A Formal Development of a Polychronous Polytimed Coordination Language

Hai NGuyen Van Frederic Boulanger Burkhart Wolff

April 2, 2019

Contents

Chapter 1

A Gentle Introduction to TESL

1.1 Context

The design of complex systems involves different formalisms for modeling their different parts or aspects. The global model of a system may therefore consist of a coordination of concurrent sub-models that use different paradigms such as differential equations, state machines, synchronous dataflow 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).

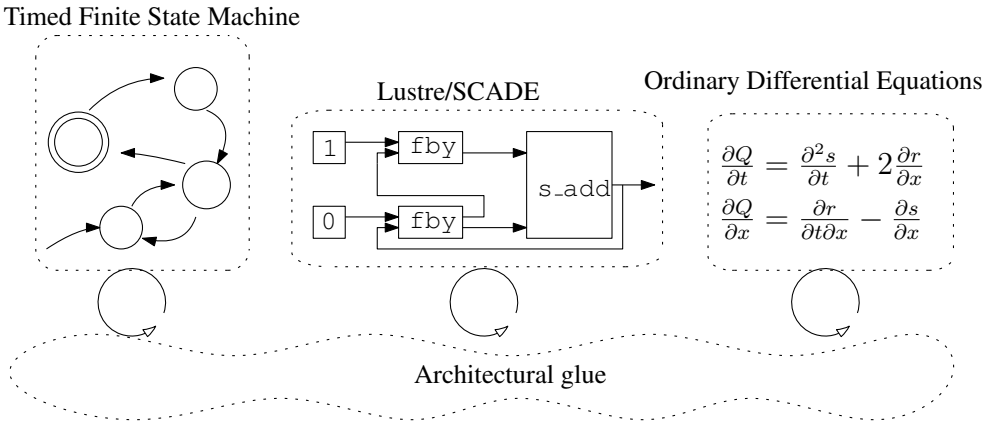

Figure 1.1: A Heterogeneous Timed System Model

In order to tackle the heterogeneous nature of the subsystems, we abstract their behavior as clocks. Each clock models an event – something that can occur or not at a given time. This time is measured in a time frame associated with each clock, and the nature of time (integer, rational, real or any type with a linear order) is specific to each clock. When the event associated with a clock occurs, the clock ticks. In order to support any kind of behavior for the subsystems, we are only interested in specifying what we can observe at a series of discrete instants. There are two constraints on observations: a clock may tick only at an observation instant, and the time on any clock cannot decrease from an instant to the next one. However, it is always possible to add arbitrary observation instants, which allows for stuttering and modular composition of systems. As a consequence, the key concept of our setting is the notion of a clock-indexed Kripke model: $\Sigma^\infty = \mathbb{N} \to \mathcal{K} \to (\mathbb{B} \times \mathcal{T})$, where $\mathcal{K}$ is an enumerable set of clocks, $\mathbb{B}$ is the set of booleans – used to indicate that a clock ticks at a given instant – and $\mathcal{T}$ is a universal metric time space for which we only assume that it is large enough to contain all individual time spaces of clocks and that it is ordered by some linear ordering ($\leq_\mathcal{T}$).

The elements of Σ^∞ are called runs. A specification language is a set of operators that constrains the set of possible monotonic runs. Specifications are composed by intersecting the denoted run sets of constraint operators.

Consequently, such specification languages do not limit the number of clocks used to model a system (as long as it is finite) and it is always possible to add clocks to a specification. Moreover they are *compositional* by construction since the composition of specifications consists of the conjunction of their constraints.

This work provides the following contributions:

- defining the non-trivial language *TESL** in terms of clock-indexed Kripke models,
- proving that this denotational semantics is stuttering invariant,
- defining an adapted form of symbolic primitives and presenting the set of operational semantic rules,
- presenting formal proofs for soundness, completeness, and progress of the latter.

1.2 The TESL Language

The TESL language [1] was initially designed to coordinate the execution of heterogeneous components during the simulation of a system. We define here a minimal kernel of operators that will form the basis of a family of specification languages, including the original TESL language, which is described at http://wdi.supelec.fr/software/TESL/.

1.2.1 Instantaneous Causal Operators

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

- `c1 implies c2` means that at any instant where `c1` ticks, `c2` has to tick too.
- `c1 implies not c2` means that at any instant where `c1` ticks, `c2` cannot tick.
- `c1 kills c2` means that at any instant where `c1` ticks, and at any future instant, `c2` cannot tick.

1.2.2 Temporal Operators

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

- `c sporadic t` means that clock `c` must have a tick at time `t` on its own time scale.
- `c1 sporadic t on c2` means that clock `c1` must have a tick at an instant where the time on `c2` is `t`.
- `c1 time delayed by d on m implies c2` means every time clock `c1` ticks, `c2` must have a tick at an 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 closk `m`.
- `time relation (c1, c2) in R` means that at every instant, the current times 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 will consider only linear relations here so that finding solutions is decidable.

1.2.3 Asynchronous Operators

The last category of TESL operators allows the specification of asynchronous relations between event occurrences. They do not tell when ticks have to occur, then 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 instant or at the same instant. This can also be expressed by saying that at each instant, the number of ticks on `c2` since the beginning of the run must be lower or equal to the number of ticks 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

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

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.
- Instant indexes, (FIXME) which are natural integers, should not be used directly but appear here for technical and historical reasons.
- Tag constants are just constants of a type which denotes the metric time space.
- Tag variables represent the time at a given instant on a given clock.
- Tag expressions are used to represent either a tag constant or a delayed time with respect to a tag variable.

datatype *clock* = *Clk* ‹*string*›
type-synonym *instant-index* = ‹*nat*›

datatype $'\tau$ *tag-const* =

TConst $'\tau$ (τ_{cst})

datatype *tag-var* =
TSchematic ‹*clock* ∗ *instant-index*› (τ_{var})

2.1.2 Operators for the TESL language

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

datatype $'\tau$ *TESL-atomic* =
SporadicOn ‹*clock*› ‹$'\tau$ *tag-const*› ‹*clock*› (- *sporadic* - *on* - 55)
| *TagRelation* ‹*clock*› ‹*clock*› ‹($'\tau$ *tag-const* × $'\tau$ *tag-const*) ⇒ *bool*›
(*time*−*relation* ⌊-, -⌋ ∈ - 55)
| *Implies* ‹*clock*› ‹*clock*› (**infixr** *implies* 55)
| *ImpliesNot* ‹*clock*› ‹*clock*› (**infixr** *implies not* 55)
| *TimeDelayedBy* ‹*clock*› ‹$'\tau$ *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 $'\tau$ *TESL-formula* = ‹$'\tau$ *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* :: ‹$'\tau$ *TESL-atomic* ⇒ *bool*› **where**
‹*positive-atom* (- *sporadic* - *on* -) = *True*›
| ‹*positive-atom* - = *False*›

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

abbreviation *NoSporadic* :: ‹$'\tau$ *TESL-formula* ⇒ $'\tau$ *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, tag must be in a field. We show here that this is the case when the type parameter of $'\tau$ *tag-const* is itself a field.

instantiation *tag-const* :: (*plus*)*plus*
begin
fun *plus-tag-const* :: ‹$'a$ *tag-const* ⇒ $'a$ *tag-const* ⇒ $'a$ *tag-const*›

where
TConst-plus: ‹(*TConst n*) + (*TConst p*) = (*TConst* (*n* + *p*))›

instance by (*rule Groups.class.Groups.plus.of-class.intro*)
end

instantiation *tag-const* :: (*minus*)*minus*
begin
fun *minus-tag-const* :: ‹'*a tag-const* ⇒ '*a tag-const* ⇒ '*a tag-const*›
where
TConst-minus: ‹(*TConst n*) − (*TConst p*) = (*TConst* (*n* − *p*))›

instance by (*rule Groups.class.Groups.minus.of-class.intro*)
end

instantiation *tag-const* :: (*times*)*times*
begin
fun *times-tag-const* :: ‹'*a tag-const* ⇒ '*a tag-const* ⇒ '*a tag-const*›
where
TConst-times: ‹(*TConst n*) ∗ (*TConst p*) = (*TConst* (*n* ∗ *p*))›

instance by (*rule Groups.class.Groups.times.of-class.intro*)
end

instantiation *tag-const* :: (*divide*)*divide*
begin
fun *divide-tag-const* :: ‹'*a tag-const* ⇒ '*a tag-const* ⇒ '*a tag-const*›
where
TConst-divide: ‹*divide* (*TConst n*) (*TConst p*) = (*TConst* (*divide n p*))›

instance by (*rule Rings.class.Rings.divide.of-class.intro*)
end

instantiation *tag-const* :: (*inverse*)*inverse*
begin
fun *inverse-tag-const* :: ‹'*a tag-const* ⇒ '*a tag-const*›
where
TConst-inverse: ‹*inverse* (*TConst n*) = (*TConst* (*inverse n*))›

instance by (*rule Fields.class.Fields.inverse.of-class.intro*)
end

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
  show ‹⋀x  :: 'a tag-const. x ≤ x›
    by (metis (full-types) Int-less-eq order-refl tag-const.exhaust)
  show ‹⋀x y z  :: 'a tag-const. x ≤ y ⟹ y ≤ z ⟹ x ≤ z›
    using less-eq-tag-const.simps by auto
  show ‹⋀x y  :: 'a tag-const. x ≤ y ⟹ y ≤ x ⟹ x = y›
    using less-eq-tag-const.simps by auto
 qed
end

instantiation tag-const :: (linorder)linorder
begin
 instance proof
  show ‹⋀x y. (x::'a tag-const) ≤ y ∨ y ≤ x›
    by (metis (full-types) Int-less-eq le-cases tag-const.exhaust)
 qed
end

end
```

2.2 Defining Runs

```
theory Run
imports TESL

begin
```

Runs are sequences of instants, each instant mapping a clock to a pair that whether the clock ticks or not and what 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, τ_cst 0)) ∈ {ϱ. ∀ c. mono (λn. time (ϱ n c))}›
    unfolding mono-def by blast
qed
```

lemma *Abs-run-inverse-rewrite*:
‹∀ *c*. *mono* (λ*n*. *time* (ϱ *n c*)) ⟹ *Rep-run* (*Abs-run* ϱ) = ϱ›
by (*simp add*: *Abs-run-inverse*)

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*›

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)›

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* (λ*n*. *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* (λ*n*. *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

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

3.1 Denotational interpretation for atomic TESL formulae

fun *TESL-interpretation-atomic*
:: ‹('τ::*linordered-field*) *TESL-atomic* ⇒ 'τ *run set*› (⟦ - ⟧$_{TESL}$) **where**
‹⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}$ =
{ ϱ. ∃ n::*nat*. *hamlet* ((*Rep-run* ϱ) *n* K_1) ∧ *time* ((*Rep-run* ϱ) *n* K_2) = τ }›
| ‹⟦ *time−relation* ⌊K_1, K_2⌋ ∈ *R* ⟧$_{TESL}$ =
{ ϱ. ∀ n::*nat*. *R* (*time* ((*Rep-run* ϱ) *n* K_1), *time* ((*Rep-run* ϱ) *n* K_2)) }›
| ‹⟦ *master implies slave* ⟧$_{TESL}$ =
{ ϱ. ∀ n::*nat*. *hamlet* ((*Rep-run* ϱ) *n master*) ⟶ *hamlet* ((*Rep-run* ϱ) *n slave*) }›
| ‹⟦ *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* ⟧$_{TESL}$ =
— When master ticks, let's call @termt_0 the current date on measuring. Then, at the first instant when the date on measuring is @termt_0+δ*t*, 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 ⟧$_{TESL}$ =

$\{ \varrho.\ \forall n{::}nat.\ (run\text{-}tick\text{-}count\ \varrho\ K_2\ n) \leq (run\text{-}tick\text{-}count\ \varrho\ K_1\ n)\ \}\rangle$
| ‹$[\![K_1$ *strictly precedes* $K_2\ ]\!]_{TESL} =$
$\{ \varrho.\ \forall n{::}nat.\ (run\text{-}tick\text{-}count\ \varrho\ K_2\ n) \leq (run\text{-}tick\text{-}count\text{-}strictly\ \varrho\ K_1\ n)\ \}\rangle$
| ‹$[\![K_1$ *kills* $K_2\ ]\!]_{TESL} =$
$\{ \varrho.\ \forall n{::}nat.\ hamlet\ ((Rep\text{-}run\ \varrho)\ n\ K_1) \longrightarrow (\forall m{\geq}n.\ \neg\ hamlet\ ((Rep\text{-}run\ \varrho)\ m\ K_2))\ \}\rangle$

3.2 Denotational interpretation for TESL formulae

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 *auto*

3.2.1 Image interpretation lemma

theorem *TESL-interpretation-image*:
‹[[[Φ]]]$_{TESL}$ = $\bigcap$ ((λφ. [[φ]]$_{TESL}$) ' *set* Φ)›
proof (*induct* Φ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Φ)
then show *?case* **by** *auto*
qed

3.2.2 Expansion law

Similar to the expansion laws of lattices

theorem *TESL-interp-homo-append*:
shows ‹[[[Φ_1 @ Φ_2]]]$_{TESL}$ = [[[Φ_1]]]$_{TESL}$ ∩ [[[Φ_2]]]$_{TESL}$›
proof (*induct* Φ_1)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Φ_1)
then show *?case* **by** *auto*
qed

3.3 Equational laws for TESL formulae denotationally interpreted

lemma *TESL-interp-assoc*:
shows ‹$[\![(\Phi_1 \mathbin{@} \Phi_2) \mathbin{@} \Phi_3]\!]_{TESL} = [\![\Phi_1 \mathbin{@} (\Phi_2 \mathbin{@} \Phi_3)]\!]_{TESL}$›
by *auto*

lemma *TESL-interp-commute*:
shows ‹$[\![\Phi_1 \mathbin{@} \Phi_2]\!]_{TESL} = [\![\Phi_2 \mathbin{@} \Phi_1]\!]_{TESL}$›
by (*simp add*: *TESL-interp-homo-append inf-sup-aci*(*1*))

lemma *TESL-interp-left-commute*:
shows ‹$[\![\Phi_1 \mathbin{@} (\Phi_2 \mathbin{@} \Phi_3)]\!]_{TESL} = [\![\Phi_2 \mathbin{@} (\Phi_1 \mathbin{@} \Phi_3)]\!]_{TESL}$›
unfolding *TESL-interp-homo-append* **by** *auto*

lemma *TESL-interp-idem*:
shows ‹$[\![\Phi \mathbin{@} \Phi]\!]_{TESL} = [\![\Phi]\!]_{TESL}$›
using *TESL-interp-homo-append* **by** *auto*

lemma *TESL-interp-left-idem*:
shows ‹$[\![\Phi_1 \mathbin{@} (\Phi_1 \mathbin{@} \Phi_2)]\!]_{TESL} = [\![\Phi_1 \mathbin{@} \Phi_2]\!]_{TESL}$›
using *TESL-interp-homo-append* **by** *auto*

lemma *TESL-interp-right-idem*:
shows ‹$[\![(\Phi_1 \mathbin{@} \Phi_2) \mathbin{@} \Phi_2]\!]_{TESL} = [\![\Phi_1 \mathbin{@} \Phi_2]\!]_{TESL}$›
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*

lemma *TESL-interp-neutral1*:
shows ‹$[\![[] \mathbin{@} \Phi]\!]_{TESL} = [\![\Phi]\!]_{TESL}$›
by *simp*

lemma *TESL-interp-neutral2*:
shows ‹$[\![\Phi \mathbin{@} []]\!]_{TESL} = [\![\Phi]\!]_{TESL}$›
by *simp*

3.4 Decreasing interpretation of TESL formulae

lemma *TESL-sem-decreases-head*:
‹$[\![\Phi]\!]_{TESL} \supseteq [\![\varphi \mathbin{\#} \Phi]\!]_{TESL}$›
by *simp*

lemma *TESL-sem-decreases-tail*:
‹$[\![\Phi]\!]_{TESL} \supseteq [\![\Phi \mathbin{@} [\varphi]]\!]_{TESL}$›
by (*simp add*: *TESL-interp-homo-append*)

lemma *TESL-interp-formula-stuttering*:
assumes *bel*: ‹$\varphi \in$ *set* Φ›
shows ‹$[\![\varphi \# \Phi]\!]_{TESL} = [\![\Phi]\!]_{TESL}$›
by (*metis Int-subset-iff TESL-interp-homo-append TESL-interpretation.simps*(*2*) *bel in-set-conv-decomp-first subset-antisym subset-refl*)

lemma *TESL-interp-decreases*:
shows ‹$[\![\Phi]\!]_{TESL} \supseteq [\![\varphi \# \Phi]\!]_{TESL}$›
by (*rule TESL-sem-decreases-head*)

lemma *TESL-interp-remdups-absorb*:
shows ‹$[\![\Phi]\!]_{TESL} = [\![$ *remdups* $\Phi]\!]_{TESL}$›
proof (*induct* Φ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Φ)
then show *?case*
using *TESL-interp-formula-stuttering* **by** *auto*
qed

lemma *TESL-interp-set-lifting*:
assumes ‹*set* Φ = *set* Φ'›
shows ‹$[\![\Phi]\!]_{TESL} = [\![\Phi']\!]_{TESL}$›
proof −
have ‹*set* (*remdups* Φ) = *set* (*remdups* Φ')›
by (*simp add*: *assms*)
moreover have *fxpnt*Φ: ‹$\bigcap$ $((\lambda\varphi.\ [\![\varphi]\!]_{TESL})$ ‘ *set* $\Phi) = [\![\Phi]\!]_{TESL}$›
by (*simp add*: *TESL-interpretation-image*)
moreover have *fxpnt*Φ': ‹$\bigcap$ $((\lambda\varphi.\ [\![\varphi]\!]_{TESL})$ ‘ *set* $\Phi') = [\![\Phi']\!]_{TESL}$›
by (*simp add*: *TESL-interpretation-image*)
moreover have ‹$\bigcap$ $((\lambda\varphi.\ [\![\varphi]\!]_{TESL})$ ‘ *set* $\Phi) = \bigcap$ $((\lambda\varphi.\ [\![\varphi]\!]_{TESL})$ ‘ *set* Φ')›
by (*simp add*: *assms*)
ultimately show *?thesis* **using** *TESL-interp-remdups-absorb* **by** *auto*
qed

theorem *TESL-interp-decreases-setinc*:
assumes *incl*: ‹*set* $\Phi \subseteq$ *set* Φ'›
shows ‹$[\![\Phi]\!]_{TESL} \supseteq [\![\Phi']\!]_{TESL}$›
proof −
obtain Φ_r **where** *decompose*: ‹*set* $(\Phi$ @ $\Phi_r)$ = *set* Φ'› **using** *incl* **by** *auto*
have ‹*set* $(\Phi$ @ $\Phi_r)$ = *set* Φ'› **using** *incl decompose* **by** *blast*
moreover have ‹(*set* $\Phi) \cup$ (*set* Φ_r) = *set* Φ'› **using** *incl decompose* **by** *auto*
moreover have ‹$[\![\Phi']\!]_{TESL} = [\![\Phi$ @ $\Phi_r]\!]_{TESL}$› **using** *TESL-interp-set-lifting decompose* **by** *blast*
moreover have ‹$[\![\Phi$ @ $\Phi_r]\!]_{TESL} = [\![\Phi]\!]_{TESL} \cap [\![\Phi_r]\!]_{TESL}$› **by** (*simp add*: *TESL-interp-homo-append*)

moreover have ‹$[\![\Phi]\!]_{TESL} \supseteq [\![\Phi]\!]_{TESL} \cap [\![\Phi_r]\!]_{TESL}$› **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *TESL-interp-decreases-add-head*:
assumes *incl*: ‹*set* $\Phi \subseteq$ *set* Φ'›
shows ‹$[\![\varphi \# \Phi]\!]_{TESL} \supseteq [\![\varphi \# \Phi']\!]_{TESL}$›
using *TESL-interp-decreases-setinc incl* **by** *auto*

lemma *TESL-interp-decreases-add-tail*:
assumes *incl*: ‹*set* $\Phi \subseteq$ *set* Φ'›
shows ‹$[\![\Phi @ [\varphi]]\!]_{TESL} \supseteq [\![\Phi' @ [\varphi]]\!]_{TESL}$›
by (*metis TESL-interp-commute TESL-interp-decreases-add-head append-Cons append-Nil incl*)

lemma *TESL-interp-absorb1*:
assumes *incl*: ‹*set* $\Phi_1 \subseteq$ *set* Φ_2›
shows ‹$[\![\Phi_1 @ \Phi_2]\!]_{TESL} = [\![\Phi_2]\!]_{TESL}$›
by (*simp add*: *Int-absorb1 TESL-interp-decreases-setinc TESL-interp-homo-append incl*)

lemma *TESL-interp-absorb2*:
assumes *incl*: ‹*set* $\Phi_2 \subseteq$ *set* Φ_1›
shows ‹$[\![\Phi_1 @ \Phi_2]\!]_{TESL} = [\![\Phi_1]\!]_{TESL}$›
using *TESL-interp-absorb1 TESL-interp-commute incl* **by** *blast*

3.5 Some special cases

lemma *NoSporadic-stable* [*simp*]:
shows ‹$[\![\Phi]\!]_{TESL} \subseteq [\![$ *NoSporadic* $\Phi]\!]_{TESL}$›
by (*meson filter-is-subset TESL-interp-decreases-setinc*)

lemma *NoSporadic-idem* [*simp*]:
shows ‹$[\![\Phi]\!]_{TESL} \cap [\![$ *NoSporadic* $\Phi]\!]_{TESL} = [\![\Phi]\!]_{TESL}$›
by (*meson Int-absorb2 filter-is-subset TESL-interp-decreases-setinc*)

lemma *NoSporadic-setinc*:
shows ‹*set* (*NoSporadic* Φ) $\subseteq$ *set* Φ›
by *auto*

end
theory *SymbolicPrimitive*
imports *Run*

begin
datatype *cnt-expr* =
TickCountLess ‹*clock*› ‹*instant-index*› ($\#^<$)
| *TickCountLeq* ‹*clock*› ‹*instant-index*› ($\#^\leq$)

3.5.1 Symbolic Primitives for Runs

datatype *'τ constr* =
  Timestamp ‹*clock*› ‹*instant-index*› ‹*'τ tag-const*› (- ⇓ - @ -)
 | *TimeDelay* ‹*clock*› ‹*instant-index*› ‹*'τ tag-const*› ‹*clock*› (- @ - ⊕ - ⇒ -)
 | *Ticks* ‹*clock*› ‹*instant-index*› (- ⇑ -)
 | *NotTicks* ‹*clock*› ‹*instant-index*› (- ¬⇑ -)
 | *NotTicksUntil* ‹*clock*› ‹*instant-index*› (- ¬⇑ < -)
 | *NotTicksFrom* ‹*clock*› ‹*instant-index*› (- ¬⇑ ≥ -)
 | *TagArith* ‹*tag-var*› ‹*tag-var*› ‹(*'τ tag-const* × *'τ tag-const*) ⇒ *bool*› (⌊-, -⌋ ∈ -)
 | *TickCntArith* ‹*cnt-expr*› ‹*cnt-expr*› ‹(*nat* × *nat*) ⇒ *bool*› (⌈-, -⌉ ∈ -)
 | *TickCntLeq* ‹*cnt-expr*› ‹*cnt-expr*› (- ≼ -)

type-synonym *'τ system* = ‹*'τ constr list*›

— The abstract machine follows the intuition: past [@termΓ], current index [n], present [@termΨ], future [@termΦ] Beware: This type is slightly different from the one originally implemented in Heron
type-synonym *'τ config* = ‹*'τ system* ∗ *instant-index* ∗ *'τ TESL-formula* ∗ *'τ TESL-formula*›

3.6 Semantics of Primitive Constraints

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* ⊢ #$^{\leq}$ *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* @ n_0 ⊕ *δt* ⇒ *K'* ⟧$_{prim}$ = { *ϱ*. ∀ *n* ≥ n_0. *first-time ϱ K n* (*time* ((*Rep-run ϱ*) n_0 *K*) + *δt*) ⟶ *hamlet* ((*Rep-run ϱ*) *n K'*)}›
 | ‹⟦ *K* ¬⇑ *n* ⟧$_{prim}$ = { *ϱ*. ¬*hamlet* ((*Rep-run ϱ*) *n K*) }›
 | ‹⟦ *K* ¬⇑ < *n* ⟧$_{prim}$ = { *ϱ*. ∀ *i* < *n*. ¬ *hamlet* ((*Rep-run ϱ*) *i K*)}›
 | ‹⟦ *K* ¬⇑ ≥ *n* ⟧$_{prim}$ = { *ϱ*. ∀ *i* ≥ *n*. ¬ *hamlet* ((*Rep-run ϱ*) *i K*) }›
 | ‹⟦ *K* ⇓ *n* @ *τ* ⟧$_{prim}$ = { *ϱ*. *time* ((*Rep-run ϱ*) *n K*) = *τ* }›
 | ‹⟦ ⌊τ_{var}(K_1, n_1), τ_{var}(K_2, n_2)⌋ ∈ *R* ⟧$_{prim}$ = { *ϱ*. *R* (*time* ((*Rep-run ϱ*) n_1 K_1), *time* ((*Rep-run ϱ*) n_2 K_2)) }›
 | ‹⟦ ⌈e_1, e_2⌉ ∈ *R* ⟧$_{prim}$ = { *ϱ*. *R* (⟦ *ϱ* ⊢ e_1 ⟧$_{cntexpr}$, ⟦ *ϱ* ⊢ e_2 ⟧$_{cntexpr}$) }›
 | ‹⟦ *cnt-e*$_1$ ≼ *cnt-e*$_2$ ⟧$_{prim}$ = { *ϱ*. ⟦ *ϱ* ⊢ *cnt-e*$_1$ ⟧$_{cntexpr}$ ≤ ⟦ *ϱ* ⊢ *cnt-e*$_2$ ⟧$_{cntexpr}$ }›

fun *symbolic-run-interpretation* :: ‹(*'τ*::*linordered-field*) *constr list* ⇒ (*'τ*::*linordered-field*) *run set*› (⟦⟦ - ⟧⟧$_{prim}$) **where**

⟨$[\![[]]\!]_{prim} = \{$ -. *True* $\}$⟩
| ⟨$[\![\gamma \# \Gamma]\!]_{prim} = [\![\gamma]\!]_{prim} \cap [\![\Gamma]\!]_{prim}$⟩

lemma *symbolic-run-interp-cons-morph*:
⟨$[\![\gamma]\!]_{prim} \cap [\![\Gamma]\!]_{prim} = [\![\gamma \# \Gamma]\!]_{prim}$⟩
by *auto*

definition *consistent-context* :: ⟨('τ::*linordered-field*) *constr list* ⇒ *bool*⟩ **where**
⟨*consistent-context* $\Gamma \equiv \exists \varrho.\ \varrho \in [\![\Gamma]\!]_{prim}$⟩

3.6.1 Defining a method for witness construction

— Initial states
abbreviation *initial-run* :: ⟨('τ::*linordered-field*) *run*⟩ ($\varrho_\odot$) **where**
⟨$\varrho_\odot \equiv$ *Abs-run* ((λ- -. (*False*, τ_{cst} *0*)) ::*nat* ⇒ *clock* ⇒ (*bool* × 'τ *tag-const*))⟩

— To ensure monotonicity, time tag is set at a specific instant and forever after (stuttering)
fun *time-update*
:: ⟨*nat* ⇒ *clock* ⇒ ('τ::*linordered-field*) *tag-const* ⇒ (*nat* ⇒ *clock* ⇒ (*bool* × 'τ *tag-const*)) ⇒ (*nat* ⇒ *clock* ⇒ (*bool* × 'τ *tag-const*))⟩ **where**
⟨*time-update n K* τ ϱ = (λ*n' K'*. *if K* = *K'* ∧ *n* ≤ *n' then* (*hamlet* (ϱ *n K*), τ) *else* ϱ *n' K'*)⟩

3.7 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⟩

lemma *context-consistency-preservationE*:
assumes *consist*: ⟨*consistent-context* Γ⟩
and *indepen*: ⟨γ ⋈ Γ⟩
shows ⟨*consistent-context* (γ # Γ)⟩
oops

3.8 Major Theorems

3.8.1 Fixpoint lemma

theorem *symrun-interp-fixpoint*:
⟨$\bigcap$ (($\lambda\gamma$. [[γ]]$_{prim}$) ' *set* Γ) = [[[Γ]]]$_{prim}$⟩
proof (*induct* Γ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Γ)
then show *?case* **by** *auto*
qed

3.8.2 Expansion law

Similar to the expansion laws of lattices

theorem *symrun-interp-expansion*:
shows ⟨[[[Γ_1 @ Γ_2]]]$_{prim}$ = [[[Γ_1]]]$_{prim}$ ∩ [[[Γ_2]]]$_{prim}$⟩
by (*induction* Γ_1, *auto*)

3.9 Equational laws for TESL formulae denotationally interpreted

3.9.1 General laws

lemma *symrun-interp-assoc*:
shows ⟨[[[(Γ_1 @ Γ_2) @ Γ_3]]]$_{prim}$ = [[[Γ_1 @ (Γ_2 @ Γ_3)]]]$_{prim}$⟩
by *auto*

lemma *symrun-interp-commute*:
shows ⟨[[[Γ_1 @ Γ_2]]]$_{prim}$ = [[[Γ_2 @ Γ_1]]]$_{prim}$⟩
by (*simp add*: *symrun-interp-expansion inf-sup-aci*(*1*))

lemma *symrun-interp-left-commute*:
shows ⟨[[[Γ_1 @ (Γ_2 @ Γ_3)]]]$_{prim}$ = [[[Γ_2 @ (Γ_1 @ Γ_3)]]]$_{prim}$⟩
unfolding *symrun-interp-expansion* **by** *auto*

lemma *symrun-interp-idem*:
shows ⟨[[[Γ @ Γ]]]$_{prim}$ = [[[Γ]]]$_{prim}$⟩
using *symrun-interp-expansion* **by** *auto*

lemma *symrun-interp-left-idem*:
shows ⟨[[[Γ_1 @ (Γ_1 @ Γ_2)]]]$_{prim}$ = [[[Γ_1 @ Γ_2]]]$_{prim}$⟩
using *symrun-interp-expansion* **by** *auto*

lemma *symrun-interp-right-idem*:
shows ⟨[[[(Γ_1 @ Γ_2) @ Γ_2]]]$_{prim}$ = [[[Γ_1 @ Γ_2]]]$_{prim}$⟩

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*:
shows ‹$[\![[] @ \Gamma]\!]_{prim} = [\![\Gamma]\!]_{prim}$›
by *simp*

lemma *symrun-interp-neutral2*:
shows ‹$[\![\Gamma @ []]\!]_{prim} = [\![\Gamma]\!]_{prim}$›
by *simp*

3.9.2 Decreasing interpretation of TESL formulae

lemma *TESL-sem-decreases-head*:
‹$[\![\Gamma]\!]_{prim} \supseteq [\![\gamma \# \Gamma]\!]_{prim}$›
by *simp*

lemma *TESL-sem-decreases-tail*:
‹$[\![\Gamma]\!]_{prim} \supseteq [\![\Gamma @ [\gamma]]\!]_{prim}$›
by (*simp add*: *symrun-interp-expansion*)

lemma *symrun-interp-formula-stuttering*:
assumes *bel*: ‹$\gamma \in set\ \Gamma$›
shows ‹$[\![\gamma \# \Gamma]\!]_{prim} = [\![\Gamma]\!]_{prim}$›
by (*metis Int-absorb1 Int-left-commute bel inf-le1 split-list symbolic-run-interpretation.simps*(*2*) *symrun-interp-expansion*)

lemma *symrun-interp-decreases*:
shows ‹$[\![\Gamma]\!]_{prim} \supseteq [\![\gamma \# \Gamma]\!]_{prim}$›
by (*rule TESL-sem-decreases-head*)

lemma *symrun-interp-remdups-absorb*:
shows ‹$[\![\Gamma]\!]_{prim} = [\![remdups\ \Gamma]\!]_{prim}$›
proof (*induct* Γ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Γ)
then show *?case*
using *symrun-interp-formula-stuttering* **by** *auto*
qed

lemma *symrun-interp-set-lifting*:
assumes ‹$set\ \Gamma = set\ \Gamma'$›
shows ‹$[\![\Gamma]\!]_{prim} = [\![\Gamma']\!]_{prim}$›
proof −

have ‹*set* (*remdups* Γ) = *set* (*remdups* Γ′)›
by (*simp add*: *assms*)
moreover have *fxpnt*Γ: ‹⋂ ((λγ. ⟦ γ ⟧$_{prim}$) ‘ *set* Γ) = ⟦⟦ Γ ⟧⟧$_{prim}$›
by (*simp add*: *symrun-interp-fixpoint*)
moreover have *fxpnt*Γ′: ‹⋂ ((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

theorem *symrun-interp-decreases-setinc*:
assumes *incl*: ‹*set* Γ ⊆ *set* Γ′›
shows ‹⟦⟦ Γ ⟧⟧$_{prim}$ ⊇ ⟦⟦ Γ′ ⟧⟧$_{prim}$›
proof −
obtain Γ$_r$ **where** *decompose*: ‹*set* (Γ @ Γ$_r$) = *set* Γ′› **using** *incl* **by** *auto*
have ‹*set* (Γ @ Γ$_r$) = *set* Γ′› **using** *incl decompose* **by** *blast*
moreover have ‹(*set* Γ) ∪ (*set* Γ$_r$) = *set* Γ′› **using** *incl 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 *incl*: ‹*set* Γ ⊆ *set* Γ′›
shows ‹⟦⟦ γ # Γ ⟧⟧$_{prim}$ ⊇ ⟦⟦ γ # Γ′ ⟧⟧$_{prim}$›
using *symrun-interp-decreases-setinc incl* **by** *auto*

lemma *symrun-interp-decreases-add-tail*:
assumes *incl*: ‹*set* Γ ⊆ *set* Γ′›
shows ‹⟦⟦ Γ @ [γ] ⟧⟧$_{prim}$ ⊇ ⟦⟦ Γ′ @ [γ] ⟧⟧$_{prim}$›
by (*metis symrun-interp-commute symrun-interp-decreases-add-head append-Cons append-Nil incl*)

lemma *symrun-interp-absorb1*:
assumes *incl*: ‹*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 incl*)

lemma *symrun-interp-absorb2*:
assumes *incl*: ‹*set* Γ$_2$ ⊆ *set* Γ$_1$›
shows ‹⟦⟦ Γ$_1$ @ Γ$_2$ ⟧⟧$_{prim}$ = ⟦⟦ Γ$_1$ ⟧⟧$_{prim}$›
using *symrun-interp-absorb1 symrun-interp-commute incl* **by** *blast*

end

Chapter 4

Operational Semantics

theory *Operational*
imports
SymbolicPrimitive

begin

4.1 Operational steps

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)

inductive *operational-semantics-intro* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config*
⇒ *bool* (- ↪$_i$ - *70*) **where**
instant-i:
(Γ, n ⊢ [] ▷ Φ)
↪$_i$ (Γ, *Suc* n ⊢ Φ ▷ [])

inductive *operational-semantics-elim* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒
bool (- ↪$_e$ - *70*) **where**
sporadic-on-e1:
(Γ, n ⊢ ((K_1 *sporadic* τ *on* K_2) # Ψ) ▷ Φ)
↪$_e$ (Γ, n ⊢ Ψ ▷ ((K_1 *sporadic* τ *on* K_2) # Φ))
| *sporadic-on-e2*:
(Γ, n ⊢ ((K_1 *sporadic* τ *on* K_2) # Ψ) ▷ Φ)
↪$_e$ (((K_1 ⇑ n) # (K_2 ⇓ n @ τ) # Γ), n ⊢ Ψ ▷ Φ)
| *tagrel-e*:
(Γ, n ⊢ ((*time−relation* ⌊K_1, K_2⌋ ∈ R) # Ψ) ▷ Φ)
↪$_e$ ((($\lfloor\tau_{var}(K_1, n), \tau_{var}(K_2, n)\rfloor \in R$) # Γ), n ⊢ Ψ ▷ ((*time−relation* ⌊K_1, K_2⌋ ∈ R) # Φ))
| *implies-e1*:
(Γ, n ⊢ ((K_1 *implies* K_2) # Ψ) ▷ Φ)
↪$_e$ (((K_1 ¬⇑ n) # Γ), n ⊢ Ψ ▷ ((K_1 *implies* K_2) # Φ))

| *implies-e2*:
(Γ, $n \vdash ((K_1$ *implies* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \Uparrow n) \# (K_2 \Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *implies* $K_2) \# \Phi))$
| *implies-not-e1*:
(Γ, $n \vdash ((K_1$ *implies not* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2) \# \Phi))$
| *implies-not-e2*:
(Γ, $n \vdash ((K_1$ *implies not* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \Uparrow n) \# (K_2 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2) \# \Phi))$
| *timedelayed-e1*:
(Γ, $n \vdash ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3)$ $\# \Phi))$
| *timedelayed-e2*:
(Γ, $n \vdash ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \Uparrow n) \# (K_2 @ n \oplus \delta\tau \Rightarrow K_3) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Phi))$
| *weakly-precedes-e*:
(Γ, $n \vdash ((K_1$ *weakly precedes* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((\lceil \#^{\leq} K_2\ n, \#^{\leq} K_1\ n \rceil \in (\lambda(x,y).\ x \leq y)) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *weakly precedes* $K_2) \# \Phi))$
| *strictly-precedes-e*:
(Γ, $n \vdash ((K_1$ *strictly precedes* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((\lceil \#^{\leq} K_2\ n, \#^{<} K_1\ n \rceil \in (\lambda(x,y).\ x \leq y)) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *strictly precedes* $K_2) \# \Phi))$
| *kills-e1*:
(Γ, $n \vdash ((K_1$ *kills* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *kills* $K_2) \# \Phi))$
| *kills-e2*:
(Γ, $n \vdash ((K_1$ *kills* $K_2) \# \Psi) \triangleright \Phi$)
$\hookrightarrow_e$ $(((K_1 \Uparrow n) \# (K_2 \neg\Uparrow \geq n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *kills* $K_2) \# \Phi))$

inductive *operational-semantics-step* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒ *bool* (- ↪ - *70*) **where**
intro-part: $(\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow_i (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$
$\Longrightarrow (\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$
| *elims-part*: $(\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow_e (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$
$\Longrightarrow (\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$

abbreviation *operational-semantics-step-rtranclp* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒ *bool* (- ↪** - *70*) **where**
$\mathcal{C}_1 \hookrightarrow^{**} \mathcal{C}_2 \equiv$ *operational-semantics-step*** $\mathcal{C}_1\ \mathcal{C}_2$

abbreviation *operational-semantics-step-tranclp* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒ *bool* (- ↪++ - *70*) **where**
$\mathcal{C}_1 \hookrightarrow^{++} \mathcal{C}_2 \equiv$ *operational-semantics-step*$^{++}$ $\mathcal{C}_1\ \mathcal{C}_2$

abbreviation *operational-semantics-step-reflclp* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒ *bool* (- ↪== - *70*) **where**

$\mathcal{C}_1 \hookrightarrow^{==} \mathcal{C}_2 \equiv$ *operational-semantics-step*$^{==}$ $\mathcal{C}_1$ $\mathcal{C}_2$

abbreviation *operational-semantics-step-relpowp* :: ('τ::*linordered-field*) *config* ⇒ *nat* ⇒ 'τ *config* ⇒ *bool* (- ↪$^{-}$ - *70*) **where**
$\mathcal{C}_1 \hookrightarrow^n \mathcal{C}_2 \equiv$ (*operational-semantics-step* ^^ *n*) $\mathcal{C}_1$ $\mathcal{C}_2$

definition *operational-semantics-elim-inv* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config* ⇒ *bool* (- ↪$_e^{\leftarrow}$ - *70*) **where**
$\mathcal{C}_1 \hookrightarrow_e^{\leftarrow} \mathcal{C}_2 \equiv \mathcal{C}_2 \hookrightarrow_e \mathcal{C}_1$

4.2 Basic Lemmas

lemma *operational-semantics-trans-generalized*:
assumes $\mathcal{C}_1 \hookrightarrow^n \mathcal{C}_2$
assumes $\mathcal{C}_2 \hookrightarrow^m \mathcal{C}_3$
shows $\mathcal{C}_1 \hookrightarrow^{n\,+\,m} \mathcal{C}_3$
by (*metis* (*no-types*, *hide-lams*) *assms*(*1*) *assms*(*2*) *relcompp.relcompI relpowp-add*)

abbreviation *Cnext-solve* :: ('τ::*linordered-field*) *config* ⇒ 'τ *config set* ($\mathcal{C}_{next}$ -) **where**
$\mathcal{C}_{next}\ \mathcal{S} \equiv \{\ \mathcal{S}'.\ \mathcal{S} \hookrightarrow \mathcal{S}'\ \}$

lemma *Cnext-solve-instant*:
shows ($\mathcal{C}_{next}$ (Γ, $n \vdash [] \triangleright \Phi$))
⊇ { Γ, *Suc* $n \vdash \Phi \triangleright []$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-intro.instant-i*)

lemma *Cnext-solve-sporadicon*:
shows ($\mathcal{C}_{next}$ (Γ, $n \vdash ((K_1$ *sporadic* τ *on* $K_2) \#\ \Psi) \triangleright \Phi$))
⊇ { Γ, $n \vdash \Psi \triangleright ((K_1$ *sporadic* τ *on* $K_2) \#\ \Phi)$,
$((K_1 \Uparrow n) \#\ (K_2 \Downarrow n\ @\ \tau) \#\ \Gamma)$, $n \vdash \Psi \triangleright \Phi$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.sporadic-on-e1 operational-semantics-elim.sporadic-on-e2*)

lemma *Cnext-solve-tagrel*:
shows ($\mathcal{C}_{next}$ (Γ, $n \vdash ((time{-}relation\ \lfloor K_1, K_2 \rfloor \in R) \#\ \Psi) \triangleright \Phi$))
⊇ { $((\lfloor \tau_{var}(K_1, n), \tau_{var}(K_2, n) \rfloor \in R) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((time{-}relation\ \lfloor K_1, K_2 \rfloor \in R) \#\ \Phi)$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.tagrel-e*)

lemma *Cnext-solve-implies*:
shows ($\mathcal{C}_{next}$ (Γ, $n \vdash ((K_1$ *implies* $K_2) \#\ \Psi) \triangleright \Phi$))
⊇ { $((K_1 \neg\Uparrow n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1$ *implies* $K_2) \#\ \Phi)$,
$((K_1 \Uparrow n) \#\ (K_2 \Uparrow n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1$ *implies* $K_2) \#\ \Phi)$}
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.implies-e1 operational-semantics-elim.implies-e2*)

lemma *Cnext-solve-implies-not*:
shows ($\mathcal{C}_{next}$ (Γ, $n \vdash ((K_1$ *implies not* $K_2) \#\ \Psi) \triangleright \Phi$))

$\supseteq$ { $((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2) \# \Phi)$,
$((K_1 \Uparrow n) \# (K_2 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2) \# \Phi)$}
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.implies-not-e1 operational-semantics-elim.implies-not-e2*)

lemma *Cnext-solve-timedelayed*:
shows $(\mathcal{C}_{next}\ (\Gamma, n \vdash ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Psi) \triangleright \Phi))$
$\supseteq$ { $((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Phi)$,
$((K_1 \Uparrow n) \# (K_2 @ n \oplus \delta\tau \Rightarrow K_3) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3) \# \Phi)$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.timedelayed-e1 operational-semantics-elim.timedelayed-e2*)

lemma *Cnext-solve-weakly-precedes*:
shows $(\mathcal{C}_{next}\ (\Gamma, n \vdash ((K_1$ *weakly precedes* $K_2) \# \Psi) \triangleright \Phi))$
$\supseteq$ { $((\lceil \#^{\leq}\ K_2\ n, \#^{\leq}\ K_1\ n \rceil \in (\lambda(x,y).\ x \leq y)) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *weakly precedes* $K_2) \# \Phi)$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.weakly-precedes-e*)

lemma *Cnext-solve-strictly-precedes*:
shows $(\mathcal{C}_{next}\ (\Gamma, n \vdash ((K_1$ *strictly precedes* $K_2) \# \Psi) \triangleright \Phi))$
$\supseteq$ { $((\lceil \#^{\leq}\ K_2\ n, \#^{<}\ K_1\ n \rceil \in (\lambda(x,y).\ x \leq y)) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *strictly precedes* $K_2) \# \Phi)$ }
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.strictly-precedes-e*)

lemma *Cnext-solve-kills*:
shows $(\mathcal{C}_{next}\ (\Gamma, n \vdash ((K_1$ *kills* $K_2) \# \Psi) \triangleright \Phi))$
$\supseteq$ { $((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *kills* $K_2) \# \Phi)$,
$((K_1 \Uparrow n) \# (K_2 \neg\Uparrow \geq n) \# \Gamma), n \vdash \Psi \triangleright ((K_1$ *kills* $K_2) \# \Phi)$}
by (*simp add*: *operational-semantics-step.simps operational-semantics-elim.kills-e1 operational-semantics-elim.kills-e2*)

lemma *empty-spec-reductions*:
shows $([], 0 \vdash [] \triangleright []) \hookrightarrow^k ([], k \vdash [] \triangleright [])$
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
then show *?case*
using *instant-i operational-semantics-step.simps* **by** *fastforce*
qed

end

Chapter 5

Equivalence of Operational and Denotational Semantics

theory *Corecursive-Prop*
imports
SymbolicPrimitive
Operational
Denotational

begin

5.1 Stepwise denotational interpretation of TESL atoms

Denotational interpretation of TESL bounded by index

fun *TESL-interpretation-atomic-stepwise*
:: ‹('τ::*linordered-field*) *TESL-atomic* ⇒ *nat* ⇒ 'τ *run set*› (⟦ - ⟧$_{TESL}^{\geq}$ -)
where
‹⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq i}$ =
{ ϱ. ∃ n≥i. *hamlet* ((*Rep-run* ϱ) *n* K_1) = *True* ∧ *time* ((*Rep-run* ϱ) *n* K_2) = τ }›
| ‹⟦ *time−relation* ⌊K_1, K_2⌋ ∈ *R* ⟧$_{TESL}^{\geq i}$ =
{ ϱ. ∀ n≥i. *R* (*time* ((*Rep-run* ϱ) *n* K_1), *time* ((*Rep-run* ϱ) *n* K_2)) }›
| ‹⟦ *master implies slave* ⟧$_{TESL}^{\geq i}$ =
{ ϱ. ∀ n≥i. *hamlet* ((*Rep-run* ϱ) *n master*) ⟶ *hamlet* ((*Rep-run* ϱ) *n slave*) }›
| ‹⟦ *master implies not slave* ⟧$_{TESL}^{\geq i}$ =
{ ϱ. ∀ n≥i. *hamlet* ((*Rep-run* ϱ) *n master*) ⟶ ¬ *hamlet* ((*Rep-run* ϱ) *n slave*) }›
| ‹⟦ *master time−delayed by* δτ *on measuring implies slave* ⟧$_{TESL}^{\geq i}$ =
{ 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)

$\longrightarrow$ *hamlet* ((*Rep-run* ϱ) m *slave*)
)
}⟩
| ⟨⟦ K_1 *weakly precedes* K_2 ⟧$_{TESL}^{\geq i}$ =
{ ϱ. $\forall n{\geq}i$. (*run-tick-count* ϱ K_2 n) $\leq$ (*run-tick-count* ϱ K_1 n) }⟩
| ⟨⟦ K_1 *strictly precedes* K_2 ⟧$_{TESL}^{\geq i}$ =
{ ϱ. $\forall n{\geq}i$. (*run-tick-count* ϱ K_2 n) $\leq$ (*run-tick-count-strictly* ϱ K_1 n) }⟩
| ⟨⟦ K_1 *kills* K_2 ⟧$_{TESL}^{\geq i}$ =
{ ϱ. $\forall n{\geq}i$. *hamlet* ((*Rep-run* ϱ) n K_1) $\longrightarrow$ ($\forall m{\geq}n$. $\neg$ *hamlet* ((*Rep-run* ϱ) m K_2)) }⟩

theorem *predicate-Inter-unfold*:
⟨{ ϱ. $\forall n$. P ϱ n} = $\bigcap$ {Y. $\exists n$. Y = { ϱ. P ϱ n }}⟩
by (*simp add*: *Collect-all-eq full-SetCompr-eq*)

theorem *predicate-Union-unfold*:
⟨{ ϱ. $\exists n$. P ϱ n} = $\bigcup$ {Y. $\exists n$. Y = { ϱ. P ϱ n }}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-sporadicon*:
shows ⟨⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}$ = $\bigcup$ {Y. $\exists n$::*nat*. Y = ⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq n}$}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-tagrelgen*:
shows ⟨⟦ *time−relation* ⌊K_1, K_2⌋ $\in$ R ⟧$_{TESL}$ = $\bigcap$ {Y. $\exists n$::*nat*. Y = ⟦ *time−relation* ⌊K_1, K_2⌋ $\in$ R ⟧$_{TESL}^{\geq n}$}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-implies*:
shows ⟨⟦ *master implies slave* ⟧$_{TESL}$ = $\bigcap$ {Y. $\exists n$::*nat*. Y = ⟦ *master implies slave* ⟧$_{TESL}^{\geq n}$}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-implies-not*:
shows ⟨⟦ *master implies not slave* ⟧$_{TESL}$ = $\bigcap$ {Y. $\exists n$::*nat*. Y = ⟦ *master implies not slave* ⟧$_{TESL}^{\geq n}$}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-timedelayed*:
shows ⟨⟦ *master time−delayed by* $\delta\tau$ *on measuring implies slave* ⟧$_{TESL}$
= $\bigcap$ {Y. $\exists n$::*nat*. Y = ⟦ *master time−delayed by* $\delta\tau$ *on measuring implies slave* ⟧$_{TESL}^{\geq n}$}⟩
by *auto*

lemma *TESL-interp-unfold-stepwise-weakly-precedes*:
shows ⟨⟦ K_1 *weakly precedes* K_2 ⟧$_{TESL}$ = $\bigcap$ {Y. $\exists n$::*nat*. Y = ⟦ K_1 *weakly precedes* K_2 ⟧$_{TESL}^{\geq n}$}⟩

by *auto*

lemma *TESL-interp-unfold-stepwise-strictly-precedes*:
shows ‹$[\![K_1$ *strictly precedes* $K_2]\!]_{TESL} = \bigcap \{Y.\ \exists n{::}nat.\ Y = [\![K_1$ *strictly precedes* $K_2]\!]_{TESL}{}^{\geq n}\}$›
by *auto*

lemma *TESL-interp-unfold-stepwise-kills*:
shows ‹$[\![$ *master kills slave* $]\!]_{TESL} = \bigcap \{Y.\ \exists n{::}nat.\ Y = [\![$ *master kills slave* $]\!]_{TESL}{}^{\geq n}\}$›
by *auto*

theorem *TESL-interp-unfold-stepwise-positive-atoms*:
assumes ‹*positive-atom* φ›
shows ‹$[\![\varphi{::}'\tau{::}$*linordered-field TESL-atomic* $]\!]_{TESL} = \bigcup \{Y.\ \exists n{::}nat.\ Y = [\![\varphi]\!]_{TESL}{}^{\geq n}\}$›
by (*metis TESL-interp-unfold-stepwise-sporadicon assms positive-atom.elims*(2))

theorem *TESL-interp-unfold-stepwise-negative-atoms*:
assumes ‹$\neg$ *positive-atom* φ›
shows ‹$[\![\varphi]\!]_{TESL} = \bigcap \{Y.\ \exists n{::}nat.\ Y = [\![\varphi]\!]_{TESL}{}^{\geq n}\}$›
proof (*cases* φ)
case *SporadicOn* **thus** *?thesis* **using** *assms* **by** *simp*
next
case (*TagRelation x41 x42 x43*)
thus *?thesis* **using** *TESL-interp-unfold-stepwise-tagrelgen* **by** *simp*
next
case (*Implies x51 x52*)
thus *?thesis* **using** *TESL-interp-unfold-stepwise-implies* **by** *simp*
next
case (*ImpliesNot x51 x52*)
thus *?thesis* **using** *TESL-interp-unfold-stepwise-implies-not* **by** *simp*
next
case (*TimeDelayedBy x61 x62 x63 x64*)
thus *?thesis* **using** *TESL-interp-unfold-stepwise-timedelayed* **by** *simp*
next
case (*WeaklyPrecedes x61 x62*)
then show *?thesis*
using *TESL-interp-unfold-stepwise-weakly-precedes* **by** *simp*
next
case (*StrictlyPrecedes x61 x62*)
then show *?thesis*
using *TESL-interp-unfold-stepwise-strictly-precedes* **by** *simp*
next
case (*Kills x63 x64*)
then show *?thesis*
using *TESL-interp-unfold-stepwise-kills* **by** *simp*
qed

lemma *forall-nat-expansion*:
‹$(\forall\, n_1 \geq (n_0{::}nat).\ P\ n_1) = (P\ n_0 \wedge (\forall\, n_1 \geq Suc\ n_0.\ P\ n_1))$›
by (*metis Suc-le-eq le-less*)

lemma *exists-nat-expansion*:
‹$(\exists\, n_1 \geq (n_0{::}nat).\ P\ n_1) = (P\ n_0 \vee (\exists\, n_1 \geq Suc\ n_0.\ P\ n_1))$›
proof (*cases* ‹$P\ n_0$›)
case *True*
thus *?thesis* **by** *auto*
next
case *False*
thus *?thesis* **by** (*metis Suc-le-eq le-less*)
qed

5.2 Coinduction Unfolding Properties

lemma *TESL-interp-stepwise-sporadicon-cst-coind-unfold*:
shows ‹$[\![\ K_1\ sporadic\ \tau\ on\ K_2\ ]\!]_{TESL}{}^{\geq n} =$
$[\![\ K_1 \Uparrow n\ ]\!]_{prim} \cap [\![\ K_2 \Downarrow n\ @\ \tau\ ]\!]_{prim}$
$\cup\ [\![\ K_1\ sporadic\ \tau\ on\ K_2\ ]\!]_{TESL}{}^{\geq Suc\ n}$›
proof −
have ‹$\{\ \varrho.\ \exists\, m{\geq}n.\ hamlet\ ((Rep\text{-}run\ \varrho)\ m\ K_1) = True \wedge time\ ((Rep\text{-}run\ \varrho)\ m\ K_2) = \tau\ \}$
$= \{\ \varrho.\ hamlet\ ((Rep\text{-}run\ \varrho)\ n\ K_1) = True \wedge time\ ((Rep\text{-}run\ \varrho)\ n\ K_2) = \tau$
$\vee\ (\exists\, m{\geq}Suc\ n.\ hamlet\ ((Rep\text{-}run\ \varrho)\ m\ K_1) = True \wedge time\ ((Rep\text{-}run\ \varrho)\ m\ K_2) = \tau)\ \}$›
using *Suc-leD not-less-eq-eq* **by** *fastforce*
moreover have ‹$\{\ \varrho.\ hamlet\ ((Rep\text{-}run\ \varrho)\ n\ K_1) = True \wedge time\ ((Rep\text{-}run\ \varrho)\ n\ K_2) = \tau$
$\vee\ (\exists\, m{\geq}Suc\ n.\ hamlet\ ((Rep\text{-}run\ \varrho)\ m\ K_1) = True \wedge time\ ((Rep\text{-}run\ \varrho)\ m\ K_2) = \tau)\ \}$
$= [\![\ K_1 \Uparrow n\ ]\!]_{prim} \cap [\![\ K_2 \Downarrow n\ @\ \tau\ ]\!]_{prim} \cup [\![\ K_1\ sporadic\ \tau\ on\ K_2\ ]\!]_{TESL}{}^{\geq Suc\ n}$›
by (*simp add*: *Collect-conj-eq Collect-disj-eq*)
ultimately show *?thesis* **by** *auto*
qed

lemma TESL-interp-stepwise-sporadicon-cst-coind-unfold: shows ‹[\![K_1 sporadic τ on K_2]\!]_TESL^≥ n = [\![K_1 ⇑ n]\!]_prim ∩ [\![K_2 ⇓ n @ τ]\!]_prim ∪ [\![K_1 sporadic τ on K_2]\!]_TESL^≥ Suc n› proof have ‹{ ϱ. ∃m≥n. hamlet ((Rep-run ϱ) m K_1) = True ∧ time ((Rep-run ϱ) m K_2) = τ } = { ϱ. hamlet ((Rep-run ϱ) n K_1) = True ∧ time ((Rep-run ϱ) n K_2) = τ ∨ (∃m≥Suc n. hamlet ((Rep-run ϱ) m K_1) = True ∧ time ((Rep-run ϱ) m K_2) = τ) }› using Suc-leD not-less-eq-eq by fastforce then show ?thesis by auto qed

lemma *TESL-interp-stepwise-sporadicon-coind-unfold*:

shows ‹⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq n}$ =
⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦ $K_2 \Downarrow n$ @ τ ⟧$_{prim}$
∪ ⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq Suc\ n}$›
using *TESL-interp-stepwise-sporadicon-cst-coind-unfold* **by** *blast*

lemma *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 *TESL-interp-stepwise-tagrel-coind-unfold*:
shows ‹⟦ *time−relation* ⌊K_1, K_2⌋ ∈ *R* ⟧$_{TESL}^{\geq n}$ =
⟦ ⌊$\tau_{var}(K_1, n)$, $\tau_{var}(K_2, n)$⌋ ∈ *R* ⟧$_{prim}$
∩ ⟦ *time−relation* ⌊K_1, K_2⌋ ∈ *R* ⟧$_{TESL}^{\geq Suc\ n}$›
proof −
have ‹{ ϱ. ∀m≥n. *R* (*time* ((*Rep-run* ϱ) *m* K_1), *time* ((*Rep-run* ϱ) *m* K_2)) }
= { ϱ. *R* (*time* ((*Rep-run* ϱ) *n* K_1), *time* ((*Rep-run* ϱ) *n* K_2)) }
∩ { ϱ. ∀m≥Suc n. *R* (*time* ((*Rep-run* ϱ) *m* K_1), *time* ((*Rep-run* ϱ) *m* K_2))
}›
using *nat-set-suc*[*of* ‹*n*› ‹λx y. *R* (*time* ((*Rep-run x*) *y* K_1), *time* ((*Rep-run x*) *y* K_2))›] **by** *simp*
then show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-implies-coind-unfold*:
shows ‹⟦ *master implies slave* ⟧$_{TESL}^{\geq n}$ =
(⟦ *master* ¬⇑ *n* ⟧$_{prim}$ ∪ ⟦ *master* ⇑ *n* ⟧$_{prim}$ ∩ ⟦ *slave* ⇑ *n* ⟧$_{prim}$)
∩ ⟦ *master implies slave* ⟧$_{TESL}^{\geq Suc\ n}$›
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 *nat-set-suc*[*of* ‹*n*› ‹λx y. *hamlet* ((*Rep-run x*) *y master*) ⟶ *hamlet*

((*Rep-run x*) *y slave*)⟩] **by** *simp*
then show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-implies-not-coind-unfold*:
shows ⟨⟦ *master implies not slave* ⟧$_{TESL}^{\geq n}$ =
(⟦ *master* ¬⇑ *n* ⟧$_{prim}$ ∪ ⟦ *master* ⇑ *n* ⟧$_{prim}$ ∩ ⟦ *slave* ¬⇑ *n* ⟧$_{prim}$)
∩ ⟦ *master implies not slave* ⟧$_{TESL}^{\geq Suc\ n}$⟩
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 *nat-set-suc*[*of* ⟨*n*⟩ ⟨λ*x y*. *hamlet* ((*Rep-run x*) *y master*) ⟶ ¬ *hamlet* ((*Rep-run x*) *y slave*)⟩] **by** *simp*
then show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-timedelayed-coind-unfold*:
shows ⟨⟦ *master time−delayed by* δτ *on measuring implies slave* ⟧$_{TESL}^{\geq n}$ =
(⟦ *master* ¬⇑ *n* ⟧$_{prim}$ ∪ (⟦ *master* ⇑ *n* ⟧$_{prim}$ ∩ ⟦ *measuring* @ *n* ⊕ δτ ⇒ *slave* ⟧$_{prim}$))
∩ ⟦ *master time−delayed by* δτ *on measuring implies slave* ⟧$_{TESL}^{\geq Suc\ n}$⟩
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 *nat-set-suc*[*of* ⟨*n*⟩ *?prop*] **by** *blast*
also have ⟨... = { ϱ. *?prop* ϱ *n* } ∩ ⟦ *master time−delayed by* δτ *on measuring implies slave* ⟧$_{TESL}^{\geq Suc\ n}$⟩ **by** *simp*
finally show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-weakly-precedes-coind-unfold*:
shows ⟨⟦ K_1 *weakly precedes* K_2 ⟧$_{TESL}^{\geq n}$ =
⟦ (⌈#$^{\leq}$ K_2 *n*, #$^{\leq}$ K_1 *n*⌉ ∈ (λ(*x*,*y*). *x*≤*y*)) ⟧$_{prim}$
∩ ⟦ K_1 *weakly precedes* K_2 ⟧$_{TESL}^{\geq Suc\ n}$⟩
proof −
have ⟨{ ϱ. ∀ *p*≥*n*. (*run-tick-count* ϱ K_2 *p*) ≤ (*run-tick-count* ϱ K_1 *p*) }
= { ϱ. (*run-tick-count* ϱ K_2 *n*) ≤ (*run-tick-count* ϱ K_1 *n*) }
∩ { ϱ. ∀ *p*≥*Suc n*. (*run-tick-count* ϱ K_2 *p*) ≤ (*run-tick-count* ϱ K_1 *p*) }⟩
using *nat-set-suc*[*of* ⟨*n*⟩ ⟨λϱ *n*. (*run-tick-count* ϱ K_2 *n*) ≤ (*run-tick-count* ϱ K_1 *n*)⟩]
by *simp*

then show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-strictly-precedes-coind-unfold*:
shows ‹⟦ K_1 *strictly precedes* K_2 ⟧$_{TESL}^{\geq n}$ =
⟦ (⌈#$^{\leq}$ K_2 n, #$^{<}$ K_1 n⌉ ∈ (λ(x,y). x≤y)) ⟧$_{prim}$
∩ ⟦ K_1 *strictly precedes* K_2 ⟧$_{TESL}^{\geq Suc\ n}$›
proof −
have ‹{ ϱ. ∀p≥n. (*run-tick-count* ϱ K_2 p) ≤ (*run-tick-count-strictly* ϱ K_1 p) }
= { ϱ. (*run-tick-count* ϱ K_2 n) ≤ (*run-tick-count-strictly* ϱ K_1 n) }
∩ { ϱ. ∀p≥*Suc* n. (*run-tick-count* ϱ K_2 p) ≤ (*run-tick-count-strictly* ϱ K_1 p) }›
using *nat-set-suc*[*of* ‹n› ‹λϱ n. (*run-tick-count* ϱ K_2 n) ≤ (*run-tick-count-strictly* ϱ K_1 n)›]
by *simp*
then show *?thesis* **by** *auto*
qed

lemma *TESL-interp-stepwise-kills-coind-unfold*:
shows ‹⟦ K_1 *kills* K_2 ⟧$_{TESL}^{\geq n}$ =
(⟦ K_1 ¬↑ n ⟧$_{prim}$ ∪ ⟦ K_1 ↑ n ⟧$_{prim}$ ∩ ⟦ K_2 ¬↑ ≥ n ⟧$_{prim}$)
∩ ⟦ K_1 *kills* K_2 ⟧$_{TESL}^{\geq Suc\ n}$›
proof −
let *?kills* = ‹λn ϱ. ∀p≥n. *hamlet* ((*Rep-run* ϱ) p K_1) ⟶ (∀m≥p. ¬ *hamlet* ((*Rep-run* ϱ) m K_2))›
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 ‹⟦ K_1 *kills* K_2 ⟧$_{TESL}^{\geq n}$ = {ϱ. *?kills* n ϱ}› **by** *simp*
also have ‹... = ({ϱ. ¬ *?ticks* n ϱ K_1} ∩ { ϱ. *?kills* (*Suc* n) ϱ})
∪ ({ϱ. *?ticks* n ϱ K_1} ∩ {ϱ. *?dead* n ϱ K_2})›
proof
{ fix ϱ::‹'τ::*linordered-field run*›
assume ‹ϱ ∈ {ϱ. *?kills* n ϱ}›
hence ‹*?kills* n ϱ› **by** *simp*
hence ‹(*?ticks* n ϱ K_1 ∧ *?dead* n ϱ K_2) ∨ (¬*?ticks* n ϱ K_1 ∧ *?kills* (*Suc* n) ϱ)›
using *Suc-leD* **by** *blast*
hence ‹ϱ ∈ ({ϱ. *?ticks* n ϱ K_1} ∩ {ϱ. *?dead* n ϱ K_2})
∪ ({ϱ. ¬ *?ticks* n ϱ K_1} ∩ {ϱ. *?kills* (*Suc* n) ϱ})›
by *blast*
} thus ‹{ϱ. *?kills* n ϱ}
⊆ {ϱ. ¬ *?ticks* n ϱ K_1} ∩ {ϱ. *?kills* (*Suc* n) ϱ}
∪ {ϱ. *?ticks* n ϱ K_1} ∩ {ϱ. *?dead* n ϱ K_2}› **by** *blast*
next
{ fix ϱ::‹'τ::*linordered-field run*›
assume ‹ϱ ∈ ({ϱ. ¬ *?ticks* n ϱ K_1} ∩ { ϱ. *?kills* (*Suc* n) ϱ})
∪ ({ϱ. *?ticks* n ϱ K_1} ∩ {ϱ. *?dead* n ϱ K_2})›
hence ‹¬ *?ticks* n ϱ K_1 ∧ *?kills* (*Suc* n) ϱ
∨ *?ticks* n ϱ K_1 ∧ *?dead* n ϱ K_2› **by** *blast*

```
    hence ‹?kills n ϱ› by (metis dual-order.trans eq-iff not-less-eq-eq)
  } 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

fun TESL-interpretation-stepwise :: ‹'τ::linordered-field TESL-formula ⇒ nat ⇒
'τ run set› ([[[ - ]]]TESL^≥ -) where
   ‹[[[ [] ]]]TESL^≥ n = { -. True }›
 | ‹[[[ φ # Φ ]]]TESL^≥ n = ⟦ φ ⟧TESL^≥ n ∩ [[[ Φ ]]]TESL^≥ n›

lemma TESL-interpretation-stepwise-fixpoint:
 ‹[[[ Φ ]]]TESL^≥ n = ⋂ ((λφ. ⟦ φ ⟧TESL^≥ n) ‘ set Φ)›
 proof (induct Φ)
   case Nil
   then show ?case by simp
 next
   case (Cons a Φ)
   then show ?case by auto
 qed

lemma TESL-interpretation-stepwise-zero:
 ‹⟦ φ ⟧TESL = ⟦ φ ⟧TESL^≥ 0›
 proof (induct φ)
   case (SporadicOn K₁ τ K₂)
   then show ?case by simp
 next
   case (TagRelation x1 x2 x3)
   then show ?case by simp
 next
   case (Implies x1 x2)
   then show ?case by simp
 next
   case (ImpliesNot x1 x2)
   then show ?case by simp
 next
   case (TimeDelayedBy x1 x2 x3 x4)
   then show ?case by simp
 next
   case (WeaklyPrecedes x1 x2)
   then show ?case by simp
```

next
case (*StrictlyPrecedes x1 x2*)
then show *?case* **by** *simp*
next
case (*Kills x1 x2*)
then show *?case* **by** *simp*
qed

lemma *TESL-interpretation-stepwise-zero'*:
‹$[\![\Phi]\!]_{TESL} = [\![\Phi]\!]_{TESL}{}^{\geq 0}$›
proof (*induct* Φ)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Φ)
then show *?case*
by (*simp add*: *TESL-interpretation-stepwise-zero*)
qed

lemma *TESL-interpretation-stepwise-cons-morph*:
‹$[\![\varphi]\!]_{TESL}{}^{\geq n} \cap [\![\Phi]\!]_{TESL}{}^{\geq n} = [\![\varphi \# \Phi]\!]_{TESL}{}^{\geq n}$›
by *auto*

theorem *TESL-interp-stepwise-composition*:
shows ‹$[\![\Phi_1 @ \Phi_2]\!]_{TESL}{}^{\geq n} = [\![\Phi_1]\!]_{TESL}{}^{\geq n} \cap [\![\Phi_2]\!]_{TESL}{}^{\geq n}$›
proof (*induct* Φ_1)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a* Φ_1)
then show *?case* **by** *auto*
qed

5.3 Interpretation of configurations

fun *HeronConf-interpretation* :: ‹'τ::*linordered-field config* ⇒ 'τ *run set*› ($[\![$ - $]\!]_{config}$ *71*) **where**
‹$[\![\Gamma, n \vdash \Psi \triangleright \Phi]\!]_{config} = [\![\Gamma]\!]_{prim} \cap [\![\Psi]\!]_{TESL}{}^{\geq n} \cap [\![\Phi]\!]_{TESL}{}^{\geq Suc\ n}$›

lemma *HeronConf-interp-composition*:
shows ‹$[\![\Gamma_1, n \vdash \Psi_1 \triangleright \Phi_1]\!]_{config} \cap [\![\Gamma_2, n \vdash \Psi_2 \triangleright \Phi_2]\!]_{config}$
$= [\![(\Gamma_1 @ \Gamma_2), n \vdash (\Psi_1 @ \Psi_2) \triangleright (\Phi_1 @ \Phi_2)]\!]_{config}$›
using *TESL-interp-stepwise-composition symrun-interp-expansion*
by (*simp add*: *TESL-interp-stepwise-composition symrun-interp-expansion inf-assoc inf-left-commute*)

lemma *HeronConf-interp-stepwise-instant-cases*:
shows ‹$[\![\Gamma, n \vdash [] \triangleright \Phi]\!]_{config}$
$= [\![\Gamma, Suc\ n \vdash \Phi \triangleright []]\!]_{config}$›

proof −
have ‹⟦ Γ, $n \vdash [] \triangleright \Phi$ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ [] ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦ Γ, *Suc n* ⊢ Φ ▷ [] ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$ ∩ ⟦⟦ [] ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ [] ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$
= ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$ ∩ ⟦⟦ [] ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
ultimately show *?thesis* **by** *blast*
qed

lemma *HeronConf-interp-stepwise-sporadicon-cases*:
shows ‹⟦ Γ, $n \vdash ((K_1$ *sporadic* τ *on* $K_2) \# \Psi) \triangleright \Phi$ ⟧$_{config}$
= ⟦ Γ, $n \vdash \Psi \triangleright ((K_1$ *sporadic* τ *on* $K_2) \# \Phi)$ ⟧$_{config}$
∪ ⟦ $((K_1 \Uparrow n) \# (K_2 \Downarrow n @ \tau) \# \Gamma)$, $n \vdash \Psi \triangleright \Phi$ ⟧$_{config}$›
proof −
have ‹⟦ Γ, $n \vdash (K_1$ *sporadic* τ *on* $K_2) \# \Psi \triangleright \Phi$ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ (K_1 *sporadic* τ *on* K_2) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦ Γ, $n \vdash \Psi \triangleright ((K_1$ *sporadic* τ *on* $K_2) \# \Phi)$ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K_1 *sporadic* τ *on* K_2) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦ $((K_1 \Uparrow n) \# (K_2 \Downarrow n @ \tau) \# \Gamma), n \vdash \Psi \triangleright \Phi$ ⟧$_{config}$ = ⟦⟦ $((K_1 \Uparrow n) \# (K_2 \Downarrow n @ \tau) \# \Gamma)$ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
ultimately show *?thesis*
proof −
have ‹(⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦ $K_2 \Downarrow n @ \tau$ ⟧$_{prim}$ ∪ ⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq Suc\ n}$) ∩ (⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$) = ⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq n}$ ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$)›
using *TESL-interp-stepwise-sporadicon-coind-unfold* **by** *blast*
then have ‹⟦⟦ $((K_1 \Uparrow n) \# (K_2 \Downarrow n @ \tau) \# \Gamma)$ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∪ ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦ K_1 *sporadic* τ *on* K_2 ⟧$_{TESL}^{\geq Suc\ n}$ = ⟦⟦ (K_1 *sporadic* τ *on* K_2) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$›
by *auto*
then show *?thesis*
by *auto*
qed
qed

lemma *HeronConf-interp-stepwise-tagrel-cases*:
shows ‹⟦ Γ, $n \vdash ((time{-}relation\ \lfloor K_1, K_2 \rfloor \in R) \# \Psi) \triangleright \Phi$ ⟧$_{config}$
= ⟦ $((\lfloor \tau_{var}(K_1, n), \tau_{var}(K_2, n) \rfloor \in R) \# \Gamma), n \vdash \Psi \triangleright ((time{-}relation\ \lfloor K_1, K_2 \rfloor \in R) \# \Phi)$ ⟧$_{config}$›
proof −
have ‹⟦ Γ, $n \vdash (time{-}relation\ \lfloor K_1, K_2 \rfloor \in R) \# \Psi \triangleright \Phi$ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$

$\cap$ ⟦⟦ (*time−relation* ⌊K_1, K_2⌋ ∈ R) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
by *simp*
moreover have ⟨⟦ ((⌊$\tau_{var}(K_1, n), \tau_{var}(K_2, n)$⌋ ∈ R) # Γ), $n \vdash \Psi \triangleright ((time{-}relation$ ⌊K_1, K_2⌋ ∈ R) # Φ) ⟧$_{config}$
= ⟦⟦ (⌊$\tau_{var}(K_1, n), \tau_{var}(K_2, n)$⌋ ∈ R) # Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (*time−relation* ⌊K_1, K_2⌋ ∈ R) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
by *simp*
ultimately show *?thesis*
proof −
have ⟨⟦ ⌊$\tau_{var}(K_1, n), \tau_{var}(K_2, n)$⌋ ∈ R ⟧$_{prim}$ ∩ ⟦ *time−relation* ⌊K_1, K_2⌋ ∈ R ⟧$_{TESL}^{\geq Suc\ n}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ = ⟦⟦ (*time−relation* ⌊K_1, K_2⌋ ∈ R) # Ψ ⟧⟧$_{TESL}^{\geq n}$⟩
using *TESL-interp-stepwise-tagrel-coind-unfold TESL-interpretation-stepwise-cons-morph* **by** *blast*
then show *?thesis*
by *auto*
qed
qed

lemma *HeronConf-interp-stepwise-implies-cases*:
shows ⟨⟦ Γ, $n \vdash ((K_1\ implies\ K_2)$ # Ψ) ▷ Φ ⟧$_{config}$
= ⟦ ((K$_1$ ¬↑ n) # Γ), $n \vdash \Psi \triangleright ((K_1\ implies\ K_2)$ # Φ) ⟧$_{config}$
∪ ⟦ ((K$_1$ ↑ n) # (K$_2$ ↑ n) # Γ), $n \vdash \Psi \triangleright ((K_1\ implies\ K_2)$ # Φ) ⟧$_{config}$⟩
proof −
have ⟨⟦ Γ, $n \vdash (K_1\ implies\ K_2)$ # Ψ ▷ Φ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ (K$_1$ *implies* K$_2$) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
by *simp*
moreover have ⟨⟦ ((K$_1$ ¬↑ n) # Γ), $n \vdash \Psi \triangleright ((K_1\ implies\ K_2)$ # Φ) ⟧$_{config}$ = ⟦⟦ (K$_1$ ¬↑ n) # Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K$_1$ *implies* K$_2$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
by *simp*
moreover have ⟨⟦ ((K$_1$ ↑ n) # (K$_2$ ↑ n) # Γ), $n \vdash \Psi \triangleright ((K_1\ implies\ K_2)$ # Φ) ⟧$_{config}$ = ⟦⟦ ((K$_1$ ↑ n) # (K$_2$ ↑ n) # Γ) ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K$_1$ *implies* K$_2$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
by *simp*
ultimately show *?thesis*
proof −
have *f1*: ⟨(⟦ K$_1$ ¬↑ n ⟧$_{prim}$ ∪ ⟦ K$_1$ ↑ n ⟧$_{prim}$ ∩ ⟦ K$_2$ ↑ n ⟧$_{prim}$) ∩ ⟦ K$_1$ *implies* K$_2$ ⟧$_{TESL}^{\geq Suc\ n}$ ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$) = ⟦⟦ (K$_1$ *implies* K$_2$) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩
using *TESL-interp-stepwise-implies-coind-unfold TESL-interpretation-stepwise-cons-morph* **by** *blast*
have ⟨⟦ K$_1$ ¬↑ n ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$ ∪ ⟦ K$_1$ ↑ n ⟧$_{prim}$ ∩ ⟦⟦ (K$_2$ ↑ n) # Γ ⟧⟧$_{prim}$ = (⟦ K$_1$ ¬↑ n ⟧$_{prim}$ ∪ ⟦ K$_1$ ↑ n ⟧$_{prim}$ ∩ ⟦ K$_2$ ↑ n ⟧$_{prim}$) ∩ ⟦⟦ Γ ⟧⟧$_{prim}$⟩
by *force*
then have ⟨⟦ Γ, $n \vdash ((K_1\ implies\ K_2)$ # Ψ) ▷ Φ ⟧$_{config}$ = (⟦ K$_1$ ¬↑ n ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$ ∪ ⟦ K$_1$ ↑ n ⟧$_{prim}$ ∩ ⟦⟦ (K$_2$ ↑ n) # Γ ⟧⟧$_{prim}$) ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K$_1$ *implies* K$_2$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$)⟩

using *f1* **by** (*simp add*: *inf-left-commute inf-sup-aci*(*2*))
then show *?thesis*
by (*simp add*: *Int-Un-distrib2 inf-sup-aci*(*2*))
qed
qed

lemma *HeronConf-interp-stepwise-implies-not-cases*:
shows ‹⟦ Γ, $n \vdash ((K_1$ *implies not* $K_2)$ # Ψ) ▷ Φ ⟧$_{config}$
= ⟦ $((K_1 \neg\Uparrow n)$ # Γ), $n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2)$ # Φ) ⟧$_{config}$
∪ ⟦ $((K_1 \Uparrow n)$ # $(K_2 \neg\Uparrow n)$ # Γ), $n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2)$ # Φ) ⟧$_{config}$›
proof −
have ‹⟦ Γ, $n \vdash (K_1$ *implies not* $K_2)$ # Ψ ▷ Φ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ (K_1 *implies not* K_2) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦ $((K_1 \neg\Uparrow n)$ # Γ), $n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2)$ # Φ) ⟧$_{config}$
= ⟦⟦ $(K_1 \neg\Uparrow n)$ # Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K_1 *implies not* K_2) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
moreover have ‹⟦ $((K_1 \Uparrow n)$ # $(K_2 \neg\Uparrow n)$ # Γ), $n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2)$ # Φ) ⟧$_{config}$
= ⟦⟦ $((K_1 \Uparrow n)$ # $(K_2 \neg\Uparrow n)$ # Γ) ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K_1 *implies not* K_2) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
by *simp*
ultimately show *?thesis*
proof −
have *f1*: ‹(⟦ $K_1 \neg\Uparrow n$ ⟧$_{prim}$ ∪ ⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦ $K_2 \neg\Uparrow n$ ⟧$_{prim}$) ∩ ⟦ K_1 *implies not* K_2 ⟧$_{TESL}^{\geq Suc\ n}$ ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$)
= ⟦⟦ (K_1 *implies not* K_2) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$›
using *TESL-interp-stepwise-implies-not-coind-unfold TESL-interpretation-stepwise-cons-morph*
by *blast*
have ‹⟦ $K_1 \neg\Uparrow n$ ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$ ∪ ⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦⟦ $(K_2 \neg\Uparrow n)$ # Γ ⟧⟧$_{prim}$ = (⟦ $K_1 \neg\Uparrow n$ ⟧$_{prim}$ ∪ ⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦ $K_2 \neg\Uparrow n$ ⟧$_{prim}$) ∩ ⟦⟦ Γ ⟧⟧$_{prim}$›
by *force*
then have ‹⟦ Γ, $n \vdash ((K_1$ *implies not* $K_2)$ # Ψ) ▷ Φ ⟧$_{config}$
= (⟦ $K_1 \neg\Uparrow n$ ⟧$_{prim}$ ∩ ⟦⟦ Γ ⟧⟧$_{prim}$ ∪ ⟦ $K_1 \Uparrow n$ ⟧$_{prim}$ ∩ ⟦⟦ $(K_2 \neg\Uparrow n)$ # Γ ⟧⟧$_{prim}$) ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (K_1 *implies not* K_2) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$)›
using *f1* **by** (*simp add*: *inf-left-commute inf-sup-aci*(*2*))
then show *?thesis*
by (*simp add*: *Int-Un-distrib2 inf-sup-aci*(*2*))
qed
qed

lemma *HeronConf-interp-stepwise-timedelayed-cases*:
shows ‹⟦ Γ, $n \vdash ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3)$ # Ψ) ▷ Φ ⟧$_{config}$
= ⟦ $((K_1 \neg\Uparrow n)$ # Γ), $n \vdash \Psi \triangleright ((K_1$ *time−delayed by* $\delta\tau$ *on* K_2 *implies* $K_3)$ # Φ) ⟧$_{config}$

$\cup$ ⟦ ((*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$⟩

proof −

have *1*:⟨⟦ Γ, *n* ⊢ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ▷ Φ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩

by *simp*

moreover have ⟨⟦ ((*K*$_1$ ¬⇑ *n*) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$ = ⟦⟦ (*K*$_1$ ¬⇑ *n*) # Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩

by *simp*

moreover have ⟨⟦ ((*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$ = ⟦⟦ (*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$⟩

by *simp*

ultimately show *?thesis*

proof −

have ⟨⟦ Γ, *n* ⊢ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ▷ Φ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ (⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$)⟩

using *1* **by** *blast*

then have ⟨⟦ Γ, *n* ⊢ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ▷ Φ ⟧$_{config}$ = (⟦ *K*$_1$ ¬⇑ *n* ⟧$_{prim}$ ∪ ⟦ *K*$_1$ ⇑ *n* ⟧$_{prim}$ ∩ ⟦ *K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$ ⟧$_{prim}$) ∩ (⟦⟦ Γ ⟧⟧$_{prim}$ ∩ (⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ ⟧⟧$_{TESL}^{\geq Suc\ n}$))⟩

using *TESL-interpretation-stepwise-cons-morph TESL-interp-stepwise-timedelayed-coind-unfold*

proof −

have ⟨⟦⟦ (*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ ⟧⟧$_{TESL}^{\geq n}$ = (⟦ *K*$_1$ ¬⇑ *n* ⟧$_{prim}$ ∪ ⟦ *K*$_1$ ⇑ *n* ⟧$_{prim}$ ∩ ⟦ *K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$ ⟧$_{prim}$) ∩ ⟦ *K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$ ⟧$_{TESL}^{\geq Suc\ n}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$⟩

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*:

shows ⟨⟦ Γ, *n* ⊢ ((*K*$_1$ *weakly precedes K*$_2$) # Ψ) ▷ Φ ⟧$_{config}$ = ⟦ ((⌈#$^{\leq}$ *K*$_2$ *n*, #$^{\leq}$ *K*$_1$ *n*⌉ ∈ (λ(*x*,*y*). *x*≤*y*)) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *weakly precedes K*$_2$) # Φ) ⟧$_{config}$⟩

proof −

have ⟨⟦ Γ, *n* ⊢ (*K*$_1$ *weakly precedes K*$_2$) # Ψ ▷ Φ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦

$(K_1$ *weakly precedes* $K_2)$ # Ψ $]\!]_{TESL}{}^{\geq\, n} \cap [\![$ Φ $]\!]_{TESL}{}^{\geq\, Suc\ n}\rangle$
by *simp*
moreover have ‹[$\!$[(($\lceil \#^{\leq}\ K_2\ n, \#^{\leq}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y))$ # Γ), $n \vdash \Psi \triangleright$ $((K_1$ *weakly precedes* $K_2)$ # $\Phi)$ $]\!]_{config}$
$= [\![(\lceil \#^{\leq}\ K_2\ n, \#^{\leq}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y))$ # Γ $]\!]_{prim} \cap [\![$ Ψ $]\!]_{TESL}{}^{\geq\, n} \cap [\![(K_1$ *weakly precedes* $K_2)$ # Φ $]\!]_{TESL}{}^{\geq\, Suc\ n}\rangle$
by *simp*
ultimately show *?thesis*
proof −
have ‹[$\!$[$\lceil \#^{\leq}\ K_2\ n, \#^{\leq}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y)$ $]\!]_{prim} \cap [\![$ K_1 *weakly precedes* K_2 $]\!]_{TESL}{}^{\geq\, Suc\ n} \cap [\![$ Ψ $]\!]_{TESL}{}^{\geq\, n} = [\![(K_1$ *weakly precedes* $K_2)$ # Ψ $]\!]_{TESL}{}^{\geq\, n}\rangle$
using *TESL-interp-stepwise-weakly-precedes-coind-unfold TESL-interpretation-stepwise-cons-morph*
by *blast*
then show *?thesis*
by *auto*
qed
qed

lemma *HeronConf-interp-stepwise-weakly-precedes-cases′*:
shows ‹[$\!$[Γ, $n \vdash ((K_1$ *weakly precedes* $K_2)$ # $\Psi) \triangleright \Phi$ $]\!]_{config}$
$=$ [$\!$[$(((\#^{\leq}\ K_2\ n) \preceq (\#^{\leq}\ K_1\ n))$ # $\Gamma)$, $n \vdash \Psi \triangleright ((K_1$ *weakly precedes* $K_2)$ # $\Phi)$ $]\!]_{config}\rangle$
oops

lemma *HeronConf-interp-stepwise-strictly-precedes-cases*:
shows ‹[$\!$[Γ, $n \vdash (K_1$ *strictly precedes* $K_2)$ # $\Psi \triangleright \Phi$ $]\!]_{config}$
$=$ [$\!$[$((\lceil \#^{\leq}\ K_2\ n, \#^{<}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y))$ # $\Gamma)$, $n \vdash \Psi \triangleright ((K_1$ *strictly precedes* $K_2)$ # $\Phi)$ $]\!]_{config}\rangle$
proof −
have ‹[$\!$[Γ, $n \vdash (K_1$ *strictly precedes* $K_2)$ # $\Psi \triangleright \Phi$ $]\!]_{config} = [\![$ Γ $]\!]_{prim} \cap [\![$ $(K_1$ *strictly precedes* $K_2)$ # Ψ $]\!]_{TESL}{}^{\geq\, n} \cap [\![$ Φ $]\!]_{TESL}{}^{\geq\, Suc\ n}\rangle$
by *simp*
moreover have ‹[$\!$[$((\lceil \#^{\leq}\ K_2\ n, \#^{<}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y))$ # $\Gamma)$, $n \vdash \Psi \triangleright$ $((K_1$ *strictly precedes* $K_2)$ # $\Phi)$ $]\!]_{config}$
$= [\![(\lceil \#^{\leq}\ K_2\ n, \#^{<}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y))$ # Γ $]\!]_{prim} \cap [\![$ Ψ $]\!]_{TESL}{}^{\geq\, n} \cap [\![(K_1$ *strictly precedes* $K_2)$ # Φ $]\!]_{TESL}{}^{\geq\, Suc\ n}\rangle$
by *simp*
ultimately show *?thesis*
proof −
have ‹[$\!$[$\lceil \#^{\leq}\ K_2\ n, \#^{<}\ K_1\ n\rceil \in (\lambda(x,y).\ x{\leq}y)$ $]\!]_{prim} \cap [\![$ K_1 *strictly precedes* K_2 $]\!]_{TESL}{}^{\geq\, Suc\ n} \cap [\![$ Ψ $]\!]_{TESL}{}^{\geq\, n} = [\![(K_1$ *strictly precedes* $K_2)$ # Ψ $]\!]_{TESL}{}^{\geq\, n}\rangle$
using *TESL-interp-stepwise-strictly-precedes-coind-unfold TESL-interpretation-stepwise-cons-morph*
by *blast*
then show *?thesis*
by *auto*
qed
qed

lemma *HeronConf-interp-stepwise-kills-cases*:
shows ‹⟦ Γ, $n \vdash ((K_1\ \mathit{kills}\ K_2) \#\ \Psi) \triangleright \Phi$ ⟧$_{config}$
$= ⟦$ $((K_1 \neg\Uparrow n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1\ \mathit{kills}\ K_2) \#\ \Phi)$ ⟧$_{config}$
$\cup ⟦$ $((K_1 \Uparrow n) \#\ (K_2 \neg\Uparrow \geq n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1\ \mathit{kills}\ K_2) \#\ \Phi)$ ⟧$_{config}$›
proof −
have ‹⟦ Γ, $n \vdash ((K_1\ \mathit{kills}\ K_2) \#\ \Psi) \triangleright \Phi$ ⟧$_{config}$ = ⟦⟦ Γ ⟧⟧$_{prim}$ ∩ ⟦⟦ $(K_1\ \mathit{kills}\ K_2) \#\ \Psi$ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ Φ ⟧⟧$_{TESL}^{\geq \mathit{Suc}\ n}$›
by *simp*
moreover have ‹⟦ $((K_1 \neg\Uparrow n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1\ \mathit{kills}\ K_2) \#\ \Phi)$ ⟧$_{config}$
= ⟦⟦ $(K_1 \neg\Uparrow n) \#\ \Gamma$ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ $(K_1\ \mathit{kills}\ K_2) \#\ \Phi$ ⟧⟧$_{TESL}^{\geq \mathit{Suc}\ n}$›
by *simp*
moreover have ‹⟦ $((K_1 \Uparrow n) \#\ (K_2 \neg\Uparrow \geq n) \#\ \Gamma)$, $n \vdash \Psi \triangleright ((K_1\ \mathit{kills}\ K_2) \#\ \Phi)$ ⟧$_{config}$
= ⟦⟦ $(K_1 \Uparrow n) \#\ (K_2 \neg\Uparrow \geq n) \#\ \Gamma$ ⟧⟧$_{prim}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$ ∩ ⟦⟦ $(K_1\ \mathit{kills}\ K_2) \#\ \Phi$ ⟧⟧$_{TESL}^{\geq \mathit{Suc}\ n}$›
by *simp*
ultimately show *?thesis*
proof −
have ‹⟦⟦ $(K_1\ \mathit{kills}\ K_2) \#\ \Psi$ ⟧⟧$_{TESL}^{\geq n}$ = (⟦ $(K_1 \neg\Uparrow n)$ ⟧$_{prim}$ ∪ ⟦ $(K_1 \Uparrow n)$ ⟧$_{prim}$ ∩ ⟦ $(K_2 \neg\Uparrow \geq n)$ ⟧$_{prim}$) ∩ ⟦ $(K_1\ \mathit{kills}\ K_2)$ ⟧$_{TESL}^{\geq \mathit{Suc}\ n}$ ∩ ⟦⟦ Ψ ⟧⟧$_{TESL}^{\geq n}$›
using *TESL-interp-stepwise-kills-coind-unfold TESL-interpretation-stepwise-cons-morph*
by *blast*
then show *?thesis*
by *auto*
qed
qed

end

Chapter 6

Main Theorems

theory *Hygge-Theory*
imports
Corecursive-Prop

begin

6.1 Initial configuration

Solving a specification Ψ means to start operational semantics at initial configuration $[]$, $0 \vdash \Psi \triangleright []$

theorem *solve-start*:
shows ‹$[\![\![\Psi]\!]\!]_{TESL} = [\![[], 0 \vdash \Psi \triangleright []]\!]_{config}$›
proof −
have ‹$[\![\![\Psi]\!]\!]_{TESL} = [\![\![\Psi]\!]\!]_{TESL}^{\geq 0}$›
by (*simp add*: *TESL-interpretation-stepwise-zero'*)
moreover have ‹$[\![[], 0 \vdash \Psi \triangleright []]\!]_{config} = [\![\![[]]\!]\!]_{prim} \cap [\![\![\Psi]\!]\!]_{TESL}^{\geq 0} \cap [\![\![[]]\!]\!]_{TESL}^{\geq Suc\ 0}$›
by *simp*
ultimately show *?thesis* **by** *auto*
qed

6.2 Soundness

lemma *sound-reduction*:
assumes ‹$(\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$›
shows ‹$[\![\![\Gamma_1]\!]\!]_{prim} \cap [\![\![\Psi_1]\!]\!]_{TESL}^{\geq n_1} \cap [\![\![\Phi_1]\!]\!]_{TESL}^{\geq Suc\ n_1}$
$\supseteq [\![\![\Gamma_2]\!]\!]_{prim} \cap [\![\![\Psi_2]\!]\!]_{TESL}^{\geq n_2} \cap [\![\![\Phi_2]\!]\!]_{TESL}^{\geq Suc\ n_2}$› (**is** *?P*)
proof −
from *assms* **consider**
(a) ‹$(\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow_i (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$›
| (b) ‹$(\Gamma_1, n_1 \vdash \Psi_1 \triangleright \Phi_1) \hookrightarrow_e (\Gamma_2, n_2 \vdash \Psi_2 \triangleright \Phi_2)$›
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 $\Gamma\ n\ K_1\ \tau\ K_2\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash (K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Psi \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (\Gamma,\ n \vdash \Psi \triangleright ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Phi))$›
thus *?P*
using *HeronConf-interp-stepwise-sporadicon-cases HeronConf-interpretation.simps*
by *blast*
next
fix $\Gamma\ n\ K_1\ \tau\ K_2\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash (K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Psi \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \Uparrow n)\ \#\ (K_2 \Downarrow n\ @\ \tau)\ \#\ \Gamma),\ n \vdash \Psi \triangleright \Phi)$›
thus *?P*
using *HeronConf-interp-stepwise-sporadicon-cases HeronConf-interpretation.simps*
by *blast*
next
fix $\Gamma\ n\ K_1\ K_2\ R\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash (\mathit{time{-}relation}\ \lfloor K_1,\ K_2 \rfloor \in R)\ \#\ \Psi \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = ((((\lfloor \tau_{var}\ (K_1,\ n),\ \tau_{var}\ (K_2,\ n) \rfloor \in R)\ \#\ \Gamma),\ n \vdash \Psi \triangleright ((\mathit{time{-}relation}\ \lfloor K_1,\ K_2 \rfloor \in R)\ \#\ \Phi))$›
thus *?P*
using *HeronConf-interp-stepwise-tagrel-cases HeronConf-interpretation.simps*
by *blast*
next
fix $\Gamma\ n\ K_1\ K_2\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash (K_1\ \mathit{implies}\ K_2)\ \#\ \Psi \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \neg\Uparrow n)\ \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{implies}\ K_2)\ \#\ \Phi))$›
thus *?P*
using *HeronConf-interp-stepwise-implies-cases HeronConf-interpretation.simps*
by *blast*
next
fix $\Gamma\ n\ K_1\ K_2\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{implies}\ K_2)\ \#\ \Psi) \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \Uparrow n)\ \#\ (K_2 \Uparrow n)\ \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{implies}\ K_2)\ \#\ \Phi))$›
thus *?P*
using *HeronConf-interp-stepwise-implies-cases HeronConf-interpretation.simps*
by *blast*
next
fix $\Gamma\ n\ K_1\ K_2\ \Psi\ \Phi$
assume ‹$(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{implies\ not}\ K_2)\ \#\ \Psi) \triangleright \Phi)$›
and ‹$(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \neg\Uparrow n)\ \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{implies\ not}\ K_2)$

$\# \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-implies-not-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 K_2 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{implies\ not}\ K_2) \#\ \Psi) \triangleright \Phi)\rangle$
and $\langle(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \Uparrow n) \#\ (K_2 \neg\Uparrow n) \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1$ *implies not* $K_2) \#\ \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-implies-not-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 $\delta\tau$ K_2 K_3 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{time{-}delayed\ by}\ \delta\tau\ \mathit{on}\ K_2$ *implies* $K_3) \#\ \Psi) \triangleright \Phi)\rangle$
and $\langle(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \neg\Uparrow n) \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{time{-}delayed}$ *by* $\delta\tau$ *on* K_2 *implies* $K_3) \#\ \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-timedelayed-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 $\delta\tau$ K_2 K_3 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{time{-}delayed\ by}\ \delta\tau\ \mathit{on}\ K_2$ *implies* $K_3) \#\ \Psi) \triangleright \Phi)\rangle$
and $\langle(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = (((K_1 \Uparrow n) \#\ (K_2\ @\ n \oplus \delta\tau \Rightarrow K_3) \#\ \Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{time{-}delayed\ by}\ \delta\tau\ \mathit{on}\ K_2\ \mathit{implies}\ K_3) \#\ \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-timedelayed-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 K_2 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{weakly\ precedes}\ K_2) \#\ \Psi) \triangleright \Phi)\rangle$
and $\langle(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = ((\lceil \#^{\leq}\ K_2\ n,\ \#^{\leq}\ K_1\ n \rceil \in (\lambda(x,\ y).\ x \leq y)) \#$ $\Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{weakly\ precedes}\ K_2) \#\ \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-weakly-precedes-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 K_2 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{strictly\ precedes}\ K_2) \#\ \Psi) \triangleright \Phi)\rangle$
and $\langle(\Gamma_2,\ n_2 \vdash \Psi_2 \triangleright \Phi_2) = ((\lceil \#^{\leq}\ K_2\ n,\ \#^{<}\ K_1\ n \rceil \in (\lambda(x,\ y).\ x \leq y)) \#$ $\Gamma),\ n \vdash \Psi \triangleright ((K_1\ \mathit{strictly\ precedes}\ K_2) \#\ \Phi))\rangle$
thus *?P*
using *HeronConf-interp-stepwise-strictly-precedes-cases HeronConf-interpretation.simps*
by *blast*
next
fix Γ n K_1 K_2 Ψ Φ
assume $\langle(\Gamma_1,\ n_1 \vdash \Psi_1 \triangleright \Phi_1) = (\Gamma,\ n \vdash ((K_1\ \mathit{kills}\ K_2) \#\ \Psi) \triangleright \Phi)\rangle$

```
    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 −
  from assms consider
    (a) ‹S1 ↪i S2›
  | (b) ‹S1 ↪e S2›
    using step-elim by blast
  thus ?thesis
  proof (cases)
    case a thus ?thesis by (rule operational-semantics-intro.cases, simp)
  next
    case b thus ?thesis using assms
    by (metis (full-types) HeronConf-interpretation.cases HeronConf-interpretation.simps
sound-reduction)
  qed
qed

lemma sound-reduction-generalized:
  assumes ‹S1 ↪^k S2›
    shows ‹⟦ S1 ⟧config ⊇ ⟦ S2 ⟧config›
proof −
  from assms show ?thesis
  proof (induct 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 initial configuration, any reduction step number k providing a configuration $\mathcal{S}$ will denote runs from initial specification Ψ.

```
theorem soundness:
  assumes ‹([], 0 ⊢ Ψ ▷ []) ↪k S›
    shows ‹⟦⟦ Ψ ⟧⟧TESL ⊇ ⟦ S ⟧config›
  using assms sound-reduction-generalized solve-start by blast
```

6.3 Completeness

```
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)
      then show ?case
      proof (cases ψ)
        case (SporadicOn K1 τ K2)
        then show ?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)
        then show ?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)
```

```
      then show ?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)
      then show ?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)
      then show ?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)
      then show ?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)
      then show ?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

lemma branch-existence:
  assumes ⟨ϱ ∈ ⟦ S1 ⟧config⟩
```

shows ‹$\exists \mathcal{S}_2.\ (\mathcal{S}_1 \hookrightarrow \mathcal{S}_2) \wedge (\varrho \in [\![\ \mathcal{S}_2\ ]\!]_{config})$›
by (*metis* (*mono-tags*, *lifting*) *UN-iff assms complete-direct-successors' mem-Collect-eq set-rev-mp*)

lemma *branch-existence'*:
assumes ‹$\varrho \in [\![\ \mathcal{S}_1\ ]\!]_{config}$›
shows ‹$\exists \mathcal{S}_2.\ (\mathcal{S}_1 \hookrightarrow^k \mathcal{S}_2) \wedge (\varrho \in [\![\ \mathcal{S}_2\ ]\!]_{config})$›
proof (*induct k*)
case *0*
then show *?case* **by** (*simp add*: *assms*)
next
case (*Suc k*)
then show *?case*
using *branch-existence relpowp-Suc-I*[*of* ‹*k*› ‹*operational-semantics-step*›] **by** *blast*
qed

Any run from initial specification Ψ has a corresponding configuration $\mathcal{S}$ at any reduction step number k starting from initial configuration.

theorem *completeness*:
assumes ‹$\varrho \in [\![\![\ \Psi\ ]\!]\!]_{TESL}$›
shows ‹$\exists \mathcal{S}.\ (([], 0 \vdash \Psi \triangleright []) \hookrightarrow^k \mathcal{S}) \wedge \varrho \in [\![\ \mathcal{S}\ ]\!]_{config}$›
using *assms branch-existence' solve-start* **by** *blast*

6.4 Progress

lemma *instant-index-increase*:
assumes ‹$\varrho \in [\![\ \Gamma,\ n \vdash \Psi \triangleright \Phi\ ]\!]_{config}$›
shows ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((\Gamma,\ n \vdash \Psi \triangleright \Phi) \hookrightarrow^k (\Gamma_k,\ Suc\ n \vdash \Psi_k \triangleright \Phi_k)) \wedge \varrho \in [\![\ \Gamma_k,\ Suc\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config}$›
proof (*insert assms*, *induct* Ψ *arbitrary*: Γ Φ)
case (*Nil* Γ Φ)
then show *?case*
proof −
have ‹$(\Gamma,\ n \vdash [] \triangleright \Phi) \hookrightarrow^1 (\Gamma,\ Suc\ n \vdash \Phi \triangleright [])$›
using *instant-i intro-part*
by *fastforce*
moreover have ‹$[\![\ \Gamma,\ n \vdash [] \triangleright \Phi\ ]\!]_{config} = [\![\ \Gamma,\ Suc\ n \vdash \Phi \triangleright []\ ]\!]_{config}$›
by *auto*
moreover have ‹$\varrho \in [\![\ \Gamma,\ Suc\ n \vdash \Phi \triangleright []\ ]\!]_{config}$›
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* $K_1\ \tau\ K_2$)
have *branches*: ‹$[\![\ \Gamma,\ n \vdash ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Psi) \triangleright \Phi\ ]\!]_{config}$
$= [\![\ \Gamma,\ n \vdash \Psi \triangleright ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Phi)\ ]\!]_{config}$
$\cup\ [\![\ ((K_1 \Uparrow n)\ \#\ (K_2 \Downarrow n\ @\ \tau)\ \#\ \Gamma),\ n \vdash \Psi \triangleright \Phi\ ]\!]_{config}$›
using *HeronConf-interp-stepwise-sporadicon-cases* **by** *simp*
have *br1*: ‹$\varrho \in [\![\ \Gamma,\ n \vdash \Psi \triangleright ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Phi)\ ]\!]_{config}$
$\Longrightarrow \exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$
$((\Gamma,\ n \vdash ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Psi) \triangleright \Phi) \hookrightarrow^k (\Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k)) \wedge$
$\varrho \in [\![\ \Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config}$›
proof −
assume *h1*: ‹$\varrho \in [\![\ \Gamma,\ n \vdash \Psi \triangleright ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Phi)\ ]\!]_{config}$›
then have ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((\Gamma,\ n \vdash \Psi \triangleright ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Phi))$
$\hookrightarrow^k (\Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k)) \wedge (\varrho \in [\![\ \Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config})$›
using *h1 SporadicOn.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 sporadic-on-e1*)
qed
moreover have *br2*: ‹$\varrho \in [\![\ ((K_1 \Uparrow n)\ \#\ (K_2 \Downarrow n\ @\ \tau)\ \#\ \Gamma),\ n \vdash \Psi \triangleright \Phi$
$]\!]_{config} \Longrightarrow \exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$
$((\Gamma,\ n \vdash ((K_1\ \mathit{sporadic}\ \tau\ \mathit{on}\ K_2)\ \#\ \Psi) \triangleright \Phi) \hookrightarrow^k (\Gamma_k,\ \mathit{Suc}\ n \vdash$
$\Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\ \Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config}$›
proof −
assume *h2*: ‹$\varrho \in [\![\ ((K_1 \Uparrow n)\ \#\ (K_2 \Downarrow n\ @\ \tau)\ \#\ \Gamma),\ n \vdash \Psi \triangleright \Phi\ ]\!]_{config}$›
then have ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((((K_1 \Uparrow n)\ \#\ (K_2 \Downarrow n\ @\ \tau)\ \#\ \Gamma),\ n \vdash \Psi \triangleright$
$\Phi) \hookrightarrow^k (\Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\ \Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config}$›
using *h2 SporadicOn.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 sporadic-on-e2*)
qed
ultimately show *?case*
by (*metis SporadicOn.prems*(*2*) *UnE branches*)
next
case (*TagRelation* $K_1\ K_2\ R$)
have *branches*: ‹$[\![\ \Gamma,\ n \vdash ((\mathit{time{-}relation}\ \lfloor K_1,\ K_2 \rfloor \in R)\ \#\ \Psi) \triangleright \Phi\ ]\!]_{config}$
$= [\![\ (((\lfloor \tau_{var}(K_1,\ n),\ \tau_{var}(K_2,\ n) \rfloor \in R)\ \#\ \Gamma),\ n \vdash \Psi \triangleright ((\mathit{time{-}relation}$
$\lfloor K_1,\ K_2 \rfloor \in R)\ \#\ \Phi)\ ]\!]_{config}$›
using *HeronConf-interp-stepwise-tagrel-cases* **by** *simp*
then show *?case*
proof −
have ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((((\lfloor \tau_{var}(K_1,\ n),\ \tau_{var}(K_2,\ n) \rfloor \in R)\ \#\ \Gamma),\ n \vdash \Psi \triangleright$
$((\mathit{time{-}relation}\ \lfloor K_1,\ K_2 \rfloor \in R)\ \#\ \Phi))$
$\hookrightarrow^k (\Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k)) \wedge \varrho \in [\![\ \Gamma_k,\ \mathit{Suc}\ n \vdash \Psi_k \triangleright \Phi_k\ ]\!]_{config}$›
using *TagRelation.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 tagrel-e*)
qed
next

case (*Implies* K_1 K_2)
have *branches*: ‹$[\![\Gamma, n \vdash ((K_1 \textit{ implies } K_2) \# \Psi) \triangleright \Phi]\!]_{config}$
$= [\![((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$
$\cup [\![((K_1 \Uparrow n) \# (K_2 \Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$›
using *HeronConf-interp-stepwise-implies-cases* **by** *simp*
have *br1*: ‹$\varrho \in [\![((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$
$\Longrightarrow \exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((\Gamma, n \vdash ((K_1 \textit{ implies } K_2) \# \Psi) \triangleright \Phi) \hookrightarrow^k (\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
proof −
assume *h1*: ‹$\varrho \in [\![((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$›
then have ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$
$((((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)) \hookrightarrow^k (\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
using *h1 Implies.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 implies-e1*)
qed
moreover have *br2*: ‹$\varrho \in [\![((K_1 \Uparrow n) \# (K_2 \Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$
$\Longrightarrow \exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$
$((\Gamma, n \vdash ((K_1 \textit{ implies } K_2) \# \Psi) \triangleright \Phi) \hookrightarrow^k (\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k)) \wedge$
$\varrho \in [\![\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
proof −
assume *h2*: ‹$\varrho \in [\![((K_1 \Uparrow n) \# (K_2 \Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)]\!]_{config}$›
then have ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$ (
$(((K_1 \Uparrow n) \# (K_2 \Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ implies } K_2) \# \Phi)) \hookrightarrow^k (\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k)$
$) \wedge \varrho \in [\![\Gamma_k, \textit{Suc } n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
using *h2 Implies.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 implies-e2*)
qed
ultimately show *?case*
using *Implies.prems*(*2*) **by** *fastforce*
next
case (*ImpliesNot* K_1 K_2)
then show *?case*
by (*metis* (*no-types*, *lifting*) *HeronConf-interp-stepwise-implies-not-cases Un-iff elims-part implies-not-e1 implies-not-e2 relpowp-Suc-I2*)
next
case (*TimeDelayedBy* K_1 $\delta\tau$ K_2 K_3)
have *branches*: ‹$[\![\Gamma, n \vdash ((K_1 \textit{ time}-\textit{delayed by } \delta\tau \textit{ on } K_2 \textit{ implies } K_3) \# \Psi) \triangleright \Phi]\!]_{config}$
$= [\![((K_1 \neg\Uparrow n) \# \Gamma), n \vdash \Psi \triangleright ((K_1 \textit{ time}-\textit{delayed by } \delta\tau \textit{ on } K_2 \textit{ implies } K_3) \# \Phi)]\!]_{config}$

$\cup$ ⟦ ((*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$⟩
using *HeronConf-interp-stepwise-timedelayed-cases* **by** *simp*
have *br1*: ⟨*ϱ* ∈ ⟦ ((*K*$_1$ ¬⇑ *n*) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$
⟹ ∃ Γ$_k$ Ψ$_k$ Φ$_k$ *k*.
((Γ, *n* ⊢ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Ψ) ▷ Φ) ↪^{k} (Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$))
∧ *ϱ* ∈ ⟦ Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$ ⟧$_{config}$⟩
proof −
assume *h1*: ⟨*ϱ* ∈ ⟦ ((*K*$_1$ ¬⇑ *n*) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$⟩
then have ⟨∃ Γ$_k$ Ψ$_k$ Φ$_k$ *k*.
((((*K*$_1$ ¬⇑ *n*) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ)) ↪^{k} (Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$))
∧ *ϱ* ∈ ⟦ Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$ ⟧$_{config}$⟩
using *h1 TimeDelayedBy.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 timedelayed-e1*)
qed
moreover have *br2*: ⟨*ϱ* ∈ ⟦ ((*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$
⟹ ∃ Γ$_k$ Ψ$_k$ Φ$_k$ *k*.
((Γ, *n* ⊢ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # 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* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ) ⟧$_{config}$⟩
then have ⟨∃ Γ$_k$ Ψ$_k$ Φ$_k$ *k*. ((((*K*$_1$ ⇑ *n*) # (*K*$_2$ @ *n* ⊕ *δτ* ⇒ *K*$_3$) # Γ), *n* ⊢ Ψ ▷ ((*K*$_1$ *time−delayed by δτ on K*$_2$ *implies K*$_3$) # Φ)) ↪^{k} (Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$)) ∧ *ϱ* ∈ ⟦ Γ$_k$, *Suc n* ⊢ Ψ$_k$ ▷ Φ$_k$ ⟧$_{config}$⟩
using *h2 TimeDelayedBy.prems* **by** *simp*
then show *?thesis*
by (*meson elims-part relpowp-Suc-I2 timedelayed-e2 sporadic-on-e1*)
qed
ultimately show *?case*
using *TimeDelayedBy.prems*(*2*) *HeronConf-interp-stepwise-timedelayed-cases* **by** *blast*
next
case (*WeaklyPrecedes K*$_1$ *K*$_2$)
then show *?case*
by (*metis* (*no-types*, *lifting*) *HeronConf-interp-stepwise-weakly-precedes-cases elims-part*
weakly-precedes-e relpowp-Suc-I2)
next
case (*StrictlyPrecedes K*$_1$ *K*$_2$)
then show *?case*
by (*metis* (*no-types*, *lifting*) *HeronConf-interp-stepwise-strictly-precedes-cases*

elims-part
strictly-precedes-e relpowp-Suc-I2)
next
case (*Kills* K_1 K_2)
then show *?case*
by (*metis* (*no-types*, *lifting*) *HeronConf-interp-stepwise-kills-cases UnE*
elims-part kills-e1 kills-e2 relpowp-Suc-I2)
qed
qed

lemma *instant-index-increase-generalized*:
assumes ‹$n < n_k$›
assumes ‹$\varrho \in [\![\Gamma, n \vdash \Psi \triangleright \Phi]\!]_{config}$›
shows ‹$\exists \Gamma_k\ \Psi_k\ \Phi_k\ k.\ ((\Gamma, n \vdash \Psi \triangleright \Phi) \hookrightarrow^k (\Gamma_k, n_k \vdash \Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\Gamma_k, n_k \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
proof −
obtain δk **where** *diff*: ‹$n_k = \delta k + Suc\ n$›
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*
then show *?case*
using *instant-index-increase assms*(*2*) **by** *simp*
next
case (*Suc* δk)
have *f0*: ‹$\varrho \in [\![\Gamma, n \vdash \Psi \triangleright \Phi]\!]_{config} \Longrightarrow \exists \Gamma_k\ \Psi_k\ \Phi_k\ k.$
$((\Gamma, n \vdash \Psi \triangleright \Phi) \hookrightarrow^k (\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k))$
$\wedge\ \varrho \in [\![\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
using *Suc.hyps* **by** *blast*
obtain $\Gamma_k\ \Psi_k\ \Phi_k\ k$
where *cont*: ‹$((\Gamma, n \vdash \Psi \triangleright \Phi) \hookrightarrow^k (\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k)) \wedge \varrho \in [\![\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k]\!]_{config}$›
using *f0 assms*(*1*) *Suc.prems* **by** *blast*
then have *fcontinue*: ‹$\exists \Gamma_k{'}\ \Psi_k{'}\ \Phi_k{'}\ k'.\ ((\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k) \hookrightarrow^{k'} (\Gamma_k{'}, Suc\ (\delta k + Suc\ n) \vdash \Psi_k{'} \triangleright \Phi_k{'}))$
$\wedge\ \varrho \in [\![\Gamma_k{'}, Suc\ (\delta k + Suc\ n) \vdash \Psi_k{'} \triangleright \Phi_k{'}]\!]_{config}$›
using *f0 cont instant-index-increase* **by** *blast*
obtain $\Gamma_k{'}\ \Psi_k{'}\ \Phi_k{'}\ k'$ **where** *cont2*: ‹$((\Gamma_k, \delta k + Suc\ n \vdash \Psi_k \triangleright \Phi_k) \hookrightarrow^{k'} (\Gamma_k{'}, Suc\ (\delta k + Suc\ n) \vdash \Psi_k{'} \triangleright \Phi_k{'}))$
$\wedge\ \varrho \in [\![\Gamma_k{'}, Suc\ (\delta k + Suc\ n) \vdash \Psi_k{'} \triangleright \Phi_k{'}]\!]_{config}$›
using *Suc.prems* **using** *fcontinue cont* **by** *blast*
have *trans*: ‹$(\Gamma, n \vdash \Psi \triangleright \Phi) \hookrightarrow^{k + k'} (\Gamma_k{'}, Suc\ (\delta k + Suc\ n) \vdash \Psi_k{'} \triangleright \Phi_k{'})$›
using *operational-semantics-trans-generalized cont cont2*
by *blast*
moreover have *suc-assoc*: ‹$Suc\ \delta k + Suc\ n = Suc\ (\delta k + Suc\ n)$›
by *arith*
ultimately show *?case*

```
    proof (subst suc-assoc)
    show ‹∃Γ_k Ψ_k Φ_k k.
          ((Γ, n ⊢ Ψ ▷ Φ) ↪^k (Γ_k, Suc (δk + Suc n) ⊢ Ψ_k ▷ Φ_k))
        ∧ ϱ ∈ ⟦ Γ_k, Suc δk + Suc n ⊢ Ψ_k ▷ Φ_k ⟧_config›
      using cont2 local.trans by auto
    qed
  qed
qed
```

Any run from initial specification Ψ has a corresponding configuration indexed at n-th instant starting from initial configuration.

```
theorem progress:
  assumes ‹ϱ ∈ ⟦⟦ Ψ ⟧⟧_TESL›
  shows   ‹∃k Γ_k Ψ_k Φ_k. ((□, 0 ⊢ Ψ ▷ □) ↪^k (Γ_k, n ⊢ Ψ_k ▷ Φ_k))
                         ∧ ϱ ∈ ⟦ Γ_k, n ⊢ Ψ_k ▷ Φ_k ⟧_config›
  using instant-index-increase-generalized
  by (metis assms neq0-conv relpowp-0-I solve-start)
```

6.5 Local termination

```
primrec measure-interpretation :: ‹'τ :: linordered-field TESL-formula ⇒ nat› (μ)
where
   ‹μ □ = (0::nat)›
 | ‹μ (φ # Φ) = (case φ of
                   - sporadic - on - ⇒ 1 + μ Φ
                 | -                 ⇒ 2 + μ Φ)›

fun measure-interpretation-config :: ‹'τ :: linordered-field config ⇒ nat› (μ_config)
where
   ‹μ_config (Γ, n ⊢ Ψ ▷ Φ) = μ Ψ›

lemma elimation-rules-strictly-decreasing:
  assumes ‹(Γ_1, n_1 ⊢ Ψ_1 ▷ Φ_1) ↪_e (Γ_2, n_2 ⊢ Ψ_2 ▷ Φ_2)›
  shows ‹μ Ψ_1 > μ Ψ_2›
  by (insert assms, erule operational-semantics-elim.cases, auto)

lemma elimation-rules-strictly-decreasing-meas:
  assumes ‹(Γ_1, n_1 ⊢ Ψ_1 ▷ Φ_1) ↪_e (Γ_2, n_2 ⊢ Ψ_2 ▷ Φ_2)›
  shows ‹(Ψ_2, Ψ_1) ∈ measure μ›
  by (insert assms, erule operational-semantics-elim.cases, auto)

lemma elimation-rules-strictly-decreasing-meas':
  assumes ‹S_1 ↪_e S_2›
  shows ‹(S_2, S_1) ∈ measure μ_config›
  using elimation-rules-strictly-decreasing-meas
  by (metis assms in-measure measure-interpretation-config.elims)
```

The relation made up of elimination rules is well-founded.

theorem *instant-computation-termination*:
shows ‹*wfP* ($\lambda(\mathcal{S}_1$:: *'a* :: *linordered-field config*) $\mathcal{S}_2$. ($\mathcal{S}_1$ $\hookrightarrow_e{}^{\leftarrow}$ $\mathcal{S}_2$))›
proof (*simp add*: *wfP-def*)
show ‹*wf* {(($\mathcal{S}_1$:: *'a* :: *linordered-field config*), $\mathcal{S}_2$). $\mathcal{S}_1$ $\hookrightarrow_e{}^{\leftarrow}$ $\mathcal{S}_2$}›
proof (*rule wf-subset*)
have ‹*measure* μ_{config} = { ($\mathcal{S}_2$, ($\mathcal{S}_1$:: *'a* :: *linordered-field config*)). μ_{config} $\mathcal{S}_2$ < μ_{config} $\mathcal{S}_1$ }›
by (*simp add*: *inv-image-def less-eq measure-def*)
then show ‹{(($\mathcal{S}_1$:: *'a* :: *linordered-field config*), $\mathcal{S}_2$). $\mathcal{S}_1$ $\hookrightarrow_e{}^{\leftarrow}$ $\mathcal{S}_2$} ⊆ (*measure* μ_{config})›
using *elimation-rules-strictly-decreasing-meas' operational-semantics-elim-inv-def*
by *blast*
show ‹*wf* (*measure measure-interpretation-config*)›
by *simp*
qed
qed

end

Chapter 7

Properties of TESL

7.1 Stuttering Invariance

theory *StutteringDefs*

imports *Denotational*

begin

7.1.1 Definition of stuttering

A dilating function inserts empty instants in a run. It is strictly increasing, the image of a *nat* is greater than it, no instant is inserted before the first one and if n is not in the image of the function, no clock ticks at instant n.

definition *dilating-fun*
where
‹*dilating-fun* (*f*::*nat* ⇒ *nat*) (*r*::'*a*::*linordered-field run*)
≡ *strict-mono f* ∧ (*f 0* = *0*) ∧ (∀ *n*. *f n* ≥ *n*
∧ ((∄ n_0. *f* n_0 = *n*) ⟶ (∀ *c*. ¬(*hamlet* ((*Rep-run r*) *n c*))))
∧ ((∄ n_0. *f* n_0 = (*Suc n*)) ⟶ (∀ *c*. *time* ((*Rep-run r*) (*Suc n*) *c*) = *time* ((*Rep-run r*) *n c*)))
›

Dilating a run. A run r is a dilation of a run *sub* by function f if:

- f is a dilating function on the hamlet of r
- time is preserved in stuttering instants
- 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 *tick-count r c n* is a 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*)}›

A *tick-count-strict r c n* is a 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*)}›

definition *contracting-fun*
where ‹*contracting-fun g* ≡ *mono g* ∧ *g 0* = *0* ∧ (∀ *n*. *g n* ≤ *n*)›

definition *contracting*
where
‹*contracting g r sub f* ≡ *contracting-fun g*
∧ (∀ *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*))›

definition ‹*dil-inverse f*::(*nat* ⇒ *nat*) ≡ (λ*n*. *Max* {*i*. *f i* ≤ *n*})›

end

7.1.2 Stuttering Lemmas

theory *StutteringLemmas*

imports *StutteringDefs*

begin

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

7.1.3 Lemmas used to prove the invariance by stuttering

A dilating function 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*

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

lemma *ticks-image*:
assumes ‹*dilating-fun f r*›
and ‹*hamlet* ((*Rep-run r*) *n c*)›
shows ‹∃ n_0. *f* n_0 = *n*›
using *dilating-fun-def assms* **by** *blast*

The image of the ticks in a interval by a dilating function is the interval bounded by the image of the bound 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** k_0 **where** *k0prop*:‹*f* k_0 = *k* ∧ *hamlet* ((*Rep-run r*) (*f* k_0) *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** k_0 **where** *k0prop*:‹*k* = *f* k_0 ∧ k_0 ∈ *?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:‹$k \in$ *image f ?SET*›
from h **obtain** k_0 **where** *k0prop*:‹$k = f\ k_0 \wedge k_0 \in$ *?SET*› **by** *blast*
hence ‹$k \in$ *?IMG*› **using** *assms* **by** (*simp add*: *dilating-fun-def strict-mono-less-eq*)
} thus ‹*image f ?SET* $\subseteq$ *?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* $\{p.\ n < p \wedge p < m \wedge$ *hamlet* ((*Rep-run r*) (*f p*) *c*)$\}$
$=$ *card* $\{p.\ f\ n < p \wedge p < f\ m \wedge$ *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* $\{p.\ n \leq p \wedge p < m \wedge$ *hamlet* ((*Rep-run r*) (*f p*) *c*)$\}$
$=$ *card* $\{p.\ f\ n \leq p \wedge p < f\ m \wedge$ *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* $\{p.\ n < p \wedge p \leq m \wedge$ *hamlet* ((*Rep-run r*) (*f p*) *c*)$\}$
$=$ *card* $\{p.\ f\ n < p \wedge p \leq f\ m \wedge$ *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

lemma *dilating-injects*:
assumes ‹*dilating f sub r*›
shows ‹*inj-on f A*›
using *assms* **by** (*simp add*: *dilating-def dilating-fun-def strict-mono-imp-inj-on*)

If there is a tick at instant n in a dilated run, n is necessarily the image of some instant in the subrun.

lemma *ticks-image-sub*:
assumes ‹*dilating f sub r*›
and ‹*hamlet* ((*Rep-run r*) *n c*)›
shows ‹∃ *n*$_0$. *f n*$_0$ = *n*›
proof −
from *assms*(*1*) **have** ‹*dilating-fun f r*› **by** (*simp add*: *dilating-def*)
from *ticks-image*[*OF this assms*(*2*)] **show** *?thesis* **.**
qed

lemma *ticks-image-sub′*:
assumes ‹*dilating f sub r*›
and ‹∃ *c*. *hamlet* ((*Rep-run r*) *n c*)›
shows ‹∃ *n*$_0$. *f n*$_0$ = *n*›
proof −
from *assms*(*1*) **have** ‹*dilating-fun f r*› **by** (*simp add*: *dilating-def*)
with *dilating-fun-def assms*(*2*) **show** *?thesis* **by** *blast*
qed

Time is preserved by dilation when ticks occur.

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 ‹$\exists\, k_0.\ f\ k_0 = k \wedge time\ ((Rep\text{-}run\ sub)\ k_0\ c) = \tau$›
proof −
from *ticks-image-sub'*[*OF assms*(*1*,*2*)] **have** ‹$\exists\, k_0.\ f\ k_0 = k$› **.**
from *this* **obtain** k_0 **where** ‹$f\ k_0 = k$› **by** *blast*
moreover with *assms*(*1*,*3*) **have** ‹$time\ ((Rep\text{-}run\ sub)\ k_0\ c) = \tau$› **by** (*simp add*: *dilating-def*)
ultimately show *?thesis* **by** *blast*
qed

TESL operators are preserved by dilation.

lemma *ticks-sub*:
assumes ‹*dilating f sub r*›
shows ‹$hamlet\ ((Rep\text{-}run\ sub)\ n\ a) = hamlet\ ((Rep\text{-}run\ r)\ (f\ n)\ a)$›
using *assms* **by** (*simp add*: *dilating-def*)

lemma *no-tick-sub*:
assumes ‹*dilating f sub r*›
shows ‹$(\nexists\, n_0.\ f\ n_0 = n) \longrightarrow \neg hamlet\ ((Rep\text{-}run\ r)\ n\ a)$›
using *assms dilating-def dilating-fun-def* **by** *blast*

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

definition *opt-lift*::‹$('a \Rightarrow 'a) \Rightarrow ('a\ option \Rightarrow 'a\ option)$›
where
‹$opt\text{-}lift\ f \equiv \lambda x.\ case\ x\ of\ None \Rightarrow None \mid Some\ y \Rightarrow Some\ (f\ y)$›

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 ‹$\{k.\ hamlet\ ((Rep\text{-}run\ r)\ k\ c)\} = image\ f\ \{k.\ hamlet\ ((Rep\text{-}run\ sub)\ k\ c)\}$›
(**is** ‹*?R* = *image f ?S*›)
proof
{ fix k **assume** *h*:‹$k \in$ *?R*›
with *no-tick-sub*[*OF assms*] **have** ‹$\exists\, k_0.\ f\ k_0 = k$› **by** *blast*
from *this* **obtain** k_0 **where** *k0prop*:‹$f\ k_0 = k$› **by** *blast*
with *ticks-sub*[*OF assms*] *h* **have** ‹$hamlet\ ((Rep\text{-}run\ sub)\ k_0\ c)$› **by** *blast*
with *k0prop* **have** ‹$k \in image\ f\ ?S$› **by** *blast*
}
thus ‹*?R* ⊆ *image f ?S*› **by** *blast*
next
{ fix k **assume** *h*:‹$k \in image\ f\ ?S$›
from *this* **obtain** k_0 **where** ‹$f\ k_0 = k \wedge hamlet\ ((Rep\text{-}run\ sub)\ k_0\ c)$› **by** *blast*
with *assms* **have** ‹$k \in$ *?R*› **using** *ticks-sub* **by** *blast*
}
thus ‹$image\ f\ ?S \subseteq$ *?R*› **by** *blast*
qed

Strictly monotonous functions preserve the least element.

lemma *Least-strict-mono*:
assumes ‹*strict-mono f*›
and ‹$\exists\, x \in S.\ \forall\, y \in S.\ x \leq y$›
shows ‹$(LEAST\ y.\ y \in f\ `\ S) = f\ (LEAST\ x.\ x \in S)$›
using *Least-mono*[*OF strict-mono-mono*, *OF assms*] **.**

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

lemma *Least-nat-ex*:
‹$(n{::}nat) \in S \Longrightarrow \exists\, x \in S.\ (\forall\, y \in S.\ x \leq y)$›
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 ‹$\exists\, k{::}nat.$ *hamlet* $((Rep\text{-}run\ sub)\ k\ c)$›
shows ‹$(LEAST\ k.\ k \in \{t.$ *hamlet* $((Rep\text{-}run\ r)\ t\ c)\}) = f\ (LEAST\ k.\ k \in \{t.$ *hamlet* $((Rep\text{-}run\ sub)\ t\ c)\})$›
(**is** ‹$(LEAST\ k.\ k \in ?R) = f\ (LEAST\ k.\ k \in ?S)$›)
proof −
from *assms*(*2*) **have** ‹$\exists\, x.\ x \in ?S$› **by** *simp*
hence *least*:‹$\exists\, x \in ?S.\ \forall\, y \in ?S.\ x \leq y$›
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**
‹$(LEAST\ y.\ y \in f\ `\ ?S)\ = f\ (LEAST\ x.\ x \in ?S)$› **.**
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 ‹$\exists\, k.$ *hamlet* $((Rep\text{-}run\ r)\ k\ c)$›
shows ‹$\exists\, k_0.$ *hamlet* $((Rep\text{-}run\ sub)\ k_0\ c)$›
proof −
from *assms*(*2*) **obtain** *k* **where** ‹*hamlet* $((Rep\text{-}run\ r)\ k\ c)$› **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* $((Rep\text{-}run\ r)\ k\ c)$›
shows ‹$\exists\, k_0.\ f\ k_0 = k \wedge$ *hamlet* $((Rep\text{-}run\ sub)\ k_0\ c)$›
proof −
from *no-tick-sub*[*OF assms*(*1*)] *assms*(*2*) **have** ‹$\exists\, k_0.\ f\ k_0 = k$› **by** *blast*
from *this* **obtain** k_0 **where** ‹$f\ k_0 = k$› **by** *blast*

moreover with *ticks-sub*[*OF assms*(*1*)] *assms*(*2*) **have** ‹*hamlet* ((*Rep-run sub*) k_0 *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 i_0 **where** *i0prop*:‹*f* i_0 = *i* ∧ *hamlet* ((*Rep-run sub*) i_0 *c*)› **by** *blast*
with *h* **have** ‹*f m* < *f* i_0 ∧ *f* i_0 < *f n*› **by** *simp*
moreover have ‹*strict-mono f*› **using** *assms dilating-def dilating-fun-def* **by** *blast*
ultimately have ‹*m* < i_0 ∧ i_0 < *n*› **using** *strict-mono-less strict-mono-less-eq* **by** *blast*
with *i0prop* **have** ‹∃ i_0. *f* i_0 = *i* ∧ i_0 ∈ *?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
```

The set definition and the function definition of *tick-count-strict* are equivalent.

```
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

lemma cong-suc-collect:
  assumes ‹⋀r K n. P r K n = P' r K n›
      and ‹⋀r K n. Q r K n = Q' r K n›
      and ‹⋀r K n. Q r K (Suc n) = P r K n›
    shows ‹⋀K1 K2 n. {r. P' r K2 n ≤ Q' r K1 n} = {r. Q' r K2 (Suc n) ≤ Q'
r K1 n}›
  using assms by auto
```

lemma *strictly-precedes-alt-def1*:
‹{ ϱ. ∀ n::*nat*. (*run-tick-count* ϱ K_2 n) ≤ (*run-tick-count-strictly* ϱ K_1 n) }
= { ϱ. ∀ n::*nat*. (*run-tick-count-strictly* ϱ K_2 (*Suc* n)) ≤ (*run-tick-count-strictly* ϱ K_1 n) }›
using *cong-suc-collect*[*of tick-count run-tick-count tick-count-strict run-tick-count-strictly*,
OF tick-count-is-fun tick-count-strict-is-fun tick-count-strict-suc]

by *simp*

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 K_2 n) ≤ (*run-tick-count-strictly* ϱ K_1 n) }
= { ϱ. (¬*hamlet* ((*Rep-run* ϱ) *0* K_2)) ∧ (∀ n::*nat*. (*run-tick-count* ϱ K_2 (*Suc* n)) ≤ (*run-tick-count* ϱ K_1 n)) }›
(**is** ‹*?P* = *?P'*›)
proof
{ fix r::‹'*a run*›
assume ‹r ∈ *?P*›
hence ‹∀ n::*nat*. (*run-tick-count* r K_2 n) ≤ (*run-tick-count-strictly* r K_1 n)› **by** *simp*
hence *1*:‹∀ n::*nat*. (*tick-count* r K_2 n) ≤ (*tick-count-strict* r K_1 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 K_2 (*Suc* n)) ≤ (*tick-count-strict* r K_1 n)›
using *tick-count-strict-suc*[*symmetric*, *of* ‹r› ‹K_2›] **by** *simp*
hence ‹∀ n::*nat*. (*tick-count-strict* r K_2 (*Suc* (*Suc* n))) ≤ (*tick-count-strict* r K_1 (*Suc* n))› **by** *simp*
hence ‹∀ n::*nat*. (*tick-count* r K_2 (*Suc* n)) ≤ (*tick-count* r K_1 n)›
using *tick-count-strict-suc*[*symmetric*, *of* ‹r›] **by** *simp*
hence ∗:‹∀ n::*nat*. (*run-tick-count* r K_2 (*Suc* n)) ≤ (*run-tick-count* r K_1 n)›
by (*simp add*: *tick-count-is-fun*)
from *1* **have** ‹*tick-count* r K_2 *0* <= *tick-count-strict* r K_1 *0*› **by** *simp*
moreover have ‹*tick-count-strict* r K_1 *0* = *0*› **unfolding** *tick-count-strict-def* **by** *simp*
ultimately have ‹*tick-count* r K_2 *0* = *0*› **by** *simp*
hence ‹¬*hamlet* ((*Rep-run* r) *0* K_2)› **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 K_2 (*Suc* n)) ≤ (*run-tick-count* r K_1 n)› **by** *simp*
hence ‹∀ n::*nat*. (*tick-count* r K_2 (*Suc* n)) ≤ (*tick-count* r K_1 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

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)

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* $\{i.\ m < i \wedge i < n \wedge P\ i\}$› **by** *simp*
moreover have ‹*finite* $\{i.\ i = n \wedge P\ i\}$› **by** *simp*
ultimately show *?thesis* **using** *card-Un-disjoint*[*of* ‹$\{i.\ m < i \wedge i < n \wedge P\ i\}$› ‹$\{i.\ i = n \wedge P\ i\}$›] **by** *simp*
qed

lemma *card-le-leq-0*:‹*card* $\{i$::*nat*. $i \leq n \wedge P\ i\} =$ *card* $\{i.\ i < n \wedge P\ i\} +$ *card* $\{i.\ i = n \wedge P\ i\}$›
proof −
have ‹$\{i$::*nat*. $i < n \wedge P\ i\} \cap \{i.\ i = n \wedge P\ i\} = \{\}$› **by** *auto*
moreover have ‹$\{i.\ i < n \wedge P\ i\} \cup \{i.\ i = n \wedge P\ i\} = \{i.\ i \leq n \wedge P\ i\}$› **by** *auto*
moreover have ‹*finite* $\{i.\ i < n \wedge P\ i\}$› **by** *simp*
moreover have ‹*finite* $\{i.\ i = n \wedge P\ i\}$› **by** *simp*
ultimately show *?thesis* **using** *card-Un-disjoint*[*of* ‹$\{i.\ i < n \wedge P\ i\}$› ‹$\{i.\ i = n \wedge P\ i\}$›] **by** *simp*
qed

lemma *card-mnm*:
assumes ‹$m < n$›
shows ‹*card* $\{i$::*nat*. $i < n \wedge P\ i\} =$ *card* $\{i.\ i \leq m \wedge P\ i\} +$ *card* $\{i.\ m < i \wedge i < n \wedge P\ i\}$›
proof −
have *1*:‹$\{i$::*nat*. $i \leq m \wedge P\ i\} \cap \{i.\ m < i \wedge i < n \wedge P\ i\} = \{\}$› **by** *auto*
from *assms* **have** ‹$\forall\, i$::*nat*. $i < n = (i \leq m) \vee (m < i \wedge i < n)$› **using** *less-trans* **by** *auto*
hence *2*:
‹$\{i$::*nat*. $i < n \wedge P\ i\} = \{i.\ i \leq m \wedge P\ i\} \cup \{i.\ m < i \wedge i < n \wedge P\ i\}$› **by** *blast*
have *3*:‹*finite* $\{i.\ i \leq m \wedge P\ i\}$› **by** *simp*
have *4*:‹*finite* $\{i.\ m < i \wedge i < n \wedge P\ i\}$› **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*. $i < n \wedge P\ i\} =$ *card* $\{i.\ i < m \wedge P\ i\} +$ *card* $\{i.\ m \leq i \wedge i < n \wedge P\ i\}$›
proof −
have *1*:‹$\{i$::*nat*. $i < m \wedge P\ i\} \cap \{i.\ m \leq i \wedge i < n \wedge P\ i\} = \{\}$› **by** *auto*
from *assms* **have** ‹$\forall\, i$::*nat*. $i < n = (i < m) \vee (m \leq i \wedge i < n)$› **using** *less-trans* **by** *auto*
hence *2*:
‹$\{i$::*nat*. $i < n \wedge P\ i\} = \{i.\ i < m \wedge P\ i\} \cup \{i.\ m \leq i \wedge i < n \wedge P\ i\}$› **by** *blast*
have *3*:‹*finite* $\{i.\ i < m \wedge P\ i\}$› **by** *simp*
have *4*:‹*finite* $\{i.\ m \leq i \wedge i < n \wedge P\ i\}$› **by** *simp*
from *card-Un-disjoint*[*OF 3 4 1*] *2* **show** *?thesis* **by** *simp*
qed

lemma *nat-interval-union*:
assumes ‹$m \leq n$›
shows ‹$\{i::nat.\ i \leq n \wedge P\ i\} = \{i::nat.\ i \leq m \wedge P\ i\} \cup \{i::nat.\ m < i \wedge i \leq n \wedge P\ i\}$›
using *assms le-cases nat-less-le* **by** *auto*

lemma *no-tick-before-suc*:
assumes ‹*dilating f sub r*›
and ‹$(f\ n) < k \wedge k < (f\ (Suc\ n))$›
shows ‹$\neg$*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*:‹$f\ n < k \wedge k < f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)›
hence ‹$\exists\, k_0.\ f\ k_0 = k$› **using** *assms*(*1*) *dilating-def dilating-fun-def* **by** *blast*
from *this* **obtain** k_0 **where** ‹$f\ k_0 = k$› **by** *blast*
with *h* **have** ‹$f\ n < f\ k_0 \wedge f\ k_0 < f\ (Suc\ n)$› **by** *simp*
hence *False* **using** *smf not-less-eq strict-mono-less* **by** *blast*
} thus *?thesis* **using** *assms*(*2*) **by** *blast*
qed

lemma *tick-count-fsuc*:
assumes ‹*dilating f sub r*›
shows ‹*tick-count r c* (*f* (*Suc n*)) = *tick-count r c* (*f n*) + *card* $\{k.\ k = f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$›
proof −
have *smf*:‹*strict-mono f*› **using** *assms dilating-def dilating-fun-def* **by** *blast*
moreover have ‹*finite* $\{k.\ k \leq f\ n \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$› **by** *simp*
moreover have ∗:‹*finite* $\{k.\ f\ n < k \wedge k \leq f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$› **by** *simp*
ultimately have ‹$\{k.\ k \leq f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\} =$
$\{k.\ k \leq f\ n \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$
$\cup\ \{k.\ f\ n < k \wedge k \leq f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$›
by (*simp add*: *nat-interval-union strict-mono-less-eq*)
moreover have ‹$\{k.\ k \leq f\ n \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$
$\cap\ \{k.\ f\ n < k \wedge k \leq f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\} = \{\}$›
by *auto*
ultimately have ‹*card* $\{k.\ k \leq f\ (Suc\ n) \wedge$ *hamlet* (*Rep-run r k c*)$\} =$
card $\{k.\ k \leq f\ n \wedge$ *hamlet* (*Rep-run r k c*)$\}$
$+$ *card* $\{k.\ f\ n < k \wedge k \leq f\ (Suc\ n) \wedge$ *hamlet* (*Rep-run r k c*)$\}$›
by (*simp add*: ∗ *card-Un-disjoint*)
moreover from *no-tick-before-suc*[*OF assms*] **have**
‹$\{k.\ f\ n < k \wedge k \leq f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\} =$
$\{k.\ k = f\ (Suc\ n) \wedge$ *hamlet* ((*Rep-run r*) *k c*)$\}$›
using *smf strict-mono-less* **by** *fastforce*
ultimately show *?thesis* **by** (*simp add*: *tick-count-def*)
qed

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

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*)› ‹λ*k*. *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*)

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 ‹... = $\#_{\leq}$ *r c* (*f n*)› **using** *tick-count-is-fun*[*of r c* ‹*f n*›] .
finally show *?thesis* .
qed

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

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

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

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

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

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

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 n_p
      where ‹f n_p < n ∧ (∀ k. f n_p < k ∧ k ≤ n ⟶ (∄ k₀. f k₀ = k))› by blast
    with Suc(2) have ‹f n_p < (Suc n) ∧ (∀ k. f n_p < k ∧ k ≤ (Suc n) ⟶ (∄ k₀.
f k₀ = k))›
      by (metis le-SucE less-Suc-eq)
    thus ?thesis by blast
  qed
qed

lemma strict-mono-suc:
  assumes ‹strict-mono f›
      and ‹f sn = Suc (f n)›
    shows ‹sn = Suc n›
by (metis Suc-lessI assms lessI not-less-eq strict-mono-def strict-mono-less)

lemma next-non-stuttering:
  assumes ‹dilating f sub r›
      and ‹f n_p < n ∧ (∀ k. f n_p < k ∧ k ≤ n ⟶ (∄ k₀. f k₀ = k))›
      and ‹f sn₀ = Suc n›
    shows ‹sn₀ = Suc n_p›
proof −
 from assms(1) have smf:‹strict-mono f› by (simp add: dilating-def dilating-fun-def)
  from assms(2) have ‹f n_p < n› by simp
  with smf assms(3) have ∗:‹sn₀ > n_p› using strict-mono-less by fastforce
   from assms(2) have ‹f (Suc n_p) > n› by (metis lessI not-le-imp-less smf
strict-mono-less)
  hence ‹Suc n ≤ f (Suc n_p)› by simp
 hence ‹sn₀ ≤ Suc n_p› using assms(3) smf using strict-mono-less-eq by fastforce
  with ∗ show ?thesis by simp
qed

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

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* ⟹ (∄ *k*$_0$. *f k*$_0$ = *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

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*)
hence ∗∗:‹*time* ((*Rep-run r*) (*f n*) *c*) = *t* ∧ (∀ *m* < *n*. *time*((*Rep-run r*) (*f m*) *c*) < *t*)›
using *assms*(*1*) *dilating-def* **by** *metis*
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* ‹∃ *m*$_0$. *f m*$_0$ = *m*›)
case *True*
from *this* **obtain** *m*$_0$ **where** *mm0*:‹*m* = *f m*$_0$› **by** *blast*
with *hyp* **have** *m0n*:‹*m*$_0$ < *n*› **using** *assms*(*1*) **by** (*simp add*: *dilating-def dilating-fun-def strict-mono-less*)
hence ‹*time* ((*Rep-run sub*) *m*$_0$ *c*) < *t*› **using** ∗ **by** *blast*
thus *?thesis* **by** (*simp add*: *mm0 m0n* ∗∗)
next
case *False*
hence ‹∃ *m*$_p$. *f m*$_p$ < *m* ∧ (∀ *k*. *f m*$_p$ < *k* ∧ *k* ≤ *m* ⟶ (∄ *k*$_0$. *f k*$_0$ = *k*))› **using** *greatest-prev-image*[*OF assms*] **by** *simp*
from *this* **obtain** *m*$_p$ **where** *mp*:‹*f m*$_p$ < *m* ∧ (∀ *k*. *f m*$_p$ < *k* ∧ *k* ≤ *m* ⟶ (∄ *k*$_0$. *f k*$_0$ = *k*))› **by** *blast*
hence ‹*time* ((*Rep-run r*) *m c*) = *time* ((*Rep-run sub*) *m*$_p$ *c*)› **using** *time-stuttering*[*OF assms*] **by** *blast*
moreover from *mp* **have** ‹*time* ((*Rep-run sub*) *m*$_p$ *c*) < *t*› **using** ∗
by (*meson assms dilating-def dilating-fun-def hyp less-trans strict-mono-less*)

```
      ultimately 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

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

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 ‹{i. f i ≤ n} ≠ {}›
   by (metis dilating-def dilating-fun-def empty-iff le0 mem-Collect-eq)
 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

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))⟩
    by (auto simp add: dilating-def dilating-fun-def)
  have finite-prefix:⟨⋀n. finite {i. f i ≤ n}⟩
    by (metis assms dilating-def dilating-fun-def finite-less-ub)
  have prefix-not-empty:⟨⋀n. {i. f i ≤ n} ≠ {}⟩
    by (metis assms dilating-def dilating-fun-def empty-iff le0 mem-Collect-eq)


  have 1:⟨mono (dil-inverse f)⟩
  proof −
  { fix x::⟨nat⟩ and y::⟨nat⟩ assume hyp:⟨x ≤ y⟩
    from smf have finite:⟨finite {i. f i ≤ y}⟩
      by (metis (full-types) assms dilating-def dilating-fun-def finite-less-ub)
    from assms have f 0 = 0 by (simp add: dilating-def dilating-fun-def)
    hence notempty:⟨{i. f i ≤ x} ≠ {}⟩ by (metis empty-Collect-eq le0)
    hence inc:⟨{i. f i ≤ x} ⊆ {i. f i ≤ y}⟩
      by (simp add: hyp Collect-mono le-trans)
    from Max-mono[OF inc notempty finite] have (dil-inverse f) x ≤ (dil-inverse
f) y
      unfolding dil-inverse-def .
  } thus ?thesis unfolding mono-def by simp
  qed

  from assms have f0:f 0 = 0 by (simp add: dilating-def dilating-fun-def)
  from first-dilated-instant[OF smf this] have 2:⟨(dil-inverse f) 0 = 0⟩
    unfolding dil-inverse-def .

  from assms(1) dilating-def dilating-fun-def have fge:⟨∀ n. f n ≥ n⟩ by blast
  hence ⟨∀ 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 4:⟨∀ 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 5:⟨(∀ 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
      thus ?thesis by (metis assms dilating-def)
    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 * 5 4 show ?thesis unfolding contracting-def by simp
qed

end
```

7.1.4 Main Theorems

```
theory Stuttering
imports StutteringLemmas

begin
```

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* ∈ ⟦c_1 *implies* c_2⟧$_{TESL}$›
shows ‹*r* ∈ ⟦c_1 *implies* c_2⟧$_{TESL}$›
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* c_1) ⟶ *hamlet* ((*Rep-run r*) *n* c_2)}› **by** *simp*
hence ‹∀ *n*. *hamlet* ((*Rep-run sub*) *n* c_1) ⟶ *hamlet* ((*Rep-run sub*) *n* c_2)› **by** *simp*
ultimately have ‹∀ *n*. *hamlet* ((*Rep-run r*) *n* c_1) ⟶ *hamlet* ((*Rep-run r*) *n* c_2)›
using *ticks-imp-ticks-subk ticks-sub* **by** *blast*
thus *?thesis* **by** *simp*
qed

theorem *implies-not-sub*:
assumes ‹*sub* ≪ *r*›
and ‹*sub* ∈ ⟦c_1 *implies not* c_2⟧$_{TESL}$›
shows ‹*r* ∈ ⟦c_1 *implies not* c_2⟧$_{TESL}$›
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* c_1) ⟶ ¬ *hamlet* ((*Rep-run r*) *n* c_2)}›
by *simp*
hence ‹∀ *n*. *hamlet* ((*Rep-run sub*) *n* c_1) ⟶ ¬ *hamlet* ((*Rep-run sub*) *n* c_2)› **by** *simp*
ultimately have ‹∀ *n*. *hamlet* ((*Rep-run r*) *n* c_1) ⟶ ¬ *hamlet* ((*Rep-run r*) *n* c_2)›
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* ∈ ⟦c_1 *weakly precedes* c_2⟧$_{TESL}$›
shows ‹*r* ∈ ⟦c_1 *weakly precedes* c_2⟧$_{TESL}$›
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* c_2 *n*) ≤ (*run-tick-count r* c_1 *n*)}› **by** *simp*
hence ‹∀ *n*. (*run-tick-count sub* c_2 *n*) ≤ (*run-tick-count sub* c_1 *n*)› **by** *simp*
from *dil-tick-count*[*OF assms*(*1*) *this*] **have** ‹∀ *n*. (*run-tick-count r* c_2 *n*) ≤ (*run-tick-count r* c_1 *n*)› **by** *simp*
thus *?thesis* **by** *simp*
qed

theorem *strictly-precedes-sub*:
assumes ‹*sub* ≪ *r*›
and ‹*sub* ∈ ⟦*c*$_1$ *strictly precedes* *c*$_2$⟧$_{TESL}$›
shows ‹*r* ∈ ⟦*c*$_1$ *strictly precedes* *c*$_2$⟧$_{TESL}$›
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 *c*$_2$ *n*) ≤ (*run-tick-count-strictly* ϱ *c*$_1$ *n*) }› **by** *simp*
with *strictly-precedes-alt-def2*[*of* ‹*c*$_2$› ‹*c*$_1$›] **have**
‹*sub* ∈ { ϱ. (¬*hamlet* ((*Rep-run* ϱ) *0* *c*$_2$)) ∧ (∀ *n*::*nat*. (*run-tick-count* ϱ *c*$_2$ (*Suc* *n*)) ≤ (*run-tick-count* ϱ *c*$_1$ *n*)) }›
by *blast*
hence ‹(¬*hamlet* ((*Rep-run sub*) *0* *c*$_2$)) ∧ (∀ *n*::*nat*. (*run-tick-count sub* *c*$_2$ (*Suc* *n*)) ≤ (*run-tick-count sub* *c*$_1$ *n*))›
by *simp*
hence
1:‹(¬*hamlet* ((*Rep-run sub*) *0* *c*$_2$)) ∧ (∀ *n*::*nat*. (*tick-count sub* *c*$_2$ (*Suc* *n*)) ≤ (*tick-count sub* *c*$_1$ *n*))›
by (*simp add*: *tick-count-is-fun*)
have ‹∀ *n*::*nat*. (*tick-count r* *c*$_2$ (*Suc* *n*)) ≤ (*tick-count r* *c*$_1$ *n*)›
proof −
{ fix *n*::*nat*
have ‹*tick-count r* *c*$_2$ (*Suc* *n*) ≤ *tick-count r* *c*$_1$ *n*›
proof (*cases* ‹∃ *n*$_0$. *f* *n*$_0$ = *n*›)
case *True* — n is in the image of f
from *this* **obtain** *n*$_0$ **where** *fn*:‹*f* *n*$_0$ = *n*› **by** *blast*
show *?thesis*
proof (*cases* ‹∃ *sn*$_0$. *f* *sn*$_0$ = *Suc* *n*›)
case *True* — Suc n is in the image of f
from *this* **obtain** *sn*$_0$ **where** *fsn*:‹*f* *sn*$_0$ = *Suc* *n*› **by** *blast*
with *fn* **have** ‹*sn*$_0$ = *Suc* *n*$_0$› **using** *strict-mono-suc* ∗ *dilating-def dilating-fun-def* **by** *blast*
with *1* **have** ‹*tick-count sub* *c*$_2$ *sn*$_0$ ≤ *tick-count sub* *c*$_1$ *n*$_0$› **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*) *c*$_2$)›
using ∗ **by** (*simp add*: *dilating-def dilating-fun-def*)
hence ‹*tick-count r* *c*$_2$ (*Suc* *n*) = *tick-count r* *c*$_2$ *n*› **by** (*simp add*: *tick-count-suc*)
also have ‹... = *tick-count sub* *c*$_2$ *n*$_0$› **using** *fn tick-count-sub*[*OF* ∗] **by** *simp*
finally have ‹*tick-count r* *c*$_2$ (*Suc* *n*) = *tick-count sub* *c*$_2$ *n*$_0$› .
moreover have ‹*tick-count sub* *c*$_2$ *n*$_0$ ≤ *tick-count sub* *c*$_2$ (*Suc* *n*$_0$)›
by (*simp add*: *tick-count-suc*)
ultimately have ‹*tick-count r* *c*$_2$ (*Suc* *n*) ≤ *tick-count sub* *c*$_2$ (*Suc* *n*$_0$)› **by** *simp*
moreover have ‹*tick-count sub* *c*$_2$ (*Suc* *n*$_0$) ≤ *tick-count sub* *c*$_1$ *n*$_0$›

using *1* **by** *simp*
ultimately have ‹*tick-count r* c_2 (*Suc n*) ≤ *tick-count sub* c_1 n_0› **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** n_p
where *np-prop*:‹*f* n_p < *n* ∧ (∀ *k*. *f* n_p < *k* ∧ *k* ≤ *n* ⟶ (∄ k_0. *f* k_0 = *k*))› **by** *blast*
from *tick-count-latest*[*OF* ∗ *this*] **have** ‹*tick-count r* c_1 *n* = *tick-count r* c_1 (*f* n_p)› .
hence *a*:‹*tick-count r* c_1 *n* = *tick-count sub* c_1 n_p› **using** *tick-count-sub*[*OF* ∗] **by** *simp*
have *b*: ‹*tick-count sub* c_2 (*Suc* n_p) ≤ *tick-count sub* c_1 n_p› **using** *1* **by** *simp*
show *?thesis*
proof (*cases* ‹∃ sn_0. *f* sn_0 = *Suc n*›)
case *True* — Suc n is in the image of f
from *this* **obtain** sn_0 **where** *fsn*:‹*f* sn_0 = *Suc n*› **by** *blast*
from *next-non-stuttering*[*OF* ∗ *np-prop this*] **have** *sn-prop*:‹sn_0 = *Suc* n_p› .
with *b* **have** ‹*tick-count sub* c_2 sn_0 ≤ *tick-count sub* c_1 n_p› **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*) c_2)›
using ∗ **by** (*simp add*: *dilating-def dilating-fun-def*)
hence ‹*tick-count r* c_2 (*Suc n*) = *tick-count r* c_2 *n*› **by** (*simp add*: *tick-count-suc*)
also have ‹... = *tick-count sub* c_2 n_p› **using** *np-prop tick-count-sub*[*OF* ∗]
by (*simp add*: *tick-count-latest*[*OF* ∗ *np-prop*])
finally have ‹*tick-count r* c_2 (*Suc n*) = *tick-count sub* c_2 n_p› .
moreover have ‹*tick-count sub* c_2 n_p ≤ *tick-count sub* c_2 (*Suc* n_p)›
by (*simp add*: *tick-count-suc*)
ultimately have ‹*tick-count r* c_2 (*Suc n*) ≤ *tick-count sub* c_2 (*Suc* n_p)› **by** *simp*
moreover have ‹*tick-count sub* c_2 (*Suc* n_p) ≤ *tick-count sub* c_1 n_p› **using** *1* **by** *simp*
ultimately have ‹*tick-count r* c_2 (*Suc n*) ≤ *tick-count sub* c_1 n_p› **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* c_2)›
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 ∗∗:‹∀ n_0. *hamlet* ((*Rep-run r*) (*f* n_0) *a*)
⟶ (∀ m_0 ≥ n_0. *first-time r ms* (*f* m_0) (*time* ((*Rep-run r*) (*f* n_0) *ms*) + δτ)
⟶ *hamlet* ((*Rep-run r*) (*f* m_0) *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** n_0 **where** *nfn0*:‹*n* = *f* n_0› **by** *blast*
with ∗∗ *assm* **have** *ft0*:
‹(∀ m_0 ≥ n_0. *first-time r ms* (*f* m_0) (*time* ((*Rep-run r*) (*f* n_0) *ms*) + δτ)
⟶ *hamlet* ((*Rep-run r*) (*f* m_0) *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* ‹∃ m_0. *f* m_0 = *m*›)
case *True*
from *this* **obtain** m_0 **where** ‹*m* = *f* m_0› **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 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
```

A time relation is 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

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

lemma atomic-sub:
 assumes ‹sub ≪ r›
    and ‹sub ∈ ⟦ φ ⟧TESL›
  shows ‹r ∈ ⟦ φ ⟧TESL›
proof (cases φ)
 case (SporadicOn)
   thus ?thesis using assms(2) sporadic-sub[OF assms(1)] by simp
next
 case (TagRelation)
   thus ?thesis using assms(2) tagrel-sub[OF assms(1)] by simp
next
```

```
  case (Implies)
    thus ?thesis using assms(2) implies-sub[OF assms(1)] by simp
next
  case (ImpliesNot)
    thus ?thesis using assms(2) implies-not-sub[OF assms(1)] by simp
next
  case (TimeDelayedBy)
    thus ?thesis using assms(2) time-delayed-sub[OF assms(1)] by simp
next
  case (WeaklyPrecedes)
    thus ?thesis using assms(2) weakly-precedes-sub[OF assms(1)] by simp
next
  case (StrictlyPrecedes)
    thus ?thesis using assms(2) strictly-precedes-sub[OF assms(1)] by simp
next
  case (Kills)
    thus ?thesis using assms(2) kill-sub[OF assms(1)] by simp
qed

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
```

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.