...
  then have eq: "m^2 = 2 * n^2" by ...
  then have "2 divides m^2" by ...
  then have "2 divides m" by ...
  have "2 divides n" by ...
  with ‹2 divides m› have "2 divides gcd m n" by ...
  with lowest_terms have "2 divides 1" by ...
  then show False by ...
qed
```

#### 2.3.4.3 *Type Classes and Instanciation*

In algebraic structures, it is necessary to reason at higher order than sets. For instance, a semigroup $(S, \otimes)$ is the pair of a set $S$ and an inner connective $\otimes$ that satisfies the property of associativity. Likewise, Isabelle allows to specify that some type with some operations shall satisfy a specific algebraic structure: this is the idea of *type classes* [Haf13]. In this case, we write

```
class semigroup =
  fixes mult :: "'α ⇒ 'α ⇒ 'α" (infixl "⊗" 70)
  assumes assoc: "(x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)"
```

A non-refutable idea is that the set of mathematical integers (type `int`) with usual addition is a semigroup. By omitting proofs, we simply write

```
instantiation int :: semigroup
begin
  definition mult_int_def : "i ⊗ j = i + (j::int)"
  instance proof by ... qed
end
```

#### 2.3.4.4 *Semantic Subtyping*

Another feature of Isabelle/HOL is the introduction of new types with the `typedef` command to create *semantic subtypes* in the style of Gordon/HOL. From a type $\alpha$, it uses a set comprehension $S = \{x : \alpha \mid P\ x\}$ over type $\alpha$ to create another type $\beta$, and axiomatizes two isomorphisms

$$\begin{aligned} \mathrm{Abs} &: \alpha \to \beta \\ \mathrm{Rep} &: \beta \to \alpha \end{aligned}$$

These arrows satisfy the equations

$$\begin{aligned} &\forall x && \mathrm{Rep}\, x \in S \\ &\forall x && \mathrm{Abs}(\mathrm{Rep}\, x) = x \\ &\forall x \in S && \mathrm{Rep}(\mathrm{Abs}\, x) = x \end{aligned}$$

As an example, we can define a type `even_int` for even integers as follows. Note that the type definition comes with a proof obligation that states that the subtype admits an inhabitant. Otherwise a type defined by an empty-set would lead to inconsistencies.

```
typedef even_int = "{ n::int. n mod 2 = 0 }"
by (meson mem_Collect_eq mod_0)
```

# 3

## LANGUAGE CORE: TESL$\varepsilon$

*Less is more.*

Our aim is to design a declarative language for specifying timed behaviors of discrete events and their synchronization. In our setting, 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 $\mathbb{U}$, which is used for purely logical clocks where time does not progress.

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 clock `min` implies the clock `move`.

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 clock `min` ticks every minute, we can require that `min` clock *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.

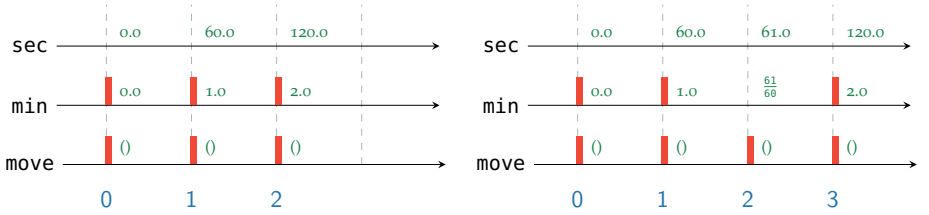
TAG RELATIONS. When all clocks are combined in a specification, each of them lives in its own "time island", with a potentially independent time scale. The purpose of tag relations is to link these different time scales. For instance, time runs 60 times as fast on clock `sec` as on clock `min`. This 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 the examples; this reduces the complexity of constraint solving to handling linear equation systems.

Here is a TESL specification for the examples above:

Listing 2: Specification of a clock watch in TESLε

```
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 0.0. 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 will be detailed further in Section 4.1.




(a) Returned by simulation engine

(b) Alternative possibility at instant 3

Figure 15: Two partially satisfying runs of the clock watch

We study a specification language that defines the *set* of possible execution traces or *runs* of a global system. In Figure 15 we present two of them:

- *Runs* are presented by ticks (solid red rectangles).
- They are timestamped with *tags* (small green numbers) on the time-scales of the clocks `sec`, `min` and `move`.
- Additionally, they are grouped in a sequence of synchronization *instants* (dashed vertical lines above blue numbers).

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 15b shows a run with an additional tick on `move`, which may correspond to a movement of the minute hand caused by manually setting the time on the watch.

The original TESL simulator only computes "minimal" runs, as first shown in Figure 15a, 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.

## 3.1 STRUCTURES FOR EXECUTION TRACES

More formally, we chose to model a run as a map from *clocks* to *event occurrences*. The latter consists of a boolean indicating the occurrence, and a time tag which gives the date of the occurrence in the time frame of the clock.

**Definition 20 (*Run*)**

Runs are defined as

| | |
|---|---|
| $\mathbb{K}$ | countable set of clocks $K_1, K_2, \ldots$ |
| $\mathbb{T}$ | domain of tags |
| $\Sigma = \mathbb{K} \to (\mathbb{B} \times \mathbb{T})$ | set of instants |
| $\Sigma^\infty = \mathbb{N} \to \Sigma$ | set of runs |
| $\rho_n$ | $n^{\text{th}}$ position (instant) in the run $\rho \in \Sigma^\infty$ |

For clarity purposes, we elide some conditions applied to tags:

ALGEBRAIC STRUCTURE. The tag set $\mathbb{T}$ is an abbreviation for a field together with a total order $\langle \mathbb{T}, +_{\mathbb{T}}, \times_{\mathbb{T}}, 0_{\mathbb{T}}, 1_{\mathbb{T}}, \leqslant_{\mathbb{T}} \rangle$;

MONOTONY. For a given run $\rho \in \Sigma^\infty$, the tags on any clock $K$ are non-decreasing with respect to the instant index $n$.

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

**Definition 21 (*Projections for Ticks and Tags*)**

| | |
|---|---|
| $\mathsf{ticks}(\sigma(K))$ | 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) |

**Example 22.** For instance in Figure 15a, where the tags are rationals for clocks `sec` and `min`: $\mathbb{T}_{\mathtt{sec}} = \mathbb{T}_{\mathtt{min}} = \mathbb{Q}$; while unit for clock `move`: $\mathbb{T}_{\mathtt{move}} = \mathbb{U}$. The run is noted $\rho$, we have

$$\mathsf{ticks}(\rho_0(\mathtt{sec})) = \mathsf{false} \qquad \mathsf{tag}(\rho_0(\mathtt{min})) = 0.0$$
$$\mathsf{ticks}(\rho_0(\mathtt{min})) = \mathsf{true} \qquad \mathsf{tag}(\rho_0(\mathtt{move})) = ()$$

## 3.2 SYNTAX

The first collection of operators of the TESL language consists of basic formulae that capture the essence of timed constraints along with synchronous causality. They are expressive enough to illustrate causality and time scale relation between events. Here is a grammar.

**Definition 23 (*Grammar of TESLε*)**

A TESLε formula $\Psi$ is given by

$$
\begin{array}{lcl}
\Psi & ::= & \langle atom \rangle \wedge \ldots \wedge \langle atom \rangle \\
\langle atom \rangle & ::= & \langle clock \rangle \texttt{ sporadic } \langle tag \rangle \texttt{ on } \langle clock \rangle \\
& | & \texttt{tag relation } \lfloor \langle clock \rangle, \langle clock \rangle \rfloor \in \langle relation \rangle \\
& | & \langle clock \rangle \texttt{ implies } \langle clock \rangle \\
& | & \langle clock \rangle \texttt{ time delayed by } \langle tag \rangle \texttt{ on } \langle clock \rangle \texttt{ implies } \langle clock \rangle
\end{array}
$$

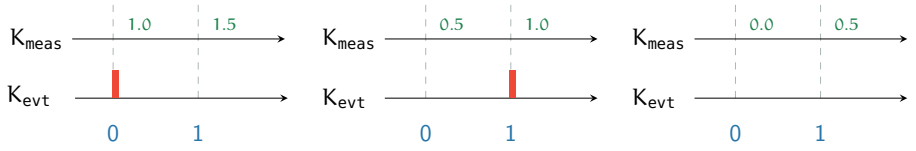
where $\langle clock \rangle \in \mathbb{K}$, $\langle tag \rangle \in \mathbb{T}$, and $\langle relation \rangle \subseteq \mathbb{T} \times \mathbb{T}$.

A specification $\Psi$ is a conjunction of atomic formulae that must be all satisfied. We give some intuition of the required behavior for each atomic formulae with examples that illustrate technical points that we detail further.

SPORADIC ON.

$$\mathsf{K}_{\mathtt{evt}} \texttt{ sporadic } \tau \texttt{ on } \mathsf{K}_{\mathtt{meas}}$$

specifies that some event will occur on the first clock $\mathsf{K}_{\mathtt{evt}}$ at timestamp (tag) $\tau$ measured on the second clock $\mathsf{K}_{\mathtt{meas}}$. Figure 16 shows three runs of length 2 that satisfy the statement where $\tau = 1.0$. Both Figure 16a and Figure 16b show the event occurrence depicted by a tick on clock $\mathsf{K}_{\mathtt{evt}}$ and a time tag of 1.0 on $\mathsf{K}_{\mathtt{meas}}$ instantaneously: these runs are said to be *satisfying*. On the other side Figure 16c shows a run when the time on clock $\mathsf{K}_{\mathtt{meas}}$ has not elapsed enough so that $\mathsf{K}_{\mathtt{evt}}$ could tick. Even if it does not show an event occurrence that would match with the statement, it can serve as a prefix for a future run where time would have progressed enough to reach the desired time tag. As the future part is not yet clear, this run is said to be *partially satisfying*




(a) Event occurrence with time 1.0 at instant 0

(b) Delay of then event occurrence to instant 1

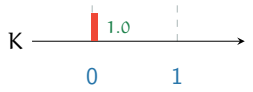
(c) Delay even further

Figure 16: Satisfying runs for the `sporadic on` atom where $\tau = 1.0$

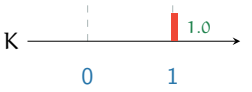
SPORADIC.

$$\mathsf{K} \texttt{ sporadic } \tau$$

is a syntactic sugar that mixes the event and measuring clocks. It is a direct synonym for K `sporadic` τ `on` K and admits the same way two satisfying runs as shown in Figure 17.




(a) Event occurrence with time 1.0 at instant 0



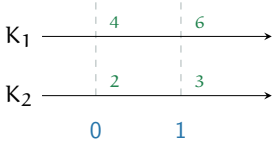

(b) Delay of then event occurrence to instant 1

Figure 17: Satisfying runs for the `sporadic` syntactic sugar

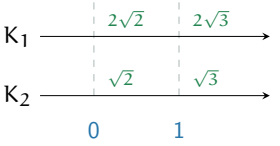
TAG RELATION.

$$\texttt{tag relation}\ \lfloor K_1\,,\ K_2 \rfloor\ \in R$$

gives a relation between the time frames of two clocks $K_1$ and $K_2$. The tags of the clocks must satisfy the relation at every instant. Such can be interpreted as a way to express how time flows faster or slower on some clock, compared to others. It enforces the tag-driven paradigm where time is precisely measured and stamped with a tag, but not a number of tick counts as in many discrete models. Figure 18 shows two satisfying runs of the formula with two different tag domains (integers and reals) and the arithmetic relation $R = ((x_1, x_2) \mapsto x_1 = 2x_2)$.




(a) Tag relation on integers $\mathbb{N}$




(b) Tag relation on reals $\mathbb{R}$

Figure 18: Satisfying runs for the `tag relation` formula where $R = ((x_1, x_2) \mapsto x_1 = 2x_2)$

IMPLIES.

$$K_1\ \texttt{implies}\ K_2$$

is a synchronous implication. It specifies in every instant, if the clock $K_1$ is ticking, then the second clock $K_2$ is ticking too. It is a stream-wise interpretation of the logical implication in propositional logic. Figure 19 shows a satisfying run where the master clock $K_1$ is false (idle) or true (ticking), and the consequence

on the slave clock $K_2$ at every instant. It is not possible for $K_1$ to tick and $K_2$ not to.

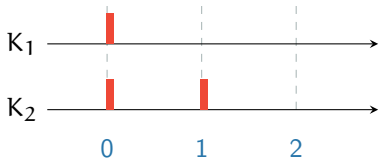



Figure 19: A satisfying run for the `implies` formula

TIME DELAYED.

$$K_{\texttt{master}}\ \texttt{time delayed by}\ \tau\ \texttt{on}\ K_{\texttt{meas}}\ \texttt{implies}\ K_{\texttt{slave}}$$

is a time-triggered implication. Whenever the first (master) clock $K_{\texttt{master}}$ is ticking, the time tag on the second (measuring) clock $K_{\texttt{meas}}$ is instantaneously measured and delayed by a duration of $\tau$ to obtain the date in a future instant at which the third (slave) clock $K_{\texttt{slave}}$ will have to tick. Figure 20 shows a satisfying run with two ticks on the clock $K_{\texttt{master}}$, whose instantaneous timestamp on $K_{\texttt{meas}}$ is delayed with a duration of 0.1 and triggers at that new timestamp a tick on $K_{\texttt{slave}}$.

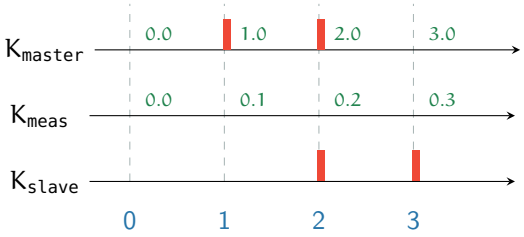



Figure 20: A satisfying run for the `time delayed` formula where $\tau = 0.1$

PERIODIC. Note on Figure 20 that the minimum number of ticks on $K_{\texttt{slave}}$ depends on the number of ticks on $K_{\texttt{master}}$. Conflating both clocks would create a loop where each tick requires another successor tick. This leads to the definition of *periodic* clocks by means of a syntactic sugar: K `periodic` $\tau$ means K `time delayed by` $\tau$ `on` K `implies` K. It admits a partially satisfying run given by Figure 21 where $\tau = 0.1$. As every tick requires a successor one, the fourth tick requires another fifth, and so on. Hence, this run cannot fully satisfy the formula. Consequently, its only satisfying runs are infinite.

*Remark* 24. Following the previous remark, we observe that satisfying runs of TESL can be either finite or infinite. Nevertheless, according to our experiments,

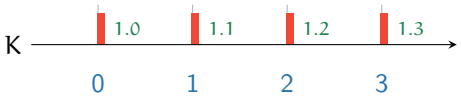



Figure 21: A satisfying run for the `periodic` formula

we believe that the infinite satisfying runs can be folded and symbolically and finitely described under some conditions. A periodicity of range modulo renaming of timestamps with respect to some relation has been observed each time. This idea will be explored in future work.

### 3.2.1 *The Radiotherapy Machine Example*

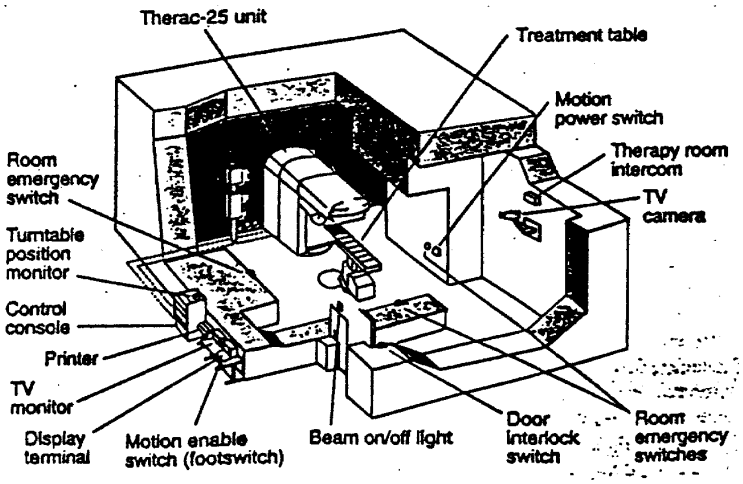



Figure 22: The Therac-25 radiotherapy machine [LT93]

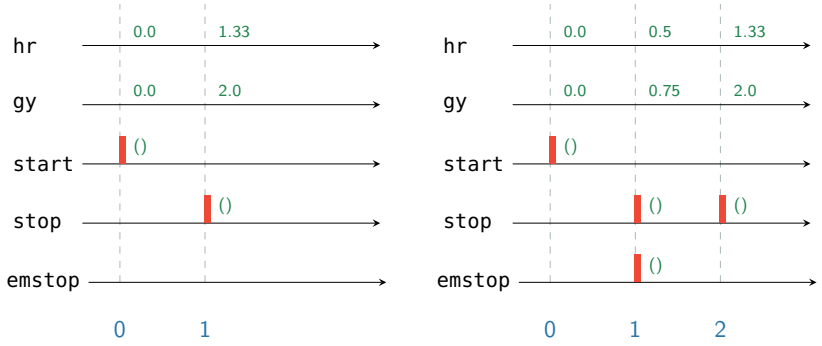
This small core language is able to express needs for physical computations and event-driven behaviors. We are interested in modeling the simple behavior of a radiotherapy machine used in cancer treatment. The patient has a prescription of 2 Gy of radiation in low-dose-rate of 1.5 $Gy.h^{-1}$. Here is a TESL$\varepsilon$ specification expressing the case in Listing 3:

Listing 3: Specification of a radiotherapy machine in TESL$\varepsilon$

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

```
8 emstop implies stop
```

Lines 1 to 5 declare clocks `hr` and `gy` with rational tags, and clocks `start`, `stop` and `emstop` with the unit tag. The constraint `sporadic` enforces a tick on `start`. Line 6 specifies that time on `hr` flows 1.5 times as fast as on `gy`. Line 7 specifies that each time clock `start` ticks, clock `stop` will tick after a delay of `2.0` measured on the time scale of clock `gy`. Line 8 requires that each time the `emstop` clock ticks, the `stop` clock instantaneously ticks as well.




(a) Normal situation (b) Emergency stop

Figure 23: Two partially satisfying behaviors for the radiotherapy machine

Two behaviors are illustrated in Figure 23. They show possible execution traces or *runs* satisfying the TESL specification. Likewise, a run consists of a sequence of synchronization *instants* (dashed gray lines). Each of them contains *ticks* (solid red rectangles) timestamped with *tags* (small numbers) on the time-scales of the clocks `hr`, `gy`, `start`, `stop` and `emstop`.

The original TESL simulator only computes "minimal" runs, as depicted by Figure 23a, which corresponds to a deterministic interpretation creating only mandatory event occurrences. On the other side, Figure 23b shows a behavior with an additional tick on clock `emstop` when the time on clock `hr` is 0.5, which may correspond to a manual push on an emergency stop button. Needless to say that the specification admits an infinite number of satisfying runs.

### 3.2.2 *The Power Window Example*

Recall the example of the car power window from Section 2.2.2. It consists of four subsystems: a button, a Timed Finite State Machine (TFSM), a SDF model of the electromechanical servo, and a Discrete Events (DE) model of the Controller Area Network (CAN) bus, which interconnects the three latter subsystems. Let us only consider raising up the window.

1. The command given by the button can only by pulled up and released, modeled by the clocks `btn_up` and `btn_neutral`.

2. From the controller side, clocks `up` and `stop` both stand for the input events of the TFSM ; and clock `power` for the output event interpreted as sending a power command to the electromechanical servo.

3. The electromechanical servo has an input event denoted by the clock `update_power`, corresponding to an update of the power to deliver to the motor engine. As it is modeled by a SDF, the event can only be considered when it reacts to compute the next state occurring every 50 ms and modeled by a input event of clock `react`

Listing 4: Specification of a power window in TESL$\varepsilon$

```
1  unit-clock btn_up       // the button is pulled up
2  unit-clock btn_neutral  // the button is released
3  unit-clock up           // the TFSM receives an up event
4  unit-clock stop         // the TFSM receives a stop event
5  unit-clock power        // the TFSM produces a power event
6  unit-clock update_power // the SDF model gets a new power command
7  unit-clock react        // the SDF model reacts to its inputs
8  rational-clock realtime // real-time in seconds
9  rational-clock bus      // time scale of the CAN bus
10
11 // The transmission delay on the CAN bus is 2 ms
12 tag relation realtime = 0.002 * bus
13 btn_up time delayed by 1.0 on bus implies up
14 btn_neutral time delayed by 1.0 on bus implies stop
15
16 // When the TFSM receives an input, it updates its output
       instantaneously
17 up implies power
18 stop implies power
19
20 // The transmission delay on the CAN bus is 2 ms
21 power time delayed by 1.0 on bus implies update_power
22
23 // The window must react every 50ms (periodic clock)
24 react time delayed by 0.05 on realtime implies react
```

This specification ignores the values that are sent over the bus, it specifies only when things happen since its goal is to coordinate the behaviors of the subsystems. Lines 1 to 7 declare the clocks that compose the interface of the subsystems for the architectural glue, as explained on Figure 3. The `unit-clock` keyword simply restricts the domain of the time stamps of these clocks. Lines 8 and 9 declare chronometric clocks used to measure elapsed time on the CAN bus and in the real world. Their time domains are restricted to rationals. Line 12 specifies that when 1 unit of time elapses on the bus clock, 0.002 s elapses on the real time clock, which means that time is measured in units of 2 ms on the

bus clock. Lines 13 and 14 specify that when the button is pulled up or released, the TFSM receives its `up` or `stop` input event 1 unit of time later, measured in the time frame of the bus clock (2 ms in real time). Lines 17 and 18 specify that the state machine reacts instantaneously to its inputs by producing its `power` output event. Line 21 specifies the transmission delay on the bus between the state machine and the SDF subsystem. Last, line 24 specifies that the reaction of the SDF subsystem is periodic, because it implies itself with a time delay of 50 ms.

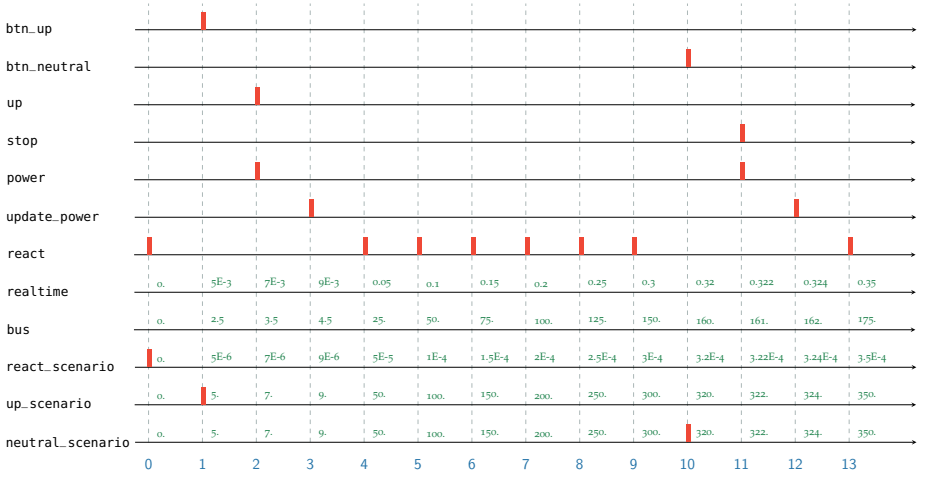



Figure 24: A satisfying run for the power window specification

Figure 24 depicts a satisfying run. From a physical interpretation, let us set our time reference on the `realtime` clock. The user pulls the button up (clock `btn_up`) at 5ms. The controller receives this information (clock `up`) at 7ms due to the transmission delay on the CAN bus, and immediately sets the power for the window motor (clock `power`). Then, the mechanical part receives the command at 9ms (clock `update_power`). The next tick of the periodic `react` clock occurs at 50ms, which is the time at which the new value of the power is taken into account and the window starts moving up. At 320ms, the user releases the button, which switches back to neutral (clock `btn_neutral`). With the transmission delay between the button and the controller, then again between the controller and the mechanical servo, the new value of the power is updated at 324ms. The next reaction of the window (clock `react`) occurs at 350ms, which is the time at which the window stops moving up. The additional clocks `react_scenario`, `up_scenario` and `neutral_scenario` are used to describe the user interface simulation scenario.

## 3.3 FORMAL SEMANTICS

This section aims at defining formal semantics that describe the previous formulae, which we call *denotational* (Section 3.3.1) and *operational* (Section 3.3.2).

### 3.3.1 *Denotational Semantics*

A denotational semantics is a mathematical description of a program, mainly based on ordered set theory, similarly to Scott and Strachey [Sto77]. It is usually expressed as a mathematical function, so-called *interpretation*, which denotes every program into a set of satisfying states or traces. Moreover, one of the main required properties is *compositionality*: there is a morphism from program composition to the denoted space.

To describe all satisfying runs of a TESL specification, we give them a mathematical description in Definition 25, which we call *denotational semantics*. This captures precisely the set of runs conforming to a specification with no regards to the order of formulae. Indeed, this is directly reflected by the interpretation of formula conjunction as run space intersection. More precisely, a run $\rho$ satisfies:

- a `sporadic on` atom, as previously but when clock $\mathsf{K}_{\mathtt{evt}}$ is ticking and the tag on $\mathsf{K}_{\mathtt{meas}}$ is $\tau$ at the same instant;
- a `tag relation` atom, if for every index $n$, the arithmetic relation holds;
- an `implies` atom, when for every index $n$, if clock $\mathsf{K}_1$ is ticking then $\mathsf{K}_2$ is ticking too at the same instant;
- a `time delayed` atom, whenever a clock $\mathsf{K}_{\mathsf{master}}$ is ticking, there exists in the future instants a tick on $\mathsf{K}_{\mathsf{slave}}$ with a tag distance of $\delta\tau$ on the measuring clock $\mathsf{K}_{\mathsf{meas}}$.

**Definition 25 (*Interpretation of TESLε formulae*)**

The denotational semantics of TESL formulas is given inductively on formulae as sets of runs.

$$[\![\psi_0 \wedge \ldots \wedge \psi_k]\!]_{\text{TESL}} \stackrel{\text{def}}{=} [\![\psi_0]\!]_{\text{TESL}} \cap \ldots \cap [\![\psi_k]\!]_{\text{TESL}}$$

$$[\![K_{evt} \texttt{ sporadic } \tau \texttt{ on } K_{meas}]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \exists n \in \mathbb{N} \quad \text{ticks}(\rho_n(K_{evt})) \text{ is true} \quad \text{and} \quad \text{tag}(\rho_n(K_{meas})) = \tau\}$$

$$[\![\texttt{tag relation } \lfloor K_1, K_2 \rfloor \in R]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \in \mathbb{N} \quad \text{tag}(\rho_n(K_1)) \text{ and } \text{tag}(\rho_n(K_2)) \text{ are related by } R\}$$

$$[\![K_1 \texttt{ implies } K_2]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \in \mathbb{N} \quad \text{ticks}(\rho_n(K_1)) \quad \text{implies} \quad \text{ticks}(\rho_n(K_2))\}$$

$$[\![K_{master} \texttt{ time delayed by } \delta\tau \texttt{ on } K_{meas} \texttt{ implies } K_{slave}]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \in \mathbb{N} \quad \text{ticks}(\rho_n(K_{master})) \text{ implies } \exists m \geqslant n \quad \text{ticks}(\rho_m(K_{slave})) \text{ and } \text{tag}(\rho_m(K_{meas})) = \text{tag}(\rho_n(K_{meas})) + \delta\tau\}$$

Usual properties of associativity, commutativity, idempotency and neutrality of TESL conjunction are preserved under set intersection ($\varnothing$ is a TESL specification which denotes the empty conjunction).

**Lemma 26 (*Associativity, Commutativity, Idempotence and Neutrality*)**

*Let $\Psi_1, \Psi_2, \Psi_3$ be TESLε formulae. We have*

$$[\![(\Psi_1 \wedge \Psi_2) \wedge \Psi_3]\!]_{\text{TESL}} = [\![\Psi_1 \wedge (\Psi_2 \wedge \Psi_3)]\!]_{\text{TESL}} \quad \text{(associativity)}$$

$$[\![\Psi_1 \wedge \Psi_2]\!]_{\text{TESL}} = [\![\Psi_2 \wedge \Psi_1]\!]_{\text{TESL}} \quad \text{(commutativity)}$$

$$[\![\Psi_1 \wedge \Psi_1]\!]_{\text{TESL}} = [\![\Psi_1]\!]_{\text{TESL}} \quad \text{(idempotency)}$$

$$[\![\varnothing \wedge \Psi_1]\!]_{\text{TESL}} = [\![\Psi_1]\!]_{\text{TESL}} \quad \text{(neutrality)}$$

**Proof:** By set-theoretical reasoning and usual properties of $\cap$. □

*Remark 27.* While investigating CCSL which is a close language to TESL, we observed that the mathematical structure that lies under the denotational semantics proposed by [DAG14] suggests the usage of indexes for ticks and precedence relation. This means that every tick on a clock lives as an existing entity and shall be additionally annotated with an incremental index. This is different from our approach where ticks do not exist themselves, but are a reflection of event occurrence modeled as a boolean object. Our indexes are attached to

instants, instead of ticks. This abstains from defining a notion of coincidence as an instant is by its essence a coincidence instant. For these reasons, we believe our approaches have made semantics formalization simpler and suited for mechanization.

### 3.3.2 *Operational Semantics*

Defining an operational semantics allows to give a meaning to a program as a sequence of machine configurations. The goal is to consider a program as a transition system or an abstract machine, where evaluation steps are explicited to follow an accurate order and a precise granularity of operations. The literature depicts this approach for various paradigms:

- imperative programming (IMP, [Win93]),
- functional programming (λ-calculus, [Chu32]),
- object-oriented programming (σ-calculus, [AC96]).

Likewise, our concern is to provide a semantics that considers TESL$\varepsilon$ as a language that can be evaluated to derive possible satisfying runs of a given specification. As such, we have defined a semantics that non-deterministically executes a specification where branches may lead to different – but still possibly satisfying – behaviors. The idea we follow is that a TESL$\varepsilon$ formula can be consumed by producing smaller primitives. In this setting, a configuration is composed of three parts which we informally call: *past*, *present* and *future*. This semantics behaves by unfolding past-present-future where future constraints are moved into present constraints, and present constraints into past constraints (as primitives) by means of reduction rules.

#### 3.3.2.1 *Primitives for Run Contexts*

*Symbolic runs* are defined by *run contexts* constructed from a set of *run primitives* introduced below. Run contexts may contain variables that can be arbitrarily instantiated; instances of symbolic runs with ground terms are called *concrete runs*. We define *tag variables* in Definition 28, and then *primitives* in Definition 29 that are used to describe prefixes of satisfying runs, as filled in run contexts. Note that compared to TESL atomic formulae, they now deal with fixed instant indexes.

**Definition 28 (*Tag Variables*)**

The set of *tag variables* $\mathbb{V}$ consists of symbols $\mathrm{tvar}_n^K$ where $n \in \mathbb{N}$ and $K \in \mathbb{K}$. $\mathrm{tvar}_n^K$ reads as the *symbolic time stamp on clock* $K$ *at instant* $n$.

**Definition 29 (*Run Primitives*)**

A *run primitive* $\gamma \in \Gamma$ is a constraint symbol of the following kind:

- $K \Uparrow_n$ forces clock $K$ to tick at instant index $n$ ;
- $K \not\Uparrow_n$ forces clock $K$ to not tick (to be absent) at instant index $n$ ;
- $K \Downarrow_n x$ forces clock $K$ to have timestamp (tag constant or variable) $x$ at instant index $n$ ;
- $\lfloor \mathrm{tvar}^{K_1}_{n_1}, \mathrm{tvar}^{K_2}_{n_2} \rfloor \in R$ enforces the arithmetic relation $R$ between the variables $\mathrm{tvar}^{K_1}_{n_1}$ and $\mathrm{tvar}^{K_2}_{n_2}$.

They are interpreted by $[\![\_]\!]_{\mathsf{prim}}$ as:

$$
\begin{array}{lcl}
[\![\{\gamma_0 \,;\, \ldots \,;\, \gamma_k\}]\!]_{\mathsf{prim}} & \stackrel{\text{def}}{=} & [\![\gamma_0]\!]_{\mathsf{prim}} \cap \ldots \cap [\![\gamma_k]\!]_{\mathsf{prim}} \\
[\![K \Uparrow_n]\!]_{\mathsf{prim}} & \stackrel{\text{def}}{=} & \{\rho \in \Sigma^\infty \mid \mathsf{ticks}(\rho_n(K)) \text{ is true}\} \\
[\![K \not\Uparrow_n]\!]_{\mathsf{prim}} & \stackrel{\text{def}}{=} & \{\rho \in \Sigma^\infty \mid \mathsf{ticks}(\rho_n(K)) \text{ is false}\} \\
[\![K \Downarrow_n x]\!]_{\mathsf{prim}} & \stackrel{\text{def}}{=} & \left\{ \rho \in \Sigma^\infty \;\middle|\; \mathsf{tag}(\rho_n(K)) = \begin{cases} x & \text{if } x \in \mathbb{T} \\ \mathsf{tag}(\rho_{n'}(K')) & \text{if } x = \mathrm{tvar}^{K'}_{n'} \in \mathbb{V} \end{cases} \right\} \\
[\![\lfloor \mathrm{tvar}^{K_1}_{n_1}, \mathrm{tvar}^{K_2}_{n_2} \rfloor \in R]\!]_{\mathsf{prim}} & \stackrel{\text{def}}{=} & \{\rho \in \Sigma^\infty \mid \mathsf{tag}(\rho_{n_1}(K_1)) \text{ and } \mathsf{tag}(\rho_{n_2}(K_2))) \text{ are related by } R\}
\end{array}
$$

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 30 (*Decidability of Run Contexts*)**

*If tag relations are affine relations, then the consistency of a run context $\Gamma$ is decidable. That is we can decide whether*

$$[\![\Gamma]\!]_{\mathsf{prim}} \quad \neq \quad \varnothing.$$

**Proof:** The affine relations as found in the original TESL language 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 remains decidable. □

Without loss of generality, any class of decidable arithmetic relations preserves decidability of consistency for run contexts. Our model is agnostic enough that the combination of SAT with any decidable arithmetic problem remains decidable again.

This lemma is particularly important as it allows us to construct consistent runs from TESL specifications. The consequence is that it is possible to decide whether there exists a run prefix of fixed length that would partially satisfy the specification.

#### 3.3.2.2 *Configurations of the Execution Process*

We now define the machinery for constructing symbolic runs. We chose to treat TESL as a logic of resources. 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:

**Definition 31 (*Configuration*)**

A *configuration* is a tuple $\Gamma \models_n \Psi \triangleright \Phi$, where

- $n$ is the current simulation instant index ;
- $\Gamma$ is the run context containing primitives describing the "past" ;
- $\Psi$ is the TESL-formula to satisfy in the "present" ;
- $\Phi$ is the TESL-formula to satisfy in the "future" of the process.

#### 3.3.2.3 *Execution Rules*

The operational semantics can be seen as an abstract machine, in which a configuration corresponds to an abstract state comprising the past ($\Gamma$), present ($\Psi$) and future ($\Phi$) of the symbolic run under construction. The execution of this kind of abstract machine is two-fold:

1. Moving parts from the future to the present (introduction)
2. Consuming the present to produce the past (elimination)

The introduction rule (Definition 32) initializes a new instant to construct by incrementing the index counter and moving future parts into the present. On the other side, the elimination rules (Definition 33) produce past primitives by consuming present formulae. Iterating the application of these rules will produce runs by adding more and more constraints in $\Gamma$, thus constructing a run prefix.

**Definition 32 (*Introduction Rule* $\rightarrow_i$)**

The relation $\rightarrow_i$ is the smallest relation satisfying:

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

**Definition 33 (*Elimination Rules* $\rightarrow_e$)**

The relation $\rightarrow_e$ is the smallest relation satisfying the following rules, where `time delayed by` is abbreviated as `tdby` to save space.

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2) \triangleright \Phi \qquad (\text{sporadic} - \text{on}_{e1})$$
$$\rightarrow_e \Gamma \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2)$$
$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ sporadic } \tau \texttt{ on } K_2) \triangleright \Phi \qquad (\text{sporadic} - \text{on}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{l} K_1 \Uparrow_n \\ K_2 \Downarrow_n \tau \end{array} \right\} \models_n \Psi \triangleright \Phi$$
$$\Gamma \models_n \Psi \wedge (\texttt{tag relation } \lfloor K_1, K_2 \rfloor \in R) \triangleright \Phi \qquad (\text{tagrel}_e)$$
$$\rightarrow_e \Gamma \cup \left\{ \lfloor \text{tvar}_n^{K_1}, \text{tvar}_n^{K_2} \rfloor \in R \right\} \models_n \Psi \triangleright \Phi \wedge (\texttt{tag relation } \lfloor K_1, K_2 \rfloor \in R)$$
$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ implies } K_2) \triangleright \Phi \qquad (\text{implies}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{ K_1 \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ implies } K_2)$$
$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ implies } K_2) \triangleright \Phi \qquad (\text{implies}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{l} K_1 \Uparrow_n \\ K_2 \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ implies } K_2)$$
$$\Gamma \models_n \Psi \wedge (K_{\text{master}} \texttt{ tdby } \delta t \texttt{ on } K_{\text{meas}} \texttt{ implies } K_{\text{slave}}) \triangleright \Phi \qquad (\text{time} - \text{delayed}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{ K_{\text{master}} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{\text{master}} \texttt{ tdby } \delta t \texttt{ on } K_{\text{meas}} \texttt{ implies } K_{\text{slave}})$$
$$\Gamma \models_n \Psi \wedge (K_{\text{master}} \texttt{ tdby } \delta t \texttt{ on } K_{\text{meas}} \texttt{ implies } K_{\text{slave}}) \triangleright \Phi \qquad (\text{time} - \text{delayed}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ K_{\text{master}} \Uparrow_n \right\} \models_n \Psi \wedge (K_{\text{slave}} \texttt{ sporadic } (\text{tvar}_n^{K_{\text{meas}}} + \delta t) \texttt{ on } K_{\text{meas}})$$
$$\triangleright \quad \Phi \wedge (K_{\text{master}} \texttt{ tdby } \delta t \texttt{ on } K_{\text{meas}} \texttt{ implies } K_{\text{slave}})$$

These rules are used to completely consume the present in order to be allowed to progress to the next instant. In particular, it is necessary to repeat the elimination process as much as necessary to empty the present formula. Here are the different possibilities to eliminate specification constraints from the present.

- $K_1$ `sporadic` $\tau$ `on` $K_2$: similar to the previous one, but with the time stamp constraint on $K_2$ (Rule $\text{sporadic} - \text{on}_{e1}$ and Rule $\text{sporadic} - \text{on}_{e2}$);
- `tag relation` $\lfloor K_1, K_2 \rfloor \in R$: the corresponding primitive is added to instantaneously constraint the timescales of clocks $K_1$ and $K_2$, and the formula is put into the future since it has to be satisfied at every instant (Rule $\text{tagrel}_e$);
- $K_1$ `implies` $K_2$: either clock $K_1$ is not ticking (Rule $\text{implies}_{e1}$), or both clocks $K_1$ and $K_2$ are instantaneously ticking (Rule $\text{implies}_{e2}$). In both cases, the formula is copied into the future as it has to be satisfied at every instant;
- $K_{\text{master}}$ `time delayed by` $\delta t$ `on` $K_{\text{meas}}$ `implies` $K_{\text{slave}}$: either master clock $K_{\text{master}}$ is not ticking and we only copy the formula into the future because there is no tick to delay (Rule $\text{time} - \text{delayed}_{e1}$); or it is ticking and we need to force a tick on slave clock $K_{\text{slave}}$ when the time on $K_{\text{meas}}$ reaches $\text{tvar}^n_{K_{\text{meas}}} + \delta t$, which is the current tag on measuring clock $K_{\text{meas}}$

delayed by duration $\delta t$, and lastly the formula is copied into the future (Rule $\mathsf{time-delayed}_{e2}$).

### 3.3.3 *Simulation Steps*

The goal of consuming formulae by means of reduction rules defined previously is to compute the current instant (or step) of simulation. To ensure this computation, we give a property that ensures that computing one step necessarily terminates. In particular, this is ensured by stating that the relation $\to_e$ is well-founded.

**Proposition 34 (*Local Termination*)**

*The relation $\to_e$ is well-founded.*

**Proof:** All the elimination rules seem to strictly decrease the number of formulas in the "present" part of the state, and a state with an empty "present" part is in normal form with respect to $\to_e$. The fined-grained proof exhibits an integer interpretation that values more importantly the future part, and shows that the relation is well-founded. □

A *reduction step* is an introduction or an elimination,

**Definition 35 (*Reduction* $\to$)**

We define $\to \stackrel{\text{def}}{=} \to_i \cup \to_e$.

*Remark* 36. If a sequence of reductions under $\to$ is infinite, Proposition 34 implies that this sequence contains an infinite number of introduction steps. In other words, reducing a configuration eventually increases the instant index: time flows. This will be echoed in Theorem 62.

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

1. Initializing a new instant with reduction $\to_i$ (uniquely defined by the Rule $\mathsf{instant}_i$);
2. a) Repeating elimination rules given in $\to_e$.
   b) Until irreducibility, i.e. when $\Psi = \varnothing$.

**Definition 37 (*Simulation Step* $\to_{\lightning}$)**

A simulation step is defined as a reduction rule

$$\to_{\lightning} := \{(\Gamma_1 \models_n \varnothing \triangleright \Phi_1) \to_i \cdot \to_e^* (\Gamma_2 \models_{n+1} \varnothing \triangleright \Phi_2) \mid \Gamma_1 \text{ and } \Gamma_2 \text{ are consistent}\} \quad \text{(simulation)}$$

where $\cdot$ is the composition of relations, and $\to_e^*$ the reflexive transitive closure of $\to_e$.

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 $\mathsf{K} \Uparrow_n$ to a context that already contains $\mathsf{K} \not\Uparrow_n$.

Following the specification given as an example in Listing 2 (denoted as $\Phi_{\texttt{handwatch}}$), we illustrate the use of our operational rules in Figure 25 where `tag relation` is abbreviated as `trel`. We start with an empty symbolic run and show the two first simulation steps on the left hand-side. Then, focus is set on the first step and exhibits 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 ($\mathsf{sporadic-on}_{e2}$, $\mathsf{tagrel}_e$, $\mathsf{implies}_{e2}$, $\mathsf{time-delayed}_{e1}$), until irreducibility.

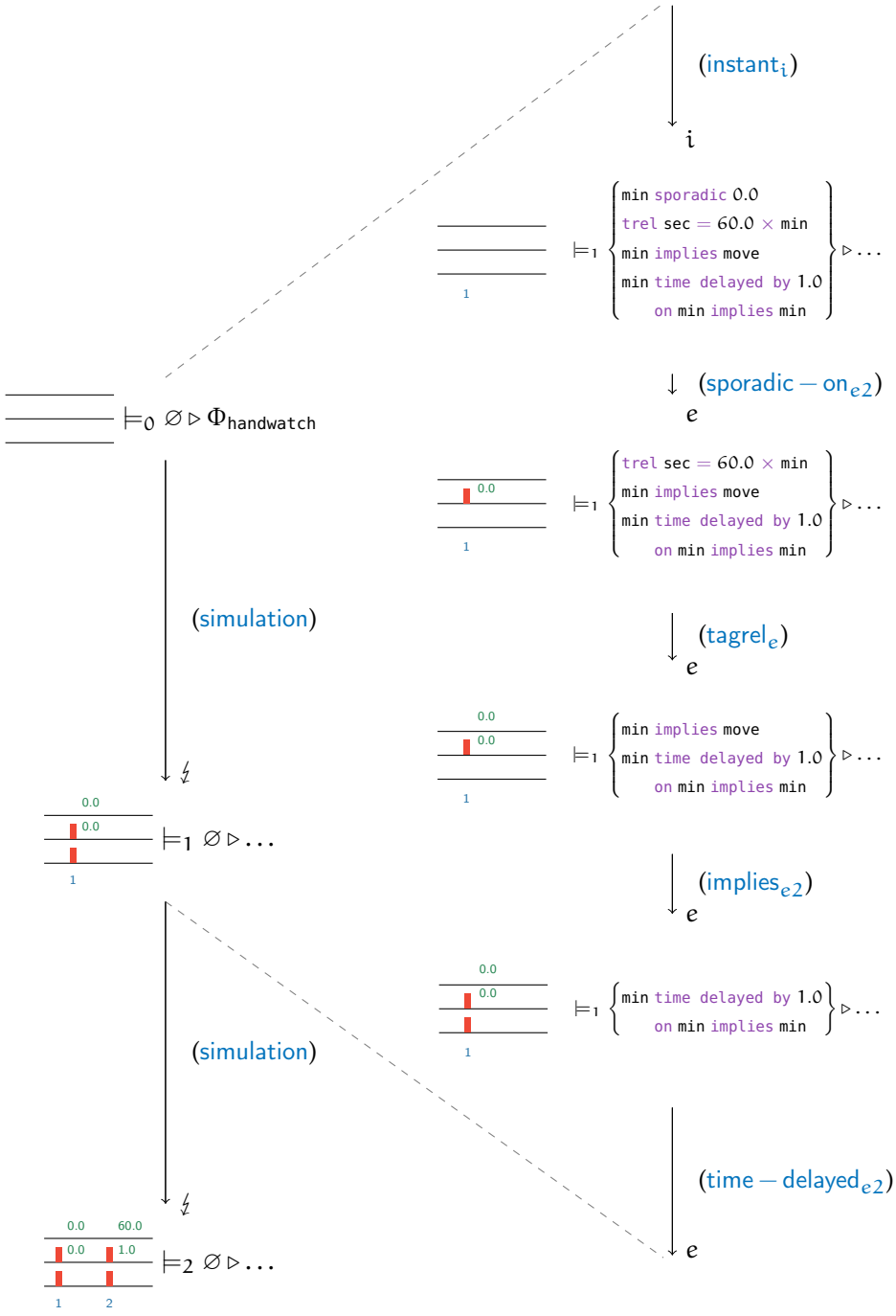



Figure 25: Detail of the reduction steps of the operational semantics

# 4

# LANGUAGE WITH ASYNCHRONOUS EXTENSIONS: TESL*

Our study investigated even further than the operators of the original TESL. Only the synchronous constraints of CCSL were preserved in the original TESL language in order to provide a constructive and concrete way to solve specifications with time tags. Indeed, the past-asynchronous formulae of CCSL combined with the support for tags in TESL were initially incompatible, as past-asynchrony requires to backtrack and compute a non-concretizable time tag in the past. We observed that the abstraction of tags in TESL$\varepsilon$ with a symbolic approach overcomes very well the previous limitations of the TESL solver. We propose in this chapter an extension which we call TESL*.

## 4.1 SYNTAX

The extended language consists of TESL$\varepsilon$ along with two asynchronous operators and an additional synchronous operator: (`strictly` or `weakly`) `precedes`, `kills` and `implies not`. From the grammar of TESL$\varepsilon$ in Chapter 3, we extend and add these symbols to the atomic formulae

**Definition 38 (*Grammar of TESL**)**

A TESL* formula $\Psi$ is given by the addition to TESL$\varepsilon$ of the following

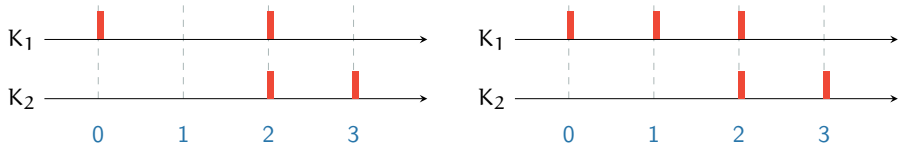
$$
\begin{array}{lll}
\langle atom \rangle & ::= & \ldots \\
& | & \langle clock \rangle \texttt{ strictly precedes } \langle clock \rangle \\
& | & \langle clock \rangle \texttt{ weakly precedes } \langle clock \rangle \\
& | & \langle clock \rangle \texttt{ kills } \langle clock \rangle \\
& | & \langle clock \rangle \texttt{ implies not } \langle clock \rangle
\end{array}
$$

In the next paragraphs, we detail and illustrate these operators.

### STRICTLY PRECEDES.

$$K_1 \texttt{ strictly precedes } K_2$$

is a formula that specifies that a tick on $K_2$ injects on a tick of $K_1$ on past instants. An event occurrence shown on $K_2$ admits a necessary and unique consequence of an event occurrence on $K_1$. This kind of causality is slightly different from causality as understood in implication, it considers that events on $K_2$ necessarily admit a parent event: we call it *necessary causality*. Figure 26a shows a satisfying run where each tick is injected on a tick of the previous instants: the first tick on $K_2$ points to the one in instant 0 on $K_1$, and the second tick on $K_2$ points to the one in instant 2 on $K_1$. Figure 26b has yet an additional tick which does not change the property as there are enough ticks on $K_1$ to "fulfil the needs" of $K_2$.




(a) Two events in $K_2$ depend on exactly two others on $K_1$ strictly asynchronously

(b) Additional ticks on $K_1$ still preserve the dependence property

Figure 26: Satisfying runs for the `strictly precedes` formula

WEAKLY PRECEDES.

$$K_1 \texttt{ weakly precedes } K_2$$

is similar to the strict precedence. It is a weakened version of the latter, and allows some tick on $K_2$ to inject on a tick on $K_1$ potentially instantaneously. In Figure 27, the first tick of $K_2$ depends on the tick at instant 0 on $K_1$, while the second one instantaneously depends on the tick at instant 3 on $K_1$.

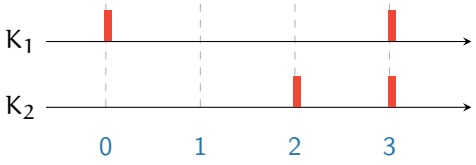



Figure 27: A satisfying run for the `weakly precedes` formula

KILLS.

$$K_1 \texttt{ kills } K_2$$

is a formula that specifies that whenever $K_1$ ticks, $K_2$ shall remain absent from then on. This clock is said to be *dead*. In Figure 28, $K_1$ ticks and prevents $K_2$ from ticking from then on.

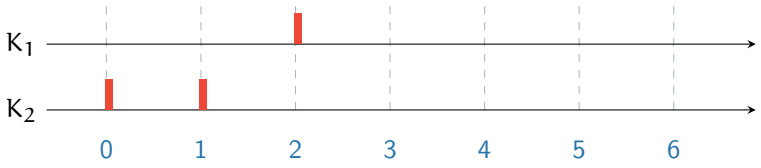



Figure 28: A satisfying run for the `kills` formula

IMPLIES NOT.

$$K_1 \texttt{ implies not } K_2$$

is a formula that specifies that whenever $K_1$ ticks, $K_2$ shall instantaneously remain absent. In Figure 29, $K_1$ ticks and prevents $K_2$ from instantaneously ticking.

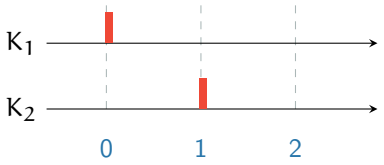



Figure 29: A satisfying run for the `implies not` statement

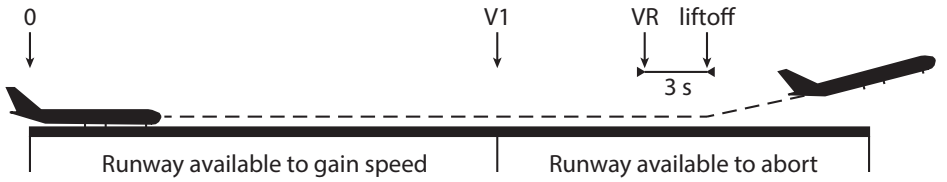
*Remark* 39 (Equivalence between CCSL and TESL*). As stated at the beginning of the section, difficulties related to the combination of past-asynchrony and time tag computation have been overcome by symbolic approaches. Here is a translation that gives the correspondence between CCSL and TESL* formulae.

| CCSL FORMULA | EQUIVALENT TESL* FORMULA |
|---|---|
| $K_1$ coincides with $K_2$ | $K_1$ implies $K_2 \wedge K_2$ implies $K_1$ |
| $K_1$ in exclusion with $K_2$ | $K_1$ implies not $K_2 \wedge K_2$ implies not $K_1$ |
| $K_1$ is subclock of $K_2$ | $K_2$ implies $K_1$ |
| $K_1$ precedes $K_2$ | $K_1$ strictly precedes $K_2$ |
| $K_1$ depends on $K_2$ | $K_1$ weakly precedes $K_2$ |

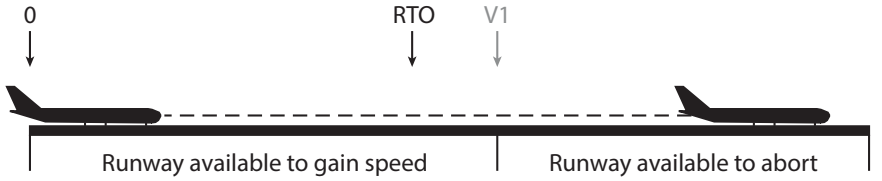
### 4.1.1 *The Airplane Takeoff Example*

Let us introduce our extensions of the language by modeling the takeoff procedure of a public transportation airplane as illustrated by Figure 30. The protocol revolves around airspeed thresholds giving obligations and restrictions for pilots. The first is V1 (decision speed), and gives a speed threshold from which pilots are not allowed to reject the takeoff procedure (RTO), otherwise risking braking without enough runway remaining. Above this value, takeoff

is mandatory. At VR (rotate speed) the pilot commands the aircraft to rotate on its main wheels, and the aircraft finally lifts off after about 3 seconds.




(a) Go situation




(b) No Go situation

Figure 30: Takeoff procedure according to the certification standards

Consider a Boeing 737-800 with flaps 1 rolling on wet runway and gaining speed on a uniform acceleration of 1.875 $m.s^{-2}$. The precomputed speed thresholds are: V1 = 110 kt and VR = 135 kt. In our setting, we consider six clocks that describe the behavior of events:

| | |
|---|---|
| `time-SI` | physical time in seconds (s) |
| `speed-SI` | aircraft speed in $m.s^{-1}$ |
| `speed-KT` | aircraft speed in knots (kt) |
| `V1-reach` | V1 speed threshold event |
| `VR-reach` | VR speed threshold event |
| `RTO` | rejected takeoff procedure event |
| `liftoff` | aircraft liftoff event |

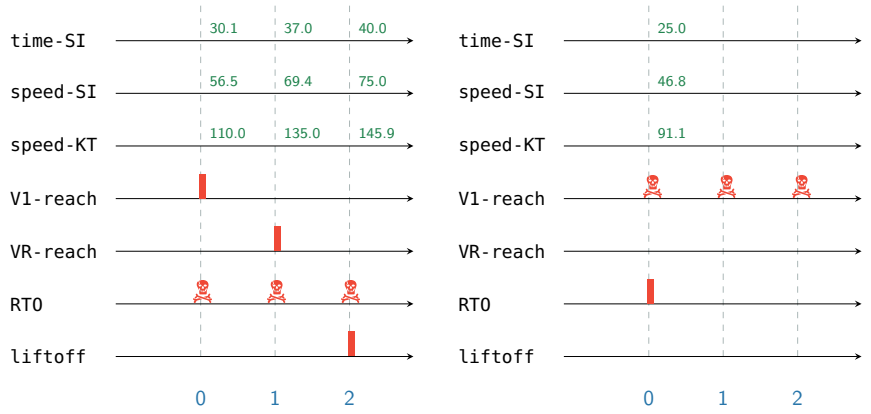
Here is a TESL* specification expressing the case in Listing 5:

Listing 5: Specification of an airplane takeoff in TESL*

```
1  rational-clock time-SI   // in [s]
2  rational-clock speed-SI  // in [m.s^-1]
3  rational-clock speed-KT  // in [kt]
4
5  // Uniform acceleration
6  tag relation speed-SI = 1.875 * time-SI
7
8  // Unit conversion between [m.s^-1] and [kt]
9  tag relation speed-KT = 1.944 * speed-SI
10
```

```
11 // Speed thresholds
12 V1-reach strictly precedes VR-reach
13 V1-reach sporadic 110.0 on speed-KT
14 VR-reach sporadic 135.0 on speed-KT
15
16 // Takeoff rejection is forbidden after reaching V1
17 V1-reach kills RTO
18 // Rejecting takeoff prevents from reaching takeoff speeds
19 RTO kills V1-reach
20 // Liftoff occurs 3s after reaching VR
21 VR-reach time delayed by 3.0 on time-SI implies liftoff
```

Lines 1 to 3 specify the tag domain for clocks `time-SI`, `speed-SI` and `speed-KT` to be rationals. Then, the next two constraints deal with how time flows: Line 6 describes the uniform acceleration profile of the aircraft with a linear constraint, while Line 9 describes unit conversion for speeds between knots and $m.s^{-1}$ (as in SI units). Line 12 specifies that reaching VR is possible whenever V1 has been reached in the past, that is `VR-reach` ticks if `V1-reach` has ticked before. Lines 13 and 14 specify that clocks `V1-reach` and `VR-reach` shall tick whenever their associated threshold values are measured on the timeframe of clock `speed-KT`. Lines 17 and 19 describe some kind of *race condition* between clocks `V1-reach` and `RTO`. Whichever event occurs first, permanently prevents the other. It is not possible to reject takeoff if V1 has been reached. Besides, it is not possible to reach V1 if takeoff rejection procedure is engaged. Finally, Line 21 specifies that whenever VR is reached, time is measured on the timeframe of `time-SI` and delayed with a duration of 3s to trigger aircraft lift off.




(a) Takeoff in normal conditions (b) Rejected takeoff at 25 sec

Figure 31: Two partially satisfying behaviors for the takeoff procedure

Two behaviors are given in Figure 31. Additionally, a skull stands for permanent absence of event occurrence (clock death). Due to space restrictions, tags

may be truncated up to four digits. The first behavior depicted by Figure 31a, corresponds to a "minimal" interpretation of the model, where the simulation starts when speed reaches 110.0 on clock `speed-KT`, and a tick appears on clock `V1-reach` to notify reaching decision speed. As a matter of fact, clock `RTO` immediately dies, as it is impossible to engage any emergency stop procedure from then on. Then, whenever speed is 135.0 on `speed-KT`, clock `VR-reach` ticks. After 3 s, the aircraft finally lifts off the ground and clock `liftoff` is ticking. On the other side, Figure 31b shows a behavior with a additional tick on clock `RTO` when the time on clock `time-SI` is 25, which corresponds to engaging stop procedure. As it occurred when speed was less than V1, clock `V1-reach` is now prevented from ticking and cannot cause lift off.

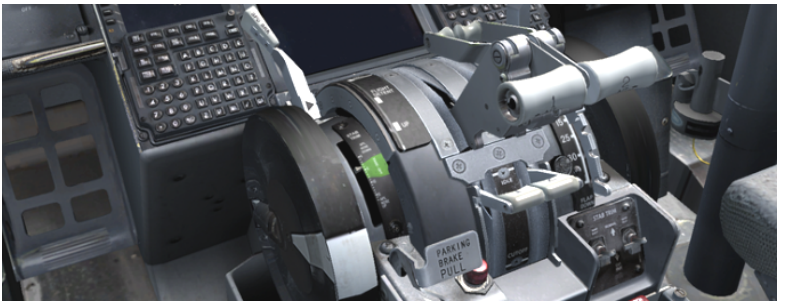



Figure 32: Throttle console of a Boeing 737 used in takeoff rejection procedure

## 4.2 FORMAL SEMANTICS

### 4.2.1 *Denotational Semantics*

Similarly to Definition 25 for TESLε we wish to provide a mathematical understanding of what we expect from these operators, as long as *compositionality* remains the leading property. To carry out this idea, we extend the interpretation function $[\![\_]\!]_{\mathsf{TESL}}$ in Definition 40. A run $\rho$ satisfies:

- an `implies not` atom, when for every index $n$, if clock $K_1$ is ticking then $K_2$ must not tick at the same instant;
- a `weakly precedes` atom, when from every index $n$, the number of times clock $K_1$ has ticked is greater or equal than the number of times clock $K_2$ has ticked;
- a `strictly precedes` atom, when for every index $n$, the number of times clock $K_1$ has ticked from the previous instant is greater or equal than the number of times clock $K_2$ has ticked from the current instant;
- a `kills` atom, whenever clock $K_1$ is ticking, clock $K_2$ will be prevented from ticking forever from that instant.

**Definition 40 (*Interpretation of TESL⋆ formulae*)**

The denotational semantics of TESL⋆ formulae is given inductively as sets of runs in Definition 25 and extended with the following.

$$[\![K_1 \texttt{ implies not } K_2]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \in \mathbb{N} \quad \text{ticks}(\rho_n(K_1)) \quad \text{implies} \quad \neg\, \text{ticks}(\rho_n(K_2))\}$$

$$[\![K_1 \texttt{ weakly precedes } K_2]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \left\{\rho \in \Sigma^\infty \;\middle|\; \forall n \in \mathbb{N} \quad \text{card}\left\{j \leqslant n \;\middle|\; \text{ticks}(\rho_j(K_1))\right\} \geqslant \text{card}\left\{j \leqslant n \;\middle|\; \text{ticks}(\rho_j(K_2))\right\}\right\}$$

$$[\![K_1 \texttt{ strictly precedes } K_2]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \left\{\rho \in \Sigma^\infty \;\middle|\; \forall n \in \mathbb{N} \quad \text{card}\left\{j < n \;\middle|\; \text{ticks}(\rho_j(K_1))\right\} \geqslant \text{card}\left\{j \leqslant n \;\middle|\; \text{ticks}(\rho_j(K_2))\right\}\right\}$$

$$[\![K_1 \texttt{ kills } K_2]\!]_{\text{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \in \mathbb{N} \quad \text{ticks}(\rho_n(K_1)) \quad \text{implies} \quad \forall n' \geqslant n \quad \neg\, \text{ticks}(\rho_{n'}(K_2))\}$$

### 4.2.2 *Towards Stuttering Invariance*

An important property that is derived from the denotational semantics is *invariance by stuttering* and ensures compositionality of models. Similarly to the composition of automata, the addition of stutter or silent instants, allows the accommodation for their differences and hence interleave each of their steps. For instance, the seminal specification language LTL is known to have a stuttering-invariant fragment [Lam83, KS05]. Our intuition that any TESL specification is stuttering-invariant is proved by Boulanger[1]. The proof is based on dilating functions that are applied on satisfying run sets that are shown to preserve property satisfaction. This idea can be illustrated by the previous example of the airplane takeoff. The satisfying run shown in Figure 31a is a minimal observation of the execution of the system considering the only necessary event occurrences. The idea is that the semantic model is not too restrictive and allows to compose with other models for other systems.

In practice, simulation can also be run with an additional chronometric model that ticks at fixed time value. This observation is exhibited by composing the specification Listing 5 with a chronometer that would be ticking every 5 sec.

```
22 rational-clock chronometer sporadic 0.0
23 tag relation chronometer = time-SI
24 chronometer time delayed by 5.0 on chronometer implies chronometer
```

The process of dilating time is hence illustrated by Figure 33. Every event occurring in the "minimal" run in Figure 31a is finally retrieved in the new run from reaching V1 to airplane liftoff.

1 See https://heron-solver.github.io/hygge/Stuttering.html

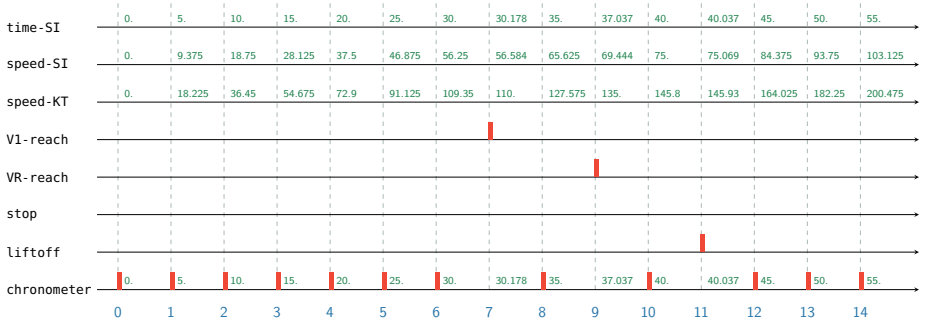



Figure 33: Takeoff in normal conditions with a chronometer in parallel

### 4.2.3 *Operational Semantics*

As seen in the previous paragraphs, the operator `precedes` is an asynchronous operator that is oriented towards the past. It enforces a property that needs to count ticks that already exist in the past of the run being built. We observe that the primitives given in Definition 29 are not sufficient to encompass this need. Indeed, they only deal with fixed instants, and the operational semantics of TESL$\varepsilon$ in Definition 33 only progresses by unfolding formulae towards the future. To overcome this limitation, we first need to extend the run primitives. In particular, we add a tick counting arithmetic relation, in the style of relation between tag variables.

**Definition 41 (*Extended Run Primitives for TESL**)**

A run primitive $\gamma \in \Gamma$ is a constraint as in Definition 29 or a symbol of the following kind:

- $\lceil c_1, c_2 \rceil \in R$ enforces the arithmetic relation $R$ between tick counters $c_1$ and $c_2$;
- $K \not\upharpoonright_{\geqslant n}$ forces clock $K$ to not tick (remain idle) from instant index $n$ on,

which are interpreted by $[\![\_]\!]_{\text{prim}}$ as:

$$[\![K \not\upharpoonright_{\geqslant n}]\!]_{\text{prim}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^{\infty} \mid \forall n' \geqslant n \quad \text{ticks}(\rho_n(K)) \text{ is false}\}$$

$$[\![\lceil c_1, c_2 \rceil \in R]\!]_{\text{prim}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^{\infty} \mid [\![c_1]\!]^{\rho}_{\text{cnt}} \text{ and } [\![c_1]\!]^{\rho}_{\text{cnt}} \text{ are related by } R\},$$

and where $c_1$ and $c_2$ are tick counter symbols of the following kind

- $\#^{<n}K$ is the number of ticking instants on clock $K$ in run $\rho$ from instant $0$ until $n$ excluded;
- $\#^{\leqslant n}K$ is the number of ticking instants on clock $K$ in run $\rho$ from instant $0$ until $n$ included;

which are evaluated by $[\![\_]\!]^{-}_{\mathsf{cnt}}$ as:

$$[\![\#^{<n}K]\!]^{\rho}_{\mathsf{cnt}} \stackrel{\text{def}}{=} \mathrm{card}\left\{j < n \mid \mathsf{ticks}(\rho_j(K)) \text{ is true}\right\}$$
$$[\![\#^{\leqslant n}K]\!]^{\rho}_{\mathsf{cnt}} \stackrel{\text{def}}{=} \mathrm{card}\left\{j \leqslant n \mid \mathsf{ticks}(\rho_j(K)) \text{ is true}\right\}$$

*Remark* 42. The structure of primitives now contains *tick counters* that are expressions meant to yield the number of times an event occurs until a given run and a given instant index. On top of that, the addition of these primitives exhibits a duality between ticks and tags.

| TICKS | TAGS |
|---|---|
| $\_ \Uparrow \_$ | $\_ \Downarrow \_$ |
| $\lceil \_, \_ \rceil \in \_$ | $\lfloor \_, \_ \rfloor \in \_$ |

Table 1: Duality in run primitives between ticks and tags

From these extensions on run primitives, we can express our operators in the new setting as the addition of new elimination rules:

**Definition 43 (*Extended Elimination Rules* $\rightarrow_e$ *for* TESL$^\star$)**

The relation $\rightarrow_e$ is the smallest relation satisfying the rules given in Definition 33 and as follows

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ implies not } K_2) \triangleright \Phi \qquad (\text{implies} - \text{not}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{K_1 \not\Uparrow_n\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ implies not } K_2)$$

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ implies not } K_2) \triangleright \Phi \qquad (\text{implies} - \text{not}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{\begin{array}{l} K_1 \Uparrow_n \\ K_2 \not\Uparrow_n \end{array}\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ implies not } K_2)$$

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ weakly precedes } K_2) \triangleright \Phi \qquad (\text{weakly} - \text{precedes}_e)$$
$$\rightarrow_e \Gamma \cup \left\{\lfloor \#^{\leqslant K_1} n, \#^{\leqslant K_2} n \rfloor \in \geqslant\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ weakly precedes } K_2)$$

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ strictly precedes } K_2) \triangleright \Phi \qquad (\text{strictly} - \text{precedes}_e)$$
$$\rightarrow_e \Gamma \cup \left\{\lceil \#^{< K_1} n, \#^{\leqslant K_2} n \rceil \in \geqslant\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ strictly precedes } K_2)$$

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ kills } K_2) \triangleright \Phi \qquad (\text{kills}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{K_1 \not\Uparrow_n\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ kills } K_2)$$

$$\Gamma \models_n \Psi \wedge (K_1 \texttt{ kills } K_2) \triangleright \Phi \qquad (\text{kills}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{\begin{array}{l} K_1 \Uparrow_n \\ K_2 \not\Uparrow_{\geqslant n} \end{array}\right\} \models_n \Psi \triangleright \Phi \wedge (K_1 \texttt{ kills } K_2)$$

The same way as in TESL$\varepsilon$ these rules are used to completely consume the present to produce the past. The elimination steps are repeated until irreducibility. The extension of elimination rules proceeds this way

- $K_1$ `implies not` $K_2$: either clock $K_1$ is not ticking (Rule $\mathsf{implies-not}_{e1}$), or clock $K_1$ ticks and prevents $K_2$ to instantaneously tick (Rule $\mathsf{implies-not}_{e2}$). In both cases, the formula is copied into the future as it has to be satisfied at every instant;
- $K_1$ `weakly precedes` $K_2$: at *this* instant, the number of times clock $K_1$ has ticked is greater or equal to the number of times clock $K_2$ has ticked (Rule $\mathsf{weakly-precedes}_e$). Then, the constraint is repeated in the future;
- $K_1$ `strictly precedes` $K_2$: at the *previous* instant, the number of times clock $K_1$ has ticked is greater or equal to the number of times clock $K_2$ has from this instant (Rule $\mathsf{strictly-precedes}_e$). Again, the constraint is repeated in the future;
- $K_1$ `kills` $K_2$: either clock $K_1$ is not ticking (Rule $\mathsf{kills}_{e1}$), or clock $K_1$ ticks and prevents $K_2$ to tick from this instant and forever on (Rule $\mathsf{kills}_{e2}$). In both cases, the formula is copied into the future as it has to be satisfied at every instant.

The property of local termination remains preserved as elimination rules presented above ensures the strict decrement of the number of $\Psi$-formulae.

*Remark* 44. The usage of the primitive $\_ \not\gg \_$ has been illustrated in Figure 31 with the superficial symbol ☠ (skull) to emphasize on the clock death. No event can ever occur on the died clock from the very moment that the killing clock has triggered. Moreover, the extended primitives preserve decidability with the same assumptions of Lemma 30.

# 5

# LANGUAGE WITH SEQUENTIAL OPERATORS: TESL

The operators we have defined in TESL$\varepsilon$ and TESL$^\star$ will be proved to exhibit guarantees in Chapter 6. In the next paragraphs, we explore another class of operators for which we give an operational semantics. These operators carry a state, e.g., a counter or a boolean, that may change during the execution of the operational semantics, similarly to sequential circuits containing registers. Except for one operator which is meant to complete and formalize the original TESL language.

We give hereafter five operators of the complete language for which we provide operational rules that are used for deriving runs, but are not (yet) proven to be safe. This is meant to be done in future work. They consist of five operators as found in the original TESL language: `sustained implies`, `await implies`, `delayed implies`, `filtered implies` and `when implies`. Note that the last operator is not sequential but meant to complete and retrieve the original TESL language.

## 5.1 SYNTAX

From the grammar of TESL$\varepsilon$ in Chapter 3, we add these symbols to the already existing atomic formulae. They correspond to those found in the original TESL language.

**Definition 45 (*Grammar of TESL*)**

A TESL formula $\Psi$ is given by the addition to TESL$\varepsilon$ of the following

$$
\begin{aligned}
\langle atom \rangle \quad ::= \quad & \ldots \\
\mid \quad & \langle clock \rangle \texttt{ sustained from } \langle clock \rangle \texttt{ to } \langle clock \rangle \texttt{ implies } \langle clock \rangle \\
\mid \quad & \texttt{await } \langle clock \rangle \ldots \langle clock \rangle \texttt{ implies } \langle clock \rangle \\
\mid \quad & \langle clock \rangle \texttt{ delayed by } \langle nat \rangle \texttt{ on } \langle clock \rangle \texttt{ implies } \langle clock \rangle \\
\mid \quad & \langle clock \rangle \texttt{ filtered by } \langle nat \rangle\texttt{, } \langle nat \rangle \texttt{ (} \langle nat \rangle\texttt{, } \langle nat \rangle\texttt{)}^* \texttt{ implies } \langle clock \rangle \\
\mid \quad & \langle clock \rangle \texttt{ when } \langle clock \rangle \texttt{ implies } \langle clock \rangle
\end{aligned}
$$

where $\langle nat \rangle \in \mathbb{N}$.

SUSTAINED IMPLICATION.

$$K_{master} \texttt{ sustained from } K_{begin} \texttt{ to } K_{end} \texttt{ implies } K_{slave}$$

is a formula that enforces the implication between $K_{master}$ and $K_{slave}$ when enabled by a tick on $K_{begin}$ and disabled by a tick on $K_{end}$. In Figure 34, the range of instants where sustained implication holds is $(1;4]$ as it starts when $K_{begin}$ ticks at instant 1 and stops when $K_{end}$ ticks at instant 4. Notice that any additional tick on clock $K_{begin}$ during that instant range will not affect the state of the formula as long as it is not deactivated by $K_{end}$.

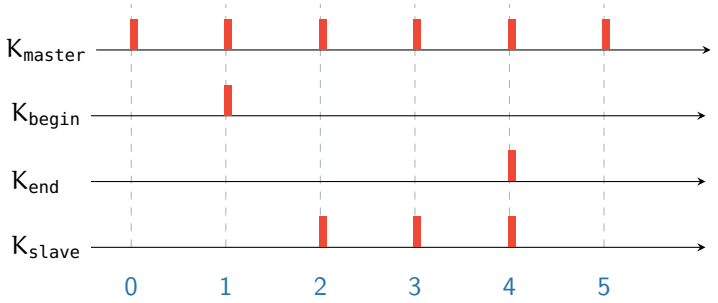



Figure 34: A satisfying run for the `sustained implies` formula

AWAIT IMPLICATION.

$$\texttt{await } K_{m1}\ K_{m2}\ K_{m3} \ldots \texttt{ implies } K_{slave}$$

is a formula which specifies that as soon as all clocks $K_{m1}$, $K_{m2}$, $K_{m3}$ ...have ticked, $K_{slave}$ shall tick instantaneously. In Figure 35, we illustrate with only two master clocks $K_{m1}$ and $K_{m2}$. They ticked in the instant range $[1;2]$ which has led to the instant tick on $K_{slave}$ at instant 2. Again at instant 3, both $K_{m1}$ and $K_{m2}$ ticked, hence will $K_{slave}$ accordingly.

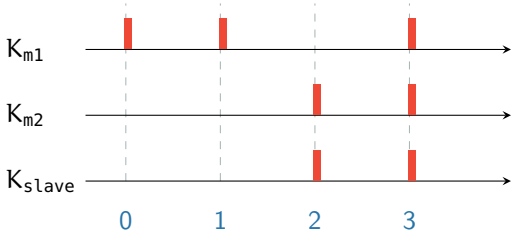



Figure 35: A satisfying run for the `await implies` formula

DELAYED IMPLICATION.

$$K_{master} \texttt{ delayed by } n \texttt{ on } K_{count} \texttt{ implies } K_{slave}$$

is a formula which specifies a delayed implication. Whenever clock $K_{master}$ ticks at some instant, a number of ticks $n$ shall be counted on clock $K_{count}$, to trigger a tick on clock $K_{slave}$. In Figure 36, the presented run partially satisfies the specification where $n = 3$. A tick on clock $K_{master}$ at instant 0, will create a tick depending on counting tick occurrences on clock $K_{count}$. These happen at instants 1, 2, 4 and instantaneously trigger clock $K_{slave}$. Compared to the `time delayed by` atom, the measurement is no longer made on tags, but on *counting ticks*. Similarly, whenever $K_{master}$ ticks, an intermediate operator `reaches implies` is produced. It does not focus on tags, but stores a decreasing counter; whenever the counter reaches 1, the slave clock is triggered.

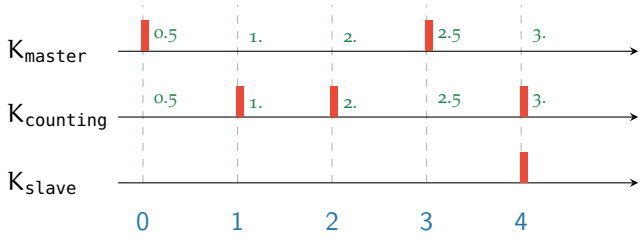



Figure 36: A satisfying run for the `delayed implies` formula where $n = 3$

FILTERED IMPLICATION.

$$K_{master} \texttt{ filtered by } s, k\ (rs, rk)^* \texttt{ implies } K_{slave}$$

is a formula which specifies implication between clocks $K_{master}$ and $K_{slave}$ under a skip-keep pattern condition: $s$ ticks on $K_{master}$ are skipped and $k$ ticks are kept, then we repeat the process by skipping $rs$ ticks and keeping $rk$ ticks. In the example of satisfying run in Figure 37, we chose these values to illustrate: $s = 1$, $k = 2$, $rs = 1$ and $rk = 3$.

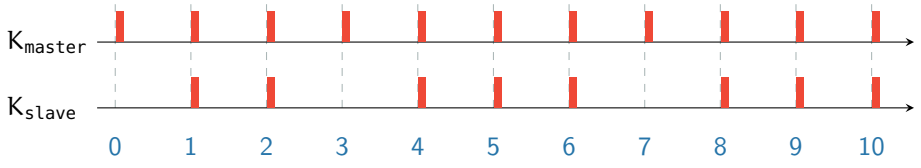



Figure 37: A satisfying run for the `filtered implies` formula

WHEN IMPLICATION.

$$K_{master} \texttt{ when } K_{sampling} \texttt{ implies } K_{slave}$$

is a formula which specifies that whenever $K_{master}$ and $K_{sampling}$ ticks at the same instant, a tick shall appear on $K_{slave}$. Stream-wise, it is equivalent to con-

junction in propositional logic in premise of the implication. The run presented in Figure 38 shows that whenever $K_{master}$ and $K_{sampling}$ simultaneously tick, then $K_{slave}$ ticks to enforce implication.

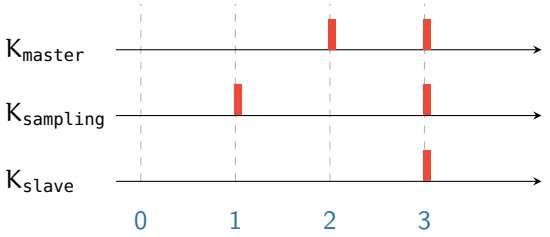



Figure 38: A satisfying run for the `when implies` formula

### 5.1.1 *The Concurrent Computations Example*

Compared to the previous language variants, the formulae given in this chapter contain a "state". They are useful whenever some information needs to be stored in order to rule how some event will be triggered. To illustrate this process, we consider the example from [BJHP14] of two CPUs. Each of them lives in an independent timeframe, and respectively computes a value A and a value B. Whenever both results are available, the first CPU computes $A + B$.

Listing 6: Specification of concurrent computations for two CPUs in TESL

```
1  rational-clock CPU1_time                // time scale on CPU 1
2  rational-clock compute_A sporadic 1.0   // start computing A at 1.0
3  tag relation compute_A = CPU1_time
4  unit-clock A_available
5  compute_A time delayed by 0.5 on CPU1_time implies A_available
6
7  rational-clock CPU2_time                // time scale on CPU 2
8  rational-clock compute_B sporadic 2.0   // start computing B at 2.0
9  tag relation compute_B = CPU2_time
10 unit-clock B_available
11 compute_B time delayed by 1.5 on CPU2_time implies B_available
12
13 // start computing A+B when both A and B are available
14 unit-clock compute_A_plus_B
15 await A_available B_available implies compute_A_plus_B
16
17 unit-clock A_plus_B_available
18 compute_A_plus_B time delayed by 1.0 on CPU1_time implies
       A_plus_B_available
```

From Line 1 to 5 on Listing 6, the first CPU is in charge of computing the value of A on a timeframe given by clock `CPU1_time`. The computation starts at

1.0 and the value will be available after a time delay of 0.5 on that timeframe. On the other side, Lines 7 to 11 specify the computation of the value of B by the second CPU. Likewise, the computation however starts at 2.0 and finishes after a time delay of 1.5 based on the timeframe of `CPU2_time`. Lines 15 and 17 gives the constraint on the way to trigger clock `compute_A_plus_B`: as soon as clocks `A_available` and `B_available` are triggered, clock `compute_A_plus_B` will be triggered too. Finally, Line 18 states that the value of $A + B$ is available after a duration of 1.0 on the timeframe of the first CPU.

*Remark* 46. The reader will carefully observe that both timeframes are unrelated, which allows any interleaving of events and any timescale relation between them. This lack of specification is necessary in this case to describe physical systems evolving independently and related in unknown ways, e.g., as would be found in distributed computing. Timed automata are known to hardly address these issues [ABG+08].


Figure 39: A satisfying run of concurrent computations example

A satisfying run of the previous specification is given in Figure 39. Both clocks `compute_A` and `compute_B` start on their own timeframes, respectively on timestamps 1.0 and 2.0. On the next instant, results for A and B are available and shown by the ticking clocks `A_available` and `B_available` while time is 1.5 on `CPU1_time` and 3.5 on `CPU2_time`. They instantaneously trigger the computation of the sum of A and B by triggering clock `compute_A_plus_B`. Finally, this result is available on the following instant 2.0 exhibited by the ticking clock `A_plus_B_available` when the time is 2.5 on `CPU1_time`.

## 5.2 OPERATIONAL SEMANTICS

In this section, we give the operational semantics for the full TESL language. Compared to TESL*, we do not need to extend primitives, and keep relying on those defined in Definition 29.

**Definition 47 (*Extended Elimination Rules* $\rightarrow_e$ *for* ***TESL*****)**

The relation $\rightarrow_e$ is the smallest relation satisfying the rules given in Definition 33 and in the following subsections.

### 5.2.1 *Sustained Implication*

In the following operational rules, we have defined an intermediate formula named `sustained until`. The difference between formulae `sustained implies` and `sustained until` lies under the idea of whether the operator is "activated" or not. It remains deactivated if $K_{begin}$ does not tick (Rule sustained − from$_{e1}$) or gets activated otherwise (Rule sustained − from$_{e2}$).

$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{sustained from}\ K_{begin}\ \texttt{to}\ K_{end}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad \text{(sustained} - \text{from}_{e1}\text{)}$$
$$\rightarrow_e \Gamma \cup \left\{ K_{begin} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{sustained from}\ K_{begin}\ \texttt{to}\ K_{end}\ \texttt{implies}\ K_{slave})$$
$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{sustained from}\ K_{begin}\ \texttt{to}\ K_{end}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad \text{(sustained} - \text{from}_{e2}\text{)}$$
$$\rightarrow_e \Gamma \cup \left\{ K_{begin} \Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{sustained until}\ K_{end}\ \texttt{restarts}\ K_{begin}\ \texttt{implies}\ K_{slave})$$

In case the operator is already enabled and that clock $K_{end}$ does not tick, synchronous implication between clocks $K_{master}$ and $K_{slave}$ is checked and the operator remains activated in the future of the process (Rules sustained − until$_{e1}$ and sustained − until$_{e1}$).

$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{sustained until}\ K_{end}\ \texttt{restarts}\ K_{begin}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad \text{(sustained} - \text{until}_{e1}\text{)}$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{l} K_{end} \not\Uparrow_n \\ K_{master} \not\Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{sustained until}\ K_{end}\ \texttt{restarts}\ K_{begin}\ \texttt{implies}\ K_{slave})$$
$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{sustained until}\ K_{end}\ \texttt{restarts}\ K_{begin}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad \text{(sustained} - \text{until}_{e2}\text{)}$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{l} K_{end} \not\Uparrow_n \\ K_{master} \Uparrow_n \\ K_{slave} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{sustained until}\ K_{end}\ \texttt{restarts}\ K_{begin}\ \texttt{implies}\ K_{slave})$$

If $K_{end}$ ticks, then causality is checked for the last time, then operator gets disabled from the next instant (Rules sustained − until$_{e3}$ and sustained − until$_{e4}$).

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\mathsf{master}}\ \texttt{sustained until}\ \mathsf{K}_{\mathsf{end}}\ \texttt{restarts}\ \mathsf{K}_{\mathsf{begin}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{sustained} - \text{until}_{e3})$$
$$\to_e \Gamma \cup \left\{ \begin{array}{l} \mathsf{K}_{\mathsf{end}} \Uparrow_n \\ \mathsf{K}_{\mathsf{master}} \not\Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\mathsf{master}}\ \texttt{sustained from}\ \mathsf{K}_{\mathsf{begin}}\ \texttt{to}\ \mathsf{K}_{\mathsf{end}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}})$$
$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\mathsf{master}}\ \texttt{sustained until}\ \mathsf{K}_{\mathsf{end}}\ \texttt{restarts}\ \mathsf{K}_{\mathsf{begin}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{sustained} - \text{until}_{e4})$$
$$\to_e \Gamma \cup \left\{ \begin{array}{l} \mathsf{K}_{\mathsf{end}} \Uparrow_n \\ \mathsf{K}_{\mathsf{master}} \Uparrow_n \\ \mathsf{K}_{\mathsf{slave}} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\mathsf{master}}\ \texttt{sustained from}\ \mathsf{K}_{\mathsf{begin}}\ \texttt{to}\ \mathsf{K}_{\mathsf{end}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}})$$

### 5.2.2 *Await Implication*

To handle the asynchronous operator `await implies`, we provide four rules. Compared to the previous rules, they deal with collections of clocks and are applied under syntactic conditions (empty or non-empty collections). Moreover, they can reduce into the present or into the future; compared to the previous rules, which directly reduce into the future of the process.

We need a state $\mathcal{K}_{\mathsf{await}}$ containing all clocks that need to be listened. Another state $\mathcal{K}_{\mathsf{insts}}$ that contains clocks that need to be listened in this instant and $\mathsf{K}_{\mathsf{listen}}$ which is the current clock to deal with for the elimination rule. If $\mathsf{K}_{\mathsf{listen}}$ ticks, then we remove it from $\mathcal{K}_{\mathsf{remn}}$ and we continue with the instantaneous clocks to listen $\mathcal{K}_{\mathsf{insts}}$ in the next reduction steps still in the present of the process (Rule $\text{await}_{e1}$). Otherwise, $\mathcal{K}_{\mathsf{remn}}$ remains unchanged and we continue with the next listening clocks $\mathcal{K}_{\mathsf{insts}}$ (Rule $\text{await}_{e2}$).

$$\Gamma \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}}\ \texttt{instantly}\ \{\mathsf{K}_{\mathsf{listen}}\} \cup \mathcal{K}_{\mathsf{insts}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{await}_{e1})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\mathsf{listen}} \Uparrow_n \right\} \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}} \setminus \{\mathsf{K}_{\mathsf{listen}}\}\ \texttt{instantly}\ \mathcal{K}_{\mathsf{insts}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi$$
$$\Gamma \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}}\ \texttt{instantly}\ \{\mathsf{K}_{\mathsf{listen}}\} \cup \mathcal{K}_{\mathsf{insts}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{await}_{e2})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\mathsf{listen}} \not\Uparrow_n \right\} \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}}\ \texttt{instantly}\ \mathcal{K}_{\mathsf{insts}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi$$

When we exhausted $\mathcal{K}_{\mathsf{insts}}$, the operator has listened to all instantaneous listening clocks, and will eventually trigger if the remaining clocks can be heard in the future; thus the formula jumps into the future (Rule $\text{await}_{e3}$). Finally, if there is no remaining clock to listen either instantaneously, either asynchronously, then we must trigger $\mathsf{K}_{\mathsf{slave}}$ (Rule $\text{await}_{e4}$).

$$\Gamma \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}} \neq \varnothing\ \texttt{instantly}\ \varnothing\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{await}_{e3})$$
$$\to_e \Gamma \models_n \Psi \triangleright \Phi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{remn}}\ \texttt{instantly}\ \mathcal{K}_{\mathsf{remn}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}})$$
$$\Gamma \models_n \Psi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \varnothing\ \texttt{instantly}\ \varnothing\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{await}_{e4})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\mathsf{slave}} \Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\texttt{await}\ \mathcal{K}_{\mathsf{await}}\ \texttt{remains}\ \mathcal{K}_{\mathsf{await}}\ \texttt{instantly}\ \mathcal{K}_{\mathsf{await}}\ \texttt{implies}\ \mathsf{K}_{\mathsf{slave}})$$

### 5.2.3 *Delayed Implication*

Here two groups of reduction rules appear. To eliminate a `delayed implies` formula, we focus on clock $\mathsf{K}_{\texttt{master}}$. If $\mathsf{K}_{\texttt{master}}$ is absent, then the operator is simply

discarded and the formula switches to the future (Rule $\text{delayed}_{e1}$). Otherwise, in the case $\mathsf{K}_{\texttt{master}}$ is ticking, a counter implication is created and stores in the future that, as soon as $m$ ticks have been counted on clock $\mathsf{K}_{\texttt{counting}}$, a tick will be fired on $\mathsf{K}_{\texttt{slave}}$ (Rule $\text{delayed}_{e2}$).

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{master}} \texttt{ delayed by } m \texttt{ on } \mathsf{K}_{\texttt{counting}} \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{delayed}_{e1})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\texttt{master}} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\texttt{master}} \texttt{ delayed by } m \texttt{ on } \mathsf{K}_{\texttt{counting}} \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$
$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{master}} \texttt{ delayed by } m \texttt{ on } \mathsf{K}_{\texttt{counting}} \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{delayed}_{e2})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\texttt{master}} \Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } m \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$
$$\wedge (\mathsf{K}_{\texttt{master}} \texttt{ delayed by } m \texttt{ on } \mathsf{K}_{\texttt{counting}} \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$

The second group of reduction rules deals with the elimination of the `reaches implies` operators. On the counting clock $\mathsf{K}_{\texttt{counting}}$, if it is not ticking, then it gets discarded and the counter remains the same $m$ (Rule $\text{reaches} - \text{implies}_{e1}$). Otherwise, the counter is decremented (Rule $\text{reaches} - \text{implies}_{e2}$). As soon as the counter value is 1, $\mathsf{K}_{\texttt{slave}}$ is triggered and the operators disappears (Rule $\text{reaches} - \text{implies}_{e3}$).

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } m \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{reaches} - \text{implies}_{e1})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\texttt{counting}} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } m \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$
$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } m+1 \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{reaches} - \text{implies}_{e2})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\texttt{counting}} \Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } m \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$
$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{counting}} \texttt{ reaches } 1 \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{reaches} - \text{implies}_{e3})$$
$$\to_e \Gamma \cup \left\{ \begin{array}{c} \mathsf{K}_{\texttt{counting}} \Uparrow_n \\ \mathsf{K}_{\texttt{slave}} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi$$

### 5.2.4 *Filtered Implication*

In this case, the operator carries itself the state of whether it is skipping or keeping synchronous causality between master and slave clocks. Four cases of reduction rules correspond to the four integer counters of the filtered implication. If clock $\mathsf{K}_{\texttt{master}}$ does not tick, we simply discard the operator switching it to the future and counters remain the same values (Rule $\text{filtered}_{e1}$).

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\texttt{master}} \texttt{ filtered by } s, k\ (rs, rk)^* \texttt{ implies } \mathsf{K}_{\texttt{slave}}) \triangleright \Phi \qquad (\text{filtered}_{e1})$$
$$\to_e \Gamma \cup \left\{ \mathsf{K}_{\texttt{master}} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\texttt{master}} \texttt{ filtered by } s, k\ (rs, rk)^* \texttt{ implies } \mathsf{K}_{\texttt{slave}})$$

Otherwise, $\mathsf{K}_{\texttt{master}}$ ticks and three cases appear with respect to the values of the counter of ticks to skip $s$, and the counter of ticks to keep $k$. If the number of ticks we skip $s$ is non-zero, it means the process is in a state of skipping, so the current tick is skipped and the counter decremented (Rule $\text{filtered}_{e2}$).

$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{filtered by}\ s+1, k+1\ (rs, rk)^*\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{filtered}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ K_{master} \Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{filtered by}\ s, k+1\ (rs, rk)^*\ \texttt{implies}\ K_{slave})$$

Otherwise it is zero, which means it has entered a state of keeping ticks on the master clock. If the "keeping" counter is greater or equals to 2, a tick is triggered on $K_{slave}$ and the $k$ counter is decremented (Rule $\text{filtered}_{e3}$). Eventually $k = 1$ and we keep for the last time, then we reset counter values with the repeating counters $rs$ and $rk$ (Rule $\text{filtered}_{e4}$).

$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{filtered by}\ 0, k+2\ (rs, rk)^*\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{filtered}_{e3})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{c} K_{master} \Uparrow_n \\ K_{slave} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{filtered by}\ 0, k+1\ (rs, rk)^*\ \texttt{implies}\ K_{slave})$$
$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{filtered by}\ 0, 1\ (rs, rk)^*\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{filtered}_{e4})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{c} K_{master} \Uparrow_n \\ K_{slave} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{filtered by}\ rs, rk\ (rs, rk)^*\ \texttt{implies}\ K_{slave})$$

### 5.2.5 *When Implication*

Similarly to propositional logic, we provide three rules to reduce a `when implies`. Note that this operator does not contain a state but is meant to complete the formalization and retrieve the original TESL language. If the master clocks is not ticking, then the operator is discarded (Rules $\text{when} - \text{implies}_{e1}$ and $\text{when} - \text{implies}_{e2}$). Otherwise, both clocks are ticking and so must $K_{slave}$ (Rule $\text{when} - \text{implies}_{e3}$).

$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{when} - \text{implies}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{ K_{master} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave})$$
$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{when} - \text{implies}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ K_{sampling} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave})$$
$$\Gamma \models_n \Psi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave}) \triangleright \Phi \qquad (\text{when} - \text{implies}_{e3})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{c} K_{master} \Uparrow_n \\ K_{sampling} \Uparrow_n \\ K_{slave} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (K_{master}\ \texttt{when}\ K_{sampling}\ \texttt{implies}\ K_{slave})$$

Additionally, we also provide rules to reduce a `when not implies` which are analogous to the previous formula.

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{when} - \text{not} - \text{implies}_{e1})$$
$$\rightarrow_e \Gamma \cup \left\{ \mathsf{K}_{\mathsf{master}} \not\Uparrow_n \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}})$$

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{when} - \text{not} - \text{implies}_{e2})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{c} \mathsf{K}_{\mathsf{master}} \Uparrow_n \\ \mathsf{K}_{\mathsf{sampling}} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}})$$

$$\Gamma \models_n \Psi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}}) \triangleright \Phi \qquad (\text{when} - \text{not} - \text{implies}_{e3})$$
$$\rightarrow_e \Gamma \cup \left\{ \begin{array}{c} \mathsf{K}_{\mathsf{master}} \Uparrow_n \\ \mathsf{K}_{\mathsf{sampling}} \not\Uparrow_n \\ \mathsf{K}_{\mathsf{slave}} \Uparrow_n \end{array} \right\} \models_n \Psi \triangleright \Phi \wedge (\mathsf{K}_{\mathsf{master}} \texttt{ when not } \mathsf{K}_{\mathsf{sampling}} \texttt{ implies } \mathsf{K}_{\mathsf{slave}})$$

# 6

# FORMAL AND MECHANIZED CERTIFICATION

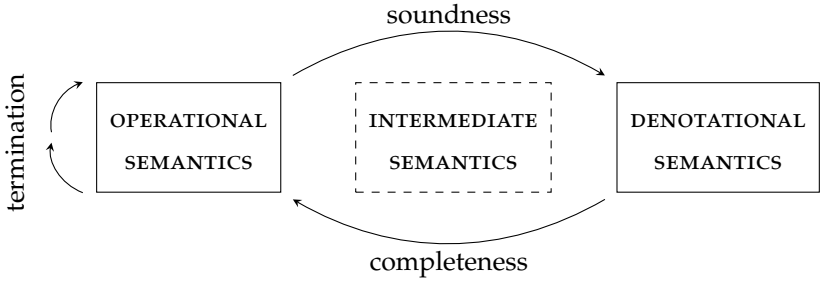



Figure 40: Map of relations between semantics

In this chapter, we give key properties to ensure the consistency of the operational semantics of TESL*. We are particularly interested in establishing essential properties of soundness (Theorem 57), completeness (Theorem 60), and progress (Theorem 62). Such properties precisely rely on the denotational semantics first defined. To achieve this goal, the key idea is to exhibit how one semantics is reflected in the other. We thus decompose the denotational semantics into a *stepwise denotational semantics* that serves as an intermediate semantics between both operational and denotational semantics. We observe that

1. it is trivially equivalent to the denotational semantics (Lemma 50);
2. it directly reflects the behavior of the operational semantics through a coinductive characterization (Proposition 53).

Therefore it follows that each behavior given by a branch of the operational semantics derivation is step-by-step captured by the stepwise denotational semantics. We aim at showing that operational and denotational semantics derive and denote the same runs.

*Remark* 48. All of these properties are valid for general tag relations. Compared to the original TESL language which only considers affine tag relations, our model is agnostic to this restriction and considers any arithmetic relation. Yet, they are decidable only for some fragments of arithmetic, such as affine relations.

## 6.1 INTERMEDIATE SEMANTICS AND EXPANSION PROPERTIES

In Definition 40 we have defined a denotational semantics to mathematically describe all satisfying runs of a specification. Our goal is to decompose it into smaller parts in order to reason step-by-step over one specific instant, instead of runs as a whole. Our first step is to weaken the previous definition of the interpretation of TESL formulae. In Definition 49 we give a slight variation of this definition where the required behavior is ensured only from a given step $i$, no matter what happens before. This is depicted by quantification over index $n$, minored by parameter $i$.

**Definition 49 (*Stepwise Interpretation of TESL* formulae*)**

The *stepwise interpretation* of a TESL$^\star$ formula $\Psi$, denoted with $[\![\Psi]\!]^{\geqslant i}_{\mathsf{TESL}}$, is defined as

$$[\![\psi_0 \wedge \ldots \wedge \psi_k]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} [\![\psi_0]\!]^{\geqslant i}_{\mathsf{TESL}} \cap \ldots \cap [\![\psi_k]\!]^{\geqslant i}_{\mathsf{TESL}}$$

$$[\![K_1 \texttt{ sporadic } \tau \texttt{ on } K_2]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \exists n \geqslant i \quad \mathsf{ticks}(\rho_n(K_1)) \text{ is true} \quad \text{and} \quad \mathsf{tag}(\rho_n(K_2)) = \tau\}$$

$$[\![\texttt{tag relation } \lfloor K_1, K_2 \rfloor \in R]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \geqslant i \quad \mathsf{tag}(\rho_n(K_1)) \text{ and } \mathsf{tag}(\rho_n(K_2)) \text{ are in relation } R\}$$

$$[\![K_1 \texttt{ implies } K_2]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \geqslant i \quad \mathsf{ticks}(\rho_n(K_1)) \quad \text{implies} \quad \mathsf{ticks}(\rho_n(K_2))\}$$

$$[\![K_{\mathsf{master}} \texttt{ time delayed by } \delta\tau \texttt{ on } K_{\mathsf{meas}} \texttt{ implies } K_{\mathsf{slave}}]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \geqslant i \quad \mathsf{ticks}(\rho_n(K_{\mathsf{master}})) \text{ implies } \exists m \geqslant n \quad \mathsf{ticks}(\rho_m(K_{\mathsf{slave}})) \text{ and } \mathsf{tag}(\rho_m(K_{\mathsf{meas}})) = \mathsf{tag}(\rho_n(K_{\mathsf{meas}})) + \delta\tau\}$$

$$[\![K_1 \texttt{ weakly precedes } K_2]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \left\{\rho \in \Sigma^\infty \;\middle|\; \forall n \geqslant i \quad \mathsf{card}\left\{j \leqslant n \;\middle|\; \mathsf{ticks}(\rho_j(K_1))\right\} \geqslant \mathsf{card}\left\{j \leqslant n \;\middle|\; \mathsf{ticks}(\rho_j(K_2))\right\}\right\}$$

$$[\![K_1 \texttt{ strictly precedes } K_2]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \left\{\rho \in \Sigma^\infty \;\middle|\; \forall n \geqslant i \quad \mathsf{card}\left\{j < n \;\middle|\; \mathsf{ticks}(\rho_j(K_1))\right\} \geqslant \mathsf{card}\left\{j \leqslant n \;\middle|\; \mathsf{ticks}(\rho_j(K_2))\right\}\right\}$$

$$[\![K_1 \texttt{ kills } K_2]\!]^{\geqslant i}_{\mathsf{TESL}} \stackrel{\text{def}}{=} \{\rho \in \Sigma^\infty \mid \forall n \geqslant i \quad \mathsf{ticks}(\rho_n(K_1)) \quad \text{implies} \quad \forall p \geqslant n \quad \neg\, \mathsf{ticks}(\rho_p(K_2))\}$$

It is trivial that the stepwise interpretation links to the corresponding denotational interpretation by starting at instant index 0 (Lemma 50).

**Lemma 50 (*Start step*)**

*For any TESL* formula $\Psi$,*

$$[\![\Psi]\!]_{\mathsf{TESL}} = [\![\Psi]\!]^{\geqslant 0}_{\mathsf{TESL}}.$$

**Proof:** The runs in $[\![\Psi]\!]^{\geqslant 0}_{\mathsf{TESL}}$ contains all instants of index greater or equal to $0$, yet indices are always greater or equal to $0$. □

Likewise, associativity, commutativity, idempotency and neutrality are preserved (Lemma 51).

**Lemma 51 (*Stepwise Associativity, Commutativity, Idempotence and Neutrality*)**

*For any TESL specification $\Psi$ and any instant index $i$,*

$$
\begin{array}{lcll}
[\![(\Psi_1 \wedge \Psi_2) \wedge \Psi_3]\!]^{\geqslant i}_{\mathsf{TESL}} & = & [\![\Psi_1 \wedge (\Psi_2 \wedge \Psi_3)]\!]^{\geqslant i}_{\mathsf{TESL}} & \text{(associativity)} \\
[\![\Psi_1 \wedge \Psi_2]\!]^{\geqslant i}_{\mathsf{TESL}} & = & [\![\Psi_2 \wedge \Psi_1]\!]^{\geqslant i}_{\mathsf{TESL}} & \text{(commutativity)} \\
[\![\Psi \wedge \Psi]\!]^{\geqslant i}_{\mathsf{TESL}} & = & [\![\Psi]\!]^{\geqslant i}_{\mathsf{TESL}} & \text{(idempotency)} \\
[\![\varnothing \wedge \Psi]\!]^{\geqslant i}_{\mathsf{TESL}} & = & [\![\Psi]\!]^{\geqslant i}_{\mathsf{TESL}} & \text{(neutrality)}
\end{array}
$$

**Proof:** By set-theoretical reasoning and usual properties of $\cap$. □

The next observation that leads our study is that quantifiers we use in the denotational definition of TESL are bounded by integers. This allows to unfold them into smaller parts. This is depicted in Proposition 53 that gives the link between both operational and denotational semantics. This unfolding property shows a pattern that strongly resembles that of reduction rules given by the operational semantics. In particular, coinductively unfolding the denotational semantics is "similar" to deriving a reduction step of the operational semantics.

*Remark* 52. This result strongly resembles to *expansion laws* in LTL. Indeed, the mechanism in Proposition 53 unfolds the same way that LTL formulae are unfolded. Recall that $\#^{\leqslant}\_$ denotes a counter of ticks as in Definition 41.

**Proposition 53 (*Coinductive Unfolding*)**

*The stepwise interpretation can be coinductively unfolded as*

$$
\begin{aligned}
&[\![ \mathsf{K}_1 \ \texttt{sporadic}\ \tau\ \texttt{on}\ \mathsf{K}_2 ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \mathsf{K}_1 \ \texttt{sporadic}\ \tau\ \texttt{on}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&\quad \cup \quad [\![ \mathsf{K}_1 \Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_2 \Downarrow_i \tau ]\!]_{\mathsf{prim}} \\
&[\![ \texttt{tag relation}\ \lfloor \mathsf{K}_1, \mathsf{K}_2 \rfloor \in R ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \lfloor \mathsf{tvar}_i^{\mathsf{K}_1}, \mathsf{tvar}_i^{\mathsf{K}_2} \rfloor \in R ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \texttt{tag relation}\ \lfloor \mathsf{K}_1, \mathsf{K}_2 \rfloor \in R ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&[\![ \mathsf{K}_1 \ \texttt{implies}\ \mathsf{K}_2 ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \mathsf{K}_1 \not\Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{implies}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&\quad \cup \quad [\![ \mathsf{K}_1 \Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_2 \Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{implies}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&[\![ \mathsf{K}_{master} \ \texttt{time delayed by}\ \delta\tau\ \texttt{on}\ \mathsf{K}_{meas}\ \texttt{implies}\ \mathsf{K}_{slave} ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \mathsf{K}_{master} \not\Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_{master} \ \texttt{time delayed by}\ \delta\tau\ \texttt{on}\ \mathsf{K}_{meas}\ \texttt{implies}\ \mathsf{K}_{slave} ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&\quad \cup \quad [\![ \mathsf{K}_{master} \Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_{slave} \ \texttt{sporadic}\ \mathsf{tvar}_i^{\mathsf{K}_{meas}} + \delta\tau\ \texttt{on}\ \mathsf{K}_{meas} ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\qquad\qquad\qquad\quad \cap \quad [\![ \mathsf{K}_{master} \ \texttt{time delayed by}\ \delta\tau\ \texttt{on}\ \mathsf{K}_{meas}\ \texttt{implies}\ \mathsf{K}_{slave} ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&[\![ \mathsf{K}_1 \ \texttt{weakly precedes}\ \mathsf{K}_2 ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \lceil \#^{\leqslant i}\mathsf{K}_1, \#^{\leqslant i}\mathsf{K}_2 \rceil \in \geqslant ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{weakly precedes}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&[\![ \mathsf{K}_1 \ \texttt{strictly precedes}\ \mathsf{K}_2 ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \lceil \#^{< i}\mathsf{K}_1, \#^{\leqslant i}\mathsf{K}_2 \rceil \in \geqslant ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{strictly precedes}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&[\![ \mathsf{K}_1 \ \texttt{kills}\ \mathsf{K}_2 ]\!]^{\geqslant i}_{\mathsf{TESL}} \\
&\quad = \quad [\![ \mathsf{K}_1 \not\Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{kills}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}} \\
&\quad \cup \quad [\![ \mathsf{K}_1 \Uparrow_i ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_2 \not\Uparrow_{\geqslant i} ]\!]_{\mathsf{prim}} \quad \cap \quad [\![ \mathsf{K}_1 \ \texttt{kills}\ \mathsf{K}_2 ]\!]^{\geqslant i+1}_{\mathsf{TESL}}
\end{aligned}
$$

**Proof:** By unfolding universal and existential quantifiers and substituting parts with Definition 29 and Definition 49. The rules of state that $[\![\Psi]\!]^{\geqslant i}_{\mathsf{TESL}}$ can be decomposed in what happens at index $i$ and what happens starting from index $i+1$. □

Again, we find the pattern past-present-future:

- the past is described by $[\![\_]\!]_{\mathsf{prim}}$, which is the denotation of fixed primitives,
- the present by $[\![\_]\!]^{\geqslant i}_{\mathsf{TESL}}$, which denotes runs that are instantaneously valid,
- and the future by $[\![\_]\!]^{\geqslant i+1}_{\mathsf{TESL}}$, which denotes runs that are valid in the future instants.

For instance, when eliminating a `time delayed` formula, Rule $\mathsf{time-delayed}_{e1}$ and Rule $\mathsf{time-delayed}_{e2}$ are respectively mirrored at denotational level by the union of two sets of runs.

Following the previous observation, we interpret configurations as follows:

**Definition 54 (*Interpretation of Configurations*)**

The interpretation of a configuration $\Gamma \models_n \Psi \triangleright \Phi$ is

$$[\![\Gamma \models_n \Psi \triangleright \Phi]\!]_{\mathsf{config}} \stackrel{\text{def}}{=} [\![\Gamma]\!]_{\mathsf{prim}} \cap [\![\Psi]\!]^{\geqslant n}_{\mathsf{TESL}} \cap [\![\Phi]\!]^{\geqslant n+1}_{\mathsf{TESL}}.$$

Consequently, it is trivial to show that the interpretation of a TESL$^\star$ formula $\Psi$ is the same as the interpretation of the initial configuration starting at $\Psi$ (Lemma 55).

**Lemma 55 (*Start Configuration*)**

*For any TESL$^\star$ formula $\Psi$, we have*

$$[\![\Psi]\!]_{\mathsf{TESL}} = [\![\varnothing \models_0 \Psi \triangleright \varnothing]\!]_{\mathsf{config}}.$$

**Proof:** By unfolding Definition 54: $[\![\varnothing]\!]_{\mathsf{prim}}$ and $[\![\varnothing]\!]^{\geqslant n+1}_{\mathsf{TESL}}$ are the whole set of runs, since $\varnothing$ is not constraining anything, and $[\![\Psi]\!]^{\geqslant 0}_{\mathsf{TESL}}$ is $[\![\Psi]\!]_{\mathsf{TESL}}$ by Lemma 50. □

## 6.2 CERTIFYING DENOTATIONAL AND OPERATIONAL SEMANTICS

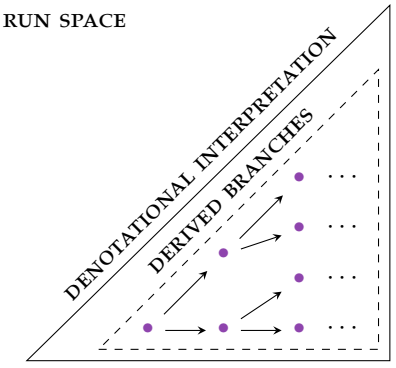



Figure 41: Deriving and denoting in the run space

Such a *coinductive* pattern is useful as it explains the behavior of the operational semantics at a denotational level and bridges the gap between both semantics. The goal of the following theorems is to state that denotational and operational will respectively denote and derive the same runs.

### 6.2.1 *Soundness*

To establish *soundness*, we ensure that any successor configuration contains runs that were indeed coming from the previous configuration. We show that each reduction step is sound, in the sense that a run matched by a successor configuration is inferred from the configuration it originates from (Lemma 56).

**Lemma 56 (*Sound Reduction*)**

*For any reduction* $\Gamma \models_n \Psi \triangleright \Phi \to \Gamma' \models_{n'} \Psi' \triangleright \Phi'$, *we have*

$$[\![\Gamma \models_n \Psi \triangleright \Phi]\!]_{\mathsf{config}} \quad \supseteq \quad [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\mathsf{config}}.$$

**Proof:** Using Definition 43 and 54, and by case analysis on $\to$. The case for $\to_i$ is trivial, the reduction is of the form $\Gamma \models_n \Psi \triangleright \varnothing \to \Gamma \models_{n+1} \varnothing \triangleright \Psi$: the semantics of both sides are the same. In the case for $\to_e$, $n' = n + 1$. The case is solved invoking Proposition 53 to decompose the semantics at instant $n$ using the semantics at instant $n + 1$. □

Finally, soundness generalizes Lemma 55 and 56 to an arbitrary number of reductions starting from the initial configuration.

**Theorem 57 (*Soundness*)**

*Let* $\Psi$ *be a TESL* formula. For all* $k$ *and all configurations* $\Gamma' \models_{n'} \Psi' \triangleright \Phi'$ *such that* $\varnothing \models_0 \Psi \triangleright \varnothing \to^k \Gamma' \models_{n'} \Psi' \triangleright \Phi'$, *we have*

$$[\![\Psi]\!]_{\mathsf{TESL}} \quad \supseteq \quad [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\mathsf{config}}.$$

**Proof:** By induction on $k$. For the base case, when $k = 0$ we have $\Gamma' = \Psi' = \varnothing$ and $n' = 0$. Lemma 55 then tells us that $[\![\Psi]\!]_{\mathsf{TESL}} = [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\mathsf{config}}$. For the inductive case, we suppose that the result is true for $k$ and we consider $k + 1$ reductions:

$$\varnothing \models_0 \Psi \triangleright \varnothing \quad \to^k \quad \Gamma' \models_{n'} \Psi' \triangleright \Phi' \quad \to \quad \Gamma'' \models_{n''} \Psi'' \triangleright \Phi''.$$

The induction hypothesis tells us that $[\![\Psi]\!]_{\mathsf{TESL}} \supseteq [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\mathsf{config}}$, and we can conclude using Lemma 56 and transitivity of $\supseteq$. □

### 6.2.2 *Completeness*

*Completeness* consists of showing that if a run $\rho$ belongs to the denotation of a configuration, this configuration rewrites to one whose denotation also contains $\rho$. To achieve this, we first define an operator that captures direct successors (Definition 58). Then we show that any denoted run can be retrieved in some successor configuration (Lemma 59).

**Definition 58 (*Direct Successors*)**

For any configuration $\Gamma \models_n \Psi \triangleright \Phi$, we define

$$\mathcal{C}_{\text{next}}(\Gamma \models_n \Psi \triangleright \Phi) \stackrel{\text{def}}{=} \left\{ \Gamma' \models_{n'} \Psi' \triangleright \Phi' \;\middle|\; (\Gamma \models_n \Psi \triangleright \Phi) \to (\Gamma' \models_{n'} \Psi' \triangleright \Phi') \right\}.$$

**Lemma 59 (*Complete Direct Successors*)**

*For any configuration* $\Gamma \models_n \Psi \triangleright \Phi$, *we have*

$$\left[\!\left[\Gamma \models_n \Psi \triangleright \Phi\right]\!\right]_{\text{config}} \subseteq \bigcup_{X \in \mathcal{C}_{\text{next}}(\Gamma \models_n \Psi \triangleright \Phi)} \left[\!\left[X\right]\!\right]_{\text{config}}.$$

**Proof:** Similarly to the proof of Lemma 56. We proceed by induction on the number of formulae in $\Psi$. If $\Psi$ is empty, the only possible reduction is $\to_i$, the reduction is of the form $\Gamma \models_n \Psi \triangleright \varnothing \to \Gamma \models_{n+1} \varnothing \triangleright \Psi$: there is only one possible $X$, whose semantics is equal to $\left[\!\left[\Gamma \models_n \Psi \triangleright \Phi\right]\!\right]_{\text{config}}$. If $\Psi$ is not empty, the reduction is a $\to_e$-reduction. The case is solved invoking Proposition 53 to decompose the semantics at instant $n$ using the semantics of the possible reducts at instant $n+1$. □

Hence, completeness holds for an arbitrary number of reductions starting from initial configuration.

**Theorem 60 (*Completeness*)**

*Let* $\Psi$ *be a* TESL$^\star$ *formula and* $\rho$ *a satisfying run, i.e.,* $\rho \in \left[\!\left[\Psi\right]\!\right]_{\text{TESL}}$. *For all* $k$, *there is a configuration* $\Gamma' \models_{n'} \Psi' \triangleright \Phi'$ *such that*

$$\varnothing \models_0 \Psi \triangleright \varnothing \to^k \Gamma' \models_{n'} \Psi' \triangleright \Phi' \quad \textit{and} \quad \rho \in \left[\!\left[\Gamma' \models_{n'} \Psi' \triangleright \Phi'\right]\!\right]_{\text{config}}.$$

**Proof:** By induction on $k$. When $k = 0$, we conclude using Lemma 55. For the inductive case, we assume that the result is true for $k$ and consider the $k+1$ case. From induction hypothesis we find a configuration $\Gamma' \models_{n'} \Psi' \triangleright \Phi'$ such that $\varnothing \models_0 \Psi \triangleright \varnothing \to^k \Gamma' \models_{n'} \Psi' \triangleright \Phi'$ and $\rho \in \left[\!\left[\Gamma' \models_{n'} \Psi' \triangleright \Phi'\right]\!\right]_{\text{config}}$. From Lemma 59, we deduce that there is some $X \in \mathcal{C}_{\text{next}}(\Gamma \models_n \Psi \triangleright \Phi)$ such that $\rho \in \left[\!\left[X\right]\!\right]_{\text{config}}$. This $X$ is the configuration we are looking for to close the inductive case. □

### 6.2.3 *Progress*

*Progress* ensures the increase of the length of the run in construction. We establish that for any instant index, a configuration can be "executed" to produce a run prefix whose length is incremented by 1 (Lemma 61). Then in Theorem 62

we show that for any instant index, a specification can be "executed" to produce a run prefix of such length from the initial configuration.

**Lemma 61 (*Instant Index Increase*)**

*Let* $\Gamma \models_n \Psi \triangleright \Phi$ *be a configuration and* $\rho$ *a satisfying run, i.e.,* $\rho \in [\![\Gamma \models_n \Psi \triangleright \Phi]\!]_{\text{config}}$. *There is* $\Gamma'$, $\Psi'$, $\Phi'$ *and a number of reductions* $k$ *such that*

$$\Gamma \models_n \Psi \triangleright \Phi \to^k \Gamma' \models_{n+1} \Psi' \triangleright \Phi' \quad \textit{and} \quad \rho \in [\![\Gamma' \models_{n+1} \Psi' \triangleright \Phi']\!]_{\text{config}}.$$

**Proof:** By induction on the size of $\Psi$. When $\Psi$ is empty, we can just pick $k = 1$ as the reduction will be a $\to_i$-reduction, and both sides of the reduction will have the same semantics. Now, supposing that the result is true for any $\Psi$ containing $i$ formulae, assume that $\Psi$ contains $i + 1$ formulae. Lemma 59 tells us that there exists a configuration $X$ such that $\Gamma \models_n \Psi \triangleright \Phi \to X$ and $\rho \in [\![X]\!]_{\text{config}}$. Since $\Psi$ is not empty, the reduction is a $\to_e$-reduction and the "present" part of $X$ is now of size $i$: we can apply the induction hypothesis and close the case. □

**Theorem 62 (*Progress*)**

*Let* $\Psi$ *be a TESL* formula and* $\rho$ *a satisfying run, i.e.,* $\rho \in [\![\Psi]\!]_{\text{TESL}}$. *For all* $n'$, *there is* $\Gamma'$, $\Psi'$, $\Phi'$ *and a number of reductions* $k$ *such that*

$$\varnothing \models_0 \Psi \triangleright \varnothing \to^k \Gamma' \models_{n'} \Psi' \triangleright \Phi' \quad \textit{and} \quad \rho \in [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\text{config}}.$$

**Proof:** By induction on $n'$. For the base case, $n = 0$: we can pick $k = 0$, and both sides of the reduction are equal. Suppose now that the result is true for $n'$, and let us prove it for $n' + 1$. We can apply the induction hypothesis: it yields $\Gamma'$, $\Psi'$, $\Phi'$ and a number $k$ such that $\varnothing \models_0 \Psi \triangleright \varnothing \to^k \Gamma' \models_{n'} \Psi' \triangleright \Phi'$ and $\rho \in [\![\Gamma' \models_{n'} \Psi' \triangleright \Phi']\!]_{\text{config}}$. We can then invoke Lemma 61 yielding us the required configuration at instant $n' + 1$. □

## 6.3 HYGGE: A MECHANIZED THEORY IN ISABELLE/HOL

The whole theory developed in this section has been formalized in a proof assistant. It contains approximately 2000 lines of Isabelle/HOL code, and is compliant with Isabelle2018. It is distributed as a free library, named Hygge, at

`https://github.com/heron-solver/hygge`

We give some excerpts that highlight the main ideas presented above. Figure 42 presents a dependency graph that introduces the architecture of the theory as concretized in the proof environment and published online. Note that this formal library tackles the issue of formalization to the extent of fragments and extensions presented in Chapter 4 that have led to TESL*. To keep

this presentation short and simple, we give excerpts and elide the complete definitions.

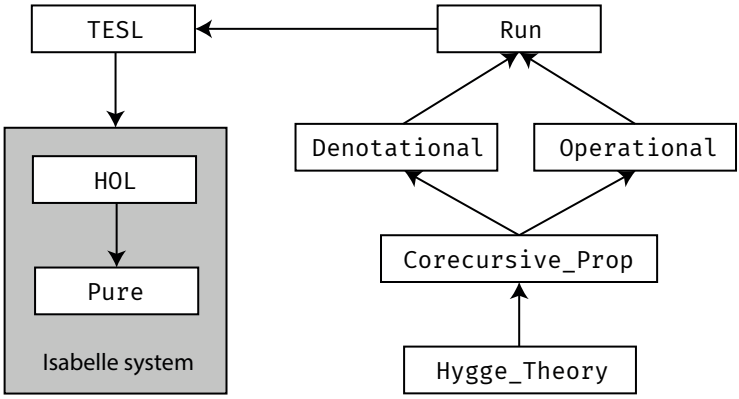



Figure 42: Dependency graph of the mechanized theory

The formal theory starts with `Pure` and `HOL` which are the core libraries of Isabelle/HOL. They consist of the usual definitions and theorems for logical reasoning in higher-order logic including standard data structures, e.g., inductive functions, lists... Then `TESL` and `Run` provide basic definitions and notations used in the formalization of the language as in Section 3.1 and Section 3.2. On one hand, `Denotational` gives the denotational semantics as in Section 3.3.1 and Section 4.2.1. On the other hand, the theory `Operational` gives the operational semantics as in Section 3.3.2 and Section 4.2.3. The link between both theories as described earlier in Chapter 6 is made in `Corecursive_Prop` which gives the main coinductive nature of the semantics. Finally, `Hygge_Theory` states the key properties of soundness, completeness, progress and local termination.

### 6.3.1 *Basic Types and Definitions of the Theory*

Usual datatypes are the basic constructs that serve to represent TESL* terms. A formula is given as a list of atomic formulae.

```
datatype 'τ TESL_atomic =
    SporadicOn     "clock" "'τ tag_expr"  "clock"
  | TagRelation    "clock" "clock" "('τ tag_const × 'τ tag_const) ⇒ bool"
  | Implies        "clock" "clock"
  | ImpliesNot     "clock" "clock"
  | TimeDelayedBy  "clock" "'τ tag_const" "clock" "clock"
  | WeaklyPrecedes "clock" "clock"
  | StrictlyPrecedes "clock" "clock"
  | Kills          "clock" "clock"

type_synonym 'τ TESL_formula = "'τ TESL_atomic list"
```

A timestamp is also defined by a datatype which has structural properties: plus, minus, times, divide, inverse, order, total order. This reflects precisely the conditions applied to tags which state that their are a totally ordered field as required in Definition 20. This mechanism is achieved through type class instanciations.

```
datatype 'τ tag_const =
    TConst   'τ                    ("τcst")
instantiation tag_const :: (plus)plus
instantiation tag_const :: (minus)minus
instantiation tag_const :: (times)times
instantiation tag_const :: (divide)divide
instantiation tag_const :: (inverse)inverse
instantiation tag_const :: (order)order
instantiation tag_const :: (linorder)linorder
```

In the same way, we define primitives as seen in Definition 29 and 41.

```
datatype 'τ constr =
    Timestamp     "clock"    "instant_index" "'τ tag_expr"                    ("_ ⇓ _ @ _")
  | Ticks         "clock"    "instant_index"                                  ("_ ⇑ _")
  | NotTicks      "clock"    "instant_index"                                  ("_ ¬⇑ _")
  | NotTicksUntil "clock"    "instant_index"                                  ("_ ¬⇑ < _")
  | NotTicksFrom  "clock"    "instant_index"                                  ("_ ¬⇑ ≥ _")
  | TagArith      "tag_var" "tag_var" "('τ tag_const × 'τ tag_const) ⇒ bool" ("⌊_, _⌋ ∈ . ")
  | TickCntArith  "cnt_expr" "cnt_expr" "(nat × nat) ⇒ bool"                  ("⌈_, _⌉ ∈ _")
  | TickCntLeq    "cnt_expr" "cnt_expr"                                       ("_ ≼ _")

type_synonym 'τ system = "'τ constr list"
```

A configuration as in Definition 31 is a tuple.

```
type_synonym 'τ config = "'τ system * instant_index * 'τ TESL_formula * 'τ TESL_formula"
```

Finally, a run as described in Definition 20 is defined as a new type `'τ run`, which is a semantic subtype of `nat ⇒ 'τ instant`. It restricts to the only monotonic runs with respect to each clock timeline. Indeed, we require that time (as depicted by timestamps) does not rollback.

```
abbreviation hamlet where "hamlet ≡ fst"
abbreviation time where "time ≡ snd"
type_synonym 'τ instant = "clock ⇒ (bool × 'τ tag_const)"
typedef (overloaded) 'τ::linordered_field run =
  "{ ϱ::nat ⇒ 'τ instant. ∀c. mono (λn. time (ϱ n c)) }"
```

### 6.3.2 *Denotational and Operational Semantics*

The denotational semantics is defined as an interpretation function from TESL* terms to the type of runs, as done in Definition 25 and 40.

```
fun TESL_interpretation_atomic
    :: "('τ::linordered_field) TESL_atomic ⇒ 'τ run set" ("⟦ _ ⟧TESL") where
    "⟦ K1 sporadic ⟨τ⟩ on K2 ⟧TESL =
        { ϱ. ∃n::nat. hamlet ((Rep_run ϱ) n K1) = True
              ∧ time ((Rep_run ϱ) n K2) = τ }"
  | "⟦ K1 sporadic ⟨τvar(Ki, ni) ⊕ δτ⟩ on K2 ⟧TESL =
        { ϱ. ∃n::nat. hamlet ((Rep_run ϱ) n K1) = True
              ∧ time ((Rep_run ϱ) n K2) = time ((Rep_run ϱ) ni Ki) + δτ }"
  | "⟦ time-relation ⌊K1, K2⌋ ∈ R ⟧TESL =
        { ϱ. ∀n::nat. R (time ((Rep_run ϱ) n K1), time ((Rep_run ϱ) n K2)) }"
  | "⟦ master implies slave ⟧TESL =
        { ϱ. ∀n::nat. hamlet ((Rep_run ϱ) n master) ⟶
                      hamlet ((Rep_run ϱ) n slave) }"
```

On the other side, the operational semantics is defined by a inductive property that directly expresses the reduction rules defined in Definition 32, 33 and 43.

```
inductive operational_semantics_intro
:: "('τ::linordered_field) config ⇒ 'τ config ⇒ bool" ("_ ↪i _" 70) where
  instant_i:
  "(Γ, n ⊢ [] ▷ Φ)
     ↪i  (Γ, Suc n ⊢ Φ ▷ [])"

inductive operational_semantics_elim
:: "('τ::linordered_field) config ⇒ 'τ config ⇒ bool" ("_ ↪e _" 70) where
  sporadic_on_e1:
  "(Γ, n ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ)
     ↪e  (Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ))"
| sporadic_on_e2:
  "(Γ, n ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)"
| tagrel_e:
  "(Γ, n ⊢ ((time-relation ⌊K1, K2⌋ ∈ R) # Ψ) ▷ Φ)
     ↪e  ((⌊τvar(K1, n), τvar(K2, n)⌋ ∈ R) # Γ), n
                                  ⊢ Ψ ▷ ((time-relation ⌊K1, K2⌋ ∈ R) # Φ))"
| implies_e1:
  "(Γ, n ⊢ ((K1 implies K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # Φ))"
| implies_e2:
  "(Γ, n ⊢ ((K1 implies K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 ⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # Φ))"
```

Then the core property of our formalization is given by the coinductive unfolding (Proposition 53). It describes how unfolding TESL$\varepsilon$ formulae in denotational semantics is similar to deriving runs in operational semantics. Here are a few excerpts that illustrate this mechanization. Proofs are based on elementary set-theoretical reasoning.

```
lemma TESL_interp_stepwise_sporadicon_coind_unfold:
  shows "⟦ K₁ sporadic τ on K₂ ⟧TESL^≥ n =
    ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ⇓ n @ τ ⟧prim
    ∪ ⟦ K₁ sporadic τ on K₂ ⟧TESL^≥ Suc n"
lemma TESL_interp_stepwise_tagrel_coind_unfold:
  shows "⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL^≥ n =
    ⟦ ⌊τvar(K₁, n), τvar(K₂, n)⌋ ∈ R ⟧prim
    ∩ ⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL^≥ Suc n"
lemma TESL_interp_stepwise_implies_coind_unfold:
  shows "⟦ master implies slave ⟧TESL^≥ n =
    (⟦ master ¬⇑ n ⟧prim ∪ ⟦ master ⇑ n ⟧prim ∩ ⟦ slave ⇑ n ⟧prim)
    ∩ ⟦ master implies slave ⟧TESL^≥ Suc n"
lemma TESL_interp_stepwise_timedelayed_coind_unfold:
  shows "⟦ master time-delayed by δτ on measuring implies slave ⟧TESL^≥ n =
    (⟦ master ¬⇑ n ⟧prim
     ∪ ⟦ master ⇑ n ⟧prim
       ∩ ⟦ slave sporadic (τvar(measuring, n) ⊕ δτ) on measuring ⟧TESL^≥ n)
    ∩ ⟦ master time-delayed by δτ on measuring implies slave ⟧TESL^≥ Suc n"
```

### 6.3.3 *Guarantees and Safety Properties*

Finally, soundness (Theorem 57), completeness (Theorem 60), progress (Theorem 62) are given by the following theorems. Their proofs are mainly based on induction over Ψ, i.e., induction over formulae, and induction over the length of run k.

```
theorem soundness:
  assumes "([], 0 ⊢ Ψ ▷ []) ↪^k 𝒮"
  shows "[[ Ψ ]]TESL ⊇ ⟦ 𝒮 ⟧config"
theorem completeness:
  assumes "ϱ ∈ [[ Ψ ]]TESL"
  shows "∃𝒮. (([], 0 ⊢ Ψ ▷ []) ↪^k 𝒮)
             ∧ ϱ ∈ ⟦ 𝒮 ⟧config"
theorem progress:
  assumes "ϱ ∈ [[ Ψ ]]TESL"
  shows   "∃k Γk Ψk Φk. (([], 0 ⊢ Ψ ▷ []) ↪^k (Γk, n ⊢ Ψk ▷ Φk))
                        ∧ ϱ ∈ ⟦ Γk, n ⊢ Ψk ▷ Φk ⟧config"
```

The local termination property (Proposition 34) states that the elimination rules are terminating. To prove so, we prove that such a reduction is well-founded (using predicate `WfP`).

```
theorem instant_computation_termination:
  shows "wfP (λ(𝒮1:: 'a :: linordered_field config) 𝒮2. (𝒮1 ↪e← 𝒮2))"
```

### 6.3.4 *Towards a Certified Solver*

The Isabelle/HOL proof assistant features a code generator, turning HOL theories into corresponding executable programs. One of our approaches was to take advantage of the Isabelle execution engine to generate a certified solver.

Listing 7: Basic example with two sporadic constraints

```
K1 sporadic 1
K1 sporadic 2
K1 implies K2
```

The above specification is a small example that illustrates our experiment.

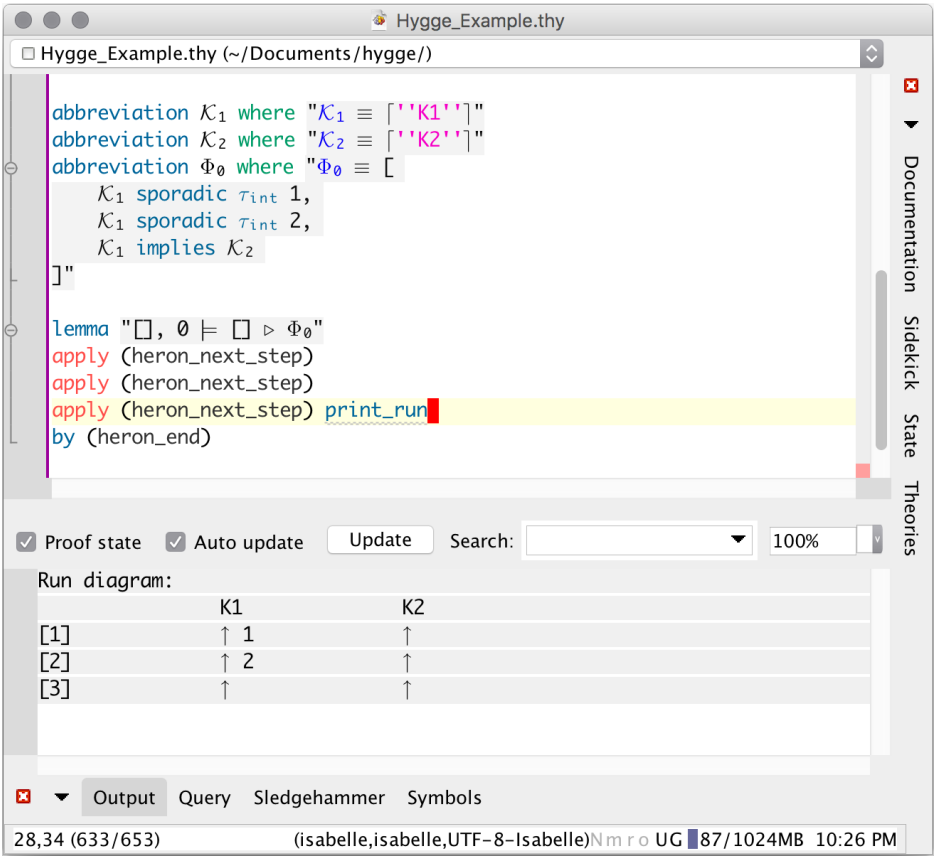



Figure 43: Executing the operational semantics in Isabelle/HOL

Figure 43 shows the implementation we integrated inside the Isabelle proof environment. The solver consists of Standard ML code based upon the Isabelle calculus engine and uses parts of the Eisbach module [MMW16]. From the operational rules, prefixes of symbolic runs are generated with a tactic

`heron_next_step` that mimics the computation of a simulation step as in Definition 37. However, due to a design choice of reusing existing modules and avoiding adhoc unsafe tweaks, it appeared that computing runs was hardly feasible. That was highly reflected by Isabelle computing states each time it attempts at backtracking, instead of saving them. This is even worsened by the semantic subtyping on runs which ensures that tags on clocks are monotonic as shown in Table 2.

| NUMBER OF STEPS | TIME (IN SEC) |
|---|---|
| 1 | 57.2 |
| 2 | Timeout |
| 3 | Timeout |

Table 2: Generating run prefixes in the Hygge theory

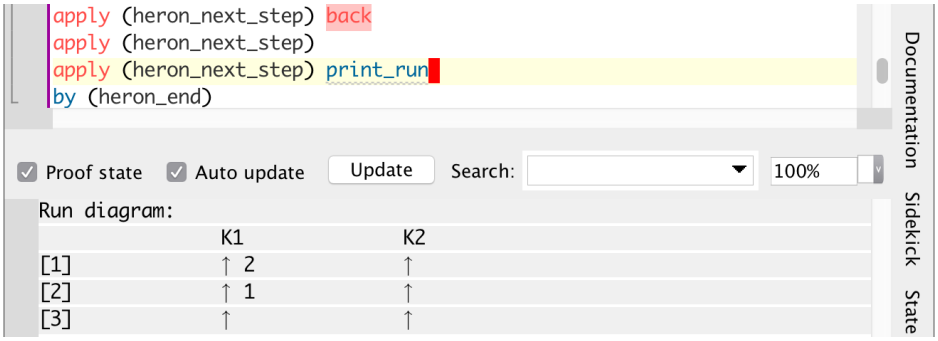



Figure 44: A spurious run

If we discard the property of monotonic run tags that is modeled in the type of concrete runs with semantic subtyping, we gain time but lose consistency in produced runs. Figure 44 illustrates a spurious run that can be produced if we backtrack to exhibit another generated run. Table 3 illustrates time to produce the run in Figure 43.

| NUMBER OF STEPS | TIME (IN SEC) |
|---|---|
| 1 | 0.19 |
| 2 | 1.31 |
| 3 | 5.83 |

Table 3: Generating run prefixes in the Hygge theory ignoring semantic subtyping

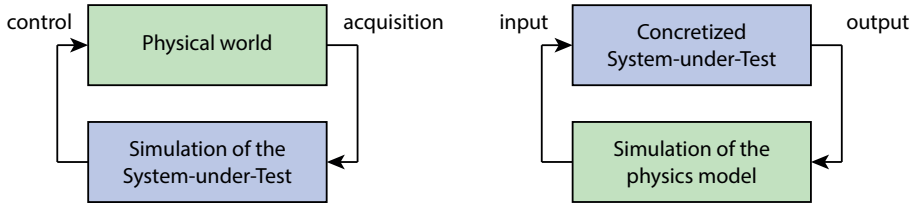
To provide a usable tool in concrete situations, we decided to develop a parallel implementation in Standard ML that is fully described next.

# 7

# APPLICATION TO TESTING AND MONITORING

Testing is a collection of verification processes with the intention to highlight errors and misbehaviors in systems. As an example in the context of Cyber-Physical Systems (CPS), the aim is the observation of a dynamic entity mostly embedded in the physical world given some limited interface. In an analogous manner to the assessment of airline pilots on simulators, testing in the context of simulation also applies for subsystems that need to be assessed against scenarios and environments to ensure their good behaviors. Singularly with the increase of MBD in development processes, the integration of simulation has reduced cost and time for verification procedures. In particular, Hardware-in-the-loop (HIL) simulation is a real-time simulation used to test real development hardware against realistic but virtual stimuli (actuators, sensors, intermediate control). This particularly suits for testing systems in need of critical properties, such as autopilots [JT07] or collision avoidance systems in aircrafts [LLL00]. Simulation can help the goal of testing and monitoring systems. In a continuously-reactive system, simulation is done by means of a closed-loop with feedback where

- either the System Under Test (SUT) that is modeled, is simulated against the physical world that provides reaction through control and acquisition drivers (Figure 45a),
- either the SUT is a hardware code or circuit concretized from the model, and that is tested against a simulated physics model (Figure 45b).




(a) Simulated system

(b) Simulated physical environment

Figure 45: Heterogeneous simulation as in HIL

Due to the compositional nature of our framework, the separation between submodels (serving as representations for subsystems) is clear and surrounds HIL simulation testing by considering the whole external stimuli as again another parallel subsystem. In this section, we scrutinize the idea of testing and monitoring in the context of simulation. Both approaches may seem similar, they admit a few differences.

TESTING. A specific input is associated with a specific expected output of the system. In this input/output semantics approach, we wait until the system comes to a proper termination and then observe its final output (through predefined limited interfaces).

MONITORING. To go further, the previous idea is broadened and now tracks the system behavior during execution (at runtime) to detect unexpected behaviors. This form of trace semantics is suitable to reactive systems that may loop, or may get stuck in deadlock. Once again, the observation is limited by runtime interfaces.

Our purpose is to provide a testing/monitoring framework with the interest of considering timed aspects of systems. Under some specific conditions, some event shall occur (or not) under time constraints. Such issues have already been investigated in [KT09] in the context of timed automata. Here we study in the context of our framework. Given the operational semantics for TESL that we studied in the previous chapters, the design and implementation of a solver for TESL specifications is straightforward. A collection of specification formulae can be constructively solved to produce runs that satisfy the specification requirements. Additionally, these runs are sound and complete, based on the equivalence with the denotational semantics shown in Chapter 6 to the scale of the TESL$^\star$ fragment.

## 7.1 HERON: A SOLVER FOR TESL SPECIFICATIONS

Since the operational semantics of TESL can be seen as an abstract execution machine, its implementation is a natural result. The prototype solver is called Heron [NBB+17], and is distributed as free software at

`https://github.com/heron-solver/heron`

It 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 [Wee06]. Heron is a standalone command-line interpreter, which takes a TESL specification as input and produces prefixes of satisfying symbolic runs. The solver is complete in the sense discussed in Chapter 6, i.e., it produces all satisfying runs up to a fixed step index. Assuming that the 'future' formula contains no contradiction, this means

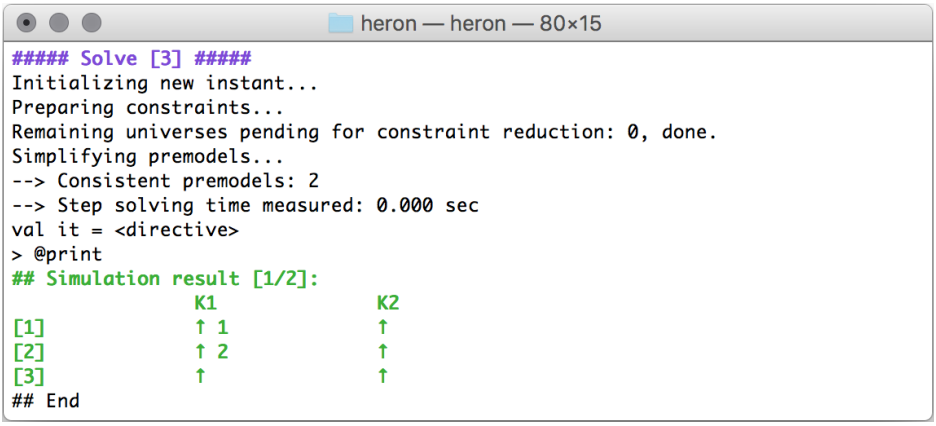



Figure 46: Running the Heron solver on Listing 7

that the satisfying symbolic runs have instances which are *exactly* the prefixes of *all* satisfying concrete runs.

Heron can be used in four modes:

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 SUT observed at its interfaces (see Figure 3). 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 [Tre96] test scenario.

## 7.2 SCENARIO 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 3) 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 [BW16].

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 47. 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 gray 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.

EXAMPLE: based on the specification shown in Listing 2 on page 36, 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
```

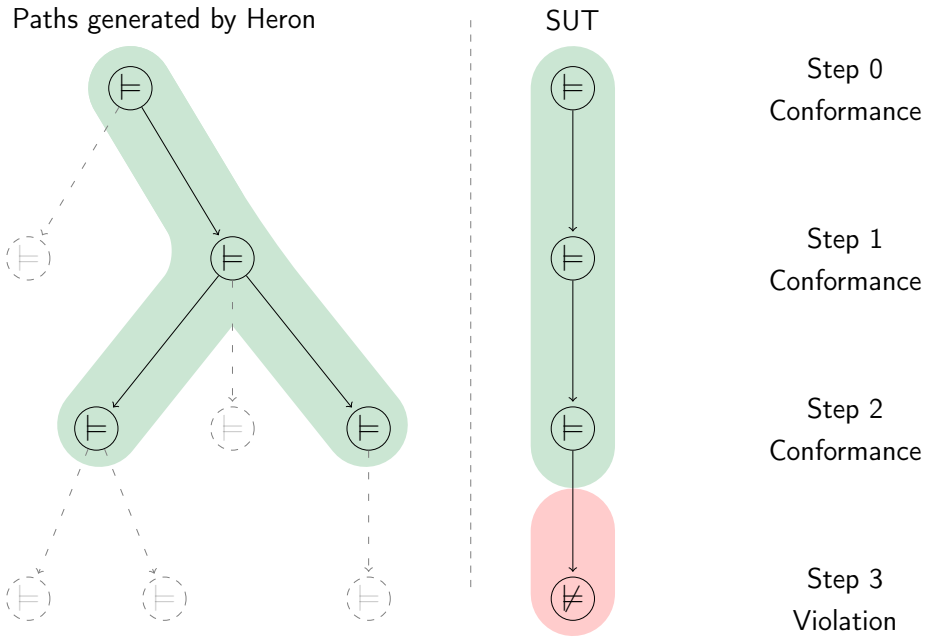



Figure 47: Executing Heron and the SUT in parallel

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 15b:

```
@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 con-

forms" 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:

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

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

```
### ERROR: No further state found.
           Simulation is now stuck in inconsistent mode.
```

## 7.3 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*:

```
7  @driving-clock move
```

After this declaration, Heron may be instrumented by:

```
8  @event-solve 2
```

which leads to the invocation of a constraint solver (Lemma 30) for step 2, which by default chooses for the driving clocks, an input that satisfies the constraints. More sophisticated generation policies could be implemented.

CONFORMANCE: if the future of a configuration becomes empty or *stable*, the observed run "fully conforms" to the TESL specification. A (future) specification is stable, if it represents a Büchi-automaton producing an infinite behavior 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.

## 7.4 PERFORMANCE

Benchmarks have been realized on a conventional laptop computer with an Intel Core™ i5-2520M CPU @ 2.50GHz and 8 GB of RAM. The results are logged in Table 4 based on examples provided by the official gallery of TESL[1]:

- HandWatch: The minute hand of a clockwatch as in Listing 2.
- LightSwitch: A fluorescent light bulb takes some time to become completely lit after switched on, and some other time to be off after being switched off.
- ConcurrentComp: Two CPUs are executing in parallel and on different timeframes to compute the final result $A + B$, provided that each of them is in charge of computing $A$ on one side and $B$ on the other side (as in Listing 6).
- LeapYears: Determine leap years.
- Engine: Ignition in a four-stroke petrol engine with related timeframes for the crankshaft, the camshaft and real time.

The results measure time (in sec or min:sec) and memory usage (in kB) for each specification with respect to three policies and with respect to the number of simulation steps: exhaustive exploration, minimal run (given by the policy `asap`) and system monitoring.

In the first case, the state-space explosion is notably highlighted by timeouts at the first instants. This is a direct consequence of the branching nature of the operational semantics. On the other side, executing the specification (with the minimal run policy) or monitoring an SUT are feasible in reasonable time. Indeed, the only conforming behaviors are preserved, and hence can be handled with reasonable time and space resources.

1 See http://wdi.supelec.fr/software/TESL/Gallery

| 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 4: Benchmarking Heron on TESL official gallery

# 8

# CONCLUSION AND PERSPECTIVES

## 8.1 SUMMARY

The context of this thesis revolves around designing semantic structures for a timed discrete-event specification language for the composition and the simulation of composite models. More specifically, our study was focused on the TESL language that is at the heart of the ModHel'X simulation environment. The initial problem arises when such a semantic framework needs to qualify for two main properties:

COMPOSITIONALITY The semantic composition of two models yields the semantics of the supermodel.

EXECUTABILITY It should be constructive (in other words, runnable) to allow the derivation of execution traces.

Our solution tackles the issues in question with the introduction and the study of three language variants named TESL$\varepsilon$, TESL$^\star$ and TESL that target specification and verification of timed systems. The first two variants, TESL$\varepsilon$ and TESL$^\star$, both admit a denotational and an operational semantics. They are linked and have been proven to be equivalent by means of an intermediate semantics that is stepwise compared to the denotational one. This has led us to a property exhibiting the reflection between both denotational and operational semantics, and allows the derivation of properties ensuring the well-foundedness of our approaches: soundness, completeness, progress and local termination. In an effort to fully validate our approaches, our results are fully mechanized using the Isabelle/HOL proof assistant.

Finally, the addition of the remaining operational rules that cover the complete language in the TESL variant allows us to develop a constructive solver of the complete original language. It is an efficient implementation of the complete operational semantics that can be used for runtime monitoring and testing. It integrates as well extensions of TESL$^\star$.

## 8.2 PERSPECTIVES

Following our contributions, we foresee several open questions to be addressed.

GENERIC SEQUENTIAL FORMULAE. The structure of formulae presented in the third extension of TESL seems to follow a common pattern in which causality formulae trigger events with respect to a conditioned state. For example, `delayed by` contains the current number of ticks to skip before triggering a clock, while `await` contains a state enumerating clocks that have ticked to keep track of the clocks that need to be awaited before triggering another clock. A monadic approach seems necessary to factorize these components. This would consist of a generic type (lists, integers, tuples, or any of their composition) along with predicates ruling the order of execution:

⟨clock⟩ `implies` ⟨clock⟩ `conditioned by` ⟨predicate⟩ `on` ⟨state⟩
`resettable whenever` ⟨predicate⟩ `initially` ⟨state⟩

FOLDING INFINITE RUNS. In our experiment, we believe that infinite runs can be folded modulo renaming of the tag variables given that tag relations are affine. The direct consequence of this makes TESL a description language for regular behaviors to the degree of abstraction of symbolic tag variables. This would allow static analysis on properties such as deadlock, mutual exclusion...

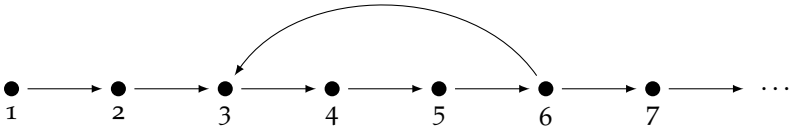



Figure 48: A sequence of configurations with "equivalence" between 6th and 3rd

INTEGRATION AS A UML MARTE SOLVER. As stated in Remark 39, we believe our framework turns the solver into a suitable tool for solving coordination of timed models in UML MARTE [SG13]. Moreover, its foundations are proved to compute the exact satisfying runs of a specification. Hence, it could be integrated to reliably monitor realtime systems for faults and error detection.

CERTIFIED SOLVERS AT A LARGER SCALE The language variants we present seem close to existing models but we believe are more expressive while enjoying good properties. A further study should highlight the power of expressiveness of TESL compared to other languages and paradigms. Furthermore, the coinductive characterization that serves our properties would likely apply in the context of other languages, and as we exhibited, the process of mechanizing and generating code for such formalized theories opens doors towards certified solvers for timed languages.

# BIBLIOGRAPHY


[ABG+08] S. Akshay, Benedikt Bollig, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Distributed timed automata with independently evolving clocks. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 82–97, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996.

[AD94a] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183 – 235, 1994.

[AD94b] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[And09] Charles André. Syntax and Semantics of the Clock Constraint Specification Language (CCSL). Research Report RR-6925, INRIA, 2009.

[BCC+08] Albert Benveniste, Benoît Caillaud, Luca Carloni, Paul Caspi, and Alberto Sangiovanni-Vincentelli. Composing Heterogeneous Reactive Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4), 2008.

[Ber00] Gérard Berry. The foundations of Esterel. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 425–454. MIT Press, Cambridge, MA, USA, 2000.

[Ber07] Gérard Berry. Scade: Synchronous design and validation of embedded control software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.

[BHJM11] F. Boulanger, C. Hardebolle, C. Jacquet, and D. Marcadet. Semantic adaptation for models of computation. In *2011 Eleventh International Conference on Application of Concurrency to System Design*, pages 153–162, June 2011.

[BJHP14] Frédéric Boulanger, Christophe Jacquet, Cécile Hardebolle, and Iuliana Prodan. TESL: a language for reconciling heterogeneous

execution traces. In *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, pages 114–123, Lausanne, Switzerland, Oct 2014.

[BW16] Achim D. Brucker and Burkhart Wolff. 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*, pages 17–36, 2016.

[CCF+15] Benoit Combemale, Betty H.C. Cheng, Robert B. France, Jean-Marc Jezequel, and Bernhard Rumpe. *Globalizing Domain-Specific Languages*, volume 9400 of *LNCS, Programming and Software Engineering*. Springer International Publishing, 2015.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[DAG14] Julien Deantoni, Charles André, and Régis Gascon. CCSL denotational semantics. Research Report RR-8628, Inria, November 2014.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120, September 2001.

[Dal04] D. Dalen. *Logic and Structure*. Universitext (1979). Springer, 2004.

[EJL+03] J. Eker, J. W. Janneck, E. A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[GBBG87] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Synchronous data flow programming with the language SIGNAL. *IFAC Proceedings Volumes*, 20(2):359 – 364, 1987. 2nd IFAC Workshop on Adaptive Systems in Control and Signal Processing 1986, Lund, Sweden, 30 June-2 July 1986.

[GBFA13] Manuel Garnacho, Jean-Paul Bodeveix, and Mamoun Filali-Amine. A mechanized semantic framework for real-time systems. In Víctor Braberman and Laurent Fribourg, editors, *Formal Modeling and Analysis of Timed Systems: 11th International Conference, FORMATS 2013, Buenos Aires, Argentina, August 29-31, 2013. Proceedings*, pages 106–120, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Haf13] Florian Haftmann. Haskell-style type classes with Isabelle/Isar, 2013.

[HB09] Cécile Hardebolle and Frédéric Boulanger. Multi-formalism modelling and model execution. *International Journal of Computers and their Applications*, 31(3):193–203, July 2009. Special Issue on the International Summer School on Software Engineering.

[HCOH93] Roger Hale, Rachel Cardell-Oliver, and John Herbert. An embedding of timed transition systems in HOL. *Formal Methods in System Design*, 3(1):151–174, Aug 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Introduction to automata theory, languages, and computation, 2nd edition. *SIGACT News*, 32(1):60–65, March 2001.

[JT07] Dongwon Jung and Panagiotis Tsiotras. Modeling and hardware-in-the-loop simulation for a small unmanned aerial vehicle. In *AIAA Infotech@ Aerospace 2007 Conference and Exhibit*, page 2768, 2007.

[Kah74] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[KS05] Antonín Kučera and Jan Strejček. The stuttering principle revisited. *Acta Informatica*, 41(7–8):415–434, 2005.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *IFIP Congress on Information Processing*, pages 657–668, 1983.

[LGGTB15] Paul Le Guernic, Thierry Gautier, Jean-Pierre Talpin, and Loïc Besnard. Polychronous automata. In *TASE 2015, 9th International Symposium on Theoretical Aspects of Software Engineering*, pages 95–102, Nanjing, China, September 2015. IEEE Computer Society.

[LLL00] C. Livadas, J. Lygeros, and N. A. Lynch. High-level modeling and analysis of the traffic alert and collision avoidance system (tcas). *Proceedings of the IEEE*, 88(7):926–948, July 2000.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[LSV96] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. The tagged signal model a preliminary version of a denotational

framework for comparing models of computation. Technical Report UCB/ERL M96/33, EECS Department, University of California, Berkeley, 1996.

[LT93] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[Mal08] Frédéric Mallet. Clock constraint specification language: specifying clock constraints with UML/Marte. *Innovations in Systems and Software Engineering*, 4(3):309–314, 2008.

[MDADS10] Frédéric Mallet, Julien Deantoni, Charles André, and Robert De Simone. The Clock Constraint Specification Language for building timed causality models. *Innovations in Systems and Software Engineering*, 6(1-2):99–106, March 2010.

[MMW16] Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, Mar 2016.

[MSE04] P. J. Mosterman, J. Sztipanovits, and S. Engell. Computer-automated multiparadigm modeling in control systems technology. *IEEE Transactions on Control Systems Technology*, 12(2):223–234, March 2004.

[Mul99] L. Muliadi. Discrete event modeling in ptolemy ii. Technical Report UCB/ERL M99/29, EECS Department, University of California, Berkeley, 1999.

[MV04] Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *SIMULATION*, 80(9):433–450, 2004.

[NBB+17] Hai Nguyen Van, Thibaut Balabonski, Frédéric Boulanger, Chantal Keller, Benoît Valiron, and Burkhart Wolff. A symbolic operational semantics for TESL - with an application to heterogeneous system testing. In *Formal Modeling and Analysis of Timed Systems - 15th International Conference, FORMATS 2017, Berlin, Germany, September 5-7, 2017, Proceedings*, pages 318–334, 2017.

[Nip03] Tobias Nipkow. Structured proofs in Isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, pages 259–278, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[PFD11] Marie-Agnès Peraldi-Frati and Julien Deantoni. Scheduling Multi Clock Real Time Systems: From Requirements to Implementation. In *International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, number 14th in IEEE international Symposium on Object/Component/service Oriented Real-Time Distributed Computing, page 50; 57, Newport Beach, United States, March 2011. IEEE computer society. NewPort Beach.

[PM01] Christine Paulin-Mohring. Modelisation of timed automata in Coq. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software: 4th International Symposium, TACS 2001 Sendai, Japan, October 29–31, 2001 Proceedings*, pages 298–315, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[SG13] B. Selic and S. Gerard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE: Developing Cyber-Physical Systems*. The MK/OMG Press. Elsevier Science, 2013.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[TBG+13] Jean-Pierre Talpin, Jens Brandt, Mike Gemünde, Klaus Schneider, and Sandeep Shukla. Constructive polychronous systems. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, volume 7734 of *Lecture Notes in Computer Science*, San Diego, CA, United States, January 2013. Springer.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[VLDCM15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (BCOoL). In *18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015)*, August 2015.

[Wee06] Stephen Weeks. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML*, ML '06, pages 1–1, New York, NY, USA, 2006. ACM.

[Wel99] J.B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111 – 156, 1999.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[ZM16] M. Zhang and F. Mallet. An executable semantics of Clock Constraint Specification Language and its applications. In *Formal Techniques for Safety-Critical Systems: 4th International Workshop, FTSCS 2015*, pages 37–51, Cham, 2016. Springer.

# LIST OF FIGURES

## LIST OF TABLES

## LISTINGS

# LIST OF DEFINITIONS

## LIST OF THEOREMS AND LEMMAS

# LIST OF SYMBOLS

| | |
|---|---|
| $\mathbb{N}$ | Natural integers |
| $\mathbb{Z}$ | Mathematical integers |
| $\mathbb{Q}$ | Rationals |
| $\mathbb{R}$ | Reals |
| $\mathbb{U}$ | Unit |
| $\mathbb{B}$ | Booleans |
| | |
| $\sigma$ | Instant |
| $\Sigma$ | Set of instants |
| $\mathfrak{n}$ | Instant index, *i.e.* integer |
| | |
| $\rho$ | Run |
| $\Sigma^{\propto}$ | Set of runs |
| | |
| $\gamma$ | Run primitive |
| $\Gamma$ | Run context, *i.e.* set of run primitives |
| $\_ \Uparrow \_$ | Primitive for occurring (ticking) event |
| $\_ \not\Uparrow \_$ | Primitive for absent event |
| $\_ \not\Uparrow_{\geqslant} \_$ | Primitive for dead event |
| $\_ \Downarrow \_ \; \_$ | Primitive for tagged event |
| $\lfloor \_, \_ \rfloor \in \_$ | Arithmetic relation between tags |
| $\lceil \_, \_ \rceil \in \_$ | Arithmetic relation between tick counters |
| $\#^{\leqslant} \_ \_$ | Tick counter from the past |
| $\#^{<} \_ \_$ | Tick counter strictly from the past |
| | |
| $\mathsf{K}$ | Clock |
| $\mathbb{K}$ | Set of clocks |
| | |
| $\tau$ | Tag (also called timestamp) |
| $\mathbb{T}$ | Domain of tags |
| $\mathsf{tvar}_{\_}$ | Tag variable |

| | |
|---|---|
| $\varphi, \psi$ | Atomic TESL formula |
| $\Phi, \Psi$ | TESL formula, *i.e.* set of atomic formulae |
| $R$ | Arithmetic relation |
| $\_ \models_{\_} \_ \triangleright \_$ | Configuration |
| $\rightarrow_{i}$ | Introduction Rule |
| $\rightarrow_{e}$ | Elimination Rule |
| $\rightarrow$ | Reduction |
| $\rightarrow_{\lightning}$ | Simulation step |
| $[\![\_]\!]^{-}_{\mathsf{cnt}}$ | Evaluation of a tick counter expression |
| $[\![\_]\!]_{\mathsf{prim}}$ | Interpretation of a run context |
| $[\![\_]\!]_{\mathsf{TESL}}$ | Interpretation of a TESL formula |
| $[\![\_]\!]_{\mathsf{config}}$ | Interpretation of a configuration |

# LIST OF ACRONYMS

| | |
|---|---|
| CAN | Controller Area Network. 42 |
| CCSL | Clock Constraint Specification Language. 7, 16, 46, 55, 57 |
| CPS | Cyber-Physical Systems. 89 |
| DE | Discrete Events. 42 |
| HIL | Hardware-in-the-loop. 89, 90 |
| HOL | Higher Order Logic. 29–31 |
| MARTE | Modeling and Analysis of Real-Time and Embedded systems. 16, 17, 98 |
| MBD | Model-Based Design. 3, 89 |
| MoC | Model of Computation. 15, 17, 19 |
| ND | Natural Deduction. 28, 30, 31 |
| ODE | Ordinary Differential Equation. 15 |
| PID | Proportional–Integral–Derivative. 3 |
| SDF | Synchronous Data Flow. 16, 42–44 |
| SUT | System Under Test. 89, 91–93, 95 |
| TESL | Tagged Events Specification Language. 7, 8, 10, 17–19, 36, 37, 40, 46–48, 55, 61, 65, 73, 75–77, 90, 97, 98, 104 |
| TFSM | Timed Finite State Machine. 42, 43 |
| UML | Unified Modeling Language. 16, 98 |

# ACKNOWLEDGMENTS

*Whatever happens, happens for the best.*

First and foremost, there may not be enough words on Earth to express deep gratitude to my supervisors Frédéric Boulanger and Burkhart Wolff. Without their advice and constant assistance I may not have been able to conduct this research programme. I am also indebted to Chantal Keller, Lina Ye, Thibaut Balabonski, Safouan Taha, and Benoît Valiron, who volunteered in the project. Their contribution and the amount of time spent with them was truly rewarding. Surely, I will never forget the great generosity of Véronique Benzaken and Evelyne Contejean, whom I have attended a heart-warming welcome since day one: they kept glowing like sunshine every single day at work.

Likewise, I cannot forget the great support of my family and friends who have witnessed challenges I faced: my father Diep, my sister Anh, my brother Minh, my cousin Quynh Anh and my aunt Thúy. Nor can I forget those I have met on the last stretch of this journey and who I consider as part of my own family: Zaza, Alexandra, Huy and Arvid. They all constantly stood by my side: *Gia đình là số một!* I also wish to thank Romain, Julien, Damien, Farah and Marielle for being part of my life. We were seperated apart, but I know you will find happiness.

I would like to thank as well the VALS research group and my fellow (past and present) mates for all the funny and amazing discussions over lunch and smoke breaks (I know that sounds wrong): Catherine L., Stefania D., Nicolas B., Clément F., Houssem H., Marie L., Chuan X., Fabien D., Pierre B., Georges O., George M., Alexandra Z., Romain A., Yakoub N., Frédéric T., but also Régine B., Delphine L., Sylvie B., Sylvain C., Frédéric V. and Kim N. Also, David D. and Aygul J. were were not just colleagues, they were also my friends and I appreciate how supportive they were to me. The technical and administrative staff also gave me invaluable help and I am thankful to Stéphanie D., Martine, Claude, Gladys and Myriam who helped me with the paperwork. Also teaching duties would not have been possible without Sandrine D. at UFR Sciences.

Last but not the least, I am grateful to all the uplifting people that I had the chance to meet on the prefix of my path. They enlightened this road so strongly that I had been able to drive towards success. This starts with school teachers who witnessed my greatest loss and guided me through: Mrs Fontaine, Mr Balbastre, Florence Maignan, Francis Berr, and Isabelle Thevenin. Then, I was lucky enough to have met Olivier Robert and Pierdavide Coïsson from Institut de Physique du Globe de Paris who gave me the opportunity to be part of

something launched at 800 km altitude on orbit. I also wish to thank Sébastien Bardin and Zaynah Dargaye at CEA LIST who gave me a glimpse into the world of research. My grateful thanks are also extended to my past teachers during Bachelors and Masters studies at Université Paris Diderot. Their broad passion for teaching was inspiring: Delia Kesner, Paul-André Melliès, Roberto Di Cosmo, Sedki Boughattas, Yann Regis-Gianas and Gilles Dowek.

I appreciate the amount of time spent by Frédéric Mallet and Stephan Merz for reviewing this work, but also Timothy Bourke for proof-reading it, and finally Catherine Dubois, Marc Pantel, and Mihaela Sighireanu for being part of the final jury that seals the big chapter of my academic career.

## DECLARATION OF AUTHORSHIP

I, Hai Nguyen Van, declare that this thesis titled, "Formalizing Time and Causality in Polychronous Polytimed Models" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

*Orsay, France, September 2018*

Hai Nguyen Van


FACULTÉ
DES SCIENCES
D'ORSAY

université PARIS-SACLAY

CentraleSupélec

PhD thesis defended on September 27th, 2018

| | |
|---|---|
| Frédéric Mallet | Professor at Université Nice Sophia Antipolis |
| Stephan Merz | Senior Researcher at Inria Nancy |
| Catherine Dubois | Professor at ENSIIE |
| Timothy Bourke | Research Scientist at Inria Paris |
| Marc Pantel | Associate Professor at IRIT/INPT, Université de Toulouse |
| Mihaela Sighireanu | Associate Professor at Université Paris Diderot |
| Frédéric Boulanger | Professor at CentraleSupélec |
| Burkhart Wolff | Professor at Université Paris-Sud |

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

# RÉSUMÉ ÉTENDU

L'intégration de composants dans un système peut s'avérer difficile lorsque ces composants ont été conçus selon différents paradigmes ou s'ils se basent sur différents cadres de temps devant être synchronisés. Il peut s'agir d'équations différentielles, d'automates, de réseaux de processus, de graphes de flot de données synchrones, de circuits séquentiels... Cette synchronisation peut être alors dirigée par les évènements (un évènement est provoqué par un autre), ou bien dirigée par le temps (un évènement se produit parce qu'il en est l'heure). En considérant que chaque composant admet son propre cadre de temps et qu'ils peuvent ne pas être reliés, il est possible qu'une unique ligne de temps globale n'existe pas. Cette question de temps non-newtonien intervient aussi bien dans les calculs physiques de systèmes soumis aux ralentissements temporels expliqués par la théorie de la relativité, que dans les calculs de systèmes distribués en informatique.

Dans le cadre industriel existant, ces systèmes sont souvent modélisés par des outils de type UML/-SysML ou Simulink/Stateflow, où non seulement leur sémantique est informelle mais leurs interactions restent complètement implicites et dépendantes du solveur associé au formalisme. Pour palier cela, des environnements plus développés et plus précis, tels que Ptolemy II, ModHel'X/-TESL ont été conçus pour donner une sémantique d'exécution afin de comprendre et de simuler les comportements engendrés par ces modèles hétérogènes. Ces outils sont toutefois limités par leur but premier, qui consiste en la simulation et l'exécution des modèles, et non leur vérification. Nous tentons de dépasser de telles limitations en cherchant une famille d'approches sémantiques plus favorables à notre problématique de vérification de propriétés. Pour cela, nous nous intéressons à la spécification de schémas de synchronisation pour de tels systèmes polychrones et polytemporisés. Notre étude nous a mené à la conception de modèles sémantiques pour un langage temporisé à évènements discrets, appelé TESL et développé par Boulanger et al. Ce langage a été conçu pour coordonner la simulation de modèles composites et pour tester l'intégration de systèmes.

Dans cette thèse, nous présentons une famille de sémantiques répondant chacune à des problématiques spécifiques. Tout d'abord, nous donnons une sémantique dénotationnelle fournissant une compréhension précise et logiquement cohérente du langage. Puis dans une approche d'exécution, nous proposons une sémantique opérationnelle afin de dériver des traces d'exécutions satisfaisant une spécification TESL. Celle-ci a été utilisée pour les problématiques de test des systèmes, à travers l'implantation d'un solveur nommé Heron. Celui-ci permet notamment de mettre en évidence des erreurs et des fautes de comportements qu'un système-sous-test pourrait déclencher. Le solveur calcule précisément et de manière symbolique, tous les comportements qu'un système devrait satisfaire.

Pour résoudre la question de cohérence et de correction de ces règles de sémantique opérationnelle, nous avons également développé une sémantique intermédiaire, qui est dénotationnelle et pas-à-pas. Celle-ci relie les deux sémantiques dénotationnelle et opérationnelle en les mettant en correspondance. Elle explicite leurs fortes similarités par le biais d'une propriété de dépliage coinductif des schémas de formules du langage. Cela nous a permis de dériver et d'établir des propriétés sur la relation entre les deux sémantiques : la correction (la dérivation d'une exécution est dénotationnellement correcte), la complétude (une exécution dénotée est opérationnellement dérivable), le progrès (toute spécification cohérente admet une exécution de longueur arbitraire), et la terminaison locale (le calcul d'un instant d'exécution est terminant). Enfin, dans un effort de rigueur dans nos approches formelles, notre formalisation ainsi que les preuves associées ont été entièrement mécanisées dans l'assistant de preuve Isabelle/HOL.

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

# FORMALISATION DU TEMPS ET DE LA CAUSALITÉ DANS LES MODÈLES POLYCHRONES POLYTEMPORISÉS

L'intégration de composants dans un système peut s'avérer difficile lorsque ces composants ont été conçus selon différents paradigmes ou s'ils se basent sur différents cadres de temps devant être synchronisés. Cette synchronisation peut être dirigée par les évènements (un évènement est provoqué par un autre), ou dirigée par le temps (un évènement se produit parce qu'il en est l'heure). En considérant que chaque composant admet son propre cadre de temps et qu'ils peuvent ne pas être reliés, il est possible qu'une unique ligne de temps globale n'existe pas.

Nous nous intéressons à la spécification de schémas de synchronisation pour de tels systèmes polychrones et polytemporisés. Notre étude nous a mené à la conception de modèles sémantiques pour un langage temporisé à évènements discrets, appelé TESL et développé par Boulanger et al. Ce langage a été utilisé pour coordonner la simulation de modèles composites et pour tester l'intégration de systèmes.

Dans cette thèse, nous présentons une sémantique dénotationnelle fournissant une compréhension précise et logiquement cohérente du langage. Puis nous proposons une sémantique opérationnelle afin de dériver des traces d'exécutions satisfaisant une spécification TESL. Celui-ci a été utilisé pour les problématiques de test des systèmes, à travers l'implantation d'un solveur nommé Heron. Pour résoudre la question de cohérence et de correction de ces règles sémantiques, nous avons également développé une sémantique intermédiaire coinductive reliant les deux sémantiques dénotationnelles et opérationnelles. Nous établissons des propriétés sur la relation entre les deux sémantiques : correction, complétude, progrès ainsi que terminaison locale. Enfin, notre formalisation ainsi que les preuves associées ont été entièrement mécanisées dans l'assistant de preuve Isabelle/HOL.

**Mots clés** : Hétérogénéité, Synchronie, Concurrence, Coordination, Sémantique, Simulation, Test, Surveillance

# FORMALIZING TIME AND CAUSALITY IN POLYCHRONOUS POLYTIMED MODELS

Integrating components into systems turns out to be difficult when these components were designed according to different paradigms or when they rely on different time frames which must be synchronized. This synchronization may be event-driven (an event occurs because another event occurs) or time-driven (an event occurs because it is time for it to occur). Considering that each component admits its own time frame, and that they may not be related, a unique global time line may not exist.

We are interested in specifying synchronization patterns for such polychronous and polytimed systems. Our study had led us to design semantic models for a timed discrete-event language, called the TESL language developed by Boulanger et al. This language has been used for coordinating the simulation of composite models and testing system integration.

In this thesis, we present a denotational semantics providing an accurate and logic-consistent understanding of the language. Then we propose an operational semantics to derive satisfying runs from TESL specifications. It has been used for testing purposes, through the implementation of a solver, named Heron. To tackle the issue of the consistency and correctness of these semantic rules, we developed a co-inductive intermediate semantics that relates both the denotational and the operational semantics. Then we establish properties over the relation of our semantic models: soundness, completeness and progress, as well as local termination. Finally, our formalization and these proofs have been fully mechanized in the Isabelle/HOL proof assistant.

**Keywords**: Heterogeneity, Synchrony, Concurrency, Coordination, Semantics, Simulation, Testing, Monitoring

**Université Paris-Saclay**
Espace Technologique / Immeuble Discovery
Route de l'Orme des Merisiers RD 128 / 91190 Saint-Aubin, France