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›
```

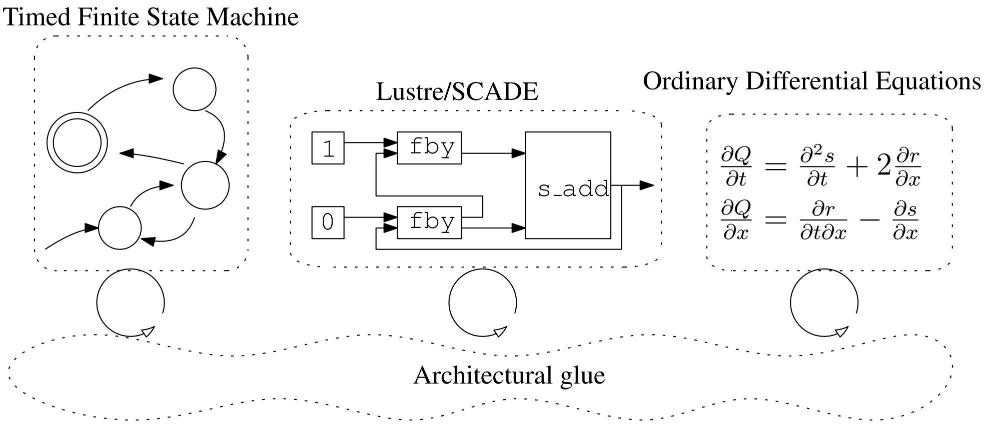

Figure 1.1: A Heterogeneous Timed System Model

```
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: $\Sigma^\infty = \mathbb{N} \rightarrow \mathcal{K} \rightarrow (\mathbb{B} \times \mathcal{T})$, 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
```

Multiplication is associative.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
                               and c::‹'τ::field tag_const›
  obtain u v w where ‹a = τ_cst u› and ‹b = τ_cst v› and ‹c = τ_cst w›
    using tag_const.exhaust by metis
```

```
  thus ‹a * b * c = a * (b * c)›
    by (simp add: TESL.times_tag_const.simps)
next
```

Multiplication is commutative.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
  obtain u v where ‹a = τcst u› and ‹b = τcst v› using tag_const.exhaust by metis
  thus ‹ a * b = b * a›
    by (simp add: TESL.times_tag_const.simps)
next
```

One is neutral for multiplication.

```
  fix a::‹'τ::field tag_const›
  obtain u where ‹a = τcst u› using tag_const.exhaust by blast
  thus ‹1 * a = a›
    by (simp add: TESL.times_tag_const.simps one_tag_const_def)
next
```

Addition is associative.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
                               and c::‹'τ::field tag_const›
  obtain u v w where ‹a = τcst u› and ‹b = τcst v› and ‹c = τcst w›
    using tag_const.exhaust by metis
  thus ‹a + b + c = a + (b + c)›
    by (simp add: TESL.plus_tag_const.simps)
next
```

Addition is commutative.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
  obtain u v where ‹a = τcst u› and ‹b = τcst v› using tag_const.exhaust by metis
  thus ‹a + b = b + a›
    by (simp add: TESL.plus_tag_const.simps)
next
```

Zero is neutral for addition.

```
  fix a::‹'τ::field tag_const›
  obtain u where ‹a = τcst u› using tag_const.exhaust by blast
  thus ‹0 + a = a›
    by (simp add: TESL.plus_tag_const.simps zero_tag_const_def)
next
```

The sum of an element and its opposite is zero.

```
  fix a::‹'τ::field tag_const›
  obtain u where ‹a = τcst u› using tag_const.exhaust by blast
  thus ‹-a + a = 0›
    by (simp add: TESL.plus_tag_const.simps
                  TESL.uminus_tag_const.simps
                  zero_tag_const_def)
next
```

Subtraction is adding the opposite.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
  obtain u v where ‹a = τcst u› and ‹b = τcst v› using tag_const.exhaust by metis
  thus ‹a - b = a + -b›
    by (simp add: TESL.minus_tag_const.simps
```

```
                 TESL.plus_tag_const.simps
                 TESL.uminus_tag_const.simps)
next
```

Distributive property of multiplication over addition.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
                               and c::‹'τ::field tag_const›
  obtain u v w where ‹a = τcst u› and ‹b = τcst v› and ‹c = τcst w›
    using tag_const.exhaust by metis
  thus ‹(a + b) * c = a * c + b * c›
    by (simp add: TESL.plus_tag_const.simps
                 TESL.times_tag_const.simps
                 ring_class.ring_distribs(2))
next
```

The neutral elements are distinct.

```
  show ‹(0::('τ::field tag_const)) ≠ 1›
    by (simp add: one_tag_const_def zero_tag_const_def)
next
```

The product of an element and its inverse is 1.

```
  fix a::‹'τ::field tag_const› assume h:‹a ≠ 0›
  obtain u where ‹a = τcst u› using tag_const.exhaust by blast
  moreover with h have ‹u ≠ 0› by (simp add: zero_tag_const_def)
  ultimately show ‹inverse a * a = 1›
    by (simp add: TESL.inverse_tag_const.simps
                 TESL.times_tag_const.simps
                 one_tag_const_def)
next
```

Dividing is multiplying by the inverse.

```
  fix a::‹'τ::field tag_const› and b::‹'τ::field tag_const›
  obtain u v where ‹a = τcst u› and ‹b = τcst v› using tag_const.exhaust by metis
  thus ‹a div b = a * inverse b›
    by (simp add: TESL.divide_tag_const.simps
                 TESL.inverse_tag_const.simps
                 TESL.times_tag_const.simps
                 divide_inverse)
next
```

Zero is its own inverse.

```
  show ‹inverse (0::('τ::field tag_const)) = 0›
    by (simp add: TESL.inverse_tag_const.simps zero_tag_const_def)
qed

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
    show ‹⋀x y :: 'a tag_const. (x < y) = (x ≤ y ∧ ¬ y ≤ x)›
      using less_eq_tag_const.simps less_tag by auto
  next
    fix x::‹'a tag_const›
    from tag_const.exhaust obtain x₀::'a where ‹x = TConst x₀› by blast
    with Int_less_eq show ‹x ≤ x› by simp
  next
    show ‹⋀x y z  :: 'a tag_const. x ≤ y ⟹ y ≤ z ⟹ x ≤ z›
      using less_eq_tag_const.simps by auto
  next
    show ‹⋀x y  :: 'a tag_const. x ≤ y ⟹ y ≤ x ⟹ x = y›
      using less_eq_tag_const.simps by auto
  qed

end
```

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

```
instantiation tag_const :: (linorder)linorder
begin
  instance proof
    fix x::‹'a tag_const› and y::‹'a tag_const›
    from tag_const.exhaust obtain x₀::'a where ‹x = TConst x₀› by blast
    moreover from tag_const.exhaust obtain y₀::'a where ‹y = TConst y₀› by blast
    ultimately show ‹x ≤ y ∨ y ≤ x› using less_eq_tag_const.simps by fastforce
  qed

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
  show ‹(λ_ _. (True, τcst 0)) ∈ {ϱ. ∀c. mono (λn. time (ϱ n c))}›
    unfolding mono_def by blast
qed
```

```
lemma Abs_run_inverse_rewrite:
  ‹∀c. mono (λn. time (ϱ n c)) ⟹ Rep_run (Abs_run ϱ) = ϱ›
by (simp add: Abs_run_inverse)
```

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 -
  have ‹mono (λn. time (Rep_run ϱ n K))› using Rep_run by blast
  moreover from assms(2) have ‹m ≤ n› using less_imp_le by simp
  moreover have ‹mono (λn. time (Rep_run ϱ n K))› using Rep_run by blast
  ultimately have  ‹time ((Rep_run ϱ) m K) ≤ time ((Rep_run ϱ) n K)›
    by (simp add:mono_def)
  moreover from assms(1) have ‹time ((Rep_run ϱ) n K) = τ›
    using first_time_def by blast
  moreover from assms have ‹time ((Rep_run ϱ) m K) ≠ τ›
    using first_time_def by blast
  ultimately show ?thesis by simp
qed
```

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 -
  from assms(1) have ‹∀m < n. time ((Rep_run ϱ) m K) ≠ τ›
    by (simp add: less_le)
  with assms(2) show ?thesis by (simp add: first_time_def)
qed

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›
by simp
```

3.2.1 Image interpretation lemma

```
theorem TESL_interpretation_image:
  ‹⟦⟦ Φ ⟧⟧TESL = ⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ)›
by (induction Φ, simp+)
```

3.2.2 Expansion law

Similar to the expansion laws of lattices.

```
theorem TESL_interp_homo_append:
  ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₁ ⟧⟧TESL ∩ ⟦⟦ Φ₂ ⟧⟧TESL›
by (induction Φ₁, simp, auto)
```

3.3 Equational laws for the denotation of TESL formulae

```
lemma TESL_interp_assoc:
  ‹⟦⟦ (Φ₁ @ Φ₂) @ Φ₃ ⟧⟧TESL = ⟦⟦ Φ₁ @ (Φ₂ @ Φ₃) ⟧⟧TESL›
by auto

lemma TESL_interp_commute:
  shows ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₂ @ Φ₁ ⟧⟧TESL›
by (simp add: TESL_interp_homo_append inf_sup_aci(1))
```

```
lemma TESL_interp_left_commute:
  ‹⟦⟦ Φ₁ @ (Φ₂ @ Φ₃) ⟧⟧TESL = ⟦⟦ Φ₂ @ (Φ₁ @ Φ₃) ⟧⟧TESL›
unfolding TESL_interp_homo_append by auto

lemma TESL_interp_idem:
  ‹⟦⟦ Φ @ Φ ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL›
using TESL_interp_homo_append by auto

lemma TESL_interp_left_idem:
  ‹⟦⟦ Φ₁ @ (Φ₁ @ Φ₂) ⟧⟧TESL = ⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL›
using TESL_interp_homo_append by auto

lemma TESL_interp_right_idem:
  ‹⟦⟦ (Φ₁ @ Φ₂) @ Φ₂ ⟧⟧TESL = ⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL›
unfolding TESL_interp_homo_append by auto

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›
by simp

lemma TESL_interp_neutral2:
  ‹⟦⟦ Φ @ [] ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL›
by simp
```

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›
by simp

lemma TESL_sem_decreases_tail:
  ‹⟦⟦ Φ ⟧⟧TESL ⊇ ⟦⟦ Φ @ [φ] ⟧⟧TESL›
by (simp add: TESL_interp_homo_append)
```

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 -
  have ‹φ # Φ = [φ] @ Φ› by simp
  hence ‹⟦⟦ φ # Φ ⟧⟧TESL = ⟦⟦ [φ] ⟧⟧TESL ∩ ⟦⟦ Φ ⟧⟧TESL›
    using TESL_interp_homo_append by simp
  thus ?thesis using assms TESL_interpretation_image by fastforce
qed
```

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

```
lemma TESL_interp_remdups_absorb:
  ‹⟦⟦ Φ ⟧⟧TESL = ⟦⟦ remdups Φ ⟧⟧TESL›
```

```
proof (induction Φ)
  case Cons
    thus ?case using TESL_interp_formula_stuttering by auto
qed simp
```

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 -
  have ‹set (remdups Φ) = set (remdups Φ')›
    by (simp add: assms)
  moreover have fxpntΦ: ‹⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ) = ⟦⟦ Φ ⟧⟧TESL›
    by (simp add: TESL_interpretation_image)
  moreover have fxpntΦ': ‹⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ') = ⟦⟦ Φ' ⟧⟧TESL›
    by (simp add: TESL_interpretation_image)
  moreover have ‹⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ) = ⋂ ((λφ. ⟦ φ ⟧TESL) ` set Φ')›
    by (simp add: assms)
  ultimately show ?thesis using TESL_interp_remdups_absorb by auto
qed
```

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 -
  obtain Φr where decompose: ‹set (Φ @ Φr) = set Φ'› using assms by auto
  hence ‹set (Φ @ Φr) = set Φ'› using assms by blast
  moreover have ‹(set Φ) ∪ (set Φr) = set Φ'›
    using assms decompose by auto
  moreover have ‹⟦⟦ Φ' ⟧⟧TESL = ⟦⟦ Φ @ Φr ⟧⟧TESL›
    using TESL_interp_set_lifting decompose by blast
  moreover have ‹⟦⟦ Φ @ Φr ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL ∩ ⟦⟦ Φr ⟧⟧TESL›
    by (simp add: TESL_interp_homo_append)
  moreover have ‹⟦⟦ Φ ⟧⟧TESL ⊇ ⟦⟦ Φ ⟧⟧TESL ∩ ⟦⟦ Φr ⟧⟧TESL› by simp
  ultimately show ?thesis by simp
qed

lemma TESL_interp_decreases_add_head:
  assumes ‹set Φ ⊆ set Φ'›
    shows ‹⟦⟦ φ # Φ ⟧⟧TESL ⊇ ⟦⟦ φ # Φ' ⟧⟧TESL›
using assms TESL_interp_decreases_setinc by auto

lemma TESL_interp_decreases_add_tail:
  assumes ‹set Φ ⊆ set Φ'›
    shows ‹⟦⟦ Φ @ [φ] ⟧⟧TESL ⊇ ⟦⟦ Φ' @ [φ] ⟧⟧TESL›
using TESL_interp_decreases_setinc[OF assms]
  by (simp add: TESL_interpretation_image dual_order.trans)

lemma TESL_interp_absorb1:
  assumes ‹set Φ1 ⊆ set Φ2›
    shows ‹⟦⟦ Φ1 @ Φ2 ⟧⟧TESL = ⟦⟦ Φ2 ⟧⟧TESL›
by (simp add: Int_absorb1 TESL_interp_decreases_setinc
                          TESL_interp_homo_append assms)

lemma TESL_interp_absorb2:
  assumes ‹set Φ2 ⊆ set Φ1›
    shows ‹⟦⟦ Φ1 @ Φ2 ⟧⟧TESL = ⟦⟦ Φ1 ⟧⟧TESL›
```

```
using TESL_interp_absorb1 TESL_interp_commute assms by blast
```

3.5 Some special cases

```
lemma NoSporadic_stable [simp]:
  ‹⟦⟦ Φ ⟧⟧TESL ⊆ ⟦⟦ NoSporadic Φ ⟧⟧TESL›
proof -
  from filter_is_subset have ‹set (NoSporadic Φ) ⊆ set Φ› .
  from TESL_interp_decreases_setinc[OF this] show ?thesis .
qed

lemma NoSporadic_idem [simp]:
  ‹⟦⟦ Φ ⟧⟧TESL ∩ ⟦⟦ NoSporadic Φ ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL›
using NoSporadic_stable by blast

lemma NoSporadic_setinc:
  ‹set (NoSporadic Φ) ⊆ set Φ›
by (rule filter_is_subset)

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› (‹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 }›
| ‹⟦ ⌊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›
by auto

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›
unfolding consistent_context_def by auto
```

— This is very restrictive

```
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:
  ‹⋂ ((λγ. ⟦ γ ⟧prim) ‘ set Γ) = ⟦⟦ Γ ⟧⟧prim›
by (induction Γ, simp+)
```

4.3.2 Expansion law

Similar to the expansion laws of lattices

```
theorem symrun_interp_expansion:
  ‹⟦⟦ Γ1 @ Γ2 ⟧⟧prim = ⟦⟦ Γ1 ⟧⟧prim ∩ ⟦⟦ Γ2 ⟧⟧prim›
by (induction Γ1, simp, auto)
```

4.4 Equations for the interpretation of symbolic primitives

4.4.1 General laws

```
lemma symrun_interp_assoc:
  ‹⟦⟦ (Γ1 @ Γ2) @ Γ3 ⟧⟧prim = ⟦⟦ Γ1 @ (Γ2 @ Γ3) ⟧⟧prim›
by auto

lemma symrun_interp_commute:
  ‹⟦⟦ Γ1 @ Γ2 ⟧⟧prim = ⟦⟦ Γ2 @ Γ1 ⟧⟧prim›
by (simp add: symrun_interp_expansion inf_sup_aci(1))

lemma symrun_interp_left_commute:
  ‹⟦⟦ Γ1 @ (Γ2 @ Γ3) ⟧⟧prim = ⟦⟦ Γ2 @ (Γ1 @ Γ3) ⟧⟧prim›
unfolding symrun_interp_expansion by auto

lemma symrun_interp_idem:
  ‹⟦⟦ Γ @ Γ ⟧⟧prim = ⟦⟦ Γ ⟧⟧prim›
using symrun_interp_expansion by auto

lemma symrun_interp_left_idem:
  ‹⟦⟦ Γ1 @ (Γ1 @ Γ2) ⟧⟧prim = ⟦⟦ Γ1 @ Γ2 ⟧⟧prim›
using symrun_interp_expansion by auto

lemma symrun_interp_right_idem:
  ‹⟦⟦ (Γ1 @ Γ2) @ Γ2 ⟧⟧prim = ⟦⟦ Γ1 @ Γ2 ⟧⟧prim›
unfolding symrun_interp_expansion by auto

lemmas symrun_interp_aci =  symrun_interp_commute
                            symrun_interp_assoc
                            symrun_interp_left_commute
                            symrun_interp_left_idem
```

— Identity element

```
lemma symrun_interp_neutral1:
  ‹⟦⟦ [] @ Γ ⟧⟧prim = ⟦⟦ Γ ⟧⟧prim›
by simp

lemma symrun_interp_neutral2:
  ‹⟦⟦ Γ @ [] ⟧⟧prim = ⟦⟦ Γ ⟧⟧prim›
```

```
by simp
```

4.4.2 Decreasing interpretation of symbolic primitives

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

```
lemma TESL_sem_decreases_head:
  ‹⟦⟦ Γ ⟧⟧prim ⊇ ⟦⟦ γ # Γ ⟧⟧prim›
by simp

lemma TESL_sem_decreases_tail:
  ‹⟦⟦ Γ ⟧⟧prim ⊇ ⟦⟦ Γ @ [γ] ⟧⟧prim›
by (simp add: symrun_interp_expansion)
```

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

```
lemma symrun_interp_formula_stuttering:
  assumes ‹γ ∈ set Γ›
    shows ‹⟦⟦ γ # Γ ⟧⟧prim = ⟦⟦ Γ ⟧⟧prim›
proof -
  have ‹γ # Γ = [γ] @ Γ› by simp
  hence ‹⟦⟦ γ # Γ ⟧⟧prim = ⟦⟦ [γ] ⟧⟧prim ∩ ⟦⟦ Γ ⟧⟧prim›
    using symrun_interp_expansion by simp
  thus ?thesis using assms symrun_interp_fixpoint by fastforce
qed
```

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

```
lemma symrun_interp_remdups_absorb:
  ‹⟦⟦ Γ ⟧⟧prim = ⟦⟦ remdups Γ ⟧⟧prim›
proof (induction Γ)
  case Cons
    thus ?case using symrun_interp_formula_stuttering by auto
qed simp
```

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 ‹⟦⟦ Γ ⟧⟧prim = ⟦⟦ Γ' ⟧⟧prim›
proof -
  have ‹set (remdups Γ) = set (remdups Γ')›
    by (simp add: assms)
  moreover have fxpntΓ: ‹⋂ ((λγ. ⟦ γ ⟧prim) ` set Γ) = ⟦⟦ Γ ⟧⟧prim›
    by (simp add: symrun_interp_fixpoint)
  moreover have fxpntΓ': ‹⋂ ((λγ. ⟦ γ ⟧prim) ` set Γ') = ⟦⟦ Γ' ⟧⟧prim›
    by (simp add: symrun_interp_fixpoint)
  moreover have ‹⋂ ((λγ. ⟦ γ ⟧prim) ` set Γ) = ⋂ ((λγ. ⟦ γ ⟧prim) ` set Γ')›
    by (simp add: assms)
  ultimately show ?thesis using symrun_interp_remdups_absorb by auto
qed
```

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

```
theorem symrun_interp_decreases_setinc:
  assumes ‹set Γ ⊆ set Γ'›
    shows ‹⟦⟦ Γ ⟧⟧prim ⊇ ⟦⟦ Γ' ⟧⟧prim›
proof -
```

```
  obtain Γ_r where decompose: ‹set (Γ @ Γ_r) = set Γ'› using assms by auto
  hence ‹set (Γ @ Γ_r) = set Γ'› using assms by blast
  moreover have ‹(set Γ) ∪ (set Γ_r) = set Γ'› using assms decompose by auto
  moreover have ‹⟦⟦ Γ' ⟧⟧_prim = ⟦⟦ Γ @ Γ_r ⟧⟧_prim›
    using symrun_interp_set_lifting decompose by blast
  moreover have ‹⟦⟦ Γ @ Γ_r ⟧⟧_prim = ⟦⟦ Γ ⟧⟧_prim ∩ ⟦⟦ Γ_r ⟧⟧_prim›
    by (simp add: symrun_interp_expansion)
  moreover have ‹⟦⟦ Γ ⟧⟧_prim ⊇ ⟦⟦ Γ ⟧⟧_prim ∩ ⟦⟦ Γ_r ⟧⟧_prim› by simp
  ultimately show ?thesis by simp
qed

lemma symrun_interp_decreases_add_head:
  assumes ‹set Γ ⊆ set Γ'›
    shows ‹⟦⟦ γ # Γ ⟧⟧_prim ⊇ ⟦⟦ γ # Γ' ⟧⟧_prim›
using symrun_interp_decreases_setinc assms by auto

lemma symrun_interp_decreases_add_tail:
  assumes ‹set Γ ⊆ set Γ'›
    shows ‹⟦⟦ Γ @ [γ] ⟧⟧_prim ⊇ ⟦⟦ Γ' @ [γ] ⟧⟧_prim›
proof -
  from symrun_interp_decreases_setinc[OF assms] have ‹⟦⟦ Γ' ⟧⟧_prim ⊆ ⟦⟦ Γ ⟧⟧_prim› .
  thus ?thesis by (simp add: symrun_interp_expansion dual_order.trans)
qed

lemma symrun_interp_absorb1:
  assumes ‹set Γ_1 ⊆ set Γ_2›
    shows ‹⟦⟦ Γ_1 @ Γ_2 ⟧⟧_prim = ⟦⟦ Γ_2 ⟧⟧_prim›
by (simp add: Int_absorb1 symrun_interp_decreases_setinc
                          symrun_interp_expansion assms)

lemma symrun_interp_absorb2:
  assumes ‹set Γ_2 ⊆ set Γ_1›
    shows ‹⟦⟦ Γ_1 @ Γ_2 ⟧⟧_prim = ⟦⟦ Γ_1 ⟧⟧_prim›
using symrun_interp_absorb1 symrun_interp_commute assms by blast

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›
using relcompp.relcompI[of  ‹operational_semantics_step ^^ n› _ _
                            ‹operational_semantics_step ^^ m›, OF assms]
by (simp add: relpowp_add)
```

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 ▷ [] }›
by (simp add: operational_semantics_step.simps operational_semantics_intro.instant_i)
```

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 }›
by (simp add: operational_semantics_step.simps
              operational_semantics_elim.sporadic_on_e1
              operational_semantics_elim.sporadic_on_e2)

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) }›
by (simp add: operational_semantics_step.simps operational_semantics_elim.tagrel_e)

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) }›
by (simp add: operational_semantics_step.simps operational_semantics_elim.implies_e1
              operational_semantics_elim.implies_e2)

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) }›
by (simp add: operational_semantics_step.simps
              operational_semantics_elim.implies_not_e1
              operational_semantics_elim.implies_not_e2)

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) }›
by (simp add: operational_semantics_step.simps
              operational_semantics_elim.timedelayed_e1
              operational_semantics_elim.timedelayed_e2)

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) }›
by (simp add: operational_semantics_step.simps
              operational_semantics_elim.weakly_precedes_e)

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) }›
by (simp add: operational_semantics_step.simps
              operational_semantics_elim.strictly_precedes_e)

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) }›
by (simp add: operational_semantics_step.simps operational_semantics_elim.kills_e1
              operational_semantics_elim.kills_e2)
```

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

```
lemma empty_spec_reductions:
  ‹([], 0 ⊢ [] ▷ []) ↪ᵏ ([], k ⊢ [] ▷ [])›
proof (induct k)
  case 0 thus ?case by simp
next
  case Suc thus ?case
    using instant_i operational_semantics_step.simps by fastforce
qed

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}›
by auto

lemma TESL_interp_unfold_stepwise_tagrelgen:
  ‹⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL≥ n}›
by auto

lemma TESL_interp_unfold_stepwise_implies:
  ‹⟦ master implies slave ⟧TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ master implies slave ⟧TESL≥ n}›
by auto

lemma TESL_interp_unfold_stepwise_implies_not:
  ‹⟦ master implies not slave ⟧TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ master implies not slave ⟧TESL≥ n}›
by auto

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}›
by auto

lemma TESL_interp_unfold_stepwise_weakly_precedes:
  ‹⟦ K₁ weakly precedes K₂ ⟧TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ K₁ weakly precedes K₂ ⟧TESL≥ n}›
by auto

lemma TESL_interp_unfold_stepwise_strictly_precedes:
  ‹⟦ K₁ strictly precedes K₂ ⟧TESL
    = ⋂ {Y. ∃n::nat. Y = ⟦ K₁ strictly precedes K₂ ⟧TESL≥ n}›
by auto

lemma TESL_interp_unfold_stepwise_kills:
  ‹⟦ master kills slave ⟧TESL = ⋂ {Y. ∃n::nat. Y = ⟦ master kills slave ⟧TESL≥ n}›
by auto
```

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 -
  from positive_atom.elims(2)[OF assms]
    obtain u v w where ‹φ = (u sporadic v on w)› by blast
  with TESL_interp_unfold_stepwise_sporadicon show ?thesis by simp
qed
```

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 (cases φ)
  case SporadicOn thus ?thesis using assms by simp
next
  case TagRelation
    thus ?thesis using TESL_interp_unfold_stepwise_tagrelgen by simp
next
  case Implies
    thus ?thesis using TESL_interp_unfold_stepwise_implies by simp
next
  case ImpliesNot
    thus ?thesis using TESL_interp_unfold_stepwise_implies_not by simp
next
  case TimeDelayedBy
    thus ?thesis using TESL_interp_unfold_stepwise_timedelayed by simp
next
  case WeaklyPrecedes
    thus ?thesis
      using TESL_interp_unfold_stepwise_weakly_precedes by simp
next
  case StrictlyPrecedes
    thus ?thesis
      using TESL_interp_unfold_stepwise_strictly_precedes by simp
next
  case Kills
    thus ?thesis
      using TESL_interp_unfold_stepwise_kills by simp
qed
```

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 -
  have ‹(∀n ≥ (n0::nat). P n) = (∀n. (n = n0 ∨ n > n0) ⟶ P n)›
    using le_less by blast
  also have ‹... = (P n0 ∧ (∀n > n0. P n))› by blast
  finally show ?thesis using Suc_le_eq by simp
qed

lemma exists_nat_expansion:
  ‹(∃n ≥ (n0::nat). P n) = (P n0 ∨ (∃n ≥ Suc n0. P n))›
proof -
  have ‹(∃n ≥ (n0::nat). P n) = (∃n. (n = n0 ∨ n > n0) ∧ P n)›
    using le_less by blast
  also have ‹... = (∃n. (P n0) ∨ (n > n0 ∧ P n))› by blast
  finally show ?thesis using Suc_le_eq by simp
qed

lemma forall_nat_set_suc:‹{x. ∀m ≥ n. P x m} = {x. P x n} ∩ {x. ∀m ≥ Suc n. P x m}›
proof
  { fix x assume h:‹x ∈ {x. ∀m ≥ n. P x m}›
    hence ‹P x n› by simp
    moreover from h have ‹x ∈ {x. ∀m ≥ Suc n. P x m}› by simp
    ultimately have ‹x ∈ {x. P x n} ∩ {x. ∀m ≥ Suc n. P x m}› by simp
  } thus ‹{x. ∀m ≥ n. P x m} ⊆ {x. P x n} ∩ {x. ∀m ≥ Suc n. P x m}› ..
```

```
next
  { fix x  assume h:‹x ∈ {x. P x n} ∩ {x. ∀m ≥ Suc n. P x m}›
    hence ‹P x n› by simp
    moreover from h have ‹∀m ≥ Suc n. P x m› by simp
    ultimately have ‹∀m ≥ n. P x m› using forall_nat_expansion by blast
    hence ‹x ∈ {x. ∀m ≥ n. P x m}› by simp
  } thus ‹{x. P x n} ∩ {x. ∀m ≥ Suc n. P x m} ⊆ {x. ∀m ≥ n. P x m}› ..
qed

lemma exists_nat_set_suc:‹{x. ∃m ≥ n. P x m} = {x. P x n} ∪ {x. ∃m ≥ Suc n. P x m}›
proof
  { fix x assume h:‹x ∈ {x. ∃m ≥ n. P x m}›
    hence ‹x ∈ {x. ∃m. (m = n ∨ m ≥ Suc n) ∧ P x m}›
      using Suc_le_eq antisym_conv2 by fastforce
    hence ‹x ∈ {x. P x n} ∪ {x. ∃m ≥ Suc n. P x m}› by blast
  } thus ‹{x. ∃m ≥ n. P x m} ⊆ {x. P x n} ∪ {x. ∃m ≥ Suc n. P x m}› ..
next
  { fix x  assume h:‹x ∈ {x. P x n} ∪ {x. ∃m ≥ Suc n. P x m}›
    hence ‹x ∈ {x. ∃m ≥ n. P x m}› using Suc_leD by blast
  } thus ‹{x. P x n} ∪ {x. ∃m ≥ Suc n. P x m} ⊆ {x. ∃m ≥ n. P x m}› ..
qed
```

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
unfolding TESL_interpretation_atomic_stepwise.simps(1)
          symbolic_run_interpretation_primitive.simps(1,6)
using exists_nat_set_suc[of ‹n› ‹λϱ n. hamlet (Rep_run ϱ n K₁)
                                       ∧ time (Rep_run ϱ n K₂) = τ›]
by (simp add: Collect_conj_eq)


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 -
  have ‹{ϱ. ∀m≥n. R (time ((Rep_run ϱ) m K₁), time ((Rep_run ϱ) m K₂))}
       = {ϱ. R (time ((Rep_run ϱ) n K₁), time ((Rep_run ϱ) n K₂))}
       ∩ {ϱ. ∀m≥Suc n. R (time ((Rep_run ϱ) m K₁), time ((Rep_run ϱ) m K₂))}›
    using forall_nat_set_suc[of ‹n› ‹λx y. R (time ((Rep_run x) y K₁),
                                          time ((Rep_run x) y K₂))›] by simp
  thus ?thesis by auto
qed

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 -
  have ‹{ϱ. ∀m≥n. hamlet ((Rep_run ϱ) m master) ⟶ hamlet ((Rep_run ϱ) m slave)}
        = {ϱ. hamlet ((Rep_run ϱ) n master) ⟶ hamlet ((Rep_run ϱ) n slave)}
        ∩ {ϱ. ∀m≥Suc n. hamlet ((Rep_run ϱ) m master)
                    ⟶ hamlet ((Rep_run ϱ) m slave)}›
    using forall_nat_set_suc[of ‹n› ‹λx y. hamlet ((Rep_run x) y master)
                              ⟶ hamlet ((Rep_run x) y slave)›] by simp
  thus ?thesis by auto
qed

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 -
  have ‹{ϱ. ∀m≥n. hamlet ((Rep_run ϱ) m master) ⟶ ¬ hamlet ((Rep_run ϱ) m slave)}
       = {ϱ. hamlet ((Rep_run ϱ) n master) ⟶ ¬ hamlet ((Rep_run ϱ) n slave)}
          ∩ {ϱ. ∀m≥Suc n. hamlet ((Rep_run ϱ) m master)
                    ⟶ ¬ hamlet ((Rep_run ϱ) m slave)}›
    using forall_nat_set_suc[of ‹n› ‹λx y. hamlet ((Rep_run x) y master)
                             ⟶ ¬hamlet ((Rep_run x) y slave)›] by simp
  thus ?thesis by auto
qed

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 -
  let ?prop = ‹λϱ m. hamlet ((Rep_run ϱ) m master) ⟶
                (let measured_time = time ((Rep_run ϱ) m measuring) in
                 ∀p ≥ m. first_time ϱ measuring p (measured_time + δτ)
                         ⟶ hamlet ((Rep_run ϱ) p slave))›
  have ‹{ϱ. ∀m ≥ n. ?prop ϱ m} = {ϱ. ?prop ϱ n} ∩ {ϱ. ∀m ≥ Suc n. ?prop ϱ m}›
    using forall_nat_set_suc[of ‹n› ?prop] by blast
  also have ‹... = {ϱ. ?prop ϱ n}
              ∩ ⟦ master time-delayed by δτ on measuring implies slave ⟧_TESL^≥ Suc n›
    by simp
  finally show ?thesis by auto
qed

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 -
  have ‹{ϱ. ∀p≥n. (run_tick_count ϱ K₂ p) ≤ (run_tick_count ϱ K₁ p)}
         = {ϱ. (run_tick_count ϱ K₂ n) ≤ (run_tick_count ϱ K₁ n)}
         ∩ {ϱ. ∀p≥Suc n. (run_tick_count ϱ K₂ p) ≤ (run_tick_count ϱ K₁ p)}›
    using forall_nat_set_suc[of ‹n› ‹λϱ n. (run_tick_count ϱ K₂ n)
                                ≤ (run_tick_count ϱ K₁ n)›]
    by simp
  thus ?thesis by auto
qed
```

```
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 -
  have ‹{ϱ. ∀p≥n. (run_tick_count ϱ K₂ p) ≤ (run_tick_count_strictly ϱ K₁ p)}
       = {ϱ. (run_tick_count ϱ K₂ n) ≤ (run_tick_count_strictly ϱ K₁ n)}
       ∩ {ϱ. ∀p≥Suc n. (run_tick_count ϱ K₂ p) ≤ (run_tick_count_strictly ϱ K₁ p)}›
    using forall_nat_set_suc[of ‹n› ‹λϱ n. (run_tick_count ϱ K₂ n)
                                  ≤ (run_tick_count_strictly ϱ K₁ n)›]
    by simp
  thus ?thesis by auto
qed

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 -
  let ?kills = ‹λn ϱ. ∀p≥n. hamlet ((Rep_run ϱ) p K₁)
                          ⟶ (∀m≥p. ¬ hamlet ((Rep_run ϱ) m K₂))›
  let ?ticks = ‹λn ϱ c. hamlet ((Rep_run ϱ) n c)›
  let ?dead = ‹λn ϱ c. ∀m ≥ n. ¬hamlet ((Rep_run ϱ) m c)›
  have ‹⟦ K₁ kills K₂ ⟧TESL≥ n = {ϱ. ?kills n ϱ}› by simp
  also have ‹... = ({ϱ. ¬ ?ticks n ϱ K₁}  ∩ {ϱ. ?kills (Suc n) ϱ})
                 ∪ ({ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂})›
  proof
    { fix ϱ::‹'τ::linordered_field run›
      assume ‹ϱ ∈ {ϱ. ?kills n ϱ}›
      hence ‹?kills n ϱ› by simp
      hence ‹(?ticks n ϱ K₁ ∧ ?dead n ϱ K₂) ∨ (¬?ticks n ϱ K₁ ∧ ?kills (Suc n) ϱ)›
        using Suc_leD by blast
      hence ‹ϱ ∈ ({ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂})
              ∪ ({ϱ. ¬ ?ticks n ϱ K₁} ∩ {ϱ. ?kills (Suc n) ϱ})›
        by blast
    } thus ‹{ϱ. ?kills n ϱ}
           ⊆ {ϱ. ¬ ?ticks n ϱ K₁} ∩ {ϱ. ?kills (Suc n) ϱ}
            ∪ {ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂}› by blast
  next
    { fix ϱ::‹'τ::linordered_field run›
      assume ‹ϱ ∈ ({ϱ. ¬ ?ticks n ϱ K₁}  ∩ {ϱ. ?kills (Suc n) ϱ})
                 ∪ ({ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂})›
      hence ‹¬ ?ticks n ϱ K₁ ∧ ?kills (Suc n) ϱ
             ∨ ?ticks n ϱ K₁ ∧ ?dead n ϱ K₂› by blast
      moreover have ‹((¬ ?ticks n ϱ K₁) ∧ (?kills (Suc n) ϱ)) ⟶ ?kills n ϱ›
        using dual_order.antisym not_less_eq_eq by blast
      ultimately have ‹?kills n ϱ ∨ ?ticks n ϱ K₁ ∧ ?dead n ϱ K₂› by blast
      hence ‹?kills n ϱ› using le_trans by blast
    } thus ‹({ϱ. ¬ ?ticks n ϱ K₁}  ∩ {ϱ. ?kills (Suc n) ϱ})
                 ∪ ({ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂})
          ⊆ {ϱ. ?kills n ϱ}› by blast
  qed
  also have ‹... = {ϱ. ¬ ?ticks n ϱ K₁} ∩ {ϱ. ?kills (Suc n) ϱ}
                 ∪ {ϱ. ?ticks n ϱ K₁} ∩ {ϱ. ?dead n ϱ K₂} ∩ {ϱ. ?kills (Suc n) ϱ}›
    using Collect_cong Collect_disj_eq by auto
  also have ‹... = ⟦ K₁ ¬⇑ n ⟧prim ∩ ⟦ K₁ kills K₂ ⟧TESL≥ Suc n
```

```
                ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ¬⇑ ≥ n ⟧prim
                ∩ ⟦ K₁ kills K₂ ⟧TESL≥ Suc n› by simp
  finally show ?thesis by blast
qed
```

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)›
by (induction Φ, simp, auto)
```

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›
by (induction φ, simp+)

lemma TESL_interpretation_stepwise_zero':
  ‹⟦⟦ Φ ⟧⟧TESL = ⟦⟦ Φ ⟧⟧TESL≥ 0›
by (induction Φ, simp, simp add: TESL_interpretation_stepwise_zero)

lemma TESL_interpretation_stepwise_cons_morph:
  ‹⟦ φ ⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ n = ⟦⟦ φ # Φ ⟧⟧TESL≥ n›
by auto

theorem TESL_interp_stepwise_composition:
  shows ‹⟦⟦ Φ₁ @ Φ₂ ⟧⟧TESL≥ n = ⟦⟦ Φ₁ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ₂ ⟧⟧TESL≥ n›
by (induction Φ₁, simp, auto)
```

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›
  using TESL_interp_stepwise_composition symrun_interp_expansion
```

```
by (simp add: TESL_interp_stepwise_composition
             symrun_interp_expansion inf_assoc inf_left_commute)
```

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 -
  have ‹⟦ Γ, n ⊢ [] ▷ Φ ⟧config = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ [] ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦ Γ, Suc n ⊢ Φ ▷ [] ⟧config
                = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n ∩ ⟦⟦ [] ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ [] ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n
               = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n ∩ ⟦⟦ [] ⟧⟧TESL≥ Suc n›
    by simp
  ultimately show ?thesis by blast
qed
```

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 -
  have ‹⟦ Γ, n ⊢ (K1 sporadic τ on K2) # Ψ ▷ Φ ⟧config
        = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K1 sporadic τ on K2) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦ Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ) ⟧config
                = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                 ∩ ⟦⟦ (K1 sporadic τ on K2) # Φ ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦ ((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ ⟧config
                = ⟦⟦ ((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ) ⟧⟧prim
                 ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
    by simp
  ultimately show ?thesis
  proof -
    have ‹(⟦ K1 ⇑ n ⟧prim ∩ ⟦ K2 ⇓ n @ τ ⟧prim ∪ ⟦ K1 sporadic τ on K2 ⟧TESL≥ Suc n)
          ∩ (⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n)
        = ⟦ K1 sporadic τ on K2 ⟧TESL≥ n ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Γ ⟧⟧prim)›
      using TESL_interp_stepwise_sporadicon_coind_unfold by blast
    hence ‹⟦⟦ ((K1 ⇑ n) # (K2 ⇓ n @ τ) # Γ) ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
          ∪ ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦ K1 sporadic τ on K2 ⟧TESL≥ Suc n
         = ⟦⟦ (K1 sporadic τ on K2) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Γ ⟧⟧prim› by auto
    thus ?thesis by auto
  qed
qed

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 -
```

```
  have ‹⟦ Γ, n ⊢ (time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ ▷ Φ ⟧config
       = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ ⟧⟧TESL≥ n
       ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((⌊τvar(K₁, n), τvar(K₂, n)⌋ ∈ R) # Γ), n
                 ⊢ Ψ ▷ ((time-relation ⌊K₁, K₂⌋ ∈ R) # Φ) ⟧config
                = ⟦⟦ (⌊τvar(K₁, n), τvar(K₂, n)⌋ ∈ R) # Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                ∩ ⟦⟦ (time-relation ⌊K₁, K₂⌋ ∈ R) # Φ ⟧⟧TESL≥ Suc n›
    by simp
  ultimately show ?thesis
  proof -
    have ‹⟦ ⌊τvar(K₁, n), τvar(K₂, n)⌋ ∈ R ⟧prim
          ∩ ⟦ time-relation ⌊K₁, K₂⌋ ∈ R ⟧TESL≥ Suc n
          ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n = ⟦⟦ (time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ ⟧⟧TESL≥ n›
      using TESL_interp_stepwise_tagrel_coind_unfold
            TESL_interpretation_stepwise_cons_morph by blast
    thus ?thesis by auto
  qed
qed

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 -
  have ‹⟦ Γ, n ⊢ (K₁ implies K₂) # Ψ ▷ Φ ⟧config
        = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K₁ implies K₂) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config
               = ⟦⟦ (K₁ ¬⇑ n) # Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
               ∩ ⟦⟦ (K₁ implies K₂) # Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config
               =  ⟦⟦ ((K₁ ⇑ n) # (K₂ ⇑ n) # Γ) ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                ∩ ⟦⟦ (K₁ implies K₂) # Φ ⟧⟧TESL≥ Suc n› by simp
  ultimately show ?thesis
  proof -
    have f1: ‹(⟦ K₁ ¬⇑ n ⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ⇑ n ⟧prim)
               ∩ ⟦ K₁ implies K₂ ⟧TESL≥ Suc n ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n
               ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n)
             = ⟦⟦ (K₁ implies K₂) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
      using TESL_interp_stepwise_implies_coind_unfold
            TESL_interpretation_stepwise_cons_morph by blast
    have ‹⟦ K₁ ¬⇑ n ⟧prim ∩ ⟦⟦ Γ ⟧⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦⟦ (K₂ ⇑ n) # Γ ⟧⟧prim
         = (⟦ K₁ ¬⇑ n ⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ⇑ n ⟧prim) ∩ ⟦⟦ Γ ⟧⟧prim›
      by force
    hence ‹⟦ Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ ⟧config
      = (⟦ K₁ ¬⇑ n ⟧prim ∩ ⟦⟦ Γ ⟧⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦⟦ (K₂ ⇑ n) # Γ ⟧⟧prim)
        ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ (K₁ implies K₂) # Φ ⟧⟧TESL≥ Suc n)›
      using f1 by (simp add: inf_left_commute inf_assoc)
    thus ?thesis by (simp add: Int_Un_distrib2 inf_assoc)
  qed
qed

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 -
  have ‹⟦ Γ, n ⊢ (K₁ implies not K₂) # Ψ ▷ Φ ⟧config
        = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K₁ implies not K₂) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
```

```
    by simp
  moreover have ‹⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ) ⟧config
                  = ⟦⟦ (K₁ ¬⇑ n) # Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                  ∩ ⟦⟦ (K₁ implies not K₂) # Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((K₁ ⇑ n) # (K₂ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ) ⟧config
                  = ⟦⟦ ((K₁ ⇑ n) # (K₂ ¬⇑ n) # Γ) ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                  ∩ ⟦⟦ (K₁ implies not K₂) # Φ ⟧⟧TESL≥ Suc n› by simp
  ultimately show ?thesis
  proof -
    have f1: ‹(⟦ K₁ ¬⇑ n ⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ¬⇑ n ⟧prim)
              ∩ ⟦ K₁ implies not K₂ ⟧TESL≥ Suc n
              ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n)
              = ⟦⟦ (K₁ implies not K₂) # Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n›
      using TESL_interp_stepwise_implies_not_coind_unfold
            TESL_interpretation_stepwise_cons_morph by blast
    have ‹⟦ K₁ ¬⇑ n ⟧prim ∩ ⟦⟦ Γ ⟧⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦⟦ (K₂ ¬⇑ n) # Γ ⟧⟧prim
          = (⟦ K₁ ¬⇑ n ⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ ¬⇑ n ⟧prim) ∩ ⟦⟦ Γ ⟧⟧prim›
      by force
    then have ‹⟦ Γ, n ⊢ ((K₁ implies not K₂) # Ψ) ▷ Φ ⟧config
                = (⟦ K₁ ¬⇑ n ⟧prim ∩ ⟦⟦ Γ ⟧⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim
                  ∩ ⟦⟦ (K₂ ¬⇑ n) # Γ ⟧⟧prim) ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n
                  ∩ ⟦⟦ (K₁ implies not K₂) # Φ ⟧⟧TESL≥ Suc n)›
      using f1 by (simp add: inf_left_commute inf_assoc)
    thus ?thesis by (simp add: Int_Un_distrib2 inf_assoc)
  qed
qed

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 -
  have 1:‹⟦ Γ, n ⊢ (K₁ time-delayed by δτ on K₂ implies K₃) # Ψ ▷ Φ ⟧config
         = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K₁ time-delayed by δτ on K₂ implies K₃) # Ψ ⟧⟧TESL≥ n
          ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((K₁ ¬⇑ n) # Γ), n
                  ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config
                 = ⟦⟦ (K₁ ¬⇑ n) # Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                  ∩ ⟦⟦ (K₁ time-delayed by δτ on K₂ implies K₃) # Φ ⟧⟧TESL≥ Suc n›
    by simp
  moreover have ‹⟦ ((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
                  ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config
                 = ⟦⟦ (K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                  ∩ ⟦⟦ (K₁ time-delayed by δτ on K₂ implies K₃) # Φ ⟧⟧TESL≥ Suc n›
    by simp
  ultimately show ?thesis
  proof -
    have ‹⟦ Γ, n ⊢ (K₁ time-delayed by δτ on K₂ implies K₃) # Ψ ▷ Φ ⟧config
      = ⟦⟦ Γ ⟧⟧prim ∩ (⟦⟦ (K₁ time-delayed by δτ on K₂ implies K₃) # Ψ ⟧⟧TESL≥ n
        ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n)›
      using 1 by blast
    hence ‹⟦ Γ, n ⊢ (K₁ time-delayed by δτ on K₂ implies K₃) # Ψ ▷ Φ ⟧config
          = (⟦ K₁ ¬⇑ n ⟧prim ∪ ⟦ K₁ ⇑ n ⟧prim ∩ ⟦ K₂ @ n ⊕ δτ ⇒ K₃ ⟧prim)
            ∩ (⟦⟦ Γ ⟧⟧prim ∩ (⟦⟦ Ψ ⟧⟧TESL≥ n
            ∩ ⟦⟦ (K₁ time-delayed by δτ on K₂ implies K₃) # Φ ⟧⟧TESL≥ Suc n))›
      using TESL_interpretation_stepwise_cons_morph
            TESL_interp_stepwise_timedelayed_coind_unfold
    proof -
```

```
      have ‹⟦⟦ (K1 time-delayed by δτ on K2 implies K3) # Ψ ⟧⟧TESL≥ n
            = (⟦ K1 ¬⇑ n ⟧prim ∪ ⟦ K1 ⇑ n ⟧prim ∩ ⟦ K2 @ n ⊕ δτ ⇒ K3 ⟧prim)
            ∩ ⟦ K1 time-delayed by δτ on K2 implies K3 ⟧TESL≥ Suc n ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n›
        using TESL_interp_stepwise_timedelayed_coind_unfold
              TESL_interpretation_stepwise_cons_morph by blast
      then show ?thesis
        by (simp add: Int_assoc Int_left_commute)
    qed
    then show ?thesis by (simp add: inf_assoc inf_sup_distrib2)
  qed
qed

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 -
  have ‹⟦ Γ, n ⊢ (K1 weakly precedes K2) # Ψ ▷ Φ ⟧config
        = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K1 weakly precedes K2) # Ψ ⟧⟧TESL≥ n
          ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((⌈#≤ K2 n, #≤ K1 n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
                  ⊢ Ψ ▷ ((K1 weakly precedes K2) # Φ) ⟧config
                = ⟦⟦ (⌈#≤ K2 n, #≤ K1 n⌉ ∈ (λ(x,y). x≤y)) # Γ ⟧⟧prim
                ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n ∩ ⟦⟦ (K1 weakly precedes K2) # Φ ⟧⟧TESL≥ Suc n›
    by simp
  ultimately show ?thesis
  proof -
    have ‹⟦ ⌈#≤ K2 n, #≤ K1 n⌉ ∈ (λ(x,y). x≤y) ⟧prim
            ∩ ⟦ K1 weakly precedes K2 ⟧TESL≥ Suc n ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
          = ⟦⟦ (K1 weakly precedes K2) # Ψ ⟧⟧TESL≥ n›
      using TESL_interp_stepwise_weakly_precedes_coind_unfold
            TESL_interpretation_stepwise_cons_morph by blast
    thus ?thesis by auto
  qed
qed

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 -
  have ‹⟦ Γ, n ⊢ (K1 strictly precedes K2) # Ψ ▷ Φ ⟧config
        = ⟦⟦ Γ ⟧⟧prim ∩ ⟦⟦ (K1 strictly precedes K2) # Ψ ⟧⟧TESL≥ n
          ∩ ⟦⟦ Φ ⟧⟧TESL≥ Suc n› by simp
  moreover have ‹⟦ ((⌈#≤ K2 n, #< K1 n⌉ ∈ (λ(x,y). x≤y)) # Γ), n
                  ⊢ Ψ ▷ ((K1 strictly precedes K2) # Φ) ⟧config
                = ⟦⟦ (⌈#≤ K2 n, #< K1 n⌉ ∈ (λ(x,y). x≤y)) # Γ ⟧⟧prim
                ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
                ∩ ⟦⟦ (K1 strictly precedes K2) # Φ ⟧⟧TESL≥ Suc n› by simp
  ultimately show ?thesis
  proof -
    have ‹⟦ ⌈#≤ K2 n, #< K1 n⌉ ∈ (λ(x,y). x≤y) ⟧prim
            ∩ ⟦ K1 strictly precedes K2 ⟧TESL≥ Suc n ∩ ⟦⟦ Ψ ⟧⟧TESL≥ n
          = ⟦⟦ (K1 strictly precedes K2) # Ψ ⟧⟧TESL≥ n›
      using TESL_interp_stepwise_strictly_precedes_coind_unfold
            TESL_interpretation_stepwise_cons_morph by blast
    thus ?thesis by auto
  qed
qed
```

```
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 -
  have ‹⟦ Γ, n ⊢ ((K1 kills K2) # Ψ) ▹ Φ ⟧_config
        = ⟦⟦ Γ ⟧⟧_prim ∩ ⟦⟦ (K1 kills K2) # Ψ ⟧⟧_TESL^≥ n ∩ ⟦⟦ Φ ⟧⟧_TESL^≥ Suc n›
    by simp
  moreover have ‹⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ) ⟧_config
                = ⟦⟦ (K1 ¬⇑ n) # Γ ⟧⟧_prim ∩ ⟦⟦ Ψ ⟧⟧_TESL^≥ n
                  ∩ ⟦⟦ (K1 kills K2) # Φ ⟧⟧_TESL^≥ Suc n› by simp
  moreover have ‹⟦ ((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ) ⟧_config
                = ⟦⟦ (K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ ⟧⟧_prim ∩ ⟦⟦ Ψ ⟧⟧_TESL^≥ n
                  ∩ ⟦⟦ (K1 kills K2) # Φ ⟧⟧_TESL^≥ Suc n› by simp
  ultimately show ?thesis
    proof -
      have ‹⟦⟦ (K1 kills K2) # Ψ ⟧⟧_TESL^≥ n
            = (⟦ (K1 ¬⇑ n) ⟧_prim ∪ ⟦ (K1 ⇑ n) ⟧_prim ∩ ⟦ (K2 ¬⇑ ≥ n) ⟧_prim)
              ∩ ⟦ (K1 kills K2) ⟧_TESL^≥ Suc n ∩ ⟦⟦ Ψ ⟧⟧_TESL^≥ n›
        using TESL_interp_stepwise_kills_coind_unfold
              TESL_interpretation_stepwise_cons_morph by blast
      thus ?thesis by auto
    qed
qed

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 ‹⟦⟦ Ψ ⟧⟧TESL = ⟦ [], 0 ⊢ Ψ ▷ [] ⟧config›
  proof -
    have ‹⟦⟦ Ψ ⟧⟧TESL = ⟦⟦ Ψ ⟧⟧TESL≥ 0›
    by (simp add: TESL_interpretation_stepwise_zero')
    moreover have ‹⟦ [], 0 ⊢ Ψ ▷ [] ⟧config =
                    ⟦⟦ [] ⟧⟧prim ∩ ⟦⟦ Ψ ⟧⟧TESL≥ 0 ∩ ⟦⟦ [] ⟧⟧TESL≥ Suc 0›
    by simp
    ultimately show ?thesis by auto
  qed
```

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 ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1)  ↪  (Γ2, n2 ⊢ Ψ2 ▷ Φ2)›
  shows ‹⟦⟦ Γ1 ⟧⟧prim ∩ ⟦⟦ Ψ1 ⟧⟧TESL≥ n1 ∩ ⟦⟦ Φ1 ⟧⟧TESL≥ Suc n1
         ⊇  ⟦⟦ Γ2 ⟧⟧prim ∩ ⟦⟦ Ψ2 ⟧⟧TESL≥ n2 ∩ ⟦⟦ Φ2 ⟧⟧TESL≥ Suc n2› (is ?P)
proof -
```

```
from assms consider
  (a) ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁)  ↪ᵢ  (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)›
| (b) ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁)  ↪ₑ  (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂)›
  using operational_semantics_step.simps by blast
thus ?thesis
proof (cases)
  case a
    thus ?thesis by (simp add: operational_semantics_intro.simps)
next
  case b thus ?thesis
  proof (rule operational_semantics_elim.cases)
    fix  Γ n K₁ τ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ (K₁ sporadic τ on K₂) # Ψ ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (Γ, n ⊢ Ψ ▷ ((K₁ sporadic τ on K₂) # Φ))›
    thus ?P using HeronConf_interp_stepwise_sporadicon_cases
                  HeronConf_interpretation.simps by blast
  next
    fix  Γ n K₁ τ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ (K₁ sporadic τ on K₂) # Ψ ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((K₁ ⇑ n) # (K₂ ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)›
    thus ?P using HeronConf_interp_stepwise_sporadicon_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ K₂ R Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ (time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((⌊τ_var (K₁, n), τ_var (K₂, n)⌋ ∈ R) # Γ), n
                                  ⊢ Ψ ▷ ((time-relation ⌊K₁, K₂⌋ ∈ R) # Φ))›
    thus ?P using HeronConf_interp_stepwise_tagrel_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ (K₁ implies K₂) # Ψ ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))›
    thus ?P using HeronConf_interp_stepwise_implies_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n
                                ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))›
    thus ?P using HeronConf_interp_stepwise_implies_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ ((K₁ implies not K₂) # Ψ) ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ))›
    thus ?P using HeronConf_interp_stepwise_implies_not_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ K₂ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) = (Γ, n ⊢ ((K₁ implies not K₂) # Ψ) ▷ Φ)›
    and ‹(Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) = (((K₁ ⇑ n) # (K₂ ¬⇑ n) # Γ), n
                                ⊢ Ψ ▷ ((K₁ implies not K₂) # Φ))›
    thus ?P using HeronConf_interp_stepwise_implies_not_cases
                  HeronConf_interpretation.simps by blast
  next
    fix Γ n K₁ δτ K₂ K₃ Ψ Φ
    assume ‹(Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁) =
             (Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)›
```

```
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2) =
            (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 time-delayed by δτ on K2 implies K3) # Φ))›
      thus ?P using HeronConf_interp_stepwise_timedelayed_cases
                  HeronConf_interpretation.simps by blast
    next
      fix Γ n K1 δτ K2 K3 Ψ Φ
      assume ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1) =
               (Γ, n ⊢ ((K1 time-delayed by δτ on K2 implies K3) # Ψ) ▷ Φ)›
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2)
            = (((K1 ⇑ n) # (K2 @ n ⊕ δτ ⇒ K3) # Γ), n
                ⊢ Ψ ▷ ((K1 time-delayed by δτ on K2 implies K3) # Φ))›
      thus ?P using HeronConf_interp_stepwise_timedelayed_cases
                  HeronConf_interpretation.simps by blast
    next
      fix Γ n K1 K2 Ψ Φ
      assume ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1) = (Γ, n ⊢ ((K1 weakly precedes K2) # Ψ) ▷ Φ)›
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2) = ((([#≤ K2 n, #≤ K1 n] ∈ (λ(x, y). x ≤ y)) # Γ), n
                                 ⊢ Ψ ▷ ((K1 weakly precedes K2) # Φ))›
      thus ?P using HeronConf_interp_stepwise_weakly_precedes_cases
                  HeronConf_interpretation.simps by blast
    next
      fix Γ n K1 K2 Ψ Φ
      assume ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1) = (Γ, n ⊢ ((K1 strictly precedes K2) # Ψ) ▷ Φ)›
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2) = ((([#≤ K2 n, #< K1 n] ∈ (λ(x, y). x ≤ y)) # Γ), n
                                 ⊢ Ψ ▷ ((K1 strictly precedes K2) # Φ))›
      thus ?P using HeronConf_interp_stepwise_strictly_precedes_cases
                  HeronConf_interpretation.simps by blast
    next
      fix Γ n K1 K2 Ψ Φ
      assume ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1) = (Γ, n ⊢ ((K1 kills K2) # Ψ) ▷ Φ)›
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2) = (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 kills K2) # Φ))›
      thus ?P using HeronConf_interp_stepwise_kills_cases
                  HeronConf_interpretation.simps by blast
    next
      fix Γ n K1 K2 Ψ Φ
      assume ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1) = (Γ, n ⊢ ((K1 kills K2) # Ψ) ▷ Φ)›
      and ‹(Γ2, n2 ⊢ Ψ2 ▷ Φ2) =
            (((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K1 kills K2) # Φ))›
      thus ?P using HeronConf_interp_stepwise_kills_cases
                  HeronConf_interpretation.simps by blast
    qed
  qed
qed

inductive_cases step_elim:‹𝒮1 ↪ 𝒮2›

lemma sound_reduction':
  assumes ‹𝒮1 ↪ 𝒮2›
  shows ‹⟦ 𝒮1 ⟧config ⊇ ⟦ 𝒮2 ⟧config›
proof -
  have ‹∀s1 s2. (⟦ s2 ⟧config ⊆ ⟦ s1 ⟧config) ∨ ¬(s1 ↪ s2)›
    using sound_reduction by fastforce
  thus ?thesis using assms by blast
qed

lemma sound_reduction_generalized:
  assumes ‹𝒮1 ↪^k 𝒮2›
    shows ‹⟦ 𝒮1 ⟧config ⊇ ⟦ 𝒮2 ⟧config›
proof -
```

```
  from assms show ?thesis
  proof (induction k arbitrary: 𝒮₂)
    case 0
      hence *: ‹𝒮₁ ↪⁰ 𝒮₂ ⟹ 𝒮₁ = 𝒮₂› by auto
      moreover have ‹𝒮₁ = 𝒮₂› using * "0.prems" by linarith
      ultimately show ?case by auto
  next
    case (Suc k)
      thus ?case
      proof -
        fix k :: nat
        assume ff: ‹𝒮₁ ↪^Suc k 𝒮₂›
        assume hi: ‹⋀𝒮₂. 𝒮₁ ↪ᵏ 𝒮₂ ⟹ ⟦ 𝒮₂ ⟧config ⊆ ⟦ 𝒮₁ ⟧config›
        obtain 𝒮ₙ where red_decomp: ‹(𝒮₁ ↪ᵏ 𝒮ₙ) ∧ (𝒮ₙ ↪ 𝒮₂)› using ff by auto
        hence ‹⟦ 𝒮₁ ⟧config ⊇ ⟦ 𝒮ₙ ⟧config› using hi by simp
        also have ‹⟦ 𝒮ₙ ⟧config ⊇ ⟦ 𝒮₂ ⟧config› by (simp add: red_decomp sound_reduction')
        ultimately show ‹⟦ 𝒮₁ ⟧config ⊇ ⟦ 𝒮₂ ⟧config› by simp
      qed
  qed
qed
```

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›
  using assms sound_reduction_generalized solve_start by blast
```

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∈𝒞next (Γ, n ⊢ Ψ ▷ Φ). ⟦ X ⟧config)›
  proof (induct Ψ)
    case Nil
    show ?case
      using HeronConf_interp_stepwise_instant_cases operational_semantics_step.simps
            operational_semantics_intro.instant_i
      by fastforce
  next
    case (Cons ψ Ψ)  thus ?case
      proof (cases ψ)
        case (SporadicOn K1 τ K2) thus ?thesis
          using HeronConf_interp_stepwise_sporadicon_cases
                                       [of ‹Γ› ‹n› ‹K1› ‹τ› ‹K2› ‹Ψ› ‹Φ›]
                Cnext_solve_sporadicon[of ‹Γ› ‹n› ‹Ψ› ‹K1› ‹τ› ‹K2› ‹Φ›] by blast
      next
        case (TagRelation K₁ K₂ R) thus ?thesis
          using HeronConf_interp_stepwise_tagrel_cases
                                    [of ‹Γ› ‹n› ‹K₁› ‹K₂› ‹R› ‹Ψ› ‹Φ›]
                Cnext_solve_tagrel[of ‹K₁› ‹n› ‹K₂› ‹R› ‹Γ› ‹Ψ› ‹Φ›] by blast
      next
        case (Implies K1 K2) thus ?thesis
```

```
        using HeronConf_interp_stepwise_implies_cases
                                [of ‹Γ› ‹n› ‹K1› ‹K2› ‹Ψ› ‹Φ›]
              Cnext_solve_implies[of ‹K1› ‹n› ‹Γ› ‹Ψ› ‹K2› ‹Φ›] by blast
    next
      case (ImpliesNot K1 K2) thus ?thesis
        using HeronConf_interp_stepwise_implies_not_cases
                                    [of ‹Γ› ‹n› ‹K1› ‹K2› ‹Ψ› ‹Φ›]
              Cnext_solve_implies_not[of ‹K1› ‹n› ‹Γ› ‹Ψ› ‹K2› ‹Φ›] by blast
    next
      case (TimeDelayedBy Kmast τ Kmeas Kslave) thus ?thesis
        using HeronConf_interp_stepwise_timedelayed_cases
                       [of ‹Γ› ‹n› ‹Kmast› ‹τ› ‹Kmeas› ‹Kslave› ‹Ψ› ‹Φ›]
              Cnext_solve_timedelayed
                       [of ‹Kmast› ‹n› ‹Γ› ‹Ψ› ‹τ› ‹Kmeas› ‹Kslave› ‹Φ›] by blast
    next
      case (WeaklyPrecedes K1 K2) thus ?thesis
        using HeronConf_interp_stepwise_weakly_precedes_cases
                                        [of ‹Γ› ‹n› ‹K1› ‹K2› ‹Ψ› ‹Φ›]
              Cnext_solve_weakly_precedes[of ‹K2› ‹n› ‹K1› ‹Γ› ‹Ψ›  ‹Φ›]
        by blast
    next
      case (StrictlyPrecedes K1 K2)  thus ?thesis
        using HeronConf_interp_stepwise_strictly_precedes_cases
                                          [of ‹Γ› ‹n› ‹K1› ‹K2› ‹Ψ› ‹Φ›]
              Cnext_solve_strictly_precedes[of ‹K2› ‹n› ‹K1› ‹Γ› ‹Ψ›  ‹Φ›]
        by blast
    next
      case (Kills K1 K2) thus ?thesis
        using HeronConf_interp_stepwise_kills_cases[of ‹Γ› ‹n› ‹K1› ‹K2› ‹Ψ› ‹Φ›]
              Cnext_solve_kills[of ‹K1› ‹n› ‹Γ› ‹Ψ› ‹K2› ‹Φ›] by blast
    qed
  qed

lemma complete_direct_successors':
  shows ‹⟦ 𝒮 ⟧config ⊆ (⋃X∈𝒞next 𝒮. ⟦ X ⟧config)›
proof -
  from HeronConf_interpretation.cases obtain Γ n Ψ Φ
    where ‹𝒮 = (Γ, n ⊢ Ψ ▷ Φ)› by blast
  with complete_direct_successors[of ‹Γ› ‹n› ‹Ψ› ‹Φ›] show ?thesis by simp
qed
```

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 -
  from assms complete_direct_successors' have ‹ϱ ∈ (⋃X∈𝒞next 𝒮1. ⟦ X ⟧config)› by blast
  hence ‹∃s∈𝒞next 𝒮1. ϱ ∈ ⟦ s ⟧config› by simp
  thus ?thesis by blast
qed

lemma branch_existence':
  assumes ‹ϱ ∈ ⟦ 𝒮1 ⟧config›
  shows ‹∃𝒮2. (𝒮1 ↪k 𝒮2) ∧ (ϱ ∈ ⟦ 𝒮2 ⟧config)›
proof (induct k)
  case 0
    thus ?case by (simp add: assms)
```

```
next
  case (Suc k)
    thus ?case
      using branch_existence relpowp_Suc_I[of ‹k› ‹operational_semantics_step›]
    by blast
qed
```

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›
  using assms branch_existence' solve_start by blast
```

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 (insert assms, induct Ψ arbitrary: Γ Φ)
  case (Nil Γ Φ)
    then show ?case
    proof -
      have ‹(Γ, n ⊢ [] ▷ Φ) ↪¹ (Γ, Suc n ⊢ Φ ▷ [])›
        using instant_i intro_part by fastforce
      moreover have ‹⟦ Γ, n ⊢ [] ▷ Φ ⟧config = ⟦ Γ, Suc n ⊢ Φ ▷ [] ⟧config›
        by auto
      moreover have ‹ϱ ∈ ⟦ Γ, Suc n ⊢ Φ ▷ [] ⟧config›
        using assms Nil.prems calculation(2) by blast
      ultimately show ?thesis by blast
    qed
next
  case (Cons ψ Ψ)
    then show ?case
    proof (induct ψ)
      case (SporadicOn K1 τ K2)
        have branches: ‹⟦ 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›
          using HeronConf_interp_stepwise_sporadicon_cases by simp
        have br1: ‹ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ) ⟧config
                   ⟹ ∃Γk Ψk Φk k.
                      ((Γ, n ⊢ ((K1 sporadic τ on K2) # Ψ) ▷ Φ)
                          ↪ᵏ (Γk, Suc n ⊢ Ψk ▷ Φk))
                      ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
        proof -
          assume h1: ‹ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ) ⟧config›
          hence ‹∃Γk Ψk Φk k. ((Γ, n ⊢ Ψ ▷ ((K1 sporadic τ on K2) # Φ))
                                  ↪ᵏ (Γk, Suc n ⊢ Ψk ▷ Φk))
                              ∧ (ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config)›
```

```
          using h1 SporadicOn.prems by simp
        from this obtain Γ_k Ψ_k Φ_k k where
            fp:‹((Γ, n ⊢ Ψ ▷ ((K_1 sporadic τ on K_2) # Φ))
                  ↪^k (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))
              ∧ ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config› by blast
        have
          ‹(Γ, n ⊢ ((K_1 sporadic τ on K_2) # Ψ) ▷ Φ)
            ↪ (Γ, n ⊢ Ψ ▷ ((K_1 sporadic τ on K_2) # Φ))›
          by (simp add: elims_part sporadic_on_e1)
        with fp relpowp_Suc_I2 have
          ‹((Γ, n ⊢ ((K_1 sporadic τ on K_2) # Ψ) ▷ Φ)
            ↪^(Suc k) (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))› by auto
        thus ?thesis using fp by blast
      qed
      have br2: ‹ϱ ∈ ⟦ ((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ ⟧_config
                ⟹ ∃Γ_k Ψ_k Φ_k k. ((Γ, n ⊢ ((K_1 sporadic τ on K_2) # Ψ) ▷ Φ)
                                     ↪^k (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))
                          ∧ ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config›
      proof -
        assume h2: ‹ϱ ∈ ⟦ ((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ ⟧_config›
        hence ‹∃Γ_k Ψ_k Φ_k k. ((((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)
                                   ↪^k (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))
                          ∧ ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config›
          using h2 SporadicOn.prems by simp

          from this obtain Γ_k Ψ_k Φ_k k
          where fp:‹((((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)
                          ↪^k (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k))›
            and rc:‹ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config› by blast
          have pc:‹(Γ, n ⊢ ((K_1 sporadic τ on K_2) # Ψ) ▷ Φ)
            ↪ (((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)›
          by (simp add: elims_part sporadic_on_e2)
          hence ‹(Γ, n ⊢ (K_1 sporadic τ on K_2) # Ψ ▷ Φ)
                ↪^(Suc k) (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k)›
              using fp relpowp_Suc_I2 by auto
          with rc show ?thesis by blast
      qed
      from branches SporadicOn.prems(2) have
        ‹ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ ((K_1 sporadic τ on K_2) # Φ) ⟧_config
          ∪ ⟦ ((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ ⟧_config›
        by simp
      with br1 br2 show ?case by blast
  next
    case (TagRelation K_1 K_2 R)
      have branches: ‹⟦ Γ, n ⊢ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Ψ) ▷ Φ ⟧_config
          = ⟦ ((⌊τ_var(K_1, n), τ_var(K_2, n)⌋ ∈ R) # Γ), n
              ⊢ Ψ ▷ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Φ) ⟧_config›
        using HeronConf_interp_stepwise_tagrel_cases by simp
      thus ?case
      proof -
        have ‹∃Γ_k Ψ_k Φ_k k.
            ((((⌊τ_var(K_1, n), τ_var(K_2, n)⌋ ∈ R) # Γ), n
                ⊢ Ψ ▷ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Φ))
              ↪^k (Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k)) ∧ ϱ ∈ ⟦ Γ_k, Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config›
          using TagRelation.prems by simp

        from this obtain Γ_k Ψ_k Φ_k k
          where fp:‹((((⌊τ_var(K_1, n), τ_var(K_2, n)⌋ ∈ R) # Γ), n
                      ⊢ Ψ ▷ ((time-relation ⌊K_1, K_2⌋ ∈ R) # Φ))
```

```
              ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ))›
          and rc:‹ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config› by blast
      have pc:‹(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))›
        by (simp add: elims_part tagrel_e)
      hence ‹(Γ, n ⊢ (time-relation ⌊K₁, K₂⌋ ∈ R) # Ψ ▷ Φ)
              ↪Suc k (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
  next
    case (Implies K₁ K₂)
      have branches: ‹⟦ 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›
        using HeronConf_interp_stepwise_implies_cases by simp
      moreover have br1: ‹ϱ ∈ ⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config
              ⟹ ∃Γₖ Ψₖ Φₖ k. ((Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ)
                                  ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ))
                ∧ ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config›
      proof -
        assume h1: ‹ϱ ∈ ⟦ ((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config›
        then have ‹∃Γₖ Ψₖ Φₖ k.
                  ((((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))
                      ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ))
                ∧ ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config›
          using h1 Implies.prems by simp
        from this obtain Γₖ Ψₖ Φₖ k where
          fp:‹((((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))
            ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ))›
          and rc:‹ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config› by blast
        have pc:‹(Γ, n ⊢ (K₁ implies K₂) # Ψ ▷ Φ)
                ↪ (((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))›
          by (simp add: elims_part implies_e1)
        hence ‹(Γ, n ⊢ (K₁ implies K₂) # Ψ ▷ Φ) ↪Suc k (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ)›
          using fp relpowp_Suc_I2 by auto
        with rc show ?thesis by blast
      qed
      moreover have br2: ‹ϱ ∈ ⟦ ((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n
                              ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config
                          ⟹ ∃Γₖ Ψₖ Φₖ k. ((Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ)
                                              ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ))
                            ∧ ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config›
      proof -
        assume h2: ‹ϱ ∈ ⟦ ((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n
                        ⊢ Ψ ▷ ((K₁ implies K₂) # Φ) ⟧config›
        then have ‹∃Γₖ Ψₖ Φₖ k. (
                      (((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))
                        ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ)
                  ) ∧ ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config›
          using h2 Implies.prems by simp
        from this obtain Γₖ Ψₖ Φₖ k where
            fp:‹(((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))
              ↪ᵏ (Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ)›
        and rc:‹ϱ ∈ ⟦ Γₖ, Suc n ⊢ Ψₖ ▷ Φₖ ⟧config› by blast
        have ‹(Γ, n ⊢ ((K₁ implies K₂) # Ψ) ▷ Φ)
              ↪ (((K₁ ⇑ n) # (K₂ ⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ implies K₂) # Φ))›
          by (simp add: elims_part implies_e2)
```

```
      hence ‹(Γ, n ⊢ ((K1 implies K2) # Ψ) ▷ Φ) ↪Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
    ultimately show ?case using Implies.prems(2) by blast
next
  case (ImpliesNot K1 K2)
    have branches: ‹⟦ 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›
      using HeronConf_interp_stepwise_implies_not_cases by simp
    moreover have br1: ‹ϱ ∈ ⟦ ((K1 ¬⇑ n) # Γ), n
                           ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config
              ⟹ ∃Γk Ψk Φk k. ((Γ, n ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
                                    ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
                 ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
    proof -
      assume h1: ‹ϱ ∈ ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config›
      then have ‹∃Γk Ψk Φk k.
                 ((((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))
                   ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
                 ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
        using h1 ImpliesNot.prems by simp
      from this obtain Γk Ψk Φk k where
        fp:‹((((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))
              ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))›
        and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
      have pc:‹(Γ, n ⊢ (K1 implies not K2) # Ψ ▷ Φ)
                 ↪ (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))›
        by (simp add: elims_part implies_not_e1)
      hence ‹(Γ, n ⊢ (K1 implies not K2) # Ψ ▷ Φ) ↪Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
    moreover have br2: ‹ϱ ∈ ⟦ ((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n
                           ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config
                           ⟹ ∃Γk Ψk Φk k. ((Γ, n ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
                                                 ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
                                  ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
    proof -
      assume h2: ‹ϱ ∈ ⟦ ((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n
                         ⊢ Ψ ▷ ((K1 implies not K2) # Φ) ⟧config›
      then have ‹∃Γk Ψk Φk k. (
                  (((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n
                    ⊢ Ψ ▷ ((K1 implies not K2) # Φ)) ↪k (Γk, Suc n ⊢ Ψk ▷ Φk)
                 ) ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
        using h2 ImpliesNot.prems by simp
      from this obtain Γk Ψk Φk k where
          fp:‹(((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))
              ↪k (Γk, Suc n ⊢ Ψk ▷ Φk)›
      and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
      have ‹(Γ, n ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
            ↪ (((K1 ⇑ n) # (K2 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K1 implies not K2) # Φ))›
        by (simp add: elims_part implies_not_e2)
      hence ‹(Γ, n ⊢ ((K1 implies not K2) # Ψ) ▷ Φ)
              ↪Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
```

```
    ultimately show ?case  using ImpliesNot.prems(2) by blast
next
  case (TimeDelayedBy K₁ δτ K₂ K₃)
    have branches:
      ‹⟦ 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›
      using HeronConf_interp_stepwise_timedelayed_cases by simp
    moreover have br1:
      ‹ϱ ∈ ⟦ ((K₁ ¬⇑ n) # Γ), n
            ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config
        ⟹ ∃Γk Ψk Φk k.
          ((Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)
              ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
          ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
    proof -
      assume h1: ‹ϱ ∈ ⟦ ((K₁ ¬⇑ n) # Γ), n
                      ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config›
      then have ‹∃Γk Ψk Φk k.
        ((((K₁ ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ))
          ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
        ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
        using h1 TimeDelayedBy.prems by simp
      from this obtain Γk Ψk Φk k
        where fp:‹(((K₁ ¬⇑ n) # Γ), n
                    ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ))
                  ↪k (Γk, Suc n ⊢ Ψk ▷ Φk)›
          and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
      have ‹(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))›
        by (simp add: elims_part timedelayed_e1)
      hence ‹(Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)
              ↪Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
    moreover have br2:
      ‹ϱ ∈ ⟦ ((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
            ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config
        ⟹ ∃Γk Ψk Φk k.
            ((Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)
              ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
            ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
    proof -
      assume h2: ‹ϱ ∈ ⟦ ((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
                  ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ) ⟧config›
      then have ‹∃Γk Ψk Φk k. ((((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
                              ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ))
                              ↪k (Γk, Suc n ⊢ Ψk ▷ Φk))
                            ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config›
        using h2 TimeDelayedBy.prems by simp
      from this obtain Γk Ψk Φk k
        where fp:‹(((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
                      ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ))
                    ↪k (Γk, Suc n ⊢ Ψk ▷ Φk)›
          and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
```

```
      have ‹(Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)
            ↪ (((K₁ ⇑ n) # (K₂ @ n ⊕ δτ ⇒ K₃) # Γ), n
                ⊢ Ψ ▷ ((K₁ time-delayed by δτ on K₂ implies K₃) # Φ))›
        by (simp add: elims_part timedelayed_e2)
      with fp relpowp_Suc_I2 have
        ‹(Γ, n ⊢ ((K₁ time-delayed by δτ on K₂ implies K₃) # Ψ) ▷ Φ)
          ↪^Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        by auto
      with rc show ?thesis by blast
    qed
    ultimately show ?case using TimeDelayedBy.prems(2) by blast
next
  case (WeaklyPrecedes K₁ K₂)
    have ‹⟦ 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›
      using HeronConf_interp_stepwise_weakly_precedes_cases by simp
    moreover have ‹ϱ ∈ ⟦ ((⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                          ⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ) ⟧config
          ⟹ (∃Γk Ψk Φk k. ((Γ, n ⊢ ((K₁ weakly precedes K₂) # Ψ) ▷ Φ)
                                  ↪^k (Γk, Suc n ⊢ Ψk ▷ Φk))
                ∧ (ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config))›
    proof -
      assume ‹ϱ ∈ ⟦ ((⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                     ⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ) ⟧config›
      hence ‹∃Γk Ψk Φk k. ((((⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                                  ⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ))
                              ↪^k (Γk, Suc n ⊢ Ψk ▷ Φk))
                           ∧ (ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config)›
        using  WeaklyPrecedes.prems by simp
      from this obtain Γk Ψk Φk k
        where fp:‹(((⌈#≤ K₂ n, #≤ K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                                  ⊢ Ψ ▷ ((K₁ weakly precedes K₂) # Φ))
                              ↪^k (Γk, Suc n ⊢ Ψk ▷ Φk)›
          and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
      have ‹(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))›
        by (simp add: elims_part weakly_precedes_e)
      with fp relpowp_Suc_I2 have ‹(Γ, n ⊢ ((K₁ weakly precedes K₂) # Ψ) ▷ Φ)
                                       ↪^Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        by auto
      with rc show ?thesis by blast
    qed
    ultimately show ?case using WeaklyPrecedes.prems(2) by blast
next
  case (StrictlyPrecedes K₁ K₂)
    have ‹⟦ 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›
      using HeronConf_interp_stepwise_strictly_precedes_cases by simp
    moreover have ‹ϱ ∈ ⟦ ((⌈#≤ K₂ n, #< K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                          ⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ) ⟧config
          ⟹ (∃Γk Ψk Φk k. ((Γ, n ⊢ ((K₁ strictly precedes K₂) # Ψ) ▷ Φ)
                                  ↪^k (Γk, Suc n ⊢ Ψk ▷ Φk))
                ∧ (ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config))›
    proof -
      assume ‹ϱ ∈ ⟦ ((⌈#≤ K₂ n, #< K₁ n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                     ⊢ Ψ ▷ ((K₁ strictly precedes K₂) # Φ) ⟧config›
```

```
      hence ‹∃Γk Ψk Φk k. ((((⌈#≤ K2 n, #< K1 n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                             ⊢ Ψ ▹ ((K1 strictly precedes K2) # Φ))
                          ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk))
                         ∧ (ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config)›
        using  StrictlyPrecedes.prems by simp
      from this obtain Γk Ψk Φk k
        where fp:‹(((⌈#≤ K2 n, #< K1 n⌉ ∈ (λ(x, y). x ≤ y)) # Γ), n
                             ⊢ Ψ ▹ ((K1 strictly precedes K2) # Φ))
                          ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk)›
          and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config› by blast
      have ‹(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))›
        by (simp add: elims_part strictly_precedes_e)
      with fp relpowp_Suc_I2 have ‹(Γ, n ⊢ ((K1 strictly precedes K2) # Ψ) ▹ Φ)
                                  ↪^(Suc k) (Γk, Suc n ⊢ Ψk ▹ Φk)›
        by auto
      with rc show ?thesis by blast
    qed
    ultimately show ?case using StrictlyPrecedes.prems(2) by blast
next
  case (Kills K1 K2)
    have branches: ‹⟦ 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›
      using HeronConf_interp_stepwise_kills_cases by simp
    moreover have br1: ‹ϱ ∈ ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ) ⟧_config
              ⟹ ∃Γk Ψk Φk k. ((Γ, n ⊢ ((K1 kills K2) # Ψ) ▹ Φ)
                                  ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk))
                ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config›
    proof -
      assume h1: ‹ϱ ∈ ⟦ ((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ) ⟧_config›
      then have ‹∃Γk Ψk Φk k.
                  ((((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ))
                  ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk))
                ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config›
        using h1 Kills.prems by simp
      from this obtain Γk Ψk Φk k where
        fp:‹((((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ))
              ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk))›
        and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config› by blast
      have pc:‹(Γ, n ⊢ (K1 kills K2) # Ψ ▹ Φ)
                ↪ (((K1 ¬⇑ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ))›
        by (simp add: elims_part kills_e1)
      hence ‹(Γ, n ⊢ (K1 kills K2) # Ψ ▹ Φ) ↪^(Suc k) (Γk, Suc n ⊢ Ψk ▹ Φk)›
        using fp relpowp_Suc_I2 by auto
      with rc show ?thesis by blast
    qed
    moreover have br2:
      ‹ϱ ∈ ⟦ ((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ) ⟧_config
        ⟹ ∃Γk Ψk Φk k. ((Γ, n ⊢ ((K1 kills K2) # Ψ) ▹ Φ)
                               ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk))
                        ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config›
    proof -
      assume h2: ‹ϱ ∈ ⟦((K1 ⇑ n)#(K2 ¬⇑ ≥ n)#Γ), n ⊢ Ψ ▹ ((K1 kills K2)#Φ)⟧_config›
      then have ‹∃Γk Ψk Φk k. (
                  (((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▹ ((K1 kills K2) # Φ))
                    ↪^k (Γk, Suc n ⊢ Ψk ▹ Φk)
                  ) ∧ ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▹ Φk ⟧_config›
```

```
          using h2 Kills.prems by simp
        from this obtain Γk Ψk Φk k where
            fp:‹(((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K1 kills K2) # Φ))
                ↪k (Γk, Suc n ⊢ Ψk ▷ Φk)›
        and rc:‹ϱ ∈ ⟦ Γk, Suc n ⊢ Ψk ▷ Φk ⟧config› by blast
        have ‹(Γ, n ⊢ ((K1 kills K2) # Ψ) ▷ Φ)
              ↪ (((K1 ⇑ n) # (K2 ¬⇑ ≥ n) # Γ), n ⊢ Ψ ▷ ((K1 kills K2) # Φ))›
          by (simp add: elims_part kills_e2)
        hence ‹(Γ, n ⊢ ((K1 kills K2) # Ψ) ▷ Φ) ↪Suc k (Γk, Suc n ⊢ Ψk ▷ Φk)›
          using fp relpowp_Suc_I2 by auto
        with rc show ?thesis by blast
      qed
      ultimately show ?case using Kills.prems(2) by blast
  qed
qed

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 -
  obtain δk where diff: ‹nk = δk + Suc n›
    using add.commute assms(1) less_iff_Suc_add by auto
  show ?thesis
    proof (subst diff, subst diff, insert assms(2), induct δk)
      case 0  thus ?case
        using instant_index_increase assms(2) by simp
    next
      case (Suc δk)
        have f0: ‹ϱ ∈ ⟦ Γ, n ⊢ Ψ ▷ Φ ⟧config ⟹ ∃Γk Ψk Φk k.
                  ((Γ, n ⊢ Ψ ▷ Φ) ↪k (Γk, δk + Suc n ⊢ Ψk ▷ Φk))
                ∧ ϱ ∈ ⟦ Γk, δk + Suc n ⊢ Ψk ▷ Φk ⟧config›
          using Suc.hyps by blast
        obtain Γk Ψk Φk k
          where cont: ‹((Γ, n ⊢ Ψ ▷ Φ) ↪k (Γk, δk + Suc n ⊢ Ψk ▷ Φk))
                      ∧ ϱ ∈ ⟦ Γk, δk + Suc n ⊢ Ψk ▷ Φk ⟧config›
          using f0 assms(1) Suc.prems by blast
        then have fcontinue: ‹∃Γk' Ψk' Φk' k'. ((Γk, δk + Suc n ⊢ Ψk ▷ Φk)
                                  ↪k' (Γk', Suc (δk + Suc n) ⊢ Ψk' ▷ Φk'))
                              ∧ ϱ ∈ ⟦ Γk', Suc (δk + Suc n) ⊢ Ψk' ▷ Φk' ⟧config›
          using f0 cont instant_index_increase by blast
        obtain Γk' Ψk' Φk' k'
          where cont2: ‹((Γk, δk + Suc n ⊢ Ψk ▷ Φk)
                        ↪k' (Γk', Suc (δk + Suc n) ⊢ Ψk' ▷ Φk'))
                      ∧ ϱ ∈ ⟦ Γk', Suc (δk + Suc n) ⊢ Ψk' ▷ Φk' ⟧config›
          using Suc.prems using fcontinue cont by blast
        have trans: ‹(Γ, n ⊢ Ψ ▷ Φ) ↪k + k' (Γk', Suc (δk + Suc n) ⊢ Ψk' ▷ Φk')›
          using operational_semantics_trans_generalized cont cont2 by blast
        moreover have suc_assoc: ‹Suc δk + Suc n = Suc (δk + Suc n)› by arith
        ultimately show ?case
          proof (subst suc_assoc)
            show ‹∃Γk Ψk Φk k.
                  ((Γ, n ⊢ Ψ ▷ Φ) ↪k (Γk, Suc (δk + Suc n) ⊢ Ψk ▷ Φk))
                ∧ ϱ ∈ ⟦ Γk, Suc δk + Suc n ⊢ Ψk ▷ Φk ⟧config›
            using cont2 local.trans by auto
          qed
    qed
qed
```

Any run that belongs to a specification Ψ has a corresponding configuration that develops it up to the n^{th} 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 -
  have 1:‹∃Γk Ψk Φk k. (([], 0 ⊢ Ψ ▷ []) ↪^k (Γk, 0 ⊢ Ψk ▷ Φk))
                      ∧ ϱ ∈ ⟦ Γk, 0 ⊢ Ψk ▷ Φk ⟧config›
    using assms relpowp_0_I solve_start by fastforce
  show ?thesis
  proof (cases ‹n = 0›)
    case True
      thus ?thesis using assms relpowp_0_I solve_start by fastforce
  next
    case False hence pos:‹n > 0› by simp
      from assms solve_start have ‹ϱ ∈ ⟦ [], 0 ⊢ Ψ ▷ [] ⟧config › by blast
      from instant_index_increase_generalized[OF pos this] show ?thesis by blast
  qed
qed
```

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 elimation_rules_strictly_decreasing:
  assumes ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1)  ↪e  (Γ2, n2 ⊢ Ψ2 ▷ Φ2)›
    shows ‹μ Ψ1 > μ Ψ2›
using assms by (auto elim: operational_semantics_elim.cases)

lemma elimation_rules_strictly_decreasing_meas:
  assumes ‹(Γ1, n1 ⊢ Ψ1 ▷ Φ1)  ↪e  (Γ2, n2 ⊢ Ψ2 ▷ Φ2)›
    shows ‹(Ψ2, Ψ1) ∈ measure μ›
using assms by (auto elim: operational_semantics_elim.cases)

lemma elimation_rules_strictly_decreasing_meas':
  assumes ‹S1  ↪e  S2›
  shows ‹(S2, S1) ∈ measure μconfig›
proof -
  from assms obtain Γ1 n1 Ψ1 Φ1 where p1:‹S1 = (Γ1, n1 ⊢ Ψ1 ▷ Φ1)›
    using measure_interpretation_config.cases by blast
  from assms obtain Γ2 n2 Ψ2 Φ2 where p2:‹S2 = (Γ2, n2 ⊢ Ψ2 ▷ Φ2)›
```

```
    using measure_interpretation_config.cases by blast
  from elimation_rules_strictly_decreasing_meas assms p1 p2
    have ‹(Ψ₂, Ψ₁) ∈ measure μ› by blast
  hence ‹μ Ψ₂ < μ Ψ₁› by simp
  hence ‹μ_config (Γ₂, n₂ ⊢ Ψ₂ ▷ Φ₂) < μ_config (Γ₁, n₁ ⊢ Ψ₁ ▷ Φ₁)› by simp
  with p1 p2 show ?thesis by simp
qed
```

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 (simp add: wfp_def)
  show ‹wf {((𝒮₁::'a::linordered_field config), 𝒮₂). 𝒮₁ ↪ₑ← 𝒮₂}›
  proof (rule wf_subset)
    have ‹measure μ_config = {(𝒮₂, (𝒮₁::'a::linordered_field config)).
                              μ_config 𝒮₂ < μ_config 𝒮₁}›
      by (simp add: inv_image_def less_eq measure_def)
    thus ‹{((𝒮₁::'a::linordered_field config), 𝒮₂). 𝒮₁ ↪ₑ← 𝒮₂} ⊆ (measure μ_config)›
      using elimation_rules_strictly_decreasing_meas'
            operational_semantics_elim_inv_def by blast
  next
    show ‹wf (measure measure_interpretation_config)› by simp
  qed
qed


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))))
```

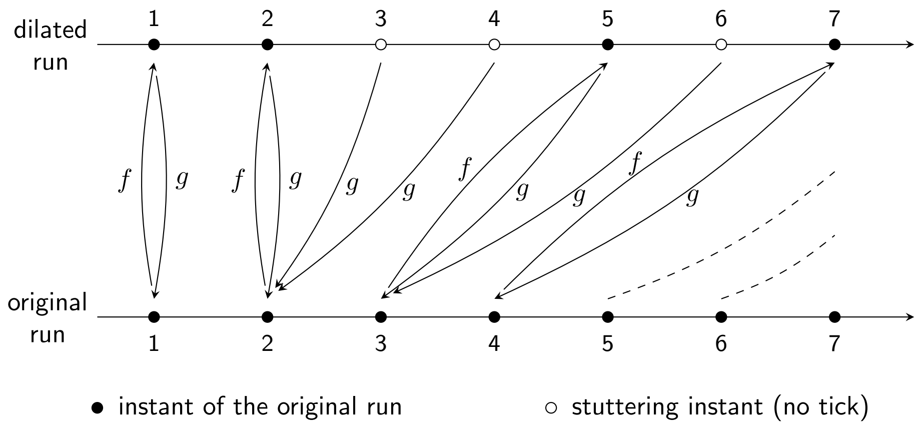

Figure 8.1: Dilating and contracting functions

```
  ∧ ((∄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 (induction k)
  case 0
    with assms(1)[of 0] show ?case by simp
next
  case (Suc k')
    with assms[of ‹Suc k'›] show ?case by force
qed
```

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›
using assms dilating_fun_def strict_mono_imp_inj_on by blast

lemma dilating_injects:
  assumes ‹dilating f sub r›
  shows   ‹inj_on f A›
using assms dilating_def dilating_fun_injects by blast
```

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›
using dilating_fun_def assms by blast

lemma ticks_image_sub:
  assumes ‹dilating f sub r›
  and     ‹hamlet ((Rep_run r) n c)›
  shows   ‹∃n₀. f n₀ = n›
using assms dilating_def ticks_image by blast

lemma ticks_image_sub':
  assumes ‹dilating f sub r›
  and     ‹∃c. hamlet ((Rep_run r) n c)›
  shows   ‹∃n₀. f n₀ = n›
using ticks_image_sub[OF assms(1)] assms(2) by blast
```

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
  { fix k assume h:‹k ∈ ?IMG›
    from h obtain k0 where k0prop:‹f k0 = k ∧ hamlet ((Rep_run r) (f k0) c)›
      using ticks_image[OF assms] by blast
    with h have ‹k ∈ image f ?SET›
      using assms dilating_fun_def strict_mono_less by blast
  } thus ‹?IMG ⊆ image f ?SET› ..
next
  { fix k assume h:‹k ∈ image f ?SET›
    from h obtain k0 where k0prop:‹k = f k0 ∧ k0 ∈ ?SET› by blast
    hence ‹k ∈ ?IMG› using assms by (simp add: dilating_fun_def strict_mono_less)
  } thus ‹image f ?SET ⊆ ?IMG› ..
qed

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
  { fix k assume h:‹k ∈ ?IMG›
    from h obtain k0 where k0prop:‹f k0 = k ∧ hamlet ((Rep_run r) (f k0) c)›
      using ticks_image[OF assms] by blast
    with h have ‹k ∈ image f ?SET›
      using assms dilating_fun_def strict_mono_less strict_mono_less_eq by fastforce
  } thus ‹?IMG ⊆ image f ?SET› ..
next
  { fix k assume h:‹k ∈ image f ?SET›
    from h obtain k0 where k0prop:‹k = f k0 ∧ k0 ∈ ?SET› by blast
    hence ‹k ∈ ?IMG›
      using assms dilating_fun_def strict_mono_less strict_mono_less_eq by fastforce
  } thus ‹image f ?SET ⊆ ?IMG› ..
qed

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
  { fix k assume h:‹k ∈ ?IMG›
    from h obtain k0 where k0prop:‹f k0 = k ∧ hamlet ((Rep_run r) (f k0) c)›
      using ticks_image[OF assms] by blast
    with h have ‹k ∈ image f ?SET›
      using assms dilating_fun_def strict_mono_less strict_mono_less_eq by fastforce
  } thus ‹?IMG ⊆ image f ?SET› ..
next
  { fix k assume h:‹k ∈ image f ?SET›
    from h obtain k0 where k0prop:‹k = f k0 ∧ k0 ∈ ?SET› by blast
    hence ‹k ∈ ?IMG›
      using assms dilating_fun_def strict_mono_less strict_mono_less_eq by fastforce
```

```
  } thus ‹image f ?SET ⊆ ?IMG› ..
qed

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
  { fix k assume h:‹k ∈ ?IMG›
    from h obtain k₀ where k0prop:‹f k₀ = k ∧ hamlet ((Rep_run r) (f k₀) c)›
      using ticks_image[OF assms] by blast
    with h have ‹k ∈ image f ?SET›
      using assms dilating_fun_def strict_mono_less_eq by blast
  } thus ‹?IMG ⊆ image f ?SET› ..
next
  { fix k assume h:‹k ∈ image f ?SET›
    from h obtain k₀ where k0prop:‹k = f k₀ ∧ k₀ ∈ ?SET› by blast
    hence ‹k ∈ ?IMG› using assms by (simp add: dilating_fun_def strict_mono_less_eq)
  } thus ‹image f ?SET ⊆ ?IMG› ..
qed
```

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 -
  from dilating_fun_injects[OF assms] have ‹inj_on f ?SET› .
  moreover have ‹finite ?SET› by simp
  from inj_on_iff_eq_card[OF this] calculation
    have ‹card (image f ?SET) = card ?SET› by blast
  moreover from dilating_fun_image_strict[OF assms] have ‹?IMG = image f ?SET› .
  ultimately show ?thesis by auto
qed

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 -
  from dilating_fun_injects[OF assms] have ‹inj_on f ?SET› .
  moreover have ‹finite ?SET› by simp
  from inj_on_iff_eq_card[OF this] calculation
    have ‹card (image f ?SET) = card ?SET› by blast
  moreover from dilating_fun_image_left[OF assms] have ‹?IMG = image f ?SET› .
  ultimately show ?thesis by auto
qed

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 -
  from dilating_fun_injects[OF assms] have ‹inj_on f ?SET› .
```

```
  moreover have ‹finite ?SET› by simp
  from inj_on_iff_eq_card[OF this] calculation
    have ‹card (image f ?SET) = card ?SET› by blast
  moreover from dilating_fun_image_right[OF assms] have ‹?IMG = image f ?SET› .
  ultimately show ?thesis by auto
qed

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 -
  from dilating_fun_injects[OF assms] have ‹inj_on f ?SET› .
  moreover have ‹finite ?SET› by simp
  from inj_on_iff_eq_card[OF this] calculation
    have ‹card (image f ?SET) = card ?SET› by blast
  moreover from dilating_fun_image[OF assms] have ‹?IMG = image f ?SET› .
  ultimately show ?thesis by auto
qed
```

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 -
  from ticks_image_sub'[OF assms(1,2)] have ‹∃k₀. f k₀ = k› .
  from this obtain k₀ where ‹f k₀ = k› by blast
  moreover with assms(1,3) have ‹time ((Rep_run sub) k₀ c) = τ›
    by (simp add: dilating_def)
  ultimately show ?thesis by blast
qed
```

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)›
using assms by (simp add: dilating_def)

lemma no_tick_sub:
  assumes ‹dilating f sub r›
  shows   ‹(∄n₀. f n₀ = n) ⟶ ¬hamlet ((Rep_run r) n a)›
using assms dilating_def dilating_fun_def by blast
```

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
  { fix k assume h:‹k ∈ ?R›
    with no_tick_sub[OF assms] have ‹∃k₀. f k₀ = k› by blast
    from this obtain k₀ where k0prop:‹f k₀ = k› by blast
    with ticks_sub[OF assms] h have ‹hamlet ((Rep_run sub) k₀ c)› by blast
    with k0prop have ‹k ∈ image f ?S› by blast
  }
  thus ‹?R ⊆ image f ?S› by blast
next
  { fix k assume h:‹k ∈ image f ?S›
    from this obtain k₀ where ‹f k₀ = k ∧ hamlet ((Rep_run sub) k₀ c)› by blast
    with assms have ‹k ∈ ?R› using ticks_sub by blast
  }
  thus ‹image f ?S ⊆ ?R› by blast
qed
```

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)›
using Least_mono[OF strict_mono_mono, OF assms] .
```

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

```
lemma Least_nat_ex:
  ‹(n::nat) ∈ S ⟹ ∃x ∈ S. (∀y ∈ S. x ≤ y)›
by (induction n rule: nat_less_induct, insert not_le_imp_less, blast)
```

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 -
  from assms(2) have ‹∃x. x ∈ ?S› by simp
  hence least:‹∃x ∈ ?S. ∀y ∈ ?S. x ≤ y›
    using Least_nat_ex ..
  from assms(1) have ‹strict_mono f› by (simp add: dilating_def dilating_fun_def)
  from Least_strict_mono[OF this least] have
    ‹(LEAST y. y ∈ f ‘ ?S)  = f (LEAST x. x ∈ ?S)› .
  with tick_set_sub[OF assms(1), of ‹c›] show ?thesis by auto
qed
```

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 -
  from assms(2) obtain k where ‹hamlet ((Rep_run r) k c)› by blast
  with ticks_image_sub[OF assms(1)] ticks_sub[OF assms(1)] show ?thesis by blast
qed
```

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 -
  from no_tick_sub[OF assms(1)] assms(2) have ‹∃k0. f k0 = k› by blast
  from this obtain k0 where ‹f k0 = k› by blast
  moreover with ticks_sub[OF assms(1)] assms(2)
    have ‹hamlet ((Rep_run sub) k0 c)› by blast
  ultimately show ?thesis by blast
qed
```

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
  { fix i assume h:‹i ∈ ?SUB›
    hence ‹m < i ∧ i < n› by simp
    hence ‹f m < f i ∧ f i < (f n)› using assms
      by (simp add: dilating_def dilating_fun_def strict_monoD strict_mono_less_eq)
    moreover from h have ‹hamlet ((Rep_run sub) i c)› by simp
    hence ‹hamlet ((Rep_run r) (f i) c)› using ticks_sub[OF assms] by blast
    ultimately have ‹f i ∈ ?RUN› by simp
  } thus ‹image f ?SUB ⊆ ?RUN› by blast
next
  { fix i assume h:‹i ∈ ?RUN›
    hence ‹hamlet ((Rep_run r) i c)› by simp
    from ticks_imp_ticks_subk[OF assms this]
      obtain i0 where i0prop:‹f i0 = i ∧ hamlet ((Rep_run sub) i0 c)› by blast
    with h have ‹f m < f i0 ∧ f i0 < f n› by simp
    moreover have ‹strict_mono f› using assms dilating_def dilating_fun_def by blast
    ultimately have ‹m < i0 ∧ i0 < n›
      using strict_mono_less strict_mono_less_eq by blast
    with i0prop have ‹∃i0. f i0 = i ∧ i0 ∈ ?SUB› by blast
  } thus ‹?RUN ⊆ image f ?SUB› by blast
qed

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
  { fix i assume h:‹i ∈ ?SUB›
    hence ‹m ≤ i ∧ i < n› by simp
    hence ‹f m ≤ f i ∧ f i < (f n)› using assms
      by (simp add: dilating_def dilating_fun_def strict_monoD strict_mono_less_eq)
    moreover from h have ‹hamlet ((Rep_run sub) i c)› by simp
    hence ‹hamlet ((Rep_run r) (f i) c)› using ticks_sub[OF assms] by blast
    ultimately have ‹f i ∈ ?RUN› by simp
  } thus ‹image f ?SUB ⊆ ?RUN› by blast
next
  { fix i assume h:‹i ∈ ?RUN›
    hence ‹hamlet ((Rep_run r) i c)› by simp
    from ticks_imp_ticks_subk[OF assms this]
```

```
    obtain i0 where i0prop:‹f i0 = i ∧ hamlet ((Rep_run sub) i0 c)› by blast
   with h have ‹f m ≤ f i0 ∧ f i0 < f n› by simp
   moreover have ‹strict_mono f› using assms dilating_def dilating_fun_def by blast
   ultimately have ‹m ≤ i0 ∧ i0 < n›
    using strict_mono_less strict_mono_less_eq by blast
   with i0prop have ‹∃i0. f i0 = i ∧ i0 ∈ ?SUB› by blast
  } thus ‹?RUN ⊆ image f ?SUB› by blast
qed

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
  { fix i  assume h:‹i ∈ ?SUB›
    hence ‹m < i ∧ i ≤ n› by simp
    hence ‹f m < f i ∧ f i ≤ (f n)› using assms
      by (simp add: dilating_def dilating_fun_def strict_monoD strict_mono_less_eq)
    moreover from h have ‹hamlet ((Rep_run sub) i c)› by simp
    hence ‹hamlet ((Rep_run r) (f i) c)› using ticks_sub[OF assms] by blast
    ultimately have ‹f i ∈ ?RUN› by simp
  } thus ‹image f ?SUB ⊆ ?RUN› by blast
next
  { fix i assume h:‹i ∈ ?RUN›
    hence ‹hamlet ((Rep_run r) i c)› by simp
    from ticks_imp_ticks_subk[OF assms this]
      obtain i0 where i0prop:‹f i0 = i ∧ hamlet ((Rep_run sub) i0 c)› by blast
    with h have ‹f m < f i0 ∧ f i0 ≤ f n› by simp
    moreover have ‹strict_mono f› using assms dilating_def dilating_fun_def by blast
    ultimately have ‹m < i0 ∧ i0 ≤ n›
      using strict_mono_less strict_mono_less_eq by blast
    with i0prop have ‹∃i0. f i0 = i ∧ i0 ∈ ?SUB› by blast
  } thus ‹?RUN ⊆ image f ?SUB› by blast
qed

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
  { fix i assume h:‹i ∈ ?SUB›
    hence ‹m ≤ i ∧ i ≤ n› by simp
    hence ‹f m ≤ f i ∧ f i ≤ (f n)›
      using assms by (simp add: dilating_def dilating_fun_def strict_mono_less_eq)
    moreover from h have ‹hamlet ((Rep_run sub) i c)› by simp
    hence ‹hamlet ((Rep_run r) (f i) c)› using ticks_sub[OF assms] by blast
    ultimately have ‹f i ∈?RUN› by simp
  } thus ‹image f ?SUB ⊆ ?RUN› by blast
next
  { fix i assume h:‹i ∈ ?RUN›
    hence ‹hamlet ((Rep_run r) i c)› by simp
    from ticks_imp_ticks_subk[OF assms this]
      obtain i0 where i0prop:‹f i0 = i ∧ hamlet ((Rep_run sub) i0 c)› by blast
    with h have ‹f m ≤ f i0 ∧ f i0 ≤ f n› by simp
    moreover have ‹strict_mono f› using assms dilating_def dilating_fun_def by blast
    ultimately have ‹m ≤ i0 ∧ i0 ≤ n› using strict_mono_less_eq by blast
    with i0prop have ‹∃i0. f i0 = i ∧ i0 ∈ ?SUB› by blast
```

```
  } thus ‹?RUN ⊆ image f ?SUB› by blast
qed
```

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 -
  from assms have False by (simp add: dilating_def dilating_fun_def)
  thus ?thesis ..
qed

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 -
  from assms have ‹strict_mono f› by (simp add: dilating_def dilating_fun_def)
  hence ‹f 0 ≤ f n› unfolding strict_mono_def by (simp add: less_mono_imp_le_mono)
  hence ‹∀i. i ≤ f n = (i < f 0) ∨ (f 0 ≤ i ∧ i ≤ f n)› by auto
  hence ‹{i. i ≤ f n ∧ hamlet ((Rep_run r) i c)}
       = {i. i < f 0 ∧ hamlet ((Rep_run r) i c)}
       ∪ {i. f 0 ≤ i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}›
    by auto
  also have ‹... = {i. f 0 ≤ i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}›
     using empty_dilated_prefix[OF assms] by blast
  finally show ?thesis by simp
qed

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 -
  have ‹{i. 0 ≤ i ∧ i ≤ f n ∧ hamlet ((Rep_run r) i c)}
       = image f {i. 0 ≤ i ∧ i ≤ n ∧ hamlet ((Rep_run sub) i c)}›
    using dilated_ticks[OF assms] empty_dilated_prefix'[OF assms] by blast
  thus ?thesis by simp
qed

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 -
  from assms have dil:‹dilating_fun f r› unfolding dilating_def by simp
  from dil have f0:‹f 0 = 0›  using dilating_fun_def by blast
  from dilating_fun_image_left[OF dil, of ‹0› ‹n› ‹c›]
  have ‹{i. f 0 ≤ i ∧ i < f n ∧ hamlet ((Rep_run r) i c)}
       = image f {i. 0 ≤ i ∧ i < n ∧ hamlet ((Rep_run r) (f i) c)}› .
  hence ‹{i. i < f n ∧ hamlet ((Rep_run r) i c)}
       = image f {i. i < n ∧ hamlet ((Rep_run r) (f i) c)}›
    using f0 by simp
  also have ‹... = image f {i. i < n ∧ hamlet ((Rep_run sub) i c)}›
    using assms dilating_def by blast
  finally show ?thesis by simp
qed
```

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)}›
by auto
```

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 (induction n)
  case 0
    have ‹tick_count r c 0 = card {i. i ≤ 0 ∧ hamlet ((Rep_run r) i c)}›
      by (simp add: tick_count_def)
    also have ‹... = card {i::nat. i = 0 ∧ hamlet ((Rep_run r) 0 c)}›
      using le_zero_eq nat_sing_prop[of ‹0› ‹λi. hamlet ((Rep_run r) i c)›] by simp
    also have ‹... = (if hamlet ((Rep_run r) 0 c) then 1 else 0)› by simp
    also have ‹... = run_tick_count r c 0› by simp
    finally show ?case .
next
  case (Suc k)
    show ?case
    proof (cases ‹hamlet ((Rep_run r) (Suc k) c)›)
      case True
        hence ‹{i. i ≤ Suc k ∧ hamlet ((Rep_run r) i c)}
             = insert (Suc k) {i. i ≤ k ∧ hamlet ((Rep_run r) i c)}› by auto
        hence ‹tick_count r c (Suc k) = Suc (tick_count r c k)›
          by (simp add: tick_count_def)
        with Suc.IH have ‹tick_count r c (Suc k) = Suc (run_tick_count r c k)› by simp
        thus ?thesis by (simp add: True)
    next
      case False
        hence ‹{i. i ≤ Suc k ∧ hamlet ((Rep_run r) i c)}
             = {i. i ≤ k ∧ hamlet ((Rep_run r) i c)}›
          using le_Suc_eq by auto
        hence ‹tick_count r c (Suc k) = tick_count r c k›
          by (simp add: tick_count_def)
        thus ?thesis using Suc.IH by (simp add: False)
    qed
qed
```

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›
  unfolding tick_count_def tick_count_strict_def using less_Suc_eq_le by auto

lemma tick_count_strict_is_fun[code]:
  ‹tick_count_strict r c n = run_tick_count_strictly r c n›
proof (cases ‹n = 0›)
  case True
    hence ‹tick_count_strict r c n = 0› unfolding tick_count_strict_def by simp
    also have ‹... = run_tick_count_strictly r c 0›
      using run_tick_count_strictly.simps(1)[symmetric] .
    finally show ?thesis using True by simp
next
  case False
    from not0_implies_Suc[OF this] obtain m where *:‹n = Suc m› by blast
    hence ‹tick_count_strict r c n = tick_count r c m›
      using tick_count_strict_suc by simp
    also have ‹... = run_tick_count r c m› using tick_count_is_fun[of ‹r› ‹c› ‹m›] .
```

```
    also have ‹... = run_tick_count_strictly r c (Suc m)›
      using run_tick_count_strictly.simps(2)[symmetric] .
    finally show ?thesis using * by simp
qed
```

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) }›
by auto
```

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›
  using assms neq0_conv by blast

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
  { fix r::‹'a run›
    assume ‹r ∈ ?P›
    hence ‹∀n::nat. (run_tick_count r K2 n) ≤ (run_tick_count_strictly r K1 n)›
      by simp
    hence 1:‹∀n::nat. (tick_count r K2 n) ≤ (tick_count_strict r K1 n)›
      using tick_count_is_fun[symmetric, of r] tick_count_strict_is_fun[symmetric, of r]
      by simp
    hence ‹∀n::nat. (tick_count_strict r K2 (Suc n)) ≤ (tick_count_strict r K1 n)›
      using tick_count_strict_suc[symmetric, of ‹r› ‹K2›] by simp
    hence ‹∀n::nat. (tick_count_strict r K2 (Suc (Suc n))) ≤ (tick_count_strict r K1 (Suc n))›
      by simp
    hence ‹∀n::nat. (tick_count r K2 (Suc n)) ≤ (tick_count r K1 n)›
      using tick_count_strict_suc[symmetric, of ‹r›] by simp
    hence *:‹∀n::nat. (run_tick_count r K2 (Suc n)) ≤ (run_tick_count r K1 n)›
      by (simp add: tick_count_is_fun)
    from 1 have ‹tick_count r K2 0 <= tick_count_strict r K1 0› by simp
    moreover have ‹tick_count_strict r K1 0 = 0› unfolding tick_count_strict_def by simp
    ultimately have ‹tick_count r K2 0 = 0› by simp
    hence ‹¬hamlet ((Rep_run r) 0 K2)› unfolding tick_count_def by auto
    with * have ‹r ∈ ?P'› by simp
  } thus ‹?P ⊆ ?P'› ..
  { fix r::‹'a run›
    assume h:‹r ∈ ?P'›
    hence ‹∀n::nat. (run_tick_count r K2 (Suc n)) ≤ (run_tick_count r K1 n)› by simp
    hence ‹∀n::nat. (tick_count r K2 (Suc n)) ≤ (tick_count r K1 n)›
      by (simp add: tick_count_is_fun)
    hence ‹∀n::nat. (tick_count r K2 (Suc n)) ≤ (tick_count_strict r K1 (Suc n))›
      using tick_count_strict_suc[symmetric, of ‹r› ‹K1›] by simp
    hence *:‹∀n. n > 0 ⟶ (tick_count r K2 n) ≤ (tick_count_strict r K1 n)›
      using gr0_implies_Suc by blast
    have ‹tick_count_strict r K1 0 = 0› unfolding tick_count_strict_def by simp
    moreover from h have ‹¬hamlet ((Rep_run r) 0 K2)› by simp
    hence ‹tick_count r K2 0 = 0› unfolding tick_count_def by auto
```

```
    ultimately have ‹tick_count r K₂ 0 ≤ tick_count_strict r K₁ 0› by simp
    from zero_gt_all[of ‹λn. tick_count r K₂ n ≤ tick_count_strict r K₁ n›, OF this ] *
      have ‹∀n. (tick_count r K₂ n) ≤ (tick_count_strict r K₁ n)› by simp
    hence ‹∀n. (run_tick_count r K₂ n) ≤ (run_tick_count_strictly r K₁ n)›
      by (simp add: tick_count_is_fun tick_count_strict_is_fun)
    hence ‹r ∈ ?P› ..
  } thus ‹?P' ⊆ ?P› ..
qed
```

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)›
by simp

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)›
by (simp add: tick_count_is_fun)
```

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 -
  have ‹{i. i ≤ n ∧ P i} ∩ {i. i = (Suc n) ∧ P i} = {}› by auto
  moreover have ‹{i. i ≤ n ∧ P i} ∪ {i. i = (Suc n) ∧ P i}
              = {i. i ≤ (Suc n) ∧ P i}› by auto
  moreover have ‹finite {i. i ≤ n ∧ P i}› by simp
  moreover have ‹finite {i. i = (Suc n) ∧ P i}› by simp
  ultimately show ?thesis
    using card_Un_disjoint[of ‹{i. i ≤ n ∧ P i}› ‹{i. i = Suc n ∧ P i}›] by simp
qed

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 -
  have ‹{i::nat. m < i ∧ i < n ∧ P i} ∩ {i. i = n ∧ P i} = {}› by auto
  moreover with assms have
    ‹{i::nat. m < i ∧ i < n ∧ P i} ∪ {i. i = n ∧ P i} = {i. m < i ∧ i ≤ n ∧ P i}›
  by auto
  moreover have ‹finite {i. m < i ∧ i < n ∧ P i}› by simp
  moreover have ‹finite {i. i = n ∧ P i}› by simp
  ultimately show ?thesis
    using card_Un_disjoint[of ‹{i. m < i ∧ i < n ∧ P i}› ‹{i. i = n ∧ P i}›] by simp
qed

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 -
  have ‹{i::nat. i < n ∧ P i} ∩ {i. i = n ∧ P i} = {}› by auto
  moreover have ‹{i. i < n ∧ P i} ∪ {i. i = n ∧ P i} = {i. i ≤ n ∧ P i}› by auto
  moreover have ‹finite {i. i < n ∧ P i}› by simp
  moreover have ‹finite {i. i = n ∧ P i}› by simp
  ultimately show ?thesis
```

```
    using card_Un_disjoint[of ‹{i. i < n ∧ P i}› ‹{i. i = n ∧ P i}›] by simp
qed

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 -
  have 1:‹{i::nat. i ≤ m ∧ P i} ∩ {i. m < i ∧ i < n ∧ P i} = {}› by auto
  from assms have ‹∀i::nat. i < n = (i ≤ m) ∨ (m < i ∧ i < n)›
    using less_trans by auto
  hence 2:
    ‹{i::nat. i < n ∧ P i} = {i. i ≤ m ∧ P i} ∪ {i. m < i ∧ i < n ∧ P i}› by blast
  have 3:‹finite {i. i ≤ m ∧ P i}› by simp
  have 4:‹finite {i. m < i ∧ i < n ∧ P i}› by simp
  from card_Un_disjoint[OF 3 4 1] 2 show ?thesis by simp
qed

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 -
  have 1:‹{i::nat. i < m ∧ P i} ∩ {i. m ≤ i ∧ i < n ∧ P i} = {}› by auto
  from assms have ‹∀i::nat. i < n = (i < m) ∨ (m ≤ i ∧ i < n)›
    using less_trans by auto
  hence 2:
    ‹{i::nat. i < n ∧ P i} = {i. i < m ∧ P i} ∪ {i. m ≤ i ∧ i < n ∧ P i}› by blast
  have 3:‹finite {i. i < m ∧ P i}› by simp
  have 4:‹finite {i. m ≤ i ∧ i < n ∧ P i}› by simp
  from card_Un_disjoint[OF 3 4 1] 2 show ?thesis by simp
qed

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}›
using assms le_cases nat_less_le by auto

lemma card_sing_prop:‹card {i. i = n ∧ P i} = (if P n then 1 else 0)›
proof (cases ‹P n›)
  case True
    hence ‹{i. i = n ∧ P i} = {n}› by (simp add: Collect_conv_if)
    with ‹P n› show ?thesis by simp
next
  case False
    hence ‹{i. i = n ∧ P i} = {}› by (simp add: Collect_conv_if)
    with ‹¬P n› show ?thesis by simp
qed

lemma card_prop_mono:
  assumes ‹m ≤ n›
    shows ‹card {i::nat. i ≤ m ∧ P i} ≤ card {i. i ≤ n ∧ P i}›
proof -
  from assms have ‹{i. i ≤ m ∧ P i} ⊆ {i. i ≤ n ∧ P i}› by auto
  moreover have ‹finite {i. i ≤ n ∧ P i}› by simp
  ultimately show ?thesis by (simp add: card_mono)
qed
```

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 assms(1) have smf:‹strict_mono f› by (simp add: dilating_def dilating_fun_def)
  { fix k assume h:‹f n < k ∧ k < f (Suc n) ∧ hamlet ((Rep_run r) k c)›
    hence ‹∃k0. f k0 = k› using assms(1) dilating_def dilating_fun_def by blast
    from this obtain k0 where ‹f k0 = k› by blast
    with h have ‹f n < f k0 ∧ f k0 < f (Suc n)› by simp
    hence False using smf not_less_eq strict_mono_less by blast
  } thus ?thesis using assms(2) by blast
qed
```

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 -
  have smf:‹strict_mono f› using assms dilating_def dilating_fun_def by blast
  moreover have ‹finite {k. k ≤ f n ∧ hamlet ((Rep_run r) k c)}› by simp
  moreover have *:‹finite {k. f n < k ∧ k ≤ f (Suc n) ∧ hamlet ((Rep_run r) k c)}› by simp
  ultimately have ‹{k. k ≤ f (Suc n) ∧ hamlet ((Rep_run r) k c)} =
                      {k. k ≤ f n ∧ hamlet ((Rep_run r) k c)}
                    ∪ {k. f n < k ∧ k ≤ f (Suc n) ∧ hamlet ((Rep_run r) k c)}›
    by (simp add: nat_interval_union strict_mono_less_eq)
  moreover have ‹{k. k ≤ f n ∧ hamlet ((Rep_run r) k c)}
                ∩ {k. f n < k ∧ k ≤ f (Suc n) ∧ hamlet ((Rep_run r) k c)} = {}›
     by auto
  ultimately have ‹card {k. k ≤ f (Suc n) ∧ hamlet (Rep_run r k c)} =
                    card {k. k ≤ f n ∧ hamlet (Rep_run r k c)}
                  + card {k. f n < k ∧ k ≤ f (Suc n) ∧ hamlet (Rep_run r k c)}›
    by (simp add: * card_Un_disjoint)
  moreover from no_tick_before_suc[OF assms] have
    ‹{k. f n < k ∧ k ≤ f (Suc n) ∧ hamlet ((Rep_run r) k c)} =
          {k. k = f (Suc n) ∧ hamlet ((Rep_run r) k c)}›
    using smf strict_mono_less by fastforce
  ultimately show ?thesis by (simp add: tick_count_def)
qed

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)›
using tick_count_fsuc[OF assms]
      card_sing_prop[of ‹f (Suc n)› ‹λk. hamlet ((Rep_run r) k c)›] by simp

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))›
```

```
using tick_count_f_suc[OF assms] by simp

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))›
using tick_count_f_suc_suc[OF assms] assms by (simp add: dilating_def)
```

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 -
  have union:‹{i. i ≤ n ∧ hamlet ((Rep_run r) i c)} =
          {i. i ≤ f n_p ∧ hamlet ((Rep_run r) i c)}
        ∪ {i. f n_p < i ∧ i ≤ n ∧ hamlet ((Rep_run r) i c)}› using assms(2) by auto
  have partition: ‹{i. i ≤ f n_p ∧ hamlet ((Rep_run r) i c)}
        ∩ {i. f n_p < i ∧ i ≤ n ∧ hamlet ((Rep_run r) i c)} = {}›
    by (simp add: disjoint_iff_not_equal)
  from assms have ‹{i. f n_p < i ∧ i ≤ n ∧ hamlet ((Rep_run r) i c)} = {}›
    using no_tick_sub by fastforce
  with union and partition show ?thesis by (simp add: tick_count_def)
qed
```

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 -
  have ‹tick_count sub c n = card {i. i ≤ n ∧ hamlet ((Rep_run sub) i c)}›
    using tick_count_def[of ‹sub› ‹c› ‹n›] .
  also have ‹... = card (image f {i. i ≤ n ∧ hamlet ((Rep_run sub) i c)})›
    using assms dilating_def dilating_injects[OF assms] by (simp add: card_image)
  also have ‹... = card {i. i ≤ f n ∧ hamlet ((Rep_run r) i c)}›
    using dilated_prefix[OF assms, symmetric, of ‹n› ‹c›] by simp
  also have ‹... = tick_count r c (f n)›
    using tick_count_def[of ‹r› ‹c› ‹f n›] by simp
  finally show ?thesis .
qed

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 -
  have ‹run_tick_count sub c n = tick_count sub c n›
    using tick_count_is_fun[of ‹sub› c n, symmetric] .
  also from tick_count_sub[OF assms] have ‹... = tick_count r c (f n)› .
  also have ‹... = #≤ r c (f n)› using tick_count_is_fun[of r c ‹f n›] .
  finally show ?thesis .
qed
```

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 -
  from assms have ‹f 0 = 0› by (simp add: dilating_def dilating_fun_def)
  thus ?thesis unfolding tick_count_strict_def by simp
qed
```

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 -
  from assms(1) have smf:‹strict_mono f› by (simp add: dilating_def dilating_fun_def)
  from assms(2) have ‹f n < k› by simp
  hence ‹∀i. k ≤ i ⟶ f n < i› by simp
  with no_tick_before_suc[OF assms(1)] have
    *:‹∀i. k ≤ i ∧ i < f (Suc n) ⟶ ¬hamlet ((Rep_run r) i c)› by blast
  from tick_count_strict_def have
    ‹tick_count_strict r c (f (Suc n)) = card {i. i < f (Suc n) ∧ hamlet ((Rep_run r) i c)}› .
  also have
    ‹... = card {i. i < k ∧ hamlet ((Rep_run r) i c)}
        + card {i. k ≤ i ∧ i < f (Suc n) ∧ hamlet ((Rep_run r) i c)}›
    using card_mnm' assms(2) by simp
  also have ‹... = card {i. i < k ∧ hamlet ((Rep_run r) i c)}› using * by simp
  finally show ?thesis by (simp add: tick_count_strict_def)
qed
```

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 -
  have ‹tick_count_strict sub c n = card {i. i < n ∧ hamlet ((Rep_run sub) i c)}›
    using tick_count_strict_def[of ‹sub› ‹c› ‹n›] .
  also have ‹... = card (image f {i. i < n ∧ hamlet ((Rep_run sub) i c)})›
    using assms dilating_def dilating_injects[OF assms] by (simp add: card_image)
  also have ‹... = card {i. i < f n ∧ hamlet ((Rep_run r) i c)}›
    using dilated_strict_prefix[OF assms, symmetric, of ‹n› ‹c›] by simp
  also have ‹... = tick_count_strict r c (f n)›
    using tick_count_strict_def[of ‹r› ‹c› ‹f n›] by simp
  finally show ?thesis .
qed
```

The tick count on any clock can only increase.

```
lemma mono_tick_count:
  ‹mono (λ k. tick_count r c k)›
proof
  { fix x y::nat
    assume ‹x ≤ y›
    from card_prop_mono[OF this] have ‹tick_count r c x ≤ tick_count r c y›
      unfolding tick_count_def by simp
  } thus ‹⋀x y. x ≤ y ⟹ tick_count r c x ≤ tick_count r c y› .
qed
```

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 (induction n)
  case 0
    with assms have ‹f 0 = 0› by (simp add: dilating_def dilating_fun_def)
    thus ?case using "0.prems" by blast
next
  case (Suc n)
  show ?case
  proof (cases ‹∃n0. f n0 = n›)
    case True
      from this obtain n0 where ‹f n0 = n› by blast
      hence ‹f n0 < (Suc n) ∧ (∀k. f n0 < k ∧ k ≤ (Suc n) ⟶ (∄k0. f k0 = k))›
        using Suc.prems Suc_leI le_antisym by blast
      thus ?thesis by blast
  next
    case False
    from Suc.IH[OF this] obtain np
      where ‹f np < n ∧ (∀k. f np < k ∧ k ≤ n ⟶ (∄k0. f k0 = k))› by blast
    hence ‹f np < Suc n ∧ (∀k. f np < k ∧ k ≤ n ⟶ (∄k0. f k0 = k))› by simp
    with Suc(2) have ‹f np < (Suc n) ∧ (∀k. f np < k ∧ k ≤ (Suc n) ⟶ (∄k0. f k0 = k))›
      using le_Suc_eq by auto
    thus ?thesis by blast
  qed
qed
```

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 -
  from assms(2) have ‹f sn > f n› by simp
  with strict_mono_less[OF assms(1)] have ‹sn > n› by simp
  moreover have ‹sn ≤ Suc n›
  proof -
    { assume ‹sn > Suc n›
      from this obtain i where ‹n < i ∧ i < sn› by blast
      hence ‹f n < f i ∧ f i < f sn› using assms(1) by (simp add: strict_mono_def)
      with assms(2) have False by simp
    } thus ?thesis using not_less by blast
  qed
  ultimately show ?thesis by (simp add: Suc_leI)
qed
```

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 -
  from assms(1) have smf:‹strict_mono f› by (simp add: dilating_def dilating_fun_def)
  from assms(2) have *:‹∀k. f np < k ∧ k < Suc n ⟶ (∄k0. f k0 = k)› by simp
  from assms(2) have ‹f np < n› by simp
  with smf assms(3) have **:‹sn0 > np› using strict_mono_less by fastforce
  have ‹Suc n ≤ f (Suc np)›
```

```
  proof -
    { assume h:‹Suc n > f (Suc n_p)›
      hence ‹Suc n_p < sn0› using ** Suc_lessI assms(3) by fastforce
      hence ‹∃k. k > n_p ∧ f k < Suc n› using h by blast
      with * have False using smf strict_mono_less by blast
    } thus ?thesis using not_less by blast
  qed
  hence ‹sn0 ≤ Suc n_p› using assms(3) smf using strict_mono_less_eq by fastforce
  with ** show ?thesis by simp
qed
```

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 -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  show ?thesis
  proof (induction n)
    case 0
      from assms(2) have ‹run_tick_count sub a 0 ≤ run_tick_count sub b 0› ..
      with run_tick_count_sub[OF *, of _ 0] have
        ‹run_tick_count r a (f 0) ≤ run_tick_count r b (f 0)› by simp
      moreover from * have ‹f 0 = 0› by (simp add:dilating_def dilating_fun_def)
      ultimately show ?case by simp
  next
    case (Suc n') thus ?case
    proof (cases ‹∃n0. f n0 = Suc n'›)
      case True
        from this obtain n0 where fn0:‹f n0 = Suc n'› by blast
        show ?thesis
        proof (cases ‹hamlet ((Rep_run sub) n0 a)›)
          case True
            have ‹run_tick_count r a (f n0) ≤ run_tick_count r b (f n0)›
              using assms(2) run_tick_count_sub[OF *] by simp
            thus ?thesis by (simp add: fn0)
        next
          case False
            hence ‹¬ hamlet ((Rep_run r) (Suc n') a)›
              using * fn0 ticks_sub by fastforce
            thus ?thesis by (simp add: Suc.IH le_SucI)
        qed
    next
      case False
        thus ?thesis  using * Suc.IH no_tick_sub by fastforce
    qed
  qed
qed
```

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 -
  from assms have ‹∀k. k < m - (f n) ⟶ (∄k0. f k0 = Suc ((f n) + k))› by simp
```

```
  hence ‹∀k. k < m - (f n)
           ⟶ time ((Rep_run r) (Suc ((f n) + k)) c) = time ((Rep_run r) ((f n) + k) c)›
    using assms(1) by (simp add: dilating_def dilating_fun_def)
  hence *:‹∀k. k < m - (f n) ⟶ time ((Rep_run r) (Suc ((f n) + k)) c) = time ((Rep_run r) (f n)
c)›
    using bounded_suc_ind[of ‹m - (f n)› ‹λk. time (Rep_run r k c)› ‹f n›] by blast
  from assms(3) obtain m₀ where m0:‹Suc m₀ = m - (f n)› using Suc_diff_Suc by blast
  with * have ‹time ((Rep_run r) (Suc ((f n) + m₀)) c) = time ((Rep_run r) (f n) c)› by auto
  moreover from m0 have ‹Suc ((f n) + m₀) = m› by simp
  ultimately show ?thesis by simp
qed

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 -
  from assms(3) have ‹time ((Rep_run r) m c) = time ((Rep_run r) (f n) c)›
    using  stutter_no_time[OF assms(1,3,4)] by blast
  also from assms(1,2) have ‹time ((Rep_run r) (f n) c) = τ› by (simp add: dilating_def)
  finally show ?thesis .
qed
```

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
  assume ‹first_time sub c n t›
  with before_first_time[OF this]
    have *:‹time ((Rep_run sub) n c) = t ∧ (∀m < n. time((Rep_run sub) m c) < t)›
      by (simp add: first_time_def)
  moreover have ‹∀n c. time (Rep_run sub n c) = time (Rep_run r (f n) c)›
      using assms(1) by (simp add: dilating_def)
  ultimately have **:
    ‹time ((Rep_run r) (f n) c) = t ∧ (∀m < n. time((Rep_run r) (f m) c) < t)›
    by simp
  have ‹∀m < f n. time ((Rep_run r) m c) < t›
  proof -
  { fix m assume hyp:‹m < f n›
    have ‹time ((Rep_run r) m c) < t›
    proof (cases ‹∃m₀. f m₀ = m›)
      case True
        from this obtain m₀ where mm0:‹m = f m₀› by blast
        with hyp have m0n:‹m₀ < n› using assms(1)
          by (simp add: dilating_def dilating_fun_def strict_mono_less)
        hence ‹time ((Rep_run sub) m₀ c) < t› using * by blast
        thus ?thesis by (simp add: mm0 m0n **)
    next
      case False
        hence ‹∃mₚ. f mₚ < m ∧ (∀k. f mₚ < k ∧ k ≤ m ⟶ (∄k₀. f k₀ = k))›
          using greatest_prev_image[OF assms] by simp
        from this obtain mₚ where
          mp:‹f mₚ < m ∧ (∀k. f mₚ < k ∧ k ≤ m ⟶ (∄k₀. f k₀ = k))› by blast
        hence ‹time ((Rep_run r) m c) = time ((Rep_run sub) mₚ c)›
          using time_stuttering[OF assms] by blast
```

```
        also from hyp mp have ‹f m_p < f n› by linarith
        hence ‹m_p < n› using assms
          by (simp add:dilating_def dilating_fun_def strict_mono_less)
        hence ‹time ((Rep_run sub) m_p c) < t› using * by simp
        finally show ?thesis by simp
      qed
    } thus ?thesis by simp
  qed
  with ** show ‹first_time r c (f n) t› by (simp add: alt_first_time_def)
next
  assume ‹first_time r c (f n) t›
  hence *:‹time ((Rep_run r) (f n) c) = t ∧ (∀k < f n. time ((Rep_run r) k c) < t)›
    by (simp add: first_time_def before_first_time)
  hence ‹time ((Rep_run sub) n c) = t› using assms dilating_def by blast
  moreover from * have ‹(∀k < n. time ((Rep_run sub) k c) < t)›
    using assms dilating_def dilating_fun_def strict_monoD by fastforce
  ultimately show ‹first_time sub c n t› by (simp add: alt_first_time_def)
qed
```

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 -
  from assms(2) have ‹∀n > 0. f n > 0› using strict_monoD[OF assms(1)] by force
  hence ‹∀n ≠ 0. ¬(f n ≤ 0)› by simp
  with assms(2) have ‹{i. f i ≤ 0} = {0}› by blast
  thus ?thesis by simp
qed
```

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 -
  from assms(1) have smf:‹strict_mono f›
                 and fxge:‹∀x. f x ≥ x›
    by (auto simp add: dilating_def dilating_fun_def)
  have finite_prefix:‹finite {i. f i ≤ n}› by (simp add: finite_less_ub fxge)
  from assms(1) have ‹f 0 ≤ n› by (simp add: dilating_def dilating_fun_def)
  hence ‹{i. f i ≤ n} ≠ {}› by blast
  from assms(3) fxge have ‹f n₀ < n› by linarith
  from assms(2) have ‹∀x > n₀. f x > n› using Max.coboundedI[OF finite_prefix]
    using not_le by auto
  with assms(3) strict_mono_less[OF smf] show ?thesis by auto
qed
```

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 -
  from assms have smf:‹strict_mono f›
```

```
    and no_img_tick:‹∀k. (∄k0. f k0 = k) ⟶ (∀c. ¬(hamlet ((Rep_run r) k c)))›
    and no_img_time:‹⋀n. (∄n0. f n0 = (Suc n))
                      ⟶ (∀c. time ((Rep_run r) (Suc n) c) = time ((Rep_run r) n c))›
    and fxge:‹∀x. f x ≥ x› and f0n:‹⋀n. f 0 ≤ n› and f0:‹f 0 = 0›
    by (auto simp add: dilating_def dilating_fun_def)
  have finite_prefix:‹⋀n. finite {i. f i ≤ n}› by (auto simp add: finite_less_ub fxge)
  have prefix_not_empty:‹⋀n. {i. f i ≤ n} ≠ {}› using f0n by blast

  have 1:‹mono (dil_inverse f)›
  proof -
  { fix x::‹nat› and y::‹nat› assume hyp:‹x ≤ y›
    hence inc:‹{i. f i ≤ x} ⊆ {i. f i ≤ y}›
      by (simp add: hyp Collect_mono le_trans)
    from Max_mono[OF inc prefix_not_empty finite_prefix]
      have ‹(dil_inverse f) x ≤ (dil_inverse f) y› unfolding dil_inverse_def .
  } thus ?thesis unfolding mono_def by simp
  qed

  from first_dilated_instant[OF smf f0] have 2:‹(dil_inverse f) 0 = 0›
    unfolding dil_inverse_def .

  from fxge have ‹∀n i. f i ≤ n ⟶ i ≤ n› using le_trans by blast
  hence 3:‹∀n. (dil_inverse f) n ≤ n› using Max_in[OF finite_prefix prefix_not_empty]
    unfolding dil_inverse_def by blast

  from 1 2 3 have *:‹contracting_fun (dil_inverse f)› by (simp add: contracting_fun_def)

  have ‹∀n. finite {i. f i ≤ n}› by (simp add: finite_prefix)
  moreover have ‹∀n. {i. f i ≤ n} ≠ {}› using prefix_not_empty by blast
  ultimately have 4:‹∀n. f ((dil_inverse f) n) ≤ n›
    unfolding dil_inverse_def
    using assms(1) dilating_def dilating_fun_def Max_in by blast

  have 5:‹∀n c k. f ((dil_inverse f) n) < k ∧ k ≤ n
                            ⟶ ¬ hamlet ((Rep_run r) k c)›
    using not_image_stut[OF assms] no_img_tick unfolding dil_inverse_def by blast

  have 6:‹(∀n c k. f ((dil_inverse f) n) ≤ k ∧ k ≤ n
                    ⟶ time ((Rep_run r) k c) = time ((Rep_run sub) ((dil_inverse f) n) c))›
  proof -
    { fix n c k assume h:‹f ((dil_inverse f) n) ≤ k ∧ k ≤ n›
      let ?τ = ‹time (Rep_run sub ((dil_inverse f) n) c)›
      have tau:‹time (Rep_run sub ((dil_inverse f) n) c) = ?τ› ..
      have gn:‹(dil_inverse f) n = Max {i. f i ≤ n}› unfolding dil_inverse_def ..
      from time_stuttering[OF assms tau, of k] not_image_stut[OF assms gn]
      have ‹time ((Rep_run r) k c) = time ((Rep_run sub) ((dil_inverse f) n) c)›
      proof (cases ‹f ((dil_inverse f) n) = k›)
        case True
          moreover have ‹∀n c. time (Rep_run sub n c) = time (Rep_run r (f n) c)›
            using assms by (simp add: dilating_def)
          ultimately show ?thesis by simp
      next
        case False
          with h have ‹f (Max {i. f i ≤ n}) < k ∧ k ≤ n› by (simp add: dil_inverse_def)
          with time_stuttering[OF assms tau, of k] not_image_stut[OF assms gn]
            show ?thesis unfolding dil_inverse_def by auto
      qed
    } thus ?thesis by simp
  qed
```

```
  from * 4 5 6 show ?thesis unfolding contracting_def by simp
qed
```

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
  from assms(1) have *:‹⋀n. finite {i. f i ≤ n}›
    using finite_less_ub by (simp add:  dilating_def dilating_fun_def)
  from assms(1) have ‹f 0 = 0› by (simp add:  dilating_def dilating_fun_def)
  hence ‹⋀n. 0 ∈ {i. f i ≤ n}› by simp
  hence **:‹⋀n. {i. f i ≤ n} ≠ {}› by blast
  { fix n assume h:‹g n < (dil_inverse f) n›
    hence ‹∃k > g n. f k ≤ n› unfolding dil_inverse_def using Max_in[OF * **] by blast
    from this obtain k where kprop:‹g n < k ∧ f k ≤ n› by blast
    with assms(3) dense_run_def obtain c where ‹hamlet ((Rep_run sub) k c)› by blast
    hence ‹hamlet ((Rep_run r) (f k) c)› using ticks_sub[OF assms(1)] by blast
    moreover from kprop have ‹f (g n) < f k ∧ f k ≤ n› using assms(1)
      by (simp add: dilating_def dilating_fun_def strict_monoD)
    ultimately have False using assms(2) unfolding contracting_def by blast
  } hence 1:‹⋀n. ¬(g n < (dil_inverse f) n)› by blast
  { fix n assume h:‹g n > (dil_inverse f) n›
    have ‹∃k ≤ g n. f k > n›
    proof -
      { assume ‹∀k ≤ g n. f k ≤ n›
        with h have False unfolding dil_inverse_def
        using Max_gr_iff[OF * **] by blast
      }
      thus ?thesis using not_less by blast
    qed
    from this obtain k where ‹k ≤ g n ∧ f k > n› by blast
    hence ‹f (g n) ≥ f k ∧ f k > n› using assms(1)
      by (simp add: dilating_def dilating_fun_def strict_mono_less_eq)
    hence ‹f (g n) > n› by simp
    with assms(2) have False unfolding contracting_def by (simp add: leD)
  } hence 2:‹⋀n. ¬(g n > (dil_inverse f) n)› by blast
  from 1 2 show ‹⋀n. g n = (dil_inverse f) n› by (simp add: not_less_iff_gr_or_eq)
qed

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 -
  from assms(1) is_subrun_def obtain f
    where ‹dilating f sub r› by blast
  hence ‹∀n c. time ((Rep_run sub) n c) = time ((Rep_run r) (f n) c)
          ∧ hamlet ((Rep_run sub) n c) = hamlet ((Rep_run r) (f n) c)› by (simp add: dilating_def)
  moreover from assms(2) have
    ‹sub ∈ {r. ∃ n. hamlet ((Rep_run r) n c) ∧ time ((Rep_run r) n c') = τ}› by simp
  from this obtain k where ‹time ((Rep_run sub) k c') = τ ∧ hamlet ((Rep_run sub) k c)› by auto
  ultimately have ‹time ((Rep_run r) (f k) c') = τ ∧ hamlet ((Rep_run r) (f k) c)› by simp
  thus ?thesis by auto
qed
```

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 -
  from assms(1) is_subrun_def obtain f where ‹dilating f sub r› by blast
  moreover from assms(2) have
    ‹sub ∈ {r. ∀n. hamlet ((Rep_run r) n c1) ⟶ hamlet ((Rep_run r) n c2)}› by simp
  hence ‹∀n. hamlet ((Rep_run sub) n c1) ⟶ hamlet ((Rep_run sub) n c2)› by simp
  ultimately have ‹∀n. hamlet ((Rep_run r) n c1) ⟶ hamlet ((Rep_run r) n c2)›
    using ticks_imp_ticks_subk ticks_sub by blast
  thus ?thesis by simp
qed

theorem implies_not_sub:
  assumes ‹sub ≪ r›
      and ‹sub ∈ ⟦c1 implies not c2⟧TESL›
    shows ‹r ∈ ⟦c1 implies not c2⟧TESL›
proof -
  from assms(1) is_subrun_def obtain f where ‹dilating f sub r› by blast
  moreover from assms(2) have
    ‹sub ∈ {r. ∀n. hamlet ((Rep_run r) n c1) ⟶ ¬ hamlet ((Rep_run r) n c2)}› by simp
  hence ‹∀n. hamlet ((Rep_run sub) n c1) ⟶ ¬ hamlet ((Rep_run sub) n c2)› by simp
  ultimately have ‹∀n. hamlet ((Rep_run r) n c1) ⟶ ¬ hamlet ((Rep_run r) n c2)›
    using ticks_imp_ticks_subk ticks_sub by blast
  thus ?thesis by simp
qed
```

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 -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  from assms(2) have
    ‹sub ∈ {r. ∀n. (run_tick_count r c2 n) ≤ (run_tick_count r c1 n)}› by simp
  hence ‹∀n. (run_tick_count sub c2 n) ≤ (run_tick_count sub c1 n)› by simp
  from dil_tick_count[OF assms(1) this]
    have ‹∀n. (run_tick_count r c2 n) ≤ (run_tick_count r c1 n)› by simp
  thus ?thesis by simp
```

```
qed

theorem strictly_precedes_sub:
  assumes ‹sub ≪ r›
      and ‹sub ∈ ⟦c₁ strictly precedes c₂⟧TESL›
    shows ‹r ∈ ⟦c₁ strictly precedes c₂⟧TESL›
proof -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  from assms(2) have
    ‹sub ∈ { ϱ. ∀n::nat. (run_tick_count ϱ c₂ n) ≤ (run_tick_count_strictly ϱ c₁ n) }›
  by simp
  with strictly_precedes_alt_def2[of ‹c₂› ‹c₁›]  have
    ‹sub ∈ { ϱ. (¬hamlet ((Rep_run ϱ) 0 c₂))
  ∧ (∀n::nat. (run_tick_count ϱ c₂ (Suc n)) ≤ (run_tick_count ϱ c₁ n)) }›
  by blast
  hence ‹(¬hamlet ((Rep_run sub) 0 c₂))
       ∧ (∀n::nat. (run_tick_count sub c₂ (Suc n)) ≤ (run_tick_count sub c₁ n))›
    by simp
  hence
    1:‹(¬hamlet ((Rep_run sub) 0 c₂))
     ∧ (∀n::nat. (tick_count sub c₂ (Suc n)) ≤ (tick_count sub c₁ n))›
  by (simp add: tick_count_is_fun)
  have ‹∀n::nat. (tick_count r c₂ (Suc n)) ≤ (tick_count r c₁ n)›
  proof -
    { fix n::nat
      have ‹tick_count r c₂ (Suc n) ≤ tick_count r c₁ n›
      proof (cases ‹∃n₀. f n₀ = n›)
        case True — n is in the image of f
          from this obtain n₀ where fn:‹f n₀ = n› by blast
          show ?thesis
          proof (cases ‹∃sn₀. f sn₀ = Suc n›)
            case True — Suc n is in the image of f
              from this obtain sn₀ where fsn:‹f sn₀ = Suc n› by blast
              with fn strict_mono_suc * have ‹sn₀ = Suc n₀›
                using  dilating_def dilating_fun_def by blast
              with 1 have ‹tick_count sub c₂ sn₀ ≤ tick_count sub c₁ n₀› by simp
              thus ?thesis using fn fsn tick_count_sub[OF *] by simp
          next
            case False — Suc n is not in the image of f
              hence ‹¬hamlet ((Rep_run r) (Suc n) c₂)›
                using * by (simp add: dilating_def dilating_fun_def)
              hence ‹tick_count r c₂ (Suc n) = tick_count r c₂ n›
                by (simp add: tick_count_suc)
              also have ‹... = tick_count sub c₂ n₀›
                using fn tick_count_sub[OF *] by simp
              finally have ‹tick_count r c₂ (Suc n) = tick_count sub c₂ n₀› .
              moreover have ‹tick_count sub c₂ n₀ ≤ tick_count sub c₂ (Suc n₀)›
                by (simp add: tick_count_suc)
              ultimately have
                ‹tick_count r c₂ (Suc n) ≤ tick_count sub c₂ (Suc n₀)› by simp
              moreover have
                ‹tick_count sub c₂ (Suc n₀) ≤ tick_count sub c₁ n₀› using 1 by simp
              ultimately have ‹tick_count r c₂ (Suc n) ≤ tick_count sub c₁ n₀› by simp
              thus ?thesis using tick_count_sub[OF *] fn by simp
          qed
      next
        case False — n is not in the image of f
```

```
        from greatest_prev_image[OF * this] obtain np where
          np_prop:‹f np < n ∧ (∀k. f np < k ∧ k ≤ n ⟶ (∄k0. f k0 = k))› by blast
        from tick_count_latest[OF * this] have
          ‹tick_count r c1 n = tick_count r c1 (f np)› .
        hence a:‹tick_count r c1 n = tick_count sub c1 np›
          using tick_count_sub[OF *] by simp
        have b: ‹tick_count sub c2 (Suc np) ≤ tick_count sub c1 np› using 1 by simp
        show ?thesis
        proof (cases ‹∃sn0. f sn0 = Suc n›)
          case True — Suc n is in the image of f
            from this obtain sn0 where fsn:‹f sn0 = Suc n› by blast
            from next_non_stuttering[OF * np_prop this]  have sn_prop:‹sn0 = Suc np› .
            with b have ‹tick_count sub c2 sn0 ≤ tick_count sub c1 np› by simp
            thus ?thesis using tick_count_sub[OF *] fsn a by auto
        next
          case False — Suc n is not in the image of f
            hence ‹¬hamlet ((Rep_run r) (Suc n) c2)›
              using * by (simp add: dilating_def dilating_fun_def)
            hence ‹tick_count r c2 (Suc n) = tick_count r c2 n›
              by (simp add: tick_count_suc)
            also have ‹... = tick_count sub c2 np› using np_prop tick_count_sub[OF *]
              by (simp add: tick_count_latest[OF * np_prop])
            finally have ‹tick_count r c2 (Suc n) = tick_count sub c2 np› .
            moreover have ‹tick_count sub c2 np ≤ tick_count sub c2 (Suc np)›
              by (simp add: tick_count_suc)
            ultimately have
              ‹tick_count r c2 (Suc n) ≤ tick_count sub c2 (Suc np)› by simp
            moreover have
              ‹tick_count sub c2 (Suc np) ≤ tick_count sub c1 np› using 1 by simp
            ultimately have ‹tick_count r c2 (Suc n) ≤ tick_count sub c1 np› by simp
            thus ?thesis using np_prop mono_tick_count  using a by linarith
        qed
      qed
    } thus ?thesis ..
  qed
  moreover from 1 have ‹¬hamlet ((Rep_run r) 0 c2)›
    using * empty_dilated_prefix ticks_sub by fastforce
  ultimately show ?thesis by (simp add: tick_count_is_fun strictly_precedes_alt_def2)
qed
```

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 -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  from assms(2) have ‹∀n. hamlet ((Rep_run sub) n a)
                       ⟶ (∀m ≥ n. first_time sub ms m (time ((Rep_run sub) n ms) + δτ)
                                    ⟶ hamlet ((Rep_run sub) m b))›
    using TESL_interpretation_atomic.simps(5)[of ‹a› ‹δτ› ‹ms› ‹b›] by simp
  hence **:‹∀n0. hamlet ((Rep_run r) (f n0) a)
               ⟶ (∀m0 ≥ n0. first_time r ms (f m0) (time ((Rep_run r) (f n0) ms) + δτ)
                             ⟶ hamlet ((Rep_run r) (f m0) b))  ›
    using first_time_image[OF *] dilating_def * by fastforce
  hence ‹∀n. hamlet ((Rep_run r) n a)
               ⟶ (∀m ≥ n. first_time r ms m (time ((Rep_run r) n ms) + δτ)
                           ⟶ hamlet ((Rep_run r) m b))›
```

```
  proof -
    { fix n assume assm:‹hamlet ((Rep_run r) n a)›
      from ticks_image_sub[OF * assm] obtain n0 where nfn0:‹n = f n0› by blast
      with ** assm have ft0:
        ‹(∀m0 ≥ n0. first_time r ms (f m0) (time ((Rep_run r) (f n0) ms) + δτ)
                  ⟶ hamlet ((Rep_run r) (f m0) b))› by blast
      have ‹(∀m ≥ n. first_time r ms m (time ((Rep_run r) n ms) + δτ)
                     ⟶ hamlet ((Rep_run r) m b)) ›
      proof -
      { fix m assume hyp:‹m ≥ n›
        have ‹first_time r ms m (time (Rep_run r n ms) + δτ) ⟶ hamlet (Rep_run r m b)›
        proof (cases ‹∃m0. f m0 = m›)
          case True
          from this obtain m0 where ‹m = f m0› by blast
          moreover have ‹strict_mono f› using * by (simp add: dilating_def dilating_fun_def)
          ultimately show ?thesis using ft0 hyp nfn0 by (simp add: strict_mono_less_eq)
        next
          case False thus ?thesis
          proof (cases ‹m = 0›)
            case True
              hence ‹m = f 0› using * by (simp add: dilating_def dilating_fun_def)
              then show ?thesis using False by blast
          next
            case False
            hence ‹∃pm. m = Suc pm› by (simp add: not0_implies_Suc)
            from this obtain pm where mpm:‹m = Suc pm› by blast
            hence ‹∄pm0. f pm0 = Suc pm› using ‹∄m0. f m0 = m› by simp
            with *  have ‹time (Rep_run r (Suc pm) ms) = time (Rep_run r pm ms)›
              using dilating_def dilating_fun_def by blast
            hence ‹time (Rep_run r pm ms) = time (Rep_run r m ms)› using mpm by simp
            moreover from mpm have ‹pm < m› by simp
            ultimately have ‹∃m' < m. time (Rep_run r m' ms) = time (Rep_run r m ms)› by blast
            hence ‹¬(first_time r ms m (time (Rep_run r n ms) + δτ))›
              by (auto simp add: first_time_def)
            thus ?thesis by simp
          qed
        qed
      } thus ?thesis by simp
      qed
    } thus ?thesis by simp
  qed
  thus ?thesis by simp
qed
```

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 -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  moreover from assms(2) TESL_interpretation_atomic.simps(2) have
    ‹sub ∈ {r. ∀n. R (time ((Rep_run r) n c1), time ((Rep_run r) n c2))}› by blast
  hence 1:‹∀n. R (time ((Rep_run sub) n c1), time ((Rep_run sub) n c2))› by simp
  show ?thesis
  proof (induction n)
    case 0
      from 1 have ‹R (time ((Rep_run sub) 0 c1), time ((Rep_run sub) 0 c2))› by simp
```

```
      moreover from * have ‹f 0 = 0› by (simp add: dilating_def dilating_fun_def)
      moreover from * have ‹∀c. time ((Rep_run sub) 0 c) = time ((Rep_run r) (f 0) c)›
        by (simp add: dilating_def)
      ultimately show ?case by simp
  next
    case (Suc n)
    then show ?case
    proof (cases ‹∄n0. f n0 = Suc n›)
      case True
      with * have ‹∀c. time (Rep_run r (Suc n) c) = time (Rep_run r n c)›
        by (simp add: dilating_def dilating_fun_def)
      thus ?thesis using Suc.IH by simp
    next
      case False
      from this obtain n0 where n0prop:‹f n0 = Suc n› by blast
      from 1 have ‹R (time ((Rep_run sub) n0 c1), time ((Rep_run sub) n0 c2))› by simp
      moreover from n0prop * have ‹time ((Rep_run sub) n0 c1) = time ((Rep_run r) (Suc n) c1)›
        by (simp add: dilating_def)
      moreover from n0prop * have ‹time ((Rep_run sub) n0 c2) = time ((Rep_run r) (Suc n) c2)›
        by (simp add: dilating_def)
      ultimately show ?thesis by simp
    qed
  qed
qed

corollary tagrel_sub:
  assumes ‹sub ≪ r›
      and ‹sub ∈ ⟦ time-relation ⌊c1,c2⌋ ∈ R ⟧TESL›
    shows ‹r ∈ ⟦ time-relation ⌊c1,c2⌋ ∈ R ⟧TESL›
using tagrel_sub'[OF assms] unfolding TESL_interpretation_atomic.simps(3) by simp
```

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 -
  from assms(1) is_subrun_def obtain f where df:‹dilating f sub r› by blast
  moreover from assms(2) TESL_interpretation_atomic.simps(2) have
    ‹r ∈ {ρ. ∀n. R (time ((Rep_run ρ) n c1), time ((Rep_run ρ) n c2))}› by blast
  hence ‹∀n. R (time ((Rep_run r) n c1), time ((Rep_run r) n c2))› by simp
  hence ‹∀n. (∃n0. f n0 = n) ⟶ R (time ((Rep_run r) n c1), time ((Rep_run r) n c2))› by simp
  hence ‹∀n0. R (time ((Rep_run r) (f n0) c1), time ((Rep_run r) (f n0) c2))› by blast
  moreover from dilating_def df have
    ‹∀n c. time ((Rep_run sub) n c) = time ((Rep_run r) (f n) c)› by blast
  ultimately have ‹∀n0. R (time ((Rep_run sub) n0 c1), time ((Rep_run sub) n0 c2))› by auto
  thus ?thesis by simp
qed
```

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 -
  from assms(1) is_subrun_def obtain f where *:‹dilating f sub r› by blast
  from assms(2) TESL_interpretation_atomic.simps(8) have
    ‹∀n. hamlet (Rep_run sub n c1) ⟶ (∀m≥n. ¬ hamlet (Rep_run sub m c2))› by simp
```

```
  hence 1:‹∀n. hamlet (Rep_run r (f n) c₁) ⟶ (∀m≥n. ¬ hamlet (Rep_run r (f m) c₂))›
    using ticks_sub[OF *] by simp
  hence ‹∀n. hamlet (Rep_run r (f n) c₁) ⟶ (∀m≥ (f n). ¬ hamlet (Rep_run r m c₂))›
  proof -
    { fix n assume ‹hamlet (Rep_run r (f n) c₁)›
      with 1 have 2:‹∀ m ≥ n. ¬ hamlet (Rep_run r (f m) c₂)› by simp
      have ‹∀ m≥ (f n). ¬ hamlet (Rep_run r m c₂)›
      proof -
        { fix m assume h:‹m ≥ f n›
          have ‹¬ hamlet (Rep_run r m c₂)›
          proof (cases ‹∃m₀. f m₀ = m›)
            case True
              from this obtain m₀ where fm0:‹f m₀ = m› by blast
              hence ‹m₀ ≥ n›
                using * dilating_def dilating_fun_def h strict_mono_less_eq by fastforce
              with 2 show ?thesis using fm0 by blast
          next
            case False
              thus ?thesis  using ticks_image_sub'[OF *] by blast
          qed
        } thus ?thesis by simp
      qed
    } thus ?thesis by simp
  qed
  hence ‹∀n. hamlet (Rep_run r n c₁) ⟶ (∀m ≥ n. ¬ hamlet (Rep_run r m c₂))›
    using ticks_imp_ticks_subk[OF *] by blast
  thus ?thesis using TESL_interpretation_atomic.simps(8) by blast
qed

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›
using assms(2) atomic_sub_lemmas[OF assms(1)] by (cases φ, simp_all)
```

Finally, any TESL specification is invariant by stuttering.

```
theorem TESL_stuttering_invariant:
  assumes ‹sub ≪ r›
    shows ‹sub ∈ ⟦⟦ S ⟧⟧TESL ⟹ r ∈ ⟦⟦ S ⟧⟧TESL›
proof (induction S)
  case Nil
    thus ?case by simp
next
  case (Cons a s)
    from Cons.prems have sa:‹sub ∈ ⟦ a ⟧TESL› and sb:‹sub ∈ ⟦⟦ s ⟧⟧TESL›
      using TESL_interpretation_image by simp+
    from Cons.IH[OF sb] have ‹r ∈ ⟦⟦ s ⟧⟧TESL› .
    moreover from atomic_sub[OF assms(1) sa] have ‹r ∈ ⟦ a ⟧TESL› .
    ultimately show ?case using  TESL_interpretation_image by simp
qed

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 -
  have * : ‹∃ n. (([], 0 ⊢ Ψ ▷ []) ↪^n (Γ1, n1 ⊢ [] ▷ []))›
    by (simp add: rtranclp_imp_relpowp start)
  show ?thesis
  unfolding consistent_context_def consistent_def
  using * consistent_context_def init_invariant soundness by fastforce
qed
```

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 ≫^k r' ≡ 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)›
oops

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.