Probabilistic Timed Automata

Simon Wimmer and Johannes Höltz

April 20, 2020

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
  2.1 Justifying Timed Until vs until 10
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 12
  3.3 Syntactic Definition 12
4 Constructing the Corresponding Finite MDP on Regions 13
  4.1 Syntactic Definition 13
  4.2 Many Closure Properties 13
  4.3 The Region Graph is a Finite MDP 15
5 Relating the MDPs 16
  5.1 Translating From K to 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

7.1 Definitions .................................................. 49
7.2 Easier Result on All Configurations ....................... 50
7.3 Divergent Adversaries ...................................... 51
7.4 Main Result .................................................. 81
1 Bisimulation on a Relation

definition rel-set-strong :: ('a ⇒ 'b ⇒ bool) ⇒ 'a set ⇒ 'b set ⇒ bool
where rel-set-strong R A B ⇔ (∀x y. R x y → (x ∈ A ⇔ y ∈ B))

lemma T-eq-rel-half [consumes 4, case_names prob sets cont]:
fixes R :: 's ⇒ 't ⇒ bool and f :: 's ⇒ 't and S :: 's set
assumes R-def: ∀s t. R s t ↔ (s ∈ S ∧ f s = t)
assumes A[measurable]: A ∈ sets (stream-space (count-space UNIV))
and B[measurable]: B ∈ sets (stream-space (count-space UNIV))
shows MC-syntax.T K x A = MC-syntax.T L y B

proof –
interpret K: MC-syntax K by unfold-locales
interpret L: MC-syntax L by unfold-locales
have x ∈ S using (R x y) by (auto simp: R-def)

define g where g t = (SOME s. R s t) for t
have measurable-g: g ∈ count-space UNIV →ₚ count-space UNIV by auto
have g: R i j ⇒ R (g j) j for i j
unfolding g-def by (rule someI)

have K-subset: x ∈ S ⇒ K x ⊆ S for x
using KL[THEN rel-funD, of x f x, THEN rel-pmf-imp-rel-set] by (auto simp: rel-set-def R-def)

have in-S: AE ω in K.T x. ω ∈ streams S
using K.AE-T-enabled

proof eventually-clim
  case (elim ω) with (x ∈ S) show ?case
  apply (coinduction arbitrary: x ω)
  subgoal for x ω using K-subset by (cases ω) (auto simp: K.enabled-Stream)
done

qed

have L-eq: L y = map-pmf f (K x) if xy: R x y for x y
proof –
  have rel-pmf (∀x. y. x = y) (map-pmf f (K x)) (L y)
  using KL[THEN rel-funD, OF xy] by (auto intro: pmf.rel-mono-strong simp: R-def pmf.rel-map)
  then show ?thesis unfolding pmf.rel-eq by simp

qed

let ?D = λx. distr (K.T x) K.S (smap f)

have prob-space-D: ?D x ∈ space (prob-algebra K.S) for x
  by (auto simp: space-prob-algebra K.T.prob-space-distr)

have D-eq-D: ?D x = ?D x' if R x y R x' y for x x'
you write some content here
apply (simp add: space-stream-space streams-sets AE-distr-iff)
using K.AE-T-reachable[of x'] unfolding abw-HLD-iff-streams
proof eventually-clim
  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')
  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')
  have rel-pmf (R OO R\(^{-1}\)) (K x) (K x')
    using KL THEN rel-funD, OF Cons(4)] KL THEN rel-funD, OF Cons(5)]
  unfolding pmf.rel-comp pmf.rel-flip by auto
  then obtain p :: (\langle s × s'\rangle) pmf where p: \langle a b, (a, b) ∈ p \rightarrow (R OO R\(^{-1}\)) 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 \langle \#\#\rangle y \#\# smap f \#\# sstart (f ` A) (z \#\# zs) = (if f y = z then smap f \#\# sstart (f ` A) zs else
    \langle \#\#\rangle)
    by (auto)
  have \langle \#\#\rangle: ?D x (sstart (f ` A) (z \#\# zs)) = (f + 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 \langle \#\#\rangle 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 $KL[\ THEN \ rel-$fun$D]$)

then have $*$: $y' \in L \ y \Longrightarrow \exists x' \in K \ x. \ R \ x' \ y'$ for $y'$

by (auto dest!: rel-$pmf$-$imp$-$rel$-$set$ simp: rel-$set$-$def$)

have $**$: $y' \in L \ y \Longrightarrow R \ (g \ y') \ y'$ for $y'$

using $*\ [of \ y']$ unfolding $g$-def by (auto intro: some$I$)

have $D$-SCons-eq-$D$-$D$: distr $(K \ T \ i)$ $K.S \ (\lambda x. \ z \ #\ # \ smap \ f \ x) = \ distr \ (?D \ i)$ $K.S \ (\lambda x. \ z \ #\ # \ x)$ for $i \ z$

by (subst distr-$distr$) (auto simp: comp-$def$)

have $D$-eq-$D$-$gi$: $?D \ i = ?D \ (g \ (f \ i))$ if $i: i \in K \ x$ for $i$

proof —

obtain $j$ where $j \in L \ y \ R \ i \ j \ f \ i = j$

using $Kx$-$Ly$ $i$ by (force dest!: rel-$pmf$-$imp$-$rel$-$set$ simp: rel-$set$-$def$ $R$-$dest$)

then show $?thesis$

by (auto intro!: $D$-eq-$D$[OF :$(R \ i \ j)$] $g$)

qed

have $***$: $?D \ x = measure-$pmf$ $(L \ y) \gg (\lambda y. \ distr \ (?D \ (g \ y)) \ K.S \ ((#\ #) \ y))$

apply (subst $K.T$-eq-$bind$)

apply (subst distr-$bind$[of -$K.S$])

apply (rule measurable-$distr$2[of -$K.S$])

apply (simp-all add: Pi-$iff$)

apply (simp add: distr-$distr$ comp-$def$ $L$-eq-$D$[OF cont] map-$pmf$-$rep$-$eq$)

apply (subst bind-$distr$[where $K=K.S$])

apply measurable []

apply (rule measurable-$distr$2[of -$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 ex$I$[of -$A$]. 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 $R \ s \ t \iff (s \in \ streams \ S \ \land \ smap \ f \ s = t)$ for $s \ t$

proof safe

show stream-all2 $R \ s \ t \Longrightarrow s \in \ streams \ S$

apply (coinduction arbitrary: $s \ t$)

subgoal for $s \ t$ by (cases $s$; cases $t$) (auto simp: R-$def$)

done

show stream-all2 $R \ s \ t \Longrightarrow smap \ f \ s = t$

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 $\omega \in \ streams \ S \Longrightarrow \omega \in A \longleftrightarrow smap \ f \omega \in B$ for $\omega$

using $AB$ by (auto simp: rel-set-strong-$def$)

with in-$S$ have $K.T \ x \ A = K.T \ x$ (smap $f = ^{-1} B \ \cap \ \ space \ (K.T \ x))$

by (auto intro!: emeasure-$eq$-$AE$ streams-$sets$)

also have $\ldots = (distr \ (K.T \ x) \ K.S \ (smap \ f)) \ B$

by (intro emeasure-$distr$[symmetric]) auto

also have $\ldots = (L.T \ y) \ B$ unfolding $L$-eq-$D$ ..
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 ≡ (∀ t ≥ 0. ∀ u ∈ R. u ⊕ t ∈ R)

definition succ R ≡
  if is-upper-right R then R else
  (THE R'. R' ≠ R ∧ R' ∈ Succ R ∧ (∀ u ∈ R. ∀ t ≥ 0. (u ⊕ t) ∉ R → (∃ t' ≤ t. (u ⊕ t') ∈ R' ∧ 0 ≤ t')))

lemma region-continuous:
  assumes valid-region X k I r
  defines R: R ≡ region X k I r
  assumes between: 0 ≤ t1 t1 ≤ t2
  assumes elem: u ∈ R u ⊕ t2 ∈ R
  shows u ⊕ t1 ∈ R

unfolding R

proof
  from:0 ≤ t1: (u ∈ R) show ∀ x ∈ X. 0 ≤ (u ⊕ t1) x by (auto simp: R cval-add-def)

  have intv-elem x (u ⊕ t1) (I x) if x ∈ X for x
  proof –
    from elem that have intv-elem x u (I x) intv-elem x (u ⊕ t2) (I x) by (auto simp: R)
    with between show ?thesis by (cases I x, auto simp: cval-add-def)
  qed

then show ∀ x ∈ X. intv-elem x (u ⊕ t1) (I x) by blast

let ?X0 = {x ∈ X. ∃ d. I x = Intv d}

show ?X0 = ?X0 ..

from elem have ∀ x ∈ ?X0. ∀ y ∈ ?X0. (x, y) ∈ r ↔ frac (u x) ≤ frac (u y) by (auto simp: R)

moreover
  { fix x y c d assume A: x ∈ X y ∈ 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) ≤ frac (u y) ↔ frac ((u ⊕ t1) x) ≤ frac ((u ⊕ t1) y)
    by (auto simp: cval-add-def)
  }

ultimately show
\[ \forall x \in ?X_0. \forall y \in ?X_0. (x, y) \in r \leftrightarrow \frac{u + t_1}{x} \leq \frac{(u + t_1) y}{x} \]
by \text{(auto simp: cval-add-def)}
qed

\textbf{lemma} upper-right-eq:
assumes finite X valid-region X k I r
shows \((\forall x \in X. \text{isGreater} (I x)) \leftrightarrow \text{is-upper-right} (\text{region} X I r)\)
using assms
proof (safe, goal-cases)
case (1 t u)
then show "\text{case} (1 t u)" by (standard, force simp: cval-add-def)+
next
case (2 x)
from region-not-empty[OF assms] obtain u where u: \(u \in \text{region} X I r\) ..
moreover have \(1 :: \text{real} \geq 0\) by auto
ultimately have \((u + 1) \in \text{region} X I r\) using 2 by auto
with \(x \in X\) obtain intv-elem x u \((I x)\) intv-elem x \((u + 1)\) \((I x)\) by auto
then show "\text{case} (cases I x, auto simp: cval-add-def)"
qed

\textbf{lemma} bounded-region:
assumes finite X valid-region X k I r
defines \(R : R \equiv \text{region} X I r\)
assumes \(\neg \text{is-upper-right} R u \in R\)
shows \(u + 1 \notin R\)
proof -
from upper-right-eq[OF assms(1,2)] assms(4) obtain x where x: \(x \in X\) \(\neg \text{isGreater} (I x)\)
by (auto simp: R)
with assms have intv-elem x u \((I x)\) by auto
with \(x(2)\) have \(\neg \text{intv-elem} x (u + 1) (I x)\) by (cases I x, auto simp: cval-add-def)
with \(x(1)\) assms show \(\text{thesis}\) by auto
qed

\textbf{context} AlphaClosure

\textbf{begin}

\textbf{no-notation} Regions-Beta.part \([\ldots] [61,61] 61\)

\textbf{lemma} succ-ex:
assumes \(R \in \mathcal{R}\)
shows succ \(R \in \mathcal{R}\) (is \(?G1\)) \text{ and } succ \(R \in \text{Succ} \mathcal{R} R\) (is \(?G2\))
and \(\forall \:\exists \:\forall t \geq 0. (u + t) \notin R \rightarrow (\exists t' \leq t. (u + t') \in \text{succ} \mathcal{R} R \land 0 \leq t')\) (is \(?G3\))
proof -
from \(R \in \mathcal{R}\) obtain I r where R: \(R = \text{region} X I r\) valid-region X k I r
unfolding \(\text{R-def}\) by auto
from region-not-empty[OF finite] R obtain u where u: \(u \in R\)
by blast
let \(?Z = \{x \in X. \exists c. I x = \text{Const} c\}\)
let \(?\text{succ} = \lambda R'. R' \neq R \land R' \in \text{Succ} \mathcal{R} R\)
\land \(\forall u \in R. \forall t \geq 0. (u + t) \notin R \rightarrow (\exists t' \leq t. (u + t') \in R' \land 0 \leq t')\)
consider (upper-right) \(\forall x \in X. \text{isGreater} (I x)\) \((\text{intv}) \exists x \in X. \exists d. I x = \text{Intv} d \land ?Z = \{\}\)
| (const) \(?Z \neq \{\}\)
apply (cases \(\forall x \in X. \text{isGreater} (I x)\))
apply fast
apply (cases \(?Z = \{\}\))
apply safe
apply (rename-tac x)
apply (case-tac I x)
by auto
then have ?G1 ∧ ?G2 ∧ ?G3
proof cases
  case const
  with upper-right-eq[OF finite R(2)] have ∼ 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':
  ∀ u ∈ R. ∀ t>0, ∃ t'≤t. (u ⊕ t') ∈ R' ∧ t' ≥ 0 R' ∈ R ∀ u ∈ R', ∀ t≥0. (u ⊕ t) ∉ R
  unfolding R-def by auto
with region-not-empty[OF finite] obtain u' where u' ∈ R' unfolding R-def by blast
with R'(3) have neq: R' ≠ R by (fastforce simp: cval-add-def)
  obtain t: real where t > 0 by (auto intro: that[of 1])
  with R'(1,2) u ∈ R obtain t where t ≥ 0 u ⊕ t ∈ R' by auto
  with (R ∈ R) (R' ∈ R) u ∈ R have R' ∈ Succ R by (intro SuccI)
  moreover have (∀ u ∈ R. ∀ t ≥ 0. (u ⊕ t) ∉ R → (∃ t' ≤ t. (u ⊕ t') ∈ R' ∧ 0 ≤ t'))
  using R'(1) unfolding cval-add-def
  apply clarsimp
  subgoal for u t
    by (cases t = 0) auto
  done
ultimately have ∗: ?suc R using neq by auto
have suc R = R' unfolding suc-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" ∈ R. R" ≠ R t' ≥ 0 R" = [u' ⊕ t] R u' ∈ 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: R-def)
  from R" have u" ⊕ t" ∉ R using assms region-unique-spec by blast
  with ∗ t' ≥ 0 u' ∈ R obtain t" where t": t" ≤ t' u" ⊕ t" ∈ R' t" ≥ 0 by auto
  from this(2) neq have u" ⊕ t" ∉ R using R'(2) assms region-unique-spec by auto
  with t" prems u' ∈ R obtain t" where t":
  t" ≤ t' u" ⊕ t" ∈ R' t" ≥ 0 by auto
  with region-continuous[OF R" (2) - - t"[2][unfolded R"2[1]], of t" = t'' t' = t"
  t" R" regions-closed-1-spec[OF :R ∈ R: R"[5,3]]
  have u" ⊕ t" ∈ 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
next
  case inv
  then have ∗: ∀ x ∈ X. ∼ Regions.isConst (I x) by auto
  let ?X_0 = {x ∈ X. isnInv (I x)}
  let ?M = {x ∈ ?X_0. ∀ y ∈ ?X_0. (x, y) ∈ r −→ (y, x) ∈ r}
  from inv obtain x c where x: x ∈ X ∼ isGreater (I x) and c: I x = Inv c by auto
  with (x ∈ X) have ?X_0 ≠ {} by auto
  have ?X_0 = {x ∈ X. ∃ d. I x = Inv 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 ≠ {}] - this finite
  obtain x' where x': x' ∈ ?X_0 ∀ y ∈ ?X_0. x' ≠ y −→ (y, x') ∈ r by fastforce
  from this(2) have ∀ y ∈ ?X_0. (x', y) ∈ r −→ (y, x') ∈ r by auto
  with x'(1) have ∗∗: ?M ≠ {} by fastforce
  with upper-right-eq[OF finite R(2)] have ∼ 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':
\[(\forall u \in R. \forall t \geq 0. (u \oplus t) \notin R \rightarrow (\exists t' \leq t. (u \oplus t') \in R' \land 0 \leq t')) \] 
\[R' \in \mathcal{R}
\]
\[(\forall u \in R'. \forall t \geq 0. (u \oplus t) \notin R) \]

**unfolding** \[\mathcal{R}\text{-}def\] by auto

**with** region-not-empty \([\text{OF finite}]\) obtain \(u'\) where \(u' \in R'\) unfolding \(\mathcal{R}\)-def by blast

**with** \(R'(3)\) have neq: \(R' \neq R\) by \([\text{fastforce simp: cval-add-def}]\)

**from** bounded-region \([\text{OF finite } R(2), \text{ folded } R(1), \text{ OF } (\neg \text{- upper-right } R)\) 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** \(t' \in \mathcal{R}\) \(\exists R' \in \mathcal{R}\) u \in \(\exists\) \(R' \in\) Succ \(\mathcal{R}\) \(R\) by \([\text{intro Succ3}]\)

**with** \(R'(1)\) neq have \(*: \{\text{succ } R'\text{ by auto}\}

**have** succ \(\mathcal{R} R = R'\) unfolding succ-def

**proof** \([\text{simp add: } (\neg \text{- upper-right } R), \text{ intro the-equality, rule *}, \text{ goal-cases}]\)

**case** prems: \((1 R'')\)

**from** prems obtain \(t' u'\) where \(R''\):

\[R'' \in \mathcal{R} R'' \neq R t' \geq 0 R'' = [u' \oplus t']_R u' \in R\]

using \(R'(1)\) by fastforce

**from** this\((1)\) obtain \(I r\) where \(R''2:\)

\[R'' = \text{region } X I r \text{ valid-region } X k I r\]

by \([\text{auto simp: } \mathcal{R}\text{-}def]\)

**from** \(R''\) have \(u' \oplus t' \notin R\) using assms region-unique-spec by blast

**with** \(*: t' \geq 0\) \(u' \in \mathcal{R}\) 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 \([\text{OF } R''2(2) \rightarrow t''''(2)[\text{unfolded } R''2(1)], \text{ of } t'''' = t'''' t' - t'''']\)

**have** \(u' \oplus t'''' \in R''\) by \([\text{auto simp: cval-add-def } R''2(2)]\)

**with** \(t''(2)\) show \(*: \text{case using } R''(1) R''(2) \text{ region-unique-spec by blast}\)

**qed**

**with** \(R'\) * show \(*: \text{thesis by auto}\)

next

**case** upper-right

**with** upper-right-eq \([\text{OF finite } R(2)]\) have succ \(\mathcal{R} R = R\) by \([\text{auto simp: } \mathcal{R} \text{-def}]\)

**with** \(\{R \in \mathcal{R}\}\) u show \(*: \text{thesis by } \text{fastforce simp: cval-add-def intro: Succ3}\)

**qed**

**then** show \(*: G1 G2 G3 by auto\)

**qed**

**lemma** region-set\(^\prime\)-closed:

**fixes** \(d::\text{nat}\)

**assumes** \(R \in \mathcal{R} d \geq 0 \forall x \in \text{set } r. \ d \leq k x. \text{set } r \subseteq X\)

**shows** region-set\(^\prime\) \(R d R\) \(d \in \mathcal{R}\)

**proof** —

**from** region-not-empty \([\text{OF finite}]\) assms\((1)\) obtain \(u\) where \(u \in \mathcal{R}\) using \(\mathcal{R}\text{-def}\) by blast

**from** region-set\(^\prime\)-id \([\text{OF - finite, of - k, folded } \mathcal{R}\text{-def}]\) assms this show \(*: \text{thesis by fastforce}\)

**qed**

**lemma** clock-set-cong\([\text{simp}]\):

**assumes** \(\forall c \in \text{set } r. \ u = c d\)

**shows** \([r \rightarrow d] u = u\)

**proof** standard

**fix** \(c\)

**from** assms show \(([r \rightarrow d] u) c = u c\) by \([\text{cases } c \in \text{set } r; \text{ auto}]\)

**qed**

**lemma** region-reset-not-Succ:

**notes** regions-closed\(^\prime\)-spec\([\text{intro}]\)
assumes $R \in \mathcal{R}$ set $r \subseteq X$
shows region-set' $R$ $r \ 0 = R \setminus region-set' R$ $r \ 0 \notin Succ \mathcal{R}$ $R$ (is $\mathcal{R} = R \setminus -$)
proof
from assms finite obtain $u$ where $u \in R$ by (meson Succ.cases succ-ex(2))
with $R \in \mathcal{R}$ have $u \in V \ [u]_R = R$ by (auto simp: region-unique-spec dest: region-V)
with region-set'-id[OF $\mathcal{R}$ $r 0$] have $R = [r \rightarrow 0]u_R$
by (force simp: $\mathcal{R}$-def)
next
proof (cases $\forall x \in \operatorname{set} r$. $u x = 0$
  case True
  then have $[r \rightarrow 0]u = u$ by simp
  with (force simp: $\mathcal{R}$-def)
  then show ?thesis
next
the true case obtain $x$ where $x \in \operatorname{set} r \ x \neq 0$ by auto
  assume $?R \in Succ \mathcal{R} \ R$
with $\mathcal{R}$ obtain $t$ where
  $t \geq 0 \ [u \oplus t]_R = ?R \ ?R \in \mathcal{R}$
by (meson Succ.cases set-of-regions-spec)
with $u \in R$ assms(1) have $u \oplus t \in ?R$ by blast
moreover from (force simp: region-set'-def)
moreover from $x \ (t \geq 0) \ (u \in V)$ assms have $(u \oplus t) \ x \ > \ 0$ by (force simp: eval-add-def V-def)
moreover from $x$ have $(r \rightarrow 0]u \ x \ = \ 0$ by auto
ultimately have False using (force simp: region-set'-def)
then show ?thesis by auto
qed
qed
end

2.1 Justifying Timed Until vs $\text{suntill}$

lemma guard-continuous:
assumes $u \vdash g \ u \oplus t \vdash g \ 0 \leq (t'::'t::time) \ t' \leq t$
shows $u \oplus t' \vdash g$
using assms
by (induction g;
  auto 4 3
    simp: eval-add-def order-le-less-subst2 order-subst2 add-increasing2
    intro: less-le-trans)

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
definition
valid-abstraction A X k ≡ ((∀ (x, m) ∈ clkp-set A. m ≤ k x ∧ x ∈ X ∧ m ∈ N) ∧ collect-clkvt A ⊆ X ∧ finite X)
definition 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
3.2 Operational Semantics as an MDP

abbreviation (input) clock-set-set :: 'c set ⇒ 't::time ⇒ ('c,'t) cval ⇒ ('c,'t) cval
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
where
[X:=t]R ≡ region-set' R (SOME r. set r = X) t

no-notation zone-set (→ 0 [71] 71)

abbreviation zone-set-set :: ('c, 't::time) zone ⇒ 'c set ⇒ ('c, 't) zone
where
Z_X → 0 ≡ zone-set Z (SOME r. set r = X)

abbreviation (input) ccval [100] where ccval cc ≡ {v. v ⊢ cc}

locale Probabilistic-Timed-Automaton =
  fixes A :: ('c, 't::time, 's) pta
  assumes admissible-targets:
(1, g, µ) ∈ trans-of A ⇒ (X, l') ∈ µ ⇒ g[X → 0] ⊆ inv-of A l'
(l, g, µ) ∈ trans-of A ⇒ (X, l') ∈ µ ⇒ X ⊆ clocks A
— Not necessarily what we want to have
begin

3.3 Syntactic Definition

definition L = locations A

definition X = clocks A

definition S ≡ {(l, u) . l ∈ L ∧ (∀ x ∈ X. u x ≥ 0) ∧ u ⊢ 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 ∈ S ⇒ st = (l, u) ⇒ t ≥ 0 ⇒ u ⊕ t ⊢ inv-of A l ⇒ return-pmf (l, u ⊕ t) ∈ K st |
— Discrete transitions action:
st ∈ S ⇒ st = (l, u) ⇒ (l, g, µ) ∈ trans-of A ⇒ u ⊢ g ⇒ map-pmf (λ (X, l). (l, (X := 0|u))) µ ∈ K st |
— Self loops – Note that this does not assume st ∈ S loop:
return-pmf st ∈ K st

declare K.intros[intro]

sublocale MDP: Markov-Decision-Process K by (standard, auto)
4 Constructing the Corresponding Finite MDP on Regions

locale Probabilistic-Timed-Automaton-Regions =  
Probabilistic-Timed-Automaton A + Regions X

for A :: ('c, t, 's) pta +
— The following are necessary to obtain a finite MDP
  assumes finite; finite X finite L finite (trans-of A)
  assumes not-trivial; ∃ l ∈ L, ∃ u ∈ V. u ⊨ inv-of A l
  assumes valid; valid-abstraction A X k

begin

lemmas finite-R = finite-R[OF finite(1), of k, folded R-def]

4.1 Syntactic Definition

definition S = {((l, R) : l ∈ L ∧ R ∈ R ∧ R ⊆ {u. u ⊨ inv-of A l})}

lemma S-alt-def: S = {(l, u) : l ∈ L ∧ u ∈ V ∧ u ⊨ inv-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
K :: ('s * ('c, t) cval set) ⇒ ('s * ('c, t) cval set) pmf set for st :: ('s * ('c, t) cval set)
where
— Passage of time delay:
st ∈ S ⇒ st = (l, R) ⇒ R' ∈ Succ R ⇒ R' ⊆ {inv-of A l} ⇒ return-pmf (l, R') ∈ K st |
— Discrete transitions action:
st ∈ S ⇒ st = (l, R) ⇒ (l, g, μ) ∈ trans-of A ⇒ R ⊆ {g} ⇒ map-pmf (λ (X, l). (l, region-set R (SOME r. set r = X) 0)) μ ∈ K st |
— Self loops – Note that this does not assume st ∈ S loop:
return-pmf st ∈ K st

lemmas [intro] = K.intros

4.2 Many Closure Properties

lemma transition-def:
(A ⊢ l →^{g, μ, X} l') = (((l, g, μ) ∈ trans-of A ∧ (X, l') ∈ μ)
unfolding Edges-def edges-def trans-of-def by auto

lemma transitionI[intro]:
A ⊢ l →^{g, μ, X} l' if (l, g, μ) ∈ trans-of A (X, l') ∈ μ
using that unfolding transition-def ..

lemma transitionD[dest]:
(l, g, μ) ∈ trans-of A (X, l') ∈ μ if A ⊢ l →^{g, μ, X} l'
using that unfolding transition-def by auto

lemma bec-Edges:
(∃ x ∈ Edges A. P x) = (∃ l g μ X l' A ⊢ l →^{g, μ, X} l' ∧ P (l, g, μ, X, l'))
by fastforce

lemma L-trans[intro]:
assumes (l, g, μ) ∈ trans-of A (X, l') ∈ μ
shows l ∈ L l' ∈ L
using assms unfolding L-def locations-def by (auto simp: image-iff bex-Edges transition-def)

lemma transition-X:
\[ X \subseteq A \text{ if } A \vdash l \rightarrow g, \mu, X \]
using that unfolding X-def collect-clkvt-def clkp-set-def by auto

lemma admissible-targets-alt:
\[ A \vdash l \rightarrow g, \mu, X l' = \Rightarrow \{ |g| \} X \rightarrow 0 \subseteq \{ \text{inv-of } A \ l' \} \]
by (intro admissible-targets; blast)+

lemma V-reset-closed[intro]:
assumes \( u \in V \)
shows \( r \rightarrow (d::nat)u \in V \)
using assms unfolding V-def apply safe
subgoal for \( x \)
by (cases \( x \in \text{set } r \); auto)
done

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

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

lemma rep-R-ex[intro]:
assumes \( R \in R \)
shows \( \exists u. u \in R \) \( R \)
proof
- from assms region-not-empty[OF finite(1)] have \( \exists u. u \in R \) unfolding R-def by auto
then show \( \exists u. u \in R \) unfolding R-def by auto
qed

lemma V-nn-closed[intro]:
\( u \in V \) \( \Rightarrow \) \( t \geq 0 \) \( \Rightarrow \) \( u \oplus t \in V \)
unfolding V-def cvl-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]:
\( (l, u) \in S \) \( \Rightarrow \) \( u \in V \)
unfolding S-alt-def by auto

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

lemma S-V[intro]:

\((l, R) \in S \implies R \in \mathcal{R}\)

unfolding \texttt{S-def by auto}

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

4.3 The Region Graph is a Finite MDP

lemma \texttt{S-finite}:
    finite \texttt{S}
using finite finite-\texttt{R} unfolding \texttt{S-def by auto}

\begin{verbatim}
lemma \texttt{K-finite}:
    finite \((K, st)\)
proof -
  let \texttt{?B1} = \{(l', l, R). \texttt{st} \in S \land \texttt{st} = (l, R) \land R' \in \text{Succ } \texttt{R} R \land R' \subseteq \{\text{inv-of } A l'\}\}
  let \texttt{?S1} = \((\lambda(l', l, R). \text{\texttt{return-pmf } (l, R')) \circ \texttt{?B1}\)
  let \texttt{?S2} = \{(\lambda(X, l). (l, \text{region-set'} R (\text{SOME } r. \text{ set } r = X)) \emptyset)) \mu
      | R \mu. \exists l g. st \in S \land st = (l, R) \land l' \in \text{Succ } \texttt{R} R \land R' \subseteq \{\text{inv-of } A l'\}\}
  have \texttt{?B1} \subseteq \{(l', l, R). R' \in \texttt{R} \land (l, R) \in S \} unfolding \texttt{S-def by auto}
  with \texttt{\texttt{S-finite} finite-\texttt{R} have finite \texttt{?B1} by (rule finite-subset, auto)}
  moreover have \texttt{?S1} = \((\lambda(l', l, R). \text{\texttt{return-pmf } (l, R')) \circ \texttt{?B1}\) by (auto simp: image-def)
  ultimately have \((K, st)\) \texttt{finite} \texttt{R} \texttt{by auto}
  have \((\mu, \exists l g. (l, g, \mu) \in \texttt{PTA}.\text{\texttt{trans-of } A}) = ((\lambda (l, g, \mu), \mu \circ \texttt{PTA}.\text{\texttt{trans-of } A}) \texttt{by force})
  with \texttt{finite(3)} \texttt{finite-\texttt{R} have finite \{(l, \mu, r). \exists l g. R \in \texttt{R} \land (l, g, \mu) \in \texttt{trans-of } A \} by auto}
  moreover have \{(l, \mu, r) \in \texttt{S} \land st = (l, R) \land l g. st \in S \land st = (l, R) \land (l, g, \mu) \in \texttt{trans-of } A \land R \subseteq \{g\}\} \subseteq \ldots
  unfolding \texttt{S-def by fastforce}
  ultimately have \((K, st)\) \texttt{finite} \texttt{R} \texttt{by auto}
  have \texttt{K \texttt{st} = ?S1 \cup ?S2 \cup \{\text{return-pmf st}\} by (safe, cases \texttt{rule: K.cases, auto})}
  with \texttt{* * show \texttt{?thesis by auto}}
\end{verbatim}

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

\begin{verbatim}
lemma \texttt{S-not-empty}:
    \texttt{S} \neq \texttt{\{\}}
proof -
  from \texttt{not-trivial obtain l u where st: l \in L u \in V u \vdash inv-of A l by blast}
  then obtain \texttt{R where R: R \in R} u \in \texttt{R using \texttt{R-V by auto}}
\end{verbatim}

15
from valid have\[
\forall (x, m) \in \text{collect-clock-pairs} (\text{inv-of } A \ l). \ m \leq \text{real} (k \ x) \land x \in X \land m \in N\]
by (fastforce simp: clkp-set-def collect-clki-def)
from ccompatible \[OF\ this,\ folded \ R-def\] \[R \ st(\exists)\] have\[
R \subseteq \{\text{inv-of } A \ l\}\]
unfolding ccompatible-def ccollect-def by auto
with \[st(1)\ R(1)\] show \[\text{thesis}\ unfolding \ S-def\ by\ auto\]
qed

lemma \[K-S-closed\]:
assumes \[s \in S\]
shows \[(\bigcup D \in K \ s. \ \text{set-pmf} \ D) \subseteq S\]
proof (safe, cases rule: \[K\cases\], blast, goal-cases)
case \[(1 \ x a b l R)\]
then show \[?\text{case}\ unfolding \ S-def\ by\ (auto\ intro:\ \text{alpha-interp.success-ex}(1))\]
next
case \[(3 a b x)\]
with \[s \in S\] show \[?\text{case}\ by\ auto\]
next
case \[\text{prems:} (2 l' R' p l R g \mu)\]
then obtain \[X\ where \ast: (X, l') \in \text{set-pmf} \ \mu R' = \text{region-set'} \ R (\text{SOME } r. \ \text{set } r = X) \ 0\ by\ auto\]
show \[?\text{case}\ unfolding \ S-def\]
proof safe
from \[\ast(1)\] have \[(l, g, \mu, X, l') \in \text{edges} (l, g, \mu)\] unfolding \[\text{edges-def}\ by\ auto\]
with \[\text{prems}(6)\] have \[(l, g, \mu, X, l') \in \text{Edges-def}\ unfolding \ \text{Edges-def\ trans-of-def}\ by\ auto\]
then show \[l' \in L\ unfolding \ L-def\ locations-def\ by\ force\]
show \[u \vdash \text{inv-of } A l'\ if\ u \in R'\ for\ u\]
using \[\text{admissible-targets}(\text{OF}\ \text{prems}(6) \ast(1) \ \text{prems}(7))\ast(2)\ that\ by\ auto\]
from \[\text{admissible-targets}(2)(\text{OF}\ \text{prems}(6) \ast(1))\] have \[X \subseteq X'\ unfolding \ X-def\ by\ auto\]
with \[\text{finite}(1)\] have \[\text{finite} \ X\ by\ (blast\ intro: \text{finite-subset})\]
then obtain \[r\ where\ \text{set } r = X\ using \text{finite-list}\ by\ auto\]
then have \[\text{set } (\text{SOME } r. \ \text{set } r = X) = X\ by\ \text{(rule\ someI)}\]
with \[X \subseteq X'\] have \[\text{set } (\text{SOME } r. \ \text{set } r = X) \subseteq X'\ by\ \text{auto}\]
with \[\text{alpha-interp.region-set'-closed}(\text{OF } R \ 0 \ \text{SOME } r. \ \text{set } r = X) \ \text{prems}(4,5) \ast(2)\]
show \[R' \in R\ unfolding \ S-def\ X-def\ by\ auto\]
qed
qed

sublocale \[R-G: \text{Finite-Markov-Decision-Process } K S\]
by (standard, auto simp: S-finite S-not-empty K-finite K-S-closed)

lemmas \[K-S-closed\ast(intro) = R-G.set-pmf-closed\]

5 Relating the MDPs

5.1 Translating From K to K

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

lemma ccompatible-guard:
assumes \((l, g, \mu) \in \text{trans-of } A\)
shows compatible \(\mathcal{R} \land g\)

proof –
from assms valid have
\[ \forall (x, m) \in \text{collect-clock-pairs } g, m \leq \text{real } (k \times) \land x \in \mathcal{X} \land m \in \mathbb{N} \]

unfolding valid-abstraction-def clkp-set-def collect-clkt-def trans-of-def by fastforce
with assms compatible[of - k \mathcal{X}, folded \mathcal{R}-def] show ?thesis by auto
qed

lemmas compatible-def = compatible-def[unfolded ccval-def]

lemma region-set'eq:
fixes \(X' \cdot \text{c set}\)
assumes \(R \in \mathcal{R} \land u \in \mathcal{R} \land A \vdash l \rightarrow (g, \mu, \sigma) \cdot l'\)
shows 
\[ [[X := 0]u]_{\mathcal{R}} = \text{region-set'}_{R} (\text{SOME } r. \text{ set } r = X) \cdot 0 [[X := 0]u]_{\mathcal{R}} \in \mathcal{R} \cdot [X := 0]u \in [[X := 0]u]_{\mathcal{R}} \]

proof –
let \(?r = (\text{SOME } r. \text{ set } r = X)\)
from admissible-targets-alt[OF assms(2)] \(X\)-def finite have finite \(X\)
by (auto intro: finite-subset)
then obtain \(r\) where \(\text{set } r = X\) using finite-list by blast
then have \(?r = X\) by (intro someI)
with valid assms(3) have \(?r \subseteq X\)
by (simp add: transition-X)
from region-set'-id[of - \(X\) k, folded \(\mathcal{R}\)-def, OF assms(1,2) finite(1) - - this]
show 
\[ [[X := 0]u]_{\mathcal{R}} = \text{region-set'}_{R} (\text{SOME } r. \text{ set } r = X) \cdot 0 [[X := 0]u]_{\mathcal{R}} \in \mathcal{R} \cdot [X := 0]u \in [[X := 0]u]_{\mathcal{R}} \]
by force+
qed

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

lemma reset-sets-all-eqv:
assumes \(u \in V \land u' \in [[r \rightarrow (d :: nat)]u]_{\mathcal{R}} \land x \in \text{set } r \cdot r \subseteq \mathcal{X} \land d \leq k \times\)
shows \(u' \cdot x = d\)

proof –
from assms(1) have \(u : [r \rightarrow d]u \in [[r \rightarrow d]u]_{\mathcal{R}} \cdot [[r \rightarrow d]u]_{\mathcal{R}} \in \mathcal{R}\) by auto
then obtain \(I \cdot g\) where \(I : [[r \rightarrow d]u]_{\mathcal{R}} = \text{region } X\) \(I \cdot g\) valid-region \(X \land k \cdot i \cdot g\)
by (auto simp: \(\mathcal{R}\)-def)
with \(u(1)\) assms(3) have \(\text{intro-elem } x \cdot (r \rightarrow d)u \cdot (1)x\) \(\text{valid-intv } (k \times) \cdot (1)x\) by fastforce+
moreover from assms have \((r \rightarrow d)u \cdot x = d\) by simp
ultimately have \(I \cdot x = \text{Const } d\) using assms(5) by (cases \(I \cdot x\)) auto
moreover from \(I\) assms(2) have \(\text{intro-elem } x \cdot u' \cdot (1)x\) by fastforce
ultimately show \(u' \cdot x = d\) by auto
qed

lemma reset-eq:
assumes \(u \in V \cdot \left(\left[[r \rightarrow 0]u]_{\mathcal{R}} = \left(left \left[[r' \rightarrow 0]u]_{\mathcal{R}} \cdot \text{set } r \subseteq \mathcal{X} \land r' \subseteq \mathcal{X} \right. \right. \right. \right. \right. \)
shows \([r \rightarrow 0]u = [r' \rightarrow 0]u\) using assms

proof –
have \(\ast : u' \cdot x = 0\) if \(u' \in \left[[r \rightarrow 0]u]_{\mathcal{R}} \cdot \text{set } r \cdot u' \cdot x\)
using reset-sets-all-eqv[of \(u' \cdot r' \cdot 0\) \(x\)] that assms by auto
have \(\ast : u' \cdot x = 0\) if \(u' \in \left[[r' \rightarrow 0]u]_{\mathcal{R}} \cdot \text{set } r' \cdot u' \cdot x\)
using reset-sets-all-eqv[of \(u' \cdot r' \cdot 0\) \(x\)] that assms by auto
from regions-part-ex-reset[OF assms(1), of \(0\)] assms(2) have \(\ast :\)
\(\left[[r' \rightarrow 0]u] \in [[r \rightarrow 0]u]_{\mathcal{R}} \cdot (r \rightarrow 0]u] \in [[r \rightarrow 0]u]_{\mathcal{R}} \cdot [[r \rightarrow 0]u]_{\mathcal{R}} \in \mathcal{R}\)
by auto
have \(((r \rightarrow 0) u) \times (r' \rightarrow 0) u\) for 

proof (cases \(x \in \text{set } r\))
  
  case True
  then have \(((r \rightarrow 0) u) x = 0\) by simp
  moreover from \(* * *\) True have \(((r' \rightarrow 0) u) x = 0\) by auto
  ultimately show \(?thesis \). ..

next
  case False
  then have \(\text{id}: (r' \rightarrow 0) u \times u\) by simp
  show \(?thesis \)
  proof (cases \(x \in X\))
    case True
    then have \(\text{reset}: (r' \rightarrow 0) u x = 0\) by simp
    show \(?thesis \)
    qed
  next
    case False
    with \(\text{assms}(3-\)) have \(x \notin \text{set } r \neq x \notin \text{set } r'\) by auto
    then show \(?thesis \) by simp
    qed
  qed

next
  case False
  then have \(\text{reset}: (r' \rightarrow 0) u x = u x\) by simp
  with \(\text{id}\) show \(?thesis \) by simp
  qed

then show \(?thesis \) ..
  qed

lemma \textbf{admissible-targets-clocks}:
  assumes \((l, g, \mu) \in \text{trans-of } A (X, l') \in \mu\)
  shows \(X \subseteq X\) set \((\text{SOME } r. \text{set } r = X) \subseteq X\)
proof -
  from \textbf{admissible-targets}(2)[OF \textbf{assms}] \textbf{finite} have \(\text{finite } X \subseteq X\)
  by \((\text{auto intro: finite-subset simp: } X\text{-def})\)
  then obtain \(r \text{ where } \text{set } r = X\) using \textbf{finite-list} by \textbf{blast}
  with \(X \subseteq X\) show \(X \subseteq X\) set \((\text{SOME } r. \text{set } r = X) \subseteq X\)
  by \((\text{metis (mona-tags, lifting) someI-ex})\)+

  qed

lemma \textbf{rel-pmf} \((\lambda a. f a = b) \mu (\text{map-pmf } f \mu)\)
by \((\text{subst pmf.rel-map(2)})) \textbf{(rule rel-pmf-reflI, auto)}

lemma \textbf{K-pmf-rel}:
  defines \(f \equiv \lambda (l, u). (l, [u]_R)\)
  shows \(\text{rel-pmf} \lambda (l, u) \text{ st. } (l, [u]_R) = \text{st}) \mu (\text{map-pmf } f \mu)\) \textbf{unfolding } \textit{f-def}
by \((\text{subst pmf.rel-map(2)})) \textbf{(rule rel-pmf-reflI, auto)}

lemma \textbf{K-pmf-rel}:
assumes $A: \mu \in K (l, R)$
defines $f \equiv \lambda (l, u). (l, SOME u. u \in R)$
shows $\text{rel-pmf} (\lambda (l, u) st. (l, SOME u. u \in R) = st) \mu (\text{map-pmf} f \mu)$ unfolding $f$-def by $(\text{subst} \ \text{pmf}.\text{rel-map}(2))$ (rule $\text{rel-pmf-reflI}$, auto)

lemma $K$-elem-abs-inj:
assumes $A: \mu \in K (l, u)$
defines $f \equiv \lambda (l, u). (l, [u]_\mathcal{R})$
shows $\text{inj-on} f \mu$
proof –
  have $(l1, u1) = (l2, u2)$
    if $id: (l1, [u1]_\mathcal{R}) = (l2, [u2]_\mathcal{R})$ and $\text{elem}: (l1, u1) \in \mu (l2, u2) \in \mu$ for $l1 \ l2 u1 u2$
  proof –
  from $id$ have $\{\text{simp}\}: l2 = l1$ by $\text{auto}$
  from $A$
  show $\{\text{thesis}\}$
  proof ($\text{cases}, \ \text{safe}, \ \text{goal-cases}$)
  case ($4 - - \tau \mu'$)
  from $(\mu = \tau)$ $\text{elem}$ obtain $X1 X2$ where
    $u1 = [(\text{SOME} r. \text{set} r = X1) \rightarrow 0] u$ ($X1, l1) \in \mu'$
    $u2 = [(\text{SOME} r. \text{set} r = X2) \rightarrow 0] u$ ($X2, l1) \in \mu'$
  by $\text{auto}$
  with ($\cdot \in \text{trans-of} \cdot$ $\text{admissible-targets-clocks}$ $\text{have}$
    set ($\text{SOME} r. \text{set} r = X1) \subseteq X$ set ($\text{SOME} r. \text{set} r = X2) \subseteq X$
  by $\text{auto}$
  with $id$ $(u1 = \tau) \langle u2 = \tau \rangle$ $\text{reset-eq}[\text{of} u]$ ($\cdot \in S)$ $\text{show} \{\text{case}\}$ by $(\text{auto simp}: S\text{-def} V\text{-def})$
  qed ($\cdot, \ \text{insert elem}, \ \text{simp}+$
  qed
  then show $\{\text{thesis}\}$ unfolding $f$-def $\text{inj-on-def}$ by $\text{auto}$
  qed

lemma $K$-elem-repr-inj:
notes $\alpha$-interp.$\text{valid-regions-distinct-spec}[\text{intro}]$
assumes $A: \mu \in K (l, R)$
defines $f \equiv \lambda (l, R). (l, \text{SOME} u. u \in R)$
shows $\text{inj-on} f \mu$
proof –
  have $(l1, R1) = (l2, R2)$
    if $id: (l1, \text{SOME} u. u \in R1) = (l2, \text{SOME} u. u \in R2)$ and $\text{elem}: (l1, R1) \in \mu (l2, R2) \in \mu$
    for $l1 \ l2 R1 R2$
  proof –
  let $\tau r1 = \text{SOME} u. u \in R1$ and $\tau r2 = \text{SOME} u. u \in R2$
  from $id$ have $\{\text{simp}\}: l2 = l1 \tau r2 = \tau r1$ by $\text{auto}$
  \{ fix $g \mu' x$
    assume $(l, R) \in S (l, g, \mu') \in \text{PTA}.\text{trans-of} A R \subseteq \{v. \ \nu \vdash g\}$
    and $\mu = \text{map-pmf} (\lambda(X, l). (l, \text{region-set'} R (\text{SOME} r. \text{set} r = X) 0)) \mu'$
  from $(\mu = \tau)$ $\text{elem}$ obtain $X1 X2$ where
    $R1 = \text{region-set'} R (\text{SOME} r. \text{set} r = X1) 0$ ($X1, l1) \in \mu'$
    $R2 = \text{region-set'} R (\text{SOME} r. \text{set} r = X2) 0$ ($X2, l1) \in \mu'$
  by $\text{auto}$
  with ($\cdot \in \text{trans-of} \cdot$ $\text{admissible-targets-clocks}$ $\text{have}$
    set ($\text{SOME} r. \text{set} r = X1) \subseteq X$ set ($\text{SOME} r. \text{set} r = X2) \subseteq X$
  by $\text{auto}$
  with $\alpha$-interp.$\text{region-set'}$-$\text{closed}[\text{of} - 0]$ $\langle R1 = \tau \rangle \langle R2 = \tau \rangle$ ($\cdot \in S)$ $\text{have}$
    $R1 \in \mathcal{R}$ $R2 \in \mathcal{R}$
  unfolding $S\text{-def}$ by $\text{auto}$
  with $\text{region-not-empty}[\text{OF} \ \text{finite}(1)]$ $\text{have}$
    $R1 \neq \{\} \ R2 \neq \{\} \ \exists u. u \in R1 \ \exists u. u \in R2$
  by $(\text{auto simp}: \mathcal{R}\text{-def})$
  from someI-ex[\text{OF this}(3)] someI-ex[\text{OF this}(4)] $\text{have}$ $\tau r1 \in R1 \ \tau r1 \in R2$ by $\text{simp}+$
  with $\langle R1 \in \mathcal{R}\rangle \langle R2 \in \mathcal{R}\rangle$ $\text{have}$ $R1 = R2$ ..
\{ 
from A elem this show \( ? \)thesis by (cases, auto) 
qed 
then show \( ? \)thesis unfolding f-def inj-on-def by auto 
qed 

lemma \textit{K-elem-pmf-map-abs}: 
assumes \( A : \mu \in K \ (l, u) \ (l', u') \in \mu \) 
defines \( f \equiv \lambda \ (l, u). \ (l, [u]_R) \) 
shows \( \text{pmf} \ (\text{map-pmf} \ f \ \mu) \ (f \ (l', u')) = \text{pmf} \ \mu \ (l', u') \) 
using \( A \) unfolding f-def by (blast intro: pmf-map-inj K-elem-abs-inj) 

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

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

5.2 Translating Configurations 

5.2.1 States 

definition \textit{abss} :: \( s * (\ ' c, t) \ \text{cval} \Rightarrow s * (\ ' c, t) \ \text{cval} \ \text{set} \) 
where  
\( \text{abss} \equiv \lambda \ (l, u). \ \text{if} \ u \in V \ \text{then} \ (l, [u]_R) \ \text{else} \ (l, \lambda. \ -1) \) 

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

lemma \textit{S-reps-S[ intro]}:  
assumes \( s \in S \) 
shows \( \text{reps} \ s \in S \) 
using \assms \ \text{R-V} unfolding \( S \)-def \ S-def \ reps-def \ V-def by force 

lemma \textit{S-abss-S[ intro]}: 
assumes \( s \in S \) 
shows \( \text{abss} \ s \in S \) 
using \assms \ \text{ccompatible-inv} unfolding \( S \)-def \ S-alt-def \ abss-def \ ccompatible-def by force 

lemma \textit{S-abss-reps[simp]}: 
\( s \in S \Rightarrow \text{abss} \ (\text{reps} \ s) = s \) 
using \ \text{R-V \ alpha-interp.region-unique-spec} by (auto simp: \( S \)-def \ S-def \ reps-def \ abss-def; blast) 

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

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{abss-reps-id}:
  \begin{itemize}
    \item \textbf{notes} \texttt{R-G.cfg-onD-state[simp del]}
    \item \textbf{assumes} \(s' \in S\) \(s \in \text{set-pmf} (\text{action} \ \text{cfg}) \ \text{cfg} \in \text{R-G.cfg-on} \ s'\)
    \item \textbf{shows} \(\text{abss} \ (\text{reps} \ s) = s\)
  \end{itemize}
  \item \textbf{proof} –
  \begin{itemize}
    \item \textbf{from} \texttt{assms} \textbf{have} \(s \in S\) \textbf{by} auto
    \item \textbf{then show} \(\text{thesis}\) \textbf{by} auto
  \end{itemize}
\end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{abss-S[intro]}:
    \begin{itemize}
      \item \textbf{assumes} \((l, u) \in S\)
      \item \textbf{shows} \(\text{abss} \ (l, u) = (l, [u]_R)\)
    \end{itemize}
    \item \textbf{using} \texttt{assms unfolding abss-def by auto}
  \end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{fst-abss}:
    \begin{itemize}
      \item \textbf{fsts} \(\text{abss} \ \text{st} = \text{fst} \ \text{st}\) \textbf{for} \texttt{st}
      \item \textbf{by} \texttt{(cases st)} \texttt{(auto simp: abss-def)}
    \end{itemize}
  \end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{K-elem-abss-inj}:
    \begin{itemize}
      \item \textbf{assumes} \(A: \mu \in K \ (l, u) \ (l, u) \in S\)
      \item \textbf{shows} \(\text{inj-on} \ \text{abss} \ \mu\)
    \end{itemize}
    \item \textbf{proof} –
    \begin{itemize}
      \item \textbf{from} \texttt{assms} \textbf{have} \(\text{abss} \ s' = (\lambda (l, u). \ (l, [u]_R)) \ s'\) \textbf{if} \(s' \in \mu\) \textbf{for} \(s'\)
      \item \textbf{using} \texttt{that} \texttt{(auto split: prod.split)}
      \item \textbf{from} \texttt{inj-on-cong[OF this]} \texttt{K-elem-abss-inj[OF A(1)]} \texttt{show} \(\text{thesis}\) \textbf{by} force
    \end{itemize}
\end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{K-elem-reps-inj}:
    \begin{itemize}
      \item \textbf{assumes} \(A: \mu \in K \ (l, R) \ (l, R) \in S\)
      \item \textbf{shows} \(\text{inj-on} \ \text{reps} \ \mu\)
    \end{itemize}
    \item \textbf{proof} –
    \begin{itemize}
      \item \textbf{from} \texttt{assms} \textbf{have} \(\text{reps} \ s' = (\lambda (l, R). \ (l, \text{SOME} u. \ u \in R)) \ s'\) \textbf{if} \(s' \in \mu\) \textbf{for} \(s'\)
      \item \textbf{using} \texttt{that} \texttt{(auto split: prod.split)}
      \item \texttt{from inj-on-cong[OF this]} \texttt{K-elem-reps-inj[OF A(1)]} \texttt{show} \(\text{thesis}\) \textbf{by} force
    \end{itemize}
\end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{P-elem-pmf-map-abss}:
    \begin{itemize}
      \item \textbf{assumes} \(A: \mu \in K \ (l, u) \ (l, u) \in S \ s' \in \mu\)
      \item \textbf{shows} \(\text{pmf} \ (\text{map-pmf} \ \text{abss} \ \mu) \ (\text{abss} \ s') = \text{pmf} \ \mu \ s'\)
    \end{itemize}
    \item \textbf{using} \texttt{A} \texttt{by (blast intro: pmf-map-inj K-elem-abss-inj)}
  \end{itemize}
\end{proof}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{K-elem-pmf-map-reps}:
    \begin{itemize}
      \item \textbf{assumes} \(A: \mu \in K \ (l, R) \ (l, R) \in S \ (l', R') \in \mu\)
      \item \textbf{shows} \(\text{pmf} \ (\text{map-pmf} \ \text{reps} \ \mu) \ (\text{reps} \ (l', R')) = \text{pmf} \ \mu \ (l', R')\)
    \end{itemize}
    \item \textbf{using} \texttt{A} \texttt{by (blast intro: pmf-map-inj K-elem-reps-inj)}
  \end{itemize}
\end{proof}

\textbf{We need that} \(\mathcal{X}\) \textbf{is non-trivial here}

\begin{proof}
\begin{itemize}
  \item \textbf{lemmas} \texttt{not-S-reps}:
    \begin{itemize}
      \item \((l, R) \notin S \implies \text{reps} \ (l, R) \notin S\)
    \end{itemize}
    \item \textbf{proof} –
    \begin{itemize}
      \item \textbf{assume} \((l, R) \notin S\)
      \item \textbf{let} \(u = \text{SOME} \ u. \ u \in R\)
    \end{itemize}
\end{itemize}
\end{proof}
have \( \neg \exists u \vdash \text{inv-of } A \ l \) if \( R \in R \ l \in L \)

proof –

from region-not-empty[OF finite(1)] \( R \in R \) have \( \exists u. u \in R \) by (auto simp: R-def)
from someI-ex[OF this] have \( \exists u \in R \).
moreover from \((l\ R) \notin S\) that have \( \neg R \subseteq \{\text{inv-of } A \ l\} \) by (auto simp: S-def)
ultimately show \(?thesis\)
qed

with non-empty \((l\ R) \notin S\) show \(?thesis\)
unfolding ccompatible-def by fastforce

5.2.2 Intermezzo

abbreviation \text{time-bisim} (infixr \sim 60) where
\( s \sim s' \equiv \text{abss } s = \text{abss } s' \)

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]:
\([u]_R = [u']_R\) if \( u \in V \ (l, u) \sim (l', u') \)
proof –

have \((l', \neg V) \neq (l, [u]_R)\)
using that by blast

with that have \( u' \in V \)
by (force simp: abss-def)

with that show \(?thesis\)

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

lemma bisim-val-id2[intro]:
\(u' \in V \Rightarrow (l, u) \sim (l', u') \Rightarrow [u]_R = [u']_R\)
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: \((\text{action } l u \tau \mu')\)
  with assms obtain \(l1 l2 X1 X2\) where \(A:\)
  \((X1, l1) \in \text{set-pmf } \mu' (X2, l2) \in \text{set-pmf } \mu'
  x = (l, [X1:=]u) x' = (l2, [X2:=]u)\)
  by auto
  from \(x \sim x'\) \(A (s \in S) (s = (l, u))\) have \([X1:=]u]_R = [X2:=]u]_R\)
  using bisim-val-id[OF S-V] K-S-closed assms(2–4) by (auto intro!: bisim-val-id[OF S-V])
  then have \([X1:=]u]_R = [X2:=]u]_R\)
  using A admissible-targets-clocks(2)[OF 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
\(absp \varphi \equiv \varphi o \text{reps}\)

definition repp where
\(repp \varphi \equiv \varphi o absp\)

5.2.4 Distributions

definition \(abst:: (\tau s \times (\tau c, \tau t) \text{cval}) \text{pmf} \Rightarrow (\tau s \times (\tau c, \tau t) \text{cval set}) \text{pmf}\)
where
\(abst = \text{map-pmf abss}\)

lemma abss-SD:
assumes \(abss s \in S\)
obtains \(l u\) where \(s = (l, u) u \in [u]_R [u]_R \in \mathcal{R}\)
proof –
  obtain \(l u\) where \(s = (l, u)\) by force
  moreover from \(S\)-abss-\(S[\text{OF \text{assms}}]\) have \(s \in S\).
  ultimately have \(abss s = (l, [u]_R) a \in V u \in [u]_R [u]_R \in \mathcal{R}\) by auto
  with \(s = -\) show \(?thesis by (auto intro: that)\)
qed

lemma abss-SD':
assumes \( \text{abss} s \in S \) \( \text{abss} s = (l, R) \)

obtains \( u \) where \( s = (l, u) u \in [u]_R \) \([u]_R \in \mathcal{R} \) \( R = [u]_R \)

proof –

from \( \text{abss-SD[OF assms(1)]} \) obtain \( l' u \) where \( u: s = (l', u) u \in [u]_R \) \([u]_R \in \mathcal{R} \)

by blast+

with \( \mathcal{R} \cdot V \) have \( u \in V \) by auto

with \( s = \cdot \text{assms}(2) \) have \( l' = l R = [u]_R \)

unfolding \( \text{abss-def} \) by auto

with \( \langle s = - \rangle \) assms(2) have \( l' = l R \)

unfolding \( \text{infR} \) by auto

with \( u \) show \( \text{thesis} \) by (auto intro: that)

qed

definition \( \text{infR} R \equiv \lambda c. \text{of-int} \lfloor \text{SOME} u. u \in R \rfloor c \)

term let \( a = 3 \) in \( b \)

definition \( \text{delayedR} R u \equiv \)

\( u \oplus ( \)

let \( I = (\text{SOME} I, \exists r. \text{valid-region} X k I r \land R = \text{region} X I r) ; \)

\( m = 1 - \text{Max} (\{\frac{u c}{c} | c \in X \land \text{isIntv} (I c) \} \cup \{0\}) \)

in \( \text{SOME} t. \) \( u \oplus t \in R \land t \geq m / 2 \)

\)

lemma \( \text{delayedR-correct-aux-aux} : \)

fixes \( c :: \text{nat} \)

fixes \( a b :: \text{real} \)

assumes \( c < a a < \text{Suc} c b \geq 0 a + b < \text{Suc} c \)

shows \( \frac{a + b}{2} = \frac{a}{2} + b \)

proof –

have \( f1: a + b < \text{real} (c + 1) \)

using \( \text{assms}(4) \) by auto

have \( f2: \forall r. (r::real) + (\neg r + ra) = ra \)

by linarith

have \( f3: \forall r. (r::real) = (\neg r) \)

by linarith

have \( f4: \forall r. (r::real) + (ra + r) = ra \)

by linarith

then have \( f5: \forall n. r + \text{frac} r = \text{real} n \lor \neg r < \text{real} (n + 1) \lor \neg \text{real} n < r \)

using \( f2 \) by (metis \( \text{nat-intv-frac-decomp} \))

then have \( \text{frac} a + \text{real} c = a \)

using \( f4 f3 \) by (metis \( \text{One-nat-def add.right-neutral add.right-neutral Suc-right assms(1) assms(2)} \))

then show \( \text{thesis} \)

using \( f5 f1 \) \( \text{assms}(1) \) \( \text{assms}(3) \) by fastforce

qed

lemma \( \text{delayedR-correct-aux} : \)

fixes \( I r \)

defines \( R \equiv \text{region} X I r \)

assumes \( u \in R \) \( \text{valid-region} X k I r \forall c \in X. \neg \text{isConst} (I c) \)

\( \forall c \in X. \text{isIntv} (I c) \longrightarrow (u \oplus t) c < \text{intv-const} (I c) + 1 \)

\( t \geq 0 \)

shows \( u \oplus t \in R \) unfolding \( R \)-def
proof
from assms have $R \in \mathcal{R}$ unfolding $\mathcal{R}$-def by auto
with $(u \in R) \land \mathcal{R}$-V have $u \in V$ by auto
with $t \geq 0$ show $\forall x \in \mathcal{X}. 0 \leq (u + t) \cdot x$ unfolding $V$-def by (auto simp: cval-add-def)
have inv-elem $x$ $(u \oplus t) \cdot (I \cdot x)$ if $x \in \mathcal{X}$ for $x$
proof (cases $I \cdot x$
  case Const
  with assms $\langle x \in \mathcal{X} \rangle$ show ?thesis by auto
next
  case (Intv $c$)
  with assms $\langle x \in \mathcal{X} \rangle$ show ?thesis by (simp add: cval-add-def) (rule; force)
next
  case (Greater $c$)
  with assms $\langle x \in \mathcal{X} \rangle$ show ?thesis by (fastforce simp add: cval-add-def)
qed
then have $\frac{u}{x} \cdot x + t = \frac{u}{x} \cdot x + t$ if $x \in ?X_0$ for $x$
proof
  show ?thesis
    apply (rule delayedR-correct-aux-aux[where $c = \text{intv-const} \cdot (I \cdot x)]))
    using assms $\langle x \in ?X_0 \rangle$ by (force simp add: cval-add-def)+
qed
then have $\frac{u}{x} \cdot x \leq \frac{u}{x} \cdot x \leq \frac{u}{x} \cdot y$ 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. \langle (x, y) \in r \rangle = (\frac{u}{x} \cdot (u \oplus t) \cdot x) \leq \frac{u}{x} \cdot (u \oplus t) \cdot y)$
unfolding cval-add-def by auto
qed

lemma delayedR-correct-aux':
  fixes $I \cdot r$
  defines $R \equiv \text{region} \cdot X \cdot I \cdot r$
  assumes $u \oplus t_1 \in R$ valid-region $\mathcal{X} \cdot k \cdot I \cdot r \cdot c \in \mathcal{X}. \neg \text{isConst} \cdot (I \cdot c)$
  $\forall c \in \mathcal{X}. \text{isIntv} \cdot (I \cdot c) \rightarrow (u \oplus t_2) \cdot c \prec \text{intv-const} \cdot (I \cdot c) + 1$
  $t_1 \leq t_2$
  shows $u \oplus t_2 \in R$
proof
  have $(u \oplus t_1) \oplus (t_2 - t_1) \in R$ unfolding $R$-def
  using assms by (rule delayedR-correct-aux, auto simp: cval-add-def)
  then show $u \oplus t_2 \in R$ by (simp add: cval-add-def)
qed

lemma valid-regions-intv-distinct:
valid-region $\mathcal{X} \cdot k \cdot I \cdot r \rightarrow$ valid-region $\mathcal{X} \cdot k \cdot I' \cdot r' \Rightarrow u \in \text{region} \cdot X \cdot I \cdot r \Rightarrow u \in \text{region} \cdot X \cdot I' \cdot r'$
\Rightarrow $x \in X \Rightarrow I \cdot x = I' \cdot x$
proof goal-cases
  case A: $I$
  note $x = \langle x \in \mathcal{X} \rangle$
  with $A$ have valid-intv $\langle k \cdot x \rangle \cdot (I \cdot x)$ by auto
  moreover from $A(2)$ $x$ have valid-intv $\langle k \cdot x \rangle \cdot (I' \cdot x)$ by auto
  moreover from $A(3)$ $x$ have inv-elem $x$ $u \cdot (I \cdot x)$ by auto
  moreover from $A(4)$ $x$ have inv-elem $x$ $u \cdot (I' \cdot x)$ by auto
  ultimately show $I \cdot x = I' \cdot x$ using valid-intv-distinct by fastforce
qed
lemma delayedR-correct:
  fixes I r
defines R’ ≡ region X I r
assumes u ∈ R R ∈ R valid-region X k I r ∀ c ∈ X. ¬ isConst (I c) R’ ∈ Succ R R
shows
delayedR R’ u ∈ R’
  ∃ t ≥ 0. delayedR R’ u = u ⊕ t
    ∧ t ≥ (1 − Max {(frac (u c) | c. c ∈ X ∧ isIntv (I c))} ∪ {0})) / 2
proof –
  let ?u = SOME u. u ∈ R
let ?I = SOME I. ∃ r. valid-region X k I r ∧ R’ = region X I r
let ?S = {frac (u c) | c. c ∈ X ∧ isIntv (I c)}
let ?m = I − Max (?S ∪ {0})
let ?t = SOME t. u ⊕ t ∈ R’ ∧ t ≥ ?m / 2
have Max (?S ∪ {0}) ≥ 0 ?m ≤ I using finite(1) by auto
have Max (?S ∪ {0}) ∈ ?S ∪ {0} using finite(1) by (rule Max-in; auto)
with frac-it-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 l-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-intr-distinct assms(4) t(1) that unfolding R’-def by auto
have l-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 some1[where P = λ I. ∃ r. valid-region X k I r ∧ R’ = region X I r]; auto)
with l-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 ∈ invt-const (I c) + 1 if c ∈ X isIntv (I c) for c
proof (cases u c > invt-const (I c))
case True
from t that assms have u c + t < invt-const (I c) + 1 unfolding cval-add-def by fastforce
with (t ≥ 0) True have ∗: invt-const (I c) < u c u c < invt-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
  ... = invt-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 ≤ invt-const (I c) by auto
moreover from (0 ≤ ?m) (?m ≤ 1) have ?m / 2 < 1 by auto
ultimately have u c + ?m / 2 < invt-const (I c) + 1 by linarith
then show ?thesis by (simp add: cval-add-def)
qed
then 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 somel; auto)
with (?m ≥ 0) show ?thesis by auto
next
case False
have u ⊕ ?m / 2 ∈ R’ unfolding R’-def
apply (rule delayedR-correct-aux’)

26
lemma rept-ex:
S-I
qed
proof cases
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 (∃ R'. (l, u) ∈ S ∧ μ-abs = return-pmf (l, R') ∧
  ([(u)]) = R' ∧ (∀ c ∈ X. u c > k c)))
  then return-pmf (l, u ⊕ 0.5)
  else if
  (∃ R'. (l, u) ∈ S ∧ μ-abs = return-pmf (l, R') ∧ R' ∈ Succ R ([(u)] ≠ R'
  ∧ (∃ u ∈ R'. u c ∈ X. ∃ d. d ≤ k c ∧ u c = real d))
  then return-pmf (l, delayedR (SOME R'. μ-abs = return-pmf (l, R')) u)
  else SOME μ. μ ∈ K s ∧ abst μ = μ-abs

lemma S-L:
l ∈ L if (l, R) ∈ S
using that unfolding S-def by auto

lemma S-inv:
(l, R) ∈ S ⇒ R ⊆ ⟦inv-of A l⟧
unfolding S-def by auto

lemma upper-right-closed:
assumes ∀ c ∈ X. real (k c) < u c u ∈ R R ∈ R t ≥ 0
shows u ⊕ t ∈ R
proof –
from i: R ∈ R obtain I r where R:
  R = region X I r valid-region X k I r
unfolding R-def by auto
from assms R-V have u ∈ V by auto
from assms R have ∃ c ∈ X. I c = Greater (k c) by safe (case-tac I c; fastforce)
with R (u ∈ V) assms show
  u ⊕ t ∈ R
unfolding V-def by safe (rule: force simp: cval-add-def)
qed

lemma S-I[intro]:
(l, u) ∈ S if l ∈ L u ∈ V u ⊢ inv-of A l
using that by (auto simp: S-def V-def)

lemma rept-ex:
assumes μ ∈ K (abss s)
shows rept s μ ∈ K s ∧ abst (rept s μ) = μ using assms
proof cases
case prems: (delay l R R')
then have R ∈ R by auto
from prems(2) have s ∈ S by (auto intro: S-abss-S)
from abss-SD[OF prems(2)] obtain l' u' where s = (l', u') u' ∈ [(u')]R
  by metis
with prems(3) have ∃ s = (l, u') ∧ u' ∈ R
  apply simp
  apply (subst (asm) abss-S[OF 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 $R'$:
\[ t \geq 0 \; R' = [u' \oplus t]_R \]
by auto
with $(s \in S) \ast \text{have } u' \oplus t \in R' \; u' \oplus t \in V \; l \in L \text{ by auto}
with prems(5) have $(l, u' \oplus t) \in S \text{ unfolding } S\text{-def } V\text{-def by auto}
with \langle R' = [u' \oplus t]_R \rangle \text{ have } \ast \text{: abs } (l, u' \oplus t) = (l, R') \text{ by } (\text{auto simp: abs-S})
let $?\mu = \text{return-pmf } (l, u' \oplus t)$
have $?u \in K \text{ using } \ast \text{ : } (s \in S) \; (t \geq 0) \; (u' \oplus t \in R') \text{ prems by blast}$
moreover have abst $?\mu = \mu$ by (simp add: \ast abst-def prems(1))
moreover note default = calculation
have $R' \in \mathcal{R} \text{ using } \text{prems}(4) \text{ by auto}$
have $R : [u']_\mathcal{R} = R \text{ by } (\text{simp add: } \ast \text{ : } (R \in \mathcal{R}) \text{ alpha-interp.region-unique-spec})$
from $R' \in \mathcal{R}$ obtain $I \; r \text{ where } R'$:
\[ R' = \text{region } \mathcal{X} \; I \; r \text{ valid-region } \mathcal{X} \; k \; I \; r \]
unfolding $R\text{-def by auto}$
have $u' \in V \text{ using } \ast \text{ : } \text{prems } \mathcal{R}\text{-V by force}$
let $?\mu' = \text{return-pmf } (l, u' \oplus 0.5)$
have elapsed: abst (return-pmf $(l, u' \oplus t)) = \mu \text{ return-pmf } (l, u' \oplus t) \in K \; s$
if $u' \oplus t \in R' \; t \geq 0 \; \text{for } t$
proof -
  let $?u = u' \oplus t$ let $?\mu' = \text{return-pmf } (l, u' \oplus t)$
  from $?u \in R' \; (R' \in \mathcal{R}) \; \mathcal{R}\text{-V have } ?u \in V \text{ by auto}$
  with $?u \in R' \; \langle R' \in \mathcal{R} \rangle \text{ have } [?u]_\mathcal{R} = R' \text{ using } \text{alpha-interp.region-unique-spec by auto}$
  with $?u \in V \; (?u \in R' \; (l \in L) \text{ prems}(4,5) \text{ have } \text{abs } (l, ?u) = (l, R')$
  by (subst abss $\ast$)
  with prems(1) have abst $?\mu' = \mu$ by (auto simp: abst-def)
  moreover from $\ast \; (?u \in R') \; (s \in S) \; \text{prems } (t \geq 0) \; \text{have } ?\mu' \in K \; s \text{ by auto}$
  ultimately show abst $?\mu' = \mu$ $?\mu' \in K \; s \text{ by auto}$
qed
show $?\text{thesis}$
proof (cases $R = R'$)
  case $T$: True
  show $?\text{thesis}$
  proof (cases $\forall \; c \in \mathcal{X}. \; u' \; c > k \; c$)
    case $\text{True}$
    with $T \ast R \text{ prems}(1,4) \; (s \in S) \; \text{have}$
    \[ \text{rept } s \; \mu = \text{return-pmf } (l, u' \oplus 0.5) \; (\text{is } - = ?\mu) \]
    unfolding $\text{rept-def by auto}$
    from $\text{upper-right-closed}(\text{OF True}) \ast \; \langle R' \in \mathcal{R} \rangle \; T \text{ have } u' \oplus 0.5 \in R' \text{ by auto}$
    with elapsed (rept - - = - ) show $?\text{thesis by auto}$
  next
    case $F$: False
    with $T \ast R \text{ prems}(1) \text{ have}$
    \[ \text{rept } s \; \mu = (\text{SOME } \mu'. \; \mu' \in K \; s \land \text{abst } \mu' = \mu) \]
    unfolding $\text{rept-def by auto}$
    with default show $?\text{thesis by simp (rule someI; auto)}$
  qed
next
  case $\text{False}$
  with $T \ast R \text{ prems}(1) \text{ have}$
  \[ \text{rept } s \; \mu = (\text{SOME } \mu'. \; \mu' \in K \; s \land \text{abst } \mu' = \mu) \]
  unfolding $\text{rept-def by auto}$
  with default show $?\text{thesis by simp (rule someI; auto)}$
  qed
next
  case $T$: False
  show $?\text{thesis}$
  proof (cases $\forall \; u \in R'. \forall \; c \in \mathcal{X}. \; \exists \; d. \; d \leq k \; c \land u \; c = \text{real } d$)
    case $\text{False}$
    with $F \ast R \text{ prems}(1) \text{ have}$
    \[ \text{rept } s \; \mu = (\text{SOME } \mu'. \; \mu' \in K \; s \land \text{abst } \mu' = \mu) \]
    unfolding $\text{rept-def by auto}$
    with default show $?\text{thesis by simp (rule someI; auto)}$
  next
    case $\text{True}$
    from $\text{True } F \ast R \text{ prems}(1,4) \; (s \in S) \; \text{have}$
    \[ \text{rept } s \; \mu = \text{return-pmf } (l, \text{delayedR } (\text{SOME } R'. \; \mu = \text{return-pmf } (l, R'))\; u') \]
  next
(is = return-pmf (l, delayedR ?R u'))

unfolding rept-def by auto

let ?a = 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 -i ∈ 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 i 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

next

case prems: (action l R τ μ')

from abs-SD'[OF prems(2,3)] obtain u where u:

s = (l, u) u ∈ [u]R [u]R ∈ R R = [u]R

by auto

with -i ∈ 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 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
  reps 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
      reps 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 have [u ⊕ 0.5]R = R'
      using T alpha-interp.region-unique-spec u(3,4) by blast
    moreover note * = (reps - - = R' (abss s ∈ S) (abss s = - - prems(5))
    ultimately have abst (reps 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 ∈ S. ∀ c ∈ X. real (k c) < u c)
    with * have s = - - (l, u) ∈ S, ∀ c ∈ X. real (k c) < u c
    by (auto simp: T dest: S-inv)
    ultimately show ?thesis by auto
  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 s ∈ S. ∀ c ∈ X. real (k c) < u c
    then have abss s = (l, u) ⊕ 0.5 by auto
    with (s ∈ S) S-abss-S have (l, u) ∈ S by auto
    with S-inv have [u]R ⊆ {u. u ⊕ 0.5 ∈ R-V} by auto
    show ?thesis
    proof (cases ∀ c ∈ X. real (k c) < u c)
      case True
      with * have * = (∀ s ∈ S. ∀ c ∈ X. real (k c) < u c)
      then have abss s = (l, u) ⊕ 0.5 by auto
      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 = [u]R by blast
      moreover note * = (reps - - = R' (μ s = - -) (l, u) ∈ 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
  rep s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ)
unfolding rep-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 rep s μ = (SOME μ′. μ′ ∈ K s ∧ abst μ′ = μ)
unfolding rep-def by auto
with ⟨μ = -⟩
  show ?thesis by simp (rule someI[where x = return-pmf s], auto simp: abst-def)
qed
qed

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

lemma abst-rept2:
  assumes μ ∈ K s s ∈ S
  shows abst (rep (reps s) μ) = μ
using assms by auto

lemma rep-K2:
  assumes μ ∈ K s s ∈ S
  shows rep (reps s) μ ∈ K (reps s)
using assms by auto

lemma theI′:
  assumes P a
  and ∀x. P x → x = a
  shows P (THE x. P x) ∧ (∀y. P y → y = (THE x. P x))
using theI assms by metis

lemma cont-cfg-defined:
  fixes cfg s
  assumes cfg ∈ valid-cfg s ∈ abst (action cfg)
  defines x ≡ THE x. abss x = s ∧ x ∈ action cfg
  shows (abss x = s ∧ x ∈ action cfg) ∧ (∀y. abss y = s ∧ y ∈ action cfg → y = x)
proof -
  from assms(2) obtain s' where s' ∈ 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' :: (∗Tai ∗) (cval ⊢ cval) cval ⇒ (∗Tai ∗) (cval ∗) cval set cval
where
absc' cval = cval-corec
(abss (state cval))
(abst o action)
(λ cval. cont cval (THE x. abss x = s ∧ x ∈ action cval)) cval

5.2.5 Configuration
definition
\[
\text{abscl :: (}'s \ast ('c, t) \text{ cval}) \text{ cfg} \Rightarrow ('s \ast ('c, t) \text{ cval set}) \text{ cfg}
\]

**where**

\[
\text{abscl cfg = cfg-corec}
\]

\[
\begin{align*}
\text{(abss (state cfg))} \\
\text{(abst o action)} \\
(\lambda \text{cfg } s. \text{ cont cfg (THE x. abscl x = s \land x \in action cfg)}) \text{ cfg}
\end{align*}
\]

**definition**

\[
\text{repcsl :: (}'s \ast ('c, t) \text{ cval} \Rightarrow ('s \ast ('c, t) \text{ cval set}) \text{ cfg} \Rightarrow ('s \ast ('c, t) \text{ cval}) \text{ cfg}
\]

**where**

\[
\begin{align*}
\text{repcsl s cfg = cfg-corec}
\end{align*}
\]

\[
\begin{align*}
\text{s} \\
(\lambda (s, cfg). \text{rept s (action cfg)}) \\
(\lambda (s, cfg) s'. (s', \text{cont cfg (abss s')}) (s, cfg)
\end{align*}
\]

**definition**

\[
\begin{align*}
\text{repc cfg = repcs (reps (state cfg)) \text{ cfg}}
\end{align*}
\]

**lemma** \(S\)-state-absc-repc\[\text{simp}]:

\[
\text{state cfg } \in S \implies \text{state (abscl (repc cfg))} = \text{state cfg}
\]

**by** \(\text{simp add: absc-def repc-def repcs-def}\)

**lemma** action-repc:

\[
\text{action (repc cfg) = \text{rept (reps (state cfg)) (action cfg)}
\]

**unfolding** replc-def repcs-def **by** simp

**lemma** action-absc:

\[
\text{action (absc cfg) = abst (action cfg)}
\]

**unfolding** absc-def **by** simp

**lemma** action-absc\[\text{'}\]:

\[
\text{action (absc cfg) = map-pmf abss (action cfg)}
\]

**unfolding** absc-def **unfolding** abst-def **by** simp

**lemma**

\[
\text{notes R-G.cfg-onD-state[\text{simp del}]
}\]

**assumes** state cfg \(\in S\) \(s' \in \text{set-pmf (action (repc cfg)) cfg} \in R-G.cfg-onS\) \(\text{state cfg}\)

**shows** cont \(\text{(repc cfg)} s' = \text{repcsl s' (cont cfg (abss s'))}\)

**using** \(\text{assms by (auto simp: repc-def repcs-def abss-reps-id)}\)

**lemma** cont-repcs1:

\[
\text{notes R-G.cfg-onD-state[\text{simp del}]
}\]

**assumes** abss s \(\in S\) \(s' \in \text{set-pmf (action (repc s cfg)) \text{ cfg} \in R-G.cfg-onS\) \(\text{abss s}\)

**shows** cont \(\text{(reps s cfg)} s' = \text{repcsl s' (cont cfg (abss s'))}\)

**using** \(\text{assms by (auto simp: repc-def repcs-def abss-reps-id)}\)

**lemma** cont-absc-1:

\[
\text{notes MDP.cfg-onD-state[\text{simp del}]
}\]

**assumes** cfg \(\in \text{valid-cfg s' \in set-pmf (action cfg)}\)

**shows** cont \(\text{(absc cfg)} \text{ (abss s') = absc} \text{ (cont cfg s')}\)

**proof**

**define** \(x\) **where** \(x \equiv \text{THE x. x } \sim s' \land x \in \text{set-pmf (action cfg)}\)

**from** \(\text{assms(2)}\) **have** abss s' \(\in \text{set-pmf (abst (action cfg))}\) **unfolding** abst-def **by** auto

**from** \(\text{cont-cfg-defined[OF assms(1)] this\ have}\)

\[
(x \sim s' \land x \in \text{set-pmf (action cfg)}) \land (\forall y. y \sim s' \land y \in \text{set-pmf (action cfg)} \Rightarrow y = x)
\]

**unfolding** x-def.

**with** \(\text{assms have s' = x}\) **by** fastforce

**then show** \(?\text{thesis}\)

**unfolding** absc-def abst-def repc-def x-def **using** \(\text{assms(2)}\) **by** auto

qed
lemma state-repcD:
  state (repc cfg) = reps (state cfg)
unfolding repc-def repes-def by simp

lemma abss-reps-idD:
  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 cfg. P cfg ⇒ state cfg ∈ S
  assumes cfg. P cfg ⇒ action cfg ∈ K (state cfg)
  assumes cfg. t. P cfg ⇒ t ∈ action cfg ⇒ P (cont cfg t)
  shows cfg ∈ valid-cfg
proof
  from assms have cfg ∈ MDP.cfg-on (state cfg) by (coinduction arbitrary: cfg) auto
  moreover from assms have state cfg ∈ S by auto
  ultimately show ?thesis by (intro MDP.valid-cfgI)
qed

lemma state-repD[simp]:
  assumes cfg ∈ R-G.cfg-on s
  shows state (repc cfg) = reps s
using assms unfolding repc-def repes-def by auto

lemma ccompatible-plsD[intro]:
  assumes compatible R g R ⊆ R u ∈ R u ⊢ g
  shows R ⊆ {u, u ⊢ g}
using assms unfolding ccompatible-def by auto

lemma action-abscD[dest]:
  cfg ∈ MDP.cfg-on s ⇒ action (absc cfg) ∈ K (abss s)
unfolding absc-def abst-def
proof simp
  assume cfg: cfg ∈ MDP.cfg-on s
  then have action cfg ∈ K s by auto
  then show map-pmf abss (action cfg) ∈ K (abss s)
proof cases
  case prems: (delay l u t)
  then have [u ⊕ t]R ∈ R by auto
  moreover with prems ccompatible-inv[of l] have
    [u ⊕ t]R ⊆ {v. v ⊢ PTA.inv-of A l}
  unfolding ccompatible-def by force
  moreover from prems have abss (l, u ⊕ t) = (l, [u ⊕ t]R) by (subth abss-S) auto
  ultimately show ?thesis using prems by auto
next
  case prems: (action l u g μ)
  then have [u]R ∈ R by auto
  moreover with prems ccompatible-guard have [u]R ⊆ {u. u ⊢ g}
    by (intro ccompatible-pls) auto
  moreover have
    map-pmf abss (action cfg) = map-pmf (λX. l). (l, region-set’ ([u]R) (SOME r. set r = X) 0)) μ
  proof
    have abss (l’, [X:=0]u) = (l’, region-set’ ([u]R) (SOME r. set r = X) 0)
      if (X, l’) ∈ μ for X l’
proof
  from that prems have \( A \vdash l \rightarrow s' \)
  by auto
  from that prems \( \text{MDP.action-closed} \{ \text{OF - cfg} \} \) have \( (l', [X:=0]u) \in S \) by force
  then have \( \text{abss} (l', [X:=0]u) = (l', [[X:=0]u]_R) \) by auto
  also have
    \( \ldots = (l', \text{region-set'} ([u]_R) (\text{SOME } r. \text{ set } r = X) 0) \)
  using \( \text{region-set'}-eq(1) \{ \text{OF - } A \vdash l \rightarrow s' \} \) prems by auto
  finally show \(?thesis\).
qed
  then show \(?thesis\)
  unfolding \( \text{prems(1)} \)
  by (auto intro: \( \text{pmf.map-cong simp: map-pmf-comp} \))
qed
  ultimately show \(?thesis\) using \( \text{prems} \) by (auto)
next
  case prems: loop
  then show \(?thesis\) by auto
qed
qed

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

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

lemma \( \text{action-abst-repcs} \):
  assumes \( \text{cfg} \in R\text{-G.valid-cfg} \) \( \text{abss s} = \text{state cfg} \)
  shows \( \text{abst \{(action \{(repcs s cfg)\})\}} = \text{action cfg} \)
proof
  from \( \text{assms} \) show \(?thesis\)
  unfolding \( \text{repcs-def repcs-def} \)
  apply simp
  apply (subst \( \text{abst-rept-id} \))
  by (auto dest: \( \text{R-G.cfg-onD-action R-G.valid-cfgD} \))
qed
lemma action-abst-reps:
  assumes cfg ∈ R-G.valid-cfg
  shows abst (action (reps 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-reps[of assms] show ?thesis unfolding repcs-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 t' where t' = cont cfg s
      define t where t ≡ THE y. abss y = s ∧ y ∈ action (reps x cfg)
      define t'' where t'' ≡ THE y. abss y = s ∧ y ∈ action (reps x' cfg)
      from prems have valid: repcs x cfg ∈ valid-cfg by (intro repcs-valid)
      from prems have ∗: s ∈ abst (action (reps 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 (reps x cfg)
      unfolding t-def by auto
      have cont (absc (reps x cfg)) s = cont (absc (reps x cfg)) (abss t) using t by auto
      have cont (absc (reps x cfg)) s = absc (cont (reps x cfg) t)
      using t valid by (auto simp: cont-absc-1)
      also have ... = abss (reps t (cont cfg s))
      using prems t by (subst cont-repcs1) (auto dest: R-G.valid-cfgD)
      finally have cont-x: cont (absc (reps x cfg)) s = absc (reps t (cont cfg s))
      from prems have valid: repcs x' cfg ∈ valid-cfg by auto
      (s ∈ action cfg) prems have s ∈ abst (action (reps 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 (reps x' cfg)
      unfolding t'-def by auto
      have cont (absc (reps x' cfg)) s = cont (absc (reps x' cfg)) (abss t') using t' by auto
      have cont (absc (reps x' cfg)) s = absc (cont (reps 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-repcsI) (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-abs-S R-G.valid-cfg-action R-G.valid-cfg-cont)
qed

named-theorems R-G-I


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 $t$ * ($t$ ∼ $s'$) valid
  by (fastforce dest: R-G-I intro: repcs-bisim simp: cont-repcsI $cfg'$-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 \_cfg. P cfg \implies state cfg ∈ S
  assumes \_cfg. P cfg \implies action cfg ∈ K (state cfg)
  assumes \_cfg. t. P cfg \implies t ∈ action cfg \implies P (cont cfg t)
shows cfg ∈ R-G.valid-cfg
proof –
  from assms have cfg ∈ R-G.cfg-on (state cfg) by (coinduction arbitrary: cfg) auto
moreover from assms have state cfg ∈ S by auto
ultimately show \_thesis by (intro R-G.valid-cfgI)
qed

lemma absc-valid[intro]:
  assumes cfg ∈ valid-cfg
shows absc cfg ∈ 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 ≡ THE x. abss x = s' ∧ x ∈ set-pmf (action cfg)
  let ?cfg = cont cfg t
from prems obtain s where s' = abss s s ∈ action cfg by (auto simp: action-absc')
  with cont-cfg-defined[OF prems(1), of s'] have
    abss t = s' \ t ∈ set-pmf (action cfg)
    \ y. abss y = s' ∧ y ∈ set-pmf (action cfg) \implies 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 abst-def t-def)
qed

lemma K-cfg-set-absc:
  assumes cfg ∈ valid-cfg cfg' ∈ K-cfg cfg
shows absc cfg' ∈ K-cfg (absc cfg)
using assms by (auto simp: K-cfg-map-absc)

lemma abst-action-repcs:
  assumes cfg ∈ 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-rep:
  assumes cfg ∈ R-G.valid-cfg
lemma K-elem-abss-inj\':
- assumes $\mu \in K \ s$
- and $s \in S$
- shows inj-on abss (set-pmf $\mu$)
using assms K-elem-abss-inj by (simp add: K-bisim-unique inj-onI)

glemma K-cfg-rept-aux:
- assumes $cfg \in R-G-valid-cfg$ abss $s = state cfg \ x \in$ repl $s$ (action cfg)
- defines $t \equiv \lambda \ cfg'$. THE $s'$. $s' \in$ repl $s$ (action cfg) \& $s' \sim x$
- shows $t \ cfg' = x$
proof -
  from assms have $rept \ s$ (action cfg) $\in K \ s \in S$ 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

glemma K-cfg-rept-action:
- assumes $cfg \in R-G-valid-cfg$ abss $s = state cfg \ cfg' \in$ set-pmf (K-cfg cfg)
- shows $abss \ (THE \ s'$. $s' \in$ repl $s$ (action cfg) \& abss $s' = state cfg') = state cfg'$
proof -
  let $?\mu = repl \ s$ (action cfg)
  from abst-rept-id assms have $action \ cfg = abst \ ?\mu$ by (auto simp: R-G-I)
  moreover from assms have $state \ cfg' \in$ action cfg by (auto simp: set-K-cfg)
  ultimately have $state \ cfg' \in$ abst $?\mu$ by simp
  then obtain $s'$ where $s' \in ?\mu$ $abss \ s' = state \ cfg'$ by (auto simp: abst-def pmf.set-map)
  with K-cfg-rept-aux[OF assms(1,2) this(1)] show ?thesis by auto
qed

glemma K-cfg-map-repcs:
- assumes $cfg \in R-G-valid-cfg$ abss $s = state cfg$
- defines $repc' \equiv \lambda \ cfg$. $repcs \ (THE \ s'$. $s' \in$ repl $s$ (action cfg) \& abss $s' = state cfg') \ cfg')$
- shows $K-cfg$ (repcs $s$ $cfg$) = map-pmf $repc'$ (K-cfg $cfg$)
proof -
  let $?\mu = repl \ s$ (action cfg)
  define $t$ where $t \equiv \lambda \ cfg'$. THE $s$. $s \in ?\mu$ \& abss $s = state \ cfg'$
  have $t (cont \ cfg (abss \ s')) = s'$ if $s' \notin ?\mu$ for $s'$
  using K-cfg-rept-aux[OF assms(1,2) this] 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

glemma K-cfg-map-repc:
- assumes $cfg \in R-G-valid-cfg$
- defines $repc \ cfg' \equiv \ repcs \ (THE \ s$. $s \in$ repl $\ (reps \ (state \ cfg)) \ (action \ cfg) \& abss \ s = state \ cfg') \ cfg'$
  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)

glemma R-G-K-cfg-valid-cfgD:
- assumes $cfg \in R-G-valid-cfg \ cfg' \in K-cfg \ cfg$
- shows $cfg' = cont \ cfg \ (state \ cfg') \ state \ cfg' \in$ action cfg
proof -
  from assms obtain $s$ where $s \in$ action cfg $cfg' = cont \ cfg \ s$ by (auto simp: set-K-cfg)
  with assms show $cfg' = cont \ cfg \ (state \ cfg') \ state \ cfg' \in$ action cfg
    by (auto intro: R-G-valid-cfg-state-in-S R-G-valid-cfgD)
lemma \( \text{K-cfg-valid-cfgD} \):
\[
\text{assumes } \text{cfg} \in \text{valid-cfg} \quad \text{cfg}' \in \text{K-cfg} \quad \text{cfg} \\
\text{shows } \text{cfg}' = \text{cont} \text{cfg} (\text{state} \text{cfg}') \quad \text{state} \text{cfg}' \in \text{action} \text{cfg} \\
\]
\text{proof} –
\[
\text{from } \text{assms}(2) \text{ obtain } s \text{ where } s \in \text{action} \text{cfg} \quad \text{cfg}' = \text{cont} \text{cfg} s \text{ by } (\text{auto } \text{simp: set-K-cfg}) \\
\text{with } \text{assms} \text{ show} \\
\quad \text{cfg}' = \text{cont} \text{cfg} (\text{state} \text{cfg}') \quad \text{state} \text{cfg}' \in \text{action} \text{cfg} \\
\text{by } \text{auto} \\
\]
\text{qed}

lemma \( \text{absc-bisim-abss} \):
\[
\text{assumes } \text{absc} \ x = \text{absc} \ x' \\
\text{shows } \text{state} \ x \sim \text{state} \ x' \\
\]
\text{proof} –
\[
\text{from } \text{assms} \text{ have } \text{state} \ (\text{absc} \ x) = \text{state} \ (\text{absc} \ x') \text{ by } \text{simp} \\
\text{then show } ?\text{thesis} \text{ by } (\text{simp add: state-absc}) \\
\]
\text{qed}

lemma \( \text{K-cfg-bisim-unique} \):
\[
\text{assumes } \text{cfg} \in \text{valid-cfg} \quad \text{and} \quad x \in \text{K-cfg} \quad \text{cfg} \quad x' \in \text{K-cfg} \\
\text{and} \quad \text{state} \ x \sim \text{state} \ x' \\
\text{shows } x = x' \\
\]
\text{proof} –
\[
\text{define } t \text{ where } t \equiv \text{THE} x. \ x' \sim \text{state} \ x \land x' \in \text{set-pmf} \ (\text{action} \text{cfg}) \\
\text{from } \text{K-cfg-valid-cfgD} \text{ assms have } *: \\
\quad x = \text{cont} \text{cfg} (\text{state} \ x) \quad \text{state} \ x \in \text{action} \text{cfg} \\
\quad x' = \text{cont} \text{cfg} (\text{state} \ x') \quad \text{state} \ x' \in \text{action} \text{cfg} \\
\quad \text{by } \text{auto} \\
\text{with } \text{assms have} \\
\quad \text{cfg} \in \text{valid-cfg} \quad \text{abs} \ (\text{state} \ x) \in \text{set-pmf} \ (\text{abs} (\text{action} \text{cfg})) \\
\quad \text{unfolding } \text{abs-def} \text{ by } \text{auto} \\
\text{with } \text{cont-cfg-defined}[\text{of} \text{cfg} \ (\text{state} \ x)] \text{ have} \\
\quad \forall y. y \sim \text{state} \ x \land y \in \text{set-pmf} \ (\text{action} \text{cfg}) \rightarrow y = t \\
\quad \text{unfolding } \text{t-def} \text{ by } \text{auto} \\
\text{with } * \text{ assms have state} x' = t \text{ state} x = t \text{ by } \text{fastforce}+ \\
\text{with } * \text{ show } ?\text{thesis} \text{ by } \text{simp} \\
\]
\text{qed}

lemma \( \text{absc-distr-self} \):
\[
\text{MDP}.\text{MC}.\text{T} \ (\text{absc} \text{cfg}) = \text{distr} \ (\text{MDP}.\text{MC}.\text{T} \text{cfg}) \quad \text{MDP}.\text{MC}.\text{S} \ (\text{smap} \text{absc}) \text{ if } \text{cfg} \in \text{valid-cfg} \\
\text{using } (\text{cfg} \in \sim) \\
\]
\text{proof} \text{ (coinduction arbitrary: cfg rule: MDP.MC.T-coinduct)} \\
\text{case prob} \\
\text{show } ?\text{case} \text{ by } (\text{rule MDP.MC.T.prob-space-distr, simp}) \\
\text{next} \\
\text{case sets} \\
\text{show } ?\text{case} \text{ by } \text{auto} \\
\text{next} \\
\text{case prems: (cont cfg)} \\
\text{define } t \text{ where } t \equiv \lambda y. \text{THE} x. \ y = \text{absc} x \land x \in \text{K-cfg} \text{cfg} \\
\text{define } M' \text{ where } M' \equiv \lambda \text{cfg}. \quad \text{distr} \ (\text{MDP}.\text{MC}.\text{T} \ (t \text{cfg})) \quad \text{MDP}.\text{MC}.\text{S} \ (\text{smap} \text{absc}) \\
\text{show } ?\text{case} \\
\text{proof} \ (\text{rule exI}[\text{where} \ x = M'], \text{safe, goal-cases}) \\
\text{case A: } (t \ y) \\
\text{from } A \text{ prems obtain } x' \text{ where } y = \text{absc} x' x' \in \text{K-cfg} \text{cfg} \text{ by } (\text{auto } \text{simp: K-cfg-map-absc}) \\
\text{with } \text{K-cfg-bisim-unique}[\text{OF} \text{ prems } - - \text{absc-bisim-abss}] \text{ have} \\
\quad y = \text{absc} (t \ y) \ x' = t \ y \\
\quad \text{unfolding } \text{t-def} \text{ by } (\text{auto intro: theI2})
moreover have $x' \in \text{valid-cfg}$ using $\langle x' \in \cdot \rangle$ prems by auto
ultimately show ?case unfolding $M' \text{-def}$ by auto

next
case 5
show ?case unfolding $M' \text{-def}$
  apply (subst distr-distr)
  prefer 3
  apply (subst $\text{MDP.MC.T-eg-bind}$)
  apply (subst distr-bind)
  prefer 4
  apply (subst distr-distr)
  prefer 3
  apply (subst $\text{K-cfg-map-absc}$)
  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 $t (\text{absc } x) = x$ unfolding $t \text{-def}$
proof (rule the-equality, goal-cases)
case 1 with $A$ show ?case by simp
next
case (2 $x'$)
  with $\text{K-cfg-bisim-unique}[OF prems - A \text{-absc-bisim-abss}]$ 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 $\text{MDP.MC.T-subprob-space-distr}$ $\text{MDP.MC.T.prob-space-distr}$
)+
qed (auto simp: $M' \text{-def}$ intro: $\text{MDP.MC.T.prob-space-distr}$)

lemma $\text{R-G-trace-space-distr-eq}$:
assumes $\text{cfg \in R-G.valid-cfg}$ abss $s = \text{state cfg}$
shows $\text{MDP.MC.T \ text{cfg} = distr (MDP.MC.T (repcs s \text{cfg}))} \text{MDP.MC.S (smap absc)}$
using assms
proof (coinduction arbitrary: $\text{cfg s}$ rule: $\text{MDP.MC.T-coinduct}$)
case prob
  show ?case by (rule $\text{MDP.MC.T.prob-space-distr}$, simp)
next
case sets
  show ?case by auto
next
case prems: (cont $\text{cfg s}$)
  let $?\mu = \text{rept s (action cfg)}$
define $\text{repc'}$ where $\text{repc'} \equiv \lambda \text{cfg' \ s. } s \in ?\mu \land \text{abss } s = \text{state cfg'} \text{ cfg'}$
define $M'$ where $M' \equiv \lambda \text{ cfg'. distr (MDP.MC.T (repc' \text{ cfg}))} \text{MDP.MC.S (smap absc)}$
show ?case
proof (intro exI[where $x = M'$], safe, goal-cases)
case A: (1 $\text{cfg'}$)
  with $\text{K-cfg-rept-action}[OF prems]$ have
  abss $\text{(THE s. s \in ?\mu \land \text{abss } s = \text{state cfg'}) = \text{state cfg'}$ by auto
moreover from A prems have $\text{cfg'} \in R-G.valid-cfg$ by auto
ultimately show ?case unfolding $M' \text{-def}$ $\text{repc'-def}$ by best
next
case 4

show ?case unfolding $M'\text{-def}$ by (rule MDP.MC.T.prob-space-distr, simp)

next

case 5

have *: $\text{smap absc} \circ (##) (\text{repc}' \text{cfg'}) = (##) \text{cfg'} \circ \text{smap absc}$

if $\text{cfg'} \in \text{set-pmf} \ (K\text{-cfg cfg})$ for $\text{cfg'}$

proof —

from $K\text{-cfg-rep-action}[OF \text{prems that}]$ have

$\text{abss} \ (\text{THE s. s} \in ?\mu \land \text{abss s} = \text{state cfg'}) = \text{state cfg'}$

with $\text{prems that have *:}$

$\text{absc (repc' cfg')} = \text{cfg'}$

unfolding $\text{repc'}\text{-def}$ by (subst $\text{absc-repcs-id}$, auto)

then show $(\text{smap absc} \circ (##) (\text{repc'} \text{cfg'})) = ((##) \text{cfg'} \circ \text{smap absc})$ by auto

qed

from $\text{prems show ?case unfolding } M'\text{-def}$

apply (subt $\text{distr-distr}$)

apply simp+

apply (subt MDP.MC.T.eq-bind)

apply (subt $\text{distr-bind}$)

prefer 2

apply simp

apply (rule MDP.MC.T.distr-Stream-subprob)

apply simp

apply (subt $\text{distr-distr}$)

apply simp+

apply (subt $K\text{-cfg-map-repcs}[OF \text{prems}]$)

apply (subt $\text{map-pmf-rep-eq}$)

apply (subt $\text{bind-distr}$)

by (fastforce simp: * [unfolded $\text{repc}'\text{-def}$ $\text{repc'}\text{-space-subprob-algebra}$ MC-syntax.in-S

intro: $\text{bind-measure-pmf-cong}$ MDP.MC.T.subprob-space-distr)+

qed (simp add: $M'\text{-def}$)+

qed

lemma $\text{repc-inj-on-K-cfg}$:

assumes $\text{cfg} \in R\text{-G.cfg-on s s} \in S$

shows $\text{inj-on repc (set-pmf (K-cfg cfg))}$

using $\text{assms}$

by (intro inj-on-inverseI [where $g = \text{absc}$], subst $\text{absc-repc-id}$)

(auto intro: R-G.valid-cfgD R-G.valid-cfgI R-G.valid-cfg-state-in-S)

lemma $\text{smap-absc-iff}$:

assumes $\bigwedge x \ y. \ x \in X \implies \text{smap abss x} = \text{smap abss y} \implies y \in X$

shows $(\text{smap state xs} \in X) = ((\text{smap ($\lambda z. \text{abss (state z)}$)} \text{xs} \in \text{smap abss ' X})$

proof (safe, goal-cases)

case 1

then show ?case unfolding $\text{image-def}$

by clarify (inst-existentials $\text{smap state xs}$, auto simp: $\text{stream.map-comp}$)

next

case prems: (2 $\text{xs}'$)

have $(\text{smap ($\lambda z. \text{abss (state z)}$)} \text{xs} = \text{smap abss (smap state xs)}$

by (auto simp: $\text{comp-def stream.map-comp}$)

with $\text{prems}$ have $\text{smap abss (smap state xs)} = \text{smap abss xs'}$ by simp

with $\text{prems(2)}$ $\text{assms show ?case by auto}$

qed

lemma $\text{valid-abss-reps[simp]}$:

assumes $\text{cfg} \in R\text{-G.valid-cfg}$

shows $\text{abss (reps (state cfg))} = \text{state cfg}$

using $\text{assms}$ by (subt $\text{S-abss-reps}$) (auto intro: R-G.valid-cfg-state-in-S)
proof

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

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

lemma S-reps-S[intro]:
\[ \text{reps} \ s \in S \Longrightarrow s \in S \]
using S-reps-S-aux by (metis surj_pair)

lemma absc-valid-cfg-eq:
\[ \text{absc} \ ' \ \text{valid-cfg} = \text{R-G.valid-cfg} \]
apply safe
subgoal
by auto
subgoal for cfg
using absc-reps-id[where \( s = \text{reps} \ (\text{state} \ \text{cfg}) \)]
by (frule reps-valid[where \( s = \text{reps} \ (\text{state} \ \text{cfg}) \); force intro: image1])
done

lemma action-repcs:
\[ \text{action} \ (\text{repcs} \ (l, u) \ \text{cfg}) = \text{rept} \ (l, u) \ (\text{action} \ \text{cfg}) \]
by (simp add: reps-def)

5.3 Equalities Between Measures of Trace Spaces

lemma path-measure-eq-absc1-new:
fixes \( \text{cfg} \) \( s \)
defines \( \text{cfg}' \equiv \text{absc} \ \text{cfg} \)
assumes valid: \( \text{cfg} \in \text{valid-cfg} \)
assumes \( X \) measurable: \( X \in \text{R-G.St} \ \text{and} \ \text{Y measurable} \): \( Y \in \text{MDP.St} \)
assumes \( P \): \( \text{AE} \ x \ in \ (\text{R-G.T cfg}') \cdot \text{P} \ x \ \text{and} \ \text{Q} \): \( \text{AE} \ x \ in \ (\text{MDP.T cfg}). \ Q \ x \)
assumes \( P' \) measurable: \( \text{Measurable.pred R-G.St P} \)
and \( Q' \) measurable: \( \text{Measurable.pred MDP.St Q} \)
assumes \( X-Y\text{-closed} \): \( \Lambda \ x \ y \ P \ x \Longrightarrow \text{smap abss} \ y = x \Longrightarrow x \in X \Longrightarrow y \in Y \ \land \ Q y \)
assumes \( Y-X\text{-closed} \): \( \Lambda \ x \ y \ Q \ y \Longrightarrow \text{smap abss} \ y = x \Longrightarrow y \in Y \Longrightarrow x \in X \ \land \ P x \)
shows
\[ \text{emeasure} \ (\text{R-G.T cfg}') X = \text{emeasure} \ (\text{MDP.T cfg}) Y \]
proof
have *: \text{stream-all2} \ (\lambda s. (\equiv) \ (\text{absc} \ s)) x y = \text{stream-all2} \ (\equiv) \ (\text{smap absc} x) y \ \text{for} \ x y \)
by simp
have *: \text{stream-all2} \ (\lambda s t. t = \text{absc} s) x y = \text{stream-all2} \ (\equiv) \ (\text{smap absc} \ x) \ \text{for} \ x y
using \text{stream.rel-conversep}(\text{of} \ \lambda s t. t = \text{absc} s]
by (simp add: conversep_iff[abs-def])

from \( P \) have \( \text{emeasure} \ (R-G.T cfg') X = \text{emeasure} \ (R-G.T cfg') \ (x \in X. \ P x) \)
by (auto intro: emeasure-eq-AE)
moreover from \( Q \) have \( \text{emeasure} \ (MDP.T cfg) Y = \text{emeasure} \ (MDP.T cfg) \ (y \in Y. \ Q y) \)
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 = \text{absc} \ \text{and} \ S = \text{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 (rule rel-map-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:

Y X - closed: \( x \mapsto Y - X \) for x y

assumes Y-X-closed:

Y X - closed: \( x \mapsto Y - X \) for x y

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)
lemma region-compatible-suntil1:
assumes (holds (λx. ϕ (reps x)) until holds (λx. ψ (reps x))) (smap abss x)
and pred-stream (λs. ϕ (reps (abss s)) → ϕ s) x
and pred-stream (λs. ψ (reps (abss s)) → ψ s) x
shows (holds ϕ until holds ψ) x using assms
proof (induction smap abss x arbitrary; x rule: until.induct)
case base
then show ?case by (auto intro: until.base simp: stream.pred-set)
next
case step
have
  pred-stream (λs. ϕ (reps (abss s)) → ϕ s) (stl x)
  pred-stream (λs. ψ (reps (abss s)) → 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 until holds ψ) (stl x) by auto
with step.prems step.hyps(1–2) show ?case by (auto intro: until.step simp: stream.pred-set)
qed

lemma region-compatible-suntil2:
assumes (holds ϕ until holds ψ) x
and pred-stream (λs. ϕ s → ϕ (reps (abss s))) x
and pred-stream (λs. ψ s → ψ (reps (abss s))) x
shows (holds (λx. ϕ (reps x)) until holds (λx. ψ (reps x))) (smap abss x) using assms
proof (induction x rule: until.induct)
case (base x)
then show ?case by (auto intro: until.base simp: stream.pred-set)
next
case (step x)
have
  pred-stream (λs. ϕ s → ϕ (reps (abss s))) (stl x)
  pred-stream (λs. ψ s → 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: until.step simp: stream.pred-set)
qed

lemma region-compatible-suntil:
assumes pred-stream (λs. ϕ (reps (abss s)) ↔ ϕ s) x
and pred-stream (λs. ψ (reps (abss s)) ↔ ψ s) x
shows (holds (λx. ϕ (reps x)) until holds (λx. ψ (reps x))) (smap abss x) ↔ (holds ϕ until holds ψ) x using assms
using \texttt{assms region-compatible-suntil1 region-compatible-suntil2 unfolding stream.pred-set by blast}

\begin{verbatim}
lemma \texttt{reps-abss-S:}
  assumes \texttt{reps (abss s) \in S}
  shows \texttt{s \in S}
by (simp add: S-reps-S abss-S \texttt{assms})

lemma measurable-sset[measurable (raw)]:
  assumes \texttt{f[measurable]: f \in N \rightarrow_{M} stream-space M \text{ and} P[measurable]: Measurable.pred M P}
  shows \texttt{Measurable.pred N (\lambda x. \forall s \in sset (f x). P s)}
proof –
  have \texttt{\ast: (\lambda x. \forall s \in sset (f x). P s) = (\lambda x. \forall i. P (f x !! i))}
by (simp add: sset-range)
  show \texttt{\ast\texttt{thesis}}
  unfolding \texttt{* by measurable}
qed

lemma path-measure-eq-repcs''-new:
notes \texttt{in-space-UNIV[measurable]}
fixes \texttt{cfg \varphi \psi s}
defines \texttt{cfg' \equiv \texttt{reps s cfg}}
defines \texttt{\varphi' \equiv absp \varphi \text{ and} \psi' \equiv absp \psi}
assumes \texttt{s: abss s = \text{state cfg}}
assumes \texttt{valid: \texttt{cfg} \in R-G.valid-cfg}
assumes \texttt{valid': \texttt{cfg'} \in \text{valid-cfg}}
assumes \texttt{equiv-\varphi: \bigwedge x. \text{pred-stream (\lambda s. s \in S) x}}
  \implies \text{pred-stream (\lambda s. \varphi (\text{reps (abss s)}) \leftrightarrow \varphi s) (\text{state cfg' ## x})}
  \text{ and} \texttt{equiv-\psi: \bigwedge x. \text{pred-stream (\lambda s. s \in S) x}}
  \implies \text{pred-stream (\lambda s. \psi (\text{reps (abss s)}) \leftrightarrow \psi s) (\text{state cfg' ## x})}
shows
  \texttt{emeasure (R-G.T cfg)} \{\text{x \in space R-G.St. (holds \varphi' \text{ until holds } \psi')} (\text{state cfg' ## x})\} =
  \texttt{emeasure (MDP.T cfg')} \{\text{x \in space MDP.St. (holds \varphi' \text{ until holds } \psi')} (\text{state cfg' ## x})\}
unfolding \texttt{cfg'-def}
apply (rule \texttt{path-measure-eq-repcs1-new[where \texttt{P = pred-stream (\lambda s. s \in S)} \text{ and} \texttt{Q = pred-stream (\lambda s. s \in S)}])}
  apply \texttt{fact}
  apply \texttt{fact}
  apply \texttt{measurable}
subgoal
unfolding \texttt{R-G.T-def}
apply (\texttt{subt AE-distr-iiff})
apply (\texttt{auto; fail})
apply (\texttt{auto simp: \text{stream.pred-set}; fail})
apply (\texttt{rule \texttt{AE-mp[OF MDP.MC.AE-T-enabled AE-I2]})}
using \texttt{R-G.pred-stream-cfg-on[OF valid]} \texttt{by (auto simp: \text{stream.pred-set})}
subgoal
unfolding \texttt{MDP.T-def}
apply (\texttt{subt AE-distr-iiff})
apply (\texttt{auto; fail})
apply (\texttt{auto simp: \text{stream.pred-set}; fail})
apply (\texttt{rule \texttt{AE-mp[OF MDP.MC.AE-T-enabled AE-I2])}
using \texttt{MDP.pred-stream-cfg-on[OF valid', unfolded cfg'-def]} \texttt{by (auto simp: \text{stream.pred-set})}
apply \texttt{measurable}
subgoal premises \texttt{prems for ys xs}
apply \texttt{safe}
apply \texttt{measurable}
unfolding \texttt{\varphi'-def \psi'-def absp-def}
apply (\texttt{subt region-compatible-suntil[\text{symmetric}]})
subgoal
proof –
  from \texttt{prems have \text{pred-stream (\lambda s. s \in S) xs using S-abss-S \texttt{by (auto simp: \text{stream.pred-set})}}}
\end{verbatim}
with equiv-ϕ show ?thesis by (simp add: cfg'-def)
qed

subgoal
proof −
  from prems have pred-stream (λs. s ∈ S) xs using 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: stream.pred-set 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 S-abss-S; measurable; fail)
apply (auto simp: stream.map-comp)
apply (subst asm region-compatible-suntil[symmetric])
subgoal
proof −
  from prems have pred-stream (λs. s ∈ S) xs using S-abss-S by auto
  with equiv-ϕ show ?thesis using valid by (simp add: cfg'-def repc-def)
qed

using valid prems by (auto simp: 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 ⇒ 'a ⇒ bool) ⇒ 'a stream ⇒ bool for P where
  P a b ⇐⇒ pairwise P (b ## xs) ⇐⇒ pairwise P (a ## b ## xs)

lemma pairwise-Suc:
  pairwise P xs ⇒⇒ P (xs !! i) (xs !! (Suc i))
  by (induction i arbitrary: xs) (force elim: pairwise.cases)+

lemma Suc-pairwise:
  ∀ i. P (xs # i) (xs # (Suc i)) ⇒⇒ pairwise P xs
  apply (coinduction arbitrary: xs)
  apply (subst stream.collapse[symmetric])
  apply (rewrite in stl - stream.collapse[symmetric])
  apply (intro ext conj1, rule HOL.refl)
  apply (erule allE[where x = 0]; simp; fail)
  by simp (metis nth.simps(2))

lemma pairwise-iff:
  pairwise P xs ⇐⇒ (∀ i. P (xs # i) (xs # (Suc i)))
  using pairwise-Suc Suc-pairwise by blast
lemma pairwise-stlD:
  \(\text{pairwise } P \; \text{xs} \Rightarrow \text{pairwise } P \; (\text{stl} \; \text{xs})\)
by (auto elim: pairwise.cases)

lemma pairwise-pairD:
  \(\text{pairwise } P \; \text{xs} \Rightarrow P \; (\text{shd} \; \text{xs}) \; (\text{shd} \; (\text{stl} \; \text{xs}))\)
by (auto elim: pairwise.cases)

lemma pairwise-mp:
assumes pairwise \( P \; \text{xs} \) and lift \( \bigwedge x \; y. \; x \in \text{sset} \; \text{xs} \Rightarrow y \in \text{sset} \; \text{xs} \Rightarrow P \; x \; y \Rightarrow Q \; x \; y \)
shows pairwise \( Q \; \text{xs} \) using \( \text{assms} \)
proof (coinduction arbitrary: \( \text{xs} \))
subgoal for \( \text{xs} \)
apply (inst-existentials shd (\text{shd} \; \text{xs}) \; \text{shd} \; (\text{stl} \; \text{xs}))
apply (auto dest: pairwise-Suc)
apply (metis sdrop-simps(1) 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 \( P \; (\text{sdrop} \; i \; \text{xs}) \) if pairwise \( P \; \text{xs} \)
proof (coinduction arbitrary: \( i \); \( \text{xs} \))
case pairwise \( i \; \text{xs} \)
then show \( ?\text{case} \)
  apply (inst-existentials \( \text{shd} \; (\text{sdrop} \; i \; \text{xs}) \); \( \text{shd} \; (\text{stl} \; (\text{sdrop} \; i \; \text{xs})) \); \( \text{stl} \; (\text{stl} \; (\text{sdrop} \; i \; \text{xs})) \))
subgoal
  by (auto dest: pairwise-Suc)
subgoal
  by (inst-existentials \( i \); \( - \; 1 \) \( \text{stl} \; \text{xs} \))
by (metis sdrop-simps(2) stream.collapse[symmetric])
qed

6.2 Regions

lemma gt-GreaterD:
assumes \( u \in \text{region} \; X \; I \; r \) \( \text{valid-region} \; X \; k \; I \; r \) \( c \in \text{X} \) \( u \; c \; > \; k \; c \)
shows \( I \; c = \text{Greater} \; (k \; c) \)
proof –
from \( \text{assms} \) have \( \text{intv-elem} \; c \; u \; (I \; c) \) \( \text{valid-intv} \; (k \; c) \; (I \; c) \) by auto
with \( \text{assms}(4) \) show \( ?\text{thesis} \) by (cases \( I \; c \)) auto
qed

lemma const-ConstD:
assumes \( u \in \text{region} \; X \; I \; r \) \( \text{valid-region} \; X \; k \; I \; r \) \( c \in \text{X} \) \( u \; c = \; d \; d \; \leq \; k \; c \)
shows \( I \; c = \text{Const} \; d \)
proof –
from \( \text{assms} \) have \( \text{intv-elem} \; c \; u \; (I \; c) \) \( \text{valid-intv} \; (k \; c) \; (I \; c) \) by auto
with \( \text{assms}(4,5) \) show \( ?\text{thesis} \) by (cases \( I \; c \)) auto
qed

lemma not-Greater-bounded:
assumes \( I \; x \neq \text{Greater} \; (k \; x) \) \( x \in \text{X} \) \( \text{valid-region} \; X \; k \; I \; r \) \( u \in \text{region} \; X \; I \; r \)
shows \( u \; x \leq k \; x \)
proof –
from \( \text{assms} \) have \( \text{intv-elem} \; x \; u \; (I \; x) \) \( \text{valid-intv} \; (k \; x) \; (I \; x) \) by auto
with \( \text{assms}(1) \) show \( u \; x \leq k \; x \) by (cases \( I \; x \)) auto
qed

lemma Greater-closed:
fixes \( t :: \text{real} \)
assumes \( u \in \text{region } X I r \) \( \text{valid-region } X k I r c \in X I c = \text{Greater } (k c) t > k c \)
shows \( u(c := t) \in \text{region } X I r \)
using \text{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** \text{Greater-unbounded-aux}:
assumes \( \text{finite } X \) \( \text{valid-region } X k I r c \in X I c = \text{Greater } (k c) \)
shows \( \exists u \in \text{region } X I r. u c > t \)
using \text{assms} \text{Greater-closed}[OF - assms(2-4)]
proof –
let \( ?R = \text{region } X I r \)
let \( ?t = \text{if } t > k c \text{ then } t + 1 \text{ else } k c + 1 \)
have \( t \in \text{region-not-empty}[OF assms(1,2)] \) obtain \( u \) where \( u : u \in ?R \) by auto
from \text{Greater-closed}[OF this assms(2-4)] \have \( u(c := ?t) \in ?R \) by auto
with \( t \) show \( \text{thesis} \) by (inst-existentials \( u(c := ?t) \)) auto
qed

6.3 Unbounded and Zero Regions

**definition** \text{unbounded } X R \equiv \forall t. \exists u \in R. u x > t

definition \text{zero } X R \equiv \forall u \in R. u x = 0

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

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

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

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

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

48
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 φ ψ :: ('s * (c, t) cval) ⇒ bool 
assumes φ: ∀ x y. x ∈ S ⇒ x ~ y ⇒ φ x ↔ φ y 
assumes ψ: ∀ x y. x ∈ S ⇒ x ~ y ⇒ ψ x ↔ ψ y 
assumes s: S ⇒ s ∈ S
begin

definition φ' ≡ absp φ 
definition ψ' ≡ absp ψ 
definition s' ≡ abss s

lemma s-s'cfg-on[intro]: 
assumes cfg ∈ MDP.cfg-on s 
shows absc cfg ∈ R-G.cfg-on s'
proof - 
from assms s have cfg ∈ valid-cfg unfolding MDP.valid-cfg-def by auto
then have absc cfg ∈ R-G.cfg-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'-S[simp, intro]: 
s' ∈ S 
unfolding s'-def using s by auto

lemma s'-s-cfg-on[intro]: 
assumes cfg ∈ R-G.cfg-on s' 
shows reps s cfg ∈ MDP.cfg-on s
proof - 
from assms s have cfg ∈ R-G.valid-cfg unfolding R-G.valid-cfg-def by auto
with assms have reps s cfg ∈ valid-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: ∀ x y. x ∈ S ⇒ x ~ y ⇒ φ x ↔ φ y 
assumes pred-stream (λs. s ∈ S) xs 
and [intro]: x ∈ S 
shows pred-stream (λs. φ (reps (abss s)) = φ s) (x ## xs)
unfolding stream.pred-set proof clarify 
fix l u 
assume A: (l, u) ∈ sset (x ## xs) 
from assms have pred-stream (λs. s ∈ S) (x ## xs) by auto
with A have (l, u) ∈ S by (fastforce simp: stream.pred-set)
then have abss (l, u) ∈ S by auto
then have reps (abss (l, u)) ~ (l, u) by simp
with φ ⟨(l, u) ∈ S show φ (reps (abss (l, u))) = φ (l, u) by blast
qed

lemma φ-stream': 
pred-stream (λs. φ (reps (abss s)) = φ s) (x ## xs) if pred-stream (λs. s ∈ S) xs x ∈ S 
using compatible-stream[of φ, OF φ that].
lemma $\psi$-stream':

\[ \text{pred-stream } (\lambda x. \psi (\text{reps (abss s)}) = \psi s) (x \#\# xs) \text{ if } \text{pred-stream } (\lambda s. s \in S) xs x \in S \]

using compatible-stream[of $\psi$, OF $\psi$ that].

lemmas $\varphi$-stream = compatible-stream[of $\varphi$, OF $\varphi$]
lemmas $\psi$-stream = compatible-stream[of $\psi$, OF $\psi$]

7.2 Easier Result on All Configurations

lemma $\text{suntil-reps}$:

assumes
\[ \forall s \in \text{sset } (\text{smap abss y}, s \in S) \]
\[ (\text{holds } \varphi' \text{ until holds } \psi') \]
\[ (s' \#\# \text{smap abss y}) \]
shows
\[ (\text{holds } \varphi \text{ until holds } \psi) \]
\[ (s \#\# y) \]
using assms
by (subst region-compatible-$\text{suntil}$[symmetric]; (intro $\varphi$-stream $\psi$-stream)?)
(auto simp: $\varphi$'-def $\psi$'-def absp-def stream.predecessors $\text{suntil}$ holds $s'$-def comp-def)

lemma $\text{suntil-abss}$:

assumes
\[ \forall s \in \text{sset } y, s \in S \]
\[ (\text{holds } \varphi \text{ until holds } \psi) \]
\[ (s \#\# y) \]
shows
\[ (\text{holds } \varphi' \text{ until holds } \psi') \]
\[ (s' \#\# \text{smap abss y}) \]
using assms
by (subst (asm) region-compatible-$\text{suntil}$[symmetric]; (intro $\varphi$-stream $\psi$-stream)?)
(auto simp: $\varphi$'-def $\psi$'-def absp-def stream.predecessors $\text{suntil}$ holds $s'$-def comp-def)

theorem $\text{P-sup}$-$\text{suntil}$-eq:

notes [measurable] = in-space-UNIV and [iff] = pred-stream-iff
shows
\[ (\text{MDP}.P\text{-sup } s \ (\lambda x. (\text{holds } \varphi \text{ until holds } \psi) \ (s \#\# x))) \]
\[ = (\text{R-G}.P\text{-sup } s' \ (\lambda x. (\text{holds } \varphi' \text{ until holds } \psi') \ (s' \#\# x))) \]

unfolding $\text{MDP}$.$\text{P-sup}$-def $\text{R-G}$.$\text{P-sup}$-def

proof (rule SUP-eq, goal-cases)
case prems: (1 $\text{cfg}$)
let $?\text{cfg}' = \text{absc cfg}$
from prems have $\text{cfg} \in \text{valid-cfg}$ by (auto intro: MDP.valid-cfgI)
then have $?\text{cfg}' \in \text{R-G.valid-cfg}$ by (auto intro: R-G.valid-cfgI)
from $\text{cfg} \in \text{valid-cfg}$ have alw-$S$: almost-everywhere (MDP.$\text{T}$ $\text{cfg}$) (pred-stream (\lambda s. s \in S))
by (rule MDP.alw-$S$)
from $?\text{cfg}' \in \text{R-G.valid-cfg}$ have alw-$S$: almost-everywhere (R-G.$\text{T}$ $?\text{cfg}'$) (pred-stream (\lambda s. s \in S))
by (rule R-G.alw-$S$)

have emeasure (MDP.$\text{T}$ $\text{cfg}$) $\{ x \in \text{space MDP.St. (holds } \varphi \text{ until holds } \psi) \ (s \#\# x) \}$
= emeasure (R-G.$\text{T}$ $?\text{cfg}'$) $\{ x \in \text{space R-G.St. (holds } \varphi' \text{ until holds } \psi') \ (s' \#\# x) \}$
apply (rule path-measure-eq-absc1-new[symmetric, where $P = \text{pred-stream } (\lambda s. s \in S)$]
and $Q = \text{pred-stream } (\lambda s. s \in S)$]
)
using prems alw-$S$ alw-$S$ apply (auto intro: MDP.valid-cfgI simp: ![7]
by (auto simp: S-abss-$S$ intro: S-abss-$S$ intro!: until-abss until-reps, measurable)
with prems show $\text{?case by (inst-existentials } $?\text{cfg}'$) auto
next
case prems: (2 $\text{cfg}$)
let $?\text{cfg}' = \text{reps s } \text{cfg}$
have $s = \text{state } $?\text{cfg}'$ by simp
from prems have $s' = \text{state } \text{cfg}$ by auto
have pred-stream (\lambda s. $\varphi$ (reps (abss s)) = $\varphi$ s) (state (reps s $\text{cfg}$) #\# x)
if pred-stream (\lambda s. s \in S) x for $x$
using prems that by (intro $\varphi$-stream) auto
moreover
have pred-stream (\(\lambda s. \psi (\text{reps (abss s)})\)) = \(\psi s\) (state (repcs s cfg) \#\# x)
if pred-stream (\(\lambda s. s \in S\)) x for x
using prems that by (intro \(\psi\)-stream) auto

ultimately
have emeasure (R-G.T cfg) \{x \in space R-G.St. (\text{holds \(\varphi'\) until holds \(\psi'\)} \(s'\) \#\# x)\}
= emeasure (MDP.T (repcs s cfg)) \{x \in space MDP.St. (\text{holds \(\varphi\) until holds \(\psi\)} \(s\) \#\# x)\}
apply (rewrite in s \#\# - \(\langle s = - \rangle\))
apply (subst \(\langle s' = - \rangle\))
unfolding \(\varphi'\)-def \(\psi'\)-def s'-def
apply (rule path-measure-eq-repcs''-new)
using prems by (auto \#3 simp: s'-def intro: \(\varphi\)-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 \(\text{elapsed } u u' \equiv \text{Max} (\{ u' c - u c \mid c. c \in X \} \cup \{0\})\)
definition eq-elapsed \(u u' \equiv \text{elapsed } u u' > 0 \rightarrow (\forall c. c \in X. u' c - u c = \text{elapsed } u u')\)
definition dur :: \((t.\times) \text{ coal stream } \Rightarrow \text{nat } \Rightarrow t\) where
dur - 0 = 0 |
dur (x \#\# y \#\# xs) (Suc i) = elapsed x y + dur (y \#\# xs) i
definition \(\text{divergent } \omega \equiv \forall t. \exists n. \text{dur } \omega n > t\)
definition \(\text{div-cfg } cfg \equiv \text{AE } \omega \text{ in } MDP.MC.T \text{ cfg. divergent (smap (snd o state) } \omega)\)
definition \(\text{R-div } \omega \equiv \forall x. (\forall i. (\exists j \geq i. \text{zero } x (\omega !! j)) \land (\exists j \geq i. \neg \text{zero } x (\omega !! j)))\)
\land (\exists i. \forall j \geq i. \text{unbounded } x (\omega !! j))
definition \(\text{R-G-div-cfg } cfg \equiv \text{AE } \omega \text{ in } MDP.MC.T \text{ cfg. } \text{R-div (smap (snd o state) } \omega)\)
end

class Probabilistic-Timed-Automaton-Regions
begin

definition \(\text{cfg-on-div st } \equiv \text{MDP.cfg-on st } \cap \{ \text{cfg. div-cfg cfg}\}\)
definition \(\text{R-G-cfg-on-div st } \equiv \text{R-G.cfg-on div st } \cap \{ \text{cfg. R-G-div-cfg cfg}\}\)

lemma measurable-\(\text{R-div}\)|measurable]: Measurable.pred MDP.MC.S \(\text{R-div}\)
unfolding \(\text{R-div-def}\)
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:
\[ \text{dur } xs \ i \geq 0 \]

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 \text{dur-mono}:
\( i \leq j \Rightarrow \text{dur } xs \ i \leq \text{dur } xs \ j \)

proof (induction \( i \) arbitrary: \( xs \) \( j \))
  case \( 0 \)
  show ?case by (auto intro: dur-pos)

  next
  case \( \text{Suc} \ i \ xs \ j \)
  obtain \( x \ y \ ys \) where \( xs: xs = x ## y ## ys \) using stream.collapse by metis
  from \( \text{Suc} \) obtain \( j' \) where \( j' = \text{Suc} j' \) by (cases \( j' \)) auto
  with \( xs \) have \( \text{dur } xs \ j = \text{elapsed } x \ y + \text{dur } (y ## ys) \ j' \) by auto
  also from \( \text{Suc} \ j' \) have \( \ldots \geq \text{elapsed } x \ y + \text{dur } (y ## ys) \ i \) by auto
  also have \( \text{elapsed } x \ y + \text{dur } (y ## ys) \ i = \text{dur } xs (\text{Suc } i) \) by (simp add: \( xs \))
  finally show ?case.

qed

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

lemma \text{elapsed-0D}:
assumes \( c \in X \) \( \text{elapsed } u \ u' \leq 0 \)
shows \( u' c - u c \leq 0 \)

proof
  from assms have \( u' c - u c \in \{u' c - u c \mid c. c \in X\} \cup \{0\} \) by auto
  with \( \text{finite}(1) \) have \( u' c - u c \leq \text{Max } (\{u' c - u c \mid c. c \in X\} \cup \{0\}) \) by auto
  with \( \text{assms}(2) \) show ?thesis unfolding elapsed-def by auto

qed

lemma \text{elapsed-ge}:
assumes \( \text{eq-elapsed } u \ u' \ c \in X \)
shows \( \text{elapsed } u \ u' \geq u' c - u c \)
using assms unfolding eq-elapsed-def by (auto intro: elapsed-ge0 order.trans[OF elapsed-0D])

lemma \text{elapsed-eq}:
assumes \( \text{eq-elapsed } u \ u' \ c \in X \) \( u' c - u c \geq 0 \)
shows \( \text{elapsed } u \ u' = u' c - u c \)
using elapsed-ge[OF assms(1,2)] assms unfolding eq-elapsed-def by auto

lemma \text{dur-shift}:
\( \text{dur } \omega (i + j) = \text{dur } \omega i + \text{dur } (\text{sdrop } i \omega) j \)
apply (induction \( i \) arbitrary: \( \omega \))
apply simp

subgoal for \( i \) \( \omega \)
apply simp
apply (subst stream.collapse[symmetric])
apply (rewrite at stl \( \omega \) stream.collapse[symmetric])
apply (subst dur.simps)
apply (rewrite in \( \text{dur } \omega \) stream.collapse[symmetric])
apply (rewrite in \( \text{dur } (- ## \ -) \) (Suc -) stream.collapse[symmetric])
apply (subst dur.simps)
apply simp

52
lemma dur-zero:
  assumes
  \( \forall i. \text{xs}!!i \in \omega \land i \forall j \leq i. \text{zero} \ x (\omega!!j) \ x \in X \)
  \( \forall i. \text{eq-elapsed} \ (\text{xs}!!i) \ (\text{xs}!!\text{Suc} \ i) \)
  shows \( \text{dur} \ \text{xs} \ i = 0 \) using assms
proof (induction \( i \) arbitrary: \( \text{xs} \omega \))
case 0
  then show \(?case\) by simp
next
case (Suc \( i \) \( \text{xs} \omega \))
  let \(?x\) = \( \text{xs}!!0 \)
  let \(?y\) = \( \text{xs}!!1 \)
  let \(?ys\) = \( \text{stl} \ (\text{stl} \ \text{xs}) \)
  have \( \text{xs}: \text{xs} = ?x \#?y \#?ys \) by auto
  from Suc.IH[OF this[1,2] \( \text{this}(3) \) have \( \text{simpl} \)] dur \( \text{xs} \ i \) = 0 by auto
  then have \(?y x - ?x x\) = 0 unfolding zero-def by force+
  finally show \(?thesis\).
qed

lemma dur-zero-tail:
  assumes \( \forall i. \text{xs}!!i \in \omega \land \forall k \geq i. \text{ks} \ x (\omega!!k) \ x \in X \)
  \( \forall i. \text{eq-elapsed} \ (\text{xs}!!i) \ (\text{xs}!!\text{Suc} \ i) \)
  shows \( \text{dur} \ \text{xs} \ j = \text{dur} \ \text{xs} \ i \) using assms
proof
  from \( \langle j \geq i \rangle \) dur-shift[of \( \text{xs} \ i \ j \) - \( i \)] have \( \text{dur} \ \text{xs} \ j = \text{dur} \ \text{xs} \ i + \text{dur} \ (\text{sdrop} \ i \ \text{xs}) \ (j - i) \)
  by simp
  also have \( \ldots\) = \( \text{dur} \ \text{xs} \ i \)
    using assms unfolding One-nat-def using Suc.prems[of \( \text{dur} \ (\text{sdrop} \ - \ -) \ - \text{dur-zero} \) \( \text{where} \ \omega = \text{sdrop} \ i \ \omega \)\) auto dest: prop-nth-sdrop-pair[of \( \text{dur} \ \ldots\) \( \text{dur-zero} \) \( \text{where} \ \omega = \text{sdrop} \ i \ \omega \)\]]
  finally show \(?thesis\).
qed

lemma elapsed-ge-pos:
  fixes \( u :: (\text{\# c}, \text{t})\)
  assumes \( \text{eq-elapsed} \ u \ u' \ c \in X \ u \in V \ u' \in V \)
  shows \( \text{elapsed} \ u \ u' \leq u' \ c \)
proof (cases \( \text{elapsed} \ u \ u' = 0 \))
case True
  with assms show \(?thesis\) by (auto simp: V-def)
next
case False
  from \( \langle u \in V \rangle \ (c \in X) \) have \( u \ c \geq 0 \) by (auto simp: V-def)
  from False assms have \( \text{elapsed} \ u \ u' = u' \ c - u \ c \)
unfolding eq-elapsed-def by (auto simp add: less-le)
also from \( u = c \geq 0 \) have \( \ldots \leq u' \) 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 simp
apply simp
finally show \( \diamondsuit \)
  subgoal for i xs
apply (subst stream-collapse[symmetric])
apply (rewrite in stl - stream-collapse[symmetric])
apply (subst dur.simps)
apply simp
apply simp
apply simp

inductive trans where
  succ: \( t \geq 0 \implies u' = u \oplus t \implies \text{trans } u \ u' \)
  reset: set \( l \subseteq X \implies u' = \text{clock-set } l \ u \implies \text{trans } u \ u' \)
  id: \( u = u' \implies \text{trans } u \ u' \)

abbreviation stream-trans ≡ pairwise trans

lemma K-cfg-trans:
  assumes \( \text{cfg} \in \text{MDP.ccfg-on} \ (l, R) \ \text{cfg}' \in K-cfg \ \text{cfg} \ \text{state} \ \text{cfg}' = (l', R') \)
  shows \( \text{trans } R \ R' \)
using assms
apply (simp add: set-K-cfg)
apply (drule MDP.ccfg-on-D-action)
apply (cases rule: K.cases)
  apply (auto intro: K.cases)
using admisible-targets-clocks(2) by (blast intro: trans.intros(2))

lemma enabled-stream-trans:
  assumes \( \text{cfg} \in \text{valid-cfg} \ \text{MDP.MC.enabled} \ \text{cfg} \ xs \)
  shows \( \text{stream-trans } \text{smap} \ (\text{snd } o \ \text{state}) \ xs \)
using assms
proof (coinduction arbitrary: cfg xs)
  case prems: (pairwise cfg xs)
  let \( \exists \text{x} = \text{stl } (\text{stl } x) \) let \( \exists x = \text{shd } x \) let \( \exists y = \text{shd } (\text{stl } x) \)
  from MDP.pred-stream-cfg-on[OF prems] have *:
    pred-stream \( (\lambda \text{cfg}. \ \text{state } \text{cfg} \in S \land \text{cfg} \in \text{MDP.ccfg-on } (\text{state } \text{cfg}) ) \ xs \) .
  obtain \( l \ R' \ R' \) where eq: \( \text{state } \exists x = (l, R) \) state \( \exists y = (l', R') \) by force
  moreover from * have \( \exists x \in \text{MDP.ccfg-on } (\text{state } ?x ) \exists x \in \text{valid-cfg} \)
    by (auto intro: MDP.valid-cfg simp: stream.pred-set)
  moreover from prems(2) have \( ?y \in K-cfg \ ?x \) by (auto elim: MDP.MC.enabled.cases)
  ultimately have \( \text{trans } R' \)
    by (intro K-cfg-trans(where \( \text{cfg} = ?x \) and \( \text{cfg}' = ?y \) and \( l = l \) and \( l' = l' \) ) metis+)
with \( ?x \in \text{valid-cfg} \) 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 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 (λ u. u ∈ 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 ∀ k>i, k ≤ j → ¬ zero c ([xs !! k]R) j ≥ i
  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 *:
  ¬ zero c ([xs !! Suc j]R) eq-elapsed (xs !! j) (xs !! Suc j)
  if Suc j > i
  using that by auto
from Suc.prems(6)
  have xs !! j ∈ V
  by blast+
then have regions: [xs !! j]R ∈ R [xs !! Suc j]R ∈ 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 (reset l)
  show ?thesis
  proof (cases c ∈ set l)
    case False
    with prems show ?thesis
    by auto

55
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 \( \langle c \in \mathcal{X} \rangle \) elapsed-eq have
  \( \star \): elapsed \( (xs !! Suc \ j) \ (xs !! Suc \ j) = (xs !! Suc \ j) \ c - (xs !! j) \ c \)
  if Suc \( \ j \triangleright i \)
  using that by blast
show ?case
proof (cases \( i = Suc \ j \))
  case False
  with Suc have \( \quad \)
  dur \( xs \ (Suc \ j) \) = dur \( xs \ i \) = dur \( xs \ (Suc \ j) \) - dur \( xs \ j \) + \( xs !! j \) \ c - \( xs !! i \) \ c \)
  by auto
also have \( \ldots = elapsed \ (xs !! Suc \ j) \ (xs !! Suc \ j) + (xs !! j) \ c - (xs !! i) \ c \)
  by (simp add: dur-Suc)
also have
  \( \ldots = (xs !! Suc \ j) \ c - (xs !! j) \ c + (xs !! j) \ c - (xs !! i) \ c \)
  using * False Suc.prems by auto
also have \( \ldots = (xs !! Suc \ j) \ c - (xs !! i) \ c \) by simp
finally show ?thesis by auto
next
  case True
then show ?thesis by simp
qed

lemma not-reset-dur':
  assumes \( \forall j \geq i . \ 
eg \ \text{zero} \ c \ (\[xs !! j\]_{\mathcal{R}}) \ j \geq i \ c \in \mathcal{X} \) stream-trans \( xs \)
  \( \forall i . \ \text{eq-elapsed} \ (xs !! i) \ (xs !! Suc \ i) \ \forall j. \ xs !! j \in V \)
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 \( \forall j \geq i . \ 
eg \ \text{zero} \ c \ (\[xs !! j\]_{\mathcal{R}}) \ j \geq i \ c \in \mathcal{X} \) stream-trans \( xs \)
  \( \forall i . \ \text{eq-elapsed} \ (xs !! i) \ (xs !! Suc \ i) \ \forall j. \ xs !! j \in V \)
  \( \text{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 \( ?u \in ?R \) by auto
from assms(6) have \( ?R \in \mathcal{R} \) by auto
then obtain \( \ i \ \text{where} \ ?R = \text{region} \ \mathcal{X} \ I \ r \ \text{valid-region} \ \mathcal{X} \ k \ I \ r \ \text{unfolding} \ \mathcal{R}\)-def by auto
with assms(3,7) unbounded-Greater \( ?u \in ?R \) have \( ?u \ c > k \ c \) by force
also from not-reset-dur'[OF assms(1-6)] dur-mono[OF \( \forall j \geq 0, \ \text{of} \ xs \)] have \( ?u' \ c \geq ?u \ c \) by auto
finally have \( ?u' \ c > k \ c \) by auto
let \( ?R' = \[xs !! j\]_{\mathcal{R}} \)
from assms have \( ?u' \in ?R' \) by auto
from assms(6) have \( ?R' \in \mathcal{R} \) by auto
then obtain \( \ i \ \text{where} \ ?R' = \text{region} \ \mathcal{X} \ I \ r \ \text{valid-region} \ \mathcal{X} \ k \ I \ r \ \text{unfolding} \ \mathcal{R}\)-def by auto
moreover with \( ?u' \ c > k \ c \) and GreaterD \( c \in \mathcal{X} \) have \( I \ c = \text{Greater} \ (k \ c) \) by auto
ultimately show ?thesis using Greater-unbounded[OF finite(1) - \( c \in \mathcal{X} \)] by auto
qed
lemma gt-unboundedD:
  assumes u ∈ R
  and R ∈ ℜ
  and c ∈ ℥
  and real (k c) < u c.
  shows unbounded c R.
proof −
  from assms obtain I r where R = region ℥ I r valid-region ℥ k I r
  unfolding ℜ-def by auto.
  with Greater-unbounded[of ℥ k I r c] gt-GreaterD[of c ℥ I r k c] assms(1) show ?thesis by auto.
qed.

definition trans' :: (('c, t) cval ⇒ ('c, t) cval ⇒ bool) where
trans' u u' ≡
  ((∀ c ∈ ℥. u c > k c ∧ u' c > k c ∧ u c ⊕ u' c) −→ u' = u ⊕ 0.5) ∧
  ((∃ c ∈ ℥. u c = 0 ∧ u' c > 0 ∧ (∀ c ∈ ℥. ∃ d. d ≤ k c ∧ u' c = real d))
  −→ u' = delayedR ([u]⊂R) u).

lemma zeroI:
  assumes c ∈ ℥ u ∈ V u c = 0.
  shows zero c ([u]⊂R).
proof −
  from assms have u ∈ [u]⊂R [u]⊂R ∈ ℜ by auto.
  then obtain I r where [u]⊂R = region ℥ I r valid-region ℥ k I r unfolding ℜ-def by auto.
  with zero-all[OF finite(1) this(2)] (c ∈ ℥) (u ∈ [u]⊂R) (u c = 0) show ?thesis by auto.
qed.

lemma zeroD:
  u x = 0 if zero x ([u]⊂R) u ∈ V
  using that by (metis regions-part-ex(1) zero-def).

lemma not-zeroD:
  assumes ¬ zero x ([u]⊂R) u ∈ V x ∈ ℥.
  shows u x > 0.
proof −
  from zeroI assms have u x ≠ 0 by auto.
  moreover from assms have u x ≥ 0 unfolding V-def by auto.
  ultimately show ?thesis by auto.
qed.

lemma not-const-intv:
  assumes u ∈ V ∀ c ∈ ℥. ∃ d. d ≤ k c ∧ u c = real d.
  shows ∀ c ∈ ℥. ∀ u ∈ [u]⊂R. ∃ d. d ≤ k c ∧ u c = real d.
proof −
  from assms have u ∈ [u]⊂R [u]⊂R ∈ ℜ by auto.
  then obtain I r where I: [u]⊂R = region ℥ I r valid-region ℥ k I r unfolding ℜ-def by auto.
  have ∃ d. d ≤ k c ∧ u' c = real d if c ∈ ℥ u' ∈ [u]⊂R for c u'.
  proof safe
    fix d assume A: d ≤ 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 ∈ [u]⊂R) (c ∈ ℥) assms(2) by (cases; fastforce).
  qed.
  then show ?thesis by auto.
qed.
lemma $K$-cfg-trans':

assumes \( \text{repcs} \ (l, u) \ \text{cfg} \in MDP.\text{cfg-on} \ (l, u) \ \text{cfg}' \in K$-cfg (\( \text{repcs} \ (l, u) \ \text{cfg} \))

state $\text{cfg}' = (l', u') \ (l, u) \in S \ \text{cfg} \in R-G.\text{valid-cfg} \ \text{abss} \ (l, u) = \text{state} \ \text{cfg}$

shows $\text{trans}^{'} \ u \ u'$

using \( \text{assms} \)

apply (simp add: set-$K$-cfg)

apply (drule MDP.cfg-onD-action)

apply (cases rule: $K$.cases)

apply assumption

proof goal-cases

case prems: \( (l \ \text{l} \ u \ t) \)

from \( \text{prems} \) \( (-\ (l, u)) \) have \( \text{repcs} \ (l, u) \ \text{cfg} \in \text{valid-cfg} \) by (auto intro: MDP.valid-cfgI)

then have \( \text{absc} \ (\text{repcs} \ (l, u) \ \text{cfg}) \in R-G.\text{valid-cfg} \) by auto

from \( \text{prems} \) have \( *: \ \text{rept} \ (l, u) \ (\text{action} \ \text{cfg}) = \text{return-pmf} \ (l, u \ \oplus t) \) unfolding \( \text{repcs-def} \) by auto

from \( \text{prems} \) \( \) \( -\ (l, u) \) \( (\text{cfg} \in R-G.\text{valid-cfg}) \)

have \( \text{action} \ \text{cfg} \in K \ (\text{abss} \ (l, u)) \)

by (auto dest: R-G-I)

from \( \text{abst-rept-id}[\text{OF} \ \text{this}] \) \( * \) have \( \text{action} \ \text{cfg} = \text{abst} \ (\text{return-pmf} \ (l, u \ \oplus t)) \) by auto

with \( \text{prems} \) have \( **:*: \ \text{action} \ \text{cfg} = \text{return-pmf} \ (l, \ [u \ \oplus t]_R) \) unfolding \( \text{abst-def} \) by auto

show \( ?\text{thesis} \)

proof (cases \( \ \forall \ c \in X. \ u \ c > k \ c) \)

case True

from \( \text{prems} \) have \( \ u \ \oplus t \in [u]_R \) by (auto intro: upper-right-closed[\text{OF} \ True])

with \( \text{prems} \) have \( \ [u \ \oplus t]_R = [u]_R \) by (auto dest: alpha-interp.region-unique-spec)

with \( \text{prems} \) have \( \text{action} \ \text{cfg} = \text{return-pmf} \ (l, [u]_R) \) by simp

with \( \text{prems} \) have \( \text{rept} \ (l, u) \ (\text{action} \ \text{cfg}) = \text{return-pmf} \ (l, u \ \oplus 0.5) \)

unfolding \( \text{rept-def} \) using \( \text{prems} \) by auto

with \( * \) have \( u \ \oplus t = u \ \oplus 0.5 \) by auto

moreover from \( \text{prems} \) have \( u' = u \ \oplus t \) by auto

moreover from \( \text{prems} \) True have \( \forall \ c \in X. \ u' \ c > k \ c \) by (auto simp: cval-add-def)

ultimately show \( ?\text{thesis} \)

using \( \text{True} (-\ (l, u)) \) unfolding \( \text{trans}^{'}\text{-def} \) by auto

next

case F: False

show \( ?\text{thesis} \)

proof (cases \( \exists c \in X. \ u \ c = 0 \ \land 0 < u' \ c \ \land (\forall c \in X. \ \exists d. \ d \leq k \ c \land u' \ c = \text{real} \ d) \))

case True

from \( \text{prems} \) have \( u' \in [u]_R \) by auto

from \( \text{prems} \) have \( [u \ \oplus t]_R \in \text{Succ} \ R ([u]_R) \) by auto

from \( \text{True} \) obtain \( c \) where \( c \in X. \ u \ c = 0 \ u' \ c > 0 \) by auto

with \( \text{zero \ prems} \) have \( \text{zero} \ c ([u]_R) \) by auto

moreover from \( u' \in -\ (u' \ c > 0) \) have \( \neg \text{zero} \ c ([u]_R) \) unfolding \( \text{zero-def} \) by fastforce

ultimately have \( [u \ \oplus t]_R \neq [u]_R \) using \( \text{prems} \) by auto

moreover from \( \text{True not-const-intu \ prems} \) have \( \forall u \in [u \ \oplus t]_R. \forall c \in X. \ \exists d. \ d \leq k \ c \land u \ c = \text{real} \ d \)

by auto

ultimately have \( \exists R'. \ (l, u) \in S \land \)

\( \text{action} \ \text{cfg} = \text{return-pmf} \ (l, R') \land \)

\( R' \in \text{Succ} \ R ([u]_R) \land [u]_R \neq R' \land (\forall u \in R'. \forall c \in X. \ \exists d. \ d \leq k \ c \land u \ c = \text{real} \ d) \)

apply \( \)

apply (rule exI[where \( x = [u \ \oplus t]_R \)])

apply safe

using \( \text{prems} \) ** by auto

then have \( \text{rept} \ (l, u) \ (\text{action} \ \text{cfg}) \)

= \( \text{return-pmf} \ (l, \text{delayedR} (\text{SOME} R'. \ \text{action} \ \text{cfg} = \text{return-pmf} \ (l, R')) u) \)

unfolding \( \text{rept-def} \) by auto

with \( ** \text{ prems} \) have \( u' = \text{delayedR} ([u \ \oplus t]_R) u \) by auto

with \( F \) \( \text{True} \) \( \text{prems} \) show \( ?\text{thesis} \)

unfolding \( \text{trans}^{'}\text{-def} \) by auto

next

case False

with \( F (-\ (l, u)) \) show \( ?\text{thesis} \)

unfolding \( \text{trans}^{'}\text{-def} \) by auto
proof

lemma enabled-repcsI
qed

next
case prems: (2 - τ µ)
then obtain X where X: u' = ([X := 0]u) (X, l') ∈ set-pmf µ by auto
from (- ∈ S) have u ∈ V by auto
let ?r = SOME r. set r = X
show ?case
proof (cases X = { })
  case True
  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) (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) (slt xs) ⇒ shd xs = repsc 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
cfgs 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-l)
  from 'cfg' ∈ - have
    'cfg' = cont cfg (state cfg) state cfg' ∈ action cfg
    by (auto simp: set-K-cfg)
  with abst-rep-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-repcsI1:
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)
proof

using assms

enabled-stream-trans

repcs-eq-rept

⟨
  with trans
ultimately have

from K-cfg-rept-action

let

moreover have

from K-cfg-rept-action

moreover from

have

by auto

let

moreover from

have

by auto

moreover from

ultimately show

using (?x = _) by (inst-existentials xs ?st ?x st cfg) fastforce

qed


lemma replcs-eq-rept:

rept st (action cfg) = rept st" (action cfg"") if replcs st cfg = replcs st" cfg"

by (metis (mono-tags, lifting) action-cfg-corec old.prod.case replcs-def that)

lemma enabled-stream-trans:

assumes cfg ∈ R-G.valid-cfg abss st = state cfg

shows pairwise trans" (smap (snd o state) st) 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 = replcs (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 ?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_M.C.enabled (shd xs) (stl xs) by cases

ultimately show ?case

using (?x = _) by (inst-existentials xs ?st ?x st cfg) fastforce

done

then obtain st" cfg" where

enabled-repcs (shd xs) (stl xs) shd xs = replcs 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: replcs-eq-rept)

done

then obtain st" cfg" where

enabled-repcs (shd ?xs) (stl ?xs)

shd ?xs = replcs st" cfg" st" ∈ rept st' (action cfg') abss st" = state cfg'

by atomize-elim (subst (asm)enabled-repcs.simps, fastforce dest: replcs-eq-rept)

let ?x = shd xs let ?y = shd (stl xs)

let ?cfg = replcs 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 u' 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_M.C.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_M.C.enabled.cases)}
lemma divergent-R-divergent:
  assumes in-S: pred-stream \((\lambda \ u. \ u \in V) \ \text{xs}\)
  and \(\text{div}: \text{divergent} \ \text{xs}\)
  and \(\text{trans}: \text{stream-trans} \ \text{xs}\)
  shows \(\mathcal{R}\)-div \((\text{snap} \ (\lambda u. \ [u]_{\mathcal{R}}) \ \text{xs}) \ (\text{is} \ \mathcal{R}\text{-div} \ ?w)\)
unfolding \(\mathcal{R}\)-div-def proof (safe, simp-all)
fix \(x\)
assume \(x: x \in X\) and bounded: \(\forall i. \ \exists j \geq i. \ \neg \ \text{unbounded} \ (\text{xs} !! j)_{\mathcal{R}}\)
from in-S have \(\text{xs} = \omega: \ \forall i. \ \exists j \geq i \ \in \ ?\omega \ : \ i \ by\ (\text{auto simp: stream.pred-set})\)
from trans in-S have elapsed:
  \(\forall i. \ \eq\text{-elapsed} (\text{xs} !! i) \ (\text{xs} !! \text{Suc} \ i)\)
  by (fastforce intro: pairwise-trans-eq-elapsed pairwise-Suc[where \(P = \text{eq\-elapsed}\])
\{ assume \(A: \ \forall j \geq i. \ \neg \ \text{zero} \ (\text{xs} !! j)_{\mathcal{R}}\)
let \(\ ?t = \text{dur} \ \text{xs} \ i + k \ x\)
from div obtain \(j\) where \(j: \ \text{dur} \ \text{xs} \ j > \text{dur} \ \text{xs} \ i + k \ x\) unfolding divergent-def by auto
then have \(k \ x < \text{dur} \ \text{xs} \ j - \text{dur} \ \text{xs} \ i \ by\ auto\)
also with not-reset-dur"[OF \(A\ \text{less-imp-le}[\text{OF dur-monoD}, \ \text{of xs}] \ (x \in X)\) \ \text{assms elapsed have}
  \(\ldots = (\text{xs} !! j) \ x - (\text{xs} !! i) \ x\)
  by (auto simp: stream.pred-set)
also have \(\ldots \leq (\text{xs} !! j) \ x\)
  using assms(1) \((x \in X)\) unfolding V-def by (auto simp: stream.pred-set)
finally have \(\text{unbounded} \ (\text{xs} !! j)_{\mathcal{R}}\)
  using assms \((x \in X)\) by (intro gt-unboundedD) (auto simp: stream.pred-set)
moreover from dur-monoD[\(\text{of xs} \ i \ j\) \ \(A\) \ \text{have} \ \forall j' \geq j. \ \neg \ \text{zero} \ (\text{xs} !! j')_{\mathcal{R}}\) by auto
ultimately have \(\forall i \geq j. \ \text{unbounded} \ (\text{xs} !! i)_{\mathcal{R}}\)
  using elapsed assms \(x\) by (auto intro: not-reset-unbounded simp: stream.pred-set)
  with bounded have False by auto\}
then show \(\exists j \geq i. \ \neg \ \text{zero} \ (\text{xs} !! j)_{\mathcal{R}}\) by auto
\{ assume \(A: \ \forall j \geq i. \ \neg \ \text{zero} \ (\text{xs} !! j)_{\mathcal{R}}\)
from div obtain \(j\) where \(j: \ \text{dur} \ \text{xs} \ j > \text{dur} \ \text{xs} \ i \ unfolding\ divergent-def\ by\ auto\)
then have \(j \geq i\) by (auto dest: dur-monoD)
from \(A\) have \(\forall j \geq i. \ \text{zero} \ (\text{xs} !! j)\) by auto
  with dur-zero-tail[OF \(\text{xs} = \omega\) - \(x \ (i \leq j)\) elapsed] \(j\) have False by simp\}
then show \(\exists j \geq i. \ \neg \ \text{zero} \ (\text{xs} !! j)_{\mathcal{R}}\) by auto
qed

lemma (in \(-\))
fixes \(f: \ \text{nat \Rightarrow real}\)
assumes \(\forall i. \ \exists j \geq i. \ f \ i \geq 0 \ \forall i. \ \exists j \geq i. \ f \ j > d \ d > 0\)
shows \(\exists n. \ (\sum i \leq n. \ f \ i) > t\)
oops

lemma dur-ev-exceedsI:
  assumes \(\forall i. \ \exists j \geq i. \ \text{dur} \ \text{xs} \ j - \text{dur} \ \text{xs} \ i \geq d \ \text{and} \ d > 0\)
  obtains \(i\) where \(\text{dur} \ \text{xs} \ i > t\)
proof -
  have base: \(\exists i. \ \text{dur} \ \text{xs} \ i > t\) if \(t < d \ \text{for} \ t\)
  proof -
  from assms obtain \(j\) where \(\text{dur} \ \text{xs} \ j - \text{dur} \ \text{xs} \ 0 \geq d\) by fastforce
  with dur-pos[of \(\text{xs} \ 0\)] have \(\text{dur} \ \text{xs} \ j \geq d\) by simp
  with \(d > 0\) \(t < d\) show \(?thesis\) by - (rule exI[where \(x = j\); auto])
  qed
  have base2: \(\exists i. \ \text{dur} \ \text{xs} \ i > t\) if \(t < d \ \text{for} \ t\)
  proof (cases \(t = d\))
  case False
  with \(t \leq d\) base show \(?thesis\) by simp
next
  case True
  from base \langle d > 0 \rangle obtain i where dur xs i > 0 by auto
  moreover from assms obtain j where dur xs j - dur xs i \geq d by auto
  ultimately have dur xs j > d by auto
  with \langle t = d \rangle show \langle ?thesis \rangle by auto
qed
show \langle ?thesis \rangle
proof \langle cases t \geq 0 \rangle
  case False
  with dur-pos have dur xs 0 > t by auto
  then show \langle ?thesis \rangle by (fastforce intro: that)
next
  case True
  let \langle m = nat \lfloor t / d \rfloor \rangle
  from True have \exists i. dur xs i > \langle m \ast d \rangle
  proof \langle induction \langle m \rangle arbitrary: t \rangle
    case 0
    with base[OF \langle 0 < d \rangle] show \langle case by simp \rangle
next
  case \langle Suc n t \rangle
  let \langle t = t - d \rangle
  show \langle ?case \rangle
  proof \langle cases t \geq d \rangle
    case True
    have \langle t / d \rangle = t / d - 1

  proof
    have t / d + - 1 \ast ((t + - 1 \ast d) / d) + - 1 \ast (d / d) = 0
      by \langle simp add: diff-divide-distrib \rangle
    then have t / d + - 1 \ast ((t + - 1 \ast d) / d) = 1
      using assms(2) by fastforce
    then show \langle ?thesis \rangle by algebra
  qed
then have \langle \lfloor t / d \rfloor \rangle = \lfloor t / d \rfloor - 1 by simp
  with \langle Suc n = \rangle have n = nat \lfloor t / d \rfloor by simp
  with Suc \langle t \geq d \rangle obtain i where nat \lfloor t / d \rfloor \ast d < dur xs i by fastforce
from assms obtain j where dur xs j - dur xs i \geq d j \geq i by auto
  with dur xs i > \rangle have nat \lfloor t / d \rfloor \ast d + d < dur xs j by simp
  with True have dur xs j > nat \lfloor t / d \rfloor \ast d
    by \langle metis Suc.hyps(2) \langle n = nat \lfloor (t - d) / d \rfloor \rangle add.commute distrib-left mult.commute
        mult.right-neutral of-nat-Suc \rangle
  then show \langle ?thesis \rangle by blast
next
  case False
  with \langle t \geq 0 \rangle \langle d > 0 \rangle have nat \lfloor t / d \rfloor \leq I by simp
  then have nat \lfloor t / d \rfloor \ast d \leq d
    by \langle metis One-nat-def \langle Suc n = \rangle Suc.leI add.right-neutral le-antisym mult.commute
        mult.right-neutral of-nat-0 of-nat-Suc order-refl zero-less-Suc \rangle
  with base2 show \langle ?thesis \rangle by auto
qed
qed
then obtain i where dur xs i > \langle m \ast d \rangle by atomize-elim
  moreover from \langle t \geq 0 \rangle \langle d > 0 \rangle have \langle m \ast d \geq t \rangle
    using pos-divide-le-eq real-nat-ceiling-ge by blast
  ultimately show \langle ?thesis \rangle using \langle that[of i] \rangle by simp
qed
qed
lemma not-reset-mono:
assumes stream-trans xs sdrop k xs ≥ sdrop k cs2 stream-all (λ u. u ∈ V) xs cs2 ∈ 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 = sdrop k xs
let ?y = sdrop k xs
show ?thesis
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
  case 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-sld f stl-sset)
ultimately show ?thesis
  using prems by auto
qed (use prems in (auto intro: UNTIL-base))
qed auto

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 sdrop (sdrop k xs) c1 ≥ sdrop (sdrop k xs) c2 by auto
from not-reset-mono[OF - this] assms have
  (holds (λ u. u c2 ≤ u c1) until holds (λ u. 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 $u \cdot x > k \cdot x$

proof 

from assms obtain $I \cdot r$ where $R: R = \text{region } I \cdot r \, \text{valid-region } k \cdot l \cdot r$ unfolding $R$-def by auto

with unbounded-Greater ($x \in X$) assms(3) have $I \cdot x = \text{Greater } (k \cdot x)$ by simp

with ($u \in R$) $R \cdot (x \in X)$ show $\neg \text{thesis}$ by force

qed

lemma trans-not-delay-mono:

$u' \cdot c \leq u \cdot c$ if trans $u \cdot u' \cdot u \in V \cdot x \in X \cdot u' \cdot x = 0 \cdot c \in X$

using (trans $u \cdot u'$)

proof (cases)

case (reset $l$)

with that show $\neg \text{thesis}$ by (cases $c \in \text{set } l$) (auto simp: $V$-def)

qed (use that in (auto simp: $\text{cval}_\cdot \text{add-def } V$-def add-nonneg-eq-0-iff))

lemma dur-reset:

assumes pairwise eq-elapsed $xs$ pred-stream ($\lambda \cdot u. u \in V$) $xs$ zero $x$ ($[xs] \cdot ([xs] \cdot R)$ $x \in X$

shows $\text{dur } xs \cdot (Suc i) \cdot \text{dur } xs \cdot i = 0$

proof 

from assms(2) have $\text{in-V } xs \cdot ! Suc i \in V$

unfolding $\text{stream}$-pred-set by auto (metis $\text{snth}$-simps(2) $\text{snth}$-ssset)

with elapsed-ge-pos[of $xs$] $i$ $xs$ !! Suc i $x$ pairwise-Suc[OF assms(1)] assms(2--) have elapsed ($xs$ !!) ($xs$ !! Suc i) $x$ \leq ($xs$ !! Suc i) $x$

unfolding $\text{stream}$-pred-set by auto

with $\text{in-V}$ assms(3) have elapsed ($xs$ !!) ($xs$ !! Suc i) \leq 0 by (auto simp: $\text{zero}$D)

with elapsed-ge[of $xs$] $i$ $xs$ !! Suc i have elapsed ($xs$ !!) ($xs$ !! Suc i) $x$ = 0

by linarith

then show $\neg \text{thesis}$ by (subst dur-Suc)

qed

lemma resets-mono-0:

assumes pairwise eq-elapsed $xs$ stream-all ($\lambda \cdot u. u \in V$) $xs$ stream-trans $xs$

$\forall j \leq i. \text{zero } x$ ($[xs \cdot j] \cdot R$) $x \in X \cdot c \in X$

shows ($xs$ !!) ($xs$ !!) $c \lor ($xs$ !!) $c = 0$

using assms proof (induction $i$)

case 0

then show $\neg \text{case}$ by auto

next

case ($Suc i$)

from Suc.prems have $*$: ($xs$ !! Suc i) $x = 0$ ($xs$ !!) $x = 0$

by (blast intro: $\text{zero}$D $\text{snth}$-ssset, force intro: $\text{zero}$D $\text{snth}$-ssset)

from pairwise-Suc[OF Suc.prems(3)] have trans ($xs$ !!) ($xs$ !! Suc i) .

then show $\neg \text{case}$

proof cases

case prems: ($\text{succ } t$)

with $*$ have $t = 0$ unfolding $\text{cval}_\cdot \text{add-def}$ by auto

with prems have ($xs$ !! Suc i) $c = ($xs$ !!) $c$ unfolding $\text{cval}_\cdot \text{add-def}$ by auto

with Suc show $\neg \text{thesis}$ by auto

next

case prems: (reset $l$)

then have ($xs$ !! Suc i) $c = 0 \lor ($xs$ !! Suc i) $c = ($xs$ !!) $c$ by (cases $c \in \text{set } l$) auto

with Suc show $\neg \text{thesis}$ by auto

next

case id

with Suc show $\neg \text{thesis}$ by auto

qed

qed

lemma resets-mono:

assumes pairwise eq-elapsed $xs$ pred-stream ($\lambda \cdot u. u \in V$) $xs$ stream-trans $xs$

$\forall k \geq i. k \leq j \longrightarrow \text{zero } x$ ($[xs \cdot k] \cdot R$) $x \in X \cdot c \in X$ $i \leq j$
proof
from assms have 1: stream-all (λ u. u ∈ 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:
  ∀ k ≤ j = i. zero x [(sdrop i xs !! k)]_R
by (simp add: Nat.le-diff-cone2 assms(6))
from resets-mono-0 [OF 2 1 3 4 assms(5,6)] (i ≤ 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.P j M ∈ P.M 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 j i where ji: j i ≥ i ∀ P ∈ M. ∃ m > i. j i ≥ m ∧ P m by auto
define k where k Q = (LEAST k. k > j i ∧ 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 ≤ j i ∧ ?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 j i have ∃ m > i. j i ≥ m ∧ ?P m by auto
with (finite ∨) have ?j ∈ {j. i ≤ j ∧ j ≤ j i ∧ ?P j} by - (rule Max-in; auto)
have k: k Q > j i ∧ Q (k Q) if Q ∈ M for Q
proof –
  from assms(1) (Q ∈ M) obtain m where m ≥ Suc j i Q m by auto
  then have m > j i Q m by auto
  then show k Q > j i ∧ 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 \( \neg j1 < m \)
  with \( \forall j < m \) \( \exists j \in \alpha \) have \( i \leq m \leq j1 \) by auto
  with \( \forall P m \) (finite \( \Rightarrow \)) have \( ?j \geq m \) by \( (\text{rule Max-ge}; \text{auto}) \)
  with \( \forall j < m \) show False by simp
qed
with \( \forall P m \) (finite \( \Rightarrow \)) have \( k \leq m \) unfolding k-def by \( (\text{auto intro: Least-le}) \)
with \( \ast (m < \ast k) \) show False by auto
qed
moreover have \( \exists m \leq \ast k \cdot \forall j < m \wedge Q m \) if \( Q \in M \) for \( Q \)
proof
  from \( k[\alpha] (Q \in M) \) have \( k Q > j1 \wedge Q (k Q) \)
  moreover with (finite \( \Rightarrow \)) (Q \( \in M) \) have \( k Q \leq \ast k \) by \( (\text{rule Max-ge}; \text{auto}) \)
  moreover with \( \forall j < \ast k \cdot (k Q \leq \ast k) \) have \( ?j < k Q \) by auto
ultimately show \( \ast \text{thesis} \) by auto
qed
ultimately show
\[ \exists j \geq 1 \exists k > j. \forall P \in M. P j \wedge k \wedge (\forall m < k. j < m \rightarrow \neg P m) \wedge (\forall Q \in M. \exists m \leq k. j < m \wedge Q m) \]
using \( \forall j < \ast k \cdot \forall j < \ast (\forall P \ast k) \ast \text{by} \) \( (\text{inst-existentials} \forall j \ast k \ast \forall P; \text{blast}) \)
qed

lemma \( \mathcal{R}\)-divergent-divergent:
assumes in-S: pred-stream \( (\lambda u. u \in V) \) \( x s \)
and div: \( \mathcal{R}\)-div \( (\lambda u. [u]_{\mathcal{R}}) \) \( x s \)
and trans: stream-trans \( x s \)
and trans': pairwise trans' \( x s \)
and unbounded-not-const:\n\( \forall u. (\forall c \in \mathcal{X}. \text{real} (k c) < u c) \rightarrow \neg ev (\lambda x s. \text{shd} x s = u) \) \( x s \)
shows divergent \( x s \)

unfolding divergent-def proof
fix \( t \)
from pairwise-trans-eq-elapsed[\( OF \) \( \text{trans in-S} \)] have eq-elapsed: pairwise eq-elapsed \( x s \).
define \( X1 \) where \( X1 = \{ x. x \in \mathcal{X} \wedge (\exists i. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}})) \} \)
let \( ?i = \text{Max} \{ \{\text{SOME} i. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}}) \} | x. x \in \mathcal{X} \} \)
from (finite \( \Rightarrow \)) non-empty have
\( ?i \in \{ (\text{SOME} i. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}})) | x. x \in \mathcal{X} \} \)
by \( (\text{intro Max-in}) \) auto
have unbounded \( x ([x] !! j_{\mathcal{R}}) \) if \( x \in X1 j \geq \ast i \) for \( x j \)
proof
  have \( X1 \subseteq \mathcal{X} \) unfolding X1-def by auto
  with (finite \( \Rightarrow \)) non-empty \( \langle x \in X1 \rangle \) have \( \ast \):
  \( ?i \geq (\text{SOME} i. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}})) \) (is \( ?i \geq \ast k \))
  by \( (\text{intro Max-ge}) \) auto
from \( \forall x \in X1 \) have \( \exists k. \forall j \geq k. \text{unbounded} x ([x] !! j_{\mathcal{R}}) \) by \( (\text{auto simp: X1-def}) \)
then have \( \forall j \geq \ast k. \text{unbounded} x ([x] !! j_{\mathcal{R}}) \) by \( (\text{rule somel-cx}) \)
moreover from \( j \geq \ast i \cdot \forall i \geq \ast ?i \) have \( j \geq \ast k \) by auto
ultimately show \( \ast \text{thesis} \) by blast
qed
then obtain \( i \) where unbounded: \( \forall x \in X1. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}}) \)
using (finite \( \Rightarrow \)) auto
show \( \exists n. t < \text{dur} x s n \)
proof (cases \( \forall x \in \mathcal{X}. (\exists i. \forall j \geq i. \text{unbounded} x ([x] !! j_{\mathcal{R}})) \))
case True
  then have \( X1 = \mathcal{X} \) unfolding X1-def by auto
  have \( \exists k j. 0.5 < \text{dur} x s k - \text{dur} x s j \) for \( j \)
  proof
    let \( ?u = x s !! \text{max} i j \)
36
from \textit{in-S} have $?u \in [?u]_\mathcal{R} [?u]_\mathcal{R} \in \mathcal{R}$
by (auto simp: stream.pred-set)

moreover from \textit{unbounded} :$X_1 = \mathcal{X}$ have
\[ \forall x \in \mathcal{X} \text{. \textit{unbounded} } x \in ([?u]_\mathcal{R}) \]
by \textit{force}.

ultimately have \[ \forall x \in \mathcal{X} \text{. \textit{unbounded} } x > k \]
by (auto intro: unbounded-all)

with \textit{unbounded-not-const} have $\neg \text{ev} \ (\textit{alw} \ (\textit{HLD} \ ?u)) \ xs$

unfolding \textit{HLD-iff} by simp

then obtain $r$ where
\[ r \geq \text{max} \ i \ j \ xs \ !\ r \neq \ xs \ !\ Suc \ r \]
apply atomize-elim
apply (simp add: not-ev-iff not-alw-iff)
apply (drule alw-sdrop[where $n = \text{max} \ i \ j$])
apply (drule alwD)
apply (subt (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 $?k = \text{Suc} \ r$
from \textit{in-S} have $\xs !! ?k \in V$ using snth-set unfolding stream.pred-set by blast

with \textit{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 $(r \geq \sim)$ have $r \geq i ?k \geq i \ by \ auto$

with \textit{unbounded} :$X_1 = \mathcal{X}$ have
\[ \forall x \in \mathcal{X} \text{. \textit{unbounded} } x \in ([\xs \ !\ r]_\mathcal{R}) \ \forall x \in \mathcal{X} \text{. \textit{unbounded} } x \in ([\xs \ !\ ?k]_\mathcal{R}) \]
by (auto simp del: snth.simps(2))

with \textit{in-S} have $\forall x \in \mathcal{X} \text{. } (\xs \ !\ r) \ x > k \ x \ \forall x \in \mathcal{X} \text{. } (\xs \ !\ ?k) \ x > k \ x$

using $*$ by (auto intro: unbounded-all)

moreover from \textit{trans'} have $\textit{trans'} \ (\xs \ !\ r) \ (\xs \ !\ ?k)$

using pairwise-Suc by \textit{auto}

ultimately have $(\xs \ !\ ?k) = (\xs \ !\ r) \oplus 0.5$

unfolding \textit{trans'}-def using $(\xs \ !\ r \neq \sim) \ by \ \textit{auto}$

moreover from pairwise-Suc[OF eq-elapsed] have $\text{eq-elapsed} \ (\xs \ !\ r) \ (\xs \ !\ ?k)$

by \textit{auto}

ultimately have $\text{dur} \ xs \ ?k - \text{dur} \ xs \ r = 0.5$

using non-empty by (auto simp: cval-add-def dur-Suc elapsed-eq)

with \textit{dur-monotonic} of $j \ r \ xs$ $(r \geq \text{max} \ i \ j)$ have $\text{dur} \ xs \ ?k - \text{dur} \ xs \ j \geq 0.5$

by \textit{auto}

with $(r \geq \text{max} \ i \ j)$ show $?thesis \ by$ $- (\text{rule exI}[\text{where } x = ?k]; \ \textit{auto})$

qed

next

case \textit{False}

define $X_2 \ where \ X_2 = \mathcal{X} \ - \ X_1$

from \textit{False} have $X_2 \neq \{}$ unfolding X1-def X2-def by \textit{fastforce}

have inf-resets:
\[ \forall i. \exists j \geq i. \ \text{zero} \ x \ ([\xs \ !\ j]_\mathcal{R}) \ \land \ (\exists j \geq i. \neg \ \text{zero} \ x \ ([\xs \ !\ j]_\mathcal{R})) \ \text{if} \ x \in X_2 \ \text{for} \ x \]

using that \textit{div unfolding} $X_1$-def $X_2$-def \ $R$-div-def by \textit{fastforce}

have $\exists j \geq i. \exists k > j. \exists x \in X_2. \ \text{zero} \ x \ ([\xs \ !\ j]_\mathcal{R}) \ \land \ x \in ([\xs \ !\ k]_\mathcal{R})$
\[ \land \ (\forall m. \ j < m \ \land \ m < k \ \rightarrow \neg \ \text{zero} \ x \ ([\xs \ !\ m]_\mathcal{R})) \]
\[ \land \ (\forall x \in X_2. \exists m. \ j < m \ \land \ m \leq k \ \land \ \text{zero} \ x \ ([\xs \ !\ m]_\mathcal{R})) \]
\[ \land \ (\forall x \in X_1. \forall m \geq j. \ \text{unbounded} \ x \ ([\xs \ !\ m]_\mathcal{R}) \ \text{for} \ i \]

proof

from \textit{unbounded} obtain $i'$ where $i' \ \forall x \in X_1. \forall m \geq i' \ text{. \textit{unbounded} } x \ ([\xs \ !\ m]_\mathcal{R}) \ \text{by} \ \textit{auto}$

then obtain $i'$ where $i'$:
\[ i' \geq i \ \forall x \in X_1. \forall m \geq i'. \ \text{unbounded} \ x \ ([\xs \ !\ m]_\mathcal{R}) \]
by (cases $i' \geq i$; \textit{auto})
\[
\text{from } \text{finite}(1) \text{ have } \text{finite } X_2 \text{ unfolding } X_2\text{-def by auto}
\]

\[
\text{with } (X_2 \neq \emptyset) \text{ R-divergent-divergent-aux2 } \begin{cases} M = \{ \lambda i. \text{ zero } x \left( [xs !! i]_R \right) \mid x. \ x \in X_2 \} \end{cases}
\]

\[
\text{inf-resets}
\]

\[
\text{have } \exists i \geq i'. \exists k > j. \exists P \{ \lambda i. \text{ zero } x \left( [xs !! i]_R \right) \mid x. \ x \in X_2 \}. \ P \ j \wedge P \ k
\]

\[
\wedge \left( \forall m. j < m \rightarrow \neg P \ m \right) \wedge \left( \forall Q \{ \lambda i. \text{ zero } x \left( [xs !! i]_R \right) \mid x. \ x \in X_2 \}. \exists m \leq k. \ j < m \wedge Q \ m \right)
\]

\[
\text{by force}
\]

\[
\text{then obtain } j \ k \ x \text{ where}
\]

\[
j \geq i' \ k > j \ x \in X_2 \text{ zero } x \left( [xs !! j]_R \right) \text{ zero } x \left( [xs !! k]_R \right)
\]

\[
\wedge \left( \forall m. j < m \wedge m < k \rightarrow \neg \text{ zero } x \left( [xs !! m]_R \right) \right)
\]

\[
\text{and} \ x \in X_2. \exists m. i < m \wedge m < j \wedge \text{ zero } x \left( [xs !! m]_R \right)
\]

\[
\text{and} \ X_1: \forall x \in X_1. \forall m \geq i. \text{ unbounded } x \left( [xs !! m]_R \right)
\]

\[
\text{for } x \ i \ j
\]

\[
\text{proof –}
\]

\[
\text{have } \exists j' > j. \neg \text{ zero } x \left( [xs !! j']_R \right)
\]

\[
\text{proof –}
\]

\[
\text{from } \text{inf-resets} [OF x \ 1] \text{ obtain } j' \text{ where } j' \geq \text{ Suc } j \rightarrow \neg \text{ zero } x \left( [xs !! j']_R \right) \text{ by auto}
\]

\[
\text{then show } \text{?thesis by – (rule exI[where } x = j'; \text{ auto)}
\]

\[
\text{qed}
\]

\[
\text{from } \text{inf-resets} [OF x \ 1] \text{ obtain } j' \text{ where } j' \geq \text{ Suc } j \rightarrow \neg \text{ zero } x \left( [xs !! j']_R \right) \text{ by auto}
\]

\[
\text{with } \text{nat-eventually-critical-path}[OF x \ 4] \text{ this(2)]}
\]

\[
\text{obtain } j' \text{ where } j' > j \rightarrow \neg \text{ zero } x \left( [xs !! j']_R \right) \forall m \geq j. \ m < j' \rightarrow \neg \text{ zero } x \left( [xs !! m]_R \right)
\]

\[
\text{by auto}
\]

\[
\text{from } (x \in X_2) \text{ have } x \in X \text{ unfolding } X_2\text{-def by simp}
\]

\[
\text{with } (i < j) \text{ not-reset not-reset-dur } \text{ stream-trans } \rightarrow \text{ in-S pairwise-Suc } [OF eq-elapsed] \text{ have}
\]

\[
dur xs (j - 1) \rightarrow \neg \text{ dur } xs = (xs !! (j - 1)) \text{ x} \rightarrow \neg \text{ (xs !! i) \ x} \ (\text{is } ?d1 = ?d2)
\]

\[
\text{by (auto simp: stream-pred-set)}
\]

\[
\text{moreover from } (\text{zero } x \left( [xs !! i]_R \right)) \text{ in-S have } (xs !! i) \ x = 0
\]

\[
\text{by (auto simp: stream-pred-set)}
\]

\[
\text{ultimately have}
\]

\[
dur xs (j - 1) \rightarrow \neg \text{ dur } xs = (xs !! (j - 1)) \ x \ (\text{is } ?d1 = ?d2)
\]

\[
\text{by simp}
\]

\[
\text{show } \text{?thesis}
\]

\[
\text{proof (cases } ?d1 \geq 0.5
\]

\[
\text{case } \text{True}
\]

\[
\text{with } \text{dur mono}[of } j - 1 \ j \ xs \text{ have}
\]

\[
5 \ / 10 \leq \text{ dur } xs \ j \rightarrow \neg \text{ dur } xs \ i
\]

\[
\text{by simp}
\]

\[
\text{then show } \text{?thesis by blast}
\]

\[
\text{next}
\]

\[
\text{case } \text{False}
\]

\[
\text{have } \text{?c-bound: } (xs !! j) \ c \leq ?d2 \text{ if } c \in X_2 \text{ for } c
\]

\[
\text{proof (cases } (xs !! j) \ c = 0
\]

\[
\text{case } \text{True}
\]

\[
\text{from } \text{in-S } (j > \rightarrow ) \text{ True } \ (x \in X) \text{ show } \text{?thesis by (auto simp: V-def stream-pred-set)}
\]

\[
\text{next}
\]

\[
\text{case } \text{False}
\]

\[
\text{from } X_2 (c \in X_2) \text{ in-S have } \exists k > i. \ k \leq j \wedge (xs !! k) \ c = 0
\]

\[
\text{by (force simp: zeroD stream-pred-set)}
\]

\[
\text{with } \text{False have}
\]

\[
\exists k > i. \ k \leq j - \text{ Suc } 0 \wedge (xs !! k) \ c = 0
\]
by (metis Suc-le-eq Suc-pred linorder-neqE-not less not-less-zero)
morerover from that have $c \in \mathcal{X}$ by (auto simp: X2-def)
morerover from not-reset in-S ($x \in \mathcal{X}$) have
$\forall k > i, k \leq j - 1 \rightarrow (xs \# k) \ x \neq \emptyset$
by (auto simp: zeroI stream.pred-set)
ultimately have
$(xs \# (j - 1)) \ c \leq \ ?d2$
using trans in-S $\langle x = 0 \rangle \ (x \in \mathcal{X})$
by (auto intro: R-divergent-divergent-aux that simp: stream.pred-set)
morerover from
trans-not-delay-mono[OF pairwise-Suc[OF trans], of $j - 1$]
$\langle x \in \mathcal{X} \ (c \in \mathcal{X}) \ j > \omega \ \rangle \ \ ($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
morerover from False $\langle ?d1 = ?d2 \rangle$ have $?d2 < 1$ by auto
morerover from in-S have $(xs \# j) \ c \geq 0$ if $c \in \mathcal{X}$ for $c$
using that by (auto simp: V-def stream.pred-set)
ultimately have frac-bound: frac $((xs \# j) \ c) \leq ?d2$ if $c \in X2$ for $c$
using that frac-le-1I by (force simp: X2-def)

let $?u = (xs \# j)$
from in-S have $(xs \# j)_R \in \mathcal{R}$ by (auto simp: stream.pred-set)
then obtain $I \ r$ where region:
$(xs \# j)_R = region \mathcal{X} \ I \ r$ valid-region $\mathcal{X} k I r$
unfolding $\mathcal{R}$-def by auto
let $?S = \{frac \ (?u \ c) \ | \ c \in \mathcal{X} \ \&\ \ \&\ isIntv \ (I \ c)\}$
have $\mathcal{X} - X2: c \in X2$ if $c \in \mathcal{X}$ isIntv $\ (I \ c)$ for $c$
proof
  from $X1$ $j > \omega$ have $\forall x \in X1. \ \text{unbounded} \ x \ (\langle [xs \# j]_R \rangle)$ by auto
  with unbounded-Greater[OF region(2) $\langle c \in \mathcal{X} \rangle$ region(1) that(2)] have $c \notin X1$ by auto
  with $\langle c \in \mathcal{X} \rangle$ show $c \in X2$ unfolding X2-def by auto
qed
have frac-bound: frac $((xs \# j) \ c) \leq \ ?d2$ if $c \in \mathcal{X}$ isIntv $\ (I \ c)$ for $c$
using frac-bound[OF $\mathcal{X}$-X2] that .
have dur $xs \ (j' - 1) = dur \ xs \ j$ using $j' \ (x \in \mathcal{X})$ in-S eq-elapsed
by (subst dur-zero-tail where $\omega = \text{smap} \ (\lambda u. \ ?u)_R \ xs\)$
  (auto dest: pairwise-Suc simp: stream.pred-set)
morerover from dur-reset[OF eq-elapsed in-S, of $x \ j - 1$] $\langle x \in \mathcal{X}; \ \langle x(4) \ j > \omega \ \rangle$ 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
morerover have dur $xs \ j' - dur \ xs \ (j' - 1) \geq (1 - \ ?d2) \ / \ 2$
proof
  from $j' > \omega$ have $j' > 0$ by auto
  with pairwise-Suc[OF trans', of $j' - 1$] have
  trans' $(xs \# (j' - 1)) \ (xs \# j')$
  by auto
morerover from $j'$ have
$(xs \# (j' - 1)) \ x = 0$ $(xs \# j') \ x > 0$
using in-S $\langle x \in \mathcal{X} \rangle$ by (force intro: not-zeroD dest: not-zeroD simp: stream.pred-set) +
morerover note delayedR-aux = calculation
obtain $t$ where
$(xs \# j') = (xs \# (j' - 1)) \oplus t \ t \geq (1 - \ ?d2) \ / \ 2 \ t \geq 0$
proof
  from in-S have $(xs \# j')_R \in \mathcal{R}$ by (auto simp: stream.pred-set)
  then obtain $I' \ r'$ where region':
  $\langle [xs \# j']_R \rangle = region \mathcal{X} \ I' \ r'$ valid-region $\mathcal{X} k I' r'$
  unfolding $\mathcal{R}$-def by auto
  let $?S' = \{frac \ ((xs \# (j' - 1)) \ c) \ | \ c \in \mathcal{X} \ \&\ \ Regions.isIntv \ (I' \ c)\}$
from finite(1) have \( ?d2 \geq \text{Max} \ (?S' \cup \{0\}) \)
apply –
apply (rule Max.boundedI)
apply fastforce
apply fastforce
apply safe
subgoal premises prems for - c d
proof –
from \( j' \) have \( (xs !! (j' - 1)) c = ?u c \lor (xs !! (j' - 1)) c = 0 \)
  by (intro resets-mono \langle OF eq-elapsed in-S trans - \langle \exists x \in X \mid c \in X \rangle \rangle; auto)
then show \(?thesis\)
proof (standard, goal-cases)
case A: 1
  show \(?thesis\)
next
case False
  have \(?thesis\)
    using \(\langle \exists x \in X \rangle\)\ instead of \(\langle \exists x \in X \rangle\)
next
case prems: 2
  have frac \( (0 :: real) = (0 :: real) \) by auto
  then have frac \( (0 :: real) \leq (0 :: real) \) by linarith
  moreover from in-S \( x \in X \) have \( (xs !! (j' - 1)) x \geq 0 \)
    unfolding V-def stream.pred-set by auto
  ultimately show \(?thesis\) using prems by auto
qed

proof
  using in-S \( x \in X \) by (auto simp: V-def stream.pred-set)
then have le: \( (1 - ?d2) / 2 \leq (1 - \text{Max} \ (?S' \cup \{0\})) / 2 \) by simp

let \( ?u = xs !! j' \)
let \( ?u' = xs !! (j' - 1) \)
from in-S have *: \( ?u' \in V \ [?u'] \in \mathcal{R} \ ?u \in V \ [?u] \in \mathcal{R} \)
  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:
  \( (xs !! j') \in \text{Succ} \ \mathcal{R} \ (\ [xs !! (j' - 1)] \in \mathcal{R}) \land (\exists t \geq 0. ?u = ?u' \oplus t) \)
proof cases
  case prems: (succ t)
  from * have \( ?u' \in [?u'] \in \mathcal{R} \) by auto
  with prems * show \(?thesis\) by auto
next
case (reset l)
  with \( ?u' \in V \) have \( ?u x \leq ?u' x \) by (cases x \in set l) (auto simp: V-def)
from \( j' \) have zero \( x \) \( (\ [?u'] \in \mathcal{R}) \) by auto
with (\?u' \in V \ldotp) have \?u' \in X \dashv 0 unfolding zero-def by auto

with (\?u x \leq \cdot \ldotp (\?u x > 0) \dspy thesis by auto

next

case id

with * Succ-ref[\of \cdot R \cdot X \cdot k, folded \cdot R\dashv def, \cdot OF - \cdot finite(1)] \dspy thesis

unfolding cval-add-def by auto

qed

then obtain t \держан where t: \?u = xs \dashv (j' - 1) \oplus t t \geq 0 by auto

note Succ = Succ[\THEN conjunct1]

show \dspy thesis

proof (cases \exists \?u c \in X2 \ldotp \exists \?d \ldotp nat \?u c = \?d)

case True

from True obtain c and \?d \ldotp nat where c:

\?u x > 0 by fact

from pairwise-Suc[\of \cdot OF eq-elapsed, of j' - 1] \dspy j' > j \dspy have

eq-elapsed (xs \dashv (j' - 1)) \?u

by auto

moreover from
eq-elapsed-eq[\of \this (x \in X) \ldotp (xs \dashv (j' - 1)) x = 0 \ldotp (xs \dashv j') x > 0]

have elapsed (xs \dashv (j' - 1)) (xs \dashv j') > 0

by auto

ultimately have

\?u c - (xs \dashv (j' - 1)) c > 0

using \{c \in X\} unfolding eq-elapsed-def by auto

moreover from in-S have xs \dashv (j' - 1) \in V by (auto simp: stream.pred-set)

ultimately have \?u c > 0 using \{c \in X\} unfolding V-def by auto

from region' in-S \{c \in X\} have in-tr-elem c \?u (I' c)

by (force simp: stream.pred-set)

with \{\?u c = \?d\} \ldspy \?u c > 0 \dspy have \?u c \geq 1 by auto

moreover have (xs \dashv (j' - 1)) c \leq 0.5

proof

- have (xs \dashv (j' - 1)) c \leq (xs \dashv j) c

using j'(1,3)

by (auto intro: resets-mono[\of \cdot OF eq-elapsed in-S trans - (x \in X) \ldotp c \in X])

also have \ldots \leq \?d2 using j-c-bound[\of \cdot OF \{c \in X2\}].

also from \{\?d1 = \?d2\} (\vdash 5 / 10 \leq \cdot) have \ldots \leq 0.5 by simp

finally show \dspy thesis .

qed

moreover have \?d2 \geq 0 using in-S \{x \in X\} by (auto simp: V-def stream.pred-set)

ultimately have \?u c - (xs \dashv (j' - 1)) c \geq (1 - \?d2) / 2 by auto

with t have t \geq (1 - \?d2) / 2 unfolding cval-add-def by auto

with t show \dspy thesis by (auto intro: that)

next

case F: False

have not-const: \dashv isConst (I' c) if c \in X for c

proof (rule ccontr, simp)

assume A: isConst (I' c)

show False

proof (cases c \in X1)

case True

with X1 (j' > j) \ldotp (j > \cdot) have unbounded c ([xs \dashv j']_R) by auto

with unbounded-Greater (c \in X) region' have isGreater (I' c) by force

with A show False by auto

next

case False

with (c \in X) have c \in X2 unfolding X2-def by auto

from region' in-S (c \in X) have inv-tr-elem c \?u (I' c)
unfolding stream.pred-set by force
with \( c \in X2 \): A False F show False by auto
qed
qed
have \( \exists x, x \leq k \land (xs !! j') c = \text{real } x \) if \( c \in X \) for \( c \)
proof (cases \( c \in X2 \); safe)
  fix \( d \)
  assume \( c \in X2 \) (xs !! j') \( c = \text{real } d \)
  with \( F \) show False by auto
next
  fix \( d \)
  assume \( c \notin X2 \)
  with that have \( c \in X1 \) unfolding X2-def by auto
  with \( X1 \) \( j' > j \) \( j > i \) have unbounded \( c ([?\nu]_R) \) by auto
  from unbounded-all[OF - - this] \( c \in X \) in-S have \( ?\nu \ c > k \ c \)
    by (force simp: stream.pred-set)
  moreover assume \( ?\nu \ c = \text{real } d \ d \leq k \ c \)
  ultimately show False by auto
qed
with delayedR-aux have
\( (xs !! j') = \text{delayedR} ([xs !! j']_R) (xs !! (j' - 1)) \)
using \( z \in X \): unfolding trans'-def by auto
from not-const region'(1) in-S Suc(1) have
\( \exists t \geq 0. \ delayedR ([xs !! j']_R) (xs !! (j' - 1)) = xs !! (j' - 1) \oplus t \land \)
\( (1 - \text{Max} ([?S' \cup \{0\}]) / 2 \leq t \)
apply simp
apply (rule delayedR-correct[OF - - region'(2), simplified])
by (auto simp: stream.pred-set)
with le (\( \_ = \text{delayedR} \_ \) ) show \( ?\text{thesis} \) by (auto intro: that)
qed
qed
moreover from pairwise-Suc[OF eq-elapsed, of \( j' - 1 \)] \( j' > 0 \) have
\( \text{eq-elapsed} (xs !! (j' - 1)) (xs !! j') \)
by auto
ultimately show \( \text{dur } xs j' - \text{dur } xs (j' - 1) \geq (1 - ?d2) / 2 \)
using \( j' > 0 \) dur-Suc[of \( j' - 1 \)] \( z \in X \) by (auto simp: cva-add-def elapsed-eq)
qed
moreover from dur-mono[of \( i j - 1 \) xs] \( i < j \) have \( \text{dur } xs i \leq \text{dur } xs (j - 1) \) by simp
ultimately have \( \text{dur } xs j' - \text{dur } xs i \geq 0.5 \) unfolding \( ?d1 = ?d2; [\text{symmetric}] \) by auto
then show \( ?\text{thesis} \) using \( j < j' \) by - (rule exI[where \( x = j' \); auto]
qed
moreover
have \( \exists j \geq i. \ \text{dur } xs j' - \text{dur } xs i \geq 0.5 \) for \( i \)
proof -
from calculation(1)[of \( i \)] obtain \( j k x \) where
  \( j \geq k \ j x \in X2 \) zero \( x \) ([xs !! j]_R)
  zero \( x \) ([xs !! k]_R)
  \( \forall m. \ j < m < k \rightarrow \neg \text{zero } x ([xs !! m]_R) \)
  \( \forall x \in X2. \ \exists m > j. \ m < k \land \neg \text{zero } x ([xs !! m]_R) \)
  \( \forall x \in X1. \ \exists m \geq j. \ \text{unbounded } x ([xs !! m]_R) \)
  by auto
from calculation(2)[OF this(3,2,4-8)] obtain \( j' \) where
  \( j' \geq k 5 \ / 10 \leq \text{dur } xs j' - \text{dur } xs j \)
  by auto
with dur-mono[of \( i j x \) xs] \( j \geq i \) \( k > j \) show \( ?\text{thesis} \) by (intro exI[where \( x = j' \); auto]
qed
then show \( ?\text{thesis} \) by - (rule dur-ev-exceedsI[where \( d = 0.5 \); auto]
qed
qed
lemma $\text{cfg-on-div-absc}$:

notes in-space-$\text{UNIV}$ measurable]

assumes $\text{cfg} \in \text{cfg-on-div \ st \ st} \in S$

shows $\text{absc \ cfg} \in R-G-\text{cfg-on-div \ (abss \ \text{st})}$

proof –

from assms have $\ast$: $\text{cfg} \in MDP.\text{cfg-on-div \ st \ state} \ \text{cfg} = \text{st} \ \text{div-cfg} \ \text{cfg}$

unfolding $\text{cfg-on-div-def}$ by auto

with assms have $\text{cfg} \in \text{valid-cfg}$ by (auto intro: $\text{MDP.valid-cfgI}$)

have almost-everywhere ($\text{MDP.MC.T} \ \text{cfg}$) ($\text{MDP.MC.enabled} \ \text{cfg}$)

by (rule $\text{MDP.MC.AE-T-enabled}$)

moreover from $\ast$ have $\text{AE \ x \ in MDP.MC.T} \ \text{cfg, \ divergent} \ (\text{smap} \ (\text{snd} \circ \text{state}) \ x)$

by (simp add: div-cfg-def)

ultimately have $\text{AE \ x \ in MDP.MC.T} \ \text{cfg, \ R-div} \ (\text{smap} \ (\text{snd} \circ \text{state}) \ (\text{map \ absc} \ x))$

proof eventually-elim

case (elim $\omega$)

let $?xs = \text{smap} \ (\text{snd} \circ \text{state}) \ \omega$

from $\text{MDP.pred-stream-cfg-on} \ (\text{OF} \ \omega \ \in \text{valid-cfg} \ \langle \text{MDP.MC.enabled} - \rangle)$ have $\ast$:

pred-stream ($\lambda \ \omega. \ x \in S$) (smap state $\omega$)

by (auto simp: stream.pred-set)

have $\langle \text{snd} \ (\text{state} \ x) \rangle_R = \text{snd} \ (\text{abss} \ (\text{state} \ x))$ if $x \in \text{sset} \ \omega$ for $x$

proof –

from $\ast$ that have $\text{state} \ x \in S$ by (auto simp: stream.pred-set)

then have $\text{snd} \ (\text{abss} \ (\text{state} \ x)) = \langle \text{snd} \ (\text{state} \ x) \rangle_R$ by (metis abss-S snd-conv surj-pair)

then show $\ast$thesis ..

qed

then have $\text{smap} \ (\lambda z. \langle \text{snd} \ (\text{state} \ z) \rangle_R) \ \omega = (\text{smap} \ (\lambda z. \ \text{snd} \ \text{abss} \ (\text{state} \ z))) \ \omega)$ by auto

from $\ast$ have $\text{pred-stream} \ (\lambda u. \ u \in V) \ ?xs$

apply (simp add: map-def stream.pred-set)

apply (subst (asm) surjective-pairing)

using $\text{S-V \ by blast}$

moreover have $\text{stream-trans} \ ?xs$

by (rule enabled-stream-trans ($\omega \ \in \text{valid-cfg} \ \langle \text{MDP.MC.enabled} - \rangle$)+

ultimately show $\ast$case using $\langle \text{divergent} \ - \rangle \ (\text{smap} \ - \ \omega = \ -)$

by – (drule divergent-R-divergent, auto simp add: stream.map-comp state-absc)

qed

with $\langle \text{cfg} \in \text{valid-cfg} \rangle$ have $\text{R-G-div-cfg} \ (\text{abss} \ \text{cfg})$ unfolding $\text{R-G-div-cfg-def}$

by (subgoal absc-distr-self) (auto intro: $\text{MDP.valid-cfgI}$ simp: $\text{AE-distr-iff}$)

with $\text{R-G.valid-cfgD} \ \langle \text{cfg} \in \text{valid-cfg} \rangle$ * show $\ast$thesis unfolding $\text{R-G-cfg-on-div-def}$ by auto force

qed

definition alternating $\text{cfg} = (\text{AE} \ \omega \ \text{in} \ \text{MDP.MC.T} \ \text{cfg}$. $\omega \ \langle \text{ev} \ (\text{HLD} \ \{\text{cfg}. \ \forall \ \text{cfg'} \in K-cfg \ \text{cfg}. \ \text{fst} \ (\text{state} \ \text{cfg'}) = \ \text{fst} \ (\text{state} \ \text{cfg}))\rangle \ \omega)$

lemma $K-cfg$-same-loc-iff:

$(\forall \ \text{cfg} \in K-cfg \ \text{cfg}. \ \text{fst} \ (\text{state} \ \text{cfg'}) = \ \text{fst} \ (\text{state} \ \text{cfg}))$

$\iff (\forall \ \text{cfg} \in K-cfg \ \text{absc} \ \text{cfg}). \ \text{fst} \ (\text{state} \ \text{cfg'}) = \ \text{fst} \ (\text{state} \ (\text{absc} \ \text{cfg}))$

if $\text{cfg} \in \text{valid-cfg}$

using that by (auto simp: state-absc $\text{fst-abss} \ K-cfg\text{-map-absc}$)

lemma $\langle \text{in} \ - \rangle$ $\text{stream-all2-flip}$:

$\text{stream-all2} \ (\lambda a \ b. \ R \ b \ a) \ \text{xs} \ \text{ys} = \text{stream-all2} \ R \ \text{ys} \ \text{xs}$

by (standard; coinduction arbitrary: $\text{xs} \ \text{ys}; \ \text{auto dest: sym}$)

lemma $\text{AE-alw-ev-same-loc-iff}$:

assumes $\text{cfg} \in \text{valid-cfg}$

shows alternating $\text{cfg} \iff \text{alternating} \ (\text{abss} \ \text{cfg})$

unfolding alternating-def

apply (simp add: $\text{MDP.MC.T.AE-iff-emeasure-eq-1}$)

subgoal
proof –
show ?thesis (is (?x = 1 = (?y = 1))

proof –
have *: stream-all2 (\lambda 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 impl1)
apply (frule stream.rel-mono-strong[where Ra = \lambda 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 elim!: alw-ev-cong)

subgoal
by (rule rel-funI) (auto intro!: rel-pmf-reflI simp: pmf.rel-map(2) K-cfg-map-absc)
using \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \langle \angle
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 ⊆ {? 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 (absb (l, u)) absb (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 (absb (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 (absb (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-absb-S)
        moreover have absb (l, u) = (l, [u]G)
            by (metis absb-S calculation(2))
        ultimately show False
            using :rept (l, u) - = - unbounded unfolding rept-def by (auto dest: cvall-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-abs K-cfg-rept[symmetric])
ultimately have \( \text{rept} (l, u) \ (\text{action \( cfg \))} \in \?U u \)
by blast
then have \( \text{measure-pmf} \ (\text{rept} (l, u) \ (\text{action \( cfg \)))} \ {x. \ \text{snd} \ x = u} \leq \?r u \)
by (fastforce intro: Sup-upper)
mOREVER have \( \text{rept} (l, u) \ (\text{action \( cfg \))} = \text{action} \ (\text{repcs} (l, u) \ cfg) \)
by (simp add: repcs-def)
ultimately have \( \text{measure-pmf} \ (\text{action} \ (\text{repcs} (l, u) \ cfg)) \ {x. \ \text{snd} \ x = u} \leq \?r u \)
by auto
}

note \(* = this \)
let \(?S = \{ cfg. \ \exists \ cfg' \ s. \ cfg' \in \text{R-G.valid-cfg} \land cfg = \text{repcs} s \ cfg' \land \text{abs} s = \text{state} cfg' \} \)
have \text{start} have \( \text{repcs} st cfg \in \?S \)
using \(\text{cfg} \in \text{R-G.valid-cfg} \) \text{assms unfolding \text{R-G-cfg-on-div-def}}
by clarsimp \text{inst-existentials \( cfg \ fem st snd st, auto \)}

have \text{step} (\( y \in \?S \) if \( y \in \text{K-cfg} x x \in \?S \) for \( x y \))
using \text{that apply safe}

subgoal \text{for} \( cfg' l u \)
apply \text{inst-existentials \( absc \ fem state g \) s}
subgoal
by blast
subgoal
by (metis \( \text{K-cfg-valid-cfgD} \text{R-G.valid-cfgD} \text{R-G.valid-cfg-state-in-S absc-repcs-id cont-absc-1} \)
\text{cont-repcs1 \repcs-valid} )

subgoal
by (simp add: \text{state-absc})
done
done
have \(* = x \in \?S \) if \( \text{repcs} st \ cfg, x) \in \text{MDP.MC.acc for} \( x \)
proof -
from \( \text{MDP.MC.acc-relfunD[OF that]} \text{obtain} \ n \text{ where} ((\lambda a b. b \in \text{K-cfg} a) ^ n) \ (\text{repcs} st \ cfg) x \).
then show \( \text{?thesis} \)
proof (induction \( n \) \text{arbitrary:} \( x \) )
next
case \( 0 \)
with \text{start} show \( \text{?case} \)
by simp
next
case \( \text{Suc} \ n \)
from this(2)[simplified] show \( \text{?case} \)
apply (rule relcomppE)
apply (erule step)
apply (erule Suc.IH)
done
qed

qed
have \(* = \text{almost-everywhere} \ (\text{MDP.MC.T} \ (\text{repcs} st \ cfg)) \) \text{(alw} \ (\text{HLD} \ ?S)) \)
by (rule AE-mp[OF \text{MDP.MC.AE-T-reachable}]) \( \text{(fastforce dest: \(* = \text{simp}: \text{HLD-iff elim: alw-mono}) \)

from \text{alternating cfg} \text{assms have alternating} \ (\text{repcs} st \ cfg)
by (simp add: AE-alw-ev-same-loc-iff[of - st])
then have \text{alw-ev-same2: \text{almost-everywhere} \ (MDP.MC.T \ (\text{repcs} st \ cfg)})
(alw \ (\lambda \omega. \text{HLD} \ (\text{state} \ \omega \ \text{snd} \ \omega) \ \text{of} \ u) \omega \rightarrow
ev \ (\text{HLD} \ \{ cfg. \ \forall \ cfg' \in \text{set-pmf (K-cfg \ cfg)} \text{. \text{fsto} (state \ cfg') = \text{fsto} (state \ cfg)} \} \omega))
for \ u \ unfolding \text{alternating-def} \text{by (auto elim: alw-mono) \)

let \(?X = \{ \text{cfg} :: \ ('s \times ('c \Rightarrow \text{real}) \text{.} \forall \ c \in X. \ \text{snd} \ \text{(state} \ \text{cfg}) \ c > k c \})
let \(?Y = \{ \text{cfg.} \ \forall \ \text{cfg' \in K-cfg \ cfg). \text{fsto} \ (\text{state} \ \text{cfg'}) = \text{fsto} \ (\text{state} \ \text{cfg}) \})

have \( \text{(AE} \ \omega \ \text{in} \ ?M. \ ?P \ \omega) \leftrightarrow \)
(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 impl notI, goal-cases)
  case (1 ω u)
  then show ?case
  by − (intro alw-HLD-smap alw-disjoint-contr[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)

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-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
lemmas

proof

qed

lemma cfg-on-div-repcs-strong:

notes in-space-UNIV[measurable]

assumes \( \text{cfg} \in R-G\)-cfg-on-div (\( \text{absst} \)) \( \text{st} \in S \) and alternating \text{cfg}

shows \( \text{reps} \text{t} \text{cfg} \in \text{cfg}-\text{on-div} \text{st} \)

proof

\[ * \]

moreover from \( * \) have \( \text{AE} \ \omega \ \text{in} MDP.MC.T \ \text{cfg} \ \forall \ \text{u} \ (\forall \ \text{c} \in \text{c} \text{.} \ \text{real} \ (k \ c) < u \ c) \ \rightarrow \ \\
\text{valid-cfgI} \ \text{valid-cfg} \ \text{valid-cfgD}
\]

by (rule \text{reps}-\text{unbounded-} AE-\text{non-loop}-end-strong)

qed
lemma repes-unbounded-AE-non-loop-end:
assumes \( \text{cfg} \in R \text-G \cdot \text{cfg-on} \ (\text{abss} \ s) \ \text{st} \in S \)
shows \( \text{AE} \ \omega \text{in MDP} \cdot \text{MC} \cdot T \ (\text{repcs} \ s \ \text{cfg}) \).
\( (\forall \ s :: (s \times (c \Rightarrow \text{real})). (\forall \ c \in X. \ \text{snd} \ s \ c > k \ c) \rightarrow \neg (\text{ev} \ (\text{ahw} \ (\lambda \ x s. \ \text{snd} \ x s = s)) \ (\text{map} \ \text{state} \ \omega)) \) (is \( \text{AE} \ \omega \text{in} \ ?M. \ ?P \ omega \))

proof –
from assms have \( \text{cfg} \in R \cdot \text-G \cdot \text{valid-cfg} \)
by (auto intro: R \cdot \text-G \cdot \text{valid-cfgI})
with assms(1) have repes \( \text{st} \ \text{cfg} \in \text{valid-cfg} \)
by auto
from R \cdot \text-G \cdot \text{valid-cfgD} \{ \text{OF} \ \text{cfg} \in R \cdot \text-G \cdot \text{valid-cfg} \} have \( \text{cfg} \in R \cdot \text-G \cdot \text{cfg-on} \ (\text{state} \ \text{cfg}) \).
let \( ?r = \lambda \ \text{x}. \ \{ \mu \in K. \ \mu \neq \text{return-pmf} \ s \} \)
let \( ?r = \lambda \ x. \ \text{Sup} \ ((\lambda \ \mu. \ \text{measure-pmf} \ \mu \ \{ \text{x} \}) \ ?K \ \text{x}) \)
have lt-1: \( \forall \ \text{x} < l \ \text{if} \ \mu \in ?K \ \text{x} \)
proof –
have \( \ast: \text{emeasure} \ (\text{measure-pmf} \ \mu) \ {\{ \text{x} \}} < 1 \ \text{if} \ \mu \neq \text{return-pmf} \ s \ \text{for} \ \mu \)
proof (rule ccontr)
  assume \( \neg (\text{emeasure} \ (\text{measure-pmf} \ \mu) \ {\{ \text{x} \}} < 1) \)
  then have \( \text{emeasure} \ (\text{measure-pmf} \ \mu) \ {\{ \text{x} \}} = 1 \)
  using \( \text{measure-pmf} \cdot \text{measure-ge-1-iff} \) by force
  with that show False
  by (simp add: measure-pmf-eq-1-iff)
qed
let \( ?S = \{ \text{map} \ \text{pmf} \ (\lambda \ (X, l). \ (l, ((X := 0) \text{u}))) \ \mu \ | \ \mu \ l \ u \ g. \}
\text{x} = (l, \ u) \land (l, \ g, \ \mu) \in \text{trans-of} \ \text{A} \)
have \( (\lambda \ \mu. \ \text{measure-pmf} \ \mu \ \{ \text{x} \}) \ ?K \ \text{x} \)
\( \subseteq \{0, 1\} \cup \{ (\lambda \ \mu. \ \text{measure-pmf} \ \mu \ \{ \text{x} \}) \ ?S \}
\) by (force elim:: K.cases)
moreover have finite \( ?S \)
proof –
  have \( ?S \subseteq \{ (\lambda \ (l, \ g, \ \mu). \ \text{map} \ \text{pmf} \ (\lambda \ (X, l). \ (l, ((X := 0) \text{snd} \ x))) \ \mu) \ ? \text{trans-of} \ \text{A} \}
  \) by force
  also from finite(\( ?S \)) have finite \( \ldots \ldots \)
  finally show \( \ast \text{thesis} \).
qed
ultimately have finite \( (\lambda \ \mu. \ \text{measure-pmf} \ \mu \ \{ \text{x} \}) \ ?K \ \text{x} \)
by (auto intro: finite-subset)
then show \( \ast \text{thesis} \)
  using that by (auto intro: \* finite-imp-Sup-less)
qed

\{ \text{fix} \ s :: (s \times (c \Rightarrow \text{real})) \ \text{and} \ \text{cfg} :: (s \times (c \Rightarrow \text{real}) \ \text{set}) \ \text{cfg} \}
assume unbounded: \( \forall \ c \in X. \ \text{snd} \ s \ c > k \ c \) \text{and} \ \text{cfg} \in R \cdot \text-G \cdot \text{cfg-on} \ (\text{abss} \ s) \ \text{abss} \ s \in S
then have repes \( \text{s} \ \text{cfg} \in \text{valid-cfg} \)
by (auto intro: R \cdot \text-G \cdot \text{valid-cfgI})
then have \( \text{cfg-on} \ : \ \text{repcs} \ s \ \text{cfg} \in \text{MDP} \cdot \text{cfg-on} \ s \)
by (auto dest: MDP \cdot \text-G \cdot \text{valid-cfgD})
from \( \text{cfg} \in ? \) have action \( \text{cfg} \in K \ (\text{abss} \ s) \)
by (rule R \cdot \text-G \cdot \text{cfg-onD-action})
have \( \text{rept} \ s \ (\text{action} \ \text{cfg}) \neq \text{return-pmf} \ s \)
proof (rule ccontr, simp)
  assume \( \text{rept} \ s \ (\text{action} \ \text{cfg}) = \text{return-pmf} \ s \)
  then have \( \text{action} \ \text{cfg} = \text{return-pmf} \ (\text{abss} \ s) \)
  using \( \text{abst-rept-id} \ [\text{OF} \ (\text{action} \ \text{cfg} \in ?)] \)
  by (simp add: abst-def)
moreover have \( (\text{fst} \ s, \ \text{snd} \ s) \in S \)
  using \( \_ \in S \) by (auto dest: S \cdot \text{-abss-S})
moreover have \( \text{abss} \ s = (\text{fst} \ s, \ [\text{snd} \ s]_R) \)
by (metis \text{abss-S calculation}(2) \ \text{prod.collapse})
ultimately show False

79
using (rept s -→) unbounded unfolding rept-def by (cases s) (auto dest: cval-add-non-id)

qed

moreover have \( \text{rept } s \ (\text{action } cfg) \in K s \)

proof –

have \( \text{action } (\text{repcs } s \ cfg) \in K s \)

using cfg-on by blast

then show \(?\text{thesis}\)

by (simp add: repcs-def)

qed

ultimately have \( \text{rept } s \ (\text{action } cfg) \in \mathcal{K}_s \)

by blast

then have \( \text{measure-pmf} \ (\text{action } (\text{repcs } s \ cfg)) \{s\} \leq \mathcal{r}_s \)

by (auto intro: Sup-upper)

moreover have \( \text{rept } s \ (\text{action } cfg) = \text{action } (\text{repcs } s \ cfg) \)

by (simp add: repcs-def)

ultimately have \( \text{measure-pmf} \ (\text{action } (\text{repcs } s \ cfg)) \{s\} \leq \mathcal{r}_s \)

by auto

note this \( \langle \text{rept } s \ (\text{action } cfg) \in \mathcal{K}_s \rangle \)

note \(* = \text{this} \)

let \( ?S = \{ \text{cfg \ . \ \exists } \text{cfg}' s. \ \text{cfg}' \in \text{R-G.valid-cfg} \land \text{cfg} = \text{repcs } s \text{cfg}' \land \text{abss } s = \text{state } \text{cfg}' \} \)

have \( \text{start} : \text{repcs st cfg} \in ?S \)

using \( \langle \text{cfg} \in \text{R-G.valid-cfg} \rangle \) assms unfolding R-G-cfg-on-div-def

by clarsimp (inst-existentials cfg fst st snd st, auto)

have step: \( y \in ?S \) if \( y \in K-cfg x x \in ?S \) for \( x y \)

using that apply safe

subgoal for \( \text{cfg}' l u \)

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)

) substalc

by (simp add: state-absc)

done
done

have **: \( x \in ?S \) if \( \text{repcs st cfg}, x \) \( \in \text{MDP.MC.acc for } x \)

proof –

from \( \text{MDP.MC.acc-relfunD[OF that]} \) obtain \( n \) where \( \langle \lambda a b. b \in K-cfg a \rangle ~ n \rangle \) \( \text{repcs st cfg} x \).

then show \(?\text{ thesis}\)

proof (induction \( n \) arbitrary: \( x \))

case 0

with \( \text{start} \) show \(?\text{ case}\)

by simp

next

case (Suc \( n \))

from this \( \langle 2 \rangle \) [simplified] show \(?\text{ case}\)

by (elim relcomppE step Suc.IH)

qed

qed

have ***: almost-everywhere \( \text{MDP.MC.T} (\text{repcs st cfg}) \) \( \text{alw} \ (\text{HLD } ?S) \)

by (rule \( \text{AE-mp[OF MDP.MC.AE-T-reachable]} \)) (fastforce dest: ** simp: HLD-iff elim: alw-mono)

have \( \langle \text{AE } \omega \text{ in } ?M. ?P \omega \rangle \hspace{1cm} \)

\( \langle \text{AE } \omega \text{ in } ?M. \forall s :: (s \times (c \Rightarrow \text{real})). \rangle \)

(\( \forall c \in X. \text{snd } s \ c > k c \) \land \( s \in \text{state} \) \( \langle \text{MDP.MC.acc} \rangle \) \( \langle \text{repcs st cfg} \rangle \) \( \rightarrow \) \( \langle \text{ev} \ (\text{alw} \ (\lambda xs. \text{shd } xs = s)) \rangle \) \( \langle \text{smap state } \omega \rangle ) \) \( \langle \text{is } ?L \hspace{1cm} ?R \rangle \)

proof
assume $\neg L$
then show $L$
  by eventually-elim auto
next
assume $R$
with $\langle\text{MDP.MC.AE-T-reachable of repcs st cfg}\rangle$
show $L$
proof (eventually-elim, intro allI impI notI, goal-cases)
case (1 $\omega$ s)
from this(1, 2, 5, 6) show case
  by (intro alw-HLD-smap alw-disjoint-ccontr [where $S = \text{state } \cdot \text{MDP.MC.acc }\{\text{repcs st cfg}\}$ and $R = \{s\}$ and $\omega = \text{smap state } \omega$]; simp add: HLD-iff; blast)
qed

also have ... $\iff$
(\forall s :: (s \times (\lambda c. \text{real})).
  (\forall c \in X. \text{snd } s \cdot c > k c) \land s \in \text{state } \cdot (\text{MDP.MC.acc }\{\text{repcs st cfg}\}) \rightarrow
  (\text{AE } \omega \in ?M. \neg (\text{ev (alw }\lambda x s. \text{shd } xs = s)) (\text{smap state } \omega)))
using $\langle\text{MDP.MC.countable-reachable of repcs st cfg}\rangle$
by (rule AE-all-imp-countable,
  auto intro: countable-subset [where $B = \text{state } \cdot \text{MDP.MC.acc }\{\text{repcs st cfg}\}$])
also show $\neg \neg L$
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 $cfg l' u'$
    proof
      from prems have state $x = (l', u')$
        by (metis MDP.cfg-onD-state state-repcs)
      with ($\neg$ state $x$) have [simp]: $l = l' \land u = u'$
        by auto
      show $\neg \neg L$ using prems(1, 3, $\neg$) by (auto simp: R-G.valid-cfg-def intro: $\ast$)
    qed
    done
subgoal
  apply (drule $\ast$)
  apply clarsimp
  apply (rule lt-1)
  apply (rule $\ast$)
  apply (auto dest: R-G.valid-cfg-state-in-S R-G.valid-cfgD)
  done
  apply (rule MDP.valid-cfgD [OF $\langle\text{repcs st cfg }\in \text{valid-cfg}\rangle$]; fail)
using $\ast$ unfolding alw-holds-pred-stream-iff [symmetric] HLD-def $\ast$
  done
qed

end

7.4 Main Result

class\ Probabilistic-Timed-Automaton-Regions-Reachability

begin

lemma R-G-cfg-on-valid:
  $\langle\text{cfg }\in \text{R-G.valid-cfg if } \text{cfg }\in \text{R-G-cfg-on-div } s'\rangle$
using that unfolding R-G-cfg-on-div-def R-G.valid-cfg-def by auto
lemma cfg-on-valid:
\[ \forall cfg \in valid-cfg \text{ if } cfg \in cfg-on-div s \]
using that unfolding \( cfg-on-div-def \) MDP.valid-cfg-def \text{ by auto}

abbreviation path-measure \( P \ cfg \equiv \text{emeasure} (MDP.T \ cfg) \{ x \in \text{space MDP.St.} \ P \ x \} \)
abbreviation \( R-G \)-path-measure \( P \ cfg \equiv \text{emeasure} (R-G.T \ cfg) \{ x \in \text{space R-G.St.} \ P \ x \} \)
abbreviation progressive \( st \equiv \text{cfg-on-div} \ st \cap \{ \text{cfg. alternating} \ cfg \} \)
abbreviation \( R-G \)-progressive \( st \equiv \text{cfg-on-div} \ st \cap \{ \text{cfg. alternating} \ cfg \} \)

Summary of our results on divergent configurations:

lemma absc-valid-cfg-eq:
\[ \text{absc} \ \text{‘ progressive } s = R-G \text{-progressive } s' \]
apply safe
subgoal
unfolding \( s' \)-def \text{ 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)
(simp: \( s' \)-def \( R-G \)-cfg-on-div-def AE-alw-ev-same-loc-iff \[ \text{symmetric] \})
(intro: \( R-G \)-cfg-on-valid absc-repcs-id \[ \text{symmetric] \})
done

Main theorem:

theorem Min-Max-reachability:
\text{notes in-space-UNIV[measurable] and \[ \text{iff] = pred-stream-iff \]}
shows
\[ \bigvee \ \forall cfg \in \text{progressive } s. \ \text{path-measure} (\lambda x. (\text{holds } \varphi \ \text{suntil holds } \psi) (s \ \#\# x)) \ cfg \]
\[ = \bigvee \ \forall cfg \in \text{R-G \-progressive } s'. \ R-G \text{-path-measure} (\lambda x. (\text{holds } \varphi' \ \text{suntil holds } \psi') (s' \ \#\# x)) \ cfg \]
\text{\&} \ \bigvee \ \forall cfg \in \text{progressive } s. \ \text{path-measure} (\lambda x. (\text{holds } \varphi \ \text{suntil holds } \psi) (s \ \#\# x)) \ cfg \]
\[ = \bigvee \ \forall cfg \in \text{R-G \-progressive } s'. \ R-G \text{-path-measure} (\lambda x. (\text{holds } \varphi' \ \text{suntil holds } \psi') (s' \ \#\# x)) \ cfg \]
proof (rule SUP-eq-and-INF-eq; rule bexI[\text{rotated]}; erule IntE)
fix \( cfg \) assume \( cfg-on-div: \ cfg \in \text{R-G \-cfg-on-div } s' \) \text{and} \( cfg \in \text{Collect alternating} \)
then have alternating \( cfg \)
by auto
let \( ?cfg' = \text{repcs } s \ cfg \)
from alternating \( \text{cfg} \)-div \( \text{cfg-on-div} \) have alternating \( ?cfg' \)
by (simp add: R-G-cfg-on-div-def \( s' \)-def AE-alw-ev-same-loc-iff \[ \text{of - } s \])
with \( cfg\)-div (alternating \( \text{cfg} \)) show \( ?cfg' \in \text{cfg-on-div} \ s \cap \text{Collect alternating} \)
by (auto intro: \( \text{cfg-on-div-repcs-strong simp: } s' \)-def)
show \( \text{emeasure} (R-G.T \ \text{cfg}) \ \{ x \in \text{space R-G.St.} \ (\text{holds } \varphi' \ \text{suntil holds } \psi') (s' \ \#\# x) \} \)
\[ = \text{emeasure} (MDP.T \ \text{cfg}) \ \{ x \in \text{space MDP.St.} \ (\text{holds } \varphi \ \text{suntil holds } \psi) (s \ \#\# x) \} \]
(is \( ?a = ?b \)
proof
from \( \text{cfg-on-div} \) have \( \text{cfg} \in \text{R-G \-valid-cfg} \)
by (rule \( \text{R-G \-cfg-on-valid} \)
from \( \text{cfg-on-div} \) have \( \text{cfg} \in \text{R-G \-cfg-on } s' \)
unfolding \( \text{R-G \-cfg-on-div-def} \) by auto
then have state \( \text{cfg} = s' \)
by auto
have \( ?a = ?b \)
apply (rule
\[ \text{path-measure-cfg-repcs''-new} \]
of \( \text{s cfg } \varphi \ \psi \), folded \( \varphi'\)-def \( \psi'\)-def, unfolded \( \_ = s' \) state-repcs \]
)}

82
subgoal
  unfolding s'-def ..

subgoal
  by fact

subgoal
  using (?cfg' ∈ cfg-on-div s ∩ ⊨) by (blast intro: cfg-on-valid)

subgoal premises premises using prem s by (intro ϕ-stream)

subgoal premises prem
  using prem s by (intro ψ-stream)

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 until holds ψ) (s ## x)}
  = emeasure (R.G.T (absc cfg)) {x ∈ space R-G.St. (holds ϕ' until 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 (λs. s ∈ S) and Q = pred-stream (λs. s ∈ S)]
    )

(auto simp: S-abss-S intro: S-abss-S intro!: until-abss until-reps, measurable)

then show ?a = ?b
  by simp

qed

end

end

References
