A Formal Development of a Polychronous Polytimed Coordination Language

Hai Nguyen Van   Frédéric Boulanger   Burkhart Wolff

May 20, 2019

Contents

Chapter 1

A Gentle Introduction to TESL

1.1 Context

The design of complex systems involves different formalisms for modeling their different parts or aspects. The global model of a system may therefore consist of a coordination of concurrent sub-models that use different paradigms such as differential equations, state machines, synchronous data-flow networks, discrete event models and so on, as illustrated in Figure 1.1. This raises the interest in architectural composition languages that allow for "bolting the respective sub-models together", along their various interfaces, and specifying the various ways of collaboration and coordination [2].

We are interested in languages that allow for specifying the 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 order to tackle the heterogeneous nature of the subsystems, we abstract their behavior as clocks. Each clock models an event, i.e., something that can occur or not at a given time. This time 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

Timed Finite State Machine

Lustre/SCADE

Ordinary Differential Equations

$$\frac{\partial Q}{\partial t} = \frac{\partial^2 s}{\partial t} + 2\frac{\partial r}{\partial x}$$
$$\frac{\partial Q}{\partial x} = \frac{\partial r}{\partial t \partial x} - \frac{\partial s}{\partial x}$$
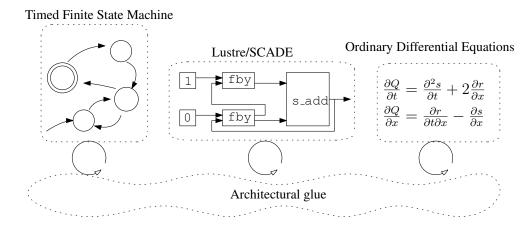
Architectural glue

Figure 1.1: A Heterogeneous Timed System Model

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. As a consequence, the key concept of our setting is the notion of a clock-indexed Kripke model: $\Sigma^\infty = \mathbb{N} \to \mathcal{K} \to (\mathbb{B} \times \mathcal{T})$, where $\mathcal{K}$ is an enumerable set of clocks, $\mathbb{B}$ is the set of booleans – used to indicate that a clock ticks at a given instant – and $\mathcal{T}$ is a universal metric time space for which we only assume that it is large enough to contain all individual time spaces of clocks and that it is ordered by some linear ordering ($\leq_\mathcal{T}$).

The elements of Σ^∞ are called runs. A specification language is a set of operators that constrains the set of possible monotonic runs. Specifications are composed by intersecting the denoted run sets of constraint operators. Consequently, such specification languages do not limit the number of clocks used to model a system (as long as it is finite) and it is always possible to add clocks to a specification. Moreover, they are *compositional* by construction since the composition of specifications consists of the conjunction of their constraints.

This work provides the following contributions:

- defining the non-trivial language TESL* in terms of clock-indexed Kripke models,

- proving that this denotational semantics is stuttering invariant,

- defining an adapted form of symbolic primitives and presenting the set of operational semantic rules,

- presenting formal proofs for soundness, completeness, and progress of the latter.

1.2   The TESL Language

The TESL language [1] was initially designed to coordinate the execution of heterogeneous components during the simulation of a system. We define here a minimal kernel of operators that

will form the basis of a family of specification languages, including the original TESL language, which is described at http://wdi.supelec.fr/software/TESL/.

1.2.1 Instantaneous Causal Operators

TESL has operators to deal with instantaneous causality, i.e., to react to an event occurrence in the very same observation instant.

- `c1 implies c2` means that at any instant where `c1` ticks, `c2` has to tick too.

- `c1 implies not c2` means that at any instant where `c1` ticks, `c2` cannot tick.

- `c1 kills c2` means that at any instant where `c1` ticks, and at any future instant, `c2` cannot tick.

1.2.2 Temporal Operators

TESL also has chronometric temporal operators that deal with dates and chronometric delays.

- `c sporadic t` means that clock `c` must have a tick at time `t` on its own time scale.

- `c1 sporadic t on c2` means that clock `c1` must have a tick at an instant where the time on `c2` is `t`.

- `c1 time delayed by d on m implies c2` means that every time clock `c1` ticks, `c2` must have a tick at the first instant where the time on `m` is `d` later than it was when `c1` had ticked. This means that every tick on `c1` is followed by a tick on `c2` after a delay `d` measured on the time scale of clock `m`.

- `time relation (c1, c2) in R` means that at every instant, the current time on clocks `c1` and `c2` must be in relation `R`. By default, the time lines of different clocks are independent. This operator allows us to link two time lines, for instance to model the fact that time in a GPS satellite and time in a GPS receiver on Earth are not the same but are related. Time being polymorphic in TESL, this can also be used to model the fact that the angular position on the camshaft of an engine moves twice as fast as the angular position on the crankshaft [1]. We may consider only linear arithmetic relations to restrict the problem to a domain where the resolution is decidable.

1.2.3 Asynchronous Operators

The last category of TESL operators allows the specification of asynchronous relations between event occurrences. They do not specify the precise instants at which ticks have to occur, they only put bounds on the set of instants at which they should occur.

- `c1 weakly precedes c2` means that for each tick on `c2`, there must be at least one tick on `c1` at a previous or at the same instant. This can also be expressed by stating that at each instant, the number of ticks since the beginning of the run must be lower or equal on `c2` than on `c1`.

---

[1]See http://wdi.supelec.fr/software/TESL/GalleryEngine for more details

- `c1 strictly precedes c2` means that for each tick on `c2`, there must be at least one tick on `c1` at a previous instant. This can also be expressed by saying that at each instant, the number of ticks on `c2` from the beginning of the run to this instant, must be lower or equal to the number of ticks on `c1` from the beginning of the run to the previous instant.

Chapter 2

The Core of the TESL Language: Syntax and Basics

theory TESL
imports Main

begin

2.1 Syntactic Representation

We define here the syntax of TESL specifications.

2.1.1 Basic elements of a specification

The following items appear in specifications:

- Clocks, which are identified by a name.

- Tag constants are just constants of a type which denotes the metric time space.

datatype   clock   = Clk ⟨string⟩
type_synonym instant_index = ⟨nat⟩

datatype   ′τ tag_const = TConst   (the_tag_const : ′τ)   ("$τ_{cst}$")

2.1.2 Operators for the TESL language

The type of atomic TESL constraints, which can be combined to form specifications.

```
datatype ′τ TESL_atomic =
    SporadicOn       ⟨clock⟩ ⟨′τ tag_const⟩ ⟨clock⟩  ("_ sporadic _ on _" 55)
  | TagRelation      ⟨clock⟩ ⟨clock⟩ ⟨(′τ tag_const × ′τ tag_const) ⇒ bool⟩
                                              ("time-relation ⌊_, _⌋ ∈ _" 55)
  | Implies          ⟨clock⟩ ⟨clock⟩                (infixr "implies" 55)
  | ImpliesNot       ⟨clock⟩ ⟨clock⟩                (infixr "implies not" 55)
  | TimeDelayedBy    ⟨clock⟩ ⟨′τ tag_const⟩ ⟨clock⟩ ⟨clock⟩
                                              ("_ time-delayed by _ on _ implies _" 55)
```

```
| WeaklyPrecedes   ⟨clock⟩ ⟨clock⟩              (infixr "weakly precedes" 55)
| StrictlyPrecedes ⟨clock⟩ ⟨clock⟩             (infixr "strictly precedes" 55)
| Kills            ⟨clock⟩ ⟨clock⟩             (infixr "kills" 55)
```

A TESL formula is just a list of atomic constraints, with implicit conjunction for the semantics.

type_synonym 'τ TESL_formula = ⟨'τ TESL_atomic list⟩

We call *positive atoms* the atomic constraints that create ticks from nothing. Only sporadic constraints are positive in the current version of TESL.

fun positive_atom :: ⟨'τ TESL_atomic ⇒ bool⟩ **where**
    ⟨positive_atom (_ sporadic _ on _) = True⟩
  | ⟨positive_atom _   = False⟩

The `NoSporadic` function removes sporadic constraints from a TESL formula.

abbreviation NoSporadic :: ⟨'τ TESL_formula ⇒ 'τ TESL_formula⟩
where
  ⟨NoSporadic f ≡ (List.filter (λf_{atom}. case f_{atom} of
      _ sporadic _ on _ ⇒ False
    | _ ⇒ True) f)⟩

2.1.3   Field Structure of the Metric Time Space

In order to handle tag relations and delays, tags must belong to a field. We show here that this is the case when the type parameter of 'τ `tag_const` is itself a field.

instantiation tag_const ::(field)field
begin
  fun inverse_tag_const
  where ⟨inverse ($τ_{cst}$ t) = $τ_{cst}$ (inverse t)⟩

  fun divide_tag_const
    where ⟨divide ($τ_{cst}$ t_1) ($τ_{cst}$ t_2) = $τ_{cst}$ (divide t_1 t_2)⟩

  fun uminus_tag_const
    where ⟨uminus ($τ_{cst}$ t) = $τ_{cst}$ (uminus t)⟩

fun minus_tag_const
  where ⟨minus ($τ_{cst}$ t_1) ($τ_{cst}$ t_2) = $τ_{cst}$ (minus t_1 t_2)⟩

definition ⟨one_tag_const ≡ $τ_{cst}$ 1⟩

fun times_tag_const
  where ⟨times ($τ_{cst}$ t_1) ($τ_{cst}$ t_2) = $τ_{cst}$ (times t_1 t_2)⟩

definition ⟨zero_tag_const ≡ $τ_{cst}$ 0⟩

fun plus_tag_const
  where ⟨plus ($τ_{cst}$ t_1) ($τ_{cst}$ t_2) = $τ_{cst}$ (plus t_1 t_2)⟩

instance ⟨*proof*⟩

end

For comparing dates (which are represented by tags) on clocks, we need an order on tags.

instantiation tag_const :: (order)order
begin

```
    inductive less_eq_tag_const :: ⟨'a tag_const ⇒ 'a tag_const ⇒ bool⟩
    where
      Int_less_eq[simp]:        ⟨n ≤ m ⟹ (TConst n) ≤ (TConst m)⟩

    definition less_tag: ⟨(x::'a tag_const) < y ⟷ (x ≤ y) ∧ (x ≠ y)⟩

    instance ⟨proof⟩

end
```

For ensuring that time does never flow backwards, we need a total order on tags.

```
instantiation tag_const :: (linorder)linorder
begin
  instance ⟨proof⟩

end

end
```

2.2 Defining Runs

```
theory Run
imports TESL

  begin
```

Runs are sequences of instants, and each instant maps a clock to a pair (h, t) where h indicates whether the clock ticks or not, and t is the current time on this clock. The first element of the pair is called the *hamlet* of the clock (to tick or not to tick), the second element is called the *time*.

```
abbreviation hamlet where ⟨hamlet ≡ fst⟩
abbreviation time   where ⟨time ≡ snd⟩

type_synonym 'τ instant = ⟨clock ⇒ (bool × 'τ tag_const)⟩
```

Runs have the additional constraint that time cannot go backwards on any clock in the sequence of instants. Therefore, for any clock, the time projection of a run is monotonous.

```
typedef (overloaded) 'τ::linordered_field run =
  ⟨{ ϱ::nat ⇒ 'τ instant. ∀c. mono (λn. time (ϱ n c)) }⟩
⟨proof⟩

lemma Abs_run_inverse_rewrite:
  ⟨∀c. mono (λn. time (ϱ n c)) ⟹ Rep_run (Abs_run ϱ) = ϱ⟩
⟨proof⟩
```

A *dense* run is a run in which something happens (at least one clock ticks) at every instant.

```
definition ⟨dense_run ϱ ≡ (∀n. ∃c. hamlet ((Rep_run ϱ) n c))⟩
```

run_tick_count ϱ K n counts the number of ticks on clock K in the interval [0, n] of run ϱ.

```
fun run_tick_count :: ⟨('τ::linordered_field) run ⇒ clock ⇒ nat ⇒ nat⟩
  ("#≤ _ _ _")
where
  ⟨(#≤ ϱ K 0)        = (if hamlet ((Rep_run ϱ) 0 K)
                          then 1
```

```
                        else 0)⟩
| ⟨(#≤ ϱ K (Suc n)) = (if hamlet ((Rep_run ϱ) (Suc n) K)
                        then 1 + (#≤ ϱ K n)
                        else (#≤ ϱ K n))⟩
```

run_tick_count_strictly ϱ K n counts the number of ticks on clock K in the interval [0, n[
of run ϱ.

fun run_tick_count_strictly :: ⟨('τ::linordered_field) run ⇒ clock ⇒ nat ⇒ nat⟩
  ("#< _ _ _")
where
  ⟨(#< ϱ K 0)   = 0⟩
| ⟨(#< ϱ K (Suc n)) = #≤ ϱ K n⟩

first_time ϱ K n τ tells whether instant n in run ϱ is the first one where the time on clock K
reaches τ.

definition first_time :: ⟨'a::linordered_field run ⇒ clock ⇒ nat ⇒ 'a tag_const
                            ⇒ bool⟩
where
  ⟨first_time ϱ K n τ ≡ (time ((Rep_run ϱ) n K) = τ)
                        ∧ (∄n'. n' < n ∧ time ((Rep_run ϱ) n' K) = τ)⟩

The time on a clock is necessarily less than τ before the first instant at which it reaches τ.

lemma before_first_time:
  assumes ⟨first_time ϱ K n τ⟩
      and ⟨m < n⟩
    shows ⟨time ((Rep_run ϱ) m K) < τ⟩
⟨*proof*⟩

This leads to an alternate definition of first_time:

lemma alt_first_time_def:
  assumes ⟨∀m < n. time ((Rep_run ϱ) m K) < τ⟩
      and ⟨time ((Rep_run ϱ) n K) = τ⟩
    shows ⟨first_time ϱ K n τ⟩
⟨*proof*⟩

end

Chapter 3

Denotational Semantics

theory Denotational
imports
    TESL
    Run

begin

The denotational semantics maps TESL formulae to sets of satisfying runs. Firstly, we define the semantics of atomic formulae (basic constructs of the TESL language), then we define the semantics of compound formulae as the intersection of the semantics of their components: a run must satisfy all the individual formulae of a compound formula.

3.1 Denotational interpretation for atomic TESL formulae

fun TESL_interpretation_atomic
    :: ⟨('τ::linordered_field) TESL_atomic ⇒ 'τ run set⟩ ("⟦ _ ⟧$_{TESL}$")
where
  — K_1 sporadic τ on K_2 means that K_1 should tick at an instant where the time on K_2 is τ.

  ⟨⟦ K_1 sporadic τ on K_2 ⟧$_{TESL}$ =
      {ϱ. ∃n::nat. hamlet ((Rep_run ϱ) n K_1) ∧ time ((Rep_run ϱ) n K_2) = τ}⟩
  — time-relation ⌊K_1, K_2⌋ ∈ R means that at each instant, the time on K_1 and the time on K_2 are in relation R.

  | ⟨⟦ time-relation ⌊K_1, K_2⌋ ∈ R ⟧$_{TESL}$ =
      {ϱ. ∀n::nat. R (time ((Rep_run ϱ) n K_1), time ((Rep_run ϱ) n K_2))}⟩
  — master implies slave means that at each instant at which master ticks, slave also ticks.

  | ⟨⟦ master implies slave ⟧$_{TESL}$ =
      {ϱ. ∀n::nat. hamlet ((Rep_run ϱ) n master) ⟶ hamlet ((Rep_run ϱ) n slave)}⟩
  — master implies not slave means that at each instant at which master ticks, slave does not tick.

  | ⟨⟦ master implies not slave ⟧$_{TESL}$ =
      {ϱ. ∀n::nat. hamlet ((Rep_run ϱ) n master) ⟶ ¬hamlet ((Rep_run ϱ) n slave)}⟩
  — master time-delayed by $\delta\tau$ on measuring implies slave means that at each instant at which master ticks, slave will tick after a delay $\delta\tau$ measured on the time scale of measuring.

  | ⟨⟦ master time-delayed by $\delta\tau$ on measuring implies slave ⟧$_{TESL}$ =
    — When master ticks, let's call t_0 the current date on measuring. Then, at the first instant when the date on measuring is t_0 + δt, slave has to tick.

      {ϱ. ∀n. hamlet ((Rep_run ϱ) n master) ⟶
              (let measured_time = time ((Rep_run ϱ) n measuring) in
                ∀m ≥ n. first_time ϱ measuring m (measured_time + $\delta\tau$)

```
                        ⟶ hamlet ((Rep_run ϱ) m slave)
            )
        }⟩
```

— K_1 `weakly precedes` K_2 means that each tick on K_2 must be preceded by or coincide with at least one tick on K_1. Therefore, at each instant n, the number of ticks on K_2 must be less or equal to the number of ticks on K_1.

| ⟨⟦ K₁ `weakly precedes` K₂ ⟧$_{TESL}$ =
    {ϱ. ∀n::nat. (run_tick_count ϱ K₂ n) ≤ (run_tick_count ϱ K₁ n)}⟩

— K_1 `strictly precedes` K_2 means that each tick on K_2 must be preceded by at least one tick on K_1 at a previous instant. Therefore, at each instant n, the number of ticks on K_2 must be less or equal to the number of ticks on K_1 at instant n - 1.

| ⟨⟦ K₁ `strictly precedes` K₂ ⟧$_{TESL}$ =
    {ϱ. ∀n::nat. (run_tick_count ϱ K₂ n) ≤ (run_tick_count_strictly ϱ K₁ n)}⟩

— K_1 `kills` K_2 means that when K_1 ticks, K_2 cannot tick and is not allowed to tick at any further instant.

| ⟨⟦ K₁ `kills` K₂ ⟧$_{TESL}$ =
    {ϱ. ∀n::nat. hamlet ((Rep_run ϱ) n K₁)
                ⟶ (∀m≥n. ¬ hamlet ((Rep_run ϱ) m K₂))}⟩

3.2   Denotational interpretation for TESL formulae

To satisfy a formula, a run has to satisfy the conjunction of its atomic formulae. Therefore, the interpretation of a formula is the intersection of the interpretations of its components.

fun `TESL_interpretation` :: ⟨('τ::linordered_field) TESL_formula ⇒ 'τ run set⟩
  ("⟦[_]⟧$_{TESL}$")
where
  ⟨⟦[[]]⟧$_{TESL}$ = {_. True}⟩
| ⟨⟦[φ # Φ]⟧$_{TESL}$ = ⟦ φ ⟧$_{TESL}$ ∩ ⟦[Φ]⟧$_{TESL}$⟩

lemma `TESL_interpretation_homo`:
  ⟨⟦ φ ⟧$_{TESL}$ ∩ ⟦[Φ]⟧$_{TESL}$ = ⟦[φ # Φ]⟧$_{TESL}$⟩
⟨*proof*⟩

3.2.1   Image interpretation lemma

theorem `TESL_interpretation_image`:
  ⟨⟦[Φ]⟧$_{TESL}$ = ⋂ (($\lambda\varphi$. ⟦ φ ⟧$_{TESL}$) ' set Φ)⟩
⟨*proof*⟩

3.2.2   Expansion law

Similar to the expansion laws of lattices.

theorem `TESL_interp_homo_append`:
  ⟨⟦[Φ_1 @ Φ_2]⟧$_{TESL}$ = ⟦[Φ_1]⟧$_{TESL}$ ∩ ⟦[Φ_2]⟧$_{TESL}$⟩
⟨*proof*⟩

3.3   Equational laws for the denotation of TESL formulae

lemma `TESL_interp_assoc`:
  ⟨⟦[(Φ_1 @ Φ_2) @ Φ_3]⟧$_{TESL}$ = ⟦[Φ_1 @ (Φ_2 @ Φ_3)]⟧$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_commute`:
  shows ⟨⟦[Φ_1 @ Φ_2]⟧$_{TESL}$ = ⟦[Φ_2 @ Φ_1]⟧$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_left_commute`:
  ⟨[[Φ_1 @ (Φ_2 @ Φ_3)]]$_{TESL}$ = [[Φ_2 @ (Φ_1 @ Φ_3)]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_idem`:
  ⟨[[Φ @ Φ]]$_{TESL}$ = [[Φ]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_left_idem`:
  ⟨[[Φ_1 @ (Φ_1 @ Φ_2)]]$_{TESL}$ = [[Φ_1 @ Φ_2]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_right_idem`:
  ⟨[[(Φ_1 @ Φ_2) @ Φ_2]]$_{TESL}$ = [[Φ_1 @ Φ_2]]$_{TESL}$⟩
⟨*proof*⟩

lemmas `TESL_interp_aci = TESL_interp_commute`
                        `TESL_interp_assoc`
                        `TESL_interp_left_commute`
                        `TESL_interp_left_idem`

The empty formula is the identity element.

lemma `TESL_interp_neutral1`:
  ⟨[[[] @ Φ]]$_{TESL}$ = [[Φ]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_neutral2`:
  ⟨[[Φ @ []]]$_{TESL}$ = [[Φ]]$_{TESL}$⟩
⟨*proof*⟩

3.4 Decreasing interpretation of TESL formulae

Adding constraints to a TESL formula reduces the number of satisfying runs.

lemma `TESL_sem_decreases_head`:
  ⟨[[Φ]]$_{TESL}$ $\supseteq$ [[φ # Φ]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_sem_decreases_tail`:
  ⟨[[Φ]]$_{TESL}$ $\supseteq$ [[Φ @ [φ]]]$_{TESL}$⟩
⟨*proof*⟩

Repeating a formula in a specification does not change the specification.

lemma `TESL_interp_formula_stuttering`:
  assumes ⟨φ ∈ set Φ⟩
    shows ⟨[[φ # Φ]]$_{TESL}$ = [[Φ]]$_{TESL}$⟩
⟨*proof*⟩

Removing duplicate formulae in a specification does not change the specification.

lemma `TESL_interp_remdups_absorb`:
  ⟨[[Φ]]$_{TESL}$ = [[`remdups` Φ]]$_{TESL}$⟩
⟨*proof*⟩

Specifications that contain the same formulae have the same semantics.

lemma `TESL_interp_set_lifting`:
  assumes ⟨set Φ = set Φ'⟩

      shows ⟨[[Φ]]$_{TESL}$ = [[Φ']]$_{TESL}$⟩
⟨*proof*⟩

The semantics of specifications is contravariant with respect to their inclusion.

theorem `TESL_interp_decreases_setinc`:
  assumes ⟨set Φ ⊆ set Φ'⟩
    shows ⟨[[Φ]]$_{TESL}$ ⊇ [[Φ']]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_decreases_add_head`:
  assumes ⟨set Φ ⊆ set Φ'⟩
    shows ⟨[[φ # Φ]]$_{TESL}$ ⊇ [[φ # Φ']]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_decreases_add_tail`:
  assumes ⟨set Φ ⊆ set Φ'⟩
    shows ⟨[[Φ @ [φ]]]$_{TESL}$ ⊇ [[Φ' @ [φ]]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_absorb1`:
  assumes ⟨set Φ$_1$ ⊆ set Φ$_2$⟩
    shows ⟨[[Φ$_1$ @ Φ$_2$]]$_{TESL}$ = [[Φ$_2$]]$_{TESL}$⟩
⟨*proof*⟩

lemma `TESL_interp_absorb2`:
  assumes ⟨set Φ$_2$ ⊆ set Φ$_1$⟩
    shows ⟨[[Φ$_1$ @ Φ$_2$]]$_{TESL}$ = [[Φ$_1$]]$_{TESL}$⟩
⟨*proof*⟩

3.5   Some special cases

lemma `NoSporadic_stable` [simp]:
  ⟨[[Φ]]$_{TESL}$ ⊆ [[NoSporadic Φ]]$_{TESL}$⟩
⟨*proof*⟩

lemma `NoSporadic_idem` [simp]:
  ⟨[[Φ]]$_{TESL}$ ∩ [[NoSporadic Φ]]$_{TESL}$ = [[Φ]]$_{TESL}$⟩
⟨*proof*⟩

lemma `NoSporadic_setinc`:
  ⟨set (NoSporadic Φ) ⊆ set Φ⟩
⟨*proof*⟩
end

Chapter 4

Symbolic Primitives for Building Runs

theory `SymbolicPrimitive`
  imports `Run`

begin

We define here the primitive constraints on runs, towards which we translate TESL specifications in the operational semantics. These constraints refer to a specific symbolic run and can therefore access properties of the run at particular instants (for instance, the fact that a clock ticks at instant `n` of the run, or the time on a given clock at that instant).

In the previous chapters, we had no reference to particular instants of a run because the TESL language should be invariant by stuttering in order to allow the composition of specifications: adding an instant where no clock ticks to a run that satisfies a formula should yield another run that satisfies the same formula. However, when constructing runs that satisfy a formula, we need to be able to refer to the time or hamlet of a clock at a given instant.

Counter expressions are used to get the number of ticks of a clock up to (strictly or not) a given instant index.

datatype `cnt_expr =`
  `TickCountLess ⟨clock⟩ ⟨instant_index⟩ ("#`$^<$`")`
`| TickCountLeq ⟨clock⟩ ⟨instant_index⟩  ("#`$^\leq$`")`

4.0.1 Symbolic Primitives for Runs

Tag values are used to refer to the time on a clock at a given instant index.

datatype `tag_val =`
  `TSchematic ⟨clock * instant_index⟩ ("`τ_{var}`")`

datatype `'τ constr =`
— `c ⇓ n @ τ` constrains clock `c` to have time τ at instant `n` of the run.

  `Timestamp    ⟨clock⟩   ⟨instant_index⟩ ⟨'τ tag_const⟩       ("_ ⇓ _ @ _")`
— `m @ n ⊕ δt ⇒ s` constrains clock `s` to tick at the first instant at which the time on `m` has increased by δt
  from the value it had at instant `n` of the run.
`| TimeDelay    ⟨clock⟩   ⟨instant_index⟩ ⟨'τ tag_const⟩ ⟨clock⟩ ("_ @ _ ⊕ _ ⇒ _")`
— `c ⇑ n` constrains clock `c` to tick at instant `n` of the run.

```
| Ticks          ⟨clock⟩   ⟨instant_index⟩                    ("_ ⇑ _")
— c ¬⇑ n constrains clock c not to tick at instant n of the run.

| NotTicks       ⟨clock⟩   ⟨instant_index⟩                    ("_ ¬⇑ _")
— c ¬⇑ < n constrains clock c not to tick before instant n of the run.

| NotTicksUntil ⟨clock⟩    ⟨instant_index⟩                    ("_ ¬⇑ < _")
— c ¬⇑ ≥ n constrains clock c not to tick at and after instant n of the run.

| NotTicksFrom  ⟨clock⟩    ⟨instant_index⟩                    ("_ ¬⇑ ≥ _")
— ⌊τ₁, τ₂⌋ ∈ R constrains tag variables τ₁ and τ₂ to be in relation R.

| TagArith      ⟨tag_val⟩ ⟨tag_val⟩ ⟨('τ tag_const × 'τ tag_const) ⇒ bool⟩ ("⌊_, _⌋ ∈ _")
— ⌈k₁, k₂⌉ ∈ R constrains counter expressions k₁ and k₂ to be in relation R.

| TickCntArith  ⟨cnt_expr⟩ ⟨cnt_expr⟩ ⟨(nat × nat) ⇒ bool⟩    ("⌈_, _⌉ ∈ _")
— k₁ ⪯ k₂ constrains counter expression k₁ to be less or equal to counter expression k₂.

| TickCntLeq    ⟨cnt_expr⟩ ⟨cnt_expr⟩                         ("_ ⪯ _")
```

type_synonym '𝜏 system = ⟨'𝜏 constr list⟩

The abstract machine has configurations composed of:

- the past Γ, which captures choices that have already be made as a list of symbolic primitive constraints on the run;

- the current index n, which is the index of the present instant;

- the present Ψ, which captures the formulae that must be satisfied in the current instant;

- the future Φ, which captures the constraints on the future of the run.

type_synonym '𝜏 config =
                ⟨'𝜏 system * instant_index * '𝜏 TESL_formula * '𝜏 TESL_formula⟩

4.1   Semantics of Primitive Constraints

The semantics of the primitive constraints is defined in a way similar to the semantics of TESL formulae.

```
fun counter_expr_eval :: ⟨('τ::linordered_field) run ⇒ cnt_expr ⇒ nat⟩
  ("⟦ _ ⊢ _ ⟧_cntexpr")
where
  ⟨⟦ ϱ ⊢ #< clk indx ⟧_cntexpr = run_tick_count_strictly ϱ clk indx⟩
| ⟨⟦ ϱ ⊢ #≤ clk indx ⟧_cntexpr = run_tick_count ϱ clk indx⟩


fun symbolic_run_interpretation_primitive
  ::⟨('τ::linordered_field) constr ⇒ 'τ run set⟩ ("⟦ _ ⟧_prim")
where
  ⟨⟦ K ⇑ n  ⟧_prim       = {ϱ. hamlet ((Rep_run ϱ) n K) }⟩
| ⟨⟦ K @ n₀ ⊕ δt ⇒ K' ⟧_prim =
                {ϱ. ∀n ≥ n₀. first_time ϱ K n (time ((Rep_run ϱ) n₀ K) + δt)
                        ⟶ hamlet ((Rep_run ϱ) n K')}⟩
| ⟨⟦ K ¬⇑ n ⟧_prim       = {ϱ. ¬hamlet ((Rep_run ϱ) n K) }⟩
| ⟨⟦ K ¬⇑ < n ⟧_prim     = {ϱ. ∀i < n. ¬ hamlet ((Rep_run ϱ) i K)}⟩
| ⟨⟦ K ¬⇑ ≥ n ⟧_prim     = {ϱ. ∀i ≥ n. ¬ hamlet ((Rep_run ϱ) i K) }⟩
| ⟨⟦ K ⇓ n @ τ ⟧_prim = {ϱ. time ((Rep_run ϱ) n K) = τ }⟩
| ⟨⟦ ⌊τ_var(K₁, n₁), τ_var(K₂, n₂)⌋ ∈ R ⟧_prim =
    { ϱ. R (time ((Rep_run ϱ) n₁ K₁), time ((Rep_run ϱ) n₂ K₂)) }⟩
```

| ⟨⟦ ⌈e$_1$, e$_2$⌉ ∈ R ⟧$_{prim}$ = { ϱ. R (⟦ ϱ ⊢ e$_1$ ⟧$_{cntexpr}$, ⟦ ϱ ⊢ e$_2$ ⟧$_{cntexpr}$) }⟩
| ⟨⟦ cnt_e$_1$ ⪯ cnt_e$_2$ ⟧$_{prim}$ = { ϱ. ⟦ ϱ ⊢ cnt_e$_1$ ⟧$_{cntexpr}$ ≤ ⟦ ϱ ⊢ cnt_e$_2$ ⟧$_{cntexpr}$ }⟩

The composition of primitive constraints is their conjunction, and we get the set of satisfying runs by intersection.

fun `symbolic_run_interpretation`
  ::⟨('τ::linordered_field) constr list ⇒ ('τ::linordered_field) run set⟩
  ("⟦⟦ _ ⟧⟧$_{prim}$")
where
  ⟨⟦⟦ [] ⟧⟧$_{prim}$ = {ϱ. True }⟩
| ⟨⟦⟦ γ # Γ ⟧⟧$_{prim}$ = ⟦ γ ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$⟩

lemma `symbolic_run_interp_cons_morph`:
  ⟨⟦ γ ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$ = ⟦⟦ γ # Γ ⟧⟧$_{prim}$⟩
⟨*proof*⟩

definition `consistent_context` :: ⟨('τ::linordered_field) constr list ⇒ bool⟩
where
  ⟨consistent_context Γ ≡ (⟦⟦ Γ ⟧⟧$_{prim}$ ≠ {}) ⟩

4.1.1 Defining a method for witness construction

In order to build a run, we can start from an initial run in which no clock ticks and the time is always 0 on any clock.

abbreviation `initial_run` :: ⟨('τ::linordered_field) run⟩ ("ϱ$_⊙$") **where**
  ⟨ϱ$_⊙$ ≡ Abs_run ((λ_ _. (False, τ_{cst} 0)) ::nat ⇒ clock ⇒ (bool × 'τ tag_const))⟩

To help avoiding that time flows backward, setting the time on a clock at a given instant sets it for the future instants too.

fun `time_update`
  :: ⟨nat ⇒ clock ⇒ ('τ::linordered_field) tag_const ⇒ (nat ⇒ 'τ instant)
      ⇒ (nat ⇒ 'τ instant)⟩
where
  ⟨time_update n K τ ϱ = (λn' K'. if K = K' ∧ n ≤ n'
                                 then (hamlet (ϱ n K), τ)
                                 else ϱ n' K')⟩

4.2 Rules and properties of consistence

lemma `context_consistency_preservationI`:
  ⟨consistent_context ((γ::('τ::linordered_field) constr)#Γ) ⟹ consistent_context Γ⟩
⟨*proof*⟩
inductive `context_independency`
  ::⟨('τ::linordered_field) constr ⇒ 'τ constr list ⇒ bool⟩ ("_ ⋈ _")
where
  `NotTicks_independency`:
  ⟨(K ⇑ n) ∉ set Γ ⟹ (K ¬⇑ n) ⋈ Γ⟩
| `Ticks_independency`:
  ⟨(K ¬⇑ n) ∉ set Γ ⟹ (K ⇑ n) ⋈ Γ⟩
| `Timestamp_independency`:
  ⟨(∄τ'. τ' = τ ∧ (K ⇓ n @ τ) ∈ set Γ) ⟹ (K ⇓ n @ τ) ⋈ Γ⟩

4.3 Major Theorems

4.3.1 Interpretation of a context

The interpretation of a context is the intersection of the interpretation of its components.

theorem `symrun_interp_fixpoint`:
⟨$\bigcap$ ($(\lambda\gamma.\ [\![\ \gamma\ ]\!]_{prim})$ ' set Γ) = $[\![\ \Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

4.3.2 Expansion law

Similar to the expansion laws of lattices

theorem `symrun_interp_expansion`:
⟨$[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim}$ = $[\![\ \Gamma_1\ ]\!]_{prim}\ \cap\ [\![\ \Gamma_2\ ]\!]_{prim}$⟩
⟨*proof*⟩

4.4 Equations for the interpretation of symbolic primitives

4.4.1 General laws

lemma `symrun_interp_assoc`:
⟨$[\![\ (\Gamma_1$ @ $\Gamma_2)$ @ $\Gamma_3\ ]\!]_{prim}$ = $[\![\ \Gamma_1$ @ $(\Gamma_2$ @ $\Gamma_3)\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_commute`:
⟨$[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim}$ = $[\![\ \Gamma_2$ @ $\Gamma_1\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_left_commute`:
⟨$[\![\ \Gamma_1$ @ $(\Gamma_2$ @ $\Gamma_3)\ ]\!]_{prim}$ = $[\![\ \Gamma_2$ @ $(\Gamma_1$ @ $\Gamma_3)\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_idem`:
⟨$[\![\ \Gamma$ @ $\Gamma\ ]\!]_{prim}$ = $[\![\ \Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_left_idem`:
⟨$[\![\ \Gamma_1$ @ $(\Gamma_1$ @ $\Gamma_2)\ ]\!]_{prim}$ = $[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_right_idem`:
⟨$[\![\ (\Gamma_1$ @ $\Gamma_2)$ @ $\Gamma_2\ ]\!]_{prim}$ = $[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemmas `symrun_interp_aci` = `symrun_interp_commute`
                                  `symrun_interp_assoc`
                                  `symrun_interp_left_commute`
                                  `symrun_interp_left_idem`

— Identity element

lemma `symrun_interp_neutral1`:
⟨$[\![\ []$ @ $\Gamma\ ]\!]_{prim}$ = $[\![\ \Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma `symrun_interp_neutral2`:
⟨$[\![\ \Gamma$ @ $[]\ ]\!]_{prim}$ = $[\![\ \Gamma\ ]\!]_{prim}$⟩

⟨*proof*⟩

4.4.2 Decreasing interpretation of symbolic primitives

Adding constraints to a context reduces the number of satisfying runs.

lemma TESL_sem_decreases_head:
⟨$[\![\ \Gamma\ ]\!]_{prim} \supseteq [\![\ \gamma$ # $\Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma TESL_sem_decreases_tail:
⟨$[\![\ \Gamma\ ]\!]_{prim} \supseteq [\![\ \Gamma$ @ $[\gamma]\ ]\!]_{prim}$⟩
⟨*proof*⟩

Adding a constraint that is already in the context does not change the interpretation of the context.

lemma symrun_interp_formula_stuttering:
assumes ⟨$\gamma \in$ set Γ⟩
shows ⟨$[\![\ \gamma$ # $\Gamma\ ]\!]_{prim} = [\![\ \Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

Removing duplicate constraints from a context does not change the interpretation of the context.

lemma symrun_interp_remdups_absorb:
⟨$[\![\ \Gamma\ ]\!]_{prim} = [\![$ remdups $\Gamma\ ]\!]_{prim}$⟩
⟨*proof*⟩

Two identical sets of constraints have the same interpretation, the order in the context does not matter.

lemma symrun_interp_set_lifting:
assumes ⟨set Γ = set Γ'⟩
shows ⟨$[\![\ \Gamma\ ]\!]_{prim} = [\![\ \Gamma'\ ]\!]_{prim}$⟩
⟨*proof*⟩

The interpretation of contexts is contravariant with regard to set inclusion.

theorem symrun_interp_decreases_setinc:
assumes ⟨set $\Gamma \subseteq$ set Γ'⟩
shows ⟨$[\![\ \Gamma\ ]\!]_{prim} \supseteq [\![\ \Gamma'\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma symrun_interp_decreases_add_head:
assumes ⟨set $\Gamma \subseteq$ set Γ'⟩
shows ⟨$[\![\ \gamma$ # $\Gamma\ ]\!]_{prim} \supseteq [\![\ \gamma$ # $\Gamma'\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma symrun_interp_decreases_add_tail:
assumes ⟨set $\Gamma \subseteq$ set Γ'⟩
shows ⟨$[\![\ \Gamma$ @ $[\gamma]\ ]\!]_{prim} \supseteq [\![\ \Gamma'$ @ $[\gamma]\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma symrun_interp_absorb1:
assumes ⟨set $\Gamma_1 \subseteq$ set Γ_2⟩
shows ⟨$[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim} = [\![\ \Gamma_2\ ]\!]_{prim}$⟩
⟨*proof*⟩

lemma symrun_interp_absorb2:
assumes ⟨set $\Gamma_2 \subseteq$ set Γ_1⟩

    shows $\langle [\![\ \Gamma_1\ @\ \Gamma_2\ ]\!]_{prim}\ =\ [\![\ \Gamma_1\ ]\!]_{prim}\rangle$
$\langle proof \rangle$

end

Chapter 5

Operational Semantics

```
theory Operational
imports
  SymbolicPrimitive

begin
```

The operational semantics defines rules to build symbolic runs from a TESL specification (a set of TESL formulae). Symbolic runs are described using the symbolic primitives presented in the previous chapter. Therefore, the operational semantics compiles a set of constraints on runs, as defined by the denotational semantics, into a set of symbolic constraints on the instants of the runs. Concrete runs can then be obtained by solving the constraints at each instant.

5.1 Operational steps

We introduce a notation to describe configurations:

- Γ is the context, the set of symbolic constraints on past instants of the run;

- n is the index of the current instant, the present;

- Ψ is the TESL formula that must be satisfied at the current instant (present);

- Φ is the TESL formula that must be satisfied for the following instants (the future).

```
abbreviation uncurry_conf
  ::⟨('τ::linordered_field) system ⇒ instant_index ⇒ 'τ TESL_formula ⇒ 'τ TESL_formula
      ⇒ 'τ config⟩                                        ("_, _ ⊢ _ ▷ _" 80)
where
  ⟨Γ, n ⊢ Ψ ▷ Φ ≡ (Γ, n, Ψ, Φ)⟩
```

The only introduction rule allows us to progress to the next instant when there are no more constraints to satisfy for the present instant.

```
inductive operational_semantics_intro
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩           ("_ ↪ᵢ _" 70)
where
  instant_i:
```

⟨(Γ, n ⊢ [] ▷ Φ) ↪$_i$ (Γ, Suc n ⊢ Φ ▷ [])⟩

The elimination rules describe how TESL formulae for the present are transformed into constraints on the past and on the future.

inductive operational_semantics_elim
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪$_e$ _" 70)
where
  sporadic_on_e1:
— A sporadic constraint can be ignored in the present and rejected into the future.

  ⟨(Γ, n ⊢ ((K$_1$ sporadic τ on K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (Γ, n ⊢ Ψ ▷ ((K$_1$ sporadic τ on K$_2$) # Φ))⟩
| sporadic_on_e2:
— It can also be handled in the present by making the clock tick and have the expected time. Once it has been
  handled, it is no longer a constraint to satisfy, so it disappears from the future.

  ⟨(Γ, n ⊢ ((K$_1$ sporadic τ on K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ⇑ n) # (K$_2$ ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)⟩
| tagrel_e:
— A relation between time scales has to be obeyed at every instant.

  ⟨(Γ, n ⊢ ((time-relation ⌊K$_1$, K$_2$⌋ ∈ R) # Ψ) ▷ Φ)
     ↪$_e$ (((⌊τ$_{var}$(K$_1$, n), τ$_{var}$(K$_2$, n)⌋ ∈ R) # Γ), n
              ⊢ Ψ ▷ ((time-relation ⌊K$_1$, K$_2$⌋ ∈ R) # Φ))⟩
| implies_e1:
— An implication can be handled in the present by forbidding a tick of the master clock. The implication is
  copied back into the future because it holds for the whole run.

  ⟨(Γ, n ⊢ ((K$_1$ implies K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K$_1$ implies K$_2$) # Φ))⟩
| implies_e2:
— It can also be handled in the present by making both the master and the slave clocks tick.

  ⟨(Γ, n ⊢ ((K$_1$ implies K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ⇑ n) # (K$_2$ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K$_1$ implies K$_2$) # Φ))⟩
| implies_not_e1:
— A negative implication can be handled in the present by forbidding a tick of the master clock. The implication
  is copied back into the future because it holds for the whole run.

  ⟨(Γ, n ⊢ ((K$_1$ implies not K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K$_1$ implies not K$_2$) # Φ))⟩
| implies_not_e2:
— It can also be handled in the present by making the master clock ticks and forbidding a tick on the slave
  clock.
  ⟨(Γ, n ⊢ ((K$_1$ implies not K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ⇑ n) # (K$_2$ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K$_1$ implies not K$_2$) # Φ))⟩
| timedelayed_e1:
— A timed delayed implication can be handled by forbidding a tick on the master clock.

  ⟨(Γ, n ⊢ ((K$_1$ time-delayed by δτ on K$_2$ implies K$_3$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K$_1$ time-delayed by δτ on K$_2$ implies K$_3$) # Φ))⟩
| timedelayed_e2:
— It can also be handled by making the master clock tick and adding a constraint that makes the slave clock
  tick when the delay has elapsed on the measuring clock.
  ⟨(Γ, n ⊢ ((K$_1$ time-delayed by δτ on K$_2$ implies K$_3$) # Ψ) ▷ Φ)
     ↪$_e$ (((K$_1$ ⇑ n) # (K$_2$ @ n ⊕ δτ ⇒ K$_3$) # Γ), n
              ⊢ Ψ ▷ ((K$_1$ time-delayed by δτ on K$_2$ implies K$_3$) # Φ))⟩
| weakly_precedes_e:
— A weak precedence relation has to hold at every instant.

  ⟨(Γ, n ⊢ ((K$_1$ weakly precedes K$_2$) # Ψ) ▷ Φ)
     ↪$_e$ (((⌈#$^≤$ K$_2$ n, #$^≤$ K$_1$ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
              ⊢ Ψ ▷ ((K$_1$ weakly precedes K$_2$) # Φ))⟩
| strictly_precedes_e:
— A strict precedence relation has to hold at every instant.

⟨(Γ, n ⊢ ((K₁ strictly precedes K₂) # Ψ) ▷ Φ)
  ↪ₑ (((⌈#≤ K₂ n, #< K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
      ⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ))⟩
| kills_e1:
— A kill can be handled by forbidding a tick of the triggering clock.

⟨(Γ, n ⊢ ((K₁ kills K₂) # Ψ) ▷ Φ)
  ↪ₑ (((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ))⟩
| kills_e2:
— It can also be handled by making the triggering clock tick and by forbidding any further tick of the killed
  clock.
⟨(Γ, n ⊢ ((K₁ kills K₂) # Ψ) ▷ Φ)
  ↪ₑ (((K₁ ⇑ n) # (K₂ ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ))⟩

A step of the operational semantics is either the application of the introduction rule or the
application of an elimination rule.

inductive operational_semantics_step
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪ _" 70)
where
  intro_part:
  ⟨(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ↪ᵢ (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)
    ⟹ (Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ↪ (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)⟩
| elims_part:
  ⟨(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ↪ₑ (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)
    ⟹ (Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ↪ (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)⟩

We introduce notations for the reflexive transitive closure of the operational semantic step, its
transitive closure and its reflexive closure.

abbreviation operational_semantics_step_rtranclp
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪** _" 70)
where
  ⟨𝒞₁ ↪** 𝒞₂ ≡ operational_semantics_step** 𝒞₁ 𝒞₂⟩

abbreviation operational_semantics_step_tranclp
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪++ _" 70)
where
  ⟨𝒞₁ ↪++ 𝒞₂ ≡ operational_semantics_step++ 𝒞₁ 𝒞₂⟩

abbreviation operational_semantics_step_reflclp
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪== _" 70)
where
  ⟨𝒞₁ ↪== 𝒞₂ ≡ operational_semantics_step== 𝒞₁ 𝒞₂⟩

abbreviation operational_semantics_step_relpowp
  ::⟨('τ::linordered_field) config ⇒ nat ⇒ 'τ config ⇒ bool⟩   ("_ ↪- _" 70)
where
  ⟨𝒞₁ ↪ⁿ 𝒞₂ ≡ (operational_semantics_step ^^ n) 𝒞₁ 𝒞₂⟩

definition operational_semantics_elim_inv
  ::⟨('τ::linordered_field) config ⇒ 'τ config ⇒ bool⟩   ("_ ↪ₑ← _" 70)
where
  ⟨𝒞₁ ↪ₑ← 𝒞₂ ≡ 𝒞₂ ↪ₑ 𝒞₁⟩

5.2 Basic Lemmas

If a configuration can be reached in m steps from a configuration that can be reached in n
steps from an original configuration, then it can be reached in n + m steps from the original

configuration.

lemma `operational_semantics_trans_generalized`:
  assumes ⟨$\mathcal{C}_1 \hookrightarrow^n \mathcal{C}_2$⟩
  assumes ⟨$\mathcal{C}_2 \hookrightarrow^m \mathcal{C}_3$⟩
    shows ⟨$\mathcal{C}_1 \hookrightarrow^{n + m} \mathcal{C}_3$⟩
⟨*proof*⟩

We consider the set of configurations that can be reached in one operational step from a given configuration.

abbreviation `Cnext_solve`
  ::⟨`('τ::linordered_field) config ⇒ 'τ config set`⟩ ("$\mathcal{C}_{next}$ _")
where
  ⟨$\mathcal{C}_{next}$ $\mathcal{S}$ ≡ { $\mathcal{S}$'. $\mathcal{S}$ ↪ $\mathcal{S}$' }⟩

Advancing to the next instant is possible when there are no more constraints on the current instant.

lemma `Cnext_solve_instant`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ [] ▷ Φ)) ⊇ { Γ, Suc n ⊢ Φ ▷ [] }⟩
⟨*proof*⟩

The following lemmas state that the configurations produced by the elimination rules of the operational semantics belong to the configurations that can be reached in one step.

lemma `Cnext_solve_sporadicon`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K_1 sporadic τ on K_2) # Ψ) ▷ Φ))
    ⊇ { Γ, n ⊢ Ψ ▷ ((K_1 sporadic τ on K_2) # Φ),
      ((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ }⟩
⟨*proof*⟩

lemma `Cnext_solve_tagrel`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Ψ) ▷ Φ))
    ⊇ { ((⌊$\tau_{var}(K_1, n)$, $\tau_{var}(K_2, n)$⌋ ∈ R) # Γ),n
        ⊢ Ψ ▷ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Φ) }⟩
⟨*proof*⟩

lemma `Cnext_solve_implies`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K_1 implies K_2) # Ψ) ▷ Φ))
    ⊇ { ((K_1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 implies K_2) # Φ),
      ((K_1 ⇑ n) # (K_2 ⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 implies K_2) # Φ) }⟩
⟨*proof*⟩

lemma `Cnext_solve_implies_not`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K_1 implies not K_2) # Ψ) ▷ Φ))
    ⊇ { ((K_1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 implies not K_2) # Φ),
      ((K_1 ⇑ n) # (K_2 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 implies not K_2) # Φ) }⟩
⟨*proof*⟩

lemma `Cnext_solve_timedelayed`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K_1 time-delayed by δτ on K_2 implies K_3) # Ψ) ▷ Φ))
    ⊇ { ((K_1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 time-delayed by δτ on K_2 implies K_3) # Φ),
      ((K_1 ⇑ n) # (K_2 @ n ⊕ δτ ⇒ K_3) # Γ), n
        ⊢ Ψ ▷ ((K_1 time-delayed by δτ on K_2 implies K_3) # Φ) }⟩
⟨*proof*⟩

lemma `Cnext_solve_weakly_precedes`:
  ⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K_1 weakly precedes K_2) # Ψ) ▷ Φ))
    ⊇ { ((⌈#$^≤$ K_2 n, #$^≤$ K_1 n⌉ ∈ (λ(x,y). x≤y)) # Γ), n

⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ) })
⟨*proof*⟩

lemma `Cnext_solve_strictly_precedes:`
⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K₁ strictly precedes K₂) # Ψ) ▷ Φ))
⊇ { ((⌈#$^{\leq}$ K₂ n, #$^{<}$ K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ) })⟩
⟨*proof*⟩

lemma `Cnext_solve_kills:`
⟨($\mathcal{C}_{next}$ (Γ, n ⊢ ((K₁ kills K₂) # Ψ) ▷ Φ))
⊇ { ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ),
((K₁ ⇑ n) # (K₂ ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ) })⟩
⟨*proof*⟩

An empty specification can be reduced to an empty specification for an arbitrary number of steps.

lemma `empty_spec_reductions:`
⟨([], 0 ⊢ [] ▷ []) ↪^{k} ([], k ⊢ [] ▷ [])⟩
⟨*proof*⟩

end

Chapter 6

Equivalence of the Operational and Denotational Semantics

```
theory Corecursive_Prop
  imports
    SymbolicPrimitive
    Operational
    Denotational

begin
```

6.1   Stepwise denotational interpretation of TESL atoms

In order to prove the equivalence of the denotational and operational semantics, we need to be able to ignore the past (for which the constraints are encoded in the context) and consider only the satisfaction of the constraints from a given instant index. For this purpose, we define an interpretation of TESL formulae for a suffix of a run. That interpretation is closely related to the denotational semantics as defined in the preceding chapters.

```
fun TESL_interpretation_atomic_stepwise
    :: ⟨('τ::linordered_field) TESL_atomic ⇒ nat ⇒ 'τ run set⟩ ("⟦ _ ⟧TESL≥ -")
where
  ⟨⟦ K₁ sporadic τ on K₂ ⟧TESL≥ i =
      {ϱ. ∃n≥i. hamlet ((Rep_run ϱ) n K₁) ∧ time ((Rep_run ϱ) n K₂) = τ}⟩
| ⟨⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL≥ i =
      {ϱ. ∀n≥i. R (time ((Rep_run ϱ) n K₁), time ((Rep_run ϱ) n K₂))}⟩
| ⟨⟦ master implies slave ⟧TESL≥ i =
      {ϱ. ∀n≥i. hamlet ((Rep_run ϱ) n master) ⟶ hamlet ((Rep_run ϱ) n slave)}⟩
| ⟨⟦ master implies not slave ⟧TESL≥ i =
      {ϱ. ∀n≥i. hamlet ((Rep_run ϱ) n master) ⟶ ¬ hamlet ((Rep_run ϱ) n slave)}⟩
| ⟨⟦ master time-delayed by δτ on measuring implies slave ⟧TESL≥ i =
      {ϱ. ∀n≥i. hamlet ((Rep_run ϱ) n master) ⟶
              (let measured_time = time ((Rep_run ϱ) n measuring) in
                ∀m ≥ n. first_time ϱ measuring m (measured_time + δτ)
                        ⟶ hamlet ((Rep_run ϱ) m slave)
              )
      }⟩
| ⟨⟦ K₁ weakly precedes K₂ ⟧TESL≥ i =
      {ϱ. ∀n≥i. (run_tick_count ϱ K₂ n) ≤ (run_tick_count ϱ K₁ n)}⟩
```

```
|  ⟨⟦ K₁ strictly precedes K₂ ⟧_TESL^≥ ⁱ =
       {ϱ. ∀n≥i. (run_tick_count ϱ K₂ n) ≤ (run_tick_count_strictly ϱ K₁ n)}⟩
|  ⟨⟦ K₁ kills K₂ ⟧_TESL^≥ ⁱ =
       {ϱ. ∀n≥i. hamlet ((Rep_run ϱ) n K₁) ⟶ (∀m≥n. ¬ hamlet ((Rep_run ϱ) m K₂))}⟩
```

The denotational interpretation of TESL formulae can be unfolded into the stepwise interpretation.

lemma `TESL_interp_unfold_stepwise_sporadicon`:
  ⟨⟦ K₁ sporadic τ on K₂ ⟧_TESL = ⋃ {Y. ∃n::nat. Y = ⟦ K₁ sporadic τ on K₂ ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_tagrelgen`:
  ⟨⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧_TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_implies`:
  ⟨⟦ master implies slave ⟧_TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ master implies slave ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_implies_not`:
  ⟨⟦ master implies not slave ⟧_TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ master implies not slave ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_timedelayed`:
  ⟨⟦ master time-delayed by $\delta\tau$ on measuring implies slave ⟧_TESL
    = ⋂ {Y. ∃n::nat.
          Y = ⟦ master time-delayed by $\delta\tau$ on measuring implies slave ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_weakly_precedes`:
  ⟨⟦ K₁ weakly precedes K₂ ⟧_TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ K₁ weakly precedes K₂ ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_strictly_precedes`:
  ⟨⟦ K₁ strictly precedes K₂ ⟧_TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ K₁ strictly precedes K₂ ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

lemma `TESL_interp_unfold_stepwise_kills`:
  ⟨⟦ master kills slave ⟧_TESL = ⋂ {Y. ∃n::nat. Y = ⟦ master kills slave ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

Positive atomic formulae (the ones that create ticks from nothing) are unfolded as the union of the stepwise interpretations.

theorem `TESL_interp_unfold_stepwise_positive_atoms`:
  assumes ⟨positive_atom φ⟩
    shows ⟨⟦ φ::'τ::linordered_field TESL_atomic ⟧_TESL
            = ⋃ {Y. ∃n::nat. Y = ⟦ φ ⟧_TESL^≥ ⁿ}⟩
⟨*proof*⟩

Negative atomic formulae are unfolded as the intersection of the stepwise interpretations.

theorem `TESL_interp_unfold_stepwise_negative_atoms`:
  assumes ⟨¬ positive_atom φ⟩

    shows $\langle[\![\ \varphi\ ]\!]_{TESL} = \bigcap\ \{$Y. $\exists$n::nat. Y $= [\![\ \varphi\ ]\!]_{TESL}^{\geq\ \text{n}}\}\rangle$
$\langle proof \rangle$

Some useful lemmas for reasoning on properties of sequences.

lemma `forall_nat_expansion`:
  $\langle(\forall$n $\geq$ (n_0::nat). P n) = (P n_0 $\wedge$ ($\forall$n $\geq$ Suc n_0. P n))$\rangle$
$\langle proof \rangle$

lemma `exists_nat_expansion`:
  $\langle(\exists$n $\geq$ (n_0::nat). P n) = (P n_0 $\vee$ ($\exists$n $\geq$ Suc n_0. P n))$\rangle$
$\langle proof \rangle$

lemma `forall_nat_set_suc`:$\langle\{$x. $\forall$m $\geq$ n. P x m$\}$ = $\{$x. P x n$\}$ $\cap$ $\{$x. $\forall$m $\geq$ Suc n. P x m$\}\rangle$
$\langle proof \rangle$

lemma `exists_nat_set_suc`:$\langle\{$x. $\exists$m $\geq$ n. P x m$\}$ = $\{$x. P x n$\}$ $\cup$ $\{$x. $\exists$m $\geq$ Suc n. P x m$\}\rangle$
$\langle proof \rangle$

6.2   Coinduction Unfolding Properties

The following lemmas show how to shorten a suffix, i.e. to unfold one instant in the construction of a run. They correspond to the rules of the operational semantics.

lemma `TESL_interp_stepwise_sporadicon_coind_unfold`:
  $\langle[\![\ K_1\ \texttt{sporadic}\ \tau\ \texttt{on}\ K_2\ ]\!]_{TESL}^{\geq\ \text{n}} =$
    $[\![\ K_1\ \Uparrow\ \text{n}\ ]\!]_{prim} \cap [\![\ K_2\ \Downarrow\ \text{n}\ @\ \tau\ ]\!]_{prim}$ — rule `sporadic_on_e2`
    $\cup [\![\ K_1\ \texttt{sporadic}\ \tau\ \texttt{on}\ K_2\ ]\!]_{TESL}^{\geq\ \text{Suc n}}\rangle$ — rule `sporadic_on_e1`
$\langle proof \rangle$

lemma `TESL_interp_stepwise_tagrel_coind_unfold`:
  $\langle[\![\ \texttt{time-relation}\ \lfloor K_1,\ K_2 \rfloor \in R\ ]\!]_{TESL}^{\geq\ \text{n}} =$ — rule `tagrel_e`
    $[\![\ \lfloor \tau_{var}(K_1,\ \text{n}),\ \tau_{var}(K_2,\ \text{n}) \rfloor \in R\ ]\!]_{prim}$
    $\cap [\![\ \texttt{time-relation}\ \lfloor K_1,\ K_2 \rfloor \in R\ ]\!]_{TESL}^{\geq\ \text{Suc n}}\rangle$
$\langle proof \rangle$

lemma `TESL_interp_stepwise_implies_coind_unfold`:
  $\langle[\![\ \texttt{master implies slave}\ ]\!]_{TESL}^{\geq\ \text{n}} =$
    $(\quad[\![\ \texttt{master}\ \neg\Uparrow\ \text{n}\ ]\!]_{prim}$ — rule `implies_e1`
     $\cup [\![\ \texttt{master}\ \Uparrow\ \text{n}\ ]\!]_{prim} \cap [\![\ \texttt{slave}\ \Uparrow\ \text{n}\ ]\!]_{prim})$ — rule `implies_e2`
    $\cap [\![\ \texttt{master implies slave}\ ]\!]_{TESL}^{\geq\ \text{Suc n}}\rangle$
$\langle proof \rangle$

lemma `TESL_interp_stepwise_implies_not_coind_unfold`:
  $\langle[\![\ \texttt{master implies not slave}\ ]\!]_{TESL}^{\geq\ \text{n}} =$
    $(\quad[\![\ \texttt{master}\ \neg\Uparrow\ \text{n}\ ]\!]_{prim}$ — rule `implies_not_e1`
     $\cup [\![\ \texttt{master}\ \Uparrow\ \text{n}\ ]\!]_{prim} \cap [\![\ \texttt{slave}\ \neg\Uparrow\ \text{n}\ ]\!]_{prim})$ — rule `implies_not_e2`
    $\cap [\![\ \texttt{master implies not slave}\ ]\!]_{TESL}^{\geq\ \text{Suc n}}\rangle$
$\langle proof \rangle$

lemma `TESL_interp_stepwise_timedelayed_coind_unfold`:
  $\langle[\![\ \texttt{master time-delayed by}\ \delta\tau\ \texttt{on measuring implies slave}\ ]\!]_{TESL}^{\geq\ \text{n}} =$
    $(\quad[\![\ \texttt{master}\ \neg\Uparrow\ \text{n}\ ]\!]_{prim}$ — rule `timedelayed_e1`
     $\cup ([\![\ \texttt{master}\ \Uparrow\ \text{n}\ ]\!]_{prim} \cap [\![\ \texttt{measuring}\ @\ \text{n}\ \oplus\ \delta\tau\ \Rightarrow\ \texttt{slave}\ ]\!]_{prim}))$
                         — rule `timedelayed_e2`

$\cap$ ⟦ master time-delayed by $\delta\tau$ on measuring implies slave ⟧$_{TESL}^{\geq \text{ Suc n}}$⟩
⟨*proof*⟩

lemma TESL_interp_stepwise_weakly_precedes_coind_unfold:
   ⟨⟦ K$_1$ weakly precedes K$_2$ ⟧$_{TESL}^{\geq \text{ n}}$ =   — rule weakly_precedes_e

     ⟦ ($\lceil$#$^{\leq}$ K$_2$ n, #$^{\leq}$ K$_1$ n$\rceil$ $\in$ (λ(x,y). x$\leq$y)) ⟧$_{prim}$
     $\cap$ ⟦ K$_1$ weakly precedes K$_2$ ⟧$_{TESL}^{\geq \text{ Suc n}}$⟩
⟨*proof*⟩

lemma TESL_interp_stepwise_strictly_precedes_coind_unfold:
   ⟨⟦ K$_1$ strictly precedes K$_2$ ⟧$_{TESL}^{\geq \text{ n}}$ =   — rule strictly_precedes_e

     ⟦ ($\lceil$#$^{\leq}$ K$_2$ n, #$^{<}$ K$_1$ n$\rceil$ $\in$ (λ(x,y). x$\leq$y)) ⟧$_{prim}$
     $\cap$ ⟦ K$_1$ strictly precedes K$_2$ ⟧$_{TESL}^{\geq \text{ Suc n}}$⟩
⟨*proof*⟩

lemma TESL_interp_stepwise_kills_coind_unfold:
   ⟨⟦ K$_1$ kills K$_2$ ⟧$_{TESL}^{\geq \text{ n}}$ =
     (⟦ K$_1$ ¬⇑ n ⟧$_{prim}$ — rule kills_e1

      $\cup$ ⟦ K$_1$ ⇑ n ⟧$_{prim}$ $\cap$ ⟦ K$_2$ ¬⇑ $\geq$ n ⟧$_{prim}$)   — rule kills_e2
     $\cap$ ⟦ K$_1$ kills K$_2$ ⟧$_{TESL}^{\geq \text{ Suc n}}$⟩
⟨*proof*⟩

The stepwise interpretation of a TESL formula is the intersection of the interpretation of its atomic components.

fun TESL_interpretation_stepwise
  ::⟨'τ::linordered_field TESL_formula $\Rightarrow$ nat $\Rightarrow$ 'τ run set⟩
  ("⟦⟦ _ ⟧⟧$_{TESL}^{\geq}$ -")
where
   ⟨⟦⟦ [] ⟧⟧$_{TESL}^{\geq \text{ n}}$ = {ϱ. True}⟩
| ⟨⟦⟦ φ # Φ ⟧⟧$_{TESL}^{\geq \text{ n}}$ = ⟦ φ ⟧$_{TESL}^{\geq \text{ n}}$ $\cap$ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq \text{ n}}$⟩

lemma TESL_interpretation_stepwise_fixpoint:
   ⟨⟦⟦ Φ ⟧⟧$_{TESL}^{\geq \text{ n}}$ = $\bigcap$ (($\lambda\varphi$. ⟦ φ ⟧$_{TESL}^{\geq \text{ n}}$) ` set Φ)⟩
⟨*proof*⟩

The global interpretation of a TESL formula is its interpretation starting at the first instant.

lemma TESL_interpretation_stepwise_zero:
   ⟨⟦ φ ⟧$_{TESL}$ = ⟦ φ ⟧$_{TESL}^{\geq \text{ 0}}$⟩
⟨*proof*⟩

lemma TESL_interpretation_stepwise_zero':
   ⟨⟦⟦ Φ ⟧⟧$_{TESL}$ = ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq \text{ 0}}$⟩
⟨*proof*⟩

lemma TESL_interpretation_stepwise_cons_morph:
   ⟨⟦ φ ⟧$_{TESL}^{\geq \text{ n}}$ $\cap$ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq \text{ n}}$ = ⟦⟦ φ # Φ ⟧⟧$_{TESL}^{\geq \text{ n}}$⟩
⟨*proof*⟩

theorem TESL_interp_stepwise_composition:
   shows ⟨⟦⟦ Φ_1 @ Φ_2 ⟧⟧$_{TESL}^{\geq \text{ n}}$ = ⟦⟦ Φ_1 ⟧⟧$_{TESL}^{\geq \text{ n}}$ $\cap$ ⟦⟦ Φ_2 ⟧⟧$_{TESL}^{\geq \text{ n}}$⟩
⟨*proof*⟩

6.3 Interpretation of configurations

The interpretation of a configuration of the operational semantics abstract machine is the intersection of:

- the interpretation of its context (the past),

- the interpretation of its present from the current instant,

- the interpretation of its future from the next instant.

fun `HeronConf_interpretation`
   $::\langle'\tau::\texttt{linordered\_field config} \Rightarrow '\tau \texttt{ run set}\rangle$ ("$[\![$ _ $]\!]_{config}$" 71)
where
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~\Psi~\triangleright~\Phi~]\!]_{config} = [\![[\![~\Gamma~]\!]]\!]_{prim} \cap [\![[\![~\Psi~]\!]]\!]_{TESL}{}^{\geq \texttt{ n}} \cap [\![[\![~\Phi~]\!]]\!]_{TESL}{}^{\geq \texttt{ Suc n}}\rangle$

lemma `HeronConf_interp_composition`:
   $\langle[\![~\Gamma_1,~\texttt{n}~\vdash~\Psi_1~\triangleright~\Phi_1~]\!]_{config} \cap [\![~\Gamma_2,~\texttt{n}~\vdash~\Psi_2~\triangleright~\Phi_2~]\!]_{config}$
      $= [\![~(\Gamma_1~@~\Gamma_2),~\texttt{n}~\vdash~(\Psi_1~@~\Psi_2)~\triangleright~(\Phi_1~@~\Phi_2)~]\!]_{config}\rangle$
   $\langle proof \rangle$

When there are no remaining constraints on the present, the interpretation of a configuration is the same as the configuration at the next instant of its future. This corresponds to the introduction rule of the operational semantics.

lemma `HeronConf_interp_stepwise_instant_cases`:
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~[]~\triangleright~\Phi~]\!]_{config} = [\![~\Gamma,~\texttt{Suc n}~\vdash~\Phi~\triangleright~[]~]\!]_{config}\rangle$
$\langle proof \rangle$

The following lemmas use the unfolding properties of the stepwise denotational semantics to give rewriting rules for the interpretation of configurations that match the elimination rules of the operational semantics.

lemma `HeronConf_interp_stepwise_sporadicon_cases`:
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~((K_1~\texttt{sporadic}~\tau~\texttt{on}~K_2)~\#~\Psi)~\triangleright~\Phi~]\!]_{config}$
      $= [\![~\Gamma,~\texttt{n}~\vdash~\Psi~\triangleright~((K_1~\texttt{sporadic}~\tau~\texttt{on}~K_2)~\#~\Phi)~]\!]_{config}$
      $\cup~[\![~((K_1 \Uparrow \texttt{n})~\#~(K_2 \Downarrow \texttt{n}~@~\tau)~\#~\Gamma),~\texttt{n}~\vdash~\Psi~\triangleright~\Phi~]\!]_{config}\rangle$
$\langle proof \rangle$

lemma `HeronConf_interp_stepwise_tagrel_cases`:
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~((\texttt{time-relation}~\lfloor K_1,~K_2 \rfloor \in R)~\#~\Psi)~\triangleright~\Phi~]\!]_{config}$
      $= [\![~((\lfloor \tau_{var}(K_1,~\texttt{n}),~\tau_{var}(K_2,~\texttt{n}) \rfloor \in R)~\#~\Gamma),~\texttt{n}$
         $\vdash~\Psi~\triangleright~((\texttt{time-relation}~\lfloor K_1,~K_2 \rfloor \in R)~\#~\Phi)~]\!]_{config}\rangle$
$\langle proof \rangle$

lemma `HeronConf_interp_stepwise_implies_cases`:
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~((K_1~\texttt{implies}~K_2)~\#~\Psi)~\triangleright~\Phi~]\!]_{config}$
      $= [\![~((K_1 \neg\Uparrow \texttt{n})~\#~\Gamma),~\texttt{n}~\vdash~\Psi~\triangleright~((K_1~\texttt{implies}~K_2)~\#~\Phi)~]\!]_{config}$
      $\cup~[\![~((K_1 \Uparrow \texttt{n})~\#~(K_2 \Uparrow \texttt{n})~\#~\Gamma),~\texttt{n}~\vdash~\Psi~\triangleright~((K_1~\texttt{implies}~K_2)~\#~\Phi)~]\!]_{config}\rangle$
$\langle proof \rangle$

lemma `HeronConf_interp_stepwise_implies_not_cases`:
   $\langle[\![~\Gamma,~\texttt{n}~\vdash~((K_1~\texttt{implies not}~K_2)~\#~\Psi)~\triangleright~\Phi~]\!]_{config}$
      $= [\![~((K_1 \neg\Uparrow \texttt{n})~\#~\Gamma),~\texttt{n}~\vdash~\Psi~\triangleright~((K_1~\texttt{implies not}~K_2)~\#~\Phi)~]\!]_{config}$
      $\cup~[\![~((K_1 \Uparrow \texttt{n})~\#~(K_2 \neg\Uparrow \texttt{n})~\#~\Gamma),~\texttt{n}~\vdash~\Psi~\triangleright~((K_1~\texttt{implies not}~K_2)~\#~\Phi)~]\!]_{config}\rangle$
$\langle proof \rangle$

lemma `HeronConf_interp_stepwise_timedelayed_cases`:

⟨⟦ Γ, n ⊢ ((K₁ time-delayed by $\delta\tau$ on K₂ implies K₃) # Ψ) ▷ Φ ⟧$_{config}$
  = ⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ time-delayed by $\delta\tau$ on K₂ implies K₃) # Φ) ⟧$_{config}$
  ∪ ⟦ ((K₁ ⇑ n) # (K₂ @ n ⊕ $\delta\tau$ ⇒ K₃) # Γ), n
      ⊢ Ψ ▷ ((K₁ time-delayed by $\delta\tau$ on K₂ implies K₃) # Φ) ⟧$_{config}$⟩
⟨*proof*⟩

lemma `HeronConf_interp_stepwise_weakly_precedes_cases`:
  ⟨⟦ Γ, n ⊢ ((K₁ weakly precedes K₂) # Ψ) ▷ Φ ⟧$_{config}$
  = ⟦ ((⌈#$^{\leq}$ K₂ n, #$^{\leq}$ K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
      ⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ) ⟧$_{config}$⟩
⟨*proof*⟩

lemma `HeronConf_interp_stepwise_strictly_precedes_cases`:
  ⟨⟦ Γ, n ⊢ ((K₁ strictly precedes K₂) # Ψ) ▷ Φ ⟧$_{config}$
  = ⟦ ((⌈#$^{\leq}$ K₂ n, #$^{<}$ K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
      ⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ) ⟧$_{config}$⟩
⟨*proof*⟩

lemma `HeronConf_interp_stepwise_kills_cases`:
  ⟨⟦ Γ, n ⊢ ((K₁ kills K₂) # Ψ) ▷ Φ ⟧$_{config}$
  = ⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ) ⟧$_{config}$
  ∪ ⟦ ((K₁ ⇑ n) # (K₂ ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K₁ kills K₂) # Φ) ⟧$_{config}$⟩
⟨*proof*⟩

end

Chapter 7

Main Theorems

theory Hygge_Theory
imports
  Corecursive_Prop

begin

Using the properties we have shown about the interpretation of configurations and the stepwise unfolding of the denotational semantics, we can now prove several important results about the construction of runs from a specification.

7.1 Initial configuration

The denotational semantics of a specification Ψ is the interpretation at the first instant of a configuration which has Ψ as its present. This means that we can start to build a run that satisfies a specification by starting from this configuration.

theorem solve_start:
  shows $\langle [\![\ \Psi\ ]\!]_{TESL} = [\![\ [], 0 \vdash \Psi \rhd [] \ ]\!]_{config} \rangle$
  $\langle proof \rangle$

7.2 Soundness

The interpretation of a configuration $\mathcal{S}_2$ that is a refinement of a configuration $\mathcal{S}_1$ is contained in the interpretation of $\mathcal{S}_1$. This means that by making successive choices in building the instants of a run, we preserve the soundness of the constructed run with regard to the original specification.

lemma sound_reduction:
  assumes $\langle (\Gamma_1,\ n_1 \vdash \Psi_1 \rhd \Phi_1)\ \hookrightarrow\ (\Gamma_2,\ n_2 \vdash \Psi_2 \rhd \Phi_2) \rangle$
  shows $\langle [\![\ \Gamma_1\ ]\!]_{prim} \cap [\![\ \Psi_1\ ]\!]_{TESL}^{\geq\ n_1} \cap [\![\ \Phi_1\ ]\!]_{TESL}^{\geq\ Suc\ n_1}$
        $\supseteq\ [\![\ \Gamma_2\ ]\!]_{prim} \cap [\![\ \Psi_2\ ]\!]_{TESL}^{\geq\ n_2} \cap [\![\ \Phi_2\ ]\!]_{TESL}^{\geq\ Suc\ n_2} \rangle$ **(is ?P)**
$\langle proof \rangle$

inductive_cases step_elim:$\langle \mathcal{S}_1 \hookrightarrow \mathcal{S}_2 \rangle$

lemma sound_reduction':
  assumes $\langle \mathcal{S}_1 \hookrightarrow \mathcal{S}_2 \rangle$
  shows $\langle [\![\ \mathcal{S}_1\ ]\!]_{config} \supseteq [\![\ \mathcal{S}_2\ ]\!]_{config} \rangle$
$\langle proof \rangle$

```
lemma sound_reduction_generalized:
  assumes ⟨S₁ ↪ᵏ S₂⟩
    shows ⟨⟦ S₁ ⟧_config ⊇ ⟦ S₂ ⟧_config⟩
⟨proof⟩
```

From the initial configuration, a configuration $\mathcal{S}$ obtained after any number k of reduction steps denotes runs from the initial specification Ψ.

```
theorem soundness:
  assumes ⟨([], 0 ⊢ Ψ ▷ []) ↪ᵏ S⟩
    shows ⟨⟦⟦ Ψ ⟧⟧_TESL ⊇ ⟦ S ⟧_config⟩
  ⟨proof⟩
```

7.3   Completeness

We will now show that any run that satisfies a specification can be derived from the initial configuration, at any number of steps.

We start by proving that any run that is denoted by a configuration $\mathcal{S}$ is necessarily denoted by at least one of the configurations that can be reached from $\mathcal{S}$.

```
lemma complete_direct_successors:
  shows ⟨⟦ Γ, n ⊢ Ψ ▷ Φ ⟧_config ⊆ (⋃X∈𝒞_next (Γ, n ⊢ Ψ ▷ Φ). ⟦ X ⟧_config)⟩
  ⟨proof⟩
```

```
lemma complete_direct_successors':
  shows ⟨⟦ S ⟧_config ⊆ (⋃X∈𝒞_next S. ⟦ X ⟧_config)⟩
⟨proof⟩
```

Therefore, if a run belongs to a configuration, it necessarily belongs to a configuration derived from it.

```
lemma branch_existence:
  assumes ⟨ϱ ∈ ⟦ S₁ ⟧_config⟩
  shows ⟨∃S₂. (S₁ ↪ S₂) ∧ (ϱ ∈ ⟦ S₂ ⟧_config)⟩
⟨proof⟩
```

```
lemma branch_existence':
  assumes ⟨ϱ ∈ ⟦ S₁ ⟧_config⟩
  shows ⟨∃S₂. (S₁ ↪ᵏ S₂) ∧ (ϱ ∈ ⟦ S₂ ⟧_config)⟩
⟨proof⟩
```

Any run that belongs to the original specification Ψ has a corresponding configuration $\mathcal{S}$ at any number k of reduction steps from the initial configuration. Therefore, any run that satisfies a specification can be derived from the initial configuration at any level of reduction.

```
theorem completeness:
  assumes ⟨ϱ ∈ ⟦⟦ Ψ ⟧⟧_TESL⟩
  shows ⟨∃S. (([], 0 ⊢ Ψ ▷ [])  ↪ᵏ  S)
            ∧ ϱ ∈ ⟦ S ⟧_config⟩
  ⟨proof⟩
```

7.4   Progress

Reduction steps do not guarantee that the construction of a run progresses in the sequence of instants. We need to show that it is always possible to reach the next instant, and therefore any future instant, through a number of steps.

```
lemma instant_index_increase:
  assumes ⟨ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ Φ ⟧config⟩
  shows    ⟨∃Γk Ψk Φk k. ((Γ, n ⊢ Ψ ▷ Φ) ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
                       ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config⟩
⟨proof⟩
```

```
lemma instant_index_increase_generalized:
  assumes ⟨n < nk⟩
  assumes ⟨ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ Φ ⟧config⟩
  shows    ⟨∃Γk Ψk Φk k. ((Γ, n ⊢ Ψ ▷ Φ) ↪k (Γk, nk ⊢ Ψk ▷ Φk))
                       ∧ ϱ ∈ ⟦ Γk, nk ⊢ Ψk ▷ Φk ⟧config⟩
⟨proof⟩
```

Any run that belongs to a specification Ψ has a corresponding configuration that develops it up to the nth instant.

```
theorem progress:
  assumes ⟨ϱ ∈ ⟦⟦ Ψ ⟧⟧TESL⟩
    shows ⟨∃k Γk Ψk Φk. (([], 0 ⊢ Ψ ▷ [])  ↪k (Γk, n ⊢ Ψk ▷ Φk))
                       ∧ ϱ ∈ ⟦ Γk, n ⊢ Ψk ▷ Φk ⟧config⟩
⟨proof⟩
```

7.5 Local termination

Here, we prove that the computation of an instant in a run always terminates. Since this computation terminates when the list of constraints for the present instant becomes empty, we introduce a measure for this formula.

```
primrec measure_interpretation :: ⟨'τ::linordered_field TESL_formula ⇒ nat⟩ ("μ")
where
  ⟨μ [] = (0::nat)⟩
| ⟨μ (φ # Φ) = (case φ of
                   _ sporadic _ on _ ⇒ 1 + μ Φ
                 | _                ⇒ 2 + μ Φ)⟩
```

```
fun measure_interpretation_config :: ⟨'τ::linordered_field config ⇒ nat⟩ ("μconfig")
where
  ⟨μconfig (Γ, n ⊢ Ψ ▷ Φ) = μ Ψ⟩
```

We then show that the elimination rules make this measure decrease.

```
lemma elimination_rules_strictly_decreasing:
  assumes ⟨(Γ1, n1 ⊢ Ψ1 ▷ Φ1)  ↪e  (Γ2, n2 ⊢ Ψ2 ▷ Φ2)⟩
    shows ⟨μ Ψ1 > μ Ψ2⟩
⟨proof⟩
```

```
lemma elimination_rules_strictly_decreasing_meas:
  assumes ⟨(Γ1, n1 ⊢ Ψ1 ▷ Φ1)  ↪e  (Γ2, n2 ⊢ Ψ2 ▷ Φ2)⟩
    shows ⟨(Ψ2, Ψ1) ∈ measure μ⟩
⟨proof⟩
```

```
lemma elimination_rules_strictly_decreasing_meas':
  assumes ⟨S1  ↪e  S2⟩
  shows ⟨(S2, S1) ∈ measure μconfig⟩
⟨proof⟩
```

Therefore, the relation made up of elimination rules is well-founded and the computation of an instant terminates.

theorem `instant_computation_termination`:
  ⟨`wfP` (λ($\mathcal{S}_1$::'a::`linordered_field config`) $\mathcal{S}_2$. ($\mathcal{S}_1$ ↪$_e$$^←$ $\mathcal{S}_2$))⟩
⟨*proof*⟩

end

Chapter 8

Properties of TESL

8.1 Stuttering Invariance

theory `StutteringDefs`

imports `Denotational`

begin

When composing systems into more complex systems, it may happen that one system has to perform some action while the rest of the complex system does nothing. In order to support the composition of TESL specifications, we want to be able to insert stuttering instants in a run without breaking the conformance of a run to its specification. This is what we call the *stuttering invariance* of TESL.

8.1.1 Definition of stuttering

We consider stuttering as the insertion of empty instants (instants at which no clock ticks) in a run. We caracterize this insertion with a dilating function, which maps the instant indices of the original run to the corresponding instant indices of the dilated run. The properties of a dilating function are:

- it is strictly increasing because instants are inserted into the run,

- the image of an instant index is greater than it because stuttering instants can only delay the original instants of the run,

- no instant is inserted before the first one in order to have a well defined initial date on each clock,

- if n is not in the image of the function, no clock ticks at instant n and the date on the clocks do not change.

definition `dilating_fun`
where
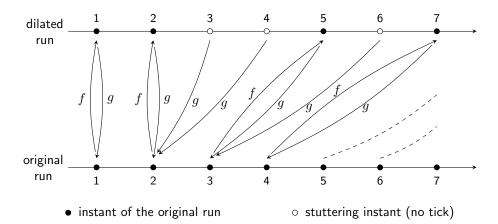 ⟨`dilating_fun` (f::nat $\Rightarrow$ nat) (r::'a::linordered_field run)
    $\equiv$ strict_mono f $\land$ (f 0 = 0) $\land$ ($\forall$n. f n $\geq$ n
    $\land$ (($\nexists n_0$. f n_0 = n) $\longrightarrow$ ($\forall$c. $\neg$(hamlet ((Rep_run r) n c))))⟩

Figure 8.1: Dilating and contracting functions

$$\wedge \ ((\nexists n_0. \ f \ n_0 = (Suc \ n)) \longrightarrow (\forall c. \ time \ ((Rep\_run \ r) \ (Suc \ n) \ c)$$
$$= time \ ((Rep\_run \ r) \ n \ c)))$$
$$)\rangle$$

A run `r` is a dilation of a run `sub` by function `f` if:

- `f` is a dilating function for `r`

- the time in `r` is the time in `sub` dilated by `f`

- the hamlet in `r` is the hamlet in sub dilated by `f`

definition dilating
where
  ⟨dilating f sub r ≡ dilating_fun f r
                    ∧ (∀n c. time ((Rep_run sub) n c) = time ((Rep_run r) (f n) c))
                    ∧ (∀n c. hamlet ((Rep_run sub) n c) = hamlet ((Rep_run r) (f n) c))⟩

A *run* is a *subrun* of another run if there exists a dilation between them.

definition is_subrun ::⟨'a::linordered_field run ⇒ 'a run ⇒ bool⟩ (**infixl** "≪" 60)
where
  ⟨sub ≪ r ≡ (∃f. dilating f sub r)⟩

A contracting function is the reverse of a dilating fun, it maps an instant index of a dilated run to the index of the last instant of a non stuttering run that precedes it. Since several successive stuttering instants are mapped to the same instant of the non stuttering run, such a function is monotonous, but not strictly. The image of the first instant of the dilated run is necessarily the first instant of the non stuttering run, and the image of an instant index is less that this index because we remove stuttering instants.

definition contracting_fun
  where ⟨contracting_fun g ≡ mono g ∧ g 0 = 0 ∧ (∀n. g n ≤ n)⟩

Figure 8.1 illustrates the relations between the instants of a run and the instants of a dilated run, with the mappings by the dilating function `f` and the contracting function `g`:

A function `g` is contracting with respect to the dilation of run `sub` into run `r` by the dilating function `f` if:

- it is a contracting function ;

- (f ○ g) n is the index of the last original instant before instant n in run r, therefore:

  - (f ○ g) n ≤ n
  - the time does not change on any clock between instants (f ○ g) n and n of run r;
  - no clock ticks before n strictly after (f ○ g) n in run r. See Figure 8.1 for a better understanding. Notice that in this example, 2 is equal to (f ○ g) 2, (f ○ g) 3, and (f ○ g) 4.

definition contracting
where
  ⟨contracting g r sub f ≡ contracting_fun g
                           ∧ (∀n. f (g n) ≤ n)
                           ∧ (∀n c k. f (g n) ≤ k ∧ k ≤ n
                               ⟶ time ((Rep_run r) k c) = time ((Rep_run sub) (g n) c))
                           ∧ (∀n c k. f (g n) < k ∧ k ≤ n
                               ⟶ ¬ hamlet ((Rep_run r) k c))⟩

For any dilating function, we can build its *inverse*, as illustrated on Figure 8.1, which is a contracting function:

definition ⟨dil_inverse f::(nat ⇒ nat) ≡ (λn. Max {i. f i ≤ n})⟩

8.1.2   Alternate definitions for counting ticks.

For proving the stuttering invariance of TESL specifications, we will need these alternate definitions for counting ticks, which are based on sets.

tick_count r c n is the number of ticks of clock c in run r upto instant n.

definition tick_count :: ⟨'a::linordered_field run ⇒ clock ⇒ nat ⇒ nat⟩
where
  ⟨tick_count r c n = card {i. i ≤ n ∧ hamlet ((Rep_run r) i c)}⟩

tick_count_strict r c n is the number of ticks of clock c in run r upto but excluding instant n.

definition tick_count_strict :: ⟨'a::linordered_field run ⇒ clock ⇒ nat ⇒ nat⟩
where
  ⟨tick_count_strict r c n = card {i. i < n ∧ hamlet ((Rep_run r) i c)}⟩

end

8.1.3   Stuttering Lemmas

theory StutteringLemmas

imports StutteringDefs

begin

In this section, we prove several lemmas that will be used to show that TESL specifications are invariant by stuttering.

The following one will be useful in proving properties over a sequence of stuttering instants.

lemma `bounded_suc_ind`:
  assumes ⟨⋀k. k < m ⟹ P (Suc (z + k)) = P (z + k)⟩
    shows ⟨k < m ⟹ P (Suc (z + k)) = P z⟩
⟨*proof*⟩

8.1.4   Lemmas used to prove the invariance by stuttering

Since a dilating function is strictly monotonous, it is injective.

lemma `dilating_fun_injects`:
  assumes ⟨dilating_fun f r⟩
  shows   ⟨inj_on f A⟩
⟨*proof*⟩

lemma `dilating_injects`:
  assumes ⟨dilating f sub r⟩
  shows   ⟨inj_on f A⟩
⟨*proof*⟩

If a clock ticks at an instant in a dilated run, that instant is the image by the dilating function of an instant of the original run.

lemma `ticks_image`:
  assumes ⟨dilating_fun f r⟩
  and   ⟨hamlet ((Rep_run r) n c)⟩
  shows   ⟨∃n_0. f n_0 = n⟩
⟨*proof*⟩

lemma `ticks_image_sub`:
  assumes ⟨dilating f sub r⟩
  and   ⟨hamlet ((Rep_run r) n c)⟩
  shows   ⟨∃n_0. f n_0 = n⟩
⟨*proof*⟩

lemma `ticks_image_sub'`:
  assumes ⟨dilating f sub r⟩
  and   ⟨∃c. hamlet ((Rep_run r) n c)⟩
  shows   ⟨∃n_0. f n_0 = n⟩
⟨*proof*⟩

The image of the ticks in an interval by a dilating function is the interval bounded by the image of the bounds of the original interval. This is proven for all 4 kinds of intervals: `]m, n[`, `[m, n[`, `]m, n]` and `[m, n]`.

lemma `dilating_fun_image_strict`:
  assumes ⟨dilating_fun f r⟩
  shows   ⟨{k. f m < k ∧ k < f n ∧ hamlet ((Rep_run r) k c)}
          = image f {k. m < k ∧ k < n ∧ hamlet ((Rep_run r) (f k) c)}⟩
  (**is** ⟨?IMG = image f ?SET⟩)
⟨*proof*⟩

lemma `dilating_fun_image_left`:
  assumes ⟨dilating_fun f r⟩
  shows   ⟨{k. f m ≤ k ∧ k < f n ∧ hamlet ((Rep_run r) k c)}
        = image f {k. m ≤ k ∧ k < n ∧ hamlet ((Rep_run r) (f k) c)}⟩
  (**is** ⟨?IMG = image f ?SET⟩)
⟨*proof*⟩

lemma `dilating_fun_image_right:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`{k. f m < k ∧ k ≤ f n ∧ hamlet ((Rep_run r) k c)}`
        `= image f {k. m < k ∧ k ≤ n ∧ hamlet ((Rep_run r) (f k) c)}`⟩
  (**is** ⟨`?IMG = image f ?SET`⟩)
⟨*proof*⟩

lemma `dilating_fun_image:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`{k. f m ≤ k ∧ k ≤ f n ∧ hamlet ((Rep_run r) k c)}`
        `= image f {k. m ≤ k ∧ k ≤ n ∧ hamlet ((Rep_run r) (f k) c)}`⟩
  (**is** ⟨`?IMG = image f ?SET`⟩)
⟨*proof*⟩

On any clock, the number of ticks in an interval is preserved by a dilating function.

lemma `ticks_as_often_strict:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`card {p. n < p ∧ p < m ∧ hamlet ((Rep_run r) (f p) c)}`
        `= card {p. f n < p ∧ p < f m ∧ hamlet ((Rep_run r) p c)}`⟩
    (**is** ⟨`card ?SET = card ?IMG`⟩)
⟨*proof*⟩

lemma `ticks_as_often_left:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`card {p. n ≤ p ∧ p < m ∧ hamlet ((Rep_run r) (f p) c)}`
        `= card {p. f n ≤ p ∧ p < f m ∧ hamlet ((Rep_run r) p c)}`⟩
    (**is** ⟨`card ?SET = card ?IMG`⟩)
⟨*proof*⟩

lemma `ticks_as_often_right:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`card {p. n < p ∧ p ≤ m ∧ hamlet ((Rep_run r) (f p) c)}`
        `= card {p. f n < p ∧ p ≤ f m ∧ hamlet ((Rep_run r) p c)}`⟩
    (**is** ⟨`card ?SET = card ?IMG`⟩)
⟨*proof*⟩

lemma `ticks_as_often:`
  assumes ⟨`dilating_fun f r`⟩
  shows ⟨`card {p. n ≤ p ∧ p ≤ m ∧ hamlet ((Rep_run r) (f p) c)}`
        `= card {p. f n ≤ p ∧ p ≤ f m ∧ hamlet ((Rep_run r) p c)}`⟩
    (**is** ⟨`card ?SET = card ?IMG`⟩)
⟨*proof*⟩

The date of an event is preserved by dilation.

lemma `ticks_tag_image:`
  assumes ⟨`dilating f sub r`⟩
  and   ⟨`∃c. hamlet ((Rep_run r) k c)`⟩
  and   ⟨`time ((Rep_run r) k c) = τ`⟩
  shows ⟨`∃k₀. f k₀ = k ∧ time ((Rep_run sub) k₀ c) = τ`⟩
⟨*proof*⟩

TESL operators are invariant by dilation.

lemma `ticks_sub:`
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`hamlet ((Rep_run sub) n a) = hamlet ((Rep_run r) (f n) a)`⟩
⟨*proof*⟩

lemma `no_tick_sub`:
  assumes ⟨dilating f sub r⟩
  shows   ⟨($\nexists n_0$. f n_0 = n) ⟶ ¬hamlet ((Rep_run r) n a)⟩
⟨*proof*⟩

Lifting a total function to a partial function on an option domain.

definition `opt_lift`::⟨('a ⇒ 'a) ⇒ ('a option ⇒ 'a option)⟩
where
  ⟨opt_lift f ≡ λx. case x of None ⇒ None | Some y ⇒ Some (f y)⟩

The set of instants when a clock ticks in a dilated run is the image by the dilation function of the set of instants when it ticks in the subrun.

lemma `tick_set_sub`:
  assumes ⟨dilating f sub r⟩
  shows   ⟨{k. hamlet ((Rep_run r) k c)} = image f {k. hamlet ((Rep_run sub) k c)}⟩
    (**is** ⟨?R = image f ?S⟩)
⟨*proof*⟩

Strictly monotonous functions preserve the least element.

lemma `Least_strict_mono`:
  assumes ⟨strict_mono f⟩
  and   ⟨∃x ∈ S. ∀y ∈ S. x ≤ y⟩
  shows   ⟨(LEAST y. y ∈ f ' S) = f (LEAST x. x ∈ S)⟩
⟨*proof*⟩

A non empty set of `nat`s has a least element.

lemma `Least_nat_ex`:
  ⟨(n::nat) ∈ S ⟹ ∃x ∈ S. (∀y ∈ S. x ≤ y)⟩
⟨*proof*⟩

The first instant when a clock ticks in a dilated run is the image by the dilation function of the first instant when it ticks in the subrun.

lemma `Least_sub`:
  assumes ⟨dilating f sub r⟩
  and   ⟨∃k::nat. hamlet ((Rep_run sub) k c)⟩
  shows   ⟨(LEAST k. k ∈ {t. hamlet ((Rep_run r) t c)})
        = f (LEAST k. k ∈ {t. hamlet ((Rep_run sub) t c)})⟩
      (**is** ⟨(LEAST k. k ∈ ?R) = f (LEAST k. k ∈ ?S)⟩)
⟨*proof*⟩

If a clock ticks in a run, it ticks in the subrun.

lemma `ticks_imp_ticks_sub`:
  assumes ⟨dilating f sub r⟩
  and   ⟨∃k. hamlet ((Rep_run r) k c)⟩
  shows   ⟨∃k_0. hamlet ((Rep_run sub) k_0 c)⟩
⟨*proof*⟩

Stronger version: it ticks in the subrun and we know when.

lemma `ticks_imp_ticks_subk`:
  assumes ⟨dilating f sub r⟩
  and   ⟨hamlet ((Rep_run r) k c)⟩
  shows   ⟨∃k_0. f k_0 = k ∧ hamlet ((Rep_run sub) k_0 c)⟩
⟨*proof*⟩

A dilating function preserves the tick count on an interval for any clock.

lemma `dilated_ticks_strict`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. f m < i ∧ i < f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. m < i ∧ i < n ∧ hamlet ((Rep_run sub) i c)}`⟩
    (**is** ⟨`?RUN = image f ?SUB`⟩)
⟨*proof*⟩

lemma `dilated_ticks_left`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. f m ≤ i ∧ i < f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. m ≤ i ∧ i < n ∧ hamlet ((Rep_run sub) i c)}`⟩
    (**is** ⟨`?RUN = image f ?SUB`⟩)
⟨*proof*⟩

lemma `dilated_ticks_right`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. f m < i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. m < i ∧ i ≤ n ∧ hamlet ((Rep_run sub) i c)}`⟩
    (**is** ⟨`?RUN = image f ?SUB`⟩)
⟨*proof*⟩

lemma `dilated_ticks`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. f m ≤ i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. m ≤ i ∧ i ≤ n ∧ hamlet ((Rep_run sub) i c)}`⟩
    (**is** ⟨`?RUN = image f ?SUB`⟩)
⟨*proof*⟩

No tick can occur in a dilated run before the image of 0 by the dilation function.

lemma `empty_dilated_prefix`:
  assumes ⟨`dilating f sub r`⟩
  and   ⟨`n < f 0`⟩
shows   ⟨`¬ hamlet ((Rep_run r) n c)`⟩
⟨*proof*⟩

corollary `empty_dilated_prefix'`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. f 0 ≤ i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}`
        `= {i. i ≤ f n ∧ hamlet ((Rep_run r) i c)}`⟩
⟨*proof*⟩

corollary `dilated_prefix`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. i ≤ f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. i ≤ n ∧ hamlet ((Rep_run sub) i c)}`⟩
⟨*proof*⟩

corollary `dilated_strict_prefix`:
  assumes ⟨`dilating f sub r`⟩
  shows ⟨`{i. i < f n ∧ hamlet ((Rep_run r) i c)}`
        `= image f {i. i < n ∧ hamlet ((Rep_run sub) i c)}`⟩
⟨*proof*⟩

A singleton of `nat` can be defined with a weaker property.

lemma `nat_sing_prop`:
  ⟨`{i::nat. i = k ∧ P(i)} = {i::nat. i = k ∧ P(k)}`⟩
⟨*proof*⟩

The set definition and the function definition of `tick_count` are equivalent.

lemma `tick_count_is_fun[code]`:⟨`tick_count r c n = run_tick_count r c n`⟩
⟨*proof*⟩

To show that the set definition and the function definition of `tick_count_strict` are equivalent, we first show that the *strictness* of `tick_count_strict` can be softened using `Suc`.

lemma `tick_count_strict_suc`:⟨`tick_count_strict r c (Suc n) = tick_count r c n`⟩
  ⟨*proof*⟩

lemma `tick_count_strict_is_fun[code]`:
  ⟨`tick_count_strict r c n = run_tick_count_strictly r c n`⟩
⟨*proof*⟩

This leads to an alternate definition of the strict precedence relation.

lemma `strictly_precedes_alt_def1`:
  ⟨{ ϱ. $\forall$`n::nat. (run_tick_count` ϱ `K`$_2$ `n)` $\leq$ `(run_tick_count_strictly` ϱ `K`$_1$ `n) }`
 = { ϱ. $\forall$`n::nat. (run_tick_count_strictly` ϱ `K`$_2$ `(Suc n))`
                   $\leq$ `(run_tick_count_strictly` ϱ `K`$_1$ `n) }`⟩
⟨*proof*⟩

The strict precedence relation can even be defined using only `run_tick_count`:

lemma `zero_gt_all`:
  assumes ⟨`P (0::nat)`⟩
      and ⟨⋀`n. n > 0` ⟹ `P n`⟩
    shows ⟨`P n`⟩
  ⟨*proof*⟩

lemma `strictly_precedes_alt_def2`:
  ⟨{ ϱ. $\forall$`n::nat. (run_tick_count` ϱ `K`$_2$ `n)` $\leq$ `(run_tick_count_strictly` ϱ `K`$_1$ `n) }`
 = { ϱ. `(`¬`hamlet ((Rep_run` ϱ`) 0 K`$_2$`))`
      $\wedge$ `(`$\forall$`n::nat. (run_tick_count` ϱ `K`$_2$ `(Suc n))` $\leq$ `(run_tick_count` ϱ `K`$_1$ `n)) }`
  (is ⟨`?P = ?P'`⟩**)**
⟨*proof*⟩

Some properties of `run_tick_count`, `tick_count` and `Suc`:

lemma `run_tick_count_suc`:
  ⟨`run_tick_count r c (Suc n) = (if hamlet ((Rep_run r) (Suc n) c)`
                                 `then Suc (run_tick_count r c n)`
                                 `else run_tick_count r c n)`⟩
⟨*proof*⟩

corollary `tick_count_suc`:
  ⟨`tick_count r c (Suc n) = (if hamlet ((Rep_run r) (Suc n) c)`
                             `then Suc (tick_count r c n)`
                             `else tick_count r c n)`⟩
⟨*proof*⟩

Some generic properties on the cardinal of sets of nat that we will need later.

lemma `card_suc`:
  ⟨`card {i. i` $\leq$ `(Suc n)` $\wedge$ `P i} = card {i. i` $\leq$ `n` $\wedge$ `P i} + card {i. i = (Suc n)` $\wedge$ `P i}`⟩
⟨*proof*⟩

lemma `card_le_leq`:
  assumes ⟨`m < n`⟩
    shows ⟨`card {i::nat. m < i` $\wedge$ `i` $\leq$ `n` $\wedge$ `P i}`

```
                 = card {i. m < i ∧ i < n ∧ P i} + card {i. i = n ∧ P i}⟩
⟨proof⟩

lemma card_le_leq_0:
  ⟨card {i::nat. i ≤ n ∧ P i} = card {i. i < n ∧ P i} + card {i. i = n ∧ P i}⟩
⟨proof⟩

lemma card_mnm:
  assumes ⟨m < n⟩
    shows ⟨card {i::nat. i < n ∧ P i}
         = card {i. i ≤ m ∧ P i} + card {i. m < i ∧ i < n ∧ P i}⟩
⟨proof⟩

lemma card_mnm':
  assumes ⟨m < n⟩
    shows ⟨card {i::nat. i < n ∧ P i}
         = card {i. i < m ∧ P i} + card {i. m ≤ i ∧ i < n ∧ P i}⟩
⟨proof⟩

lemma nat_interval_union:
  assumes ⟨m ≤ n⟩
    shows ⟨{i::nat. i ≤ n ∧ P i}
         = {i::nat. i ≤ m ∧ P i} ∪ {i::nat. m < i ∧ i ≤ n ∧ P i}⟩
⟨proof⟩

lemma card_sing_prop:⟨card {i. i = n ∧ P i} = (if P n then 1 else 0)⟩
⟨proof⟩

lemma card_prop_mono:
  assumes ⟨m ≤ n⟩
    shows ⟨card {i::nat. i ≤ m ∧ P i} ≤ card {i. i ≤ n ∧ P i}⟩
⟨proof⟩
```

In a dilated run, no tick occurs strictly between two successive instants that are the images by f of instants of the original run.

```
lemma no_tick_before_suc:
  assumes ⟨dilating f sub r⟩
      and ⟨(f n) < k ∧ k < (f (Suc n))⟩
    shows ⟨¬hamlet ((Rep_run r) k c)⟩
⟨proof⟩
```

From this, we show that the number of ticks on any clock at f (Suc n) depends only on the number of ticks on this clock at f n and whether this clock ticks at f (Suc n). All the instants in between are stuttering instants.

```
lemma tick_count_fsuc:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count r c (f (Suc n))
         = tick_count r c (f n) + card {k. k = f (Suc n) ∧ hamlet ((Rep_run r) k c)}⟩
⟨proof⟩

corollary tick_count_f_suc:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count r c (f (Suc n))
         = tick_count r c (f n) + (if hamlet ((Rep_run r) (f (Suc n)) c) then 1 else 0)⟩
⟨proof⟩

corollary tick_count_f_suc_suc:
```

  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count r c (f (Suc n)) = (if hamlet ((Rep_run r) (f (Suc n)) c)
                                 then Suc (tick_count r c (f n))
                                 else tick_count r c (f n))⟩
⟨*proof*⟩

lemma `tick_count_f_suc_sub`:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count r c (f (Suc n)) = (if hamlet ((Rep_run sub) (Suc n) c)
                                 then Suc (tick_count r c (f n))
                                 else tick_count r c (f n))⟩
⟨*proof*⟩

The number of ticks does not progress during stuttering instants.

lemma `tick_count_latest`:
  assumes ⟨dilating f sub r⟩
    and ⟨f n_p < n ∧ (∀k. f n_p < k ∧ k ≤ n ⟶ (∄k_0. f k_0 = k))⟩
    shows ⟨tick_count r c n = tick_count r c (f n_p)⟩
⟨*proof*⟩

We finally show that the number of ticks on any clock is preserved by dilation.

lemma `tick_count_sub`:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count sub c n = tick_count r c (f n)⟩
⟨*proof*⟩

corollary `run_tick_count_sub`:
  assumes ⟨dilating f sub r⟩
    shows ⟨run_tick_count sub c n = run_tick_count r c (f n)⟩
⟨*proof*⟩

The number of ticks occurring strictly before the first instant is null.

lemma `tick_count_strict_0`:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count_strict r c (f 0) = 0⟩
⟨*proof*⟩

The number of ticks strictly before an instant does not progress during stuttering instants.

lemma `tick_count_strict_stable`:
  assumes ⟨dilating f sub r⟩
  assumes ⟨(f n) < k ∧ k < (f (Suc n))⟩
  shows ⟨tick_count_strict r c k = tick_count_strict r c (f (Suc n))⟩
⟨*proof*⟩

Finally, the number of ticks strictly before an instant is preserved by dilation.

lemma `tick_count_strict_sub`:
  assumes ⟨dilating f sub r⟩
    shows ⟨tick_count_strict sub c n = tick_count_strict r c (f n)⟩
⟨*proof*⟩

The tick count on any clock can only increase.

lemma `mono_tick_count`:
  ⟨mono (λ k. tick_count r c k)⟩
⟨*proof*⟩

In a dilated run, for any stuttering instant, there is an instant which is the image of an instant in the original run, and which is the latest one before the stuttering instant.

lemma greatest_prev_image:
  assumes ⟨dilating f sub r⟩
    shows ⟨($\nexists$ n_0. f n_0 = n) $\Longrightarrow$ ($\exists$ n_p. f n_p < n $\land$ ($\forall$k. f n_p < k $\land$ k $\leq$ n $\longrightarrow$ ($\nexists$ k_0. f k_0 = k)))⟩
⟨*proof*⟩

If a strictly monotonous function on `nat` increases only by one, its argument was increased only by one.

lemma strict_mono_suc:
  assumes ⟨strict_mono f⟩
    and ⟨f sn = Suc (f n)⟩
    shows ⟨sn = Suc n⟩
⟨*proof*⟩

Two successive non stuttering instants of a dilated run are the images of two successive instants of the original run.

lemma next_non_stuttering:
  assumes ⟨dilating f sub r⟩
    and ⟨f n_p < n $\land$ ($\forall$k. f n_p < k $\land$ k $\leq$ n $\longrightarrow$ ($\nexists$ k_0. f k_0 = k))⟩
    and ⟨f sn_0 = Suc n⟩
    shows ⟨sn_0 = Suc n_p⟩
⟨*proof*⟩

The order relation between tick counts on clocks is preserved by dilation.

lemma dil_tick_count:
  assumes ⟨sub $\ll$ r⟩
    and ⟨$\forall$n. run_tick_count sub a n $\leq$ run_tick_count sub b n⟩
    shows ⟨run_tick_count r a n $\leq$ run_tick_count r b n⟩
⟨*proof*⟩

Time does not progress during stuttering instants.

lemma stutter_no_time:
  assumes ⟨dilating f sub r⟩
    and ⟨$\bigwedge$k. f n < k $\land$ k $\leq$ m $\Longrightarrow$ ($\nexists$ k_0. f k_0 = k)⟩
    and ⟨m > f n⟩
    shows ⟨time ((Rep_run r) m c) = time ((Rep_run r) (f n) c)⟩
⟨*proof*⟩

lemma time_stuttering:
  assumes ⟨dilating f sub r⟩
    and ⟨time ((Rep_run sub) n c) = τ⟩
    and ⟨$\bigwedge$k. f n < k $\land$ k $\leq$ m $\Longrightarrow$ ($\nexists$ k_0. f k_0 = k)⟩
    and ⟨m > f n⟩
    shows ⟨time ((Rep_run r) m c) = τ⟩
⟨*proof*⟩

The first instant at which a given date is reached on a clock is preserved by dilation.

lemma first_time_image:
  assumes ⟨dilating f sub r⟩
    shows ⟨first_time sub c n t = first_time r c (f n) t⟩
⟨*proof*⟩

The first instant of a dilated run is necessarily the image of the first instant of the original run.

lemma first_dilated_instant:

    assumes ⟨strict_mono f⟩
       and ⟨f (0::nat) = (0::nat)⟩
    shows ⟨Max {i. f i ≤ 0} = 0⟩
⟨*proof*⟩

For any instant n of a dilated run, let n_0 be the last instant before n that is the image of an original instant. All instants strictly after n_0 and before n are stuttering instants.

lemma not_image_stut:
    assumes ⟨dilating f sub r⟩
       and ⟨n_0 = Max {i. f i ≤ n}⟩
       and ⟨f n_0 < k ∧ k ≤ n⟩
    shows ⟨∄k_0. f k_0 = k⟩
⟨*proof*⟩

For any dilating function f, `dil_inverse f` is a contracting function.

lemma contracting_inverse:
    assumes ⟨dilating f sub r⟩
    shows ⟨contracting (dil_inverse f) r sub f⟩
⟨*proof*⟩

The only possible contracting function toward a dense run (a run with no empty instants) is the inverse of the dilating function as defined by `dil_inverse`.

lemma dense_run_dil_inverse_only:
    assumes ⟨dilating f sub r⟩
       and ⟨contracting g r sub f⟩
       and ⟨dense_run sub⟩
    shows ⟨g = (dil_inverse f)⟩
⟨*proof*⟩

end

8.1.5   Main Theorems

theory Stuttering
imports StutteringLemmas

begin

Using the lemmas of the previous section about the invariance by stuttering of various properties of TESL specifications, we can now prove that the atomic formulae that compose TESL specifications are invariant by stuttering.

Sporadic specifications are preserved in a dilated run.

lemma sporadic_sub:
    assumes ⟨sub ≪ r⟩
       and ⟨sub ∈ ⟦c sporadic τ on c'⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦c sporadic τ on c'⟧$_{TESL}$⟩
⟨*proof*⟩

Implications are preserved in a dilated run.

theorem implies_sub:
    assumes ⟨sub ≪ r⟩
       and ⟨sub ∈ ⟦c_1 implies c_2⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦c_1 implies c_2⟧$_{TESL}$⟩
⟨*proof*⟩

theorem `implies_not_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦c₁ implies not c₂⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦c₁ implies not c₂⟧$_{TESL}$⟩
⟨*proof*⟩

Precedence relations are preserved in a dilated run.

theorem `weakly_precedes_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦c₁ weakly precedes c₂⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦c₁ weakly precedes c₂⟧$_{TESL}$⟩
⟨*proof*⟩

theorem `strictly_precedes_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦c₁ strictly precedes c₂⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦c₁ strictly precedes c₂⟧$_{TESL}$⟩
⟨*proof*⟩

Time delayed relations are preserved in a dilated run.

theorem `time_delayed_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦ a time-delayed by $\delta\tau$ on ms implies b ⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦ a time-delayed by $\delta\tau$ on ms implies b ⟧$_{TESL}$⟩
⟨*proof*⟩

Time relations are preserved through dilation of a run.

lemma `tagrel_sub'`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦ time-relation ⌊c₁,c₂⌋ ∈ R ⟧$_{TESL}$⟩
    shows ⟨R (time ((Rep_run r) n c₁), time ((Rep_run r) n c₂))⟩
⟨*proof*⟩

corollary `tagrel_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦ time-relation ⌊c₁,c₂⌋ ∈ R ⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦ time-relation ⌊c₁,c₂⌋ ∈ R ⟧$_{TESL}$⟩
⟨*proof*⟩

Time relations are also preserved by contraction

lemma `tagrel_sub_inv`:
  assumes ⟨sub ≪ r⟩
      and ⟨r ∈ ⟦ time-relation ⌊c₁, c₂⌋ ∈ R ⟧$_{TESL}$⟩
    shows ⟨sub ∈ ⟦ time-relation ⌊c₁, c₂⌋ ∈ R ⟧$_{TESL}$⟩
⟨*proof*⟩

Kill relations are preserved in a dilated run.

theorem `kill_sub`:
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦ c₁ kills c₂ ⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦ c₁ kills c₂ ⟧$_{TESL}$⟩
⟨*proof*⟩

lemmas `atomic_sub_lemmas` = `sporadic_sub tagrel_sub implies_sub implies_not_sub`
                      `time_delayed_sub weakly_precedes_sub`

```
strictly_precedes_sub kill_sub
```

We can now prove that all atomic specification formulae are preserved by the dilation of runs.

lemma `atomic_sub:`
  assumes ⟨sub ≪ r⟩
      and ⟨sub ∈ ⟦ φ ⟧$_{TESL}$⟩
    shows ⟨r ∈ ⟦ φ ⟧$_{TESL}$⟩
⟨*proof*⟩

Finally, any TESL specification is invariant by stuttering.

theorem `TESL_stuttering_invariant:`
  assumes ⟨sub ≪ r⟩
    shows ⟨sub ∈ ⟦⟦ S ⟧⟧$_{TESL}$ $\Longrightarrow$ r ∈ ⟦⟦ S ⟧⟧$_{TESL}$⟩
⟨*proof*⟩

end

Bibliography

[1] F. Boulanger, C. Jacquet, C. Hardebolle, and I. Prodan. TESL: a language for reconciling heterogeneous execution traces. In *Twelfth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2014)*, pages 114–123, Lausanne, Switzerland, Oct 2014.

[2] H. Nguyen Van, T. Balabonski, F. Boulanger, C. Keller, B. Valiron, and B. 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*, volume 10419 of *LNCS*. Springer, Sep 2017.