A Formal Development of a Polychronous Polytimed Coordination Language

Hai Nguyen Van
hai.nguyenvan.phie@gmail.com

Frédéric Boulanger
frederic.boulanger@centralesupelec.fr

Burkhart Wolff
burkhart.wolff@lri.fr

March 17, 2025

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).

```
consts dummyInfty    :: ‹'a ⇒ 'a›
consts dummyTESLSTAR :: ‹'a›
consts dummyFUN      :: ‹'a set ⇒ 'b set ⇒ 'c set›
consts dummyCLOCK    :: ‹'a set›
consts dummyBOOL     :: ‹bool set›
```

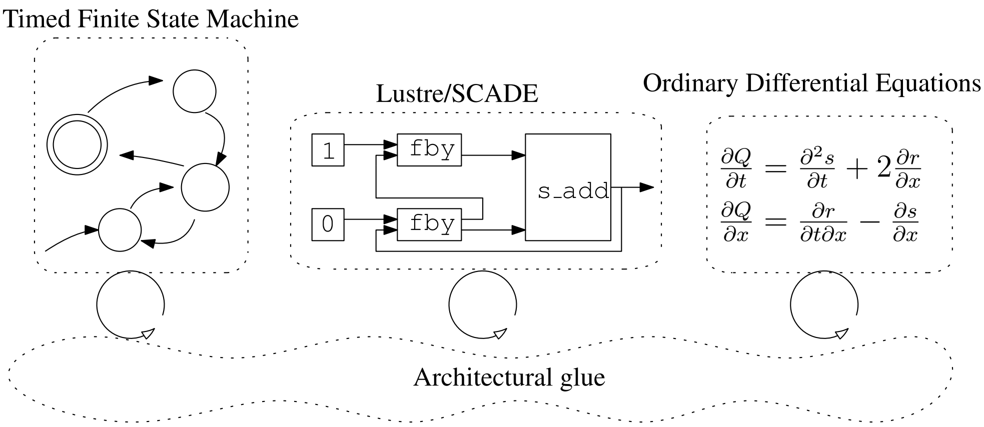

Figure 1.1: A Heterogeneous Timed System Model

```
consts dummyTIMES    :: ‹'a set›
consts dummyLEQ      :: ‹'a ⇒ 'a ⇒ bool›

notation dummyInfty    (‹(_∞)› [1000] 999)
notation dummyTESLSTAR (‹TESL*›)
notation dummyFUN      (infixl ‹→› 100)
notation dummyCLOCK    (‹𝒦›)
notation dummyBOOL     (‹𝔹›)
notation dummyTIMES    (‹𝒯›)
notation dummyLEQ      (infixl ‹≤𝒯› 100)
```

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

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`.
- `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)         (‹τcst›)
```

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 t1) (τ_cst t2) = τ_cst (divide t1 t2)›

  fun uminus_tag_const
    where ‹uminus (τ_cst t) = τ_cst (uminus t)›

fun minus_tag_const
  where ‹minus (τ_cst t1) (τ_cst t2) = τ_cst (minus t1 t2)›

definition ‹one_tag_const ≡ τ_cst 1›

fun times_tag_const
  where ‹times (τ_cst t1) (τ_cst t2) = τ_cst (times t1 t2)›

definition ‹zero_tag_const ≡ τ_cst 0›

fun plus_tag_const
  where ‹plus (τ_cst t1) (τ_cst t2) = τ_cst (plus t1 t2)›

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
  — K₁ sporadic τ on K₂ means that K₁ should tick at an instant where the time on K₂ is τ.
    ‹⟦ K₁ sporadic τ on K₂ ⟧TESL =
        {ϱ. ∃n::nat. hamlet ((Rep_run ϱ) n K₁) ∧ time ((Rep_run ϱ) n K₂) = τ}›
  — time-relation ⌊K₁, K₂⌋ ∈ R means that at each instant, the time on K₁ and the time on K₂ are in relation R.
  | ‹⟦ 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 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 δτ on measuring implies slave means that at each instant at which master ticks,
    slave will tick after a delay δτ measured on the time scale of measuring.
  | ‹⟦ master time-delayed by δτ on measuring implies slave ⟧TESL =
    — When master ticks, let's call t₀ the current date on measuring. Then, at the first instant when the date on
      measuring is t₀ + 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 + δτ)
```

```
                          ⟶ 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 = ⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ)›
⟨proof⟩
```

3.2.2 Expansion law

Similar to the expansion laws of lattices.

```
theorem TESL_interp_homo_append:
  ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₁ ⟧⟧TESL ∩ ⟦⟦ Φ₂ ⟧⟧TESL›
⟨proof⟩
```

3.3 Equational laws for the denotation of TESL formulae

```
lemma TESL_interp_assoc:
  ‹⟦⟦ (Φ₁ @ Φ₂) @ Φ₃ ⟧⟧TESL = ⟦⟦ Φ₁ @ (Φ₂ @ Φ₃) ⟧⟧TESL›
⟨proof⟩

lemma TESL_interp_commute:
  shows ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₂ @ Φ₁ ⟧⟧TESL›
⟨proof⟩
```

```
lemma TESL_interp_left_commute:
  ‹⟦⟦ Φ₁ @ (Φ₂ @ Φ₃) ⟧⟧TESL = ⟦⟦ Φ₂ @ (Φ₁ @ Φ₃) ⟧⟧TESL›
⟨proof⟩
```

```
lemma TESL_interp_idem:
  ‹⟦⟦ Φ @ Φ ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL›
⟨proof⟩
```

```
lemma TESL_interp_left_idem:
  ‹⟦⟦ Φ₁ @ (Φ₁ @ Φ₂) ⟧⟧TESL = ⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL›
⟨proof⟩
```

```
lemma TESL_interp_right_idem:
  ‹⟦⟦ (Φ₁ @ Φ₂) @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₁ @ Φ₂ ⟧⟧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 ⊇ ⟦⟦ φ # Φ ⟧⟧TESL›
⟨proof⟩
```

```
lemma TESL_sem_decreases_tail:
  ‹⟦⟦ Φ ⟧⟧TESL ⊇ ⟦⟦ 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 ‹$[\![\ \Phi\ ]\!]_{TESL}$ = $[\![\ \Phi'\ ]\!]_{TESL}$›
⟨*proof*⟩

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

theorem TESL_interp_decreases_setinc:
assumes ‹set $\Phi \subseteq$ set Φ'›
shows ‹$[\![\ \Phi\ ]\!]_{TESL} \supseteq [\![\ \Phi'\ ]\!]_{TESL}$›
⟨*proof*⟩

lemma TESL_interp_decreases_add_head:
assumes ‹set $\Phi \subseteq$ set Φ'›
shows ‹$[\![\ \varphi$ # $\Phi\ ]\!]_{TESL} \supseteq [\![\ \varphi$ # $\Phi'\ ]\!]_{TESL}$›
⟨*proof*⟩

lemma TESL_interp_decreases_add_tail:
assumes ‹set $\Phi \subseteq$ set Φ'›
shows ‹$[\![\ \Phi$ @ [φ] $]\!]_{TESL} \supseteq [\![\ \Phi'$ @ [φ] $]\!]_{TESL}$›
⟨*proof*⟩

lemma TESL_interp_absorb1:
assumes ‹set $\Phi_1 \subseteq$ set Φ_2›
shows ‹$[\![\ \Phi_1$ @ $\Phi_2\ ]\!]_{TESL}$ = $[\![\ \Phi_2\ ]\!]_{TESL}$›
⟨*proof*⟩

lemma TESL_interp_absorb2:
assumes ‹set $\Phi_2 \subseteq$ set Φ_1›
shows ‹$[\![\ \Phi_1$ @ $\Phi_2\ ]\!]_{TESL}$ = $[\![\ \Phi_1\ ]\!]_{TESL}$›
⟨*proof*⟩

3.5 Some special cases

lemma NoSporadic_stable [simp]:
‹$[\![\ \Phi\ ]\!]_{TESL} \subseteq [\![$ NoSporadic $\Phi\ ]\!]_{TESL}$›
⟨*proof*⟩

lemma NoSporadic_idem [simp]:
‹$[\![\ \Phi\ ]\!]_{TESL} \cap [\![$ NoSporadic $\Phi\ ]\!]_{TESL}$ = $[\![\ \Phi\ ]\!]_{TESL}$›
⟨*proof*⟩

lemma NoSporadic_setinc:
‹set (NoSporadic Φ) $\subseteq$ 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›  (‹#≤ ›)
```

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› (‹τvar›)

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 }›
| ‹⟦ ⌊τvar(K1, n1), τvar(K2, n2)⌋ ∈ R ⟧prim =
    { ϱ. R (time ((Rep_run ϱ) n1 K1), time ((Rep_run ϱ) n2 K2)) }›
```

```
| ‹⟦ ⌈e₁, e₂⌉ ∈ R ⟧prim = { ϱ. R (⟦ ϱ ⊢ e₁ ⟧cntexpr, ⟦ ϱ ⊢ e₂ ⟧cntexpr) }›
| ‹⟦ cnt_e₁ ≼ cnt_e₂ ⟧prim = { ϱ. ⟦ ϱ ⊢ cnt_e₁ ⟧cntexpr ≤ ⟦ ϱ ⊢ cnt_e₂ ⟧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, τcst 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:
  ‹$[\![[$ (Γ_1 @ Γ_2) @ $\Gamma_3\ ]\!]]_{prim}$ = $[\![[\ \Gamma_1$ @ (Γ_2 @ Γ_3) $]\!]]_{prim}$›
⟨*proof*⟩

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

lemma symrun_interp_left_commute:
  ‹$[\![[\ \Gamma_1$ @ (Γ_2 @ Γ_3) $]\!]]_{prim}$ = $[\![[\ \Gamma_2$ @ (Γ_1 @ Γ_3) $]\!]]_{prim}$›
⟨*proof*⟩

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

lemma symrun_interp_left_idem:
  ‹$[\![[\ \Gamma_1$ @ (Γ_1 @ Γ_2) $]\!]]_{prim}$ = $[\![[\ \Gamma_1$ @ $\Gamma_2\ ]\!]]_{prim}$›
⟨*proof*⟩

lemma symrun_interp_right_idem:
  ‹$[\![[$ (Γ_1 @ Γ_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:
  ‹$[\![\Gamma]\!]_{prim} \supseteq [\![\Gamma$ @ [γ] $]\!]_{prim}$›
⟨*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 ‹$[\![\Gamma$ @ [γ] $]\!]_{prim} \supseteq [\![\Gamma'$ @ [γ] $]\!]_{prim}$›
⟨*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 ‹$[\![\ \Gamma_1$ @ $\Gamma_2\ ]\!]_{prim}$ = $[\![\ \Gamma_1\ ]\!]_{prim}$›
⟨*proof*⟩

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 ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ)
     ↪e  (Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # 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 ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)›
| tagrel_e:
— A relation between time scales has to be obeyed at every instant.
  ‹(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:
— 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 ⊢ ((K1 implies K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # Φ))›
| implies_e2:
— It can also be handled in the present by making both the master and the slave clocks tick.
  ‹(Γ, n ⊢ ((K1 implies K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 ⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # 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 ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # 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 ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))›
| timedelayed_e1:
— A timed delayed implication can be handled by forbidding a tick on the master clock.
  ‹(Γ, n ⊢ ((K1 time-delayed by δτ on K2 implies K3) # Ψ) ▷ Φ)
     ↪e  (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 time-delayed by δτ on K2 implies K3) # 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 ⊢ ((K1 time-delayed by δτ on K2 implies K3) # Ψ) ▷ Φ)
     ↪e  (((K1 ⇑ n) # (K2 @ n ⊕ δτ ⇒ K3) # Γ), n
           ⊢ Ψ ▷ ((K1 time-delayed by δτ on K2 implies K3) # Φ))›
| weakly_precedes_e:
— A weak precedence relation has to hold at every instant.
  ‹(Γ, n ⊢ ((K1 weakly precedes K2) # Ψ) ▷ Φ)
     ↪e  ((([#≤ K2 n, #≤ K1 n] ∈ (λ(x,y). x≤y)) # Γ), n
           ⊢ Ψ ▷ ((K1 weakly precedes K2) # 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 ‹𝒞₁ ↪ⁿ 𝒞₂›
  assumes ‹𝒞₂ ↪ᵐ 𝒞₃›
    shows ‹𝒞₁ ↪ⁿ ⁺ ᵐ 𝒞₃›
⟨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› (‹𝒞next _›)
where
  ‹𝒞next 𝒮 ≡ { 𝒮'. 𝒮 ↪ 𝒮' }›
```

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

```
lemma Cnext_solve_instant:
  ‹(𝒞next (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:
  ‹(𝒞next (Γ, n ⊢ ((K₁ sporadic τ on K₂) # Ψ) ▷ Φ))
    ⊇ { Γ, n ⊢ Ψ ▷ ((K₁ sporadic τ on K₂) # Φ),
        ((K₁ ⇑ n) # (K₂ ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ }›
⟨proof⟩

lemma Cnext_solve_tagrel:
  ‹(𝒞next (Γ, n ⊢ ((time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ) ▷ Φ))
    ⊇ { ((⌊τvar(K₁, n), τvar(K₂, n)⌋ ∈ R) # Γ),n
          ⊢ Ψ ▷ ((time-relation ⌊K₁, K₂⌋ ∈ R) # Φ) }›
⟨proof⟩

lemma Cnext_solve_implies:
  ‹(𝒞next (Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ))
    ⊇ { ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ),
         ((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) }›
⟨proof⟩

lemma Cnext_solve_implies_not:
  ‹(𝒞next (Γ, n ⊢ ((K₁ implies not K₂) # Ψ) ▷ Φ))
    ⊇ { ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ),
        ((K₁ ⇑ n) # (K₂ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ) }›
⟨proof⟩

lemma Cnext_solve_timedelayed:
  ‹(𝒞next (Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ))
    ⊇ { ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ),
        ((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
          ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) }›
⟨proof⟩

lemma Cnext_solve_weakly_precedes:
  ‹(𝒞next (Γ, n ⊢ ((K₁ weakly precedes K₂) # Ψ) ▷ Φ))
    ⊇ { ((⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
```

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

lemma `Cnext_solve_strictly_precedes:`
```
  ‹(𝒞next (Γ, n ⊢ ((K₁ strictly precedes K₂) # Ψ) ▷ Φ))
    ⊇ { ((⌈#≤ K₂ n, #< K₁ n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
         ⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ) }›
```
⟨*proof*⟩

lemma `Cnext_solve_kills:`
```
  ‹(𝒞next (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 δτ on measuring implies slave ⟧TESL
    = ⋂ {Y. ∃n::nat.
          Y = ⟦ master time-delayed by δτ 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 ‹⟦ φ ⟧_TESL = ⋂ {Y. ∃n::nat. Y = ⟦ φ ⟧_TESL^≥ n}›
⟨proof⟩
```

Some useful lemmas for reasoning on properties of sequences.

```
lemma forall_nat_expansion:
  ‹(∀n ≥ (n0::nat). P n) = (P n0 ∧ (∀n ≥ Suc n0. P n))›
⟨proof⟩

lemma exists_nat_expansion:
  ‹(∃n ≥ (n0::nat). P n) = (P n0 ∨ (∃n ≥ Suc n0. P n))›
⟨proof⟩

lemma forall_nat_set_suc:‹{x. ∀m ≥ n. P x m} = {x. P x n} ∩ {x. ∀m ≥ Suc n. P x m}›
⟨proof⟩

lemma exists_nat_set_suc:‹{x. ∃m ≥ n. P x m} = {x. P x n} ∪ {x. ∃m ≥ Suc n. P x m}›
⟨proof⟩
```

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:
  ‹⟦ K1 sporadic τ on K2 ⟧_TESL^≥ n =
    ⟦ K1 ⇑ n ⟧_prim ∩ ⟦ K2 ⇓ n @ τ ⟧_prim              — rule sporadic_on_e2
    ∪ ⟦ K1 sporadic τ on K2 ⟧_TESL^≥ Suc n›      — rule sporadic_on_e1
⟨proof⟩


lemma TESL_interp_stepwise_tagrel_coind_unfold:
  ‹⟦ time-relation ⌊K1, K2⌋ ∈ R ⟧_TESL^≥ n =            — rule tagrel_e
     ⟦ ⌊τ_var(K1, n), τ_var(K2, n)⌋ ∈ R ⟧_prim
     ∩ ⟦ time-relation ⌊K1, K2⌋ ∈ R ⟧_TESL^≥ Suc n›
⟨proof⟩

lemma TESL_interp_stepwise_implies_coind_unfold:
  ‹⟦ master implies slave ⟧_TESL^≥ n =
     (   ⟦ master ¬⇑ n ⟧_prim                          — rule implies_e1
       ∪ ⟦ master ⇑ n ⟧_prim ∩ ⟦ slave ⇑ n ⟧_prim)   — rule implies_e2
     ∩ ⟦ master implies slave ⟧_TESL^≥ Suc n›
⟨proof⟩

lemma TESL_interp_stepwise_implies_not_coind_unfold:
  ‹⟦ master implies not slave ⟧_TESL^≥ n =
     (    ⟦ master ¬⇑ n ⟧_prim                            — rule implies_not_e1
        ∪ ⟦ master ⇑ n ⟧_prim ∩ ⟦ slave ¬⇑ n ⟧_prim)   — rule implies_not_e2
     ∩ ⟦ master implies not slave ⟧_TESL^≥ Suc n›
⟨proof⟩

lemma TESL_interp_stepwise_timedelayed_coind_unfold:
  ‹⟦ master time-delayed by δτ on measuring implies slave ⟧_TESL^≥ n =
     (     ⟦ master ¬⇑ n ⟧_prim                    — rule timedelayed_e1
        ∪ (⟦ master ⇑ n ⟧_prim ∩ ⟦ measuring @ n ⊕ δτ ⇒ slave ⟧_prim))
                                                 — rule timedelayed_e2
```

```
    ∩ ⟦ master time-delayed by δτ on measuring implies slave ⟧TESL≥ Suc n›
```
⟨proof⟩

lemma `TESL_interp_stepwise_weakly_precedes_coind_unfold:`
```
  ‹⟦ K₁ weakly precedes K₂ ⟧TESL≥ n =                    — rule weakly_precedes_e
     ⟦ (⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x,y). x≤y)) ⟧prim
     ∩ ⟦ K₁ weakly precedes K₂ ⟧TESL≥ Suc n›
```
⟨proof⟩

lemma `TESL_interp_stepwise_strictly_precedes_coind_unfold:`
```
  ‹⟦ K₁ strictly precedes K₂ ⟧TESL≥ n =                  — rule strictly_precedes_e
     ⟦ (⌈#≤ K₂ n, #< K₁ n⌉ ∈ (λ(x,y). x≤y)) ⟧prim
     ∩ ⟦ K₁ strictly precedes K₂ ⟧TESL≥ Suc n›
```
⟨proof⟩

lemma `TESL_interp_stepwise_kills_coind_unfold:`
```
  ‹⟦ K₁ kills K₂ ⟧TESL≥ n =
     (   ⟦ K₁ ¬⇑ n ⟧prim                                 — rule kills_e1
       ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ¬⇑ ≥ n ⟧prim)             — rule kills_e2
     ∩ ⟦ K₁ kills K₂ ⟧TESL≥ 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 ⇒ nat ⇒ 'τ run set›
  (‹⟦⟦ _ ⟧⟧TESL≥ -›)
```
where
```
  ‹⟦⟦ [] ⟧⟧TESL≥ n = {ϱ. True}›
| ‹⟦⟦ φ # Φ ⟧⟧TESL≥ n = ⟦ φ ⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ n›
```

lemma `TESL_interpretation_stepwise_fixpoint:`
```
  ‹⟦⟦ Φ ⟧⟧TESL≥ n = ⋂ ((λφ. ⟦ φ ⟧TESL≥ 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≥ 0›
```
⟨proof⟩

lemma `TESL_interpretation_stepwise_zero':`
```
  ‹⟦⟦ Φ ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL≥ 0›
```
⟨proof⟩

lemma `TESL_interpretation_stepwise_cons_morph:`
```
  ‹⟦ φ ⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ n = ⟦⟦ φ # Φ ⟧⟧TESL≥ n›
```
⟨proof⟩

theorem `TESL_interp_stepwise_composition:`
```
  shows ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL≥ n = ⟦⟦ Φ₁ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ₂ ⟧⟧TESL≥ 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
  ::‹'τ::linordered_field config ⇒ 'τ run set›        (‹⟦ _ ⟧config› 71)
where
  ‹⟦ Γ, n ⊢ Ψ ▷ Φ ⟧config = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›

lemma HeronConf_interp_composition:
   ‹⟦ Γ1, n ⊢ Ψ1 ▷ Φ1 ⟧config ∩ ⟦ Γ2, n ⊢ Ψ2 ▷ Φ2 ⟧config
     = ⟦ (Γ1 @ Γ2), n ⊢ (Ψ1 @ Ψ2) ▷ (Φ1 @ Φ2) ⟧config›
  ⟨proof⟩
```

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:
   ‹⟦ Γ, n ⊢ [] ▷ Φ ⟧config = ⟦ Γ, Suc n ⊢ Φ ▷ [] ⟧config›
⟨proof⟩
```

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:
   ‹⟦ Γ, n ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ ⟧config
    = ⟦ Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ) ⟧config
    ∪ ⟦ ((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ ⟧config›
⟨proof⟩

lemma HeronConf_interp_stepwise_tagrel_cases:
   ‹⟦ Γ, n ⊢ ((time-relation ⌊K1, K2⌋ ∈ R) # Ψ) ▷ Φ ⟧config
    = ⟦ ((⌊τvar(K1, n), τvar(K2, n)⌋ ∈ R) # Γ), n
        ⊢ Ψ ▷ ((time-relation ⌊K1, K2⌋ ∈ R) # Φ) ⟧config›
⟨proof⟩

lemma HeronConf_interp_stepwise_implies_cases:
   ‹⟦ Γ, n ⊢ ((K1 implies K2) # Ψ) ▷ Φ ⟧config
      = ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # Φ) ⟧config
      ∪ ⟦ ((K1 ⇑ n) # (K2 ⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies K2) # Φ) ⟧config›
⟨proof⟩

lemma HeronConf_interp_stepwise_implies_not_cases:
   ‹⟦ Γ, n ⊢ ((K1 implies not K2) # Ψ) ▷ Φ ⟧config
      = ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config
      ∪ ⟦ ((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config›
⟨proof⟩

lemma HeronConf_interp_stepwise_timedelayed_cases:
```

```
‹⟦ Γ, n ⊢ ((K1 time-delayed by δτ on K2 implies K3) # Ψ) ▹ Φ ⟧config
  = ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 time-delayed by δτ on K2 implies K3) # Φ) ⟧config
  ∪ ⟦ ((K1 ⇑ n) # (K2 @ n ⊕ δτ ⇒ K3) # Γ), n
      ⊢ Ψ ▹ ((K1 time-delayed by δτ on K2 implies K3) # Φ) ⟧config›
```

⟨*proof*⟩

lemma `HeronConf_interp_stepwise_weakly_precedes_cases:`

```
‹⟦ Γ, n ⊢ ((K1 weakly precedes K2) # Ψ) ▹ Φ ⟧config
  = ⟦ ((⌈#≤ K2 n, #≤ 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
  = ⟦ ((⌈#≤ 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 ‹$[\![[\![\Psi]\!]]\!]_{TESL}$ = $[\![$ [], 0 ⊢ Ψ ▷ [] $]\!]_{config}$›
  ⟨*proof*⟩

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 ‹(Γ_1, n_1 ⊢ Ψ_1 ▷ Φ_1) ↪ (Γ_2, n_2 ⊢ Ψ_2 ▷ Φ_2)›
  shows ‹$[\![[\![\Gamma_1]\!]]\!]_{prim} \cap [\![[\![\Psi_1]\!]]\!]_{TESL}{}^{\geq\ n_1} \cap [\![[\![\Phi_1]\!]]\!]_{TESL}{}^{\geq\ Suc\ n_1}$
     $\supseteq [\![[\![\Gamma_2]\!]]\!]_{prim} \cap [\![[\![\Psi_2]\!]]\!]_{TESL}{}^{\geq\ n_2} \cap [\![[\![\Phi_2]\!]]\!]_{TESL}{}^{\geq\ Suc\ n_2}$› **(is ?P)**
⟨*proof*⟩

inductive_cases step_elim:‹$\mathcal{S}_1$ ↪ $\mathcal{S}_2$›

lemma sound_reduction':
  assumes ‹$\mathcal{S}_1$ ↪ $\mathcal{S}_2$›
  shows ‹$[\![\mathcal{S}_1]\!]_{config} \supseteq [\![\mathcal{S}_2]\!]_{config}$›
⟨*proof*⟩

lemma sound_reduction_generalized:
  assumes ‹$\mathcal{S}_1 \hookrightarrow^{\mathtt{k}} \mathcal{S}_2$›
    shows ‹$[\![\ \mathcal{S}_1\ ]\!]_{config} \supseteq [\![\ \mathcal{S}_2\ ]\!]_{config}$›
⟨*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 ‹$(\mathtt{[]},\ 0 \vdash \Psi \triangleright \mathtt{[]}) \hookrightarrow^{\mathtt{k}} \mathcal{S}$›
    shows ‹$[\![\![\ \Psi\ ]\!]\!]_{TESL} \supseteq [\![\ \mathcal{S}\ ]\!]_{config}$›
  ⟨*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 ‹$[\![\ \Gamma,\ \mathtt{n} \vdash \Psi \triangleright \Phi\ ]\!]_{config} \subseteq (\bigcup \mathtt{X} \in \mathcal{C}_{next}\ (\Gamma,\ \mathtt{n} \vdash \Psi \triangleright \Phi).\ [\![\ \mathtt{X}\ ]\!]_{config})$›
  ⟨*proof*⟩

lemma complete_direct_successors':
  shows ‹$[\![\ \mathcal{S}\ ]\!]_{config} \subseteq (\bigcup \mathtt{X} \in \mathcal{C}_{next}\ \mathcal{S}.\ [\![\ \mathtt{X}\ ]\!]_{config})$›
⟨*proof*⟩

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

lemma branch_existence:
  assumes ‹$\varrho \in [\![\ \mathcal{S}_1\ ]\!]_{config}$›
  shows ‹$\exists \mathcal{S}_2.\ (\mathcal{S}_1 \hookrightarrow \mathcal{S}_2) \wedge (\varrho \in [\![\ \mathcal{S}_2\ ]\!]_{config})$›
⟨*proof*⟩

lemma branch_existence':
  assumes ‹$\varrho \in [\![\ \mathcal{S}_1\ ]\!]_{config}$›
  shows ‹$\exists \mathcal{S}_2.\ (\mathcal{S}_1 \hookrightarrow^{\mathtt{k}} \mathcal{S}_2) \wedge (\varrho \in [\![\ \mathcal{S}_2\ ]\!]_{config})$›
⟨*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 ‹$\varrho \in [\![\![\ \Psi\ ]\!]\!]_{TESL}$›
  shows ‹$\exists \mathcal{S}.\ ((\mathtt{[]},\ 0 \vdash \Psi \triangleright \mathtt{[]}) \hookrightarrow^{\mathtt{k}} \mathcal{S})$
      $\wedge\ \varrho \in [\![\ \mathcal{S}\ ]\!]_{config}$›
  ⟨*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   ‹∃Γ_k Ψ_k Φ_k k. ((Γ, n ⊢ Ψ ▷ Φ)  ↪^k  (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))
                          ∧ ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config›
⟨proof⟩

lemma instant_index_increase_generalized:
  assumes ‹n < n_k›
  assumes ‹ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ Φ ⟧_config›
  shows   ‹∃Γ_k Ψ_k Φ_k k. ((Γ, n ⊢ Ψ ▷ Φ) ↪^k (Γ_k, n_k ⊢ Ψ_k ▷ Φ_k))
                          ∧ ϱ ∈ ⟦ Γ_k, n_k ⊢ Ψ_k ▷ Φ_k ⟧_config›
⟨proof⟩
```

Any run that belongs to a specification Ψ has a corresponding configuration that develops it up to the $\mathtt{n}^{\mathrm{th}}$ instant.

```
theorem progress:
  assumes ‹ϱ ∈ ⟦⟦ Ψ ⟧⟧_TESL›
    shows ‹∃k Γ_k Ψ_k Φ_k. (([], 0 ⊢ Ψ ▷ [])  ↪^k (Γ_k, n ⊢ Ψ_k ▷ Φ_k))
                          ∧ ϱ ∈ ⟦ Γ_k, n ⊢ Ψ_k ▷ Φ_k ⟧_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› (‹μ_config›)
where
  ‹μ_config (Γ, n ⊢ Ψ ▷ Φ) = μ Ψ›
```

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

```
lemma elimation_rules_strictly_decreasing:
  assumes ‹(Γ_1, n_1 ⊢ Ψ_1 ▷ Φ_1)  ↪_e  (Γ_2, n_2 ⊢ Ψ_2 ▷ Φ_2)›
    shows ‹μ Ψ_1 > μ Ψ_2›
⟨proof⟩

lemma elimation_rules_strictly_decreasing_meas:
  assumes ‹(Γ_1, n_1 ⊢ Ψ_1 ▷ Φ_1)  ↪_e  (Γ_2, n_2 ⊢ Ψ_2 ▷ Φ_2)›
    shows ‹(Ψ_2, Ψ_1) ∈ measure μ›
⟨proof⟩

lemma elimation_rules_strictly_decreasing_meas':
  assumes ‹𝒮_1  ↪_e  𝒮_2›
  shows ‹(𝒮_2, 𝒮_1) ∈ measure μ_config›
⟨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(𝒮₁::'a::linordered_field config) 𝒮₂. (𝒮₁ ↪ₑ← 𝒮₂))›
⟨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 ⇒ nat) (r::'a::linordered_field run)
    ≡ strict_mono f ∧ (f 0 = 0) ∧ (∀n. f n ≥ n)
   ∧ ((∄n₀. f n₀ = n) ⟶ (∀c. ¬(hamlet ((Rep_run r) n c))))
```

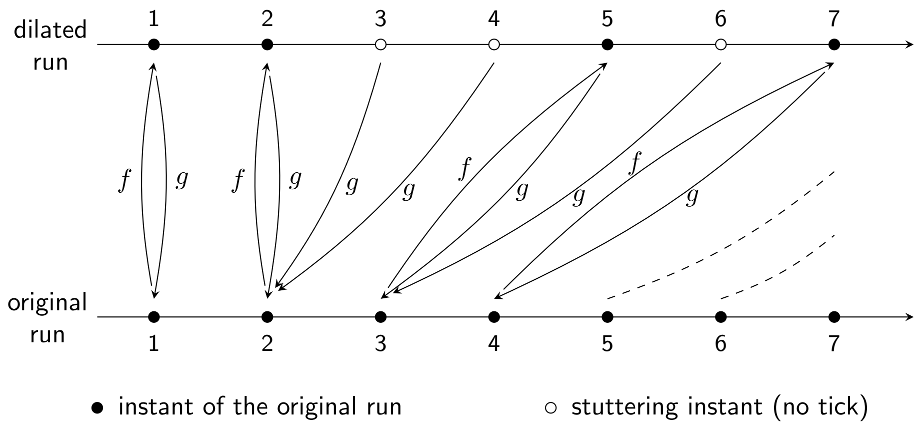

Figure 8.1: Dilating and contracting functions

```
  ∧ ((∄n₀. f n₀ = (Suc n)) ⟶ (∀c. time ((Rep_run r) (Suc n) c)
                                  = time ((Rep_run r) n c)))
)›
```

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`:

```
consts dummyf    :: ‹nat ⇒ nat›
```

```
consts dummyg    :: ‹nat ⇒ nat›
consts dummytwo  :: ‹nat›
notation dummyf    (‹f›)
notation dummyg    (‹g›)
notation dummytwo  (‹2›)
```

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   ‹∃n0. f n0 = n›
⟨proof⟩

lemma ticks_image_sub:
  assumes ‹dilating f sub r›
  and     ‹hamlet ((Rep_run r) n c)›
  shows   ‹∃n0. f n0 = n›
⟨proof⟩

lemma ticks_image_sub':
  assumes ‹dilating f sub r›
  and     ‹∃c. hamlet ((Rep_run r) n c)›
  shows   ‹∃n0. f n0 = 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   ‹(∄n₀. f n₀ = 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   ‹∃k₀. hamlet ((Rep_run sub) k₀ 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   ‹∃k₀. f k₀ = k ∧ hamlet ((Rep_run sub) k₀ 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. ∀n::nat. (run_tick_count ϱ K₂ n) ≤ (run_tick_count_strictly ϱ K₁ n) }
 = { ϱ. ∀n::nat. (run_tick_count_strictly ϱ K₂ (Suc n))
                 ≤ (run_tick_count_strictly ϱ K₁ 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. ∀n::nat. (run_tick_count ϱ K₂ n) ≤ (run_tick_count_strictly ϱ K₁ n) }
 = { ϱ. (¬hamlet ((Rep_run ϱ) 0 K₂))
      ∧ (∀n::nat. (run_tick_count ϱ K₂ (Suc n)) ≤ (run_tick_count ϱ K₁ 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 ≤ (Suc n) ∧ P i} = card {i. i ≤ n ∧ P i} + card {i. i = (Suc n) ∧ P i}›
```
⟨proof⟩

lemma `card_le_leq:`
```
  assumes ‹m < n›
    shows ‹card {i::nat. m < i ∧ i ≤ n ∧ 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 ‹(∄n0. f n0 = n) ⟹ (∃np. f np < n ∧ (∀k. f np < k ∧ k ≤ n ⟶ (∄k0. f k0 = 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 np < n ∧ (∀k. f np < k ∧ k ≤ n ⟶ (∄k0. f k0 = k))›
      and ‹f sn0 = Suc n›
    shows ‹sn0 = Suc np›
```
⟨proof⟩

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

```
lemma dil_tick_count:
  assumes ‹sub ≪ r›
      and ‹∀n. run_tick_count sub a n ≤ run_tick_count sub b n›
    shows ‹run_tick_count r a n ≤ run_tick_count r b n›
```
⟨proof⟩

Time does not progress during stuttering instants.

```
lemma stutter_no_time:
  assumes ‹dilating f sub r›
      and ‹⋀k. f n < k ∧ k ≤ m ⟹ (∄k0. f k0 = 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 ‹⋀k. f n < k ∧ k ≤ m ⟹ (∄k0. f k0 = 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 ‹n0 = Max {i. f i ≤ n}›
      and ‹f n0 < k ∧ k ≤ n›
    shows ‹∄k0. f k0 = 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 ∈ ⟦c₁ implies c₂⟧TESL›
    shows ‹r ∈ ⟦c₁ implies c₂⟧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 δτ on ms implies b ⟧TESL›
    shows ‹r ∈ ⟦ a time-delayed by δτ 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 ⟹ r ∈ ⟦⟦ S ⟧⟧TESL›
⟨proof⟩

end
theory Config_Morphisms
  imports Hygge_Theory
begin
```

TESL morphisms change the time on clocks, preserving the ticks.

```
consts morphism :: ‹'a ⇒ ('τ::linorder ⇒ 'τ::linorder) ⇒ 'a› (infixl ‹⨂› 100)
```

Applying a TESL morphism to a tag simply changes its value.

```
overloading morphism_tagconst ≡ ‹morphism :: 'τ tag_const ⇒ ('τ::linorder ⇒ 'τ) ⇒ 'τ tag_const›

begin
  definition morphism_tagconst :
          ‹(x::'τ tag_const) ⨂ (f::('τ::linorder ⇒ 'τ)) = (τcst o f)(the_tag_const x) ›
end
```

Applying a TESL morphism to an atomic formula only changes the dates.

```
overloading morphism_TESL_atomic ≡
            ‹morphism :: 'τ TESL_atomic ⇒ ('τ::linorder ⇒ 'τ) ⇒ 'τ TESL_atomic›
begin
definition morphism_TESL_atomic :
          ‹(Ψ::'τ TESL_atomic) ⨂ (f::('τ::linorder ⇒ 'τ)) =
              (case Ψ of
                (C sporadic t on C')      ⇒ (C sporadic (t⨂f) on C')
              | (time-relation ⌊C, C'⌋∈R)⇒ (time-relation ⌊C, C'⌋ ∈ (λ(t, t'). R(t⨂f,t'⨂f)))
              | (C implies C')            ⇒ (C implies C')
              | (C implies not C')        ⇒ (C implies not C')
              | (C time-delayed by t on C' implies C'')
                                          ⇒ (C time-delayed by t⨂f on C' implies C'')
              | (C weakly precedes C')    ⇒ (C weakly precedes C')
              | (C strictly precedes C') ⇒ (C strictly precedes C')
              | (C kills C')              ⇒ (C kills C'))›
```

```
end
```

Applying a TESL morphism to a formula amounts to apply it to each atomic formula.

```
overloading morphism_TESL_formula ≡
          ‹morphism :: 'τ TESL_formula ⇒ ('τ::linorder ⇒ 'τ) ⇒ 'τ TESL_formula›
begin
definition  morphism_TESL_formula :
         ‹(Ψ::'τ TESL_formula) ⨂ (f::('τ::linorder ⇒ 'τ)) = map (λx. x ⨂ f) Ψ›
end
```

Applying a TESL morphism to a configuration amounts to apply it to the present and future formulae. The past (in the context Γ) is not changed.

```
overloading morphism_TESL_config ≡
          ‹morphism :: ('τ::linordered_field) config ⇒ ('τ ⇒ 'τ) ⇒ 'τ config›
begin
fun  morphism_TESL_config
  where  ‹((Γ, n ⊢ Ψ ▷ Φ)::('τ::linordered_field) config) ⨂ (f::('τ ⇒ 'τ)) =
         (Γ, n ⊢ (Ψ⨂f) ▷ (Φ⨂f))›
end
```

A TESL formula is called consistent if it possesses Kripke-models in its denotational interpretation.

```
definition consistent :: ‹('τ::linordered_field) TESL_formula ⇒ bool›
  where  ‹consistent Ψ ≡ ⟦⟦ Ψ ⟧⟧TESL ≠ {}›
```

If we can derive a consistent finite context from a TESL formula, the formula is consistent.

```
theorem consistency_finite :
  assumes start          : ‹([], 0 ⊢ Ψ ▷ []) ↪** (Γ₁, n₁ ⊢ [] ▷ [])›
      and init_invariant  : ‹consistent_context Γ₁›
    shows ‹consistent Ψ›
⟨proof⟩
```

Snippets on runs

A run with no ticks and constant time for all clocks.

```
definition const_nontick_run :: ‹(clock ⇒ 'τ tag_const) ⇒ ('τ::linordered_field) run › (‹□_› 80)
  where ‹□f ≡  Abs_run(λn c. (False, f c))›
```

Ensure a clock ticks in a run at a given instant.

```
definition set_tick :: ‹('τ::linordered_field) run ⇒ nat ⇒ clock ⇒ ('τ) run›
  where  ‹set_tick r k c = Abs_run(λn c.  if k = n
                                     then (True , time(Rep_run r k c))
                                     else Rep_run r k c) ›
```

Ensure a clock does not tick in a run at a given instant.

```
definition unset_tick :: ‹('τ::linordered_field) run ⇒ nat ⇒ clock ⇒ ('τ) run›
  where  ‹unset_tick r k c = Abs_run(λn c.  if k = n
                                     then (False , time(Rep_run r k c))
                                     else Rep_run r k c) ›
```

Replace all instants after k in a run with the instants from another run. Warning: the result may not be a proper run since time may not be monotonous from instant k to instant k+1.

```
definition patch :: ‹('τ::linordered_field) run ⇒ nat ⇒ 'τ run ⇒ 'τ run› (‹_ ≫- _› 80)
```

```
  where   ‹r ≫ᵏr' ≡ Abs_run(λn c. if n ≤ k then Rep_run (r) n c else  Rep_run (r') n c)›
```

For some infinite cases, the idea for a proof scheme looks as follows: if we can derive from the initial configuration `[], 0 ⊢ Ψ ▷ []` a start-point of a lasso Γ_1, $n_1 \vdash \Psi_1 \triangleright \Phi_1$, and if we can traverse the lasso one time Γ_1, $n_1 \vdash \Psi_1 \triangleright \Phi_1 \hookrightarrow^{++} \Gamma_2$, $n_2 \vdash \Psi_2 \triangleright \Phi_2$ to isomorphic one, we can always always make a derivation along the lasso. If the entry point of the lasso had traces with prefixes consistent to Γ_1, then there exist traces consisting of this prefix and repetitions of the delta-prefix of the lasso which are consistent with Ψ which implies the logical consistency of Ψ.

So far the idea. Remains to prove it. Why does one symbolic run along a lasso generalises to arbitrary runs ?

```
theorem consistency_coinduct :
  assumes start             : ‹([], 0 ⊢ Ψ ▷ [])   ↪** (Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁)›
      and loop              : ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ↪++ (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)›
      and init_invariant    : ‹consistent_context Γ₁›
      and post_invariant    : ‹consistent_context Γ₂›
      and retract_condition : ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) ⨂ (f::'τ ⇒ 'τ) = (Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) ›
    shows ‹consistent (Ψ :: ('τ :: linordered_field)TESL_formula)›
⟨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.