Probabilistic Timed Automata

Simon Wimmer and Johannes Hölzl

March 17, 2025

Abstract

We present a formalization of probabilistic timed automata (PTA) for which we try to follow the formula "MDP + TA = PTA" as far as possible: our work starts from our existing formalizations of Markov decision processes (MDP) and timed automata (TA) and combines them modularly. We prove the fundamental result for probabilistic timed automata: the region construction that is known from timed automata carries over to the probabilistic setting. In particular, this allows us to prove that minimum and maximum reachability probabilities can be computed via a reduction to MDP model checking, including the case where one wants to disregard unrealizable behavior. Further information can be found in our ITP paper [2].

The definition of the PTA semantics can be found in Section 3.3, the region MDP is in Section 4.1, the bisimulation theorem is in Section 1, and the final theorems can be found in Section 7.4. The background theory we formalize is described in the seminal paper on PTA [1].

Contents

1 Bisimulation on a Relation **3**

2 Additional Facts on Regions **6**

3 Definition and Semantics **10**
3.1 Syntactic Definition 10
3.1.1 Collecting Information About Clocks 11
3.2 Operational Semantics as an MDP 11
3.3 Syntactic Definition 12

4 Constructing the Corresponding Finite MDP on Regions **12**
4.1 Syntactic Definition 13
4.2 Many Closure Properties 13
4.3 The Region Graph is a Finite MDP 14

5 Relating the MDPs **16**
5.1 Translating From K to $\mathcal{K}$ 16
5.2 Translating Configurations 20
5.2.1 States 20
5.2.2 Intermezzo 22
5.2.3 Predicates 23
5.2.4 Distributions 23
5.2.5 Configuration 31
5.3 Equalities Between Measures of Trace Spaces 42

6 Classifying Regions for Divergence **46**
6.1 Pairwise 46
6.2 Regions 47
6.3 Unbounded and Zero Regions 48

7 Reachability **48**
7.1 Definitions 48
7.2 Easier Result on All Configurations 49
7.3 Divergent Adversaries 51
7.4 Main Result 81

theory *PTA*
  imports *library/Lib*
begin

1 Bisimulation on a Relation

definition $\mathit{rel\text{-}set\text{-}strong} :: ('a \Rightarrow 'b \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{set} \Rightarrow 'b\ \mathit{set} \Rightarrow \mathit{bool}$
  where $\mathit{rel\text{-}set\text{-}strong}\ R\ A\ B \longleftrightarrow (\forall x\ y.\ R\ x\ y \longrightarrow (x \in A \longleftrightarrow y \in B))$

lemma $\mathit{T\text{-}eq\text{-}rel\text{-}half}[\mathit{consumes}\ 4,\ \mathit{case\text{-}names}\ \mathit{prob}\ \mathit{sets}\ \mathit{cont}]$:
  fixes $R :: 's \Rightarrow 't \Rightarrow \mathit{bool}$ **and** $f :: 's \Rightarrow 't$ **and** $S :: 's\ \mathit{set}$
  assumes $\mathit{R\text{-}def}$: $\bigwedge s\ t.\ R\ s\ t \longleftrightarrow (s \in S \wedge f\ s = t)$
  assumes $A[\mathit{measurable}]$: $A \in \mathit{sets}\ (\mathit{stream\text{-}space}\ (\mathit{count\text{-}space}\ \mathit{UNIV}))$
    and $B[\mathit{measurable}]$: $B \in \mathit{sets}\ (\mathit{stream\text{-}space}\ (\mathit{count\text{-}space}\ \mathit{UNIV}))$
    and AB: $\mathit{rel\text{-}set\text{-}strong}\ (\mathit{stream\text{-}all2}\ R)\ A\ B$ **and** KL: $\mathit{rel\text{-}fun}\ R\ (\mathit{rel\text{-}pmf}\ R)\ K\ L$ **and** xy: $R\ x\ y$
  shows $\mathit{MC\text{-}syntax}.T\ K\ x\ A = \mathit{MC\text{-}syntax}.T\ L\ y\ B$
proof $-$
  interpret K: $\mathit{MC\text{-}syntax}\ K$ **by** *unfold-locales*
  interpret L: $\mathit{MC\text{-}syntax}\ L$ **by** *unfold-locales*

  have $x \in S$ **using** $\langle R\ x\ y \rangle$ **by** $(\mathit{auto}\ \mathit{simp}: \mathit{R\text{-}def})$

  define g **where** $g\ t = (\mathit{SOME}\ s.\ R\ s\ t)$ **for** t
  have $\mathit{measurable\text{-}g}$: $g \in \mathit{count\text{-}space}\ \mathit{UNIV} \to_M \mathit{count\text{-}space}\ \mathit{UNIV}$ **by** *auto*
  have g: $R\ i\ j \Longrightarrow R\ (g\ j)\ j$ **for** $i\ j$
    unfolding *g-def* **by** $(\mathit{rule}\ \mathit{someI})$

  have $\mathit{K\text{-}subset}$: $x \in S \Longrightarrow K\ x \subseteq S$ **for** x
    using $KL[\mathit{THEN}\ \mathit{rel\text{-}funD},\ \mathit{of}\ x\ f\ x,\ \mathit{THEN}\ \mathit{rel\text{-}pmf\text{-}imp\text{-}rel\text{-}set}]$ **by** $(\mathit{auto}\ \mathit{simp}: \mathit{rel\text{-}set\text{-}def}\ \mathit{R\text{-}def})$

  have $\mathit{in\text{-}S}$: $\mathit{AE}\ \omega\ \mathit{in}\ K.T\ x.\ \omega \in \mathit{streams}\ S$
    using *K.AE-T-enabled*
  proof *eventually-elim*
    case $(\mathit{elim}\ \omega)$ **with** $\langle x \in S \rangle$ **show** *?case*
      apply $(\mathit{coinduction}\ \mathit{arbitrary}: x\ \omega)$
      subgoal for $x\ \omega$ **using** *K-subset* **by** $(\mathit{cases}\ \omega)$ $(\mathit{auto}\ \mathit{simp}: K.\mathit{enabled\text{-}Stream})$
      done
  qed

  have $\mathit{L\text{-}eq}$: $L\ y = \mathit{map\text{-}pmf}\ f\ (K\ x)$ **if** xy: $R\ x\ y$ **for** $x\ y$
  proof $-$
    have $\mathit{rel\text{-}pmf}\ (\lambda x\ y.\ x = y)\ (\mathit{map\text{-}pmf}\ f\ (K\ x))\ (L\ y)$
      using $KL[\mathit{THEN}\ \mathit{rel\text{-}funD},\ \mathit{OF}\ xy]$ **by** $(\mathit{auto}\ \mathit{intro}: \mathit{pmf.rel\text{-}mono\text{-}strong}\ \mathit{simp}: \mathit{R\text{-}def}\ \mathit{pmf.rel\text{-}map})$
    then show *?thesis* **unfolding** *pmf.rel-eq* **by** *simp*
  qed

  let $?D = \lambda x.\ \mathit{distr}\ (K.T\ x)\ K.S\ (\mathit{smap}\ f)$
  have $\mathit{prob\text{-}space\text{-}D}$: $?D\ x \in \mathit{space}\ (\mathit{prob\text{-}algebra}\ K.S)$ **for** x
    by $(\mathit{auto}\ \mathit{simp}: \mathit{space\text{-}prob\text{-}algebra}\ K.T.\mathit{prob\text{-}space\text{-}distr})$

  have $\mathit{D\text{-}eq\text{-}D}$: $?D\ x = ?D\ x'$ **if** $R\ x\ y\ R\ x'\ y$ **for** $x\ x'\ y$
  proof $(\mathit{rule}\ \mathit{stream\text{-}space\text{-}eq\text{-}sstart})$
    define A **where** $A = K.\mathit{acc}\ ``\ \{x,\ x'\}$
    have $\mathit{x\text{-}A}$: $x \in A\ x' \in A$ **by** $(\mathit{auto}\ \mathit{simp}: \mathit{A\text{-}def})$
    let $?\Omega = f\ `\ A$
    show $\mathit{countable}\ ?\Omega$
      unfolding *A-def* **by** $(\mathit{intro}\ \mathit{countable\text{-}image}\ K.\mathit{countable\text{-}acc})$ *auto*
    show $\mathit{prob\text{-}space}\ (?D\ x)\ \mathit{prob\text{-}space}\ (?D\ x')$ **by** $(\mathit{auto}\ \mathit{intro}!: K.T.\mathit{prob\text{-}space\text{-}distr})$
    show $\mathit{sets}\ (?D\ x) = \mathit{sets}\ L.S\ \mathit{sets}\ (?D\ x') = \mathit{sets}\ L.S$ **by** *auto*
    have $\mathit{AE\text{-}streams}$: $\mathit{AE}\ x\ \mathit{in}\ ?D\ x''.\ x \in \mathit{streams}\ ?\Omega$ **if** $x'' \in A$ **for** x''

```
    apply (simp add: space-stream-space streams-sets AE-distr-iff)
    using K.AE-T-reachable[of x''] unfolding alw-HLD-iff-streams
  proof eventually-elim
    fix s assume s ∈ streams (K.acc `` {x''})
    moreover have K.acc `` {x''} ⊆ A
      using ‹x'' ∈ A› by (auto simp: A-def Image-def intro: rtrancl-trans)
    ultimately show smap f s ∈ streams (f ` A)
      by (auto intro: smap-streams)
  qed
  with x-A show AE x in ?D x'. x ∈ streams ?Ω AE x in ?D x. x ∈ streams ?Ω
    by auto
  from ‹x ∈ A› ‹x' ∈ A› that show ?D x (sstart (f ` A) xs) = ?D x' (sstart (f ` A) xs) for xs
  proof (induction xs arbitrary: x x' y)
    case Nil
    moreover have ?D x (streams (f ` A)) = 1 if x ∈ A for x
      using AE-streams[of x] that
      by (intro prob-space.emeasure-eq-1-AE[OF K.T.prob-space-distr]) (auto simp: streams-sets)
    ultimately show ?case by simp
  next
    case (Cons z zs x x' y)
    have rel-pmf (R OO R⁻¹⁻¹) (K x) (K x')
      using KL[THEN rel-funD, OF Cons(4)] KL[THEN rel-funD, OF Cons(5)]
      unfolding pmf.rel-compp pmf.rel-flip by auto
    then obtain p :: ('s × 's) pmf where p: ⋀a b. (a, b) ∈ p ⟹ (R OO R⁻¹⁻¹) a b and
      eq: map-pmf fst p = K x map-pmf snd p = K x'
      by (auto simp: pmf.in-rel)
    let ?S = stream-space (count-space UNIV)
     have ∗: (##) y −` smap f −` sstart (f ` A) (z # zs) = (if f y = z then smap f −` sstart (f ` A) zs else
{}) for y z zs
      by auto
    have ∗∗: ?D x (sstart (f ` A) (z # zs)) = (∫⁺ y'. (if f y' = z then ?D y' (sstart (f ` A) zs) else 0) ∂K x)
for x
      apply (simp add: emeasure-distr)
      apply (subst K.T-eq-bind)
      apply (subst emeasure-bind[where N=?S])
        apply simp
       apply (rule measurable-distr2[where M=?S])
        apply measurable
      apply (intro nn-integral-cong-AE AE-pmfI)
      apply (auto simp add: emeasure-distr)
      apply (simp-all add: ∗ space-stream-space)
      done
    have fst-A: fst ab ∈ A if ab ∈ p for ab
    proof −
      have fst ab ∈ K x using ‹ab ∈ p› set-map-pmf [of fst p] by (auto simp: eq)
      with ‹x ∈ A› show fst ab ∈ A
        by (auto simp: A-def intro: rtrancl.rtrancl-into-rtrancl)
    qed
    have snd-A: snd ab ∈ A if ab ∈ p for ab
    proof −
      have snd ab ∈ K x' using ‹ab ∈ p› set-map-pmf [of snd p] by (auto simp: eq)
      with ‹x' ∈ A› show snd ab ∈ A
        by (auto simp: A-def intro: rtrancl.rtrancl-into-rtrancl)
    qed
    show ?case
      unfolding ∗∗ eq[symmetric] nn-integral-map-pmf
      apply (intro nn-integral-cong-AE AE-pmfI)
      subgoal for ab using p[of fst ab snd ab] by (auto simp: R-def intro!: Cons(1) fst-A snd-A)
      done
  qed
 qed
```

```
have L-eq-D: L.T y = ?D x
  using ‹R x y›
proof (coinduction arbitrary: x y rule: L.T-coinduct)
  case (cont x y)
  then have Kx-Ly: rel-pmf R (K x) (L y)
    by (rule KL[THEN rel-funD])
  then have ∗: y′ ∈ L y ⟹ ∃x′∈K x. R x′ y′ for y′
    by (auto dest!: rel-pmf-imp-rel-set simp: rel-set-def)
  have ∗∗: y′ ∈ L y ⟹ R (g y′) y′ for y′
    using ∗[of y′] unfolding g-def by (auto intro: someI)

  have D-SCons-eq-D-D: distr (K.T i) K.S (λx. z ## smap f x) = distr (?D i) K.S (λx. z ## x) for i z
    by (subst distr-distr) (auto simp: comp-def)
  have D-eq-D-gi: ?D i = ?D (g (f i)) if i: i ∈ K x for i
  proof −
    obtain j where j ∈ L y R i j f i = j
      using Kx-Ly i by (force dest!: rel-pmf-imp-rel-set simp: rel-set-def R-def)
    then show ?thesis
      by (auto intro!: D-eq-D[OF ‹R i j›] g)
  qed

  have ∗∗∗: ?D x = measure-pmf (L y) ≫= (λy. distr (?D (g y)) K.S ((##) y))
    apply (subst K.T-eq-bind)
    apply (subst distr-bind[of - - K.S])
       apply (rule measurable-distr2[of - - K.S])
        apply (simp-all add: Pi-iff)
    apply (simp add: distr-distr comp-def L-eq[OF cont] map-pmf-rep-eq)
    apply (subst bind-distr[where K=K.S])
       apply measurable []
      apply (rule measurable-distr2[of - - K.S])
      apply measurable []
      apply (rule measurable-compose[OF measurable-g])
      apply measurable []
     apply simp
    apply (rule bind-measure-pmf-cong[where N=K.S])
      apply (auto simp: space-subprob-algebra space-stream-space intro!: K.T.subprob-space-distr)
      unfolding D-SCons-eq-D-D D-eq-D-gi ..
  show ?case
    by (intro exI[of - λt. distr (K.T (g t)) (stream-space (count-space UNIV)) (smap f)])
       (auto simp add: K.T.prob-space-distr ∗∗∗ dest: ∗∗)
qed (auto intro: K.T.prob-space-distr)

have stream-all2 R s t ⟷ (s ∈ streams S ∧ smap f s = t) for s t
proof safe
  show stream-all2 R s t ⟹ s ∈ streams S
    apply (coinduction arbitrary: s t)
    subgoal for s t by (cases s; cases t) (auto simp: R-def)
    done
  show stream-all2 R s t ⟹ smap f s = t
    apply (coinduction arbitrary: s t rule: stream.coinduct)
    subgoal for s t by (cases s; cases t) (auto simp: R-def)
    done
qed (auto intro!: stream.rel-refl-strong simp: stream.rel-map R-def streams-iff-sset)
then have ω ∈ streams S ⟹ ω ∈ A ⟷ smap f ω ∈ B for ω
  using AB by (auto simp: rel-set-strong-def)
with in-S have K.T x A = K.T x (smap f −‘ B ∩ space (K.T x))
  by (auto intro!: emeasure-eq-AE streams-sets)
also have . . . = (distr (K.T x) K.S (smap f)) B
  by (intro emeasure-distr[symmetric]) auto
also have . . . = (L.T y) B unfolding L-eq-D ..
```

finally show *?thesis* **.**
qed

no-notation *ccval* (‹{|-|}› [*100*])

hide-const *succ*

2 Additional Facts on Regions

declare *reset-set11*[*simp*] *reset-set1*[*simp*]

Defining the closest successor of a region. Only exists if at least one interval is upper-bounded.

abbreviation *is-upper-right* **where**
is-upper-right $R \equiv (\forall\ t \geq 0.\ \forall\ u \in R.\ u \oplus t \in R)$

definition
succ $\mathcal{R}$ $R \equiv$
if is-upper-right R then R else
$(THE\ R'.\ R' \neq R \wedge R' \in Succ\ \mathcal{R}\ R \wedge (\forall\ u \in R.\ \forall\ t \geq 0.\ (u \oplus t) \notin R \longrightarrow (\exists\ t' \leq t.\ (u \oplus t') \in R' \wedge 0 \leq t')))$

lemma *region-continuous*:
assumes *valid-region X k I r*
defines *R*: $R \equiv region\ X\ I\ r$
assumes *between*: $0 \leq t1$ $t1 \leq t2$
assumes *elem*: $u \in R$ $u \oplus t2 \in R$
shows $u \oplus t1 \in R$
unfolding *R*
proof
from ‹$0 \leq t1$› ‹$u \in R$› **show** $\forall x{\in}X.\ 0 \leq (u \oplus t1)\ x$ **by** (*auto simp*: *R cval-add-def*)

have *intv-elem* $x\ (u \oplus t1)\ (I\ x)$ **if** $x \in X$ **for** *x*
proof −
from *elem that* **have** *intv-elem x u* $(I\ x)$ *intv-elem* $x\ (u \oplus t2)\ (I\ x)$ **by** (*auto simp*: *R*)
with *between* **show** *?thesis* **by** (*cases I x*, *auto simp*: *cval-add-def*)
qed
then show $\forall\ x \in X.$ *intv-elem* $x\ (u \oplus t1)\ (I\ x)$ **by** *blast*

let $?X_0 = \{x \in X.\ \exists\, d.\ I\ x = Intv\ d\}$
show $?X_0 = ?X_0$ **..**

from *elem* **have** $\forall\ x \in ?X_0.\ \forall\ y \in ?X_0.\ (x, y) \in r \longleftrightarrow frac\ (u\ x) \leq frac\ (u\ y)$ **by** (*auto simp*: *R*)
moreover
{ fix *x y c d* **assume** *A*: $x \in X$ $y \in X$ $I\ x = Intv\ c$ $I\ y = Intv\ d$
from *A elem between* **have** ∗:
$c < u\ x$ $u\ x < c + 1$ $c < u\ x + t1$ $u\ x + t1 < c + 1$
by (*fastforce simp*: *cval-add-def R*)+
moreover from *A*(*2*,*4*) *elem between* **have** ∗∗:
$d < u\ y$ $u\ y < d + 1$ $d < u\ y + t1$ $u\ y + t1 < d + 1$
by (*fastforce simp*: *cval-add-def R*)+
ultimately have $u\ x = c + frac\ (u\ x)$ $u\ y = d + frac\ (u\ y)$ **using** *nat-intv-frac-decomp* **by** *auto*
then have
$frac\ (u\ x + t1) = frac\ (u\ x) + t1$ $frac\ (u\ y + t1) = frac\ (u\ y) + t1$
using ∗(*3*,*4*) ∗∗(*3*,*4*) *nat-intv-frac-decomp* **by** *force*+
then have
$frac\ (u\ x) \leq frac\ (u\ y) \longleftrightarrow frac\ ((u \oplus t1)\ x) \leq frac\ ((u \oplus t1)\ y)$
by (*auto simp*: *cval-add-def*)
}
ultimately show

```
    ∀ x ∈ ?X₀. ∀ y ∈ ?X₀. (x, y) ∈ r ⟷ frac ((u ⊕ t1) x) ≤ frac ((u ⊕ t1) y)
  by (auto simp: cval-add-def)
qed

lemma upper-right-eq:
  assumes finite X valid-region X k I r
  shows (∀ x ∈ X. isGreater (I x)) ⟷ is-upper-right (region X I r)
using assms
proof (safe, goal-cases)
  case (1 t u)
  then show ?case
    by − (standard, force simp: cval-add-def)+
next
  case (2 x)

  from region-not-empty[OF assms] obtain u where u: u ∈ region X I r ..
  moreover have (1 :: real) ≥ 0 by auto
  ultimately have (u ⊕ 1) ∈ region X I r using 2 by auto
  with ‹x ∈ X› u have intv-elem x u (I x) intv-elem x (u ⊕ 1) (I x) by auto
  then show ?case by (cases I x, auto simp: cval-add-def)
qed

lemma bounded-region:
  assumes finite X valid-region X k I r
  defines R: R ≡ region X I r
  assumes ¬ is-upper-right R u ∈ R
  shows u ⊕ 1 ∉ R
proof −
  from upper-right-eq[OF assms(1,2)] assms(4) obtain x where x:
    x ∈ X ¬ isGreater (I x)
  by (auto simp: R)
  with assms have intv-elem x u (I x) by auto
  with x(2) have ¬ intv-elem x (u ⊕ 1) (I x) by (cases I x, auto simp: cval-add-def)
  with x(1) assms show ?thesis by auto
qed

context AlphaClosure-global
begin


no-notation Regions-Beta.part (‹[-]-› [61,61] 61)

lemma succ-ex:
  assumes R ∈ ℛ
  shows succ ℛ R ∈ ℛ (is ?G1) and succ ℛ R ∈ Succ ℛ R (is ?G2)
  and ∀ u ∈ R. ∀ t ≥ 0. (u ⊕ t) ∉ R ⟶ (∃ t' ≤ t. (u ⊕ t') ∈ succ ℛ R ∧ 0 ≤ t') (is ?G3)
proof −
  from ‹R ∈ ℛ› obtain I r where R: R = region X I r valid-region X k I r
    unfolding ℛ-def by auto
  from region-not-empty[OF finite] R obtain u where u: u ∈ R
    by blast
  let ?Z = {x ∈ X . ∃ c. I x = Const c}
  let ?succ =
    λ R'. R' ≠ R ∧ R' ∈ Succ ℛ R
     ∧ (∀ u ∈ R. ∀ t ≥ 0. (u ⊕ t) ∉ R ⟶ (∃ t' ≤ t. (u ⊕ t') ∈ R' ∧ 0 ≤ t'))
  consider (upper-right) ∀ x ∈ X. isGreater (I x) | (intv) ∃ x ∈ X. ∃ d. I x = Intv d ∧ ?Z = {}
         | (const) ?Z ≠ {}
  apply (cases ∀ x ∈ X. isGreater (I x))
   apply fast
  apply (cases ?Z = {})
   apply safe
```

apply (*rename-tac x*)
apply (*case-tac I x*)
by *auto*
then have $?G1 \wedge ?G2 \wedge ?G3$
proof *cases*
case *const*
with *upper-right-eq*[*OF finite R*(*2*)] **have** $\neg$ *is-upper-right R* **by** (*auto simp*: *R*(*1*))
from *closest-prestable-1*(*1*,*2*)[*OF const finite R*(*2*)] *closest-valid-1*[*OF const finite R*(*2*)] *R*(*1*)
obtain R' **where** R':
$\forall\ u \in R.\ \forall\ t>0.\ \exists t' \le t.\ (u \oplus t') \in R' \wedge t' \ge 0\ R' \in \mathcal{R}\ \forall\ u \in R'.\ \forall\ t \ge 0.\ (u \oplus t) \notin R$
unfolding $\mathcal{R}$-*def* **by** *auto*
with *region-not-empty*[*OF finite*] **obtain** u' **where** $u' \in R'$ **unfolding** $\mathcal{R}$-*def* **by** *blast*
with R'(*3*) **have** *neq*: $R' \neq R$ **by** (*fastforce simp*: *cval-add-def*)
obtain *t*:: *real* **where** $t > 0$ **by** (*auto intro*: *that*[*of 1*])
with R'(*1*,*2*) ‹$u \in R$› **obtain** *t* **where** $t \ge 0\ u \oplus t \in R'$ **by** *auto*
with ‹$R \in \mathcal{R}$› ‹$R' \in \mathcal{R}$› ‹$u \in R$› **have** $R' \in$ *Succ* $\mathcal{R}$ *R* **by** (*intro SuccI3*)
moreover have $(\forall\ u \in R.\ \forall\ t \ge 0.\ (u \oplus t) \notin R \longrightarrow (\exists\ t' \le t.\ (u \oplus t') \in R' \wedge 0 \le t'))$
using R'(*1*) **unfolding** *cval-add-def*
apply *clarsimp*
subgoal for *u t*
by (*cases* $t = 0$) *auto*
done
ultimately have $*$: *?succ* R' **using** *neq* **by** *auto*
have *succ* $\mathcal{R}$ $R = R'$ **unfolding** *succ-def*
proof (*simp add*: ‹$\neg$ *is-upper-right R*›, *intro the-equality*, *rule* $*$, *goal-cases*)
case *prems*: (*1* R'')
from *prems* **obtain** t' u' **where** R'':
$R'' \in \mathcal{R}\ R'' \neq R\ t' \ge 0\ R'' = [u' \oplus t']_{\mathcal{R}}\ u' \in R$
using R'(*1*) **by** *fastforce*
from *this*(*1*) **obtain** *I r* **where** $R''2$:
R'' = *region X I r valid-region X k I r*
by (*auto simp*: $\mathcal{R}$-*def*)
from R'' **have** $u' \oplus t' \notin R$ **using** *assms region-unique-spec* **by** *blast*
with $*$ ‹$t' \ge 0$› ‹$u' \in R$› **obtain** t'' **where** t'': $t'' \le t'\ u' \oplus t'' \in R'\ t'' \ge 0$ **by** *auto*
from *this*(*2*) *neq* **have** $u' \oplus t'' \notin R$ **using** R'(*2*) *assms region-unique-spec* **by** *auto*
with t'' *prems* ‹$u' \in R$› **obtain** t''' **where** t''':
$t''' \le t''\ u' \oplus t''' \in R''\ t''' \ge 0$
by *auto*
with *region-continuous*[*OF* $R''2$(*2*) - - t'''(*2*)[*unfolded* $R''2$(*1*)], *of* $t'' - t'''\ t' - t'''$]
t'' R'' *regions-closed'-spec*[*OF* ‹$R \in \mathcal{R}$› R''(*5*,*3*)]
have $u' \oplus t'' \in R''$ **by** (*auto simp*: $R''2$ *cval-add-def*)
with t''(*2*) **show** *?case* **using** R''(*1*) R'(*2*) *region-unique-spec* **by** *blast*
qed
with R' $*$ **show** *?thesis* **by** *auto*
next
case *intv*
then have $*$: $\forall x \in X.\ \neg$ *Regions.isConst* (*I x*) **by** *auto*
let $?X_0 = \{x \in X.\ isIntv\ (I\ x)\}$
let $?M = \{x \in ?X_0.\ \forall y \in ?X_0.\ (x, y) \in r \longrightarrow (y, x) \in r\}$
from *intv* **obtain** *x c* **where** *x*: $x \in X\ \neg$ *isGreater* (*I x*) **and** *c*: $I\ x = Intv\ c$ **by** *auto*
with ‹$x \in X$› **have** $?X_0 \neq \{\}$ **by** *auto*
have $?X_0 = \{x \in X.\ \exists d.\ I\ x = Intv\ d\}$ **by** *auto*
with *R*(*2*) **have** *r*: *total-on* $?X_0$ *r trans r* **by** *auto*
from *total-finite-trans-max*[*OF* ‹$?X_0 \neq \{\}$› - *this*] *finite*
obtain x' **where** x': $x' \in ?X_0\ \forall\ y \in ?X_0.\ x' \neq y \longrightarrow (y, x') \in r$ **by** *fastforce*
from *this*(*2*) **have** $\forall y \in ?X_0.\ (x', y) \in r \longrightarrow (y, x') \in r$ **by** *auto*
with x'(*1*) **have** $**$: $?M \neq \{\}$ **by** *fastforce*
with *upper-right-eq*[*OF finite R*(*2*)] **have** $\neg$ *is-upper-right R* **by** (*auto simp*: *R*(*1*))
from *closest-prestable-2*(*1*,*2*)[*OF* $*$ *finite R*(*2*) $**$] *closest-valid-2*[*OF* $*$ *finite R*(*2*) $**$] *R*(*1*)
obtain R' **where** R':
$(\forall\ u \in R.\ \forall\ t \ge 0.\ (u \oplus t) \notin R \longrightarrow (\exists\ t' \le t.\ (u \oplus t') \in R' \wedge 0 \le t'))\ R' \in \mathcal{R}$

$\forall\ u \in R'.\ \forall\ t \geq 0.\ (u \oplus t) \notin R$
unfolding $\mathcal{R}$*-def* **by** *auto*
with *region-not-empty*[*OF finite*] **obtain** u' **where** $u' \in R'$ **unfolding** $\mathcal{R}$*-def* **by** *blast*
with $R'(3)$ **have** *neq*: $R' \neq R$ **by** (*fastforce simp*: *cval-add-def*)
from *bounded-region*[*OF finite* $R(2)$, *folded* $R(1)$, *OF* ‹¬ *is-upper-right* R› u] **have**
$u \oplus (1 :: t) \notin R\ (1 :: t) \geq 0$
by *auto*
with $R'(1)$ u **obtain** t' **where** $t' \leq (1 :: t)\ (u \oplus t') \in R'\ 0 \leq t'$ **by** *fastforce*
with ‹$R \in \mathcal{R}$› ‹$R' \in \mathcal{R}$› ‹$u \in R$› **have** $R' \in Succ\ \mathcal{R}\ R$ **by** (*intro SuccI3*)
with $R'(1)$ *neq* **have** $*$: *?succ* R' **by** *auto*
have $succ\ \mathcal{R}\ R = R'$ **unfolding** *succ-def*
proof (*simp add*: ‹¬ *is-upper-right* R›, *intro the-equality*, *rule* $*$, *goal-cases*)
case *prems*: $(1\ R'')$
from *prems* **obtain** $t'\ u'$ **where** R'':
$R'' \in \mathcal{R}\ R'' \neq R\ t' \geq 0\ R'' = [u' \oplus t']_{\mathcal{R}}\ u' \in R$
using $R'(1)$ **by** *fastforce*
from *this*(1) **obtain** $I\ r$ **where** $R''2$:
$R'' = region\ X\ I\ r\ valid\text{-}region\ X\ k\ I\ r$
by (*auto simp*: $\mathcal{R}$*-def*)
from R'' **have** $u' \oplus t' \notin R$ **using** *assms region-unique-spec* **by** *blast*
with $*$ ‹$t' \geq 0$› ‹$u' \in R$› **obtain** t'' **where** t'': $t'' \leq t'\ u' \oplus t'' \in R'\ t'' \geq 0$ **by** *auto*
from *this*(2) *neq* **have** $u' \oplus t'' \notin R$ **using** $R'(2)$ *assms region-unique-spec* **by** *auto*
with t'' *prems* ‹$u' \in R$› **obtain** t''' **where** t''':
$t''' \leq t''\ u' \oplus t''' \in R''\ t''' \geq 0$
by *auto*
with *region-continuous*[*OF* $R''2(2)$ - - $t'''(2)$[*unfolded* $R''2(1)$], *of* $t'' - t'''\ t' - t'''$]
$t''\ R''$ *regions-closed'-spec*[*OF* ‹$R \in \mathcal{R}$› $R''(5,3)$]
have $u' \oplus t'' \in R''$ **by** (*auto simp*: *cval-add-def* $R''2$)
with $t''(2)$ **show** *?case* **using** $R''(1)$ $R'(2)$ *region-unique-spec* **by** *blast*
qed
with R' $*$ **show** *?thesis* **by** *auto*
next
case *upper-right*
with *upper-right-eq*[*OF finite* $R(2)$] **have** $succ\ \mathcal{R}\ R = R$ **by** (*auto simp*: R *succ-def*)
with ‹$R \in \mathcal{R}$› u **show** *?thesis* **by** (*fastforce simp*: *cval-add-def intro*: *SuccI3*)
qed
then show *?G1 ?G2 ?G3* **by** *auto*
qed

lemma *region-set'-closed*:
fixes $d :: nat$
assumes $R \in \mathcal{R}\ d \geq 0\ \forall x \in set\ r.\ d \leq k\ x\ set\ r \subseteq X$
shows $region\text{-}set'\ R\ r\ d \in \mathcal{R}$
proof −
from *region-not-empty*[*OF finite*] *assms*(1) **obtain** u **where** $u \in R$ **using** $\mathcal{R}$*-def* **by** *blast*
from *region-set'-id*[*OF* - - *finite*, *of* - k, *folded* $\mathcal{R}$*-def*] *assms this* **show** *?thesis* **by** *fastforce*
qed

lemma *clock-set-cong*[*simp*]:
assumes $\forall\ c \in set\ r.\ u\ c = d$
shows $[r \to d]u = u$
proof *standard*
fix c
from *assms* **show** $([r \to d]u)\ c = u\ c$ **by** (*cases* $c \in set\ r$; *auto*)
qed

lemma *region-reset-not-Succ*:

notes *regions-closed'-spec*[*intro*]
assumes $R \in \mathcal{R}\ set\ r \subseteq X$

shows *region-set′ R r 0 = R ∨ region-set′ R r 0 ∉ Succ* $\mathcal{R}$ *R* (**is** *?R = R ∨ -*)
proof −
  from *assms finite* **obtain** *u* **where** *u ∈ R* **by** (*meson Succ.cases succ-ex*(*2*))
  with ‹*R ∈* $\mathcal{R}$› **have** *u ∈ V* $[u]_{\mathcal{R}}$ *= R* **by** (*auto simp*: *region-unique-spec dest*: *region-V*)
  with *region-set′-id*[*OF* ‹*R ∈* $\mathcal{R}$›[*unfolded* $\mathcal{R}$*-def*] ‹*u ∈ R*› *finite*] *assms*(*2*) **have**
    ?R = $[[r \to 0]u]_{\mathcal{R}}$
  by (*force simp*: $\mathcal{R}$*-def*)
  show *?thesis*
  proof (*cases* ∀ *x ∈ set r. u x = 0*)
    case *True*
    then have $[r \to 0]u = u$ **by** *simp*
    with ‹*?R = -*› ‹*- = R*› **have** *?R = R* **by** (*force simp*: $\mathcal{R}$*-def*)
    then show *?thesis* **..**
  next
    case *False*
    then obtain *x* **where** *x*: *x ∈ set r u x ≠ 0* **by** *auto*
    { assume *?R ∈ Succ* $\mathcal{R}$ *R*
      with ‹*u ∈ R*› ‹*R ∈* $\mathcal{R}$› **obtain** *t* **where**
        t ≥ 0 $[u \oplus t]_{\mathcal{R}}$ *= ?R ?R ∈* $\mathcal{R}$
      by (*meson Succ.cases set-of-regions-spec*)
      with ‹*u ∈ R*› *assms*(*1*) **have** *u ⊕ t ∈ ?R* **by** *blast*
      moreover from ‹*?R = -*› ‹*u ∈ R*› **have** $[r \to 0]u \in$ *?R* **by** (*fastforce simp*: *region-set′-def*)
      moreover from *x* ‹*t ≥ 0*› ‹*u ∈ V*› *assms* **have** (*u ⊕ t*) *x > 0* **by** (*force simp*: *cval-add-def V-def*)
      moreover from *x* **have** ($[r \to 0]u$) *x = 0* **by** *auto*
      ultimately have *False* **using** ‹*?R ∈* $\mathcal{R}$› *x*(*1*) **by** (*fastforce simp*: *region-set′-def*)
    }
    then show *?thesis* **by** *auto*
  qed
qed

end

3 Definition and Semantics

3.1 Syntactic Definition

We do not include:

- a labelling function, as we will assume that atomic propositions are simply sets of states
- a fixed set of locations or clocks, as we will implicitly derive it from the set of transitions
- start or end locations, as we will primarily study reachability

type-synonym
  (*′c, ′t, ′s*) *transition = ′s ∗* (*′c, ′t*) *cconstraint ∗* (*′c set ∗ ′s*) *pmf*

type-synonym
  (*′c, ′t, ′s*) *pta* = (*′c, ′t, ′s*) *transition set ∗* (*′c, ′t, ′s*) *invassn*

definition
  edges :: (*′c, ′t, ′s*) *transition ⇒* (*′s ∗* (*′c, ′t*) *cconstraint ∗* (*′c set ∗ ′s*) *pmf ∗ ′c set ∗ ′s*) *set*
where
  edges ≡ λ (*l, g, p*). {(*l, g, p, X, l′*) | *X l′*. (*X, l′*) ∈ *set-pmf p*}

definition
  Edges A ≡ ⋃ {*edges t* | *t. t ∈ fst A*}

definition
  trans-of :: (*′c, ′t, ′s*) *pta ⇒* (*′c, ′t, ′s*) *transition set*

where
  trans-of ≡ *fst*

definition
  inv-of :: ('c, 'time, 's) *pta* ⇒ ('c, 'time, 's) *invassn*
where
  inv-of ≡ *snd*

no-notation *transition* (‹- ⊢ - ⟶$^{-,-,-}$ -› [61,61,61,61,61,61] 61)

abbreviation *transition* ::
  ('c, 'time, 's) *pta* ⇒ 's ⇒ ('c, 'time) *cconstraint* ⇒ ('c *set* ∗ 's) *pmf* ⇒ 'c *set* ⇒ 's ⇒ *bool*
(‹- ⊢ - ⟶$^{-,-,-}$ -› [61,61,61,61,61,61] 61) **where**
  (A ⊢ l ⟶^{g,p,X} l') ≡ (l, g, p, X, l') ∈ *Edges* A

definition
  locations :: ('c, 't, 's) *pta* ⇒ 's *set*
where
  locations A ≡ (*fst* ‘ *Edges* A) ∪ ((*snd o snd o snd o snd*) ‘ *Edges* A)

3.1.1 Collecting Information About Clocks

definition *collect-clkt* :: ('c, 't::*time*, 's) *transition set* ⇒ ('c ∗'t) *set*
where
  collect-clkt S = ⋃ {*collect-clock-pairs* (*fst* (*snd* t)) | t . t ∈ S}

definition *collect-clki* :: ('c, 't :: *time*, 's) *invassn* ⇒ ('c ∗'t) *set*
where
  collect-clki I = ⋃ {*collect-clock-pairs* (I x) | x. *True*}

definition *clkp-set* :: ('c, 't :: *time*, 's) *pta* ⇒ ('c ∗ 't) *set*
where
  clkp-set A = *collect-clki* (*inv-of* A) ∪ *collect-clkt* (*trans-of* A)

definition *collect-clkvt* :: ('c, 't :: *time*, 's) *pta* ⇒ 'c *set*
where
  collect-clkvt A = ⋃ ((*fst o snd o snd o snd*) ‘ *Edges* A)

abbreviation *clocks* **where** *clocks* A ≡ *fst* ‘ *clkp-set* A ∪ *collect-clkvt* A

definition *valid-abstraction*
where
  valid-abstraction A X k ≡
  (∀ (x,m) ∈ *clkp-set* A. m ≤ k x ∧ x ∈ X ∧ m ∈ ℕ) ∧ *collect-clkvt* A ⊆ X ∧ *finite* X

lemma *valid-abstractionD*[*dest*]:
  assumes *valid-abstraction* A X k
  shows (∀ (x,m) ∈ *clkp-set* A. m ≤ k x ∧ x ∈ X ∧ m ∈ ℕ) *collect-clkvt* A ⊆ X *finite* X
using *assms* **unfolding** *valid-abstraction-def* **by** *auto*

lemma *valid-abstractionI*[*intro*]:
  assumes (∀ (x,m) ∈ *clkp-set* A. m ≤ k x ∧ x ∈ X ∧ m ∈ ℕ) *collect-clkvt* A ⊆ X *finite* X
  shows *valid-abstraction* A X k
using *assms* **unfolding** *valid-abstraction-def* **by** *auto*

3.2 Operational Semantics as an MDP

abbreviation (*input*) *clock-set-set* :: 'c *set* ⇒ 't::*time* ⇒ ('c,'t) *cval* ⇒ ('c,'t) *cval*
(‹[-:=-]-› [65,65,65] 65)
where
  [X:=t]u ≡ *clock-set* (*SOME* r. *set* r = X) t u

term *region-set′*

abbreviation *region-set-set* :: $'c$ *set* ⇒ $'t$::*time* ⇒ $('c,'t)$ *zone* ⇒ $('c,'t)$ *zone*
(‹[-::=-]-› [*65*,*65*,*65*] *65*)
where
  $[X::=t]R \equiv$ *region-set′* R (*SOME* r. *set* $r = X$) t

no-notation *zone-set* (‹$\text{-}_{\text{-} \to 0}$› [*71*] *71*)

abbreviation *zone-set-set* :: $('c, 't$::*time*) *zone* ⇒ $'c$ *set* ⇒ $('c, 't)$ *zone*
(‹$\text{-}_{\text{-} \to 0}$› [*71*] *71*)
where
  $Z_{X \to 0} \equiv$ *zone-set* Z (*SOME* r. *set* $r = X$)

abbreviation (*input*) *ccval* (‹{|-|}› [*100*]) **where** *ccval cc* ≡ $\{v.\ v \vdash cc\}$

locale *Probabilistic-Timed-Automaton* =
  fixes A :: $('c, 't$ :: *time*, $'s)$ *pta*
  assumes *admissible-targets*:
    $(l, g, \mu) \in$ *trans-of* $A \Longrightarrow (X, l') \in \mu \Longrightarrow \{|g|\}_{X \to 0} \subseteq \{|$*inv-of* $A\ l'|\}$
    $(l, g, \mu) \in$ *trans-of* $A \Longrightarrow (X, l') \in \mu \Longrightarrow X \subseteq$ *clocks* A
  — Not necessarily what we want to have
begin

3.3 Syntactic Definition

definition L = *locations* A

definition $\mathcal{X}$ = *clocks* A

definition $S \equiv \{(l, u)\ .\ l \in L \wedge (\forall\ x \in \mathcal{X}.\ u\ x \geq 0) \wedge u \vdash$ *inv-of* $A\ l\}$

inductive-set
  K :: $('s * ('c, 't)$ *cval*) ⇒ $('s * ('c, 't)$ *cval*) *pmf set* **for** *st* :: $('s * ('c, 't)$ *cval*)
where
  — Passage of time *delay*:
  $st \in S \Longrightarrow st = (l, u) \Longrightarrow t \geq 0 \Longrightarrow u \oplus t \vdash$ *inv-of* $A\ l \Longrightarrow$ *return-pmf* $(l, u \oplus t) \in K\ st$ |
  — Discrete transitions *action*:
  $st \in S \Longrightarrow st = (l, u) \Longrightarrow (l, g, \mu) \in$ *trans-of* $A \Longrightarrow u \vdash g$
  $\Longrightarrow$ *map-pmf* $(\lambda\ (X, l).\ (l, ([X := 0]u)))\ \mu \in K\ st$ |
  — Self loops – Note that this does not assume $st \in S$ *loop*:
  return-pmf $st \in K\ st$

declare *K.intros*[*intro*]

sublocale *MDP*: *Markov-Decision-Process* K **by** (*standard*, *auto*)

end

4 Constructing the Corresponding Finite MDP on Regions

locale *Probabilistic-Timed-Automaton-Regions* =
  Probabilistic-Timed-Automaton A + *Regions-global* $\mathcal{X}$
  for A :: $('c, t, 's)$ *pta* +
  — The following are necessary to obtain a *finite* MDP
  assumes *finite*: *finite* $\mathcal{X}$ *finite* L *finite* (*trans-of* A)
  assumes *not-trivial*: $\exists\ l \in L.\ \exists\ u \in V.\ u \vdash$ *inv-of* $A\ l$
  assumes *valid*: *valid-abstraction* $A\ \mathcal{X}\ k$
begin

lemmas $finite\text{-}\mathcal{R} = finite\text{-}\mathcal{R}[OF\ finite(1),\ of\ k,\ folded\ \mathcal{R}\text{-}def]$

4.1 Syntactic Definition

definition $\mathcal{S} \equiv \{(l,\ R)\ .\ l \in L \wedge R \in \mathcal{R} \wedge R \subseteq \{u.\ u \vdash inv\text{-}of\ A\ l\}\}$

lemma $S\text{-}alt\text{-}def$: $S = \{(l,\ u)\ .\ l \in L \wedge u \in V \wedge u \vdash inv\text{-}of\ A\ l\}$ **unfolding** *V-def S-def* **by** *auto*

Note how we relax the definition to allow more transitions in the first case. To obtain a more compact MDP the commented out version can be used an proved equivalent.

inductive-set
$\mathcal{K} :: ('s * ('c,\ t)\ cval\ set) \Rightarrow ('s * ('c,\ t)\ cval\ set)\ pmf\ set$ **for** $st :: ('s * ('c,\ t)\ cval\ set)$
where

— Passage of time *delay*:
$st \in \mathcal{S} \Longrightarrow st = (l,R) \Longrightarrow R' \in Succ\ \mathcal{R}\ R \Longrightarrow R' \subseteq \{\!|inv\text{-}of\ A\ l|\!\} \Longrightarrow return\text{-}pmf\ (l,\ R') \in \mathcal{K}\ st\ |$
— Discrete transitions *action*:
$st \in \mathcal{S} \Longrightarrow st = (l,\ R\ ) \Longrightarrow (l,\ g,\ \mu) \in trans\text{-}of\ A \Longrightarrow R \subseteq \{\!|g|\!\}$
$\Longrightarrow map\text{-}pmf\ (\lambda\ (X,\ l).\ (l,\ region\text{-}set'\ R\ (SOME\ r.\ set\ r = X)\ 0))\ \mu \in \mathcal{K}\ st\ |$
— Self loops – Note that this does not assume $st \in \mathcal{S}$ *loop*:
$return\text{-}pmf\ st \in \mathcal{K}\ st$

lemmas $[intro] = \mathcal{K}.intros$

4.2 Many Closure Properties

lemma *transition-def*:
$(A \vdash l \longrightarrow^{g,\mu,X} l') = ((l,\ g,\ \mu) \in trans\text{-}of\ A \wedge (X,\ l') \in \mu)$
unfolding *Edges-def edges-def trans-of-def* **by** *auto*

lemma *transitionI*[*intro*]:
$A \vdash l \longrightarrow^{g,\mu,X} l'$ **if** $(l,\ g,\ \mu) \in trans\text{-}of\ A\ (X,\ l') \in \mu$
using *that* **unfolding** *transition-def* **..**

lemma *transitionD*[*dest*]:
$(l,\ g,\ \mu) \in trans\text{-}of\ A\ (X,\ l') \in \mu$ **if** $A \vdash l \longrightarrow^{g,\mu,X} l'$
using *that* **unfolding** *transition-def* **by** *auto*

lemma *bex-Edges*:
$(\exists\ x \in Edges\ A.\ P\ x) = (\exists\ l\ g\ \mu\ X\ l'.\ A \vdash l \longrightarrow^{g,\mu,X} l' \wedge P\ (l,\ g,\ \mu,\ X,\ l'))$
by *fastforce*

lemma *L-trans*[*intro*]:
assumes $(l,\ g,\ \mu) \in trans\text{-}of\ A\ (X,\ l') \in \mu$
shows $l \in L\ l' \in L$
using *assms* **unfolding** *L-def locations-def* **by** (*auto simp*: *image-iff bex-Edges transition-def*)

lemma *transition-$\mathcal{X}$*:
$X \subseteq \mathcal{X}$ **if** $A \vdash l \longrightarrow^{g,\mu,X} l'$
using *that* **unfolding** *$\mathcal{X}$-def collect-clkvt-def clkp-set-def* **by** *auto*

lemma *admissible-targets-alt*:
$A \vdash l \longrightarrow^{g,\mu,X} l' \Longrightarrow \{\!|g|\!\}_{X \to 0} \subseteq \{\!|inv\text{-}of\ A\ l'|\!\}$
$A \vdash l \longrightarrow^{g,\mu,X} l' \Longrightarrow X \subseteq clocks\ A$
by (*intro admissible-targets*; *blast*)+

lemma *V-reset-closed*[*intro*]:

assumes $u \in V$
shows $[r \to (d::nat)]u \in V$
using *assms* **unfolding** *V-def*
apply *safe*
subgoal for x
by (*cases* $x \in$ *set* r; *auto*)
done

lemmas *V-reset-closed'*[*intro*] = *V-reset-closed*[*of - - 0*, *simplified*]

lemma *regions-part-ex*[*intro*]:
assumes $u \in V$
shows $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$
proof −
from *assms regions-partition*[*OF meta-eq-to-obj-eq*[*OF* $\mathcal{R}$*-def*]] **have**
$\exists!R.\ R \in \mathcal{R} \land u \in R$
unfolding *V-def* **by** *auto*
then show $[u]_{\mathcal{R}} \in \mathcal{R}$ $u \in [u]_{\mathcal{R}}$
using *alpha-interp.region-unique-spec* **by** *auto*
qed

lemma *rep-*$\mathcal{R}$*-ex*[*intro*]:
assumes $R \in \mathcal{R}$
shows (*SOME* $u.\ u \in R$) $\in R$
proof −
from *assms region-not-empty*[*OF finite*(*1*)] **have** $\exists\ u.\ u \in R$ **unfolding** $\mathcal{R}$*-def* **by** *auto*
then show *?thesis* **..**
qed

lemma *V-nn-closed*[*intro*]:
$u \in V \Longrightarrow t \geq 0 \Longrightarrow u \oplus t \in V$
unfolding *V-def cval-add-def* **by** *auto*

lemma *K-S-closed*[*intro*]:
assumes $\mu \in K\ s$ $s' \in \mu$ $s \in S$
shows $s' \in S$
using *assms*
by (*cases rule*: *K.cases*, *auto simp*: *S-alt-def dest*: *admissible-targets*[*unfolded zone-set-def*])

lemma *S-V*[*intro*]:
$(l, u) \in S \Longrightarrow u \in V$
unfolding *S-alt-def* **by** *auto*

lemma *L-V*[*intro*]:
$(l, u) \in S \Longrightarrow l \in L$
unfolding *S-def* **by** *auto*

lemma $\mathcal{S}$*-V*[*intro*]:
$(l, R) \in \mathcal{S} \Longrightarrow R \in \mathcal{R}$
unfolding $\mathcal{S}$*-def* **by** *auto*

lemma *admissible-targets'*:
assumes $(l, g, \mu) \in$ *trans-of* A $(X, l') \in \mu$ $R \subseteq \{\!|g|\!\}$
shows *region-set'* R (*SOME* $r.$ *set* $r = X$) $0 \subseteq \{\!|$*inv-of* $A\ l'|\!\}$
using *admissible-targets*(*1*)[*OF assms*(*1*,*2*)] *assms*(*3*) **unfolding** *region-set'-def zone-set-def* **by** *auto*

4.3 The Region Graph is a Finite MDP

lemma $\mathcal{S}$*-finite*:
finite $\mathcal{S}$
using *finite finite-*$\mathcal{R}$ **unfolding** $\mathcal{S}$*-def* **by** *auto*

lemma $\mathcal{K}$-*finite*:
finite ($\mathcal{K}$ *st*)
proof −
let $?B1 = \{(R', l, R).\ st \in \mathcal{S} \wedge st = (l, R) \wedge R' \in Succ\ \mathcal{R}\ R \wedge R' \subseteq \{\!|inv\text{-}of\ A\ l|\!\}\}$
let $?S1 = (\lambda(R', l, R).\ return\text{-}pmf\ (l, R'))$ ‘ $?B1$
let $?S1 = \{return\text{-}pmf\ (l, R') \mid R'\ l\ R.\ st \in \mathcal{S} \wedge st = (l, R) \wedge R' \in Succ\ \mathcal{R}\ R \wedge R' \subseteq \{\!|inv\text{-}of\ A\ l|\!\}\}$
let $?S2 = \{map\text{-}pmf\ (\lambda\ (X, l).\ (l, region\text{-}set'\ R\ (SOME\ r.\ set\ r = X)\ 0))\ \mu$
$\mid R\ \mu.\ \exists\ l\ g.\ st \in \mathcal{S} \wedge st = (l, R\ ) \wedge (l, g, \mu) \in trans\text{-}of\ A \wedge R \subseteq \{\!|g|\!\}\}$
have $?B1 \subseteq \{(R', l, R).\ R' \in \mathcal{R} \wedge (l, R) \in \mathcal{S}\ \}$ **unfolding** $\mathcal{S}$-*def* **by** *auto*
with $\mathcal{S}$-*finite finite*-$\mathcal{R}$ **have** *finite ?B1* **by** − (*rule finite-subset*, *auto*)
moreover have $?S1 = (\lambda(R', l, R).\ return\text{-}pmf\ (l, R'))$ ‘ $?B1$ **by** (*auto simp*: *image-def*)
ultimately have ∗: *finite ?S1* **by** *auto*
have $\{\mu.\ \exists\, l\ g.\ (l, g, \mu) \in PTA.trans\text{-}of\ A\} = ((\lambda\ (l, g, \mu).\ \mu)$ ‘ $PTA.trans\text{-}of\ A)$ **by** *force*
with *finite*(*3*) *finite*-$\mathcal{R}$ **have** *finite* $\{(R, \mu).\ \exists\ l\ g.\ R \in \mathcal{R} \wedge (l, g, \mu) \in trans\text{-}of\ A\}$ **by** *auto*
moreover have
$\{(R, \mu).\ \exists\ l\ g.\ st \in \mathcal{S} \wedge st = (l, R\ ) \wedge (l, g, \mu) \in trans\text{-}of\ A \wedge R \subseteq \{\!|g|\!\}\} \subseteq \ldots$
unfolding $\mathcal{S}$-*def* **by** *fastforce*
ultimately have ∗∗:
finite $\{(R, \mu).\ \exists\ l\ g.\ st \in \mathcal{S} \wedge st = (l, R\ ) \wedge (l, g, \mu) \in trans\text{-}of\ A \wedge R \subseteq \{\!|g|\!\}\}$
unfolding $\mathcal{S}$-*def* **by** (*blast intro*: *finite-subset*)
then have *finite ?S2* **unfolding** $\mathcal{S}$-*def* **by** *auto*
have $\mathcal{K}\ st = ?S1 \cup ?S2 \cup \{return\text{-}pmf\ st\}$ **by** (*safe*, *cases rule*: $\mathcal{K}$.*cases*, *auto*)
with ∗ ∗∗ **show** *?thesis* **by** *auto*
qed

lemma $\mathcal{R}$-*not-empty*:
$\mathcal{R} \neq \{\}$
proof −
let $?r = \{\}$
let $?I = \lambda\ c.\ Const\ 0$
let $?R = region\ \mathcal{X}\ \ ?I\ ?r$
have *valid-region* $\mathcal{X}$ *k ?I ?r*
proof
show $\{\} = \{x \in \mathcal{X}.\ \exists\, d.\ Const\ 0 = Intv\ d\}$ **by** *auto*
show *refl-on* {} {} **and** *trans* {} **and** *total-on* {} {} **unfolding** *trans-def* **by** *auto*
show $\forall\, x \in \mathcal{X}.\ Regions.valid\text{-}intv\ (k\ x)\ (Const\ 0)$ **by** *auto*
qed
then have $?R \in \mathcal{R}$ **unfolding** $\mathcal{R}$-*def* **by** *auto*
then show $\mathcal{R} \neq \{\}$ **by** *blast*
qed

lemma $\mathcal{S}$-*not-empty*:
$\mathcal{S} \neq \{\}$
proof −
from *not-trivial* **obtain** *l u* **where** *st*: $l \in L\ u \in V\ u \vdash inv\text{-}of\ A\ l$ **by** *blast*
then obtain *R* **where** *R*: $R \in \mathcal{R}\ u \in R$ **using** $\mathcal{R}$-*V* **by** *auto*
from *valid* **have**
$\forall\,(x, m)\in collect\text{-}clock\text{-}pairs\ (inv\text{-}of\ A\ l).\ m \leq real\ (k\ x) \wedge x \in \mathcal{X} \wedge m \in \mathbb{N}$
by (*fastforce simp*: *clkp-set-def collect-clki-def*)
from *ccompatible*[*OF this*, *folded* $\mathcal{R}$-*def*] *R st*(*3*) **have**
$R \subseteq \{\!|inv\text{-}of\ A\ l|\!\}$
unfolding *ccompatible-def ccval-def* **by** *auto*
with *st*(*1*) *R*(*1*) **show** *?thesis* **unfolding** $\mathcal{S}$-*def* **by** *auto*
qed

lemma $\mathcal{K}$-$\mathcal{S}$-*closed*:
assumes $s \in \mathcal{S}$
shows $(\bigcup D{\in}\mathcal{K}\ s.\ set\text{-}pmf\ D) \subseteq \mathcal{S}$
proof (*safe*, *cases rule*: $\mathcal{K}$.*cases*, *blast*, *goal-cases*)

case (*1 x a b l R*)
then show *?case* **unfolding** *𝒮-def* **by** (*auto intro*: *alpha-interp.succ-ex*(*1*))
next
case (*3 a b x*)
with ‹$s \in \mathcal{S}$› **show** *?case* **by** *auto*
next
case *prems*: (*2* l' R' *p l R g* μ)
then obtain *X* **where** ∗: $(X, l') \in$ *set-pmf* μ $R' =$ *region-set'* *R* (*SOME r. set r = X*) *0* **by** *auto*

show *?case* **unfolding** *𝒮-def*
proof *safe*
from ∗(*1*) **have** $(l, g, \mu, X, l') \in$ *edges* (l,g, μ) **unfolding** *edges-def* **by** *auto*
with *prems*(*6*) **have** $(l, g, \mu, X, l') \in$ *Edges A* **unfolding** *Edges-def trans-of-def* **by** *auto*
then show $l' \in L$ **unfolding** *L-def locations-def* **by** *force*

show $u \vdash$ *inv-of A* l' **if** $u \in R'$ **for** *u*
using *admissible-targets'*[*OF prems*(*6*) ∗(*1*) *prems*(*7*)] ∗(*2*) *that* **by** *auto*

from *admissible-targets*(*2*)[*OF prems*(*6*) ∗(*1*)] **have** $X \subseteq \mathcal{X}$ **unfolding** *𝒳-def* **by** *auto*
with *finite*(*1*) **have** *finite X* **by** (*blast intro*: *finite-subset*)
then obtain *r* **where** *set r = X* **using** *finite-list* **by** *auto*
then have *set* (*SOME r. set r = X*) = *X* **by** (*rule someI*)
with ‹$X \subseteq \mathcal{X}$› **have** *set* (*SOME r. set r = X*) $\subseteq \mathcal{X}$ **by** *auto*
with *alpha-interp.region-set'-closed*[*of R 0 SOME r. set r = X*] *prems*(*4*,*5*) ∗(*2*)
show $R' \in \mathcal{R}$ **unfolding** *𝒮-def 𝒳-def* **by** *auto*
qed
qed

sublocale *R-G*: *Finite-Markov-Decision-Process 𝒦 𝒮*
by (*standard*, *auto simp*: *𝒮-finite 𝒮-not-empty 𝒦-finite 𝒦-𝒮-closed*)

lemmas *𝒦-𝒮-closed'*[*intro*] = *R-G.set-pmf-closed*

5 Relating the MDPs

5.1 Translating From K to $\mathcal{K}$

lemma *ccompatible-inv*:
shows *ccompatible* $\mathcal{R}$ (*inv-of A l*)
proof −
from *valid* **have**
$\forall (x, m) \in$ *collect-clock-pairs* (*inv-of A l*). $m \leq$ *real* $(k\ x) \wedge x \in \mathcal{X} \wedge m \in \mathbb{N}$
unfolding *valid-abstraction-def clkp-set-def collect-clki-def* **by** *auto*
with *ccompatible*[*of - k 𝒳*, *folded ℛ-def*] **show** *?thesis* **by** *auto*
qed

lemma *ccompatible-guard*:
assumes $(l, g, \mu) \in$ *trans-of A*
shows *ccompatible* $\mathcal{R}$ *g*
proof −
from *assms valid* **have**
$\forall (x, m) \in$ *collect-clock-pairs g*. $m \leq$ *real* $(k\ x) \wedge x \in \mathcal{X} \wedge m \in \mathbb{N}$
unfolding *valid-abstraction-def clkp-set-def collect-clkt-def trans-of-def* **by** *fastforce*
with *assms ccompatible*[*of - k 𝒳*, *folded ℛ-def*] **show** *?thesis* **by** *auto*
qed

lemmas *ccompatible-def* = *ccompatible-def*[*unfolded ccval-def*]

lemma *region-set'-eq*:
fixes *X* :: *'c set*

assumes $R \in \mathcal{R}$ $u \in R$
and $A \vdash l \longrightarrow^{g,\mu,X} l'$
shows
$[[X{:=}0]u]_{\mathcal{R}} = \textit{region-set}'\ R\ (\textit{SOME}\ r.\ \textit{set}\ r = X)\ 0\ [[X{:=}0]u]_{\mathcal{R}} \in \mathcal{R}\ [X{:=}0]u \in [[X{:=}0]u]_{\mathcal{R}}$
proof −
let $?r = (\textit{SOME}\ r.\ \textit{set}\ r = X)$
from *admissible-targets-alt*[*OF assms*(*3*)] $\mathcal{X}$*-def finite* **have** *finite X*
by (*auto intro*: *finite-subset*)
then obtain *r* **where** *set r* = *X* **using** *finite-list* **by** *blast*
then have *set ?r* = *X* **by** (*intro someI*)
with *valid assms*(*3*) **have** *set ?r* $\subseteq \mathcal{X}$
by (*simp add*: *transition-*$\mathcal{X}$)
from *region-set*$'$*-id*[*of - $\mathcal{X}$ k*, *folded* $\mathcal{R}$*-def*, *OF assms*(*1*,*2*) *finite*(*1*) - - *this*]
show
$[[X{:=}0]u]_{\mathcal{R}} = \textit{region-set}'\ R\ (\textit{SOME}\ r.\ \textit{set}\ r = X)\ 0\ [[X{:=}0]u]_{\mathcal{R}} \in \mathcal{R}\ [X{:=}0]u \in [[X{:=}0]u]_{\mathcal{R}}$
by *force+*
qed

lemma *regions-part-ex-reset*:
assumes $u \in V$
shows $[r \to (d{::}nat)]u \in [[r \to d]u]_{\mathcal{R}}$ $[[r \to d]u]_{\mathcal{R}} \in \mathcal{R}$
using *assms* **by** *auto*

lemma *reset-sets-all-equiv*:
assumes $u \in V$ $u' \in [[r \to (d :: nat)]u]_{\mathcal{R}}$ $x \in set\ r$ $set\ r \subseteq \mathcal{X}$ $d \le k\ x$
shows $u'\ x = d$
proof −
from *assms*(*1*) **have** *u*: $[r \to d]u \in [[r \to d]u]_{\mathcal{R}}$ $[[r \to d]u]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
then obtain I ϱ **where** *I*: $[[r \to d]u]_{\mathcal{R}} = region\ \mathcal{X}\ I\ \varrho$ *valid-region* $\mathcal{X}\ k\ I\ \varrho$
by (*auto simp*: $\mathcal{R}$*-def*)
with *u*(*1*) *assms*(*3*−) **have** *intv-elem x* ($([r \to d]u)$) (*I x*) *valid-intv* (*k x*) (*I x*) **by** *fastforce+*
moreover from *assms* **have** $([r \to d]u)\ x = d$ **by** *simp*
ultimately have *I x* = *Const d* **using** *assms*(*5*) **by** (*cases I x*) *auto*
moreover from *I assms*(*2*−) **have** *intv-elem x u*$'$ (*I x*) **by** *fastforce*
ultimately show $u'\ x = d$ **by** *auto*
qed

lemma *reset-eq*:
assumes $u \in V$ $([[r \to 0]u]_{\mathcal{R}}) = ([[r' \to 0]u]_{\mathcal{R}})$ $set\ r \subseteq \mathcal{X}$ $set\ r' \subseteq \mathcal{X}$
shows $[r \to 0]u = [r' \to 0]u$ **using** *assms*
proof −
have ∗: $u'\ x = 0$ **if** $u' \in [[r \to 0]u]_{\mathcal{R}}$ $x \in set\ r$ **for** $u'\ x$
using *reset-sets-all-equiv*[*of u u*$'$ *r 0 x*] *that assms* **by** *auto*
have $u'\ x = 0$ **if** $u' \in [[r' \to 0]u]_{\mathcal{R}}$ $x \in set\ r'$ **for** $u'\ x$
using *reset-sets-all-equiv*[*of u u*$'$ *r*$'$ *0 x*] *that assms* **by** *auto*
from *regions-part-ex-reset*[*OF assms*(*1*), *of* - *0*] *assms*(*2*) **have** ∗∗:
$([r' \to 0]u) \in [[r \to 0]u]_{\mathcal{R}}$ $([r \to 0]u) \in [[r \to 0]u]_{\mathcal{R}}$ $[[r \to 0]u]_{\mathcal{R}} \in \mathcal{R}$
by *auto*
have $(([r \to 0]u)\ x) = (([r' \to 0]u)\ x)$ **for** *x*
proof (*cases* $x \in set\ r$)
case *True*
then have $([r \to 0]u)\ x = 0$ **by** *simp*
moreover from ∗ ∗∗ *True* **have** $([r' \to 0]u)\ x = 0$ **by** *auto*
ultimately show *?thesis* **..**
next
case *False*
then have *id*: $([r{\to}0]u)\ x = u\ x$ **by** *simp*
show *?thesis*
proof (*cases* $x \in set\ r'$)
case *True*
then have *reset*: $([r' \to 0]u)\ x = 0$ **by** *simp*

```
      show ?thesis
      proof (cases x ∈ 𝒳)
        case True
        from **(3) obtain I ϱ where
          ([([r → 0]u)]_ℛ) = Regions.region 𝒳 I ϱ Regions.valid-region 𝒳 k I ϱ
        by (auto simp: ℛ-def)
        with ** ‹x ∈ 𝒳› have ***:
          intv-elem x ([r' → 0]u) (I x) intv-elem x ([r → 0]u) (I x)
        by auto
        with reset have I x = Const 0 by (cases I x, auto)
        with ***(2) have ([r → 0]u) x = 0 by auto
        with reset show ?thesis by auto
      next
        case False
        with assms(3−) have x ∉ set r x ∉ set r' by auto
        then show ?thesis by simp
      qed
    next
      case False
      then have reset: ([r' → 0]u) x = u x by simp
      with id show ?thesis by simp
    qed
  qed
  then show ?thesis ..
qed

lemma admissible-targets-clocks:
  assumes (l, g, μ) ∈ trans-of A (X, l') ∈ μ
  shows X ⊆ 𝒳 set (SOME r. set r = X) ⊆ 𝒳
proof −
  from admissible-targets(2)[OF assms] finite have
    finite X X ⊆ 𝒳
  by (auto intro: finite-subset simp: 𝒳-def)
  then obtain r where set r = X using finite-list by blast
  with ‹X ⊆ 𝒳› show X ⊆ 𝒳 set (SOME r. set r = X) ⊆ 𝒳
    by (metis (mono-tags, lifting) someI-ex)+
qed

lemma
  rel-pmf (λ a b. f a = b) μ (map-pmf f μ)
by (subst pmf.rel-map(2)) (rule rel-pmf-reflI, auto)

lemma K-pmf-rel:
  defines f ≡ λ (l, u). (l, [u]_ℛ)
  shows rel-pmf (λ (l, u) st. (l, [u]_ℛ) = st) μ (map-pmf f μ) unfolding f-def
by (subst pmf.rel-map(2)) (rule rel-pmf-reflI, auto)

lemma 𝒦-pmf-rel:
  assumes A: μ ∈ 𝒦 (l, R)
  defines f ≡ λ (l, u). (l, SOME u. u ∈ R)
  shows rel-pmf (λ (l, u) st. (l, SOME u. u ∈ R) = st) μ (map-pmf f μ) unfolding f-def
by (subst pmf.rel-map(2)) (rule rel-pmf-reflI, auto)

lemma K-elem-abs-inj:
  assumes A: μ ∈ K (l, u)
  defines f ≡ λ (l, u). (l, [u]_ℛ)
  shows inj-on f μ
proof −
  have (l1, u1) = (l2, u2)
    if id: (l1, [u1]_ℛ) = (l2, [u2]_ℛ) and elem: (l1, u1) ∈ μ (l2, u2) ∈ μ for l1 l2 u1 u2
  proof −
```

```
    from id have [simp]: l2 = l1 by auto
    from A
    show ?thesis
    proof (cases, safe, goal-cases)
      case (4 - - τ μ')
      from ‹μ = -› elem obtain X1 X2 where
        u1 = [(SOME r. set r = X1)→0]u (X1, l1) ∈ μ'
        u2 = [(SOME r. set r = X2)→0]u (X2, l1) ∈ μ'
      by auto
      with ‹- ∈ trans-of -› admissible-targets-clocks have
        set (SOME r. set r = X1) ⊆ 𝒳 set (SOME r. set r = X2) ⊆ 𝒳
      by auto
      with id ‹u1 = -› ‹u2 = -› reset-eq[of u] ‹- ∈ S› show ?case by (auto simp: S-def V-def)
    qed (−, insert elem, simp)+
  qed
  then show ?thesis unfolding f-def inj-on-def by auto
qed

lemma K-elem-repr-inj:
  notes alpha-interp.valid-regions-distinct-spec[intro]
  assumes A: μ ∈ 𝒦 (l, R)
  defines f ≡ λ (l, R). (l, SOME u. u ∈ R)
  shows inj-on f μ
proof −
  have (l1, R1) = (l2, R2)
    if id: (l1, SOME u. u ∈ R1) = (l2, SOME u. u ∈ R2) and elem: (l1, R1) ∈ μ (l2, R2) ∈ μ
    for l1 l2 R1 R2
  proof −
    let ?r1 = SOME u. u ∈ R1 and ?r2 = SOME u. u ∈ R2
    from id have [simp]: l2 = l1 ?r2 = ?r1 by auto
    { fix g μ' x
      assume (l, R) ∈ 𝒮 (l, g, μ') ∈ PTA.trans-of A R ⊆ {v. v ⊢ g}
         and μ = map-pmf (λ(X, l). (l, region-set' R (SOME r. set r = X) 0)) μ'
      from ‹μ = -› elem obtain X1 X2 where
        R1 = region-set' R (SOME r. set r = X1) 0 (X1, l1) ∈ μ'
        R2 = region-set' R (SOME r. set r = X2) 0 (X2, l1) ∈ μ'
      by auto
      with ‹- ∈ trans-of -› admissible-targets-clocks have
        set (SOME r. set r = X1) ⊆ 𝒳 set (SOME r. set r = X2) ⊆ 𝒳
      by auto
      with alpha-interp.region-set'-closed[of - 0] ‹R1 = -› ‹R2 = -› ‹- ∈ 𝒮› have
        R1 ∈ ℛ R2 ∈ ℛ
      unfolding 𝒮-def by auto
      with region-not-empty[OF finite(1)] have
        R1 ≠ {} R2 ≠ {} ∃u. u ∈ R1 ∃u. u ∈ R2
      by (auto simp: ℛ-def)
      from someI-ex[OF this(3)] someI-ex[OF this(4)] have ?r1 ∈ R1 ?r1 ∈ R2 by simp+
      with ‹R1 ∈ ℛ› ‹R2 ∈ ℛ› have R1 = R2 ..
    }
    from A elem this show ?thesis by (cases, auto)
  qed
  then show ?thesis unfolding f-def inj-on-def by auto
qed

lemma K-elem-pmf-map-abs:
  assumes A: μ ∈ K (l, u) (l', u') ∈ μ
  defines f ≡ λ (l, u). (l, [u]ℛ)
  shows pmf (map-pmf f μ) (f (l', u')) = pmf μ (l', u')
using A unfolding f-def by (blast intro: pmf-map-inj K-elem-abs-inj)

lemma K-elem-pmf-map-repr:
```

assumes A: $\mu \in \mathcal{K}\ (l,\ R)\ (l',\ R') \in \mu$
defines $f \equiv \lambda\ (l,\ R).\ (l,\ \mathit{SOME}\ u.\ u \in R)$
shows $\mathit{pmf}\ (\mathit{map\text{-}pmf}\ f\ \mu)\ (f\ (l',\ R')) = \mathit{pmf}\ \mu\ (l',\ R')$
using A **unfolding** *f-def* **by** (*blast intro*: *pmf-map-inj K-elem-repr-inj*)

definition $\mathit{transp} :: ('s * ('c,\ t)\ \mathit{cval} \Rightarrow \mathit{bool}) \Rightarrow 's * ('c,\ t)\ \mathit{cval}\ \mathit{set} \Rightarrow \mathit{bool}$ **where**
$\mathit{transp}\ \varphi \equiv \lambda\ (l,\ R).\ \forall\ u \in R.\ \varphi\ (l,\ u)$

5.2 Translating Configurations

5.2.1 States

definition
$\mathit{abss} :: 's * ('c,\ t)\ \mathit{cval} \Rightarrow 's * ('c,\ t)\ \mathit{cval}\ \mathit{set}$
where
$\mathit{abss} \equiv \lambda\ (l,\ u).\ \mathit{if}\ u \in V\ \mathit{then}\ (l,\ [u]_{\mathcal{R}})\ \mathit{else}\ (l,\ -V)$

definition
$\mathit{reps} :: 's * ('c,\ t)\ \mathit{cval}\ \mathit{set} \Rightarrow 's * ('c,\ t)\ \mathit{cval}$
where
$\mathit{reps} \equiv \lambda\ (l,\ R).\ \mathit{if}\ R \in \mathcal{R}\ \mathit{then}\ (l,\ \mathit{SOME}\ u.\ u \in R)\ \mathit{else}\ (l,\ \lambda\text{-}.\ -1)$

lemma $\mathcal{S}$*-reps-S*[*intro*]:
assumes $s \in \mathcal{S}$
shows $\mathit{reps}\ s \in S$
using *assms* $\mathcal{R}$*-V* **unfolding** *S-def* $\mathcal{S}$*-def reps-def V-def* **by** *force*

lemma *S-abss-*$\mathcal{S}$[*intro*]:
assumes $s \in S$
shows $\mathit{abss}\ s \in \mathcal{S}$
using *assms ccompatible-inv* **unfolding** $\mathcal{S}$*-def S-alt-def abss-def ccompatible-def* **by** *force*

lemma $\mathcal{S}$*-abss-reps*[*simp*]:
$s \in \mathcal{S} \Longrightarrow \mathit{abss}\ (\mathit{reps}\ s) = s$
using $\mathcal{R}$*-V alpha-interp.region-unique-spec* **by** (*auto simp*: *S-def* $\mathcal{S}$*-def reps-def abss-def*; *blast*)

lemma *map-pmf-abs-reps*:
assumes $s \in \mathcal{S}\ \mu \in \mathcal{K}\ s$
shows $\mathit{map\text{-}pmf}\ \mathit{abss}\ (\mathit{map\text{-}pmf}\ \mathit{reps}\ \mu) = \mu$
proof −
have $\mathit{map\text{-}pmf}\ \mathit{abss}\ (\mathit{map\text{-}pmf}\ \mathit{reps}\ \mu) = \mathit{map\text{-}pmf}\ (\mathit{abss}\ o\ \mathit{reps})\ \mu$ **by** (*simp add*: *pmf.map-comp*)
also have $\ldots = \mu$
proof (*rule map-pmf-idI*, *safe*, *goal-cases*)
case *prems*: $(1\ l'\ R')$
with *assms* **have** $(l',\ R') \in \mathcal{S}\ \mathit{reps}\ (l',\ R') \in S$ **by** *auto*
then show *?case* **by** *auto*
qed
finally show *?thesis* **by** *auto*
qed

lemma *abss-reps-id*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes $s' \in \mathcal{S}\ s \in \mathit{set\text{-}pmf}\ (\mathit{action}\ \mathit{cfg})\ \mathit{cfg} \in \mathit{R\text{-}G.cfg\text{-}on}\ s'$
shows $\mathit{abss}\ (\mathit{reps}\ s) = s$
proof −
from *assms* **have** $s \in \mathcal{S}$ **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *abss-S*[*intro*]:
  assumes $(l, u) \in S$
  shows $abss\ (l, u) = (l, [u]_{\mathcal{R}})$
using *assms* **unfolding** *abss-def* **by** *auto*

lemma *reps-$\mathcal{S}$*[*intro*]:
  assumes $(l, R) \in \mathcal{S}$
  shows $reps\ (l, R) = (l, SOME\ u.\ u \in R)$
using *assms* **unfolding** *reps-def* **by** *auto*

lemma *fst-abss*:
  $fst\ (abss\ st) = fst\ st$ **for** *st*
  by (*cases st*) (*auto simp*: *abss-def*)

lemma *K-elem-abss-inj*:
  assumes A: $\mu \in K\ (l, u)\ (l, u) \in S$
  shows *inj-on abss* μ
proof −
  from *assms* **have** $abss\ s' = (\lambda\ (l, u).\ (l, [u]_{\mathcal{R}}))\ s'$ **if** $s' \in \mu$ **for** s'
  using *that* **by** (*auto split*: *prod.split*)
  from *inj-on-cong*[*OF this*] *K-elem-abs-inj*[*OF A*(*1*)] **show** *?thesis* **by** *force*
qed

lemma *$\mathcal{K}$-elem-reps-inj*:
  assumes A: $\mu \in \mathcal{K}\ (l, R)\ (l, R) \in \mathcal{S}$
  shows *inj-on reps* μ
proof −
  from *assms* **have** $reps\ s' = (\lambda\ (l, R).\ (l, SOME\ u.\ u \in R))\ s'$ **if** $s' \in \mu$ **for** s'
  using *that* **by** (*auto split*: *prod.split*)
  from *inj-on-cong*[*OF this*] *K-elem-repr-inj*[*OF A*(*1*)] **show** *?thesis* **by** *force*
qed

lemma *P-elem-pmf-map-abss*:
  assumes A: $\mu \in K\ (l, u)\ (l, u) \in S\ s' \in \mu$
  shows $pmf\ (map\text{-}pmf\ abss\ \mu)\ (abss\ s') = pmf\ \mu\ s'$
using A **by** (*blast intro*: *pmf-map-inj K-elem-abss-inj*)

lemma *$\mathcal{K}$-elem-pmf-map-reps*:
  assumes A: $\mu \in \mathcal{K}\ (l, R)\ (l, R) \in \mathcal{S}\ (l', R') \in \mu$
  shows $pmf\ (map\text{-}pmf\ reps\ \mu)\ (reps\ (l', R')) = pmf\ \mu\ (l', R')$
using A **by** (*blast intro*: *pmf-map-inj $\mathcal{K}$-elem-reps-inj*)

We need that $\mathcal{X}$ is non-trivial here

lemma *not-$\mathcal{S}$-reps*:
  $(l, R) \notin \mathcal{S} \Longrightarrow reps\ (l, R) \notin S$
proof −
  assume $(l, R) \notin \mathcal{S}$
  let $?u = SOME\ u.\ u \in R$
  have $\neg\ ?u \vdash inv\text{-}of\ A\ l$ **if** $R \in \mathcal{R}\ l \in L$
  proof −

    from *region-not-empty*[*OF finite*(*1*)] ‹$R \in \mathcal{R}$› **have** $\exists\, u.\ u \in R$ **by** (*auto simp*: *$\mathcal{R}$-def*)
    from *someI-ex*[*OF this*] **have** $?u \in R$ **.**
    moreover from ‹$(l, R) \notin \mathcal{S}$› *that* **have** $\neg\ R \subseteq \{\!|inv\text{-}of\ A\ l|\!\}$ **by** (*auto simp*: *$\mathcal{S}$-def*)
    ultimately show *?thesis*
      using *ccompatible-inv*[*of l*] ‹$R \in \mathcal{R}$› **unfolding** *ccompatible-def* **by** *fastforce*
  qed
  with *non-empty* ‹$(l, R) \notin \mathcal{S}$› **show** *?thesis* **unfolding** *$\mathcal{S}$-def S-def reps-def* **by** *auto*
qed

lemma *neq-V-not-region*:
  $-V \notin \mathcal{R}$
using $\mathcal{R}$*-V rep-*$\mathcal{R}$*-ex* **by** *auto*

lemma $\mathcal{S}$*-abss-S*:
  $abss\ s \in \mathcal{S} \Longrightarrow s \in S$
  unfolding *abss-def* $\mathcal{S}$*-def S-def*
  apply *safe*
  subgoal for - - - u
    by (*cases* $u \in V$) *auto*
  subgoal for - - - u
    using *neq-V-not-region* **by** (*cases* $u \in V$, (*auto simp*: *V-def*; *fail*), *auto*)
  subgoal for $l'\ y\ l\ u$
    using *neq-V-not-region* **by** (*cases* $u \in V$; *auto dest*: *regions-part-ex*)
  done

lemma *S-pred-stream-abss-*$\mathcal{S}$:
  pred-stream ($\lambda\ s.\ s \in S$) *xs* $\longleftrightarrow$ *pred-stream* ($\lambda\ s.\ s \in \mathcal{S}$) (*smap abss xs*)
using *S-abss-*$\mathcal{S}$ $\mathcal{S}$*-abss-S* **by** (*auto simp*: *stream.pred-set*)

sublocale *MDP*: *Markov-Decision-Process-Invariant K S* **by** (*standard*, *auto*)

abbreviation (*input*) *valid-cfg* $\equiv$ *MDP.valid-cfg*

lemma *K-closed*:
  $s \in S \Longrightarrow (\bigcup D{\in}K\ s.\ set\text{-}pmf\ D) \subseteq S$
  by *auto*

5.2.2 Intermezzo

abbreviation *timed-bisim* (**infixr** ‹~› *60*) **where**
  $s \sim s' \equiv abss\ s = abss\ s'$

lemma *bisim-loc-id*[*intro*]:
  $(l,\ u) \sim (l',\ u') \Longrightarrow l = l'$
unfolding *abss-def* **by** (*cases* $u \in V$; *cases* $u' \in V$; *simp*)

lemma *bisim-val-id*[*intro*]:
  $[u]_{\mathcal{R}} = [u']_{\mathcal{R}}$ **if** $u \in V\ (l,\ u) \sim (l',\ u')$
proof −
  have $(l',\ -\ V) \neq (l,\ [u]_{\mathcal{R}})$
    using *that* **by** *blast*
  with *that* **have** $u' \in V$
    by (*force simp*: *abss-def*)
  with *that* **show** *?thesis*
    by (*simp add*: *abss-def*)
qed

lemma *bisim-symmetric*:
  $(l,\ u) \sim (l',\ u') = (l',\ u') \sim (l,\ u)$
by (*rule eq-commute*)

lemma *bisim-val-id2*[*intro*]:
  $u' \in V \Longrightarrow (l,\ u) \sim (l',\ u') \Longrightarrow [u]_{\mathcal{R}} = [u']_{\mathcal{R}}$
 apply (*subst* (*asm*) *eq-commute*)
 apply (*subst eq-commute*)
 apply (*rule bisim-val-id*)
by *auto*

lemma *K-bisim-unique*:

assumes $s \in S$ $\mu \in K\ s$ $x \in \mu$ $x' \in \mu$ $x \sim x'$
shows $x = x'$
using *assms*(*2*,*1*,*3*−)
proof (*cases rule*: *K.cases*)
case *prems*: (*action l u* τ μ')
with *assms* **obtain** *l1 l2 X1 X2* **where** *A*:
($X1$, $l1$) $\in$ *set-pmf* μ' ($X2$, $l2$) $\in$ *set-pmf* μ'
$x = (l1, [X1{:=}0]u)$ $x' = (l2, [X2{:=}0]u)$
by *auto*
from ‹$x \sim x'$› *A* ‹$s \in S$› ‹$s = (l, u)$› **have** $[[X1{:=}0]u]_{\mathcal{R}} = [[X2{:=}0]u]_{\mathcal{R}}$
using *bisim-val-id*[*OF S-V*] *K-S-closed assms*(*2*−*4*) **by** (*auto intro*!: *bisim-val-id*[*OF S-V*])
then have $[X1{:=}0]u = [X2{:=}0]u$
using *A admissible-targets-clocks*(*2*)[*OF prems*(*4*)] *prems*(*2*,*3*) **by** − (*rule reset-eq*, *force*)
with *A* ‹$x \sim x'$› **show** *?thesis* **by** *auto*
next
case *delay*
with *assms*(*3*−) **show** *?thesis* **by** *auto*
next
case *loop*
with *assms*(*3*−) **show** *?thesis* **by** *auto*
qed

5.2.3 Predicates

definition *absp* **where**
absp $\varphi \equiv \varphi$ *o reps*

definition *repp* **where**
repp $\varphi \equiv \varphi$ *o absp*

5.2.4 Distributions

definition
abst :: ($'s$ ∗ ($'c$, *t*) *cval*) *pmf* ⇒ ($'s$ ∗ ($'c$, *t*) *cval set*) *pmf*
where
abst = *map-pmf abss*

lemma *abss-*$\mathcal{S}D$:
assumes *abss s* $\in \mathcal{S}$
obtains *l u* **where** $s = (l, u)$ $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$
proof −
obtain *l u* **where** $s = (l, u)$ **by** *force*
moreover from $\mathcal{S}$*-abss-S*[*OF assms*] **have** $s \in S$ **.**
ultimately have *abss s* $= (l, [u]_{\mathcal{R}})$ $u \in V$ $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
with ‹s = -› **show** *?thesis* **by** (*auto intro*: *that*)
qed

lemma *abss-*$\mathcal{S}D'$:
assumes *abss s* $\in \mathcal{S}$ *abss s* $= (l, R)$
obtains *u* **where** $s = (l, u)$ $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$ $R = [u]_{\mathcal{R}}$
proof −
from *abss-*$\mathcal{S}D$[*OF assms*(*1*)] **obtain** l' *u* **where** *u*:
$s = (l', u)$ $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$
by *blast*+
with $\mathcal{R}$*-V* **have** $u \in V$ **by** *auto*
with ‹s = -› *assms*(*2*) **have** $l' = l$ $R = [u]_{\mathcal{R}}$ **unfolding** *abss-def* **by** *auto*
with *u* **show** *?thesis* **by** (*auto intro*: *that*)
qed

definition *infR R* ≡ λ *c*. *of-int* ⌊(*SOME u*. *u* ∈ *R*) *c*⌋

term *let a* = *3 in b*

definition *delayedR R u* ≡
  u ⊕ (
    let I = (*SOME I*. ∃ *r*. *valid-region* 𝒳 *k I r* ∧ *R* = *region* 𝒳 *I r*);
        m = *1* − *Max* ({*frac* (*u c*) | *c*. *c* ∈ 𝒳 ∧ *isIntv* (*I c*)} ∪ {*0*})
    in SOME t. *u* ⊕ *t* ∈ *R* ∧ *t* ≥ *m* / *2*
)

lemma *delayedR-correct-aux-aux*:
  fixes *c* :: *nat*
  fixes *a b* :: *real*
  assumes *c* < *a a* < *Suc c b* ≥ *0 a* + *b* < *Suc c*
  shows *frac* (*a* + *b*) = *frac a* + *b*

proof −
  have *f1*: *a* + *b* < *real* (*c* + *1*)
    using *assms*(*4*) **by** *auto*
  have *f2*: ⋀*r ra*. (*r*::*real*) + (− *r* + *ra*) = *ra*
    by *linarith*
  have *f3*: ⋀*r*. (*r*::*real*) = − (− *r*)
    by *linarith*
  have *f4*: ⋀*r ra*. − (*r*::*real*) + (*ra* + *r*) = *ra*
    by *linarith*
  then have *f5*: ⋀*r n*. *r* + − *frac r* = *real n* ∨ ¬ *r* < *real* (*n* + *1*) ∨ ¬ *real n* < *r*
    using *f2* **by** (*metis nat-intv-frac-decomp*)
  then have *frac a* + *real c* = *a*
    using *f4 f3* **by** (*metis One-nat-def add.right-neutral add-Suc-right assms*(*1*) *assms*(*2*))
  then show *?thesis*
    using *f5 f1 assms*(*1*) *assms*(*3*) **by** *fastforce*
qed

lemma *delayedR-correct-aux*:
  fixes *I r*
  defines *R* ≡ *region* 𝒳 *I r*
  assumes *u* ∈ *R valid-region* 𝒳 *k I r* ∀ *c* ∈ 𝒳. ¬ *isConst* (*I c*)
        ∀ *c* ∈ 𝒳. *isIntv* (*I c*) ⟶ (*u* ⊕ *t*) *c* < *intv-const* (*I c*) + *1*
        t ≥ *0*
  shows *u* ⊕ *t* ∈ *R* **unfolding** *R-def*
proof
  from *assms* **have** *R* ∈ ℛ **unfolding** *ℛ-def* **by** *auto*
  with ‹*u* ∈ *R*› *ℛ-V* **have** *u* ∈ *V* **by** *auto*
  with ‹*t* ≥ *0*› **show** ∀*x*∈𝒳. *0* ≤ (*u* ⊕ *t*) *x* **unfolding** *V-def* **by** (*auto simp*: *cval-add-def*)
  have *intv-elem x* (*u* ⊕ *t*) (*I x*) **if** *x* ∈ 𝒳 **for** *x*
  proof (*cases I x*)
    case *Const*
    with *assms* ‹*x* ∈ 𝒳› **show** *?thesis* **by** *auto*
  next
    case (*Intv c*)
    with *assms* ‹*x* ∈ 𝒳› **show** *?thesis* **by** (*simp add*: *cval-add-def*) (*rule*; *force*)
  next
    case (*Greater c*)

with *assms* ‹$x \in \mathcal{X}$› **show** *?thesis* **by** (*fastforce simp add*: *cval-add-def*)
qed
then show $\forall x \in \mathcal{X}.$ *intv-elem* x $(u \oplus t)$ $(I\ x)$ **..**

let $?X_0 = \{x \in \mathcal{X}.\ \exists d.\ I\ x = Intv\ d\}$
show $?X_0 = ?X_0$ **by** *auto*

have $frac\ (u\ x + t) = frac\ (u\ x) + t$ **if** $x \in ?X_0$ **for** x
proof −
show *?thesis*
apply (*rule delayedR-correct-aux-aux*[**where** $c = intv\text{-}const\ (I\ x)$])
using *assms* ‹$x \in ?X_0$› **by** (*force simp add*: *cval-add-def*)+
qed
then have $frac\ (u\ x) \le frac\ (u\ y) \longleftrightarrow frac\ (u\ x + t) \le frac\ (u\ y + t)$ **if** $x \in ?X_0$ $y \in ?X_0$ **for** x y
using *that* **by** *auto*
with *assms* **show**
$\forall x \in ?X_0.\ \forall y \in ?X_0.\ ((x, y) \in r) = (frac\ ((u \oplus t)\ x) \le frac\ ((u \oplus t)\ y))$
unfolding *cval-add-def* **by** *auto*
qed

lemma *delayedR-correct-aux′*:
fixes I r
defines $R \equiv region\ \mathcal{X}\ I\ r$
assumes $u \oplus t1 \in R$ *valid-region* $\mathcal{X}$ k I r $\forall\ c \in \mathcal{X}.\ \neg\ isConst\ (I\ c)$
$\forall\ c \in \mathcal{X}.\ isIntv\ (I\ c) \longrightarrow (u \oplus t2)\ c < intv\text{-}const\ (I\ c) + 1$
$t1 \le t2$
shows $u \oplus t2 \in R$
proof −
have $(u \oplus t1) \oplus (t2 - t1) \in R$ **unfolding** *R-def*
using *assms* **by** − (*rule delayedR-correct-aux*, *auto simp*: *cval-add-def*)
then show $u \oplus t2 \in R$ **by** (*simp add*: *cval-add-def*)
qed

lemma *valid-regions-intv-distinct*:
valid-region X k I $r \Longrightarrow$ *valid-region* X k I' $r' \Longrightarrow u \in region\ X\ I\ r \Longrightarrow u \in region\ X\ I'\ r'$
$\Longrightarrow x \in X \Longrightarrow I\ x = I'\ x$
proof *goal-cases*
case *A*: *1*
note $x =$ ‹$x \in X$›
with *A* **have** *valid-intv* $(k\ x)$ $(I\ x)$ **by** *auto*
moreover from *A*(*2*) x **have** *valid-intv* $(k\ x)$ $(I'\ x)$ **by** *auto*
moreover from *A*(*3*) x **have** *intv-elem* x u $(I\ x)$ **by** *auto*
moreover from *A*(*4*) x **have** *intv-elem* x u $(I'\ x)$ **by** *auto*
ultimately show $I\ x = I'\ x$ **using** *valid-intv-distinct* **by** *fastforce*
qed

lemma *delayedR-correct*:
fixes I r
defines $R' \equiv region\ \mathcal{X}\ I\ r$
assumes $u \in R$ $R \in \mathcal{R}$ *valid-region* $\mathcal{X}$ k I r $\forall\ c \in \mathcal{X}.\ \neg\ isConst\ (I\ c)$ $R' \in Succ\ \mathcal{R}\ R$
shows
delayedR R' $u \in R'$
$\exists\ t \ge 0.\ delayedR\ R'\ u = u \oplus t$
$\wedge\ t \ge (1 - Max\ (\{frac\ (u\ c) \mid c.\ c \in \mathcal{X} \wedge isIntv\ (I\ c)\} \cup \{0\}))\ /\ 2$
proof −
let $?u = SOME\ u.\ u \in R$
let $?I = SOME\ I.\ \exists\ r.\ valid\text{-}region\ \mathcal{X}\ k\ I\ r \wedge R' = region\ \mathcal{X}\ I\ r$
let $?S = \{frac\ (u\ c) \mid c.\ c \in \mathcal{X} \wedge isIntv\ (I\ c)\}$

```
  let ?m = 1 − Max (?S ∪ {0})
  let ?t = SOME t. u ⊕ t ∈ R′ ∧ t ≥ ?m / 2
  have Max (?S ∪ {0}) ≥ 0 ?m ≤ 1 using finite(1) by auto
  have Max (?S ∪ {0}) ∈ ?S ∪ {0} using finite(1) by − (rule Max-in; auto)
  with frac-lt-1 have Max (?S ∪ {0}) ≤ 1 ?m ≥ 0 by auto
  from assms(3, 6) ‹u ∈ R› obtain t where t:
    u ⊕ t ∈ R′ t ≥ 0
  by (metis alpha-interp.regions-closed′-spec alpha-interp.set-of-regions-spec)
  have I-cong: ∀ c ∈ 𝒳. I′ c = I c if valid-region 𝒳 k I′ r′ R′ = region 𝒳 I′ r′ for I′ r′
  using valid-regions-intv-distinct assms(4) t(1) that unfolding R′-def by auto
  have I-cong: ?I c = I c if c ∈ 𝒳 for c
  proof −
    from assms have
      ∃ r. valid-region 𝒳 k ?I r ∧ R′ = region 𝒳 ?I r
    by − (rule someI[where P = λ I. ∃ r. valid-region 𝒳 k I r ∧ R′ = region 𝒳 I r]; auto)
    with I-cong that show ?thesis by auto
  qed
  then have ?S = {frac (u c) | c. c ∈ 𝒳 ∧ isIntv (?I c)} by auto
  have upper-bound: (u ⊕ ?m / 2) c < intv-const (I c) + 1 if c ∈ 𝒳 isIntv (I c) for c
  proof (cases u c > intv-const (I c))
    case True
    from t that assms have u c + t < intv-const (I c) + 1 unfolding cval-add-def by fastforce
    with ‹t ≥ 0› True have ∗: intv-const (I c) < u c u c < intv-const (I c) + 1 by auto
    have frac (u c) ≤ Max (?S ∪ {0}) using finite(1) that by − (rule Max-ge; auto)
    then have ?m ≤ 1 − frac (u c) by auto
    then have ?m / 2 < 1 − frac (u c) using ∗ nat-intv-frac-decomp by fastforce
    then have (u ⊕ ?m / 2) c < u c + 1 − frac (u c) unfolding cval-add-def by auto
    also from ∗ have
      . . . = intv-const (I c) + 1
    using nat-intv-frac-decomp of-nat-1 of-nat-add by fastforce
    finally show ?thesis .
  next
    case False
    then have u c ≤ intv-const (I c) by auto
    moreover from ‹0 ≤ ?m› ‹?m ≤ 1› have ?m / 2 < 1 by auto
    ultimately have u c + ?m / 2 < intv-const (I c) + 1 by linarith
    then show ?thesis by (simp add: cval-add-def)
  qed
  have ?t ≥ 0 ∧ u ⊕ ?t ∈ R′ ∧ ?t ≥ ?m / 2
  proof (cases t ≥ ?m / 2)
    case True
    from ‹t ≥ ?m / 2› t ‹Max (?S ∪ {0}) ≤ 1› have u ⊕ ?t ∈ R′ ∧ ?t ≥ ?m / 2
      by − (rule someI; auto)
    with ‹?m ≥ 0› show ?thesis by auto
  next
    case False
    have u ⊕ ?m / 2 ∈ R′ unfolding R′-def
     apply (rule delayedR-correct-aux′)
         apply (rule t[unfolded R′-def])
        apply (rule assms)+
    using upper-bound False by auto
    with ‹?m ≥ 0› show ?thesis by − (rule someI2; fastforce)
  qed
  then show delayedR R′ u ∈ R′ ∃t≥0. delayedR R′ u = u ⊕ t ∧ t ≥ ?m / 2
    by (auto simp: delayedR-def ‹?S = -›)
qed

definition
  rept :: ′s ∗ (′c, t) cval ⇒ (′s ∗ (′c, t) cval set) pmf ⇒ (′s ∗ (′c, t) cval) pmf
where
  rept s μ-abs ≡ let (l, u) = s in
```

if $(\exists\ R'.\ (l,\ u) \in S \wedge \mu\text{-}abs = return\text{-}pmf\ (l,\ R') \wedge$
$(([u]_{\mathcal{R}} = R' \wedge (\forall\ c \in \mathcal{X}.\ u\ c > k\ c))))$
then $return\text{-}pmf\ (l,\ u \oplus 0.5)$
else if
$(\exists\ R'.\ (l,\ u) \in S \wedge \mu\text{-}abs = return\text{-}pmf\ (l,\ R') \wedge R' \in Succ\ \mathcal{R}\ ([u]_{\mathcal{R}}) \wedge [u]_{\mathcal{R}} \neq R'$
$\wedge\ (\forall\ u \in R'.\ \forall\ c \in \mathcal{X}.\ \nexists\ d.\ d \leq k\ c \wedge\ u\ c = real\ d))$
then $return\text{-}pmf\ (l,\ delayedR\ (SOME\ R'.\ \mu\text{-}abs = return\text{-}pmf\ (l,\ R'))\ u)$
else $SOME\ \mu.\ \mu \in K\ s \wedge abst\ \mu = \mu\text{-}abs$

lemma $\mathcal{S}$-*L*:
$l \in L$ **if** $(l,\ R) \in \mathcal{S}$
using *that* **unfolding** $\mathcal{S}$-*def* **by** *auto*

lemma $\mathcal{S}$-*inv*:
$(l,\ R) \in \mathcal{S} \Longrightarrow R \subseteq \{\!|inv\text{-}of\ A\ l|\!\}$
unfolding $\mathcal{S}$-*def* **by** *auto*

lemma *upper-right-closed*:
assumes $\forall\, c{\in}\mathcal{X}.\ real\ (k\ c) < u\ c\ u \in R\ R \in \mathcal{R}\ t \geq 0$
shows $u \oplus t \in R$
proof –
from ‹$R \in \mathcal{R}$› **obtain** $I\ r$ **where** R:
$R = region\ \mathcal{X}\ I\ r\ valid\text{-}region\ \mathcal{X}\ k\ I\ r$
unfolding $\mathcal{R}$-*def* **by** *auto*
from *assms* $\mathcal{R}$-*V* **have** $u \in V$ **by** *auto*
from *assms* R **have** $\forall\ c \in \mathcal{X}.\ I\ c = Greater\ (k\ c)$ **by** *safe* (*case-tac I c*; *fastforce*)
with R ‹$u \in V$› *assms* **show**
$u \oplus t \in R$
unfolding *V-def* **by** *safe* (*rule*; *force simp*: *cval-add-def*)
qed

lemma *S-I*[*intro*]:
$(l,\ u) \in S$ **if** $l \in L\ u \in V\ u \vdash inv\text{-}of\ A\ l$
using *that* **by** (*auto simp*: *S-def V-def*)

lemma *rept-ex*:
assumes $\mu \in \mathcal{K}\ (abss\ s)$
shows $rept\ s\ \mu \in K\ s \wedge abst\ (rept\ s\ \mu) = \mu$ **using** *assms*
proof *cases*
case *prems*: $(delay\ l\ R\ R')$
then have $R \in \mathcal{R}$ **by** *auto*
from *prems*(*2*) **have** $s \in S$ **by** (*auto intro*: $\mathcal{S}$-*abss-S*)
from *abss*-$\mathcal{S}$*D*[*OF prems*(*2*)] **obtain** $l'\ u'$ **where** $s = (l',\ u')\ u' \in [u']_{\mathcal{R}}$
by *metis*
with *prems*(*3*) **have** $*$: $s = (l,\ u') \wedge u' \in R$
apply *simp*
apply (*subst* (*asm*) *abss-S*[*OF* $\mathcal{S}$-*abss-S*])
using *prems*(*2*) **by** *auto*
with *prems*(*4*) *alpha-interp.set-of-regions-spec*[*OF* ‹$R \in \mathcal{R}$›] **obtain** t **where** R':
$t \geq 0\ R' = [u' \oplus t]_{\mathcal{R}}$
by *auto*
with ‹$s \in S$› $*$ **have** $u' \oplus t \in R'\ u' \oplus t \in V\ l \in L$ **by** *auto*
with *prems*(*5*) **have** $(l,\ u' \oplus t) \in S$ **unfolding** *S-def V-def* **by** *auto*
with ‹$R' = [u' \oplus t]_{\mathcal{R}}$› **have** $**$: $abss\ (l,\ u' \oplus t) = (l,\ R')$ **by** (*auto simp*: *abss-S*)
let $?\mu = return\text{-}pmf\ (l,\ u' \oplus t)$
have $?\mu \in K\ s$ **using** $*$ ‹$s \in S$› ‹$t \geq 0$› ‹$u' \oplus t \in R'$› *prems* **by** *blast*
moreover have $abst\ ?\mu = \mu$ **by** (*simp add*: $**$ *abst-def prems*(*1*))
moreover note *default* = *calculation*
have $R' \in \mathcal{R}$ **using** *prems*(*4*) **by** *auto*
have R: $[u']_{\mathcal{R}} = R$ **by** (*simp add*: $*$ ‹$R \in \mathcal{R}$› *alpha-interp.region-unique-spec*)

```
  from ‹R' ∈ ℛ› obtain I r where R':
    R' = region 𝒳 I r valid-region 𝒳 k I r
  unfolding ℛ-def by auto
  have u' ∈ V using ∗ prems ℛ-V by force
  let ?μ' = return-pmf (l, u' ⊕ 0.5)
  have elapsed: abst (return-pmf (l, u' ⊕ t)) = μ return-pmf (l, u' ⊕ t) ∈ K s
    if u' ⊕ t ∈ R' t ≥ 0 for t
  proof −
    let ?u = u' ⊕ t let ?μ' = return-pmf (l, u' ⊕ t)
    from ‹?u ∈ R'› ‹R' ∈ ℛ› ℛ-V have ?u ∈ V by auto
    with ‹?u ∈ R'› ‹R' ∈ ℛ› have [?u]ℛ = R' using alpha-interp.region-unique-spec by auto
    with ‹?u ∈ V› ‹?u ∈ R'› ‹l ∈ L› prems(4,5) have abss (l, ?u) = (l, R')
      by (subst abss-S) auto
    with prems(1) have abst ?μ' = μ by (auto simp: abst-def)
    moreover from ∗ ‹?u ∈ R'› ‹s ∈ S› prems ‹t ≥ 0› have ?μ' ∈ K s by auto
    ultimately show abst ?μ' = μ ?μ' ∈ K s by auto
  qed
  show ?thesis
  proof (cases R = R')
    case T: True
    show ?thesis
    proof (cases ∀ c ∈ 𝒳. u' c > k c)
      case True
      with T ∗ R prems(1,4) ‹s ∈ S› have
        rept s μ = return-pmf (l, u' ⊕ 0.5) (is - = ?μ)
      unfolding rept-def by auto
      from upper-right-closed[OF True] ∗ ‹R' ∈ ℛ› T have u' ⊕ 0.5 ∈ R' by auto
      with elapsed ‹rept - - = -› show ?thesis by auto
    next
      case False
      with T ∗ R prems(1) have
        rept s μ = (SOME μ'. μ' ∈ K s ∧ abst μ' = μ)
      unfolding rept-def by auto
      with default show ?thesis by simp (rule someI; auto)
    qed
  next
    case F: False
    show ?thesis
    proof (cases ∀ u ∈ R'. ∀ c ∈ 𝒳. ∄ d. d ≤ k c ∧ u c = real d)
      case False
      with F ∗ R prems(1) have
        rept s μ = (SOME μ'. μ' ∈ K s ∧ abst μ' = μ)
      unfolding rept-def by auto
      with default show ?thesis by simp (rule someI; auto)
    next
      case True
      from True F ∗ R prems(1,4) ‹s ∈ S› have
        rept s μ = return-pmf (l, delayedR (SOME R'. μ = return-pmf (l, R')) u')
        (is - = return-pmf (l, delayedR ?R u'))
      unfolding rept-def by auto
      let ?u = delayedR ?R u'
      from prems(1) have μ = return-pmf (l, ?R) by auto
      with prems(1) have ?R = R' by auto
      moreover from R' True ‹- ∈ R'› have ∀ c∈𝒳. ¬ Regions.isConst (I c) by fastforce
      moreover note delayedR-correct[of u' R I r] ∗ ‹R ∈ ℛ› R' True ‹R' ∈ Succ ℛ R›
      ultimately obtain t where ∗∗: delayedR R' u' ∈ R' t ≥ 0 delayedR R' u' = u' ⊕ t by auto
      moreover from ‹?R =- ›‹rept - - = -› have rept s μ = return-pmf (l, delayedR R' u') by auto
      ultimately show ?thesis using elapsed by auto
    qed
  qed
next
```

```
case prems: (action l R τ μ′)
from abss-𝒮D′[OF prems(2,3)] obtain u where u:
  s = (l, u) u ∈ [u]ℛ [u]ℛ ∈ ℛ R = [u]ℛ
by auto
with ‹- ∈ 𝒮› have (l, u) ∈ S by (auto intro: 𝒮-abss-S)
let ?μ = map-pmf (λ(X, l). (l, [X:=0]u)) μ′
from u prems have ?μ ∈ K s by (fastforce intro: 𝒮-abss-S)
moreover have abst ?μ = μ unfolding prems(1) abst-def
 proof (subst map-pmf-comp, rule pmf.map-cong, safe, goal-cases)
   case A: (1 X l′)
   from u have u ∈ V using ℛ-V by auto
   then have [X:=0]u ∈ V by auto
   from prems(1) A
   have (l′, region-set′ R (SOME r. set r = X) 0) ∈ μ by auto
   from A prems R-G.K-closed ‹μ ∈ -› have
     l′ ∈ L region-set′ R (SOME r. set r = X) 0 ⊆ {|inv-of A l′|}
   by (force dest: 𝒮-L 𝒮-inv)+
   with u have [X:=0]u ⊢ inv-of A l′ unfolding region-set′-def by auto
   with ‹l′ ∈ L› ‹[X:=0]u ∈ V› have (l′, [X:=0]u) ∈ S unfolding S-def V-def by auto
   then have abss (l′, [X:=0]u) = (l′, [[X:=0]u]ℛ) by auto
   also have
     ... = (l′, region-set′ R (SOME r. set r = X) 0)
   using region-set′-eq(1)[unfolded transition-def] prems A u by force
  finally show ?case .
qed
ultimately have default: ?thesis if rept s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ) using that
by simp (rule someI; auto)
show ?thesis
proof (cases ∃ R. μ = return-pmf (l, R))
  case False
  with ‹s = (l, u)› have rept s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ) unfolding rept-def by auto
  with default show ?thesis by auto
next
  case True
  then obtain R′ where R′: μ = return-pmf (l, R′) by auto
  show ?thesis
  proof (cases R = R′)
    case False
    from R′ prems(1) have
      ∀ (X, l′) ∈ μ′. (l′, region-set′ R (SOME r. set r = X) 0) = (l, R′)
    by (auto simp: map-pmf-eq-return-pmf-iff[of - μ′ (l, R′)])
    then obtain X where
      region-set′ R (SOME r. set r = X) 0 = R′ (X, l) ∈ μ′
    using set-pmf-not-empty by force
    with prems(4) have X ⊆ 𝒳 by (simp add: admissible-targets-clocks(1))
    moreover then have
      set (SOME r. set r = X) = X
    by − (rule someI-ex, metis finite-list finite(1) finite-subset)
    ultimately have set (SOME r. set r = X) ⊆ 𝒳 by auto
    with alpha-interp.region-reset-not-Succ False ‹- = R′› u(3,4) have R′ ∉ Succ ℛ R by auto
    with ‹s = (l, u)› R′ u(4) False have
      rept s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ)
    unfolding rept-def by auto
    with default show ?thesis by auto
  next
    case T: True
    show ?thesis
    proof (cases ∀ c∈𝒳. real (k c) < u c)
      case False
      with T ‹s = (l, u)› R′ u(4) have
        rept s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ)
```

```
      unfolding rept-def by auto
      with default show ?thesis by auto
    next
      case True
      with T ‹s = (l, u)› R′ u(4) ‹(l, u) ∈ S› have
        rept s μ = return-pmf (l, u ⊕ 0.5)
      unfolding rept-def by auto
      from upper-right-closed[OF True] T u ℛ-V have u ⊕ 0.5 ∈ R′ u ⊕ 0.5 ∈ V by force+
      moreover then have [u ⊕ 0.5]ℛ = R′
        using T alpha-interp.region-unique-spec u(3,4) by blast
      moreover note ∗ = ‹rept - - = -› R′ ‹abss s ∈ 𝒮› ‹abss s = -› prems(5)
      ultimately have abst (rept s μ) = μ
       apply (simp add: abst-def)
       apply (subst abss-S)
      by (auto simp: 𝒮-L S-def V-def T dest: 𝒮-inv)
      moreover from ∗ ‹s = -› ‹(l, u) ∈ S› ‹- ∈ R′› have
        rept s μ ∈ K s
       apply simp
       apply (rule K.delay)
      by (auto simp: T dest: 𝒮-inv)
      ultimately show ?thesis by auto
    qed
  qed
 qed
next
 case loop
 obtain l u where s = (l, u) by force
 show ?thesis
 proof (cases s ∈ S)
  case T: True
  with ‹s = -› have ∗: l ∈ L u ∈ [u]ℛ [u]ℛ ∈ ℛ abss s = (l, [u]ℛ) by auto
  then have abss s = (l, [u]ℛ) by auto
  with ‹s ∈ S› S-abss-𝒮 have (l, [u]ℛ) ∈ 𝒮 by auto
  with 𝒮-inv have [u]ℛ ⊆ {u. u ⊢ inv-of A l} by auto
  show ?thesis
  proof (cases ∀ c∈𝒳. real (k c) < u c)
    case True
    with ∗ ‹μ = -› ‹s = -› ‹s ∈ S› have
      rept s μ = return-pmf (l, u ⊕ 0.5)
    unfolding rept-def by auto
    from upper-right-closed[OF True] ∗ have u ⊕ 0.5 ∈ [u]ℛ by auto
    moreover with ∗ ℛ-V have u ⊕ 0.5 ∈ V by auto
    moreover with calculation ∗ alpha-interp.region-unique-spec have [u ⊕ 0.5]ℛ = [u]ℛ by blast
    moreover note ∗ ‹rept - - = -› ‹s = -› T ‹μ = -› ‹(l, -) ∈ 𝒮› 𝒮-inv
    ultimately show ?thesis unfolding rept-def
     apply simp
     apply safe
      apply fastforce
     apply (simp add: abst-def)
     apply (subst abst-def abss-S)
    by fastforce+
  next
    case False
    with ∗ ‹s = -› ‹μ = -› have
      rept s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ)
    unfolding rept-def by auto
    with ‹μ = -› show ?thesis by simp (rule someI[where x = return-pmf s], auto simp: abst-def)
  qed
 next
  case False
  with ‹s = -› ‹μ = -› have
```

$rept\ s\ \mu = (SOME\ \mu'.\ \mu' \in K\ s \wedge abst\ \mu' = \mu)$
unfolding *rept-def* **by** *auto*
with ‹μ = -› **show** *?thesis* **by** *simp* (*rule someI*[**where** x = *return-pmf s*], *auto simp*: *abst-def*)
qed
qed

lemmas *rept-K*[*intro*] = *rept-ex*[*THEN conjunct1*]
lemmas *abst-rept-id*[*simp*] = *rept-ex*[*THEN conjunct2*]

lemma *abst-rept2*:
assumes $\mu \in \mathcal{K}\ s\ s \in \mathcal{S}$
shows $abst\ (rept\ (reps\ s)\ \mu) = \mu$
using *assms* **by** *auto*

lemma *rept-K2*:
assumes $\mu \in \mathcal{K}\ s\ s \in \mathcal{S}$
shows $rept\ (reps\ s)\ \mu \in K\ (reps\ s)$
using *assms* **by** *auto*

lemma *theI'*:
assumes $P\ a$
and $\bigwedge x.\ P\ x \Longrightarrow x = a$
shows $P\ (THE\ x.\ P\ x) \wedge (\forall\ y.\ P\ y \longrightarrow y = (THE\ x.\ P\ x))$
using *theI assms* **by** *metis*

lemma *cont-cfg-defined*:
fixes *cfg s*
assumes $cfg \in valid\text{-}cfg\ s \in abst\ (action\ cfg)$
defines $x \equiv THE\ x.\ abss\ x = s \wedge x \in action\ cfg$
shows $(abss\ x = s \wedge x \in action\ cfg) \wedge (\forall\ y.\ abss\ y = s \wedge y \in action\ cfg \longrightarrow y = x)$
proof −
from *assms*(*2*) **obtain** s' **where** $s' \in action\ cfg\ s = abss\ s'$ **unfolding** *abst-def* **by** *auto*
with *assms* **show** *?thesis* **unfolding** *x-def*
by −(*rule theI'*[*of - s'*],*auto intro*: *K-bisim-unique MDP.valid-cfg-state-in-S dest*: *MDP.valid-cfgD*)
qed

definition
$absc' :: ('s * ('c,\ t)\ cval)\ cfg \Rightarrow ('s * ('c,\ t)\ cval\ set)\ cfg$
where
$absc'\ cfg = cfg\text{-}corec$
$(abss\ (state\ cfg))$
$(abst\ o\ action)$
$(\lambda\ cfg\ s.\ cont\ cfg\ (THE\ x.\ abss\ x = s \wedge x \in action\ cfg))\ cfg$

5.2.5 Configuration

definition
$absc :: ('s * ('c,\ t)\ cval)\ cfg \Rightarrow ('s * ('c,\ t)\ cval\ set)\ cfg$
where
$absc\ cfg = cfg\text{-}corec$
$(abss\ (state\ cfg))$
$(abst\ o\ action)$
$(\lambda\ cfg\ s.\ cont\ cfg\ (THE\ x.\ abss\ x = s \wedge x \in action\ cfg))\ cfg$

definition
$repcs :: 's * ('c,\ t)\ cval \Rightarrow ('s * ('c,\ t)\ cval\ set)\ cfg \Rightarrow ('s * ('c,\ t)\ cval)\ cfg$
where
$repcs\ s\ cfg = cfg\text{-}corec$
s
$(\lambda\ (s,\ cfg).\ rept\ s\ (action\ cfg))$

$(\lambda\ (s,\ cfg)\ s'.\ (s',\ cont\ cfg\ (abss\ s')))\ (s,\ cfg)$

definition
repc cfg = *repcs* (*reps* (*state cfg*)) *cfg*

lemma *𝒮-state-absc-repc*[*simp*]:
state cfg ∈ 𝒮 ⟹ *state* (*absc* (*repc cfg*)) = *state cfg*
by (*simp add*: *absc-def repc-def repcs-def*)

lemma *action-repc*:
action (*repc cfg*) = *rept* (*reps* (*state cfg*)) (*action cfg*)
unfolding *repc-def repcs-def* **by** *simp*

lemma *action-absc*:
action (*absc cfg*) = *abst* (*action cfg*)
unfolding *absc-def* **by** *simp*

lemma *action-absc′*:
action (*absc cfg*) = *map-pmf abss* (*action cfg*)
unfolding *absc-def* **unfolding** *abst-def* **by** *simp*

lemma
notes *R-G.cfg-onD-state*[*simp del*]
assumes *state cfg* ∈ 𝒮 *s′* ∈ *set-pmf* (*action* (*repc cfg*)) *cfg* ∈ *R-G.cfg-on* (*state cfg*)
shows *cont* (*repc cfg*) *s′* = *repcs s′* (*cont cfg* (*abss s′*))
using *assms* **by** (*auto simp*: *repc-def repcs-def abss-reps-id*)

lemma *cont-repcs1*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes *abss s* ∈ 𝒮 *s′* ∈ *set-pmf* (*action* (*repcs s cfg*)) *cfg* ∈ *R-G.cfg-on* (*abss s*)
shows *cont* (*repcs s cfg*) *s′* = *repcs s′* (*cont cfg* (*abss s′*))
using *assms* **by** (*auto simp*: *repc-def repcs-def abss-reps-id*)

lemma *cont-absc-1*:
notes *MDP.cfg-onD-state*[*simp del*]
assumes *cfg* ∈ *valid-cfg s′* ∈ *set-pmf* (*action cfg*)
shows *cont* (*absc cfg*) (*abss s′*) = *absc* (*cont cfg s′*)
proof −
define *x* **where** *x* ≡ *THE x*. *x* ~ *s′* ∧ *x* ∈ *set-pmf* (*action cfg*)
from *assms*(*2*) **have** *abss s′* ∈ *set-pmf* (*abst* (*action cfg*)) **unfolding** *abst-def* **by** *auto*
from *cont-cfg-defined*[*OF assms*(*1*) *this*] **have**
(*x* ~ *s′* ∧ *x* ∈ *set-pmf* (*action cfg*)) ∧ (∀ *y*. *y* ~ *s′* ∧ *y* ∈ *set-pmf* (*action cfg*) ⟶ *y* = *x*)
unfolding *x-def* **.**
with *assms* **have** *s′* = *x* **by** *fastforce*
then show *?thesis*
unfolding *absc-def abst-def repc-def x-def* **using** *assms*(*2*) **by** *auto*
qed

lemma *state-repc*:
state (*repc cfg*) = *reps* (*state cfg*)
unfolding *repc-def repcs-def* **by** *simp*

lemma *abss-reps-id′*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes *cfg* ∈ *R-G.valid-cfg s* ∈ *set-pmf* (*action cfg*)
shows *abss* (*reps s*) = *s*
using *assms* **by** (*auto intro*: *abss-reps-id R-G.valid-cfg-state-in-S R-G.valid-cfgD*)

lemma *valid-cfg-coinduct*[*coinduct set*: *valid-cfg*]:

assumes $P\ cfg$
assumes $\bigwedge cfg.\ P\ cfg \Longrightarrow state\ cfg \in S$
assumes $\bigwedge cfg.\ P\ cfg \Longrightarrow action\ cfg \in K\ (state\ cfg)$
assumes $\bigwedge cfg\ t.\ P\ cfg \Longrightarrow t \in action\ cfg \Longrightarrow P\ (cont\ cfg\ t)$
shows $cfg \in valid\text{-}cfg$
proof −
from *assms* **have** $cfg \in MDP.cfg\text{-}on\ (state\ cfg)$ **by** (*coinduction arbitrary*: *cfg*) *auto*
moreover from *assms* **have** $state\ cfg \in S$ **by** *auto*
ultimately show *?thesis* **by** (*intro MDP.valid-cfgI*)
qed

lemma *state-repcD*[*simp*]:
assumes $cfg \in R\text{-}G.cfg\text{-}on\ s$
shows $state\ (repc\ cfg) = reps\ s$
using *assms* **unfolding** *repc-def repcs-def* **by** *auto*

lemma *ccompatible-subs*[*intro*]:
assumes $ccompatible\ \mathcal{R}\ g\ R \in \mathcal{R}\ u \in R\ u \vdash g$
shows $R \subseteq \{u.\ u \vdash g\}$
using *assms* **unfolding** *ccompatible-def* **by** *auto*

lemma *action-abscD*[*dest*]:
$cfg \in MDP.cfg\text{-}on\ s \Longrightarrow action\ (absc\ cfg) \in \mathcal{K}\ (abss\ s)$
unfolding *absc-def abst-def*
proof *simp*
assume *cfg*: $cfg \in MDP.cfg\text{-}on\ s$
then have $action\ cfg \in K\ s$ **by** *auto*
then show $map\text{-}pmf\ abss\ (action\ cfg) \in \mathcal{K}\ (abss\ s)$
proof *cases*
case *prems*: $(delay\ l\ u\ t)$
then have $[u \oplus t]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
moreover with *prems ccompatible-inv*[*of l*] **have**
$[u \oplus t]_{\mathcal{R}} \subseteq \{v.\ v \vdash PTA.inv\text{-}of\ A\ l\}$
unfolding *ccompatible-def* **by** *force*
moreover from *prems* **have** $abss\ (l,\ u \oplus t) = (l,\ [u \oplus t]_{\mathcal{R}})$ **by** (*subst abss-S*) *auto*
ultimately show *?thesis* **using** *prems* **by** *auto*
next
case *prems*: $(action\ l\ u\ g\ \mu)$
then have $[u]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
moreover with *prems ccompatible-guard* **have** $[u]_{\mathcal{R}} \subseteq \{u.\ u \vdash g\}$
by (*intro ccompatible-subs*) *auto*
moreover have
$map\text{-}pmf\ abss\ (action\ cfg)$
$= map\text{-}pmf\ (\lambda(X,\ l).\ (l,\ region\text{-}set'\ ([u]_{\mathcal{R}})\ (SOME\ r.\ set\ r = X)\ 0))\ \mu$
proof −
have $abss\ (l',\ [X{:=}0]u) = (l',\ region\text{-}set'\ ([u]_{\mathcal{R}})\ (SOME\ r.\ set\ r = X)\ 0)$
if $(X,\ l') \in \mu$ **for** $X\ l'$
proof −
from *that prems* **have** $A \vdash l \longrightarrow^{g,\mu,X} l'$
by *auto*
from *that prems MDP.action-closed*[*OF - cfg*] **have** $(l',\ [X{:=}0]u) \in S$ **by** *force*
then have $abss\ (l',\ [X{:=}0]u) = (l',\ [[X{:=}0]u]_{\mathcal{R}})$ **by** *auto*
also have
$\ldots = (l',\ region\text{-}set'\ ([u]_{\mathcal{R}})\ (SOME\ r.\ set\ r = X)\ 0)$
using *region-set'-eq*(*1*)[*OF - -* ‹$A \vdash l \longrightarrow^{g,\mu,X} l'$›] *prems* **by** *auto*
finally show *?thesis* **.**
qed
then show *?thesis*
unfolding *prems*(*1*)

```
      by (auto intro: pmf.map-cong simp: map-pmf-comp)
    qed
    ultimately show ?thesis using prems by auto
  next
    case prems: loop
    then show ?thesis by auto
  qed
qed

lemma repcs-valid[intro]:
  assumes cfg ∈ R-G.valid-cfg abss s = state cfg
  shows repcs s cfg ∈ valid-cfg
using assms
proof (coinduction arbitrary: cfg s)
  case 1
  then show ?case
  by (auto simp: repcs-def S-abss-S dest: R-G.valid-cfg-state-in-S)
next
  case (2 cfg' s)
  then show ?case
   by (simp add: repcs-def) (rule rept-K, auto dest: R-G.valid-cfgD)
next
  case prems: (3 s' cfg)
  let ?cfg = cont cfg (abss s')
  from prems have abss s' ∈ abst (rept s (action cfg)) unfolding repcs-def abst-def by auto
  with prems have
    abss s' ∈ action cfg
  by (subst (asm) abst-rept-id) (auto dest: R-G.valid-cfgD)
  with prems show ?case
    by (inst-existentials ?cfg s', subst cont-repcs1)
       (auto dest: R-G.valid-cfg-state-in-S intro: R-G.valid-cfgD R-G.valid-cfg-cont)
qed

lemma repc-valid[intro]:
  assumes cfg ∈ R-G.valid-cfg
  shows repc cfg ∈ valid-cfg
using assms unfolding repc-def by (force dest: R-G.valid-cfg-state-in-S)

lemma action-abst-repcs:
  assumes cfg ∈ R-G.valid-cfg abss s = state cfg
  shows abst (action (repcs s cfg)) = action cfg
proof −
  from assms show ?thesis
  unfolding repc-def repcs-def
   apply simp
   apply (subst abst-rept-id)
  by (auto dest: R-G.cfg-onD-action R-G.valid-cfgD)
qed

lemma action-abst-repc:
  assumes cfg ∈ R-G.valid-cfg
  shows abst (action (repc cfg)) = action cfg
proof −
  from assms have abss (reps (state cfg)) = state cfg by (auto dest: R-G.valid-cfg-state-in-S)
  with action-abst-repcs[OF assms] show ?thesis unfolding repc-def by auto
qed


lemma state-absc:
  state (absc cfg) = abss (state cfg)
unfolding absc-def by auto
```

lemma *state-repcs*[*simp*]:
state (*repcs s cfg*) = *s*
unfolding *repcs-def* **by** *auto*

lemma *repcs-bisim*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes *cfg* ∈ *R-G.valid-cfg x* ∈ *S x* ~ *x′ abss x* = *state cfg*
shows *absc* (*repcs x cfg*) = *absc* (*repcs x′ cfg*)
using *assms*
proof −
from *assms* **have** *abss x′* = *state cfg* **by** *auto*
from *assms* **have** *abss x′* ∈ 𝒮 **by** *auto*
then have *x′* ∈ *S* **by** (*auto intro*: 𝒮*-abss-S*)
with *assms* **show** *?thesis*
proof (*coinduction arbitrary*: *cfg x x′*)
case *state*
then show *?case* **by** (*simp add*: *state-absc*)
next
case *action*
then show *?case* **unfolding** *absc-def repcs-def* **by** (*auto dest*: *R-G.valid-cfgD*)
next
case *prems*: (*cont s cfg x x′*)
define *cfg′* **where** *cfg′* = *cont cfg s*
define *t* **where** *t* ≡ *THE y*. *abss y* = *s* ∧ *y* ∈ *action* (*repcs x cfg*)
define *t′* **where** *t′* ≡*THE y*. *abss y* = *s* ∧ *y* ∈ *action* (*repcs x′ cfg*)
from *prems* **have** *valid*: *repcs x cfg* ∈ *valid-cfg* **by** (*intro repcs-valid*)
from *prems* **have** ∗: *s* ∈ *abst* (*action* (*repcs x cfg*))
unfolding *cfg′-def* **by** (*simp add*: *action-absc*)
with *prems* **have** *s* ∈ *action cfg* **by** (*auto dest*: *R-G.valid-cfgD simp*: *repcs-def*)
with *prems* **have** *s* ∈ 𝒮 **by** (*auto intro*: *R-G.valid-cfg-action*)
from *cont-cfg-defined*[*OF valid* ∗] **have** *t*:
abss t = *s t* ∈ *action* (*repcs x cfg*)
unfolding *t-def* **by** *auto*
have *cont* (*absc* (*repcs x cfg*)) *s* = *cont* (*absc* (*repcs x cfg*)) (*abss t*) **using** *t* **by** *auto*
have *cont* (*absc* (*repcs x cfg*)) *s* = *absc* (*cont* (*repcs x cfg*) *t*)
using *t valid* **by** (*auto simp*: *cont-absc-1*)
also have . . . = *absc* (*repcs t* (*cont cfg s*))
using *prems t* **by** (*subst cont-repcs1*) (*auto dest*: *R-G.valid-cfgD*)
finally have *cont-x*: *cont* (*absc* (*repcs x cfg*)) *s* = *absc* (*repcs t* (*cont cfg s*)) **.**
from *prems* **have** *valid*: *repcs x′ cfg* ∈ *valid-cfg* **by** *auto*
from ‹*s* ∈ *action cfg*› *prems* **have** *s* ∈ *abst* (*action* (*repcs x′ cfg*))
by (*auto dest*: *R-G.valid-cfgD simp*: *repcs-def*)
from *cont-cfg-defined*[*OF valid this*] **have** *t′*:
abss t′ = *s t′* ∈ *action* (*repcs x′ cfg*)
unfolding *t′-def* **by** *auto*
have *cont* (*absc* (*repcs x′ cfg*)) *s* = *cont* (*absc* (*repcs x′ cfg*)) (*abss t′*) **using** *t′* **by** *auto*
have *cont* (*absc* (*repcs x′ cfg*)) *s* = *absc* (*cont* (*repcs x′ cfg*) *t′*)
using *t′ valid* **by** (*auto simp*: *cont-absc-1*)
also have . . . = *absc* (*repcs t′* (*cont cfg s*))
using *prems t′* **by** (*subst cont-repcs1*) (*auto dest*: *R-G.valid-cfgD*)
finally have *cont* (*absc* (*repcs x′ cfg*)) *s* = *absc* (*repcs t′* (*cont cfg s*)) **.**
with *cont-x* ‹*s* ∈ *action cfg*› *prems*(*1*) *t t′* ‹*s* ∈ 𝒮›
show *?case*
by (*inst-existentials cont cfg s t t′*)
(*auto intro*: 𝒮*-abss-S R-G.valid-cfg-action R-G.valid-cfg-cont*)
qed
qed

named-theorems *R-G-I*

lemmas *R-G.valid-cfg-state-in-S*[*R-G-I*] *R-G.valid-cfgD*[*R-G-I*] *R-G.valid-cfg-action*

lemma *absc-repcs-id*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes *cfg* ∈ *R-G.valid-cfg abss s* = *state cfg*
shows *absc* (*repcs s cfg*) = *cfg* **using** *assms*
proof (*subst eq-commute*, *coinduction arbitrary*: *cfg s*)
case *state*
then show *?case* **by** (*simp add*: *absc-def repc-def repcs-def*)
next
case *prems*: (*action cfg*)
then show *?case* **by** (*auto simp*: *action-abst-repcs action-absc*)
next
case *prems*: (*cont s'*)
define *cfg'* **where** *cfg'* ≡ *repcs s cfg*
define *t* **where** *t* ≡ *THE x*. *abss x* = *s'* ∧ *x* ∈ *set-pmf* (*action cfg'*)
from *prems* **have** *cfg* ∈ *R-G.cfg-on* (*state cfg*) *state cfg* ∈ 𝒮 **by** (*auto dest*: *R-G-I*)
then have ∗: *cfg* ∈ *R-G.cfg-on* (*abss* (*reps* (*state cfg*))) *abss* (*reps* (*state cfg*)) ∈ 𝒮 **by** *auto*
from *prems* **have** *s'* ∈ 𝒮 **by** (*auto intro*: *R-G.valid-cfg-action*)
from *prems* **have** *valid*: *cfg'* ∈ *valid-cfg* **unfolding** *cfg'-def* **by** (*intro repcs-valid*)
from *prems* **have** *s'* ∈ *abst* (*action cfg'*) **unfolding** *cfg'-def* **by** (*subst action-abst-repcs*)
from *cont-cfg-defined*[*OF valid this*] **have** *t*:
abss t = *s' t* ∈ *action cfg'*
unfolding *t-def cfg'-def* **by** *auto*
with *prems* **have** *t* ~ *reps* (*abss t*)
apply −
apply (*subst* 𝒮*-abss-reps*)
by (*auto intro*: *R-G.valid-cfg-action*)
have *cont* (*absc cfg'*) *s'* = *cont* (*absc cfg'*) (*abss t*) **using** *t* **by** *auto*
have *cont* (*absc cfg'*) *s'* = *absc* (*cont cfg' t*) **using** *t valid* **by** (*auto simp*: *cont-absc-1*)
also have ... = *absc* (*repcs t* (*cont cfg s'*)) **using** *prems t* ∗ ‹*t* ~ -› *valid*
by (*fastforce dest*: *R-G-I intro*: *repcs-bisim simp*: *cont-repcs1 cfg'-def*)
finally show *?case*
apply −
apply (*rule exI*[**where** *x* = *cont cfg s'*], *rule exI*[**where** *x* = *t*])
unfolding *cfg'-def* **using** *prems t* **by** (*auto intro*: *R-G.valid-cfg-cont*)
qed

lemma *absc-repc-id*:
notes *R-G.cfg-onD-state*[*simp del*]
assumes *cfg* ∈ *R-G.valid-cfg*
shows *absc* (*repc cfg*) = *cfg* **using** *assms*
unfolding *repc-def* **using** *assms* **by** (*subst absc-repcs-id*) (*auto dest*: *R-G-I*)

lemma *K-cfg-map-absc*:
cfg ∈ *valid-cfg* ⟹ *K-cfg* (*absc cfg*) = *map-pmf absc* (*K-cfg cfg*)
by (*auto simp*: *K-cfg-def map-pmf-comp action-absc abst-def cont-absc-1 intro*: *map-pmf-cong*)

lemma *smap-comp*:
(*smap f o smap g*) = *smap* (*f o g*)
by (*auto simp*: *stream.map-comp*)

lemma *state-abscD*[*simp*]:
assumes *cfg* ∈ *MDP.cfg-on s*
shows *state* (*absc cfg*) = *abss s*
using *assms* **unfolding** *absc-def* **by** *auto*

lemma *R-G-valid-cfg-coinduct*[*coinduct set*: *valid-cfg*]:
  assumes *P cfg*
  assumes $\bigwedge$*cfg*. *P cfg* $\Longrightarrow$ *state cfg* $\in \mathcal{S}$
  assumes $\bigwedge$*cfg*. *P cfg* $\Longrightarrow$ *action cfg* $\in \mathcal{K}$ (*state cfg*)
  assumes $\bigwedge$*cfg t*. *P cfg* $\Longrightarrow$ *t* $\in$ *action cfg* $\Longrightarrow$ *P* (*cont cfg t*)
  shows *cfg* $\in$ *R-G.valid-cfg*
proof $-$
  from *assms* **have** *cfg* $\in$ *R-G.cfg-on* (*state cfg*) **by** (*coinduction arbitrary*: *cfg*) *auto*
  moreover from *assms* **have** *state cfg* $\in \mathcal{S}$ **by** *auto*
  ultimately show *?thesis* **by** (*intro R-G.valid-cfgI*)
qed

lemma *absc-valid*[*intro*]:
  assumes *cfg* $\in$ *valid-cfg*
  shows *absc cfg* $\in$ *R-G.valid-cfg*
using *assms*
proof (*coinduction arbitrary*: *cfg*)
  case *1*
  then show *?case* **by** (*auto simp*: *absc-def dest*: *MDP.valid-cfg-state-in-S*)
next
  case (*2 cfg'*)
  then show *?case* **by** (*subst state-abscD*) (*auto intro*: *MDP.valid-cfgD action-abscD*)
next
  case *prems*: (*3 s' cfg*)
  define *t* **where** *t* $\equiv$ *THE x*. *abss x* $=$ *s'* $\wedge$ *x* $\in$ *set-pmf* (*action cfg*)
  let *?cfg* $=$ *cont cfg t*
  from *prems* **obtain** *s* **where** *s'* $=$ *abss s s* $\in$ *action cfg* **by** (*auto simp*: *action-absc'*)
  with *cont-cfg-defined*[*OF prems*(*1*), *of s'*] **have**
    abss t $=$ *s' t* $\in$ *set-pmf* (*action cfg*)
    $\forall$ *y*. *abss y* $=$ *s'* $\wedge$ *y* $\in$ *set-pmf* (*action cfg*) $\longrightarrow$ *y* $=$ *t*
  unfolding *t-def abst-def* **by** *auto*
  with *prems* **show** *?case*
    by (*inst-existentials ?cfg*)
      (*auto intro*: *MDP.valid-cfg-cont simp*: *abst-def action-absc absc-def t-def*)
qed

lemma *K-cfg-set-absc*:
  assumes *cfg* $\in$ *valid-cfg cfg'* $\in$ *K-cfg cfg*
  shows *absc cfg'* $\in$ *K-cfg* (*absc cfg*)
using *assms* **by** (*auto simp*: *K-cfg-map-absc*)

lemma *abst-action-repcs*:
  assumes *cfg* $\in$ *R-G.valid-cfg abss s* $=$ *state cfg*
  shows *abst* (*action* (*repcs s cfg*)) $=$ *action cfg*
unfolding *repc-def repcs-def* **using** *assms* **by** (*simp*, *subst abst-rept-id*) (*auto intro*: *R-G-I*)

lemma *abst-action-repc*:
  assumes *cfg* $\in$ *R-G.valid-cfg*
  shows *abst* (*action* (*repc cfg*)) $=$ *action cfg*
using *assms* **unfolding** *repc-def* **by** (*auto intro*: *abst-action-repcs simp*: *R-G-I*)

lemma *K-elem-abss-inj'*:
  assumes $\mu \in$ *K s*
    and *s* $\in$ *S*
  shows *inj-on abss* (*set-pmf* μ)
using *assms K-elem-abss-inj* **by** (*simp add*: *K-bisim-unique inj-onI*)

lemma *K-cfg-rept-aux*:
  assumes *cfg* $\in$ *R-G.valid-cfg abss s* $=$ *state cfg x* $\in$ *rept s* (*action cfg*)
  defines *t* $\equiv \lambda$ *cfg'*. *THE s'*. *s'* $\in$ *rept s* (*action cfg*) $\wedge$ *s'* $\sim$ *x*
  shows *t cfg'* $=$ *x*

```
proof –
  from assms have rept s (action cfg) ∈ K s s ∈ S by (auto simp: R-G-I 𝒮-abss-S)
  from K-bisim-unique[OF this(2,1) - assms(3)] assms(3) show ?thesis unfolding t-def by blast
qed

lemma K-cfg-rept-action:
  assumes cfg ∈ R-G.valid-cfg abss s = state cfg cfg′ ∈ set-pmf (K-cfg cfg)
  shows abss (THE s′. s′ ∈ rept s (action cfg) ∧ abss s′ = state cfg′) = state cfg′
proof –
  let ?μ = rept s (action cfg)
  from abst-rept-id assms have action cfg = abst ?μ by (auto simp: R-G-I)
  moreover from assms have state cfg′ ∈ action cfg by (auto simp: set-K-cfg)
  ultimately have state cfg′ ∈ abst ?μ by simp
  then obtain s′ where s′ ∈ ?μ abss s′ = state cfg′ by (auto simp: abst-def pmf.set-map)
  with K-cfg-rept-aux[OF assms(1,2) this(1)] show ?thesis by auto
qed

lemma K-cfg-map-repcs:
  assumes cfg ∈ R-G.valid-cfg abss s = state cfg
  defines repc′ ≡ (λ cfg′. repcs (THE s′. s′ ∈ rept s (action cfg) ∧ abss s′ = state cfg′) cfg′)
  shows K-cfg (repcs s cfg) = map-pmf repc′ (K-cfg cfg)
proof –
  let ?μ = rept s (action cfg)
  define t where t ≡ λ cfg′. THE s. s ∈ ?μ ∧ abss s = state cfg′
  have t: t (cont cfg (abss s′)) = s′ if s′ ∈ ?μ for s′
    using K-cfg-rept-aux[OF assms(1,2) that] unfolding t-def by auto
  show ?thesis
    unfolding K-cfg-def using t
    by (subst abst-action-repcs[symmetric])
      (auto simp: repc-def repcs-def t-def map-pmf-comp abst-def assms intro: map-pmf-cong)
qed

lemma K-cfg-map-repc:
  assumes cfg ∈ R-G.valid-cfg
  defines
    repc′ cfg′ ≡ repcs (THE s. s ∈ rept (reps (state cfg)) (action cfg) ∧ abss s = state cfg′) cfg′
  shows
    K-cfg (repc cfg) = map-pmf repc′ (K-cfg cfg)
using assms unfolding repc′-def repc-def by (auto simp: R-G-I K-cfg-map-repcs)

lemma R-G-K-cfg-valid-cfgD:
  assumes cfg ∈ R-G.valid-cfg cfg′ ∈ K-cfg cfg
  shows cfg′ = cont cfg (state cfg′) state cfg′ ∈ action cfg
proof –
  from assms(2) obtain s where s ∈ action cfg cfg′ = cont cfg s by (auto simp: set-K-cfg)
  with assms show
    cfg′ = cont cfg (state cfg′) state cfg′ ∈ action cfg
  by (auto intro: R-G.valid-cfg-state-in-S R-G.valid-cfgD)
qed

lemma K-cfg-valid-cfgD:
  assumes cfg ∈ valid-cfg cfg′ ∈ K-cfg cfg
  shows cfg′ = cont cfg (state cfg′) state cfg′ ∈ action cfg
proof –
  from assms(2) obtain s where s ∈ action cfg cfg′ = cont cfg s by (auto simp: set-K-cfg)
  with assms show
    cfg′ = cont cfg (state cfg′) state cfg′ ∈ action cfg
  by auto
qed
```

```
lemma absc-bisim-abss:
  assumes absc x = absc x′
  shows state x ∼ state x′
proof −
  from assms have state (absc x) = state (absc x′) by simp
  then show ?thesis by (simp add: state-absc)
qed


lemma K-cfg-bisim-unique:
  assumes cfg ∈ valid-cfg and x ∈ K-cfg cfg x′ ∈ K-cfg cfg and state x ∼ state x′
  shows x = x′
proof −
  define t where t ≡ THE x′. x′ ∼ state x ∧ x′ ∈ set-pmf (action cfg)
  from K-cfg-valid-cfgD assms have ∗:
    x  = cont cfg (state x)  state x ∈ action cfg
    x′ = cont cfg (state x′) state x′ ∈ action cfg
    by auto
  with assms have
    cfg ∈ valid-cfg abss (state x) ∈ set-pmf (abst (action cfg))
    unfolding abst-def by auto
  with cont-cfg-defined[of cfg abss (state x)] have
    ∀ y. y ∼ state x ∧ y ∈ set-pmf (action cfg) ⟶ y = t
    unfolding t-def by auto
  with ∗ assms(4) have state x′ = t state x = t by fastforce+
  with ∗ show ?thesis by simp
qed

lemma absc-distr-self:
  MDP.MC.T (absc cfg) = distr (MDP.MC.T cfg) MDP.MC.S (smap absc) if cfg ∈ valid-cfg
  using ‹cfg ∈ -›
proof (coinduction arbitrary: cfg rule: MDP.MC.T-coinduct)
  case prob
  show ?case by (rule MDP.MC.T.prob-space-distr, simp)
next
  case sets
  show ?case by auto
next
  case prems: (cont cfg)
  define t  where t  ≡ λ y. THE x. y = absc x ∧ x ∈ K-cfg cfg
  define M′ where M′ ≡ λ cfg. distr (MDP.MC.T (t cfg)) MDP.MC.S (smap absc)
  show ?case
  proof (rule exI[where x = M′], safe, goal-cases)
    case A: (1 y)
    from A prems obtain x′ where y = absc x′ x′ ∈ K-cfg cfg by (auto simp: K-cfg-map-absc)
    with K-cfg-bisim-unique[OF prems - - absc-bisim-abss] have
      y = absc (t y) x′ = t y
      unfolding t-def by (auto intro: theI2)
    moreover have x′ ∈ valid-cfg using ‹x′ ∈ -› prems by auto
    ultimately show ?case unfolding M′-def by auto
  next
    case 5
    show ?case unfolding M′-def
      apply (subst distr-distr)
        prefer 3
        apply (subst MDP.MC.T-eq-bind)
        apply (subst distr-bind)
           prefer 4
           apply (subst distr-distr)
             prefer 3
             apply (subst K-cfg-map-absc)
```

```
          apply (rule prems)
         apply (subst map-pmf-rep-eq)
         apply (subst bind-distr)
           prefer 4
           apply (rule bind-measure-pmf-cong)
             prefer 3
    subgoal premises A for x
    proof −
      have t (absc x) = x unfolding t-def
      proof (rule the-equality, goal-cases)
        case 1 with A show ?case by simp
      next
        case (2 x′)
        with K-cfg-bisim-unique[OF prems - A absc-bisim-abss] show ?case by simp
      qed
      then show ?thesis by (auto simp: comp-def)
    qed
    by (fastforce
        simp: space-subprob-algebra MC-syntax.in-S
        intro: bind-measure-pmf-cong MDP.MC.T.subprob-space-distr MDP.MC.T.prob-space-distr
        )+
  qed (auto simp: M′-def intro: MDP.MC.T.prob-space-distr)
qed

lemma R-G-trace-space-distr-eq:
  assumes cfg ∈ R-G.valid-cfg abss s = state cfg
  shows MDP.MC.T cfg = distr (MDP.MC.T (repcs s cfg)) MDP.MC.S (smap absc)
using assms
proof (coinduction arbitrary: cfg s rule: MDP.MC.T-coinduct)
  case prob
  show ?case by (rule MDP.MC.T.prob-space-distr, simp)
next
  case sets
  show ?case by auto
next
  case prems: (cont cfg s)
  let ?μ = rept s (action cfg)
  define repc′ where repc′ ≡ λ cfg′. repcs (THE s. s ∈ ?μ ∧ abss s = state cfg′) cfg′
  define M′ where M′ ≡ λ cfg. distr (MDP.MC.T (repc′ cfg)) MDP.MC.S (smap absc)
  show ?case
  proof (intro exI[where x = M′], safe, goal-cases)
    case A: (1 cfg′)
    with K-cfg-rept-action[OF prems] have
      abss (THE s. s ∈ ?μ ∧ abss s = state cfg′) = state cfg′
    by auto
    moreover from A prems have cfg′ ∈ R-G.valid-cfg by auto
    ultimately show ?case unfolding M′-def repc′-def by best
  next
    case 4
    show ?case unfolding M′-def by (rule MDP.MC.T.prob-space-distr, simp)
  next
    case 5
    have ∗: smap absc ∘ (##) (repc′ cfg′) = (##) cfg′ ∘ smap absc
    if cfg′ ∈ set-pmf (K-cfg cfg) for cfg′
    proof −
      from K-cfg-rept-action[OF prems that] have
        abss (THE s. s ∈ ?μ ∧ abss s = state cfg′) = state cfg′
      .
      with prems that have ∗:
        absc (repc′ cfg′) = cfg′
      unfolding repc′-def by (subst absc-repcs-id, auto)
```

then show $(smap\ absc \circ (\#\#)\ (repc'\ cfg')) = ((\#\#)\ cfg' \circ smap\ absc)$ **by** *auto*
qed
from *prems* **show** *?case* **unfolding** *M'-def*
apply (*subst distr-distr*)
apply *simp+*
apply (*subst MDP.MC.T-eq-bind*)
apply (*subst distr-bind*)
prefer *2*
apply *simp*
apply (*rule MDP.MC.distr-Stream-subprob*)
apply *simp*
apply (*subst distr-distr*)
apply *simp+*
apply (*subst K-cfg-map-repcs*[*OF prems*])
apply (*subst map-pmf-rep-eq*)
apply (*subst bind-distr*)
by (*fastforce simp*: ∗[*unfolded repc'-def*] *repc'-def space-subprob-algebra MC-syntax.in-S*
intro: *bind-measure-pmf-cong MDP.MC.T.subprob-space-distr*)+
qed (*simp add*: *M'-def*)+
qed

lemma *repc-inj-on-K-cfg*:
assumes $cfg \in R\text{-}G.cfg\text{-}on\ s\ s \in \mathcal{S}$
shows *inj-on repc* (*set-pmf* (*K-cfg cfg*))
using *assms*
by (*intro inj-on-inverseI*[**where** $g = absc$], *subst absc-repc-id*)
(*auto intro*: *R-G.valid-cfgD R-G.valid-cfgI R-G.valid-cfg-state-in-S*)

lemma *smap-absc-iff*:
assumes $\bigwedge x\ y.\ x \in X \Longrightarrow smap\ abss\ x = smap\ abss\ y \Longrightarrow y \in X$
shows $(smap\ state\ xs \in X) = (smap\ (\lambda z.\ abss\ (state\ z))\ xs \in smap\ abss\ `\ X)$
proof (*safe*, *goal-cases*)
case *1*
then show *?case* **unfolding** *image-def*
by *clarify* (*inst-existentials smap state xs*, *auto simp*: *stream.map-comp*)
next
case *prems*: (*2 xs'*)
have
$smap\ (\lambda z.\ abss\ (state\ z))\ xs = smap\ abss\ (smap\ state\ xs)$
by (*auto simp*: *comp-def stream.map-comp*)
with *prems* **have** $smap\ abss\ (smap\ state\ xs) = smap\ abss\ xs'$ **by** *simp*
with *prems*(*2*) *assms* **show** *?case* **by** *auto*
qed

lemma *valid-abss-reps*[*simp*]:
assumes $cfg \in R\text{-}G.valid\text{-}cfg$
shows $abss\ (reps\ (state\ cfg)) = state\ cfg$
using *assms* **by** (*subst* $\mathcal{S}$*-abss-reps*) (*auto intro*: *R-G.valid-cfg-state-in-S*)

lemma *in-space-UNIV*: $x \in space\ (count\text{-}space\ UNIV)$
by *simp*

lemma *S-reps-*$\mathcal{S}$*-aux*:
$reps\ (l, R) \in S \Longrightarrow (l, R) \in \mathcal{S}$
using *ccompatible-inv* **unfolding** *reps-def ccompatible-def* $\mathcal{S}$*-def S-def*
by (*cases* $R \in \mathcal{R}$; *auto simp*: *non-empty*)

lemma *S-reps-*$\mathcal{S}$[*intro*]:
$reps\ s \in S \Longrightarrow s \in \mathcal{S}$
using *S-reps-*$\mathcal{S}$*-aux* **by** (*metis surj-pair*)

lemma *absc-valid-cfg-eq*:
  absc ‘ valid-cfg = R-G.valid-cfg
  apply *safe*
  subgoal
    by *auto*
  subgoal for *cfg*
    using *absc-repcs-id*[**where** s = *reps* (*state cfg*)]
    by − (*frule repcs-valid*[**where** s = *reps* (*state cfg*)]; *force intro*: *imageI*)
  done

lemma *action-repcs*:
  action (*repcs* (l, u) *cfg*) = *rept* (l, u) (*action cfg*)
  by (*simp add*: *repcs-def*)

5.3 Equalities Between Measures of Trace Spaces

lemma *path-measure-eq-absc1-new*:
  fixes *cfg s*
  defines *cfg'* ≡ *absc cfg*
  assumes *valid*: *cfg* ∈ *valid-cfg*
  assumes X[*measurable*]: X ∈ *R-G.St* **and** Y[*measurable*]: Y ∈ *MDP.St*
  assumes P: *AE x in* (*R-G.T cfg'*). P x **and** Q: *AE x in* (*MDP.T cfg*). Q x
  assumes P'[*measurable*]: *Measurable.pred R-G.St P*
    and Q'[*measurable*]: *Measurable.pred MDP.St Q*
  assumes *X-Y-closed*: $\bigwedge$ *x y*. $P\ x \Longrightarrow$ *smap abss* $y = x \Longrightarrow x \in X \Longrightarrow y \in Y \wedge Q\ y$
  assumes *Y-X-closed*: $\bigwedge$ *x y*. $Q\ y \Longrightarrow$ *smap abss* $y = x \Longrightarrow y \in Y \Longrightarrow x \in X \wedge P\ x$
  shows
    emeasure (*R-G.T cfg'*) X = *emeasure* (*MDP.T cfg*) Y
proof −
  have ∗: *stream-all2* (λ*s*. (=) (*absc s*)) *x y* = *stream-all2* (=) (*smap absc x*) *y* **for** *x y*
    by *simp*
  have ∗: *stream-all2* (λ*s t*. *t* = *absc s*) *x y* = *stream-all2* (=) *y* (*smap absc x*) **for** *x y*
    using *stream.rel-conversep*[*of* λ*s t*. *t* = *absc s*]
    by (*simp add*: *conversep-iff*[*abs-def*])

  from P **have** *emeasure* (*R-G.T cfg'*) X = *emeasure* (*R-G.T cfg'*) {*x* ∈ *X*. *P x*}
    by (*auto intro*: *emeasure-eq-AE*)
  moreover from Q **have** *emeasure* (*MDP.T cfg*) Y = *emeasure* (*MDP.T cfg*) {*y* ∈ *Y*. *Q y*}
    by (*auto intro*: *emeasure-eq-AE*)
  moreover show *?thesis*
   apply (*simp only*: *calculation*)
    unfolding *R-G.T-def MDP.T-def*
    apply (*simp add*: *emeasure-distr*)
    apply (*rule sym*)
    apply (*rule T-eq-rel-half*[**where** f = *absc* **and** S = *valid-cfg*])
         apply (*rule HOL.refl*)
    apply *measurable*
      apply (*simp add*: *space-stream-space*)
    subgoal
      unfolding *rel-set-strong-def stream.rel-eq*
      apply (*intro allI impI*)
      apply (*drule stream.rel-mono-strong*[**where** *Ra* = λ*s t*. *t* = *absc s*])
       apply (*simp*; *fail*)
      subgoal for *x y*
      using *Y-X-closed*[*of smap state x smap state* (*smap absc x*) **for** *x y*]
      using *X-Y-closed*[*of smap state* (*smap absc x*) *smap state x* **for** *x y*]
      by (*auto simp*: ∗ *stream.rel-eq stream.map-comp state-absc*)+
    done
    subgoal
      apply (*auto intro*!: *rel-funI*)

```
      apply (subst K-cfg-map-absc)
       defer
       apply (subst pmf.rel-map(2))
       apply (rule rel-pmf-reflI)
      by auto
    subgoal
      using valid unfolding cfg′-def by simp
    done
qed


lemma path-measure-eq-repcs1-new:
  fixes cfg s
  defines cfg′ ≡ repcs s cfg
  assumes s: abss s = state cfg
  assumes valid: cfg ∈ R-G.valid-cfg
  assumes X[measurable]: X ∈ R-G.St and Y[measurable]: Y ∈ MDP.St
  assumes P: AE x in (R-G.T cfg). P x and Q: AE x in (MDP.T cfg′). Q x
  assumes P′[measurable]: Measurable.pred R-G.St P
      and Q′[measurable]: Measurable.pred MDP.St Q
  assumes X-Y-closed: ⋀ x y. P x ⟹ smap abss y = x ⟹ x ∈ X ⟹ y ∈ Y ∧ Q y
  assumes Y-X-closed: ⋀ x y. Q y ⟹ smap abss y = x ⟹ y ∈ Y ⟹ x ∈ X ∧ P x
  shows
    emeasure (R-G.T cfg) X = emeasure (MDP.T cfg′) Y
proof −
  have ∗: stream-all2 (λs t. t = absc s) x y = stream-all2 (=) y (smap absc x) for x y
    using stream.rel-conversep[of λs t. t = absc s]
    by (simp add: conversep-iff[abs-def])
  from P X have
    emeasure (R-G.T cfg) X = emeasure (R-G.T cfg) {x ∈ X. P x}
    by (auto intro: emeasure-eq-AE)
  moreover from Q Y have
    emeasure (MDP.T cfg′) Y = emeasure (MDP.T cfg′) {y ∈ Y. Q y}
    by (auto intro: emeasure-eq-AE)
  moreover show ?thesis
    apply (simp only: calculation)
    unfolding R-G.T-def MDP.T-def
    apply (simp add: emeasure-distr)
    apply (rule sym)
    apply (rule T-eq-rel-half[where f = absc and S = valid-cfg])
         apply (rule HOL.refl)
        apply measurable
      apply (simp add: space-stream-space)
    subgoal
      unfolding rel-set-strong-def stream.rel-eq
      apply (intro allI impI)
      apply (drule stream.rel-mono-strong[where Ra = λs t. t = absc s])
       apply (simp; fail)
      subgoal for x y
        using Y-X-closed[of smap state x smap state (smap absc x) for x y]
        using X-Y-closed[of smap state (smap absc x) smap state x for x y]
         by (auto simp: ∗ stream.rel-eq stream.map-comp state-absc)+
      done
    subgoal
      apply (auto intro!: rel-funI)
      apply (subst K-cfg-map-absc)
       defer
       apply (subst pmf.rel-map(2))
       apply (rule rel-pmf-reflI)
      by auto
    subgoal
```

using *valid* **unfolding** *cfg′-def* **by** (*auto simp*: *s absc-repcs-id*)
done
qed

lemma *region-compatible-suntil1*:
assumes (*holds* (λx. φ (*reps* x)) *suntil holds* (λx. ψ (*reps* x))) (*smap abss* x)
and *pred-stream* (λ s. φ (*reps* (*abss* s)) $\longrightarrow$ φ s) x
and *pred-stream* (λ s. ψ (*reps* (*abss* s)) $\longrightarrow$ ψ s) x
shows (*holds* φ *suntil holds* ψ) x **using** *assms*
proof (*induction smap abss x arbitrary*: x *rule*: *suntil.induct*)
case *base*
then show *?case* **by** (*auto intro*: *suntil.base simp*: *stream.pred-set*)
next
case *step*
have
pred-stream (λs. φ (*reps* (*abss* s)) $\longrightarrow$ φ s) (*stl* x)
pred-stream (λs. ψ (*reps* (*abss* s)) $\longrightarrow$ ψ s) (*stl* x)
using *step.prems* **apply** (*cases* x; *auto*)
using *step.prems* **apply** (*cases* x; *auto*)
done
with *step.hyps*(*3*)[*of stl x*] **have** (*holds* φ *suntil holds* ψ) (*stl* x) **by** *auto*
with *step.prems step.hyps*(*1*−*2*) **show** *?case* **by** (*auto intro*: *suntil.step simp*: *stream.pred-set*)
qed

lemma *region-compatible-suntil2*:
assumes (*holds* φ *suntil holds* ψ) x
and *pred-stream* (λ s. φ s $\longrightarrow$ φ (*reps* (*abss* s))) x
and *pred-stream* (λ s. ψ s $\longrightarrow$ ψ (*reps* (*abss* s))) x
shows (*holds* (λx. φ (*reps* x)) *suntil holds* (λx. ψ (*reps* x))) (*smap abss* x) **using** *assms*
proof (*induction* x *rule*: *suntil.induct*)
case (*base* x)
then show *?case* **by** (*auto intro*: *suntil.base simp*: *stream.pred-set*)
next
case (*step* x)
have
pred-stream (λs. φ s $\longrightarrow$ φ (*reps* (*abss* s))) (*stl* x)
pred-stream (λs. ψ s $\longrightarrow$ ψ (*reps* (*abss* s))) (*stl* x)
using *step.prems* **apply** (*cases* x; *auto*)
using *step.prems* **apply** (*cases* x; *auto*)
done
with *step* **show** *?case* **by** (*auto intro*: *suntil.step simp*: *stream.pred-set*)
qed

lemma *region-compatible-suntil*:
assumes *pred-stream* (λ s. φ (*reps* (*abss* s)) $\longleftrightarrow$ φ s) x
and *pred-stream* (λ s. ψ (*reps* (*abss* s)) $\longleftrightarrow$ ψ s) x
shows (*holds* (λx. φ (*reps* x)) *suntil holds* (λx. ψ (*reps* x))) (*smap abss* x)
$\longleftrightarrow$ (*holds* φ *suntil holds* ψ) x **using** *assms*
using *assms region-compatible-suntil1 region-compatible-suntil2* **unfolding** *stream.pred-set* **by** *blast*

lemma *reps-abss-S*:
assumes *reps* (*abss* s) $\in$ S
shows $s \in S$
by (*simp add*: *S-reps-$\mathcal{S}$ $\mathcal{S}$-abss-S assms*)

lemma *measurable-sset*[*measurable* (*raw*)]:
assumes f[*measurable*]: $f \in N \to_M$ *stream-space* M **and** P[*measurable*]: *Measurable.pred* M P
shows *Measurable.pred* N (λx. $\forall s{\in}$*sset* (f x). P s)
proof −
have $*$: (λx. $\forall s{\in}$*sset* (f x). P s) = (λx. $\forall i$. P (f x !! i))
by (*simp add*: *sset-range*)

show *?thesis*
  unfolding $*$ **by** *measurable*
qed

lemma *path-measure-eq-repcs''-new*:
  notes *in-space-UNIV*[*measurable*]
  fixes *cfg* φ ψ *s*
  defines *cfg'* $\equiv$ *repcs s cfg*
  defines φ' $\equiv$ *absp* φ **and** ψ' $\equiv$ *absp* ψ
  assumes *s*: *abss s* = *state cfg*
  assumes *valid*: *cfg* $\in$ *R-G.valid-cfg*
  assumes *valid'*: *cfg'* $\in$ *valid-cfg*
  assumes *equiv-*φ: $\bigwedge$ *x*. *pred-stream* (λ *s*. *s* $\in$ *S*) *x*
    $\Longrightarrow$ *pred-stream* (λ *s*. φ (*reps* (*abss s*)) $\longleftrightarrow$ φ *s*) (*state cfg'* ## *x*)
  and *equiv-*ψ: $\bigwedge$ *x*. *pred-stream* (λ *s*. *s* $\in$ *S*) *x*
    $\Longrightarrow$ *pred-stream* (λ *s*. ψ (*reps* (*abss s*)) $\longleftrightarrow$ ψ *s*) (*state cfg'* ## *x*)
  shows
    emeasure (*R-G.T cfg*) {*x*$\in$*space R-G.St*. (*holds* φ' *suntil holds* ψ') (*state cfg* ## *x*)} =
    emeasure (*MDP.T cfg'*) {*x*$\in$*space MDP.St*. (*holds* φ *suntil holds* ψ) (*state cfg'* ## *x*)}
  unfolding *cfg'-def*
  apply (*rule path-measure-eq-repcs1-new*[**where** *P* = *pred-stream* (λ *s*. *s* $\in$ $\mathcal{S}$) **and** *Q* = *pred-stream* (λ *s*. *s* $\in$ *S*)])
        apply *fact*
       apply *fact*
      apply *measurable*
  subgoal
    unfolding *R-G.T-def*
    apply (*subst AE-distr-iff*)
      apply (*auto*; *fail*)
     apply (*auto simp*: *stream.pred-set*; *fail*)
    apply (*rule AE-mp*[*OF MDP.MC.AE-T-enabled AE-I2*])
    using *R-G.pred-stream-cfg-on*[*OF valid*] **by** (*auto simp*: *stream.pred-set*)
  subgoal
    unfolding *MDP.T-def*
    apply (*subst AE-distr-iff*)
      apply (*auto*; *fail*)
     apply (*auto simp*: *stream.pred-set*; *fail*)
    apply (*rule AE-mp*[*OF MDP.MC.AE-T-enabled AE-I2*])
    using *MDP.pred-stream-cfg-on*[*OF valid'*, *unfolded cfg'-def*] **by** (*auto simp*: *stream.pred-set*)
     apply *measurable*
  subgoal premises *prems* **for** *ys xs*
    apply *safe*
      apply *measurable*
    unfolding φ'*-def* ψ'*-def* *absp-def*
     apply (*subst region-compatible-suntil*[*symmetric*])
    subgoal
    proof −
      from *prems* **have** *pred-stream* (λ*s*. *s* $\in$ *S*) *xs* **using** $\mathcal{S}$*-abss-S* **by** (*auto simp*: *stream.pred-set*)
      with *equiv-*φ **show** *?thesis* **by** (*simp add*: *cfg'-def*)
    qed
    subgoal
    proof −
      from *prems* **have** *pred-stream* (λ*s*. *s* $\in$ *S*) *xs* **using** $\mathcal{S}$*-abss-S* **by** (*auto simp*: *stream.pred-set*)
      with *equiv-*ψ **show** *?thesis* **by** (*simp add*: *cfg'-def*)
    qed
    using *valid prems*
     apply (*auto simp*: *s comp-def* φ'*-def* ψ'*-def* *absp-def dest*: *R-G.valid-cfg-state-in-S*)
    apply (*auto simp*: *stream.pred-set intro*: $\mathcal{S}$*-abss-S dest*: *R-G.valid-cfg-state-in-S*)
    done
  subgoal premises *prems* **for** *ys xs*
    apply *safe*

using *prems* **apply** (*auto simp*: *stream.pred-set* $\mathcal{S}$*-abss-S*; *measurable*; *fail*)
using *prems* **unfolding** φ'*-def* ψ'*-def absp-def comp-def* **apply** (*simp add*: *stream.map-comp*)
apply (*subst* (*asm*) *region-compatible-suntil*[*symmetric*])
subgoal
proof −
from *prems* **have** *pred-stream* ($\lambda s.\ s \in S$) *xs* **using** $\mathcal{S}$*-abss-S* **by** *auto*
with *equiv-*φ **show** *?thesis* **using** *valid* **by** (*simp add*: *cfg*$'$*-def repc-def*)
qed
subgoal
proof −
from *prems* **have** *pred-stream* ($\lambda s.\ s \in S$) *xs* **using** $\mathcal{S}$*-abss-S* **by** *auto*
with *equiv-*ψ **show** *?thesis* **using** *valid* **by** (*simp add*: *cfg*$'$*-def*)
qed
using *valid prems* **by** (*auto simp*: *s S-abss-*$\mathcal{S}$ *stream.pred-set dest*: *R-G.valid-cfg-state-in-S*)
done

end

end
theory *PTA-Reachability*
imports *PTA*
begin

6 Classifying Regions for Divergence

6.1 Pairwise

coinductive *pairwise* :: ($'a \Rightarrow 'a \Rightarrow$ *bool*) $\Rightarrow$ $'a$ *stream* $\Rightarrow$ *bool* **for** P **where**
$P\ a\ b \Longrightarrow$ *pairwise* P (b ## *xs*) $\Longrightarrow$ *pairwise* P (a ## b ## *xs*)

lemma *pairwise-Suc*:
pairwise P *xs* $\Longrightarrow$ P (*xs* !! i) (*xs* !! (*Suc* i))
by (*induction* i *arbitrary*: *xs*) (*force elim*: *pairwise.cases*)+

lemma *Suc-pairwise*:
$\forall\ i.\ P$ (*xs* !! i) (*xs* !! (*Suc* i)) $\Longrightarrow$ *pairwise* P *xs*
apply (*coinduction arbitrary*: *xs*)
apply (*subst stream.collapse*[*symmetric*])
apply (*rewrite* **in** *stl* - *stream.collapse*[*symmetric*])
apply (*intro exI conjI*, *rule HOL.refl*)
apply (*erule allE*[**where** $x = 0$]; *simp*; *fail*)
by *simp* (*metis snth.simps*(*2*))

lemma *pairwise-iff*:
pairwise P *xs* $\longleftrightarrow$ ($\forall\ i.\ P$ (*xs* !! i) (*xs* !! (*Suc* i)))
using *pairwise-Suc Suc-pairwise* **by** *blast*

lemma *pairwise-stlD*:
pairwise P *xs* $\Longrightarrow$ *pairwise* P (*stl xs*)
by (*auto elim*: *pairwise.cases*)

lemma *pairwise-pairD*:
pairwise P *xs* $\Longrightarrow$ P (*shd xs*) (*shd* (*stl xs*))
by (*auto elim*: *pairwise.cases*)

lemma *pairwise-mp*:
assumes *pairwise* P *xs* **and** *lift*: $\bigwedge x\ y.\ x \in$ *sset xs* $\Longrightarrow y \in$ *sset xs* $\Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
shows *pairwise* Q *xs* **using** *assms*
apply (*coinduction arbitrary*: *xs*)
subgoal for *xs*

```
 apply (subst stream.collapse[symmetric])
 apply (rewrite in stl - stream.collapse[symmetric])
 apply (intro exI conjI)
  apply (rule HOL.refl)
 by (auto intro: stl-sset dest: pairwise-pairD pairwise-stlD)
done

lemma pairwise-sdropD:
  pairwise P (sdrop i xs) if pairwise P xs
  using that
proof (coinduction arbitrary: i xs)
  case (pairwise i xs)
  then show ?case
    apply (inst-existentials shd (sdrop i xs) shd (stl (sdrop i xs)) stl (stl (sdrop i xs)))
    subgoal
      by (auto dest: pairwise-Suc) (metis sdrop-simps(1) sdrop-stl stream.collapse)
    subgoal
      by (inst-existentials i − 1 stl xs) (auto dest: pairwise-Suc pairwise-stlD)
    by (metis sdrop-simps(2) stream.collapse)
qed
```

6.2 Regions

```
lemma gt-GreaterD:
  assumes u ∈ region X I r valid-region X k I r c ∈ X u c > k c
  shows I c = Greater (k c)
proof −
  from assms have intv-elem c u (I c) valid-intv (k c) (I c) by auto
  with assms(4) show ?thesis by (cases I c) auto
qed

lemma const-ConstD:
  assumes u ∈ region X I r valid-region X k I r c ∈ X u c = d d ≤ k c
  shows I c = Const d
proof −
  from assms have intv-elem c u (I c) valid-intv (k c) (I c) by auto
  with assms(4,5) show ?thesis by (cases I c) auto
qed

lemma not-Greater-bounded:
  assumes I x ≠ Greater (k x) x ∈ X valid-region X k I r u ∈ region X I r
  shows u x ≤ k x
proof −
  from assms have intv-elem x u (I x) valid-intv (k x) (I x) by auto
  with assms(1) show u x ≤ k x by (cases I x) auto
qed

lemma Greater-closed:
  fixes t :: real
  assumes u ∈ region X I r valid-region X k I r c ∈ X I c = Greater (k c) t > k c
  shows u(c := t) ∈ region X I r
  using assms
  apply (intro region.intros)
     apply (auto; fail)
    apply standard
  subgoal for x
    by (cases x = c; cases I x; force intro!: intv-elem.intros)
  by auto

lemma Greater-unbounded-aux:
  assumes finite X valid-region X k I r c ∈ X I c = Greater (k c)
```

shows $\exists\ u \in$ *region X I r*. $u\ c > t$
using *assms Greater-closed*[*OF - assms*(*2−4*)]
proof −
let *?R* = *region X I r*
let *?t* = *if* $t > k\ c$ *then* $t + 1$ *else* $k\ c + 1$
have *t*: *?t* $> k\ c$ **by** *auto*
from *region-not-empty*[*OF assms*(*1*,*2*)] **obtain** *u* **where** *u*: $u \in$ *?R* **by** *auto*
from *Greater-closed*[*OF this assms*(*2−4*) *t*] **have** *u*(*c*:=*?t*) $\in$ *?R* **by** *auto*
with *t* **show** *?thesis* **by** (*inst-existentials u*(*c*:=*?t*)) *auto*
qed

6.3 Unbounded and Zero Regions

definition *unbounded x R* $\equiv \forall\ t.\ \exists\ u \in R.\ u\ x > t$

definition *zero x R* $\equiv \forall\ u \in R.\ u\ x = 0$

lemma *Greater-unbounded*:
assumes *finite X valid-region X k I r* $c \in X$ $I\ c =$ *Greater* ($k\ c$)
shows *unbounded c* (*region X I r*)
using *Greater-unbounded-aux*[*OF assms*] **unfolding** *unbounded-def* **by** *blast*

lemma *unbounded-Greater*:
assumes *valid-region X k I r* $c \in X$ *unbounded c* (*region X I r*)
shows $I\ c =$ *Greater* ($k\ c$)
using *assms* **unfolding** *unbounded-def* **by** (*auto intro*: *gt-GreaterD*)

lemma *Const-zero*:
assumes $c \in X$ $I\ c =$ *Const 0*
shows *zero c* (*region X I r*)
using *assms* **unfolding** *zero-def* **by** *force*

lemma *zero-Const*:
assumes *finite X valid-region X k I r* $c \in X$ *zero c* (*region X I r*)
shows $I\ c =$ *Const 0*
proof −
from *assms* **obtain** *u* **where** $u \in$ *region X I r* **by** *atomize-elim* (*auto intro*: *region-not-empty*)
with *assms* **show** *?thesis* **unfolding** *zero-def* **by** (*auto intro*: *const-ConstD*)
qed

lemma *zero-all*:
assumes *finite X valid-region X k I r* $c \in X$ $u \in$ *region X I r* $u\ c = 0$
shows *zero c* (*region X I r*)
proof −
from *assms* **have** *intv-elem c u* ($I\ c$) *valid-intv* ($k\ c$) ($I\ c$) **by** *auto*
then have $I\ c =$ *Const 0* **using** *assms*(*5*) **by** *cases auto*
with *assms* **have** $u'\ c = 0$ **if** $u' \in$ *region X I r* **for** u' **using** *that* **by** *force*
then show *?thesis* **unfolding** *zero-def* **by** *blast*
qed

7 Reachability

7.1 Definitions

locale *Probabilistic-Timed-Automaton-Regions-Reachability* =
Probabilistic-Timed-Automaton-Regions k v n not-in-X A
for *k v n not-in-X* **and** *A* :: ($'c$, t, $'s$) *pta* +
fixes $\varphi\ \psi$ :: ($'s$ ∗ ($'c$, t) *cval*) ⇒ *bool* **fixes** *s*
assumes φ: $\bigwedge x\ y.\ x \in S \Longrightarrow$ *timed-bisim* $x\ y \Longrightarrow \varphi\ x \longleftrightarrow \varphi\ y$
assumes ψ: $\bigwedge x\ y.\ x \in S \Longrightarrow$ *timed-bisim* $x\ y \Longrightarrow \psi\ x \longleftrightarrow \psi\ y$

assumes $s[intro, simp]$: $s \in S$
begin

definition $\varphi' \equiv absp\ \varphi$
definition $\psi' \equiv absp\ \psi$
definition $s' \equiv abss\ s$

lemma *s-s′-cfg-on*[*intro*]:
  assumes $cfg \in MDP.cfg\text{-}on\ s$
  shows $absc\ cfg \in R\text{-}G.cfg\text{-}on\ s'$
proof −
  from *assms s* **have** $cfg \in valid\text{-}cfg$ **unfolding** *MDP.valid-cfg-def* **by** *auto*
  then have $absc\ cfg \in R\text{-}G.cfg\text{-}on\ (state\ (absc\ cfg))$ **by** (*auto intro*: *R-G.valid-cfgD*)
  with *assms* **show** *?thesis* **unfolding** *s′-def* **by** (*auto simp*: *state-absc*)
qed

lemma s'-$\mathcal{S}$[*simp*, *intro*]:
  $s' \in \mathcal{S}$
  unfolding *s′-def* **using** *s* **by** *auto*

lemma *s′-s-cfg-on*[*intro*]:
  assumes $cfg \in R\text{-}G.cfg\text{-}on\ s'$
  shows $repcs\ s\ cfg \in MDP.cfg\text{-}on\ s$
proof −
  from *assms s* **have** $cfg \in R\text{-}G.valid\text{-}cfg$ **unfolding** *R-G.valid-cfg-def* **by** *auto*
  with *assms* **have** $repcs\ s\ cfg \in valid\text{-}cfg$ **by** (*auto simp*: *s′-def intro*: *R-G.valid-cfgD*)
  then show *?thesis* **by** (*auto dest*: *MDP.valid-cfgD*)
qed

lemma (**in** *Probabilistic-Timed-Automaton-Regions*) *compatible-stream*:
  assumes φ: $\bigwedge x\ y.\ x \in S \Longrightarrow x \sim y \Longrightarrow \varphi\ x \longleftrightarrow \varphi\ y$
  assumes $pred\text{-}stream\ (\lambda s.\ s \in S)\ xs$
    and [*intro*]: $x \in S$
   shows $pred\text{-}stream\ (\lambda s.\ \varphi\ (reps\ (abss\ s)) = \varphi\ s)\ (x\ \#\#\ xs)$
unfolding *stream.pred-set* **proof** *clarify*
  fix *l u*
  assume A: $(l, u) \in sset\ (x\ \#\#\ xs)$
  from *assms* **have** $pred\text{-}stream\ (\lambda s.\ s \in S)\ (x\ \#\#\ xs)$ **by** *auto*
  with A **have** $(l, u) \in S$ **by** (*fastforce simp*: *stream.pred-set*)
  then have $abss\ (l, u) \in \mathcal{S}$ **by** *auto*
  then have $reps\ (abss\ (l, u)) \sim (l, u)$ **by** *simp*
  with φ ‹$(l, u) \in S$› **show** $\varphi\ (reps\ (abss\ (l, u))) = \varphi\ (l, u)$ **by** *blast*
qed

lemma *φ-stream′*:
  $pred\text{-}stream\ (\lambda s.\ \varphi\ (reps\ (abss\ s)) = \varphi\ s)\ (x\ \#\#\ xs)$ **if** $pred\text{-}stream\ (\lambda s.\ s \in S)\ xs\ x \in S$
  using *compatible-stream*[*of* φ, *OF* φ *that*] **.**

lemma *ψ-stream′*:
  $pred\text{-}stream\ (\lambda s.\ \psi\ (reps\ (abss\ s)) = \psi\ s)\ (x\ \#\#\ xs)$ **if** $pred\text{-}stream\ (\lambda s.\ s \in S)\ xs\ x \in S$
  using *compatible-stream*[*of* ψ, *OF* ψ *that*] **.**

lemmas *φ-stream* = *compatible-stream*[*of* φ, *OF* φ]
lemmas *ψ-stream* = *compatible-stream*[*of* ψ, *OF* ψ]

7.2 Easier Result on All Configurations

lemma *suntil-reps*:
  assumes
    $\forall s \in sset\ (smap\ abss\ y).\ s \in \mathcal{S}$
    $(holds\ \varphi'\ suntil\ holds\ \psi')\ (s'\ \#\#\ smap\ abss\ y)$

shows (*holds* φ *suntil holds* ψ) (s ## y)
using *assms*
by (*subst region-compatible-suntil*[*symmetric*]; (*intro* φ*-stream* ψ*-stream*)*?*)
(*auto simp*: φ'*-def* ψ'*-def absp-def stream.pred-set* $\mathcal{S}$*-abss-S* s'*-def comp-def*)

lemma *suntil-abss*:
assumes
$\forall s \in sset\ y.\ s \in S$
(*holds* φ *suntil holds* ψ) (s ## y)
shows
(*holds* φ' *suntil holds* ψ') (s' ## *smap abss* y)
using *assms*
by (*subst* (*asm*) *region-compatible-suntil*[*symmetric*]; (*intro* φ*-stream* ψ*-stream*)*?*)
(*auto simp*: φ'*-def* ψ'*-def absp-def stream.pred-set* s'*-def comp-def*)

theorem *P-sup-sunitl-eq*:
notes [*measurable*] = *in-space-UNIV* **and** [*iff*] = *pred-stream-iff*
shows
(*MDP.P-sup s* ($\lambda x.$ (*holds* φ *suntil holds* ψ) (s ## x)))
= (*R-G.P-sup* s' ($\lambda x.$ (*holds* φ' *suntil holds* ψ') (s' ## x)))
unfolding *MDP.P-sup-def R-G.P-sup-def*
proof (*rule SUP-eq*, *goal-cases*)
case *prems*: (*1 cfg*)
let *?cfg'* = *absc cfg*
from *prems* **have** *cfg* $\in$ *valid-cfg* **by** (*auto intro*: *MDP.valid-cfgI*)
then have *?cfg'* $\in$ *R-G.valid-cfg* **by** (*auto intro*: *R-G.valid-cfgI*)
from ‹*cfg* $\in$ *valid-cfg*› **have** *alw-S*: *almost-everywhere* (*MDP.T cfg*) (*pred-stream* ($\lambda s.\ s \in S$))
by (*rule MDP.alw-S*)
from ‹*?cfg'* $\in$ *R-G.valid-cfg*› **have** *alw-*$\mathcal{S}$: *almost-everywhere* (*R-G.T ?cfg'*) (*pred-stream* ($\lambda s.\ s \in \mathcal{S}$))
by (*rule R-G.alw-S*)
have *emeasure* (*MDP.T cfg*) $\{x \in$ *space MDP.St.* (*holds* φ *suntil holds* ψ) (s ## x)$\}$
= *emeasure* (*R-G.T ?cfg'*) $\{x \in$ *space R-G.St.* (*holds* φ' *suntil holds* ψ') (s' ## x)$\}$
apply (*rule path-measure-eq-absc1-new*[*symmetric*, **where** P = *pred-stream* ($\lambda\ s.\ s \in \mathcal{S}$)
and Q = *pred-stream* ($\lambda\ s.\ s \in S$)]
)
using *prems alw-S alw-*$\mathcal{S}$ **apply** (*auto intro*: *MDP.valid-cfgI simp*:)[*7*]
by (*auto simp*: *S-abss-*$\mathcal{S}$ *intro*: $\mathcal{S}$*-abss-S intro*!: *suntil-abss suntil-reps*, *measurable*)
with *prems* **show** *?case* **by** (*inst-existentials ?cfg'*) *auto*
next
case *prems*: (*2 cfg*)
let *?cfg'* = *repcs s cfg*
have s = *state ?cfg'* **by** *simp*
from *prems* **have** s' = *state cfg* **by** *auto*
have *pred-stream* ($\lambda s.\ \varphi$ (*reps* (*abss s*)) = φ *s*) (*state* (*repcs s cfg*) ## x)
if *pred-stream* ($\lambda s.\ s \in S$) x **for** x
using *prems that* **by** (*intro* φ*-stream*) *auto*
moreover
have *pred-stream* ($\lambda s.\ \psi$ (*reps* (*abss s*)) = ψ *s*) (*state* (*repcs s cfg*) ## x)
if *pred-stream* ($\lambda s.\ s \in S$) x **for** x
using *prems that* **by** (*intro* ψ*-stream*) *auto*
ultimately
have *emeasure* (*R-G.T cfg*) $\{x \in$ *space R-G.St.* (*holds* φ' *suntil holds* ψ') (s' ## x)$\}$
= *emeasure* (*MDP.T* (*repcs s cfg*)) $\{x \in$ *space MDP.St.* (*holds* φ *suntil holds* ψ) (s ## x)$\}$
apply (*rewrite* **in** *s* ## - ‹*s* = -›)
apply (*subst* ‹s' = -›)
unfolding φ'*-def* ψ'*-def* s'*-def*
apply (*rule path-measure-eq-repcs''-new*)
using *prems* **by** (*auto 4 3 simp*: s'*-def intro*: *R-G.valid-cfgI MDP.valid-cfgI*)
with *prems* **show** *?case* **by** (*inst-existentials ?cfg'*) *auto*

qed

end

7.3 Divergent Adversaries

context *Probabilistic-Timed-Automaton*
begin

definition $elapsed\ u\ u' \equiv Max\ (\{u'\ c - u\ c \mid c.\ c \in \mathcal{X}\} \cup \{0\})$

definition $eq\text{-}elapsed\ u\ u' \equiv elapsed\ u\ u' > 0 \longrightarrow (\forall\ c \in \mathcal{X}.\ u'\ c - u\ c = elapsed\ u\ u')$

fun $dur :: ('c,\ t)\ cval\ stream \Rightarrow nat \Rightarrow t$ **where**
$dur\ \text{-}\ 0 = 0\ |$
$dur\ (x\ \#\#\ y\ \#\#\ xs)\ (Suc\ i) = elapsed\ x\ y + dur\ (y\ \#\#\ xs)\ i$

definition $divergent\ \omega \equiv \forall\ t.\ \exists\ n.\ dur\ \omega\ n > t$

definition $div\text{-}cfg\ cfg \equiv AE\ \omega\ in\ MDP.MC.T\ cfg.\ divergent\ (smap\ (snd\ o\ state)\ \omega)$

definition $\mathcal{R}\text{-}div\ \omega \equiv$
$\forall x \in \mathcal{X}.\ (\forall\ i.\ (\exists\ j \geq i.\ zero\ x\ (\omega\ !!\ j)) \wedge (\exists\ j \geq i.\ \neg\ zero\ x\ (\omega\ !!\ j)))$
$\vee\ (\exists\ i.\ \forall\ j \geq i.\ unbounded\ x\ (\omega\ !!\ j))$

definition $R\text{-}G\text{-}div\text{-}cfg\ cfg \equiv AE\ \omega\ in\ MDP.MC.T\ cfg.\ \mathcal{R}\text{-}div\ (smap\ (snd\ o\ state)\ \omega)$

end

context *Probabilistic-Timed-Automaton-Regions*
begin

definition $cfg\text{-}on\text{-}div\ st \equiv MDP.cfg\text{-}on\ st \cap \{cfg.\ div\text{-}cfg\ cfg\}$

definition $R\text{-}G\text{-}cfg\text{-}on\text{-}div\ st \equiv R\text{-}G.cfg\text{-}on\ st \cap \{cfg.\ R\text{-}G\text{-}div\text{-}cfg\ cfg\}$

lemma $measurable\text{-}\mathcal{R}\text{-}div[measurable]$: $Measurable.pred\ MDP.MC.S\ \mathcal{R}\text{-}div$
unfolding $\mathcal{R}\text{-}div\text{-}def$
by (*intro*
pred-intros-finite[*OF beta-interp.finite*]
pred-intros-logic pred-intros-countable
measurable-count-space-const measurable-compose[*OF measurable-snth*]
) *measurable*

lemma *elapsed-ge0*[*simp*]: $elapsed\ x\ y \geq 0$
unfolding *elapsed-def* **using** *finite*(*1*) **by** *auto*

lemma *dur-pos*:
$dur\ xs\ i \geq 0$
apply (*induction i arbitrary*: *xs*)
apply (*auto*; *fail*)
subgoal for *i xs*
apply (*subst stream.collapse*[*symmetric*])
apply (*rewrite at stl xs stream.collapse*[*symmetric*])
apply (*subst dur.simps*)
by *simp*
done

lemma *dur-mono*:
$i \leq j \Longrightarrow dur\ xs\ i \leq dur\ xs\ j$
proof (*induction i arbitrary*: *xs j*)

```
  case 0 show ?case by (auto intro: dur-pos)
next
  case (Suc i xs j)
  obtain x y ys where xs: xs = x ## y ## ys using stream.collapse by metis
  from Suc obtain j' where j': j = Suc j' by (cases j) auto
  with xs have dur xs j = elapsed x y + dur (y ## ys) j' by auto
  also from Suc j' have ... ≥ elapsed x y + dur (y ## ys) i by auto
  also have elapsed x y + dur (y ## ys) i = dur xs (Suc i) by (simp add: xs)
  finally show ?case .
qed

lemma dur-monoD:
  assumes dur xs i < dur xs j
  shows i < j using assms
 by − (rule ccontr; auto 4 4 dest: leI dur-mono[where xs = xs])

lemma elapsed-0D:
  assumes c ∈ 𝒳 elapsed u u' ≤ 0
  shows u' c − u c ≤ 0
proof −
  from assms have u' c − u c ∈ {u' c − u c | c. c ∈ 𝒳} ∪ {0} by auto
  with finite(1) have u' c − u c ≤ Max ({u' c − u c | c. c ∈ 𝒳} ∪ {0}) by auto
  with assms(2) show ?thesis unfolding elapsed-def by auto
qed

lemma elapsed-ge:
  assumes eq-elapsed u u' c ∈ 𝒳
  shows elapsed u u' ≥ u' c − u c
  using assms unfolding eq-elapsed-def by (auto intro: elapsed-ge0 order.trans[OF elapsed-0D])

lemma elapsed-eq:
  assumes eq-elapsed u u' c ∈ 𝒳 u' c − u c ≥ 0
  shows elapsed u u' = u' c − u c
  using elapsed-ge[OF assms(1,2)] assms unfolding eq-elapsed-def by auto

lemma dur-shift:
  dur ω (i + j) = dur ω i + dur (sdrop i ω) j
 apply (induction i arbitrary: ω)
  apply simp
 subgoal for i ω
  apply simp
  apply (subst stream.collapse[symmetric])
  apply (rewrite at stl ω stream.collapse[symmetric])
  apply (subst dur.simps)
  apply (rewrite in dur ω stream.collapse[symmetric])
  apply (rewrite in dur (- ## ¤) (Suc -) stream.collapse[symmetric])
  apply (subst dur.simps)
  apply simp
 done
done

lemma dur-zero:
  assumes
    ∀ i. xs !! i ∈ ω !! i ∀ j ≤ i. zero x (ω !! j) x ∈ 𝒳
    ∀ i. eq-elapsed (xs !! i) (xs !! Suc i)
  shows dur xs i = 0 using assms
proof (induction i arbitrary: xs ω)
  case 0
  then show ?case by simp
next
  case (Suc i xs ω)
```

let *?x* = *xs* !! *0*
let *?y* = *xs* !! *1*
let *?ys* = *stl* (*stl xs*)
have *xs*: *xs* = *?x* ## *?y* ## *?ys* **by** *auto*
from *Suc.prems* **have**
$\forall$ *i*. (*?y* ## *?ys*) !! *i* $\in$ *stl* ω !! *i* $\forall$ *j* $\leq$ *i*. *zero x* (*stl* ω !! *j*)
$\forall$ *i*. *eq-elapsed* (*stl xs* !! *i*) (*stl xs* !! *Suc i*)
by (*metis snth.simps*(*2*) | *auto*)+
from *Suc.IH*[*OF this*(*1*,*2*) ‹*x* $\in$ -›] *this*(*3*) **have** [*simp*]: *dur* (*stl xs*) *i* = *0* **by** *auto*
from *Suc.prems*(*1*,*2*) **have** *?y x* = *0* *?x x* = *0* **unfolding** *zero-def* **by** *force+*
then have *: *?y x* − *?x x* = *0* **by** *simp*
have *dur xs* (*Suc i*) = *elapsed ?x ?y*
apply (*subst xs*)
apply (*subst dur.simps*)
by *simp*
also have ... = *0*
apply (*subst elapsed-eq*[*OF* - ‹*x* $\in$ -›])
unfolding *One-nat-def* **using** *Suc.prems*(*4*) **apply** *blast*
using * **by** *auto*
finally show *?case* **.**
qed

lemma *dur-zero-tail*:
assumes $\forall$ *i*. *xs* !! *i* $\in$ ω !! *i* $\forall$ *k* $\geq$ *i*. *k* $\leq$ *j* $\longrightarrow$ *zero x* (ω !! *k*) *x* $\in$ $\mathcal{X}$ *j* $\geq$ *i*
$\forall$ *i*. *eq-elapsed* (*xs* !! *i*) (*xs* !! *Suc i*)
shows *dur xs j* = *dur xs i*
proof −
from ‹*j* $\geq$ *i*› *dur-shift*[*of xs i j* − *i*] **have**
dur xs j = *dur xs i* + *dur* (*sdrop i xs*) (*j* − *i*)
by *simp*
also have ... = *dur xs i*
using *assms*
by (*rewrite* **in** *dur* (*sdrop* - -) - *dur-zero*[**where** ω = *sdrop i* ω])
(*auto dest*: *prop-nth-sdrop-pair*[*of eq-elapsed*] *prop-nth-sdrop prop-nth-sdrop-pair*[*of* ($\in$)])
finally show *?thesis* **.**
qed

lemma *elapsed-ge-pos*:
fixes *u* :: ('*c*, *t*) *cval*
assumes *eq-elapsed u u*′ *c* $\in$ $\mathcal{X}$ *u* $\in$ *V u*′ $\in$ *V*
shows *elapsed u u*′ $\leq$ *u*′ *c*
proof (*cases elapsed u u*′ = *0*)
case *True*
with *assms* **show** *?thesis* **by** (*auto simp*: *V-def*)
next
case *False*
from ‹*u* $\in$ *V*› ‹*c* $\in$ $\mathcal{X}$ › **have** *u c* $\geq$ *0* **by** (*auto simp*: *V-def*)
from *False assms* **have** *elapsed u u*′ = *u*′ *c* − *u c*
unfolding *eq-elapsed-def* **by** (*auto simp add*: *less-le*)
also from ‹*u c* $\geq$ *0*› **have** ... $\leq$ *u*′ *c* **by** *simp*
finally show *?thesis* **.**
qed

lemma *dur-Suc*:
dur xs (*Suc i*) − *dur xs i* = *elapsed* (*xs* !! *i*) (*xs* !! *Suc i*)
apply (*induction i arbitrary*: *xs*)
apply *simp*
apply (*subst stream.collapse*[*symmetric*])
apply (*rewrite* **in** *stl* - *stream.collapse*[*symmetric*])
apply (*subst dur.simps*)
apply *simp*

```
 apply simp
 subgoal for i xs
  apply (subst stream.collapse[symmetric])
  apply (rewrite in stl - stream.collapse[symmetric])
  apply (subst dur.simps)
  apply simp
  apply (rewrite in dur xs (Suc -) stream.collapse[symmetric])
  apply (rewrite at stl xs in - ## stl xs stream.collapse[symmetric])
  apply (subst dur.simps)
  apply simp
 done
done

inductive trans where
  succ: t ≥ 0 ⟹ u′ = u ⊕ t ⟹ trans u u′ |
  reset: set l ⊆ 𝒳 ⟹ u′ = clock-set l 0 u ⟹ trans u u′ |
  id: u = u′ ⟹ trans u u′

abbreviation stream-trans ≡ pairwise trans

lemma K-cfg-trans:
  assumes cfg ∈ MDP.cfg-on (l, R) cfg′ ∈ K-cfg cfg state cfg′ = (l′, R′)
  shows trans R R′
using assms
 apply (simp add: set-K-cfg)
 apply (drule MDP.cfg-onD-action)
 apply (cases rule: K.cases)
    apply (auto intro: trans.intros)
using admissible-targets-clocks(2) by (blast intro: trans.intros(2))

lemma enabled-stream-trans:
  assumes cfg ∈ valid-cfg MDP.MC.enabled cfg xs
  shows stream-trans (smap (snd o state) xs)
  using assms
proof (coinduction arbitrary: cfg xs)
  case prems: (pairwise cfg xs)
  let ?xs = stl (stl xs) let ?x = shd xs let ?y = shd (stl xs)
  from MDP.pred-stream-cfg-on[OF prems] have ∗:
    pred-stream (λcfg. state cfg ∈ S ∧ cfg ∈ MDP.cfg-on (state cfg)) xs .
  obtain l R l′ R′ where eq: state ?x = (l, R) state ?y = (l′, R′) by force
  moreover from ∗ have ?x ∈ MDP.cfg-on (state ?x) ?x ∈ valid-cfg
    by (auto intro: MDP.valid-cfgI simp: stream.pred-set)
  moreover from prems(2) have ?y ∈ K-cfg ?x by (auto elim: MDP.MC.enabled.cases)
  ultimately have trans R R′
    by (intro K-cfg-trans[where cfg = ?x and cfg′ = ?y and l = l and l′ = l′]) metis+
  with ‹?x ∈ valid-cfg› prems(2) show ?case
    apply (inst-existentials R R′ smap (snd o state) ?xs)
      apply (simp add: eq; fail)+
    apply (rule disjI1, inst-existentials ?x stl xs)
    by (auto simp: eq elim: MDP.MC.enabled.cases)
qed

lemma stream-trans-trans:
  assumes stream-trans xs
  shows trans (xs !! i) (stl xs !! i)
using pairwise-Suc assms by auto

lemma trans-eq-elapsed:
  assumes trans u u′ u ∈ V
  shows eq-elapsed u u′
using assms
```

proof *cases*
case (*succ t*)
with *finite*(*1*) **show** *?thesis* **by** (*auto simp*: *cval-add-def elapsed-def max-def eq-elapsed-def*)
next
case *prems*: (*reset l*)
then have $u'\ c - u\ c \leq 0$ **if** $c \in \mathcal{X}$ **for** *c*
using *that* ‹$u \in V$› **by** (*cases* $c \in$ *set l*) (*auto simp*: *V-def*)
then have *elapsed u u'* $= 0$ **unfolding** *elapsed-def* **using** *finite*(*1*)
apply *simp*
apply (*subst Max-insert2*)
by *auto*
then show *?thesis* **by** (*auto simp*: *eq-elapsed-def*)
next
case *id*
then show *?thesis*
using *finite*(*1*) **by** (*auto simp*: *Max-gr-iff elapsed-def eq-elapsed-def*)
qed

lemma *pairwise-trans-eq-elapsed*:
assumes *stream-trans xs pred-stream* ($\lambda\ u.\ u \in V$) *xs*
shows *pairwise eq-elapsed xs*
using *trans-eq-elapsed assms* **by** (*auto intro*: *pairwise-mp simp*: *stream.pred-set*)

lemma *not-reset-dur*:
assumes $\forall k{>}i.\ k \leq j \longrightarrow \neg$ *zero c* ([*xs* !! *k*]$_{\mathcal{R}}$) $j \geq i$ $c \in \mathcal{X}$ *stream-trans xs*
$\forall\ i.$ *eq-elapsed* (*xs* !! *i*) (*xs* !! *Suc i*) $\forall\ i.$ *xs* !! $i \in V$
shows *dur xs j* $-$ *dur xs i* $=$ (*xs* !! *j*) *c* $-$ (*xs* !! *i*) *c*
using *assms*
proof (*induction j*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc j*)
from *stream-trans-trans*[*OF Suc.prems*(*4*)] **have** *trans*: *trans* (*xs* !! *j*) (*xs* !! *Suc j*) **by** *auto*
from *Suc.prems* **have** ∗:
$\neg$ *zero c* ([*xs* !! *Suc j*]$_{\mathcal{R}}$) *eq-elapsed* (*xs* !! *j*) (*xs* !! *Suc j*) **if** *Suc j* $> i$
using *that* **by** *auto*
from *Suc.prems*(*6*) **have** *xs* !! $j \in V$ *xs* !! *Suc j* $\in V$ **by** *blast+*
then have *regions*: [*xs* !! *j*]$_{\mathcal{R}} \in \mathcal{R}$ [*xs* !! *Suc j*]$_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
from *trans* **have** (*xs* !! *Suc j*) *c* $-$ (*xs* !! *j*) *c* ≥ 0 **if** *Suc j* $> i$
proof (*cases*)
case *succ*
with *regions* **show** *?thesis* **by** (*auto simp*: *cval-add-def*)
next
case *prems*: (*reset l*)
show *?thesis*
proof (*cases* $c \in$ *set l*)
case *False*
with *prems* **show** *?thesis* **by** *auto*
next
case *True*
with *prems* **have** (*xs* !! *Suc j*) *c* $= 0$ **by** *auto*
moreover from *assms* **have** *xs* !! *Suc j* $\in$ [*xs* !! *Suc j*]$_{\mathcal{R}}$ **by** *blast*
ultimately have
zero c ([*xs* !! *Suc j*]$_{\mathcal{R}}$)
using *zero-all*[*OF finite*(*1*) - ‹$c \in \mathcal{X}$›] *regions*(*2*) **by** (*auto simp*: $\mathcal{R}$-*def*)
with ∗ *that* **show** *?thesis* **by** *auto*
qed
next
case *id* **then show** *?thesis* **by** *simp*
qed
with ∗ ‹$c \in \mathcal{X}$› *elapsed-eq* **have**

$*$: *elapsed* (*xs* !! *j*) (*xs* !! *Suc j*) = (*xs* !! *Suc j*) *c* − (*xs* !! *j*) *c*
if *Suc j* > *i*
using *that* **by** *blast*
show *?case*
proof (*cases i* = *Suc j*)
case *False*
with *Suc* **have**
dur xs (*Suc j*) − *dur xs i* = *dur xs* (*Suc j*) − *dur xs j* + (*xs* !! *j*) *c* − (*xs* !! *i*) *c*
by *auto*
also have ... = *elapsed* (*xs* !! *j*) (*xs* !! *Suc j*) + (*xs* !! *j*) *c* − (*xs* !! *i*) *c*
by (*simp add*: *dur-Suc*)
also have
... = (*xs* !! *Suc j*) *c* − (*xs* !! *j*) *c* + (*xs* !! *j*) *c* − (*xs* !! *i*) *c*
using $*$ *False Suc.prems* **by** *auto*
also have ... = (*xs* !! *Suc j*) *c* − (*xs* !! *i*) *c* **by** *simp*
finally show *?thesis* **by** *auto*
next
case *True*
then show *?thesis* **by** *simp*
qed
qed

lemma *not-reset-dur′*:
assumes ∀*j*≥*i*. ¬ *zero c* ([*xs* !! *j*]$_{\mathcal{R}}$) *j* ≥ *i* *c* ∈ $\mathcal{X}$ *stream-trans xs*
∀ *i*. *eq-elapsed* (*xs* !! *i*) (*xs* !! *Suc i*) ∀ *j*. *xs* !! *j* ∈ *V*
shows *dur xs j* − *dur xs i* = (*xs* !! *j*) *c* − (*xs* !! *i*) *c*
using *assms not-reset-dur* **by** *auto*

lemma *not-reset-unbounded*:
assumes ∀*j*≥*i*. ¬ *zero c* ([*xs* !! *j*]$_{\mathcal{R}}$) *j* ≥ *i* *c* ∈ $\mathcal{X}$ *stream-trans xs*
∀ *i*. *eq-elapsed* (*xs* !! *i*) (*xs* !! *Suc i*) ∀ *j*. *xs* !! *j* ∈ *V*
unbounded c ([*xs* !! *i*]$_{\mathcal{R}}$)
shows *unbounded c* ([*xs* !! *j*]$_{\mathcal{R}}$)
proof −
let *?u* = *xs* !! *i* **let** *?u′* = *xs* !! *j* **let** *?R* = [*xs* !! *i*]$_{\mathcal{R}}$
from *assms* **have** *?u* ∈ *?R* **by** *auto*
from *assms*(*6*) **have** *?R* ∈ $\mathcal{R}$ **by** *auto*
then obtain *I r* **where** *?R* = *region* $\mathcal{X}$ *I r valid-region* $\mathcal{X}$ *k I r* **unfolding** $\mathcal{R}$*-def* **by** *auto*
with *assms*(*3*,*7*) *unbounded-Greater* ‹*?u* ∈ *?R*› **have** *?u c* > *k c* **by** *force*
also from *not-reset-dur′*[*OF assms*(*1*−*6*)] *dur-mono*[*OF* ‹*j* ≥ *i*›, *of xs*] **have** *?u′ c* ≥ *?u c* **by** *auto*
finally have *?u′ c* > *k c* **by** *auto*
let *?R′* = [*xs* !! *j*]$_{\mathcal{R}}$
from *assms* **have** *?u′* ∈ *?R′* **by** *auto*
from *assms*(*6*) **have** *?R′* ∈ $\mathcal{R}$ **by** *auto*
then obtain *I r* **where** *?R′* = *region* $\mathcal{X}$ *I r valid-region* $\mathcal{X}$ *k I r* **unfolding** $\mathcal{R}$*-def* **by** *auto*
moreover with ‹*?u′ c* > -› ‹*?u′* ∈ -› *gt-GreaterD* ‹*c* ∈ $\mathcal{X}$› **have** *I c* = *Greater* (*k c*) **by** *auto*
ultimately show *?thesis* **using** *Greater-unbounded*[*OF finite*(*1*) - ‹*c* ∈ $\mathcal{X}$›] **by** *auto*
qed

lemma *gt-unboundedD*:
assumes *u* ∈ *R*
and *R* ∈ $\mathcal{R}$
and *c* ∈ $\mathcal{X}$
and *real* (*k c*) < *u c*
shows *unbounded c R*
proof −
from *assms* **obtain** *I r* **where** *R* = *region* $\mathcal{X}$ *I r valid-region* $\mathcal{X}$ *k I r*
unfolding $\mathcal{R}$*-def* **by** *auto*
with *Greater-unbounded*[*of* $\mathcal{X}$ *k I r c*] *gt-GreaterD*[*of u* $\mathcal{X}$ *I r k c*] *assms finite*(*1*) **show** *?thesis*
by *auto*
qed

definition *trans'* :: ('*c*, *t*) *cval* ⇒ ('*c*, *t*) *cval* ⇒ *bool* **where**
  trans' u u' ≡
    ((∀ $c \in \mathcal{X}$. $u\ c > k\ c \wedge u'\ c > k\ c \wedge u \neq u'$) ⟶ $u' = u \oplus 0.5$) ∧
    ((∃ $c \in \mathcal{X}$. $u\ c = 0 \wedge u'\ c > 0 \wedge (\forall c{\in}\mathcal{X}.\ \nexists d.\ d \leq k\ c \wedge u'\ c = real\ d)$)
    ⟶ $u' = delayedR\ ([u']_{\mathcal{R}})\ u$)

lemma *zeroI*:
  assumes $c \in \mathcal{X}$ $u \in V$ $u\ c = 0$
  shows *zero c* ($[u]_{\mathcal{R}}$)
proof −
  from *assms* **have** $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
  then obtain *I r* **where** $[u]_{\mathcal{R}}$ = *region* $\mathcal{X}$ *I r valid-region* $\mathcal{X}$ *k I r* **unfolding** $\mathcal{R}$*-def* **by** *auto*
  with *zero-all*[*OF finite*(*1*) *this*(*2*) ‹$c \in \mathcal{X}$›] ‹$u \in [u]_{\mathcal{R}}$› ‹$u\ c = 0$› **show** *?thesis* **by** *auto*
qed

lemma *zeroD*:
  $u\ x = 0$ **if** *zero x* ($[u]_{\mathcal{R}}$) $u \in V$
  using *that* **by** (*metis regions-part-ex*(*1*) *zero-def*)

lemma *not-zeroD*:
  assumes ¬ *zero x* ($[u]_{\mathcal{R}}$) $u \in V$ $x \in \mathcal{X}$
  shows $u\ x > 0$
proof −
  from *zeroI assms* **have** $u\ x \neq 0$ **by** *auto*
  moreover from *assms* **have** $u\ x \geq 0$ **unfolding** *V-def* **by** *auto*
  ultimately show *?thesis* **by** *auto*
qed

lemma *not-const-intv*:
  assumes $u \in V$ $\forall c{\in}\mathcal{X}.\ \nexists d.\ d \leq k\ c \wedge u\ c = real\ d$
  shows $\forall c{\in}\mathcal{X}.\ \forall u \in [u]_{\mathcal{R}}.\ \nexists d.\ d \leq k\ c \wedge u\ c = real\ d$
proof −
  from *assms* **have** $u \in [u]_{\mathcal{R}}$ $[u]_{\mathcal{R}} \in \mathcal{R}$ **by** *auto*
  then obtain *I r* **where** *I*: $[u]_{\mathcal{R}}$ = *region* $\mathcal{X}$ *I r valid-region* $\mathcal{X}$ *k I r* **unfolding** $\mathcal{R}$*-def* **by** *auto*
  have $\nexists d.\ d \leq k\ c \wedge u'\ c = real\ d$ **if** $c \in \mathcal{X}$ $u' \in [u]_{\mathcal{R}}$ **for** *c u'*
  proof *safe*
    fix *d* **assume** *A*: $d \leq k\ c$ $u'\ c = real\ d$
    from *I that* **have** *intv-elem c u'* (*I c*) *valid-intv* (*k c*) (*I c*) **by** *auto*
    then show *False*
      using *A I* ‹$u \in [u]_{\mathcal{R}}$› ‹$c \in \mathcal{X}$› *assms*(*2*) **by** (*cases*; *fastforce*)
  qed
  then show *?thesis* **by** *auto*
qed

lemma *K-cfg-trans'*:
  assumes *repcs* (*l*, *u*) *cfg* ∈ *MDP.cfg-on* (*l*, *u*) *cfg'* ∈ *K-cfg* (*repcs* (*l*, *u*) *cfg*)
        state cfg' = (*l'*, *u'*) (*l*, *u*) ∈ *S cfg* ∈ *R-G.valid-cfg abss* (*l*, *u*) = *state cfg*
  shows *trans' u u'*
using *assms*
 apply (*simp add*: *set-K-cfg*)
 apply (*drule MDP.cfg-onD-action*)
 apply (*cases rule*: *K.cases*)
 apply *assumption*
proof *goal-cases*
  case *prems*: (*1 l u t*)
  from *assms* ‹- = (*l*, *u*)› **have** *repcs* (*l*, *u*) *cfg* ∈ *valid-cfg* **by** (*auto intro*: *MDP.valid-cfgI*)
  then have *absc* (*repcs* (*l*, *u*) *cfg*) ∈ *R-G.valid-cfg* **by** *auto*

from *prems* **have** ∗: $rept\ (l,\ u)\ (action\ cfg) = return\text{-}pmf\ (l,\ u \oplus t)$ **unfolding** *repcs-def* **by** *auto*
from ‹*abss* - = -› ‹- = (*l*, *u*)› ‹*cfg* ∈ *R-G.valid-cfg*› **have**
action cfg ∈ $\mathcal{K}$ (*abss* (*l*, *u*))
by (*auto dest*: *R-G-I*)
from *abst-rept-id*[*OF this*] ∗ **have** $action\ cfg = abst\ (return\text{-}pmf\ (l,\ u \oplus t))$ **by** *auto*
with *prems* **have** ∗∗: $action\ cfg = return\text{-}pmf\ (l,\ [u \oplus t]_{\mathcal{R}})$ **unfolding** *abst-def* **by** *auto*
show *?thesis*
proof (*cases* $\forall\ c \in \mathcal{X}.\ u\ c > k\ c$)
case *True*
from *prems* **have** $u \oplus t \in [u]_{\mathcal{R}}$ **by** (*auto intro*: *upper-right-closed*[*OF True*])
with *prems* **have** $[u \oplus t]_{\mathcal{R}} = [u]_{\mathcal{R}}$ **by** (*auto dest*: *alpha-interp.region-unique-spec*)
with ∗∗ **have** $action\ cfg = return\text{-}pmf\ (l,\ [u]_{\mathcal{R}})$ **by** *simp*
with *True* **have** $rept\ (l,\ u)\ (action\ cfg) = return\text{-}pmf\ (l,\ u \oplus 0.5)$
unfolding *rept-def* **using** *prems* **by** *auto*
with ∗ **have** $u \oplus t = u \oplus 0.5$ **by** *auto*
moreover from *prems* **have** $u' = u \oplus t$ **by** *auto*
moreover from *prems True* **have** $\forall\ c \in \mathcal{X}.\ u'\ c > k\ c$ **by** (*auto simp*: *cval-add-def*)
ultimately show *?thesis* **using** *True* ‹- = (*l*, *u*)› **unfolding** *trans′-def* **by** *auto*
next
case *F*: *False*
show *?thesis*
proof (*cases* $\exists\, c{\in}\mathcal{X}.\ u\ c = 0 \wedge 0 < u'\ c \wedge (\forall\, c{\in}\mathcal{X}.\ \nexists\, d.\ d \le k\ c \wedge u'\ c = real\ d)$)
case *True*
from *prems* **have** $u' \in [u']_{\mathcal{R}}$ **by** *auto*
from *prems* **have** $[u \oplus t]_{\mathcal{R}} \in Succ\ \mathcal{R}\ ([u]_{\mathcal{R}})$ **by** *auto*
from *True* **obtain** *c* **where** $c \in \mathcal{X}\ \ u\ c = 0\ \ u'\ c > 0$ **by** *auto*
with *zeroI prems* **have** $zero\ c\ ([u]_{\mathcal{R}})$ **by** *auto*
moreover from ‹$u' \in$ -› ‹$u'\ c > 0$› **have** $\neg\ zero\ c\ ([u']_{\mathcal{R}})$ **unfolding** *zero-def* **by** *fastforce*
ultimately have $[u \oplus t]_{\mathcal{R}} \neq [u]_{\mathcal{R}}$ **using** *prems* **by** *auto*
moreover from *True not-const-intv prems* **have**
$\forall\ u \in [u \oplus t]_{\mathcal{R}}.\ \forall\, c{\in}\mathcal{X}.\ \nexists\, d.\ d \le k\ c \wedge u\ c = real\ d$
by *auto*
ultimately have $\exists\, R'.\ (l,\ u) \in S\ \wedge$
$action\ cfg = return\text{-}pmf\ (l,\ R') \wedge$
$R' \in Succ\ \mathcal{R}\ ([u]_{\mathcal{R}}) \wedge [u]_{\mathcal{R}} \neq R' \wedge (\forall\, u{\in}R'.\ \forall\, c{\in}\mathcal{X}.\ \nexists\, d.\ d \le k\ c \wedge u\ c = real\ d)$
apply −
apply (*rule exI*[**where** $x = [u \oplus t]_{\mathcal{R}}$])
apply *safe*
using *prems* ∗∗ **by** *auto*
then have
$rept\ (l,\ u)\ (action\ cfg)$
$= return\text{-}pmf\ (l,\ delayedR\ (SOME\ R'.\ action\ cfg = return\text{-}pmf\ (l,\ R'))\ u)$
unfolding *rept-def* **by** *auto*
with ∗ ∗∗ *prems* **have** $u' = delayedR\ ([u \oplus t]_{\mathcal{R}})\ u$ **by** *auto*
with *F True prems* **show** *?thesis* **unfolding** *trans′-def* **by** *auto*
next
case *False*
with *F* ‹- = (*l*, *u*)› **show** *?thesis* **unfolding** *trans′-def* **by** *auto*
qed
qed
next
case *prems*: (*2* - - τ μ)
then obtain *X* **where** *X*: $u' = ([X := 0]u)\ (X,\ l') \in set\text{-}pmf\ \mu$ **by** *auto*
from ‹- ∈ *S*› **have** $u \in V$ **by** *auto*
let $?r = SOME\ r.\ set\ r = X$
show *?case*
proof (*cases* $X = \{\}$)
case *True*
with *X* **have** $u = u'$ **by** *auto*
with *non-empty* **show** *?thesis* **unfolding** *trans′-def* **by** *auto*
next

```
    case False
    then obtain x where x ∈ X by auto
    moreover have X ⊆ 𝒳 using admissible-targets-clocks(1)[OF prems(10) X(2)] by auto
    ultimately have x ∈ 𝒳 by auto
    from ‹X ⊆ 𝒳› finite(1) obtain r where set r = X using finite-list finite-subset by blast
    then have r: set ?r = X by (rule someI)
    with ‹x ∈ X› X have u′ x = 0 by auto
    from X r ‹u ∈ V› ‹X ⊆ 𝒳› have u′ x ≤ u x for x
      by (cases x ∈ X; auto simp: V-def)
    have False if u′ x > 0 ∧ u x = 0 for x
      using ‹u′ - ≤ -›[of x] that by auto
    with ‹u′ x = 0› show ?thesis using ‹x ∈ 𝒳› unfolding trans′-def by auto
  qed
next
  case 3
  with non-empty show ?case unfolding trans′-def by auto
qed

coinductive enabled-repcs where
  enabled-repcs (shd xs) (stl xs) ⟹ shd xs = repcs st′ cfg′ ⟹ st′ ∈ rept st (action cfg)
  ⟹ abss st′ = state cfg′
  ⟹ cfg′ ∈ R-G.valid-cfg
  ⟹ enabled-repcs (repcs st cfg) xs


lemma K-cfg-rept-in:
assumes cfg ∈ R-G.valid-cfg
    and abss st = state cfg
    and cfg′ ∈ K-cfg cfg
  shows (THE s′. s′ ∈ set-pmf (rept st (action cfg)) ∧ abss s′ = state cfg′)
        ∈ set-pmf (rept st (action cfg))
proof −
  from assms(1,2) have action cfg ∈ 𝒦 (abss st) by (auto simp: R-G-I)
  from ‹cfg′ ∈ -› have
    cfg′ = cont cfg (state cfg′) state cfg′ ∈ action cfg
  by (auto simp: set-K-cfg)
  with abst-rept-id[OF ‹action - ∈ -›] pmf.set-map have
    state cfg′ ∈ abss ‘ set-pmf (rept st (action cfg)) unfolding abst-def by metis
  then obtain st′ where
    st′ ∈ rept st (action cfg) abss st′ = state cfg′
  unfolding abst-def by auto
  with K-cfg-rept-aux[OF assms(1,2) this(1)] show ?thesis by auto
qed

lemma enabled-repcsI:
  assumes cfg ∈ R-G.valid-cfg abss st = state cfg MDP.MC.enabled (repcs st cfg) xs
  shows enabled-repcs (repcs st cfg) xs using assms
proof (coinduction arbitrary: cfg xs st)
  case prems: (enabled-repcs cfg xs st)
  let ?x = shd xs and ?y = shd (stl xs)
  let ?st = THE s′. s′ ∈ set-pmf (rept st (action cfg)) ∧ abss s′ = state (absc ?x)
  from prems(3) have ?x ∈ K-cfg (repcs st cfg) by cases
  with K-cfg-map-repcs[OF prems(1,2)] obtain cfg′ where
    cfg′ ∈ K-cfg cfg ?x = repcs (THE s′. s′ ∈ rept st (action cfg) ∧ abss s′ = state cfg′) cfg′
    by auto
  let ?st = THE s′. s′ ∈ rept st (action cfg) ∧ abss s′ = state cfg′
  from K-cfg-rept-action[OF prems(1,2) ‹cfg′ ∈ -›] have abss ?st = state cfg′ .
  moreover from K-cfg-rept-in[OF prems(1,2) ‹cfg′ ∈ -›] have ?st ∈ rept st (action cfg) .
  moreover have cfg′ ∈ R-G.valid-cfg using ‹cfg′ ∈ K-cfg cfg› prems(1) by blast
  moreover from absc-repcs-id[OF this ‹abss ?st = state cfg′›] ‹?x = -› have absc ?x = cfg′
    by auto
```

```
  moreover from prems(3) have MDP.MC.enabled (shd xs) (stl xs) by cases
  ultimately show ?case
    using ‹?x = -› by (inst-existentials xs ?st absc ?x st cfg) fastforce+
qed

lemma repcs-eq-rept:
  rept st (action cfg) = rept st'' (action cfg'') if repcs st cfg = repcs st'' cfg''
   by (metis (mono-tags, lifting) action-cfg-corec old.prod.case repcs-def that)

lemma enabled-stream-trans':
  assumes cfg ∈ R-G.valid-cfg abss st = state cfg MDP.MC.enabled (repcs st cfg) xs
  shows pairwise trans' (smap (snd o state) xs)
using assms
proof (coinduction arbitrary: cfg xs st)
  case prems: (pairwise cfg xs)
  let ?xs = stl xs
  from prems have A: enabled-repcs (repcs st cfg) xs by (auto intro: enabled-repcsI)
  then obtain st' cfg' where
    enabled-repcs (shd xs) (stl xs) shd xs = repcs st' cfg' st' ∈ rept st (action cfg)
    abss st' = state cfg' cfg' ∈ R-G.valid-cfg
   apply atomize-elim
   apply (cases rule: enabled-repcs.cases)
   apply assumption
   subgoal for st' cfg' st'' cfg''
     by (inst-existentials st' cfg') (auto dest: repcs-eq-rept)
  done
  then obtain st'' cfg'' where
    enabled-repcs (shd ?xs) (stl ?xs)
    shd ?xs = repcs st'' cfg'' st'' ∈ rept st' (action cfg') abss st'' = state cfg''
    by atomize-elim (subst (asm)enabled-repcs.simps, fastforce dest: repcs-eq-rept)
  let ?x = shd xs let ?y = shd (stl xs)
  let ?cfg = repcs st cfg
  from prems have ?cfg ∈ valid-cfg by auto
  from MDP.pred-stream-cfg-on[OF ‹?cfg ∈ valid-cfg› prems(3)] have ∗:
    pred-stream (λcfg. state cfg ∈ S ∧ cfg ∈ MDP.cfg-on (state cfg)) xs .
  obtain l u l' u' where eq: st' = (l, u) st'' = (l', u')
    by force
  moreover from ∗ have
    ?x ∈ MDP.cfg-on (state ?x) ?x ∈ valid-cfg
    by (auto intro: MDP.valid-cfgI simp: stream.pred-set)
  moreover from prems(3) have ?y ∈ K-cfg ?x by (auto elim: MDP.MC.enabled.cases)
  ultimately have trans' u u'
    using ‹?x = -› ‹?y = -› ‹cfg' ∈ -› ‹abss st' = -›
    by (intro K-cfg-trans') (auto dest: MDP.valid-cfg-state-in-S)
  with ‹?x ∈ valid-cfg› ‹cfg' ∈ R-G.valid-cfg› prems(3) ‹abss - = state cfg'› show ?case
    apply (inst-existentials u u' smap (snd o state) (stl ?xs))
    apply (simp add: eq ‹?x = -› ‹?y = -›; fail)+
    by ((intro disjI1 exI)?; auto simp: ‹?x = -› ‹?y = -› eq elim: MDP.MC.enabled.cases)
qed

lemma divergent-ℛ-divergent:
  assumes in-S: pred-stream (λ u. u ∈ V) xs
     and  div:  divergent xs
     and trans: stream-trans xs
  shows ℛ-div (smap (λ u. [u]ℛ) xs) (is ℛ-div ?ω)
unfolding ℛ-div-def proof (safe, simp-all)
  fix x i
  assume x: x ∈ 𝒳 and bounded: ∀ i. ∃ j≥i. ¬ unbounded x ([xs !! j]ℛ)
  from in-S have xs-ω: ∀ i. xs !! i ∈ ?ω !! i by (auto simp: stream.pred-set)
  from trans in-S have elapsed:
    ∀  i. eq-elapsed (xs !! i) (xs !! Suc i)
```

```
    by (fastforce intro: pairwise-trans-eq-elapsed pairwise-Suc[where P = eq-elapsed])
  { assume A: ∀ j ≥ i. ¬ zero x ([xs !! j]_R)
    let ?t = dur xs i + k x
    from div obtain j where j: dur xs j > dur xs i + k x unfolding divergent-def by auto
    then have k x < dur xs j − dur xs i by auto
    also with not-reset-dur'[OF A less-imp-le[OF dur-monoD], of xs] ‹x ∈ 𝒳› assms elapsed have
      . . . = (xs !! j) x − (xs !! i) x
      by (auto simp: stream.pred-set)
    also have . . . ≤ (xs !! j) x
      using assms(1) ‹x ∈ 𝒳› unfolding V-def by (auto simp: stream.pred-set)
    finally have unbounded x ([xs !! j]_R)
      using assms ‹x ∈ 𝒳› by (intro gt-unboundedD) (auto simp: stream.pred-set)
    moreover from dur-monoD[of xs i j] j A have ∀ j' ≥ j. ¬ zero x ([xs !! j']_R) by auto
    ultimately have ∀ i≥j. unbounded x ([xs !! i]_R)
      using elapsed assms x by (auto intro: not-reset-unbounded simp: stream.pred-set)
    with bounded have False by auto
  }
  then show ∃ j≥i. zero x ([xs !! j]_R) by auto
  { assume A: ∀ j ≥ i. zero x ([xs !! j]_R)
    from div obtain j where j: dur xs j > dur xs i unfolding divergent-def by auto
    then have j ≥ i by (auto dest: dur-monoD)
    from A have ∀ j ≥ i. zero x (?ω !! j) by auto
    with dur-zero-tail[OF xs-ω - x ‹i ≤ j› elapsed] j have False by simp
  }
  then show ∃ j≥i. ¬ zero x ([xs !! j]_R) by auto
qed

lemma (in −)
  fixes f :: nat ⇒ real
  assumes ∀ i. f i ≥ 0 ∀ i. ∃ j ≥ i. f j > d d > 0
  shows ∃ n. (∑ i ≤ n. f i) > t
  oops


lemma dur-ev-exceedsI:
  assumes ∀ i. ∃ j ≥ i. dur xs j − dur xs i ≥ d and d > 0
  obtains i where dur xs i > t
proof −
  have base: ∃ i. dur xs i > t if t < d for t
  proof −
    from assms obtain j where dur xs j − dur xs 0 ≥ d by fastforce
    with dur-pos[of xs 0] have dur xs j ≥ d by simp
    with ‹d > 0› ‹t < d› show ?thesis by − (rule exI[where x = j]; auto)
  qed
  have base2: ∃ i. dur xs i > t if t ≤ d for t
  proof (cases t = d)
    case False
    with ‹t ≤ d› base show ?thesis by simp
  next
    case True
    from base ‹d > 0› obtain i where dur xs i > 0 by auto
    moreover from assms obtain j where dur xs j − dur xs i ≥ d by auto
    ultimately have dur xs j > d by auto
    with ‹t = d› show ?thesis by auto
  qed
  show ?thesis
  proof (cases t ≥ 0)
    case False
    with dur-pos have dur xs 0 > t by auto
    then show ?thesis by (fastforce intro: that)
  next
```

```
    case True
    let ?m = nat ⌈t / d⌉
    from True have ∃ i. dur xs i > ?m ∗ d
    proof (induction ?m arbitrary: t)
      case 0
      with base[OF ‹0 < d›] show ?case by simp
    next
      case (Suc n t)
      let ?t = t − d
      show ?case
      proof (cases t ≥ d)
        case True
        have ?t / d = t / d − 1


        proof −
          have t / d + − 1 ∗ ((t + − 1 ∗ d) / d) + − 1 ∗ (d / d) = 0
            by (simp add: diff-divide-distrib)
          then have t / d + − 1 ∗ ((t + − 1 ∗ d) / d) = 1
            using assms(2) by fastforce
          then show ?thesis
            by algebra
        qed
        then have ⌈?t / d⌉ = ⌈t / d⌉ − 1 by simp
        with ‹Suc n = -› have n = nat ⌈?t / d⌉ by simp
        with Suc ‹t ≥ d› obtain i where nat ⌈?t / d⌉ ∗ d < dur xs i by fastforce
        from assms obtain j where dur xs j − dur xs i ≥ d j ≥ i by auto
        with ‹dur xs i > -› have nat ⌈?t / d⌉ ∗ d + d < dur xs j by simp
        with True have dur xs j > nat ⌈t / d⌉ ∗ d
        by (metis Suc.hyps(2) ‹n = nat ⌈(t − d) / d⌉› add.commute distrib-left mult.commute
              mult.right-neutral of-nat-Suc)
        then show ?thesis by blast
      next
        case False
        with ‹t ≥ 0› ‹d > 0› have nat ⌈t / d⌉ ≤ 1 by simp
        then have nat ⌈t / d⌉ ∗ d ≤ d
        by (metis One-nat-def ‹Suc n = -› Suc-leI add.right-neutral le-antisym mult.commute
              mult.right-neutral of-nat-0 of-nat-Suc order-refl zero-less-Suc)
        with base2 show ?thesis by auto
      qed
    qed
    then obtain i where dur xs i > ?m ∗ d by atomize-elim
    moreover from ‹t ≥ 0› ‹d > 0› have ?m ∗ d ≥ t
      using pos-divide-le-eq real-nat-ceiling-ge by blast
    ultimately show ?thesis using that[of i] by simp
  qed
qed

lemma not-reset-mono:
  assumes stream-trans xs shd xs c1 ≥ shd xs c2 stream-all (λ u. u ∈ V) xs c2 ∈ 𝒳
  shows (holds (λ u. u c1 ≥ u c2) until holds (λ u. u c1 = 0)) xs using assms
proof (coinduction arbitrary: xs)
  case prems: (UNTIL xs)
  let ?xs = stl xs
  let ?x = shd xs
  let ?y = shd ?xs
  show ?case
  proof (cases ?x c1 = 0)
    case False
    show ?thesis
    proof (cases ?y c1 = 0)
```

```
      case False
      from prems have trans ?x ?y by (intro pairwise-pairD[of trans])
      then have ?y c1 ≥ ?y c2
      proof cases
        case A: (reset t)
        show ?thesis
        proof (cases c1 ∈ set t)
          case True
          with A False show ?thesis by auto
        next
          case False
          from prems have ?x c2 ≥ 0 by (auto simp: V-def)
          with A have ?y c2 ≤ ?x c2 by (cases c2 ∈ set t) auto
          with A False ‹?x c1 ≥ ?x c2› show ?thesis by auto
        qed
      qed (use prems in ‹auto simp: cval-add-def›)
      moreover from prems have stream-trans ?xs stream-all (λ u. u ∈ V) ?xs
        by (auto intro: pairwise-stlD stl-sset)
      ultimately show ?thesis
        using prems by auto
    qed (use prems in ‹auto intro: UNTIL.base›)
  qed auto
qed

lemma 𝓡-divergent-divergent-aux:
  fixes xs :: ('c, t) cval stream
  assumes stream-trans xs stream-all (λ u. u ∈ V) xs
          (xs !! i) c1 = 0 ∃ k > i. k ≤ j ∧ (xs !! k) c2 = 0
          ∀ k > i. k ≤ j ⟶ (xs !! k) c1 ≠ 0
          c1 ∈ 𝒳 c2 ∈ 𝒳
  shows (xs !! j) c1 ≥ (xs !! j) c2
proof −
  from assms obtain k where k: k > i k ≤ j (xs !! k) c2 = 0 by auto
  with assms(5) ‹k ≤ j› have (xs !! k) c1 ≠ 0 by auto
  moreover from assms(2) ‹c1 ∈ 𝒳› have (xs !! k) c1 ≥ 0 by (auto simp: V-def)
  ultimately have (xs !! k) c1 > 0 by auto
  with ‹(xs !! k) c2 = 0› have shd (sdrop k xs) c1 ≥ shd (sdrop k xs) c2 by auto
  from not-reset-mono[OF - this] assms have
    (holds (λu. u c2 ≤ u c1) until holds (λu. u c1 = 0)) (sdrop k xs)
  by (auto intro: sset-sdrop pairwise-sdropD)
  from assms(5) k(2) ‹k > i› have ∀ m ≤ j − k. (sdrop k xs !! m) c1 ≠ 0 by simp
  with holds-untilD[OF ‹(- until -) -›, of j − k] have
    (sdrop k xs !! (j − k)) c2 ≤ (sdrop k xs !! (j − k)) c1 .
  then show (xs !! j) c2 ≤ (xs !! j) c1 using k(1,2) by simp
qed

lemma unbounded-all:
  assumes R ∈ 𝓡 u ∈ R unbounded x R x ∈ 𝒳
  shows u x > k x
proof −
  from assms obtain I r where R: R = region 𝒳 I r valid-region 𝒳 k I r unfolding 𝓡-def by auto
  with unbounded-Greater ‹x ∈ 𝒳› assms(3) have I x = Greater (k x) by simp
  with ‹u ∈ R› R ‹x ∈ 𝒳› show ?thesis by force
qed

lemma trans-not-delay-mono:
  u' c ≤ u c if trans u u' u ∈ V x ∈ 𝒳 u' x = 0 c ∈ 𝒳
  using ‹trans u u'›
proof (cases)
  case (reset l)
  with that show ?thesis by (cases c ∈ set l) (auto simp: V-def)
```

qed (*use that* **in** ‹*auto simp*: *cval-add-def V-def add-nonneg-eq-0-iff*›)

lemma *dur-reset*:
  assumes *pairwise eq-elapsed xs pred-stream* (λ *u*. *u* $\in$ *V*) *xs zero x* ([*xs* !! *Suc i*]$_{\mathcal{R}}$) *x* $\in \mathcal{X}$
  shows *dur xs* (*Suc i*) $-$ *dur xs i* = *0*
proof $-$
  from *assms*(*2*) **have** *in-V*: *xs* !! *Suc i* $\in$ *V*
    unfolding *stream.pred-set* **by** *auto* (*metis snth.simps*(*2*) *snth-sset*)
  with *elapsed-ge-pos*[*of xs* !! *i xs* !! *Suc i x*] *pairwise-Suc*[*OF assms*(*1*)] *assms*(*2*$-$) **have**
    elapsed (*xs* !! *i*) (*xs* !! *Suc i*) $\leq$ (*xs* !! *Suc i*) *x*
    unfolding *stream.pred-set* **by** *auto*
  with *in-V assms*(*3*) **have** *elapsed* (*xs* !! *i*) (*xs* !! *Suc i*) $\leq$ *0* **by** (*auto simp*: *zeroD*)
  with *elapsed-ge0*[*of xs* !! *i xs* !! *Suc i*] **have** *elapsed* (*xs* !! *i*) (*xs* !! *Suc i*) = *0*
    by *linarith*
  then show *?thesis* **by** (*subst dur-Suc*)
qed

lemma *resets-mono-0′*:
  assumes *pairwise eq-elapsed xs stream-all* (λ *u*. *u* $\in$ *V*) *xs stream-trans xs*
      $\forall$ *j* $\leq$ *i*. *zero x* ([*xs* !! *j*]$_{\mathcal{R}}$) *x* $\in \mathcal{X}$ *c* $\in \mathcal{X}$
  shows (*xs* !! *i*) *c* = (*xs* !! *0*) *c* $\vee$ (*xs* !! *i*) *c* = *0*
using *assms* **proof** (*induction i*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc i*)
  from *Suc.prems* **have** $*$: (*xs* !! *Suc i*) *x* = *0* (*xs* !! *i*) *x* = *0*
    by (*blast intro*: *zeroD snth-sset*, *force intro*: *zeroD snth-sset*)
  from *pairwise-Suc*[*OF Suc.prems*(*3*)] **have** *trans* (*xs* !! *i*) (*xs* !! *Suc i*) **.**
  then show *?case*
  proof *cases*
    case *prems*: (*succ t*)
    with $*$ **have** *t* = *0* **unfolding** *cval-add-def* **by** *auto*
    with *prems* **have** (*xs* !! *Suc i*) *c* = (*xs* !! *i*) *c* **unfolding** *cval-add-def* **by** *auto*
    with *Suc* **show** *?thesis* **by** *auto*
  next
    case *prems*: (*reset l*)
    then have (*xs* !! *Suc i*) *c* = *0* $\vee$ (*xs* !! *Suc i*) *c* = (*xs* !! *i*) *c* **by** (*cases c* $\in$ *set l*) *auto*
    with *Suc* **show** *?thesis* **by** *auto*
  next
    case *id*
    with *Suc* **show** *?thesis* **by** *auto*
  qed
qed

lemma *resets-mono′*:
  assumes *pairwise eq-elapsed xs pred-stream* (λ *u*. *u* $\in$ *V*) *xs stream-trans xs*
      $\forall$ *k* $\geq$ *i*. *k* $\leq$ *j* $\longrightarrow$ *zero x* ([*xs* !! *k*]$_{\mathcal{R}}$) *x* $\in \mathcal{X}$ *c* $\in \mathcal{X}$ *i* $\leq$ *j*
  shows (*xs* !! *j*) *c* = (*xs* !! *i*) *c* $\vee$ (*xs* !! *j*) *c* = *0* **using** *assms*
proof $-$
  from *assms* **have** *1*: *stream-all* (λ *u*. *u* $\in$ *V*) (*sdrop i xs*)
    using *sset-sdrop* **unfolding** *stream.pred-set* **by** *force*
  from *assms* **have** *2*: *pairwise eq-elapsed* (*sdrop i xs*) **by** (*intro pairwise-sdropD*)
  from *assms* **have** *3*: *stream-trans* (*sdrop i xs*) **by** (*intro pairwise-sdropD*)
  from *assms* **have** *4*:
    $\forall$*k*$\leq$*j* $-$ *i*. *zero x* ([*sdrop i xs* !! *k*]$_{\mathcal{R}}$)
  by (*simp add*: *le-diff-conv2 assms*(*6*))
  from *resets-mono-0′*[*OF 2 1 3 4 assms*(*5*,*6*)] ‹*i* $\leq$ *j*› **show** *?thesis* **by** *simp*
qed

lemma *resets-mono*:

```
  assumes pairwise eq-elapsed xs pred-stream (λ u. u ∈ V) xs stream-trans xs
        ∀ k ≥ i. k ≤ j ⟶ zero x ([xs !! k]_ℛ) x ∈ 𝒳 c ∈ 𝒳 i ≤ j
  shows (xs !! j) c ≤ (xs !! i) c using assms
  using assms by (auto simp: V-def dest: resets-mono'[where c = c] simp: stream.pred-set)

lemma ℛ-divergent-divergent-aux2:
  fixes M :: (nat ⇒ bool) set
  assumes ∀ i. ∀ P ∈ M. ∃ j ≥ i. P j M ≠ {} finite M
  shows ∀ i.∃ j≥i.∃ k>j.∃ P ∈ M. P j ∧ P k ∧ (∀ m < k. j < m ⟶ ¬ P m)
      ∧ (∀ Q ∈ M. ∃ m ≤ k. j < m ∧ Q m)
proof
  fix i
  let ?j1 = Max {LEAST m. m > i ∧ P m | P. P ∈ M}
  from ‹M ≠ {}› obtain P where P ∈ M by auto
  let ?m = LEAST m. m > i ∧ P m
  from assms(1) ‹P ∈ M› obtain j where j ≥ Suc i P j by auto
  then have j > i P j by auto
  with ‹P ∈ M› have ?m > i ∧ P ?m by − (rule LeastI; auto)
  moreover with ‹finite M› ‹P ∈ M› have ?j1 ≥ ?m by − (rule Max-ge; auto)
  ultimately have ?j1 ≥ i by simp
  moreover have ∃ m > i. m ≤ ?j1 ∧ P m if P ∈ M for P
  proof −
    let ?m = LEAST m. m > i ∧ P m
    from assms(1) ‹P ∈ M› obtain j where j ≥ Suc i P j by auto
    then have j > i P j by auto
    with ‹P ∈ M› have ?m > i ∧ P ?m by − (rule LeastI; auto)
    moreover with ‹finite M› ‹P ∈ M› have ?j1 ≥ ?m by − (rule Max-ge; auto)
    ultimately show ?thesis by auto
  qed
  ultimately obtain j1 where j1: j1 ≥ i ∀ P ∈ M. ∃ m > i. j1 ≥ m ∧ P m by auto
  define k where k Q = (LEAST k. k > j1 ∧ Q k) for Q
  let ?k = Max {k Q | Q. Q ∈ M}
  let ?P = SOME P. P ∈ M ∧ k P = ?k
  let ?j = Max {j. i ≤ j ∧ j ≤ j1 ∧ ?P j}
  have ?k ∈ {k Q | Q. Q ∈ M} using assms by − (rule Max-in; auto)
  then obtain P where P: k P = ?k P ∈ M by auto
  have ?k ≥ k Q if Q ∈ M for Q using assms that by − (rule Max-ge; auto)
  have ∗: ?P ∈ M ∧ k ?P = ?k using P by − (rule someI[where x = P]; auto)
  with j1 have ∃ m > i. j1 ≥ m ∧ ?P m by auto
  with ‹finite -› have ?j ∈ {j. i ≤ j ∧ j ≤ j1 ∧ ?P j} by − (rule Max-in; auto)
  have k: k Q > j1 ∧ Q (k Q) if Q ∈ M for Q
  proof −
    from assms(1) ‹Q ∈ M› obtain m where m ≥ Suc j1 Q m by auto
    then have m > j1 Q m by auto
    then show k Q > j1 ∧ Q (k Q) unfolding k-def by − (rule LeastI; blast)
  qed
  with ∗ ‹?j ∈ -› have ?P ?k ?j < ?k by fastforce+
  have ¬ ?P m if ?j < m m < ?k for m
  proof (rule ccontr, simp)
    assume ?P m
    have m > j1
    proof (rule ccontr)
      assume ¬ j1 < m
      with ‹?j < m› ‹?j ∈ -› have i ≤ m m ≤ j1 by auto
      with ‹?P m› ‹finite -› have ?j ≥ m by − (rule Max-ge; auto)
      with ‹?j < m› show False by simp
    qed
    with ‹?P m› ‹finite -› have k ?P ≤ m unfolding k-def by (auto intro: Least-le)
    with ∗ ‹m < ?k› show False by auto
  qed
  moreover have ∃ m ≤ ?k. ?j < m ∧ Q m if Q ∈ M for Q
```

proof −
from $k[OF\ ‹Q \in M›]$ **have** $k\ Q > j1 \wedge Q\ (k\ Q)$ **.**
moreover with $‹\mathit{finite}\ \text{-}›\ ‹Q \in M›$ **have** $k\ Q \leq ?k$ **by** − $(\mathit{rule\ Max\text{-}ge};\ \mathit{auto})$
moreover with $‹?j \in \text{-}›\ ‹k\ Q > \text{-} \wedge \text{-}›$ **have** $?j < k\ Q$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
ultimately show
$\exists j{\geq}i.\exists k{>}j.\exists\ P \in M.\ P\ j \wedge P\ k \wedge (\forall\ m < k.\ j < m \longrightarrow \neg\ P\ m)$
$\wedge\ (\forall\ Q \in M.\ \exists\ m \leq k.\ j < m \wedge Q\ m)$
using $‹?j < ?k›\ ‹?j \in \text{-}›\ ‹?P\ ?k› *$ **by** $(\mathit{inst\text{-}existentials}\ ?j\ ?k\ ?P;\ \mathit{blast})$
qed

lemma $\mathcal{R}$*-divergent-divergent*:
assumes *in-S*: $\mathit{pred\text{-}stream}\ (\lambda\ u.\ u \in V)\ \mathit{xs}$
and *div*: $\mathcal{R}\text{-}\mathit{div}\ (\mathit{smap}\ (\lambda\ u.\ [u]_{\mathcal{R}})\ \mathit{xs})$
and *trans*: *stream-trans xs*
and *trans'*: *pairwise trans' xs*
and *unbounded-not-const*:
$\forall u.\ (\forall c{\in}\mathcal{X}.\ \mathit{real}\ (k\ c) < u\ c) \longrightarrow \neg\ \mathit{ev}\ (\mathit{alw}\ (\lambda \mathit{xs}.\ \mathit{shd}\ \mathit{xs} = u))\ \mathit{xs}$
shows *divergent xs*
unfolding *divergent-def* **proof**
fix *t*
from *pairwise-trans-eq-elapsed*[*OF trans in-S*] **have** *eq-elapsed*: *pairwise eq-elapsed xs* **.**
define *X1* **where** $X1 = \{x.\ x \in \mathcal{X} \wedge (\exists i.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}}))\}$
let $?i = \mathit{Max}\ \{(\mathit{SOME}\ i.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}}))\ |\ x.\ x \in \mathcal{X}\}$
from *finite*(*1*) *non-empty* **have**
$?i \in \{(\mathit{SOME}\ i.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}}))\ |\ x.\ x \in \mathcal{X}\}$
by (*intro Max-in*) *auto*
have $\mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}})$ **if** $x \in X1\ j \geq ?i$ **for** *x j*
proof −
have $X1 \subseteq \mathcal{X}$ **unfolding** *X1-def* **by** *auto*
with *finite*(*1*) *non-empty* $‹x \in X1›$ **have** $*$:
$?i \geq (\mathit{SOME}\ i.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}}))$ (**is** $?i \geq ?k$)
by (*intro Max-ge*) *auto*
from $‹x \in X1›$ **have** $\exists\ k.\ \forall\ j \geq k.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}})$ **by** (*auto simp*: *X1-def*)
then have $\forall\ j \geq ?k.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}})$ **by** (*rule someI-ex*)
moreover from $‹j \geq ?i›\ ‹?i \geq \text{-}›$ **have** $j \geq ?k$ **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
then obtain *i* **where** *unbounded*: $\forall\ x \in X1.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}})$
using *finite* **by** *auto*
show $\exists\ n.\ t < \mathit{dur}\ \mathit{xs}\ n$
proof $(\mathit{cases}\ \forall\ x \in \mathcal{X}.\ (\exists i.\ \forall\ j \geq i.\ \mathit{unbounded}\ x\ ([\mathit{xs}\ !!\ j]_{\mathcal{R}})))$
case *True*
then have $X1 = \mathcal{X}$ **unfolding** *X1-def* **by** *auto*
have $\exists k{\geq}j.\ 0.5 \leq \mathit{dur}\ \mathit{xs}\ k - \mathit{dur}\ \mathit{xs}\ j$ **for** *j*
proof −
let $?u = \mathit{xs}\ !!\ \mathit{max}\ i\ j$
from *in-S* **have** $?u \in [?u]_{\mathcal{R}}\ [?u]_{\mathcal{R}} \in \mathcal{R}$
by (*auto simp*: *stream.pred-set*)
moreover from *unbounded* $‹X1 = \mathcal{X}›$ **have**
$\forall\ x \in \mathcal{X}.\ \mathit{unbounded}\ x\ ([?u]_{\mathcal{R}})$
by *force*
ultimately have $\forall\ x \in \mathcal{X}.\ ?u\ x > k\ x$
by (*auto intro*: *unbounded-all*)
with *unbounded-not-const* **have** $\neg\ \mathit{ev}\ (\mathit{alw}\ (\mathit{HLD}\ \{?u\}))\ \mathit{xs}$
unfolding *HLD-iff* **by** *simp*
then obtain *r* **where**
$r \geq \mathit{max}\ i\ j\ \mathit{xs}\ !!\ r \neq \mathit{xs}\ !!\ \mathit{Suc}\ r$
apply *atomize-elim*
apply (*simp add*: *not-ev-iff not-alw-iff*)

apply (*drule alw-sdrop*[**where** $n = max\ i\ j$])
apply (*drule alwD*)
apply (*subst* (*asm*) (*3*) *stream.collapse*[*symmetric*])
apply *simp*
apply (*drule ev-neq-start-implies-ev-neq*[*simplified comp-def*])
using *stream.collapse*[*of sdrop* (*max i j*) *xs*] **by** (*auto 4 3 elim*: *ev-sdropD*)
let $?k = Suc\ r$
from *in-S* **have** $xs\ !!\ ?k \in V$ **using** *snth-sset* **unfolding** *stream.pred-set* **by** *blast*
with *in-S* **have** $*$:
$xs\ !!\ r \in [xs\ !!\ r]_{\mathcal{R}}\ [xs\ !!\ r]_{\mathcal{R}} \in \mathcal{R}$
$xs\ !!\ ?k \in [xs\ !!\ ?k]_{\mathcal{R}}\ [xs\ !!\ ?k]_{\mathcal{R}} \in \mathcal{R}$
by (*auto simp*: *stream.pred-set*)
from ‹$r \geq$ -› **have** $r \geq i\ ?k \geq i$ **by** *auto*
with *unbounded* ‹$X1 = \mathcal{X}$› **have**
$\forall\ x \in \mathcal{X}.\ unbounded\ x\ ([xs\ !!\ r]_{\mathcal{R}})\ \forall\ x \in \mathcal{X}.\ unbounded\ x\ ([xs\ !!\ ?k]_{\mathcal{R}})$
by (*auto simp del*: *snth.simps*(*2*))
with *in-S* **have** $\forall\ x \in \mathcal{X}.\ (xs\ !!\ r)\ x > k\ x\ \forall\ x \in \mathcal{X}.\ (xs\ !!\ ?k)\ x > k\ x$
using $*$ **by** (*auto intro*: *unbounded-all*)
moreover from *trans′* **have** *trans′* ($xs\ !!\ r$) ($xs\ !!\ ?k$)
using *pairwise-Suc* **by** *auto*
ultimately have $(xs\ !!\ ?k) = (xs\ !!\ r) \oplus 0.5$
unfolding *trans′-def* **using** ‹$xs\ !!\ r \neq$ -› **by** *auto*
moreover from *pairwise-Suc*[*OF eq-elapsed*] **have** *eq-elapsed* ($xs\ !!\ r$) ($xs\ !!\ ?k$)
by *auto*
ultimately have
$dur\ xs\ ?k - dur\ xs\ r = 0.5$
using *non-empty* **by** (*auto simp*: *cval-add-def dur-Suc elapsed-eq*)
with *dur-mono*[*of j r xs*] ‹$r \geq max\ i\ j$› **have** $dur\ xs\ ?k - dur\ xs\ j \geq 0.5$
by *auto*
with ‹$r \geq max\ i\ j$› **show** *?thesis* **by** − (*rule exI*[**where** $x = ?k$]; *auto*)
qed
then show *?thesis* **by** − (*rule dur-ev-exceedsI*[**where** $d = 0.5$]; *auto*)
next
case *False*
define *X2* **where** $X2 = \mathcal{X} - X1$
from *False* **have** $X2 \neq \{\}$ **unfolding** *X1-def X2-def* **by** *fastforce*
have *inf-resets*:
$\forall i.\ (\exists j{\geq}i.\ zero\ x\ ([xs\ !!\ j]_{\mathcal{R}})) \wedge (\exists j{\geq}i.\ \neg\ zero\ x\ ([xs\ !!\ j]_{\mathcal{R}}))$ **if** $x \in X2$ **for** x
using *that div* **unfolding** *X1-def X2-def* $\mathcal{R}$*-div-def* **by** *fastforce*
have $\exists\ j \geq i.\ \exists\ k > j.\ \exists\ x \in X2.\ zero\ x\ ([xs\ !!\ j]_{\mathcal{R}}) \wedge zero\ x\ ([xs\ !!\ k]_{\mathcal{R}})$
$\wedge\ (\forall\ m.\ j < m \wedge m < k \longrightarrow \neg\ zero\ x\ ([xs\ !!\ m]_{\mathcal{R}}))$
$\wedge\ (\forall\ x \in X2.\ \exists\ m.\ j < m \wedge m \leq k \wedge zero\ x\ ([xs\ !!\ m]_{\mathcal{R}}))$
$\wedge\ (\forall\ x \in X1.\ \forall\ m \geq j.\ unbounded\ x\ ([xs\ !!\ m]_{\mathcal{R}}))$ **for** i
proof −
from *unbounded* **obtain** i' **where** i': $\forall\ x \in X1.\ \forall\ m \geq i'.\ unbounded\ x\ ([xs\ !!\ m]_{\mathcal{R}})$ **by** *auto*
then obtain i' **where** i':
$i' \geq i\ \forall\ x \in X1.\ \forall\ m \geq i'.\ unbounded\ x\ ([xs\ !!\ m]_{\mathcal{R}})$
by (*cases* $i' \geq i$; *auto*)
from *finite*(*1*) **have** *finite X2* **unfolding** *X2-def* **by** *auto*
with ‹$X2 \neq \{\}$› $\mathcal{R}$*-divergent-divergent-aux2*[**where** $M = \{\lambda\ i.\ zero\ x\ ([xs\ !!\ i]_{\mathcal{R}})\ \mid x.\ x \in X2\}$]
inf-resets
have $\exists j{\geq}i'.\ \exists k{>}j.\ \exists P{\in}\{\lambda i.\ zero\ x\ ([xs\ !!\ i]_{\mathcal{R}})\ |x.\ x \in X2\}.\ P\ j \wedge P\ k$
$\wedge\ (\forall m{<}k.\ j < m \longrightarrow \neg\ P\ m) \wedge (\forall\ Q{\in}\{\lambda i.\ zero\ x\ ([xs\ !!\ i]_{\mathcal{R}})\ |x.\ x \in X2\}.\ \exists m{\leq}k.\ j < m \wedge Q\ m)$
by *force*
then obtain $j\ k\ x$ **where**
$j \geq i'\ k > j\ x \in X2\ zero\ x\ ([xs\ !!\ j]_{\mathcal{R}})\ zero\ x\ ([xs\ !!\ k]_{\mathcal{R}})$
$\forall m.\ j < m \wedge m < k \longrightarrow \neg\ zero\ x\ ([xs\ !!\ m]_{\mathcal{R}})$
$\forall\ Q{\in}\{\lambda i.\ zero\ x\ ([xs\ !!\ i]_{\mathcal{R}})\ |x.\ x \in X2\}.\ \exists m{\leq}k.\ j < m \wedge Q\ m$
by *auto*
moreover from *this*(*7*) **have** $\forall x{\in}X2.\ \exists\ m \leq k.\ j < m \wedge zero\ x\ ([xs\ !!\ m]_{\mathcal{R}})$ **by** *auto*
ultimately show *?thesis* **using** i'

by (*inst-existentials j k x*) *auto*
qed
moreover have $\exists\ j' \geq j.\ dur\ xs\ j' - dur\ xs\ i \geq 0.5$
if x: $x \in X2$ $i < j$ *zero* x $([xs\ !!\ i]_{\mathcal{R}})$ *zero* x $([xs\ !!\ j]_{\mathcal{R}})$
and *not-reset*: $\forall\ m.\ i < m \wedge m < j \longrightarrow \neg$ *zero* x $([xs\ !!\ m]_{\mathcal{R}})$
and *X2*: $\forall\ x \in X2.\ \exists\ m.\ i < m \wedge m \leq j \wedge$ *zero* x $([xs\ !!\ m]_{\mathcal{R}})$
and *X1*: $\forall\ x \in X1.\ \forall\ m \geq i.$ *unbounded* x $([xs\ !!\ m]_{\mathcal{R}})$
for $x\ i\ j$
proof –
have $\exists j'>j.\ \neg$ *zero* x $([xs\ !!\ j']_{\mathcal{R}})$
proof –
from *inf-resets*[*OF* $x(1)$] **obtain** j' **where** $j' \geq Suc\ j$ $\neg$ *zero* x $([xs\ !!\ j']_{\mathcal{R}})$ **by** *auto*
then show *?thesis* **by** – (*rule exI*[**where** $x = j'$]; *auto*)
qed
from *inf-resets*[*OF* $x(1)$] **obtain** j' **where** $j' \geq Suc\ j$ $\neg$ *zero* x $([xs\ !!\ j']_{\mathcal{R}})$ **by** *auto*
with *nat-eventually-critical-path*[*OF* $x(4)$ *this*(2)]
obtain j' **where** j':
$j' > j$ $\neg$ *zero* x $([xs\ !!\ j']_{\mathcal{R}})$ $\forall\ m \geq j.\ m < j' \longrightarrow$ *zero* x $([xs\ !!\ m]_{\mathcal{R}})$
by *auto*
from ‹$x \in X2$› **have** $x \in \mathcal{X}$ **unfolding** *X2-def* **by** *simp*
with ‹$i < j$› *not-reset not-reset-dur* ‹*stream-trans* -› *in-S pairwise-Suc*[*OF eq-elapsed*] **have**
$dur\ xs\ (j - 1) - dur\ xs\ i = (xs\ !!\ (j - 1))\ x - (xs\ !!\ i)\ x$ (**is** $?d1 = ?d2$)
by (*auto simp*: *stream.pred-set*)
moreover from ‹*zero* x $([xs\ !!\ i]_{\mathcal{R}})$› *in-S* **have** $(xs\ !!\ i)\ x = 0$
by (*auto intro*: *zeroD simp*: *stream.pred-set*)
ultimately have
$dur\ xs\ (j - 1) - dur\ xs\ i = (xs\ !!\ (j - 1))\ x$ (**is** $?d1 = ?d2$)
by *simp*
show *?thesis*
proof (*cases* $?d1 \geq 0.5$)
case *True*

with *dur-mono*[*of* $j - 1\ j\ xs$] **have**
$5\ /\ 10 \leq dur\ xs\ j - dur\ xs\ i$
by *simp*
then show *?thesis* **by** *blast*
next
case *False*
have *j-c-bound*: $(xs\ !!\ j)\ c \leq ?d2$ **if** $c \in X2$ **for** c
proof (*cases* $(xs\ !!\ j)\ c = 0$)
case *True*
from *in-S* ‹$j >$ -› *True* ‹$x \in \mathcal{X}$› **show** *?thesis* **by** (*auto simp*: *V-def stream.pred-set*)
next
case *False*
from *X2* ‹$c \in X2$› *in-S* **have** $\exists k>i.\ k \leq j \wedge (xs\ !!\ k)\ c = 0$
by (*force simp*: *zeroD stream.pred-set*)
with *False* **have**
$\exists k>i.\ k \leq j - Suc\ 0 \wedge (xs\ !!\ k)\ c = 0$
by (*metis Suc-le-eq Suc-pred linorder-neqE-nat not-less not-less-zero*)
moreover from *that* **have** $c \in \mathcal{X}$ **by** (*auto simp*: *X2-def*)
moreover from *not-reset in-S* ‹$x \in \mathcal{X}$› **have**
$\forall k>i.\ k \leq j - 1 \longrightarrow (xs\ !!\ k)\ x \neq 0$
by (*auto simp*: *zeroI stream.pred-set*)
ultimately have
$(xs\ !!\ (j - 1))\ c \leq ?d2$
using *trans in-S* ‹- $x = 0$› ‹$x \in \mathcal{X}$›
by (*auto intro*: *$\mathcal{R}$-divergent-divergent-aux that simp*: *stream.pred-set*)
moreover from
trans-not-delay-mono[*OF pairwise-Suc*[*OF trans*], *of* $j - 1$]
‹$x \in \mathcal{X}$› ‹$c \in \mathcal{X}$› ‹$j >$ -› *in-S* $x(4)$
have $(xs\ !!\ j)\ c \leq (xs\ !!\ (j - 1))\ c$ **by** (*auto simp*: *zeroD stream.pred-set*)

```
  ultimately show ?thesis by auto
qed
moreover from False ‹?d1 = ?d2› have ?d2 < 1 by auto
moreover from in-S have (xs !! j) c ≥ 0 if c ∈ 𝒳 for c
  using that by (auto simp: V-def stream.pred-set)
ultimately have frac-bound: frac ((xs !! j) c) ≤ ?d2 if c ∈ X2 for c
  using that frac-le-1I by (force simp: X2-def)

let ?u = (xs !! j)
from in-S have [xs !! j]ℛ ∈ ℛ by (auto simp: stream.pred-set)
then obtain I r where region:
  [xs !! j]ℛ = region 𝒳 I r valid-region 𝒳 k I r
  unfolding ℛ-def by auto
let ?S = {frac (?u c) | c. c ∈ 𝒳 ∧ isIntv (I c)}
have 𝒳-X2: c ∈ X2 if c ∈ 𝒳 isIntv (I c) for c
proof –
  from X1 ‹j > i› have ∀x∈X1. unbounded x ([xs !! j]ℛ) by auto
  with unbounded-Greater[OF region(2) ‹c ∈ 𝒳›] region(1) that(2) have c ∉ X1 by auto
  with ‹c ∈ 𝒳› show c ∈ X2 unfolding X2-def by auto
qed
have frac-bound: frac ((xs !! j) c) ≤ ?d2 if c ∈ 𝒳 isIntv (I c) for c
  using frac-bound[OF 𝒳-X2] that .
have dur xs (j' − 1) = dur xs j using j' ‹x ∈ 𝒳› in-S eq-elapsed
  by (subst dur-zero-tail[where ω = smap (λ u. [u]ℛ) xs])
    (auto dest: pairwise-Suc simp: stream.pred-set)
moreover from dur-reset[OF eq-elapsed in-S, of x j − 1] ‹x ∈ 𝒳› x(4) ‹j > -› have
  dur xs j = dur xs (j − 1)
  by (auto simp: stream.pred-set)
ultimately have dur xs (j' − 1) = dur xs (j − 1) by auto
moreover have dur xs j' − dur xs (j' − 1) ≥ (1 − ?d2) / 2
proof –
  from ‹j' > -› have j' > 0 by auto
  with pairwise-Suc[OF trans', of j' − 1] have
    trans' (xs !! (j' − 1)) (xs !! j')
    by auto
  moreover from j' have
    (xs !! (j' − 1)) x = 0 (xs !! j') x > 0
    using in-S ‹x ∈ 𝒳› by (force intro: zeroD dest: not-zeroD simp: stream.pred-set)+
  moreover note delayedR-aux = calculation
  obtain t where
    (xs !! j') = (xs !! (j' − 1)) ⊕ t t ≥ (1 − ?d2) / 2 t ≥ 0
  proof –
    from in-S have [xs !! j']ℛ ∈ ℛ by (auto simp: stream.pred-set)
    then obtain I' r' where region':
      [xs !! j']ℛ = region 𝒳 I' r' valid-region 𝒳 k I' r'
      unfolding ℛ-def by auto
    let ?S' = {frac ((xs !! (j' − 1)) c) |c. c ∈ 𝒳 ∧ Regions.isIntv (I' c)}

    from finite(1) have ?d2 ≥ Max (?S' ∪ {0})
      apply –
      apply (rule Max.boundedI)
        apply fastforce
       apply fastforce
      apply safe
      subgoal premises prems for - c d
      proof –
        from j' have (xs !! (j' − 1)) c = ?u c ∨ (xs !! (j' − 1)) c = 0
          by (intro resets-mono'[OF eq-elapsed in-S trans - ‹x ∈ 𝒳› ‹c ∈ 𝒳›]; auto)
        then show ?thesis
        proof (standard, goal-cases)
          case A: 1
```

```
        show ?thesis
        proof (cases c ∈ X1)
          case True
          with X1 ‹j' > j› ‹j > i› have unbounded c ([xs !! j']_ℛ) by auto
          with region' ‹c ∈ 𝒳› have I' c = Greater (k c)
            by (auto intro: unbounded-Greater)
          with prems show ?thesis by auto
        next
          case False
          with ‹c ∈ 𝒳› have c ∈ X2 unfolding X2-def by auto
          with j-c-bound have mono: (xs !! j) c ≤ (xs !! (j − 1)) x .
          from in-S ‹c ∈ 𝒳› have (xs !! (j' − 1)) c ≥ 0
            unfolding V-def stream.pred-set by auto
          then have
            frac ((xs !! (j' − 1)) c) ≤ (xs !! (j' − 1)) c
            using frac-le-self by auto
          with A mono show ?thesis by auto
        qed
      next
        case prems: 2

        have frac (0 :: real) = (0 :: real) by auto
        then have frac (0 :: real) ≤ (0 :: real) by linarith
        moreover from in-S ‹x ∈ 𝒳› have (xs !! (j − 1)) x ≥ 0
          unfolding V-def stream.pred-set by auto
        ultimately show ?thesis using prems by auto
      qed
    qed
    using in-S ‹x ∈ 𝒳› by (auto simp: V-def stream.pred-set)
  then have le: (1 − ?d2) / 2 ≤ (1 − Max (?S' ∪ {0})) / 2 by simp

  let ?u = xs !! j'
  let ?u' = xs !! (j' − 1)
  from in-S have ∗: ?u' ∈ V [?u']_ℛ ∈ ℛ ?u ∈ V [?u]_ℛ ∈ ℛ
    by (auto simp: stream.pred-set)
  from pairwise-Suc[OF trans, of j' − 1] ‹j' > j› have
    trans (xs !! (j' − 1)) (xs !! j')
    by auto
  then have Succ:
    [xs !! j']_ℛ ∈ Succ ℛ ([xs !! (j' − 1)]_ℛ) ∧ (∃ t≥ 0. ?u = ?u' ⊕ t)
  proof cases
    case prems: (succ t)
    from ∗ have ?u' ∈ [?u']_ℛ by auto
    with prems ∗ show ?thesis by auto
  next
    case (reset l)
    with ‹?u' ∈ V› have ?u x ≤ ?u' x by (cases x ∈ set l) (auto simp: V-def)
    from j' have zero x ([?u']_ℛ) by auto
    with ‹?u' ∈ V› have ?u' x = 0 unfolding zero-def by auto
    with ‹?u x ≤ -› ‹?u x > 0› show ?thesis by auto
  next
    case id
    with ∗ Succ-refl[of ℛ 𝒳 k, folded ℛ-def, OF - finite(1)] show ?thesis
      unfolding cval-add-def by auto
  qed
  then obtain t where t: ?u = xs !! (j' − 1) ⊕ t t ≥ 0 by auto
  note Succ = Succ[THEN conjunct1]

  show ?thesis
  proof (cases ∃ c ∈ X2. ∃ d :: nat. ?u c = d)
    case True
```

```
  from True obtain c and d :: nat where c:
    c ∈ 𝒳 c ∈ X2 ?u c = d
    by (auto simp: X2-def)
  have ?u x > 0 by fact
  from pairwise-Suc[OF eq-elapsed, of j′ − 1] ‹j′ > j› have
    eq-elapsed (xs !! (j′ − 1)) ?u
    by auto
  moreover from
    elapsed-eq[OF this ‹x ∈ 𝒳›] ‹(xs !! (j′ − 1)) x = 0› ‹(xs !! j′) x > 0›
  have elapsed (xs !! (j′ − 1)) (xs !! j′) > 0
    by auto
  ultimately have
    ?u c − (xs !! (j′ − 1)) c > 0
    using ‹c ∈ 𝒳› unfolding eq-elapsed-def by auto
  moreover from in-S have xs !! (j′ − 1) ∈ V by (auto simp: stream.pred-set)
  ultimately have ?u c > 0 using ‹c ∈ 𝒳› unfolding V-def by auto
  from region′ in-S ‹c ∈ 𝒳› have intv-elem c ?u (I′ c)
    by (force simp: stream.pred-set)
  with ‹?u c = d› ‹?u c > 0› have ?u c ≥ 1 by auto
  moreover have (xs !! (j′ − 1)) c ≤ 0.5
  proof −
    have (xs !! (j′ − 1)) c ≤ (xs !! j) c

      using j′(1,3)
      by (auto intro: resets-mono[OF eq-elapsed in-S trans - ‹x ∈ 𝒳› ‹c ∈ 𝒳›])
    also have ... ≤ ?d2 using j-c-bound[OF ‹c ∈ X2›] .
    also from ‹?d1 = ?d2› ‹¬ 5 / 10 ≤ -› have ... ≤ 0.5 by simp
    finally show ?thesis .
  qed
  moreover have ?d2 ≥ 0 using in-S ‹x ∈ 𝒳› by (auto simp: V-def stream.pred-set)
  ultimately have ?u c − (xs !! (j′ − 1)) c ≥ (1 − ?d2) / 2 by auto
  with t have t ≥ (1 − ?d2) / 2 unfolding cval-add-def by auto
  with t show ?thesis by (auto intro: that)
next
  case F: False
  have not-const: ¬ isConst (I′ c) if c ∈ 𝒳 for c
  proof (rule ccontr, simp)
    assume A: isConst (I′ c)
    show False
    proof (cases c ∈ X1)
      case True
      with X1 ‹j′ > j› ‹j > -› have unbounded c ([xs !! j′]ℛ) by auto
      with unbounded-Greater ‹c ∈ 𝒳› region′ have isGreater (I′ c) by force
      with A show False by auto
    next
      case False
      with ‹c ∈ 𝒳› have c ∈ X2 unfolding X2-def by auto
      from region′ in-S ‹c ∈ 𝒳› have intv-elem c ?u (I′ c)
        unfolding stream.pred-set by force
      with ‹c ∈ X2› A False F show False by auto
    qed
  qed
  have ∄x. x ≤ k c ∧ (xs !! j′) c = real x if c ∈ 𝒳 for c
  proof (cases c ∈ X2; safe)
    fix d
    assume c ∈ X2 (xs !! j′) c = real d
    with F show False by auto
  next
    fix d
    assume c ∉ X2
    with that have c ∈ X1 unfolding X2-def by auto
```

```
            with X1 ‹j' > j› ‹j > i› have unbounded c ([?u]ℛ) by auto
            from unbounded-all[OF - - this] ‹c ∈ 𝒳› in-S have ?u c > k c
              by (force simp: stream.pred-set)
            moreover assume ?u c = real d d ≤ k c
            ultimately show False by auto
          qed
          with delayedR-aux have
            (xs !! j') = delayedR ([xs !! j']ℛ) (xs !! (j' − 1))
            using ‹x ∈ 𝒳› unfolding trans'-def by auto
          from not-const region'(1)  in-S Succ(1) have
            ∃ t≥0. delayedR ([xs !! j']ℛ) (xs !! (j' − 1)) = xs !! (j' − 1) ⊕ t ∧
                  (1 − Max (?S' ∪ {0})) / 2 ≤ t
            apply simp
            apply (rule delayedR-correct(2)[OF - - region'(2), simplified])
            by (auto simp: stream.pred-set)
          with le ‹- = delayedR - -› show ?thesis by (auto intro: that)
        qed
      qed
      moreover from pairwise-Suc[OF eq-elapsed, of j' − 1] ‹j' > 0› have
        eq-elapsed (xs !! (j' − 1)) (xs !! j')
        by auto
      ultimately show dur xs j' − dur xs (j' − 1) ≥ (1 − ?d2) / 2
        using ‹j' > 0› dur-Suc[of - j' − 1] ‹x ∈ 𝒳› by (auto simp: cval-add-def elapsed-eq)
    qed
    moreover from dur-mono[of i j − 1 xs] ‹i < j› have dur xs i ≤ dur xs (j − 1) by simp
    ultimately have dur xs j' − dur xs i ≥ 0.5 unfolding ‹?d1 = ?d2›[symmetric] by auto
    then show ?thesis using ‹j < j'› by − (rule exI[where x = j']; auto)
  qed
qed
moreover
have ∃ j' ≥ i. dur xs j' − dur xs i ≥ 0.5 for i
proof −
  from calculation(1)[of i] obtain j k x where
    j≥i k>j x∈X2 zero x ([xs !! j]ℛ)
    zero x ([xs !! k]ℛ)
    ∀ m. j < m ∧ m < k ⟶ ¬ zero x ([xs !! m]ℛ)
    ∀ x∈X2. ∃ m>j. m ≤ k ∧ zero x ([xs !! m]ℛ)
    ∀ x∈X1. ∀ m≥j. unbounded x ([xs !! m]ℛ)
    by auto
  from calculation(2)[OF this(3,2,4−8)] obtain j' where
    j'≥k 5 / 10 ≤ dur xs j' − dur xs j
    by auto
  with dur-mono[of i j xs] ‹j ≥ i› ‹k > j› show ?thesis by (intro exI[where x = j']; auto)
qed
then show ?thesis by − (rule dur-ev-exceedsI[where d = 0.5]; auto)
  qed
qed

lemma cfg-on-div-absc:
  notes in-space-UNIV[measurable]
  assumes cfg ∈ cfg-on-div st st ∈ S
  shows absc cfg ∈ R-G-cfg-on-div (abss st)
proof −
  from assms have ∗: cfg ∈ MDP.cfg-on st state cfg = st div-cfg cfg
    unfolding cfg-on-div-def by auto
  with assms have cfg ∈ valid-cfg by (auto intro: MDP.valid-cfgI)
  have almost-everywhere (MDP.MC.T cfg) (MDP.MC.enabled cfg)
    by (rule MDP.MC.AE-T-enabled)
  moreover from ∗ have AE x in MDP.MC.T cfg. divergent (smap (snd ∘ state) x)
    by (simp add: div-cfg-def)
  ultimately have AE x in MDP.MC.T cfg. ℛ-div (smap (snd ∘ state) (smap absc x))
```

proof *eventually-elim*
case (*elim* ω)
let *?xs* = *smap* (*snd o state*) ω
from *MDP.pred-stream-cfg-on*[*OF* ‹- ∈ *valid-cfg*› ‹*MDP.MC.enabled* - -›] **have** ∗:
pred-stream (λ *x*. *x* ∈ *S*) (*smap state* ω)
by (*auto simp*: *stream.pred-set*)
have [*snd* (*state x*)]$_{\mathcal{R}}$ = *snd* (*abss* (*state x*)) **if** *x* ∈ *sset* ω **for** *x*
proof −
from ∗ *that* **have** *state x* ∈ *S* **by** (*auto simp*: *stream.pred-set*)
then have *snd* (*abss* (*state x*)) = [*snd* (*state x*)]$_{\mathcal{R}}$ **by** (*metis abss-S snd-conv surj-pair*)
then show *?thesis* **..**
qed
then have *smap* (λ*z*. [*snd* (*state z*)]$_{\mathcal{R}}$) ω = (*smap* (λ*z*. *snd* (*abss* (*state z*))) ω) **by** *auto*
from ∗ **have** *pred-stream* (λ *u*. *u* ∈ *V*) *?xs*
apply (*simp add*: *map-def stream.pred-set*)
apply (*subst* (*asm*) *surjective-pairing*)
using *S-V* **by** *blast*
moreover have *stream-trans ?xs*
by (*rule enabled-stream-trans* ‹- ∈ *valid-cfg*› ‹*MDP.MC.enabled* - -›)+
ultimately show *?case* **using** ‹*divergent* -› ‹*smap* - ω = -›
by − (*drule divergent-*$\mathcal{R}$*-divergent*, *auto simp add*: *stream.map-comp state-absc*)
qed
with ‹*cfg* ∈ *valid-cfg*› **have** *R-G-div-cfg* (*absc cfg*) **unfolding** *R-G-div-cfg-def*
by (*subst absc-distr-self*) (*auto intro*: *MDP.valid-cfgI simp*: *AE-distr-iff*)
with *R-G.valid-cfgD* ‹*cfg* ∈ *valid-cfg*› ∗ **show** *?thesis* **unfolding** *R-G-cfg-on-div-def* **by** *auto force*
qed

definition
alternating cfg = (*AE* ω *in MDP.MC.T cfg*.
alw (*ev* (*HLD* {*cfg*. ∀ *cfg*′ ∈ *K-cfg cfg*. *fst* (*state cfg*′) = *fst* (*state cfg*)})) ω)

lemma *K-cfg-same-loc-iff*:
(∀ *cfg*′∈ *K-cfg cfg*. *fst* (*state cfg*′) = *fst* (*state cfg*))
⟷ (∀ *cfg*′∈ *K-cfg* (*absc cfg*). *fst* (*state cfg*′) = *fst* (*state* (*absc cfg*)))
if *cfg* ∈ *valid-cfg*
using *that* **by** (*auto simp*: *state-absc fst-abss K-cfg-map-absc*)

lemma (**in** −) *stream-all2-flip*:
stream-all2 (λ*a b*. *R b a*) *xs ys* = *stream-all2 R ys xs*
by (*standard*; *coinduction arbitrary*: *xs ys*; *auto dest*: *sym*)

lemma *AE-alw-ev-same-loc-iff*:
assumes *cfg* ∈ *valid-cfg*
shows *alternating cfg* ⟷ *alternating* (*absc cfg*)
unfolding *alternating-def*
apply (*simp add*: *MDP.MC.T.AE-iff-emeasure-eq-1*)
subgoal
proof −
show *?thesis* (**is** (*?x* = *1*) = (*?y* = *1*))
proof −
have ∗: *stream-all2* (λ*s t*. *t* = *absc s*) *x y* = *stream-all2* (=) *y* (*smap absc x*) **for** *x y*
by (*subst stream-all2-flip*) *simp*
have *?x* = *?y*
apply (*rule T-eq-rel-half*[**where** *f* = *absc* **and** *S* = *valid-cfg*, *OF HOL.refl*, *rotated 2*])
subgoal
apply (*simp add*: *space-stream-space rel-set-strong-def*)
apply (*intro allI impI*)
apply (*frule stream.rel-mono-strong*[**where** *Ra* = λ*s t*. *t* = *absc s*])
by (*auto simp*: ∗ *stream.rel-eq stream-all2-refl alw-holds-pred-stream-iff*[*symmetric*]
K-cfg-same-loc-iff HLD-def comp-def elim!: *alw-ev-cong*)

```
      subgoal
        by (rule rel-funI) (auto intro!: rel-pmf-reflI simp: pmf.rel-map(2) K-cfg-map-absc)
      using ‹cfg ∈ valid-cfg› by simp+
    then show ?thesis
      by simp
  qed
qed
done

lemma AE-alw-ev-same-loc-iff':
  assumes cfg ∈ R-G.cfg-on (abss st) st ∈ S
  shows alternating cfg ⟷ alternating (repcs st cfg)
proof –
  from assms have cfg ∈ R-G.valid-cfg
    by (auto intro: R-G.valid-cfgI)
  with assms show ?thesis
    by (subst AE-alw-ev-same-loc-iff) (auto simp: absc-repcs-id)
qed

lemma (in –) cval-add-non-id:
  False if b ⊕ d = b d > 0 for d :: real
proof –
  from that(1) have (b ⊕ d) x = b x
    by (rule fun-cong)
  with ‹d > 0› show False
    unfolding cval-add-def by simp
qed

lemma repcs-unbounded-AE-non-loop-end-strong:
  assumes cfg ∈ R-G.cfg-on (abss st) st ∈ S
    and alternating cfg
  shows AE ω in MDP.MC.T (repcs st cfg).
      (∀ u :: ('c ⇒ real). (∀ c ∈ 𝒳. u c > real (k c)) ⟶
      ¬ (ev (alw (λ xs. shd xs = u))) (smap (snd o state) ω)) (is AE ω in ?M. ?P ω)
proof –
  from assms have cfg ∈ R-G.valid-cfg
    by (auto intro: R-G.valid-cfgI)
  with assms(1) have repcs st cfg ∈ valid-cfg
    by auto
  from R-G.valid-cfgD[OF ‹cfg ∈ R-G.valid-cfg›] have cfg ∈ R-G.cfg-on (state cfg) .
  let ?U = λ u. ⋃ l ∈ L. {μ ∈ K (l, u). μ ≠ return-pmf (l, u) ∧ (∀ x ∈ μ. fst x = l)}
  let ?r = λ u. Sup ({0} ∪ (λ μ. measure-pmf μ {x. snd x = u}) ‘ ?U u)
  have lt-1: ?r u < 1 for u
  proof –
    have ∗: emeasure (measure-pmf μ) {x. snd x = u} < 1
      if μ ≠ return-pmf (l, u) ∀x∈set-pmf μ. fst x = l for μ and l :: 's
    proof (rule ccontr)
      assume ¬ emeasure (measure-pmf μ) {x. snd x = u} < 1
      then have 1 = emeasure (measure-pmf μ) {x. snd x = u}
        using measure-pmf.emeasure-ge-1-iff by force
      also from that(2) have ... ≤ emeasure (measure-pmf μ) {(l, u)}
        by (subst emeasure-Int-set-pmf[symmetric]) (auto intro!: emeasure-mono)
      finally show False
        by (simp add: measure-pmf.emeasure-ge-1-iff measure-pmf-eq-1-iff that(1))
    qed
    let ?S =
      {map-pmf (λ (X, l). (l, ([X := 0]u))) μ | μ l g. (l, g, μ) ∈ trans-of A}
    have (λ μ. measure-pmf μ {x. snd x = u}) ‘ ?U u
      ⊆ {0, 1} ∪ (λ μ. measure-pmf μ {x. snd x = u}) ‘ ?S
      by (force elim!: K.cases)
    moreover have finite ?S
```

proof −
have *?S* ⊆ (λ (*l*, *g*, μ). *map-pmf* (λ (*X*, *l*). (*l*, ([*X* := *0*]*u*))) μ) ' *trans-of A*
by *force*
also from *finite*(*3*) **have** *finite* . . . **..**
finally show *?thesis* **.**
qed
ultimately have *finite* ((λ μ. *measure-pmf* μ {*x*. *snd x* = *u*}) ' *?U u*)
by (*auto intro*: *finite-subset*)
then show *?thesis*
by (*fastforce intro*: ∗ *finite-imp-Sup-less*)
qed
{ **fix** *l* :: ′*s* **and** *u* :: ′*c* ⇒ *real* **and** *cfg* :: (′*s* × (′*c* ⇒ *real*) *set*) *cfg*
assume *unbounded*: ∀ *c* ∈ 𝒳. *u c* > *k c* **and** *cfg* ∈ *R-G.cfg-on* (*abss* (*l*, *u*)) *abss* (*l*, *u*) ∈ 𝒮
and *same-loc*: ∀ *cfg*′ ∈ *K-cfg cfg*. *fst* (*state cfg*′) = *l*
then have *cfg* ∈ *R-G.valid-cfg repcs* (*l*, *u*) *cfg* ∈ *valid-cfg*
by (*auto intro*: *R-G.valid-cfgI*)
then have *cfg-on*: *repcs* (*l*, *u*) *cfg* ∈ *MDP.cfg-on* (*l*, *u*)
by (*auto dest*: *MDP.valid-cfgD*)
from ‹*cfg* ∈ *R-G.cfg-on* -› **have** *action cfg* ∈ 𝒦 (*abss* (*l*, *u*))
by (*rule R-G.cfg-onD-action*)

have *K-cfg-rept*: *state* ' *K-cfg* (*repcs* (*l*, *u*) *cfg*) = *rept* (*l*, *u*) (*action cfg*)
unfolding *K-cfg-def* **by** (*force simp*: *action-repcs*)
have *l* ∈ *L*
using *MDP.valid-cfg-state-in-S* ‹*repcs* (*l*, *u*) *cfg* ∈ *MDP.valid-cfg*› **by** *fastforce*
moreover have *rept* (*l*, *u*) (*action cfg*) ≠ *return-pmf* (*l*, *u*)
proof (*rule ccontr*, *simp*)
assume *rept* (*l*, *u*) (*action cfg*) = *return-pmf* (*l*, *u*)
then have *action cfg* = *return-pmf* (*abss* (*l*, *u*))
using *abst-rept-id*[*OF* ‹*action cfg* ∈ -›]
by (*simp add*: *abst-def*)
moreover have (*l*, *u*) ∈ *S*
using ‹- ∈ 𝒮› **by** (*auto dest*: 𝒮*-abss-S*)
moreover have *abss* (*l*, *u*) = (*l*, $[u]_{\mathcal{R}}$)
by (*metis abss-S calculation*(*2*))
ultimately show *False*
using ‹*rept* (*l*, *u*) - = -› *unbounded* **unfolding** *rept-def* **by** (*auto dest*: *cval-add-non-id*)
qed
moreover have *rept* (*l*, *u*) (*action cfg*) ∈ *K* (*l*, *u*)
proof −
have *action* (*repcs* (*l*, *u*) *cfg*) ∈ *K* (*l*, *u*)
using *cfg-on* **by** *blast*
then show *?thesis*
by (*simp add*: *repcs-def*)
qed
moreover have ∀ *x*∈*set-pmf* (*rept* (*l*, *u*) (*action cfg*)). *fst x* = *l*
using *same-loc K-cfg-same-loc-iff*[*of repcs* (*l*, *u*) *cfg*]
‹*repcs* (*l*, *u*) - ∈ *valid-cfg*› ‹*cfg* ∈ *R-G.valid-cfg*› ‹*cfg* ∈ *R-G.cfg-on* -›
by (*simp add*: *absc-repcs-id fst-abss K-cfg-rept*[*symmetric*])
ultimately have *rept* (*l*, *u*) (*action cfg*) ∈ *?U u*
by *blast*
then have *measure-pmf* (*rept* (*l*, *u*) (*action cfg*)) {*x*. *snd x* = *u*} ≤ *?r u*
by (*fastforce intro*: *Sup-upper*)
moreover have *rept* (*l*, *u*) (*action cfg*) = *action* (*repcs* (*l*, *u*) *cfg*)
by (*simp add*: *repcs-def*)
ultimately have *measure-pmf* (*action* (*repcs* (*l*, *u*) *cfg*)) {*x*. *snd x* = *u*} ≤ *?r u*
by *auto*
}
note ∗ = *this*
let *?S* = {*cfg*. ∃ *cfg*′ *s*. *cfg*′ ∈ *R-G.valid-cfg* ∧ *cfg* = *repcs s cfg*′ ∧ *abss s* = *state cfg*′}
have *start*: *repcs st cfg* ∈ *?S*

```
    using ‹cfg ∈ R-G.valid-cfg› assms unfolding R-G-cfg-on-div-def
    by clarsimp (inst-existentials cfg fst st snd st, auto)
  have step: y ∈ ?S if y ∈ K-cfg x x ∈ ?S for x y
    using that apply safe
    subgoal for cfg' l u
      apply (inst-existentials absc y state y)
      subgoal
        by blast
      subgoal
        by (metis
            K-cfg-valid-cfgD R-G.valid-cfgD R-G.valid-cfg-state-in-S absc-repcs-id cont-absc-1
            cont-repcs1 repcs-valid
            )
      subgoal
        by (simp add: state-absc)
      done
    done
  have **: x ∈ ?S if (repcs st cfg, x) ∈ MDP.MC.acc for x
  proof −
    from MDP.MC.acc-relfunD[OF that] obtain n where ((λ a b. b ∈ K-cfg a) ^^ n) (repcs st cfg) x .
    then show ?thesis
    proof (induction n arbitrary: x)
      case 0
      with start show ?case
        by simp
    next
      case (Suc n)
      from this(2)[simplified] show ?case
        apply (rule relcomppE)
        apply (erule step)
        apply (erule Suc.IH)
        done
    qed
  qed
  have ***: almost-everywhere (MDP.MC.T (repcs st cfg)) (alw (HLD ?S))
    by (rule AE-mp[OF MDP.MC.AE-T-reachable]) (fastforce dest: ** simp: HLD-iff elim: alw-mono)

  from ‹alternating cfg› assms have alternating (repcs st cfg)
    by (simp add: AE-alw-ev-same-loc-iff'[of - st])
  then have alw-ev-same2: almost-everywhere (MDP.MC.T (repcs st cfg))
     (alw (λω. HLD (state −‘ snd −‘ {u}) ω ⟶
      ev (HLD {cfg. ∀ cfg'∈set-pmf (K-cfg cfg). fst (state cfg') = fst (state cfg)}) ω))
    for u unfolding alternating-def by (auto elim: alw-mono)

  let ?X = {cfg :: ('s × ('c ⇒ real)) cfg. ∀ c ∈ 𝒳. snd (state cfg) c > k c}
  let ?Y = {cfg. ∀ cfg' ∈ K-cfg cfg. fst (state cfg') = fst (state cfg)}

  have (AE ω in ?M. ?P ω) ⟷
    (AE ω in ?M. ∀ u :: ('c ⇒ real).
      (∀ c ∈ 𝒳. u c > k c) ∧ u ∈ snd ‘ state ‘ (MDP.MC.acc ‘‘ {repcs st cfg}) ⟶
      ¬ (ev (alw (λ xs. shd xs = u))) (smap (snd o state) ω)) (is ?L ⟷ ?R)
  proof
    assume ?L
    then show ?R
      by eventually-elim auto
  next
    assume ?R
    with MDP.MC.AE-T-reachable[of repcs st cfg] show ?L
    proof (eventually-elim, intro allI impI notI, goal-cases)
      case (1 ω u)
      then show ?case
```

```
      by − (intro alw-HLD-smap alw-disjoint-ccontr[where
            S = (snd o state) ‘ MDP.MC.acc `` {repcs st cfg}
            and R = {u} and ω = smap (snd o state) ω
            ]; auto simp: HLD-iff comp-def)
    qed
  qed

  also have . . . ⟷
      (∀ u :: ('c ⇒ real).
        (∀ c ∈ 𝒳. u c > k c) ∧ u ∈ snd ‘ state ‘ (MDP.MC.acc `` {repcs st cfg}) ⟶
        (AE ω in ?M. ¬ (ev (alw (λ xs. shd xs = u))) (smap (snd o state) ω)))
    using MDP.MC.countable-reachable[of repcs st cfg]
    by − (rule AE-all-imp-countable,
        auto intro: countable-subset[where B = snd ‘ state ‘ MDP.MC.acc `` {repcs st cfg}])
  also show ?thesis
    unfolding calculation
    apply clarsimp
    subgoal for l u x
      apply (rule
          MDP.non-loop-tail-strong[simplified, of snd snd (state x) ?Y ?S ?r (snd (state x))]
          )
      subgoal
        apply safe
        subgoal premises prems for cfg l1 u1 - cfg' l2 u2
        proof −
          have [simp]: l2 = l1 u2 = u1
            subgoal
              by (metis MDP.cfg-onD-state Pair-inject prems(4) state-repcs)
            subgoal
              by (metis MDP.cfg-onD-state prems(4) snd-conv state-repcs)
            done
          with prems have [simp]: u2 = u
            by (metis ‹(l, u) = state x› ‹snd (l1, u1) = snd (state x)› ‹u2 = u1› snd-conv)
          have [simp]: snd −‘ {snd (state x)} = {y. snd y = snd (state x)}
            by (simp add: vimage-def)
          from prems show ?thesis
            apply simp
            apply (erule ∗[simplified])
            subgoal
              using prems(1) prems(2)[symmetric] prems(3−) by (auto simp: R-G.valid-cfg-def)
            subgoal
              using prems(1) prems(2)[symmetric] prems(3−) by (auto simp: R-G.valid-cfg-def)
            subgoal
              using K-cfg-same-loc-iff[of repcs (l1, snd (state x)) cfg']
              by (simp add: absc-repcs-id) (metis fst-abss fst-conv repcs-valid)
            done
        qed
        done
      subgoal
        by (auto intro: lt-1[simplified])
        apply (rule MDP.valid-cfgD[OF ‹repcs st cfg ∈ valid-cfg›]; fail)
      subgoal
        using ∗∗∗ unfolding alw-holds-pred-stream-iff[symmetric] HLD-def .
      subgoal
        by (rule alw-ev-same2)
      done
    done
qed

lemma cfg-on-div-repcs-strong:
  notes in-space-UNIV[measurable]
```

assumes $cfg \in R\text{-}G\text{-}cfg\text{-}on\text{-}div\ (abss\ st)\ st \in S$ **and** *alternating cfg*
shows $repcs\ st\ cfg \in cfg\text{-}on\text{-}div\ st$
proof −
let $?st = abss\ st$
let $?cfg = repcs\ st\ cfg$
from *assms* **have** ∗:
$cfg \in R\text{-}G.cfg\text{-}on\ ?st\ state\ cfg = ?st\ R\text{-}G\text{-}div\text{-}cfg\ cfg$
unfolding *R-G-cfg-on-div-def* **by** *auto*
with *assms* **have** $cfg \in R\text{-}G.valid\text{-}cfg$ **by** (*auto intro*: *R-G.valid-cfgI*)
with ‹$st \in S$› ‹$- = ?st$› **have** $?cfg \in valid\text{-}cfg$ **by** *auto*
from ∗(*1*) ‹$st \in S$› ‹*alternating cfg*› **have**
$AE\ \omega\ in\ MDP.MC.T\ ?cfg.\ \forall u.\ (\forall c{\in}\mathcal{X}.\ real\ (k\ c) < u\ c) \longrightarrow$
$\neg\ ev\ (alw\ (\lambda xs.\ shd\ xs = u))\ (smap\ (snd \circ state)\ \omega)$
by (*rule repcs-unbounded-AE-non-loop-end-strong*)
— Move to lower level
moreover from ∗(*2*,*3*) **have** $AE\ \omega\ in\ MDP.MC.T\ ?cfg.\ \mathcal{R}\text{-}div\ (smap\ (snd \circ state)\ (smap\ absc\ \omega))$
unfolding *R-G-div-cfg-def*
by (*subst* (*asm*) *R-G-trace-space-distr-eq*[*OF* ‹$cfg \in R\text{-}G.valid\text{-}cfg$›]; *simp add*: *AE-distr-iff*)
ultimately have *div-cfg ?cfg*
unfolding *div-cfg-def* **using** *MDP.MC.AE-T-enabled*[*of ?cfg*]
proof *eventually-elim*
case *prems*: (*elim* ω)
let $?xs = smap\ (snd\ o\ state)\ \omega$
from *MDP.pred-stream-cfg-on*[*OF* ‹$- \in valid\text{-}cfg$› ‹*MDP.MC.enabled - -*›] **have** ∗:
$pred\text{-}stream\ (\lambda\ x.\ x \in S)\ (smap\ state\ \omega)$
by (*auto simp*: *stream.pred-set*)
have $[snd\ (state\ x)]_{\mathcal{R}} = snd\ (abss\ (state\ x))$ **if** $x \in sset\ \omega$ **for** x
proof −
from ∗ *that* **have** $state\ x \in S$ **by** (*auto simp*: *stream.pred-set*)
then have $snd\ (abss\ (state\ x)) = [snd\ (state\ x)]_{\mathcal{R}}$ **by** (*metis abss-S snd-conv surj-pair*)
then show *?thesis* **..**
qed
then have $smap\ (\lambda z.\ [snd\ (state\ z)]_{\mathcal{R}})\ \omega = (smap\ (\lambda z.\ snd\ (abss\ (state\ z)))\ \omega)$ **by** *auto*
from ∗ **have** $pred\text{-}stream\ (\lambda\ u.\ u \in V)\ ?xs$
by (*simp add*: *map-def stream.pred-set*, *subst* (*asm*) *surjective-pairing*, *blast*)
moreover have *stream-trans ?xs*
by (*rule enabled-stream-trans* ‹$- \in valid\text{-}cfg$› ‹*MDP.MC.enabled - -*›)+
moreover have *pairwise trans′ ?xs*
using ‹$- \in R\text{-}G.valid\text{-}cfg$› ‹$state\ cfg = -$›[*symmetric*] ‹*MDP.MC.enabled - -*›
by (*rule enabled-stream-trans′*)
moreover from *prems*(*1*) **have**
$\forall u.\ (\forall c{\in}\mathcal{X}.\ real\ (k\ c) < u\ c) \longrightarrow \neg\ ev\ (alw\ (\lambda xs.\ snd\ (shd\ xs) = u))\ (smap\ state\ \omega)$
by (*simp add*: *comp-def*)
ultimately show *?case* **using** ‹$\mathcal{R}$-*div -*›
by (*simp add*: *stream.map-comp state-absc* ‹$smap\ \text{-}\ \omega = -$› *$\mathcal{R}$-divergent-divergent comp-def*)
qed
with *MDP.valid-cfgD* ‹$cfg \in R\text{-}G.valid\text{-}cfg$› ∗ **show** *?thesis* **unfolding** *cfg-on-div-def* **by** *auto force*
qed

lemma *repcs-unbounded-AE-non-loop-end*:
assumes $cfg \in R\text{-}G.cfg\text{-}on\ (abss\ st)\ st \in S$
shows $AE\ \omega\ in\ MDP.MC.T\ (repcs\ st\ cfg).$
$(\forall\ s :: ('s \times ('c \Rightarrow real)).\ (\forall\ c \in \mathcal{X}.\ snd\ s\ c > k\ c) \longrightarrow$
$\neg\ (ev\ (alw\ (\lambda\ xs.\ shd\ xs = s)))\ (smap\ state\ \omega))$ (**is** $AE\ \omega\ in\ ?M.\ ?P\ \omega$)
proof −
from *assms* **have** $cfg \in R\text{-}G.valid\text{-}cfg$
by (*auto intro*: *R-G.valid-cfgI*)
with *assms*(*1*) **have** $repcs\ st\ cfg \in valid\text{-}cfg$
by *auto*
from *R-G.valid-cfgD*[*OF* ‹$cfg \in R\text{-}G.valid\text{-}cfg$›] **have** $cfg \in R\text{-}G.cfg\text{-}on\ (state\ cfg)$ **.**
let $?K = \lambda\ x.\ \{\mu \in K\ x.\ \mu \neq return\text{-}pmf\ x\}$

```
let ?r = λ x. Sup ((λ μ. measure-pmf μ {x}) ` ?K x)
have lt-1: ?r x < 1 if μ ∈ ?K x for μ x
proof −
  have ∗: emeasure (measure-pmf μ) {x} < 1 if μ ≠ return-pmf x for μ
  proof (rule ccontr)
    assume ¬ emeasure (measure-pmf μ) {x} < 1
    then have emeasure (measure-pmf μ) {x} = 1
      using measure-pmf.emeasure-ge-1-iff by force
    with that show False
      by (simp add: measure-pmf-eq-1-iff)
  qed
  let ?S =
    {map-pmf (λ (X, l). (l, ([X := 0]u))) μ | μ l u g.
        x = (l, u) ∧ (l, g, μ) ∈ trans-of A}
  have (λ μ. measure-pmf μ {x}) ` ?K x
    ⊆ {0, 1} ∪ (λ μ. measure-pmf μ {x}) ` ?S
    by (force elim!: K.cases)
  moreover have finite ?S
  proof −
    have ?S ⊆ (λ (l, g, μ). map-pmf (λ (X, l). (l, (clock-set-set X 0(snd x)))) μ) ` trans-of A
      by force
    also from finite(3) have finite ... ..
    finally show ?thesis .
  qed
  ultimately have finite ((λ μ. measure-pmf μ {x}) ` ?K x)
    by (auto intro: finite-subset)
  then show ?thesis
    using that by (auto intro: ∗ finite-imp-Sup-less)
qed
{ fix s :: 's × ('c ⇒ real) and cfg :: ('s × ('c ⇒ real) set) cfg
  assume unbounded: ∀ c ∈ 𝒳. snd s c > k c and cfg ∈ R-G.cfg-on (abss s) abss s ∈ 𝒮
  then have repcs s cfg ∈ valid-cfg
    by (auto intro: R-G.valid-cfgI)
  then have cfg-on: repcs s cfg ∈ MDP.cfg-on s
    by (auto dest: MDP.valid-cfgD)
  from ‹cfg ∈ -› have action cfg ∈ 𝒦 (abss s)
    by (rule R-G.cfg-onD-action)
  have rept s (action cfg) ≠ return-pmf s
  proof (rule ccontr, simp)
    assume rept s (action cfg) = return-pmf s
    then have action cfg = return-pmf (abss s)
      using abst-rept-id[OF ‹action cfg ∈ -›]
      by (simp add: abst-def)
    moreover have (fst s, snd s) ∈ S
      using ‹- ∈ 𝒮› by (auto dest: 𝒮-abss-S)
    moreover have abss s = (fst s, [snd s]ℛ)
      by (metis abss-S calculation(2) prod.collapse)
    ultimately show False
      using ‹rept s - = -› unbounded unfolding rept-def by (cases s) (auto dest: cval-add-non-id)
  qed
  moreover have rept s (action cfg) ∈ K s
  proof −
    have action (repcs s cfg) ∈ K s
      using cfg-on by blast
    then show ?thesis
      by (simp add: repcs-def)
  qed
  ultimately have rept s (action cfg) ∈ ?K s
    by blast
  then have measure-pmf (rept s (action cfg)) {s} ≤ ?r s
    by (auto intro: Sup-upper)
```

```
  moreover have rept s (action cfg) = action (repcs s cfg)
    by (simp add: repcs-def)
  ultimately have measure-pmf (action (repcs s cfg)) {s} ≤ ?r s
    by auto
  note this ‹rept s (action cfg) ∈ ?K s›
}
note ∗ = this
let ?S = {cfg. ∃ cfg′ s. cfg′ ∈ R-G.valid-cfg ∧ cfg = repcs s cfg′ ∧ abss s = state cfg′}
have start: repcs st cfg ∈ ?S
  using ‹cfg ∈ R-G.valid-cfg› assms unfolding R-G-cfg-on-div-def
  by clarsimp (inst-existentials cfg fst st snd st, auto)
have step: y ∈ ?S if y ∈ K-cfg x x ∈ ?S for x y
  using that apply safe
  subgoal for cfg′ l u
    apply (inst-existentials absc y state y)
    subgoal
      by blast
    subgoal
      by (metis
          K-cfg-valid-cfgD R-G.valid-cfgD R-G.valid-cfg-state-in-S absc-repcs-id cont-absc-1
          cont-repcs1 repcs-valid
          )
    subgoal
      by (simp add: state-absc)
    done
  done
have ∗∗: x ∈ ?S if (repcs st cfg, x) ∈ MDP.MC.acc for x
proof −
  from MDP.MC.acc-relfunD[OF that] obtain n where ((λ a b. b ∈ K-cfg a) ^^ n) (repcs st cfg) x .
  then show ?thesis
  proof (induction n arbitrary: x)
    case 0
    with start show ?case
      by simp
  next
    case (Suc n)
    from this(2)[simplified] show ?case
      by (elim relcomppE step Suc.IH)
  qed
qed
have ∗∗∗: almost-everywhere (MDP.MC.T (repcs st cfg)) (alw (HLD ?S))
  by (rule AE-mp[OF MDP.MC.AE-T-reachable]) (fastforce dest: ∗∗ simp: HLD-iff elim: alw-mono)

have (AE ω in ?M. ?P ω) ⟷
  (AE ω in ?M. ∀ s :: ('s × ('c ⇒ real)).
    (∀ c ∈ 𝒳. snd s c > k c) ∧ s ∈ state ‘ (MDP.MC.acc “ {repcs st cfg}) ⟶
    ¬ (ev (alw (λ xs. shd xs = s))) (smap state ω)) (is ?L ⟷ ?R)
proof
  assume ?L
  then show ?R
    by eventually-elim auto
next
  assume ?R
  with MDP.MC.AE-T-reachable[of repcs st cfg] show ?L
  proof (eventually-elim, intro allI impI notI, goal-cases)
    case (1 ω s)
    from this(1,2,5,6) show ?case
      by (intro alw-HLD-smap alw-disjoint-ccontr[where
            S = state ‘ MDP.MC.acc “ {repcs st cfg} and R = {s} and ω = smap state ω
            ]; simp add: HLD-iff comp-def; blast)
  qed
```

```
  qed

  also have ... ⟷
      (∀ s :: ('s × ('c ⇒ real)).
        (∀ c ∈ 𝒳. snd s c > k c) ∧ s ∈ state ` (MDP.MC.acc `` {repcs st cfg}) ⟶
        (AE ω in ?M. ¬ (ev (alw (λ xs. shd xs = s))) (smap state ω)))
    using MDP.MC.countable-reachable[of repcs st cfg]
    by − (rule AE-all-imp-countable,
        auto intro: countable-subset[where B = state ` MDP.MC.acc `` {repcs st cfg}])
  also show ?thesis
    unfolding calculation
    apply clarsimp
    subgoal for l u x
      apply (rule MDP.non-loop-tail'[simplified, of state x ?S ?r (state x)])
      subgoal
        apply safe
        subgoal premises prems for cfg cfg' l' u'
        proof −
          from prems have state x = (l', u')
            by (metis MDP.cfg-onD-state state-repcs)
          with ‹- = state x› have [simp]: l = l' u = u'
            by auto
          show ?thesis
            unfolding ‹state x = -› using prems(1,3−) by (auto simp: R-G.valid-cfg-def intro: ∗)
        qed
        done
      subgoal
        apply (drule ∗∗)
        apply clarsimp
        apply (rule lt-1)
        apply (rule ∗)
        apply (auto dest: R-G.valid-cfg-state-in-S R-G.valid-cfgD)
        done
       apply (rule MDP.valid-cfgD[OF ‹repcs st cfg ∈ valid-cfg›]; fail)
      using ∗∗∗ unfolding alw-holds-pred-stream-iff[symmetric] HLD-def .
    done
qed

end
```

7.4 Main Result

```
context Probabilistic-Timed-Automaton-Regions-Reachability
begin

lemma R-G-cfg-on-valid:
  cfg ∈ R-G.valid-cfg if cfg ∈ R-G-cfg-on-div s'
  using that unfolding R-G-cfg-on-div-def R-G.valid-cfg-def by auto

lemma cfg-on-valid:
  cfg ∈ valid-cfg if cfg ∈ cfg-on-div s
  using that unfolding cfg-on-div-def MDP.valid-cfg-def by auto

abbreviation path-measure P cfg ≡ emeasure (MDP.T cfg) {x∈space MDP.St. P x}
abbreviation R-G-path-measure P cfg ≡ emeasure (R-G.T cfg) {x∈space R-G.St. P x}
abbreviation progressive st ≡ cfg-on-div st ∩ {cfg. alternating cfg}
abbreviation R-G-progressive st ≡ R-G-cfg-on-div st ∩ {cfg. alternating cfg}
```

Summary of our results on divergent configurations:

```
lemma absc-valid-cfg-eq:
  absc ` progressive s = R-G-progressive s'
```

apply *safe*
subgoal
unfolding *s′-def* **by** (*rule cfg-on-div-absc*) *auto*
subgoal
by (*simp add*: *AE-alw-ev-same-loc-iff cfg-on-valid*)
subgoal for *cfg*
unfolding *s′-def*
by (*frule cfg-on-div-repcs-strong*)
(*auto 4 4*
simp: *s′-def R-G-cfg-on-div-def AE-alw-ev-same-loc-iff′*[*symmetric*]
intro: *R-G-cfg-on-valid absc-repcs-id*[*symmetric*]
)
done

Main theorem:

theorem *Min-Max-reachability*:
notes *in-space-UNIV*[*measurable*] **and** [*iff*] = *pred-stream-iff*
shows
$(\bigsqcup cfg\in progressive\ s.\ \ path\text{-}measure\ \ (\lambda\ x.\ (holds\ \varphi\ \ suntil\ holds\ \psi)\ \ (s\ \ \#\#\ x))\ cfg)$
$= (\bigsqcup cfg\in R\text{-}G\text{-}progressive\ s'.\ R\text{-}G\text{-}path\text{-}measure\ (\lambda\ x.\ (holds\ \varphi'\ suntil\ holds\ \psi')\ (s'\ \#\#\ x))\ cfg)$
$\wedge\ (\bigsqcap cfg\in progressive\ s.\ \ path\text{-}measure\ \ (\lambda\ x.\ (holds\ \varphi\ \ suntil\ holds\ \psi)\ \ (s\ \ \#\#\ x))\ cfg)$
$= (\bigsqcap cfg\in R\text{-}G\text{-}progressive\ s'.\ R\text{-}G\text{-}path\text{-}measure\ (\lambda\ x.\ (holds\ \varphi'\ suntil\ holds\ \psi')\ (s'\ \#\#\ x))\ cfg)$
proof (*rule SUP-eq-and-INF-eq*; *rule bexI*[*rotated*]; *erule IntE*)
fix *cfg* **assume** *cfg-div*: *cfg* ∈ *R-G-cfg-on-div s′* **and** *cfg* ∈ *Collect alternating*
then have *alternating cfg*
by *auto*
let *?cfg′* = *repcs s cfg*
from ‹*alternating cfg*› *cfg-div* **have** *alternating ?cfg′*
by (*simp add*: *R-G-cfg-on-div-def s′-def AE-alw-ev-same-loc-iff′*[*of - s*])
with *cfg-div* ‹*alternating cfg*› **show** *?cfg′* ∈ *cfg-on-div s* ∩ *Collect alternating*
by (*auto intro*: *cfg-on-div-repcs-strong simp*: *s′-def*)
show $emeasure\ (R\text{-}G.T\ cfg)\ \ \{x \in space\ R\text{-}G.St.\ (holds\ \varphi'\ suntil\ holds\ \psi')\ (s'\ \#\#\ x)\}$
$= emeasure\ (MDP.T\ ?cfg')\ \{x \in space\ MDP.St.\ (holds\ \varphi\ suntil\ holds\ \psi)\ \ (s\ \#\#\ x)\}$
(**is** *?a* = *?b*)
proof −
from *cfg-div* **have** *cfg* ∈ *R-G.valid-cfg*
by (*rule R-G-cfg-on-valid*)
from *cfg-div* **have** *cfg* ∈ *R-G.cfg-on s′*
unfolding *R-G-cfg-on-div-def* **by** *auto*
then have *state cfg* = *s′*
by *auto*
have *?a* = *?b*
apply (*rule*
path-measure-eq-repcs′′-new[
of s cfg φ ψ, *folded φ′-def ψ′-def*, *unfolded* ‹- = *s′*› *state-repcs*
]
)
subgoal
unfolding *s′-def* **..**
subgoal
by *fact*
subgoal
using ‹*?cfg′* ∈ *cfg-on-div s* ∩ -› **by** (*blast intro*: *cfg-on-valid*)
subgoal premises *prems* **for** *xs*
using *prems s* **by** (*intro φ-stream*)
subgoal premises *prems*
using *prems s* **by** (*intro ψ-stream*)
done
then show *?thesis*
by *simp*
qed

next
fix *cfg* **assume** *cfg-div*: *cfg* ∈ *cfg-on-div s* **and** *cfg* ∈ *Collect alternating*
with *absc-valid-cfg-eq* **show** *absc cfg* ∈ *R-G-cfg-on-div s′* ∩ *Collect alternating*
by *auto*
show *emeasure* (*MDP.T cfg*) {*x* ∈ *space MDP.St*. (*holds* φ *suntil holds* ψ) (*s* ## *x*)}
= *emeasure* (*R-G.T* (*absc cfg*)) {*x* ∈ *space R-G.St*. (*holds* φ′ *suntil holds* ψ′) (*s′* ## *x*)}
(**is** *?a* = *?b*)
proof −
have *absc cfg* ∈ *R-G.valid-cfg*
using *R-G-cfg-on-valid* ‹*absc cfg* ∈ *R-G-cfg-on-div s′* ∩ -› **by** *blast*
from *cfg-div* **have** *cfg* ∈ *valid-cfg*
by (*simp add*: *cfg-on-valid*)
with ‹*absc cfg* ∈ *R-G.valid-cfg*› **have** *?b* = *?a*
by (*intro MDP.alw-S R-G.alw-S path-measure-eq-absc1-new*
[**where** *P* = *pred-stream* (λ*s*. *s* ∈ 𝒮) **and** *Q* = *pred-stream* (λ*s*. *s* ∈ *S*)]
)
(*auto simp*: *S-abss-*𝒮 *intro*: 𝒮*-abss-S intro*!: *suntil-abss suntil-reps*, *measurable*)
then show *?a* = *?b*
by *simp*
qed
qed

end

end

References

[1] M. Z. Kwiatkowska, G. Norman, R. Segala, and J. Sproston. Automatic verification of real-time systems with discrete probability distributions. *Th. Comp. Sci.*, 282(1).

[2] S. Wimmer and J. Hölzl. MDP + TA = PTA: Probabilistic timed automata, formalized. In J. Avigad and A. Mahboubi, editors, *ITP 2018, Proceedings*, Lecture Notes in Computer Science. Springer, 2018.