Formalization of a Monitoring Algorithm for Metric First-Order Temporal Logic

Joshua Schneider Dmitriy Traytel

March 17, 2025

Abstract

A monitor is a runtime verification tool that solves the following problem: Given a stream of time-stamped events and a policy formulated in a specification language, decide whether the policy is satisfied at every point in the stream. We verify the correctness of an executable monitor for specifications given as formulas in metric first-order temporal logic (MFOTL) [1], an expressive extension of linear temporal logic with real-time constraints and first-order quantification. The verified monitor implements a simplified variant of the algorithm used in the efficient MonPoly monitoring tool [2]. The formalization is presented in a RV 2019 paper [4], which also compares the output of the verified monitor to that of other monitoring tools on randomly generated inputs. This case study revealed several errors in the optimized but unverified tools.

Contents

1 Traces and trace prefixes

1.1 Infinite traces

coinductive *ssorted* :: *'a* :: *linorder stream* ⇒ *bool* **where**
  shd s ≤ *shd* (*stl s*) ⟹ *ssorted* (*stl s*) ⟹ *ssorted s*

lemma *ssorted_siterate*[*simp*]: (⋀*n*. *n* ≤ *f n*) ⟹ *ssorted* (*siterate f n*)
  by (*coinduction arbitrary*: *n*) *auto*

lemma *ssortedD*: *ssorted s* ⟹ *s* !! *i* ≤ *stl s* !! *i*
  by (*induct i arbitrary*: *s*) (*auto elim*: *ssorted.cases*)

lemma *ssorted_sdrop*: *ssorted s* ⟹ *ssorted* (*sdrop i s*)
  by (*coinduction arbitrary*: *i s*) (*auto elim*: *ssorted.cases ssortedD*)

lemma *ssorted_monoD*: *ssorted s* ⟹ *i* ≤ *j* ⟹ *s* !! *i* ≤ *s* !! *j*
proof (*induct j* − *i arbitrary*: *j*)
  case (*Suc x*)
  from *Suc*(*1*)[*of j* − *1*] *Suc*(*2*−*4*) *ssortedD*[*of s j* − *1*]
    show *?case* **by** (*cases j*) (*auto simp*: *le_Suc_eq Suc_diff_le*)
qed *simp*

lemma *sorted_stake*: *ssorted s* ⟹ *sorted* (*stake i s*)
  by (*induct i arbitrary*: *s*)
    (*auto elim*: *ssorted.cases simp*: *in_set_conv_nth*
      intro!: *ssorted_monoD*[*of _ 0*, *simplified*, *THEN order_trans*, *OF _ ssortedD*])

lemma *ssorted_monoI*: ∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j* ⟹ *ssorted s*
  by (*coinduction arbitrary*: *s*)
    (*auto dest*: *spec2*[*of _ Suc _ Suc _*] *spec2*[*of _ 0 Suc 0*])

lemma *ssorted_iff_mono*: *ssorted s* ⟷ (∀ *i j*. *i* ≤ *j* ⟶ *s* !! *i* ≤ *s* !! *j*)
  using *ssorted_monoI ssorted_monoD* **by** *metis*

```
lemma ssorted_iff_le_Suc: ssorted s ⟷ (∀ i. s !! i ≤ s !! Suc i)
  using mono_iff_le_Suc[of snth s] by (simp add: mono_def ssorted_iff_mono)

definition sincreasing s = (∀ x. ∃ i. x < s !! i)

lemma sincreasingI: (⋀x. ∃ i. x < s !! i) ⟹ sincreasing s
  by (simp add: sincreasing_def)

lemma sincreasing_grD:
  fixes x :: 'a :: semilattice_sup
  assumes sincreasing s
  shows ∃ j>i. x < s !! j
proof −
  let ?A = insert x {s !! n | n. n ≤ i}
  from assms obtain j where ∗: Sup_fin ?A < s !! j
    by (auto simp: sincreasing_def)
  then have x < s !! j
    by (rule order.strict_trans1[rotated]) (auto intro: Sup_fin.coboundedI)
  moreover have i < j
  proof (rule ccontr)
    assume ¬ i < j
    then have s !! j ∈ ?A by (auto simp: not_less)
    then have s !! j ≤ Sup_fin ?A
      by (auto intro: Sup_fin.coboundedI)
    with ∗ show False by simp
  qed
  ultimately show ?thesis by blast
qed

lemma sincreasing_siterate_nat[simp]:
  fixes n :: nat
  assumes (⋀n. n < f n)
  shows sincreasing (siterate f n)
unfolding sincreasing_def proof
  fix x
  show ∃ i. x < siterate f n !! i
  proof (induction x)
    case 0
    have 0 < siterate f n !! 1
      using order.strict_trans1[OF le0 assms] by simp
    then show ?case ..
  next
    case (Suc x)
    then obtain i where x < siterate f n !! i ..
    then have Suc x < siterate f n !! Suc i
      using order.strict_trans1[OF _ assms] by (simp del: snth.simps)
    then show ?case ..
  qed
qed

lemma sincreasing_stl: sincreasing s ⟹ sincreasing (stl s) for s :: 'a :: semilattice_sup stream
  by (auto 0 4 simp: gr0_conv_Suc intro!: sincreasingI dest: sincreasing_grD[of s 0])

typedef 'a trace = {s :: ('a set × nat) stream. ssorted (smap snd s) ∧ sincreasing (smap snd s)}
  by (intro exI[of _ smap (λi. ({}, i)) nats])
    (auto simp: stream.map_comp stream.map_ident cong: stream.map_cong)
```

setup_lifting *type_definition_trace*

lift_definition Γ :: $'a$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *set* **is**
$\lambda s\ i.\ fst\ (s\ !!\ i)$ **.**
lift_definition τ :: $'a$ *trace* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **is**
$\lambda s\ i.\ snd\ (s\ !!\ i)$ **.**

lemma *stream_eq_iff*: $s = s' \longleftrightarrow (\forall n.\ s\ !!\ n = s'\ !!\ n)$
by (*metis stream.map_cong0 stream_smap_nats*)

lemma *trace_eqI*: $(\bigwedge i.\ \Gamma\ \sigma\ i = \Gamma\ \sigma'\ i) \Longrightarrow (\bigwedge i.\ \tau\ \sigma\ i = \tau\ \sigma'\ i) \Longrightarrow \sigma = \sigma'$
by *transfer* (*auto simp*: *stream_eq_iff intro*!: *prod_eqI*)

lemma τ*_mono*[*simp*]: $i \leq j \Longrightarrow \tau\ s\ i \leq \tau\ s\ j$
by *transfer* (*auto simp*: *ssorted_iff_mono*)

lemma *ex_le_*τ: $\exists j \geq i.\ x \leq \tau\ s\ j$
by (*transfer fixing*: *i x*) (*auto dest*!: *sincreasing_grD*[*of _ i x*] *less_imp_le*)

lemma *le_*τ*_less*: $\tau\ \sigma\ i \leq \tau\ \sigma\ j \Longrightarrow j < i \Longrightarrow \tau\ \sigma\ i = \tau\ \sigma\ j$
by (*simp add*: *antisym*)

lemma *less_*τ*D*: $\tau\ \sigma\ i < \tau\ \sigma\ j \Longrightarrow i < j$
by (*meson* τ*_mono less_le_not_le not_le_imp_less*)

abbreviation $\Delta\ s\ i \equiv \tau\ s\ i - \tau\ s\ (i - 1)$

lift_definition *map_*Γ :: $('a\ set \Rightarrow 'b\ set) \Rightarrow 'a\ trace \Rightarrow 'b\ trace$ **is**
$\lambda f\ s.\ smap\ (\lambda(x, i).\ (f\ x, i))\ s$
by (*auto simp*: *stream.map_comp prod.case_eq_if cong*: *stream.map_cong*)

lemma Γ*_map_*Γ[*simp*]: $\Gamma\ (map\_\Gamma\ f\ s)\ i = f\ (\Gamma\ s\ i)$
by *transfer* (*simp add*: *prod.case_eq_if*)

lemma τ*_map_*Γ[*simp*]: $\tau\ (map\_\Gamma\ f\ s)\ i = \tau\ s\ i$
by *transfer* (*simp add*: *prod.case_eq_if*)

lemma *map_*Γ*_id*[*simp*]: $map\_\Gamma\ id\ s = s$
by *transfer* (*simp add*: *stream.map_id*)

lemma *map_*Γ*_comp*: $map\_\Gamma\ g\ (map\_\Gamma\ f\ s) = map\_\Gamma\ (g \circ f)\ s$
by *transfer* (*simp add*: *stream.map_comp comp_def prod.case_eq_if case_prod_beta'*)

lemma *map_*Γ*_cong*: $\sigma_1 = \sigma_2 \Longrightarrow (\bigwedge x.\ f_1\ x = f_2\ x) \Longrightarrow map\_\Gamma\ f_1\ \sigma_1 = map\_\Gamma\ f_2\ \sigma_2$
by *transfer* (*auto intro*!: *stream.map_cong*)

1.2 Finite trace prefixes

typedef $'a$ *prefix* $= \{p :: ('a\ set \times nat)\ list.\ sorted\ (map\ snd\ p)\}$
by (*auto intro*!: *exI*[*of _* []])

setup_lifting *type_definition_prefix*

lift_definition *pmap_*Γ :: $('a\ set \Rightarrow 'b\ set) \Rightarrow 'a\ prefix \Rightarrow 'b\ prefix$ **is**
$\lambda f.\ map\ (\lambda(x, i).\ (f\ x, i))$
by (*simp add*: *split_beta comp_def*)

lift_definition *last_ts* :: $'a$ *prefix* $\Rightarrow$ *nat* **is**

$\lambda p.$ (*case p of* [] ⇒ *0* | _ ⇒ *snd* (*last p*)) **.**

lift_definition *first_ts* :: *nat* ⇒ *'a prefix* ⇒ *nat* **is**
λn *p*. (*case p of* [] ⇒ *n* | _ ⇒ *snd* (*hd p*)) **.**

lift_definition *pnil* :: *'a prefix* **is** [] **by** *simp*

lift_definition *plen* :: *'a prefix* ⇒ *nat* **is** *length* **.**

lift_definition *psnoc* :: *'a prefix* ⇒ *'a set* × *nat* ⇒ *'a prefix* **is**
λp *x*. *if* (*case p of* [] ⇒ *0* | _ ⇒ *snd* (*last p*)) ≤ *snd x then p* @ [*x*] *else* []
proof (*goal_cases sorted_psnoc*)
case (*sorted_psnoc p x*)
then show *?case*
by (*induction p*) (*auto split*: *if_splits list.splits*)
qed

instantiation *prefix* :: (*type*) *order* **begin**

lift_definition *less_eq_prefix* :: *'a prefix* ⇒ *'a prefix* ⇒ *bool* **is**
λp *q*. ∃ *r*. *q* = *p* @ *r* **.**

definition *less_prefix* :: *'a prefix* ⇒ *'a prefix* ⇒ *bool* **where**
less_prefix x y = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance
proof (*standard*, *goal_cases less refl trans antisym*)
case (*less x y*)
then show *?case* **unfolding** *less_prefix_def* **..**
next
case (*refl x*)
then show *?case* **by** *transfer auto*
next
case (*trans x y z*)
then show *?case* **by** *transfer auto*
next
case (*antisym x y*)
then show *?case* **by** *transfer auto*
qed

end

lemma *psnoc_inject*[*simp*]:
last_ts p ≤ *snd x* ⟹ *last_ts q* ≤ *snd y* ⟹ *psnoc p x* = *psnoc q y* ⟷ (*p* = *q* ∧ *x* = *y*)
by *transfer auto*

lift_definition *prefix_of* :: *'a prefix* ⇒ *'a trace* ⇒ *bool* **is** λp *s*. *stake* (*length p*) *s* = *p* **.**

lemma *prefix_of_pnil*[*simp*]: *prefix_of pnil* σ
by *transfer auto*

lemma *plen_pnil*[*simp*]: *plen pnil* = *0*
by *transfer auto*

lemma *prefix_of_pmap_Γ*[*simp*]: *prefix_of* π σ ⟹ *prefix_of* (*pmap_Γ f* π) (*map_Γ f* σ)
by *transfer auto*

lemma *plen_mono*: $\pi \le \pi'$ ⟹ *plen* π ≤ *plen* π'

by *transfer auto*

lemma *prefix_of_psnocE*: *prefix_of* (*psnoc p x*) *s* $\Longrightarrow$ *last_ts p* $\leq$ *snd x* $\Longrightarrow$
(*prefix_of p s* $\Longrightarrow$ Γ *s* (*plen p*) = *fst x* $\Longrightarrow$ τ *s* (*plen p*) = *snd x* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by *transfer* (*simp del*: *stake.simps add*: *stake_Suc*)

lemma *le_pnil*[*simp*]: *pnil* $\leq$ π
by *transfer auto*

lift_definition *take_prefix* :: *nat* $\Rightarrow$ *'a trace* $\Rightarrow$ *'a prefix* **is** *stake*
by (*auto dest*: *sorted_stake*)

lemma *plen_take_prefix*[*simp*]: *plen* (*take_prefix i* σ) = *i*
by *transfer auto*

lemma *plen_psnoc*[*simp*]: *last_ts* π $\leq$ *snd x* $\Longrightarrow$ *plen* (*psnoc* π *x*) = *plen* π + *1*
by *transfer auto*

lemma *prefix_of_take_prefix*[*simp*]: *prefix_of* (*take_prefix i* σ) σ
by *transfer auto*

lift_definition *pdrop* :: *nat* $\Rightarrow$ *'a prefix* $\Rightarrow$ *'a prefix* **is** *drop*
by (*auto simp*: *drop_map*[*symmetric*] *sorted_wrt_drop*)

lemma *pdrop_0*[*simp*]: *pdrop 0* π = π
by *transfer auto*

lemma *prefix_of_antimono*: $\pi \leq \pi'$ $\Longrightarrow$ *prefix_of* π' *s* $\Longrightarrow$ *prefix_of* π *s*
by *transfer* (*auto simp del*: *stake_add simp add*: *stake_add*[*symmetric*])

lemma *prefix_of_imp_linear*: *prefix_of* π σ $\Longrightarrow$ *prefix_of* π' σ $\Longrightarrow$ $\pi \leq \pi' \vee \pi' \leq \pi$
proof *transfer*
  fix π π' **and** σ :: (*'a set* $\times$ *nat*) *stream*
  assume *assms*: *stake* (*length* π) σ = π *stake* (*length* π') σ = π'
  show ($\exists$ *r*. π' = π @ *r*) $\vee$ ($\exists$ *r*. π = π' @ *r*)
  proof (*cases length* π *length* π' *rule*: *le_cases*)
    case *le*
    then have π' = *take* (*length* π) π' @ *drop* (*length* π) π'
      by *simp*
    moreover have *take* (*length* π) π' = π
      using *assms le* **by** (*metis min.absorb1 take_stake*)
    ultimately show *?thesis* **by** *auto*
  next
    case *ge*
    then have π = *take* (*length* π') π @ *drop* (*length* π') π
      by *simp*
    moreover have *take* (*length* π') π = π'
      using *assms ge* **by** (*metis min.absorb1 take_stake*)
    ultimately show *?thesis* **by** *auto*
  qed
qed

lemma *ex_prefix_of*: $\exists$ *s*. *prefix_of p s*
proof (*transfer*, *intro bexI CollectI conjI*)
  fix *p* :: (*'a set* $\times$ *nat*) *list*
  assume $*$: *sorted* (*map snd p*)
  let *?σ* = *p* @− *smap* (*Pair undefined*) (*fromN* (*snd* (*last p*)))
  show *stake* (*length p*) *?σ* = *p* **by** (*simp add*: *stake_shift*)

```
  have le_last: snd (p ! i) ≤ snd (last p) if i < length p for i
    using sorted_nth_mono[OF *, of i length p − 1] that
    by (cases p) (auto simp: last_conv_nth nth_Cons′)
  with * show ssorted (smap snd ?σ)
    by (force simp: ssorted_iff_mono sorted_iff_nth_mono shift_snth)
  show sincreasing (smap snd ?σ)
  proof (rule sincreasingI)
    fix x
    have x < smap snd ?σ !! Suc (length p + x)
     by simp (metis Suc_pred add.commute diff_Suc_Suc length_greater_0_conv less_add_Suc1 less_diff_conv)
    then show ∃ i. x < smap snd ?σ !! i ..
  qed
qed

lemma τ_prefix_conv: prefix_of p s ⟹ prefix_of p s′ ⟹ i < plen p ⟹ τ s i = τ s′ i
  by transfer (simp add: stake_nth[symmetric])

lemma Γ_prefix_conv: prefix_of p s ⟹ prefix_of p s′ ⟹ i < plen p ⟹ Γ s i = Γ s′ i
  by transfer (simp add: stake_nth[symmetric])

lemma sincreasing_sdrop:
  fixes s :: (′a :: semilattice_sup) stream
  assumes sincreasing s
  shows sincreasing (sdrop n s)
proof (rule sincreasingI)
  fix x
  obtain i where n < i and x < s !! i
    using sincreasing_grD[OF assms] by blast
  then have x < sdrop n s !! (i − n)
    by (simp add: sdrop_snth)
  then show ∃ i. x < sdrop n s !! i ..
qed

lemma ssorted_shift:
  ssorted (xs @− s) = (sorted xs ∧ ssorted s ∧ (∀ x∈set xs. ∀ y∈sset s. x ≤ y))
proof safe
  assume *: ssorted (xs @− s)
  then show sorted xs
    by (auto simp: ssorted_iff_mono shift_snth sorted_iff_nth_mono split: if_splits)
  from ssorted_sdrop[OF *, of length xs] show ssorted s
    by (auto simp: sdrop_shift)
  fix x y assume x ∈ set xs y ∈ sset s
  then obtain i j where i < length xs xs ! i = x s !! j = y
    by (auto simp: set_conv_nth sset_range)
  with ssorted_monoD[OF *, of i j + length xs] show x ≤ y by auto
next
  assume sorted xs ssorted s ∀ x∈set xs. ∀ y∈sset s. x ≤ y
  then show ssorted (xs @− s)
  proof (coinduction arbitrary: xs s)
    case (ssorted xs s)
    with ‹ssorted s› show ?case
      by (subst (asm) ssorted.simps) (auto 0 4 simp: neq_Nil_conv shd_sset intro: exI[of _ _ # _])
  qed
qed

lemma sincreasing_shift:
  assumes sincreasing s
  shows sincreasing (xs @− s)
```

proof (*rule sincreasingI*)
  fix x
  from *assms* **obtain** i **where** $x < s\ !!\ i$
    unfolding *sincreasing_def* **by** *blast*
  then have $x < (xs\ @{-}\ s)\ !!\ (length\ xs + i)$
    by *simp*
  then show $\exists i.\ x < (xs\ @{-}\ s)\ !!\ i$ **..**
qed

lift_definition *replace_prefix* :: *'a prefix* ⇒ *'a trace* ⇒ *'a trace* **is**
  $\lambda\pi\ \sigma.$ *if ssorted* (*smap snd* (π @− *sdrop* (*length* π) σ)) *then*
    π @− *sdrop* (*length* π) σ *else smap* ($\lambda i.$ ({}, i)) *nats*
  by (*auto split*: *if_splits simp*: *stream.map_comp stream.map_ident sdrop_smap*[*symmetric*]
    simp del: *sdrop_smap intro*!: *sincreasing_shift sincreasing_sdrop cong*: *stream.map_cong*)

lemma *prefix_of_replace_prefix*:
  prefix_of (*pmap_Γ f* π) $\sigma \Longrightarrow$ *prefix_of* π (*replace_prefix* π σ)
proof (*transfer*; *safe*; *goal_cases*)
  case (*1 f* π σ)
  then show *?case*
    by (*subst* (*asm*) (*2*) *stake_sdrop*[*symmetric*, *of _ length* π])
      (*auto 0 3 simp*: *ssorted_shift split_beta o_def stake_shift sdrop_smap*[*symmetric*]
        ssorted_sdrop not_le simp del: *sdrop_smap*)
qed

lemma *map_Γ_replace_prefix*:
  $\forall x.$ *f* (*f x*) = *f x* $\Longrightarrow$ *prefix_of* (*pmap_Γ f* π) $\sigma \Longrightarrow$ *map_Γ f* (*replace_prefix* π σ) = *map_Γ f* σ
proof (*transfer*; *safe*; *goal_cases*)
  case (*1 f* π σ)
  then show *?case*
    by (*subst* (*asm*) (*2*) *stake_sdrop*[*symmetric*, *of* σ *length* π],
        subst (*3*) *stake_sdrop*[*symmetric*, *of* σ *length* π])
      (*auto simp*: *ssorted_shift split_beta o_def stake_shift sdrop_smap*[*symmetric*] *ssorted_sdrop*
        not_le simp del: *sdrop_smap cong*: *map_cong*)
qed

lemma *prefix_of_pmap_Γ_D*:
  assumes *prefix_of* (*pmap_Γ f* π) σ
  shows $\exists\sigma'.$ *prefix_of* π σ' $\wedge$ *prefix_of* (*pmap_Γ f* π) (*map_Γ f* σ')
proof −
  from *assms*(*1*) **obtain** σ' **where** *1*: *prefix_of* π σ'
    using *ex_prefix_of* **by** *blast*
  then have *prefix_of* (*pmap_Γ f* π) (*map_Γ f* σ')
    by *transfer simp*
  with *1* **show** *?thesis* **by** *blast*
qed

lemma *prefix_of_map_Γ_D*:
  assumes *prefix_of* π' (*map_Γ f* σ)
  shows $\exists\pi''.$ π' = *pmap_Γ f* π'' $\wedge$ *prefix_of* π'' σ
  using *assms*
  by *transfer* (*auto intro*!: *exI*[*of _ stake* (*length _*) *_*] *elim*: *sym dest*: *sorted_stake*)

lift_definition *pts* :: *'a prefix* ⇒ *nat list* **is** *map snd* **.**

lemma *pts_pmap_Γ*[*simp*]: *pts* (*pmap_Γ f* π) = *pts* π
  by (*transfer fixing*: *f*) (*simp add*: *split_beta*)

2 Finite tables

primrec *tabulate* :: $(nat \Rightarrow 'a) \Rightarrow nat \Rightarrow nat \Rightarrow 'a\ list$ **where**
  tabulate f x 0 = []
| *tabulate f x* (*Suc n*) = *f x* # *tabulate f* (*Suc x*) *n*

lemma *tabulate_alt*: *tabulate f x n* = *map f* [*x* ..< *x* + *n*]
  by (*induct n arbitrary*: *x*) (*auto simp*: *not_le Suc_le_eq upt_rec*)

lemma *length_tabulate*[*simp*]: *length* (*tabulate f x n*) = *n*
  by (*induction n arbitrary*: *x*) *simp_all*

lemma *map_tabulate*[*simp*]: *map f* (*tabulate g x n*) = *tabulate* ($\lambda x.$ *f* (*g x*)) *x n*
  by (*induction n arbitrary*: *x*) *simp_all*

lemma *nth_tabulate*[*simp*]: $k < n \Longrightarrow$ *tabulate f x n* ! *k* = *f* (*x* + *k*)
proof (*induction n arbitrary*: *x k*)
  case (*Suc n*)
  then show *?case* **by** (*cases k*) *simp_all*
qed *simp*

type_synonym *'a tuple* = *'a option list*
type_synonym *'a table* = *'a tuple set*

definition *wf_tuple* :: $nat \Rightarrow nat\ set \Rightarrow 'a\ tuple \Rightarrow bool$ **where**
  wf_tuple n V x $\longleftrightarrow$ *length x* = *n* $\wedge$ ($\forall\, i<n.$ *x*!*i* = *None* $\longleftrightarrow$ $i \notin V$)

definition *table* :: $nat \Rightarrow nat\ set \Rightarrow 'a\ table \Rightarrow bool$ **where**
  table n V X $\longleftrightarrow$ ($\forall\, x \in X.$ *wf_tuple n V x*)

definition *empty_table* = {}

definition *unit_table n* = {*replicate n None*}

definition *singleton_table n i x* = {*tabulate* ($\lambda j.$ *if i* = *j then Some x else None*) *0 n*}

lemma *in_empty_table*[*simp*]: $\neg$ *x* $\in$ *empty_table*
  unfolding *empty_table_def* **by** *simp*

lemma *empty_table*[*simp*]: *table n V empty_table*
  unfolding *table_def empty_table_def* **by** *simp*

lemma *unit_table_wf_tuple*[*simp*]: *V* = {} $\Longrightarrow$ *x* $\in$ *unit_table n* $\Longrightarrow$ *wf_tuple n V x*
  unfolding *unit_table_def wf_tuple_def* **by** *simp*

lemma *unit_table*[*simp*]: *V* = {} $\Longrightarrow$ *table n V* (*unit_table n*)
  unfolding *table_def* **by** *simp*

lemma *in_unit_table*: *v* $\in$ *unit_table n* $\longleftrightarrow$ *wf_tuple n* {} *v*
  unfolding *unit_table_def wf_tuple_def* **by** (*auto intro*!: *nth_equalityI*)

lemma *singleton_table_wf_tuple*[*simp*]: *V* = {*i*} $\Longrightarrow$ *x* $\in$ *singleton_table n i z* $\Longrightarrow$ *wf_tuple n V x*
  unfolding *singleton_table_def wf_tuple_def* **by** *simp*

lemma *singleton_table*[*simp*]: *V* = {*i*} $\Longrightarrow$ *table n V* (*singleton_table n i z*)
  unfolding *table_def* **by** *simp*

lemma *table_Un*[*simp*]: *table n V X* $\Longrightarrow$ *table n V Y* $\Longrightarrow$ *table n V* ($X \cup Y$)
  unfolding *table_def* **by** *auto*

```
lemma wf_tuple_length: wf_tuple n V x ⟹ length x = n
  unfolding wf_tuple_def by simp


fun join1 :: 'a tuple × 'a tuple ⇒ 'a tuple option where
  join1 ([], []) = Some []
| join1 (None # xs, None # ys) = map_option (Cons None) (join1 (xs, ys))
| join1 (Some x # xs, None # ys) = map_option (Cons (Some x)) (join1 (xs, ys))
| join1 (None # xs, Some y # ys) = map_option (Cons (Some y)) (join1 (xs, ys))
| join1 (Some x # xs, Some y # ys) = (if x = y
    then map_option (Cons (Some x)) (join1 (xs, ys))
    else None)
| join1 _ = None

definition join :: 'a table ⇒ bool ⇒ 'a table ⇒ 'a table where
  join A pos B = (if pos then Option.these (join1 ` (A × B))
    else A − Option.these (join1 ` (A × B)))

lemma join_True_code[code]: join A True B = (⋃a ∈ A. ⋃b ∈ B. set_option (join1 (a, b)))
  unfolding join_def by (force simp: Option.these_def image_iff)

lemma join_False_alt: join X False Y = X − join X True Y
  unfolding join_def by auto

lemma self_join1: join1 (xs, ys) ≠ Some xs ⟹ join1 (zs, ys) ≠ Some xs
  by (induct (zs, ys) arbitrary: zs ys xs rule: join1.induct; auto; auto)

lemma join_False_code[code]: join A False B = {a ∈ A. ∀ b ∈ B. join1 (a, b) ≠ Some a}
  unfolding join_False_alt join_True_code
  by (auto simp: Option.these_def image_iff dest: self_join1)

lemma wf_tuple_Nil[simp]: wf_tuple n A [] = (n = 0)
  unfolding wf_tuple_def by auto

lemma Suc_pred': Suc (x − Suc 0) = (case x of 0 ⇒ Suc 0 | _ ⇒ x)
  by (auto split: nat.splits)

lemma wf_tuple_Cons[simp]:
  wf_tuple n A (x # xs) ⟷ ((if x = None then 0 ∉ A else 0 ∈ A) ∧
   (∃ m. n = Suc m ∧ wf_tuple m ((λx. x − 1) ` (A − {0})) xs))
  unfolding wf_tuple_def
  by (auto 0 3 simp: nth_Cons image_iff Ball_def gr0_conv_Suc Suc_pred' split: nat.splits)

lemma join1_wf_tuple:
  join1 (v1, v2) = Some v ⟹ wf_tuple n A v1 ⟹ wf_tuple n B v2 ⟹ wf_tuple n (A ∪ B) v
  by (induct (v1, v2) arbitrary: n v v1 v2 A B rule: join1.induct)
    (auto simp: image_Un Un_Diff split: if_splits)

lemma join_wf_tuple: x ∈ join X b Y ⟹
  ∀ v ∈ X. wf_tuple n A v ⟹ ∀ v ∈ Y. wf_tuple n B v ⟹ (¬ b ⟹ B ⊆ A) ⟹ A ∪ B = C ⟹
wf_tuple n C x
  unfolding join_def
  by (fastforce simp: Option.these_def image_iff sup_absorb1 dest: join1_wf_tuple split: if_splits)

lemma join_table: table n A X ⟹ table n B Y ⟹ (¬ b ⟹ B ⊆ A) ⟹ A ∪ B = C ⟹
  table n C (join X b Y)
  unfolding table_def by (auto elim!: join_wf_tuple)
```

```
lemma wf_tuple_Suc: wf_tuple (Suc m) A a ⟷ a ≠ [] ∧
   wf_tuple m ((λx. x − 1) ‘ (A − {0})) (tl a) ∧ (0 ∈ A ⟷ hd a ≠ None)
  by (cases a) (auto simp: nth_Cons image_iff split: nat.splits)

lemma table_project: table (Suc n) A X ⟹ table n ((λx. x − Suc 0) ‘ (A − {0})) (tl ‘ X)
  unfolding table_def
  by (auto simp: wf_tuple_Suc)

definition restrict where
  restrict A v = map (λi. if i ∈ A then v ! i else None) [0 ..< length v]

lemma restrict_Nil[simp]: restrict A [] = []
  unfolding restrict_def by auto

lemma restrict_Cons[simp]: restrict A (x # xs) =
  (if 0 ∈ A then x # restrict ((λx. x − 1) ‘ (A − {0})) xs else None # restrict ((λx. x − 1) ‘ A) xs)
  unfolding restrict_def
  by (auto simp: map_upt_Suc image_iff Suc_pred' Ball_def simp del: upt_Suc split: nat.splits)

lemma wf_tuple_restrict: wf_tuple n B v ⟹ A ∩ B = C ⟹ wf_tuple n C (restrict A v)
  unfolding restrict_def wf_tuple_def by auto

lemma wf_tuple_restrict_simple: wf_tuple n B v ⟹ A ⊆ B ⟹ wf_tuple n A (restrict A v)
  unfolding restrict_def wf_tuple_def by auto

lemma nth_restrict: i ∈ A ⟹ i < length v ⟹ restrict A v ! i = v ! i
  unfolding restrict_def by auto

lemma restrict_eq_Nil[simp]: restrict A v = [] ⟷ v = []
  unfolding restrict_def by auto

lemma length_restrict[simp]: length (restrict A v) = length v
  unfolding restrict_def by auto

lemma join1_Some_restrict:
  fixes x y :: 'a tuple
  assumes wf_tuple n A x wf_tuple n B y
  shows join1 (x, y) = Some z ⟷ wf_tuple n (A ∪ B) z ∧ restrict A z = x ∧ restrict B z = y
  using assms
proof (induct (x, y) arbitrary: n x y z A B rule: join1.induct)
  case (2 xs ys)
  then show ?case
    by (cases z) (auto 4 0 simp: image_Un Un_Diff)+
next
  case (3 x xs ys)
  then show ?case
    by (cases z) (auto 4 0 simp: image_Un Un_Diff)+
next
  case (4 xs y ys)
  then show ?case
    by (cases z) (auto 4 0 simp: image_Un Un_Diff)+
next
  case (5 x xs y ys)
  then show ?case
    by (cases z) (auto 4 0 simp: image_Un Un_Diff)+
qed auto
```

lemma *restrict_idle*: *wf_tuple n A v* ⟹ *restrict A v* = *v*
by (*induct v arbitrary*: *n A*) (*auto split*: *if_splits*)

lemma *map_the_restrict*:
$i \in A$ ⟹ *map the* (*restrict A v*) ! *i* = *map the v* ! *i*
by (*induct v arbitrary*: *A i*) (*auto simp*: *nth_Cons′ gr0_conv_Suc split*: *option.splits*)

lemma *join_restrict*:
fixes *X Y* :: ′*a tuple set*
assumes ⋀*v*. *v* ∈ *X* ⟹ *wf_tuple n A v* ⋀*v*. *v* ∈ *Y* ⟹ *wf_tuple n B v* ¬ *b* ⟹ *B* ⊆ *A*
shows *v* ∈ *join X b Y* ⟷
wf_tuple n (*A* ∪ *B*) *v* ∧ *restrict A v* ∈ *X* ∧ (*if b then restrict B v* ∈ *Y else restrict B v* ∉ *Y*)
by (*auto 4 4 simp*: *join_def Option.these_def image_iff assms wf_tuple_restrict sup_absorb1 restrict_idle*
restrict_idle[*OF assms*(*1*)] *elim*: *assms*
dest: *join1_Some_restrict*[*OF assms*(*1*,*2*), *THEN iffD1*, *rotated* −*1*]
dest!: *spec*[*of* _ *Some v*]
intro!: *exI*[*of* _ *Some v*] *join1_Some_restrict*[*THEN iffD2*, *symmetric*] *bexI*[*rotated*])

lemma *join_restrict_table*:
assumes *table n A X table n B Y* ¬ *b* ⟹ *B* ⊆ *A*
shows *v* ∈ *join X b Y* ⟷
wf_tuple n (*A* ∪ *B*) *v* ∧ *restrict A v* ∈ *X* ∧ (*if b then restrict B v* ∈ *Y else restrict B v* ∉ *Y*)
using *assms* **unfolding** *table_def*
by (*simp add*: *join_restrict*)

lemma *join_restrict_annotated*:
fixes *X Y* :: ′*a tuple set*
assumes ¬ *b* =*simp*=> *B* ⊆ *A*
shows *join* {*v*. *wf_tuple n A v* ∧ *P v*} *b* {*v*. *wf_tuple n B v* ∧ *Q v*} =
{*v*. *wf_tuple n* (*A* ∪ *B*) *v* ∧ *P* (*restrict A v*) ∧ (*if b then Q* (*restrict B v*) *else* ¬ *Q* (*restrict B v*))}
using *assms*
by (*intro set_eqI*, *subst join_restrict*) (*auto simp*: *wf_tuple_restrict_simple simp_implies_def*)

lemma *in_joinI*: *table n A X* ⟹ *table n B Y* ⟹ (¬*b* ⟹ *B* ⊆ *A*) ⟹ *wf_tuple n* (*A* ∪ *B*) *v* ⟹
restrict A v ∈ *X* ⟹ (*b* ⟹ *restrict B v* ∈ *Y*) ⟹ (¬*b* ⟹ *restrict B v* ∉ *Y*) ⟹ *v* ∈ *join X b Y*
unfolding *table_def*
by (*subst join_restrict*) (*auto*)

lemma *in_joinE*: *v* ∈ *join X b Y* ⟹ *table n A X* ⟹ *table n B Y* ⟹ (¬ *b* ⟹ *B* ⊆ *A*) ⟹
(*wf_tuple n* (*A* ∪ *B*) *v* ⟹ *restrict A v* ∈ *X* ⟹ *if b then restrict B v* ∈ *Y else restrict B v* ∉ *Y* ⟹
P) ⟹ *P*
unfolding *table_def*
by (*subst* (*asm*) *join_restrict*) (*auto*)

definition *qtable* :: *nat* ⇒ *nat set* ⇒ (′*a tuple* ⇒ *bool*) ⇒ (′*a tuple* ⇒ *bool*) ⇒
′*a table* ⇒ *bool* **where**
qtable n A P Q X ⟷ *table n A X* ∧ (∀ *x*. (*x* ∈ *X* ∧ *P x* ⟶ *Q x*) ∧ (*wf_tuple n A x* ∧ *P x* ∧ *Q x*
⟶ *x* ∈ *X*))

abbreviation *wf_table* **where**
wf_table n A Q X ≡ *qtable n A* (λ_. *True*) *Q X*

lemma *wf_table_iff*: *wf_table n A Q X* ⟷ (∀ *x*. *x* ∈ *X* ⟷ (*Q x* ∧ *wf_tuple n A x*))
unfolding *qtable_def table_def* **by** *auto*

lemma *table_wf_table*: *table n A X* = *wf_table n A* (λ*v*. *v* ∈ *X*) *X*
unfolding *table_def wf_table_iff* **by** *auto*

lemma *qtableI*: *table n A X* $\Longrightarrow$
  $(\bigwedge x.\ x \in X \Longrightarrow wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow$
  $(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow x \in X) \Longrightarrow$
  qtable n A P Q X
  unfolding *qtable_def table_def* **by** *auto*

lemma *in_qtableI*: $qtable\ n\ A\ P\ Q\ X \Longrightarrow wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow x \in X$
  unfolding *qtable_def* **by** *blast*

lemma *in_qtableE*: $qtable\ n\ A\ P\ Q\ X \Longrightarrow x \in X \Longrightarrow P\ x \Longrightarrow (wf\_tuple\ n\ A\ x \Longrightarrow Q\ x \Longrightarrow R) \Longrightarrow R$
  unfolding *qtable_def table_def* **by** *blast*

lemma *qtable_empty*: $(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow False) \Longrightarrow qtable\ n\ A\ P\ Q\ empty\_table$
  unfolding *qtable_def table_def empty_table_def* **by** *auto*

lemma *qtable_empty_iff*: $qtable\ n\ A\ P\ Q\ empty\_table = (\forall x.\ wf\_tuple\ n\ A\ x \longrightarrow P\ x \longrightarrow Q\ x \longrightarrow False)$
  unfolding *qtable_def table_def empty_table_def* **by** *auto*

lemma *qtable_unit_table*: $(\bigwedge x.\ wf\_tuple\ n\ \{\}\ x \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow qtable\ n\ \{\}\ P\ Q\ (unit\_table\ n)$
  unfolding *qtable_def table_def in_unit_table* **by** *auto*

lemma *qtable_union*: $qtable\ n\ A\ P\ Q1\ X \Longrightarrow qtable\ n\ A\ P\ Q2\ Y \Longrightarrow$
  $(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ x \lor Q2\ x) \Longrightarrow qtable\ n\ A\ P\ Q\ (X \cup Y)$
  unfolding *qtable_def table_def* **by** *blast*

lemma *qtable_Union*: $finite\ I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow qtable\ n\ A\ P\ (Qi\ i)\ (Xi\ i)) \Longrightarrow$
  $(\bigwedge x.\ wf\_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow (\exists i \in I.\ Qi\ i\ x)) \Longrightarrow qtable\ n\ A\ P\ Q\ (\bigcup i \in I.\ Xi\ i)$
proof (*induct I arbitrary*: *Q rule*: *finite_induct*)
  case (*insert i F*)
  then show *?case*
    by (*auto intro*!: *qtable_union*[**where** $?Q1.0 = Qi\ i$ **and** $?Q2.0 = \lambda x.\ \exists i{\in}F.\ Qi\ i\ x$])
qed (*auto intro*!: *qtable_empty*[*unfolded empty_table_def*])

lemma *qtable_join*:
  assumes $qtable\ n\ A\ P\ Q1\ X$ $qtable\ n\ B\ P\ Q2\ Y$ $\neg\ b \Longrightarrow B \subseteq A$ $C = A \cup B$
  $\bigwedge x.\ wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow P\ (restrict\ A\ x) \land P\ (restrict\ B\ x)$
  $\bigwedge x.\ b \Longrightarrow wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \land Q2\ (restrict\ B\ x)$
  $\bigwedge x.\ \neg\ b \Longrightarrow wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \land \neg\ Q2\ (restrict\ B\ x)$
  shows $qtable\ n\ C\ P\ Q\ (join\ X\ b\ Y)$
proof (*rule qtableI*)
  from *assms*(*1*−*4*) **show** $table\ n\ C\ (join\ X\ b\ Y)$
    unfolding *qtable_def* **by** (*auto simp*: *join_table*)
next
  fix x **assume** $x \in join\ X\ b\ Y$ $wf\_tuple\ n\ C\ x$ $P\ x$
  with *assms*(*1*−*3*) *assms*(*5*−*7*)[*of x*] **show** $Q\ x$ **unfolding** *qtable_def*
    by (*auto 0 2 simp*: *wf_tuple_restrict_simple elim*!: *in_joinE split*: *if_splits*)
next
  fix x **assume** $wf\_tuple\ n\ C\ x$ $P\ x$ $Q\ x$
  with *assms*(*1*−*4*) *assms*(*5*−*7*)[*of x*] **show** $x \in join\ X\ b\ Y$ **unfolding** *qtable_def*
    by (*auto dest*: *wf_tuple_restrict_simple intro*!: *in_joinI*[*of n A X B Y*])
qed

lemma *qtable_join_fixed*:
  assumes $qtable\ n\ A\ P\ Q1\ X$ $qtable\ n\ B\ P\ Q2\ Y$ $\neg\ b \Longrightarrow B \subseteq A$ $C = A \cup B$
  $\bigwedge x.\ wf\_tuple\ n\ C\ x \Longrightarrow P\ x \Longrightarrow P\ (restrict\ A\ x) \land P\ (restrict\ B\ x)$
  shows $qtable\ n\ C\ P\ (\lambda x.\ Q1\ (restrict\ A\ x) \land (if\ b\ then\ Q2\ (restrict\ B\ x)\ else\ \neg\ Q2\ (restrict\ B\ x)))$

(*join X b Y*)
  by (*rule qtable_join*[*OF assms*]) *auto*

lemma *wf_tuple_cong*:
  assumes *wf_tuple n A v wf_tuple n A w* $\forall x \in A$. *map the v* ! *x* = *map the w* ! *x*
  shows *v* = *w*
proof −
  from *assms*(*1*,*2*) **have** *length v* = *length w* **unfolding** *wf_tuple_def* **by** *simp*
  from *this assms* **show** *v* = *w*
  proof (*induct v w arbitrary*: *n A rule*: *list_induct2*)
    case (*Cons x xs y ys*)
    let *?n* = *n* − *1* **and** *?A* = (λ*x*. *x* − *1*) ‘ (*A* − {*0*})
    have ∗: *map the xs* ! *z* = *map the ys* ! *z* **if** *z* ∈ *?A* **for** *z*
      using *that Cons*(*5*)[*THEN bspec*, *of Suc z*]
      by (*cases z*) (*auto simp*: *le_Suc_eq split*: *if_splits*)
    from *Cons*(*1*,*3*−*5*) **show** *?case*
      by (*auto intro*!: *Cons*(*2*)[*of ?n ?A*] ∗ *split*: *if_splits*)
  qed *simp*
qed

definition *mem_restr* :: ′*a list set* ⇒ ′*a tuple* ⇒ *bool* **where**
  mem_restr A x ⟷ (∃ *y*∈*A*. *list_all2* (λ*a b*. *a* ≠ *None* ⟶ *a* = *Some b*) *x y*)

lemma *mem_restrI*: *y* ∈ *A* ⟹ *length y* = *n* ⟹ *wf_tuple n V x* ⟹ ∀ *i*∈*V*. *x* ! *i* = *Some* (*y* ! *i*) ⟹ *mem_restr A x*
  unfolding *mem_restr_def wf_tuple_def* **by** (*force simp add*: *list_all2_conv_all_nth*)

lemma *mem_restrE*: *mem_restr A x* ⟹ *wf_tuple n V x* ⟹ ∀ *i*∈*V*. *i* < *n* ⟹
  (⋀*y*. *y* ∈ *A* ⟹ ∀ *i*∈*V*. *x* ! *i* = *Some* (*y* ! *i*) ⟹ *P*) ⟹ *P*
  unfolding *mem_restr_def wf_tuple_def* **by** (*fastforce simp add*: *list_all2_conv_all_nth*)

lemma *mem_restr_IntD*: *mem_restr* (*A* ∩ *B*) *v* ⟹ *mem_restr A v* ∧ *mem_restr B v*
  unfolding *mem_restr_def* **by** *auto*

lemma *mem_restr_Un_iff*: *mem_restr* (*A* ∪ *B*) *x* ⟷ *mem_restr A x* ∨ *mem_restr B x*
  unfolding *mem_restr_def* **by** *blast*

lemma *mem_restr_UNIV* [*simp*]: *mem_restr UNIV x*
  unfolding *mem_restr_def*
  by (*auto simp add*: *list.rel_map intro*!: *exI*[*of _ map the x*] *list.rel_refl*)

lemma *restrict_mem_restr*[*simp*]: *mem_restr A x* ⟹ *mem_restr A* (*restrict V x*)
  unfolding *mem_restr_def restrict_def*
  by (*auto simp*: *list_all2_conv_all_nth elim*!: *bexI*[*rotated*])

definition *lift_envs* :: ′*a list set* ⇒ ′*a list set* **where**
  lift_envs R = (λ(*a*,*b*). *a* # *b*) ‘ (*UNIV* × *R*)

lemma *lift_envs_mem_restr*[*simp*]: *mem_restr A x* ⟹ *mem_restr* (*lift_envs A*) (*a* # *x*)
  by (*auto simp*: *mem_restr_def lift_envs_def*)

lemma *qtable_project*:
  assumes *qtable* (*Suc n*) *A* (*mem_restr* (*lift_envs R*)) *P X*
  shows *qtable n* ((λ*x*. *x* − *Suc 0*) ‘ (*A* − {*0*})) (*mem_restr R*)
      (λ*v*. ∃ *x*. *P* ((*if 0* ∈ *A then Some x else None*) # *v*)) (*tl* ‘ *X*)
      (**is** *qtable n ?A* (*mem_restr R*) *?P ?X*)
proof ((*rule qtableI*; (*elim exE*)?), *goal_cases table left right*)
  case *table*

with *assms* **show** *?case*
unfolding *qtable_def* **by** (*simp add*: *table_project*)
next
case (*left v*)
from *assms* **have** $[] \notin X$
unfolding *qtable_def table_def* **by** *fastforce*
with *left(1)* **obtain** *x* **where** $x \# v \in X$
by (*metis* (*no_types*, *opaque_lifting*) *image_iff hd_Cons_tl*)
with *assms* **show** *?case*
by (*rule in_qtableE*) (*auto simp*: *left(3)* *split*: *if_splits*)
next
case (*right v x*)
with *assms* **have** (*if* $0 \in A$ *then Some x else None*) $\# v \in X$
by (*elim in_qtableI*) *auto*
then show *?case*
by (*auto simp*: *image_iff elim*: *bexI*[*rotated*])
qed

lemma *qtable_cong*: $qtable\ n\ A\ P\ Q\ X \Longrightarrow A = B \Longrightarrow (\bigwedge v.\ P\ v \Longrightarrow Q\ v \longleftrightarrow Q'\ v) \Longrightarrow qtable\ n\ B\ P\ Q'\ X$
by (*auto simp*: *qtable_def*)

3 Abstract monitors and slicing

3.1 First-order specifications

We abstract from first-order trace specifications by referring only to their semantics. A first-order specification is described by a finite set of free variables and a satisfaction function that defines for every trace the pairs of valuations and time-points for which the specification is satisfied.

locale *fo_spec* =
fixes
nfv :: *nat* **and** *fv* :: *nat set* **and**
sat :: $'a\ trace \Rightarrow\ 'b\ list \Rightarrow nat \Rightarrow bool$
assumes
fv_less_nfv: $x \in fv \Longrightarrow x < nfv$ **and**
sat_fv_cong: $(\bigwedge x.\ x \in fv \Longrightarrow v!x = v'!x) \Longrightarrow sat\ \sigma\ v\ i = sat\ \sigma\ v'\ i$
begin

definition *verdicts* :: $'a\ trace \Rightarrow (nat \times\ 'b\ tuple)\ set$ **where**
$verdicts\ \sigma = \{(i, v).\ wf\_tuple\ nfv\ fv\ v \wedge sat\ \sigma\ (map\ the\ v)\ i\}$

end

We usually employ a monitor to find the *violations* of a specification. That is, the monitor should output the satisfactions of its negation. Moreover, all monitor implementations must work with finite prefixes. We are therefore interested in co-safety properties, which allow us to identify all satisfactions on finite prefixes.

locale *cosafety_fo_spec* = *fo_spec* +
assumes *cosafety_lr*: $sat\ \sigma\ v\ i \Longrightarrow \exists \pi.\ prefix\_of\ \pi\ \sigma \wedge (\forall \sigma'.\ prefix\_of\ \pi\ \sigma' \longrightarrow sat\ \sigma'\ v\ i)$
begin

lemma *cosafety*: $sat\ \sigma\ v\ i \longleftrightarrow (\exists \pi.\ prefix\_of\ \pi\ \sigma \wedge (\forall \sigma'.\ prefix\_of\ \pi\ \sigma' \longrightarrow sat\ \sigma'\ v\ i))$
using *cosafety_lr* **by** *blast*

end

3.2 Monitor function

We model monitors abstractly as functions from prefixes to verdict sets. The following locale specifies a minimal set of properties that any reasonable monitor should have.

locale *monitor* = *fo_spec* +
  fixes M :: $'a$ *prefix* $\Rightarrow$ (*nat* $\times$ $'b$ *tuple*) *set*
  assumes
    mono_monitor: $\pi \le \pi' \Longrightarrow M\ \pi \subseteq M\ \pi'$ **and**
    wf_monitor: $(i,\ v) \in M\ \pi \Longrightarrow$ *wf_tuple nfv fv v* **and**
    sound_monitor: $(i,\ v) \in M\ \pi \Longrightarrow$ *prefix_of* $\pi\ \sigma \Longrightarrow$ *sat* σ (*map the v*) i **and**
    complete_monitor: *prefix_of* $\pi\ \sigma \Longrightarrow$ *wf_tuple nfv fv v* $\Longrightarrow$
      $(\bigwedge\sigma.$ *prefix_of* $\pi\ \sigma \Longrightarrow$ *sat* σ (*map the v*) $i) \Longrightarrow \exists\pi'.$ *prefix_of* $\pi'\ \sigma \wedge (i,\ v) \in M\ \pi'$

A monitor for a co-safety specification computes precisely the set of all satisfactions in the limit:

abbreviation (**in** *monitor*) *M_limit* $\sigma \equiv \bigcup\{M\ \pi \mid \pi.$ *prefix_of* $\pi\ \sigma\}$

locale *cosafety_monitor* = *cosafety_fo_spec* + *monitor*
begin

lemma *M_limit_eq*: *M_limit* σ = *verdicts* σ
proof
  show $\bigcup\{M\ \pi \mid \pi.$ *prefix_of* $\pi\ \sigma\} \subseteq$ *verdicts* σ
    by (*auto simp*: *verdicts_def wf_monitor sound_monitor*)
next
  show $\bigcup\{M\ \pi \mid \pi.$ *prefix_of* $\pi\ \sigma\} \supseteq$ *verdicts* σ
    unfolding *verdicts_def*
  proof *safe*
    fix *i v*
    assume *wf_tuple nfv fv v* **and** *sat* σ (*map the v*) i
    then obtain π **where** *prefix_of* $\pi\ \sigma \wedge (\forall\sigma'.$ *prefix_of* $\pi\ \sigma' \longrightarrow$ *sat* σ' (*map the v*) $i)$
      using *cosafety_lr* **by** *blast*
    with ‹*wf_tuple nfv fv v*› **obtain** π' **where** *prefix_of* $\pi'\ \sigma \wedge (i,\ v) \in M\ \pi'$
      using *complete_monitor* **by** *blast*
    then show $(i,\ v) \in \bigcup\{M\ \pi \mid \pi.$ *prefix_of* $\pi\ \sigma\}$
      by *blast*
  qed
qed

end

The monitor function M adds some information over *sat*, namely when a verdict is output. One possible behavior is that the monitor outputs its verdicts for one time-point at a time, in increasing order of time-points. Then M is uniquely defined by a progress function, which returns for every prefix the time-point up to which all verdicts are computed.

locale *progress* = *fo_spec* _ _ *sat* **for** *sat* :: $'a$ *trace* $\Rightarrow$ $'b$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *bool* +
  fixes *progress* :: $'a$ *prefix* $\Rightarrow$ *nat*
  assumes
    progress_mono: $\pi \le \pi' \Longrightarrow$ *progress* $\pi \le$ *progress* π' **and**
    ex_progress_ge: $\exists\pi.$ *prefix_of* $\pi\ \sigma \wedge x \le$ *progress* π **and**
    progress_sat_cong: *prefix_of* $\pi\ \sigma \Longrightarrow$ *prefix_of* $\pi\ \sigma' \Longrightarrow i <$ *progress* $\pi \Longrightarrow$
      sat $\sigma\ v\ i \longleftrightarrow$ *sat* $\sigma'\ v\ i$
  — The last condition is not necessary to obtain a proper monitor function. However, it corresponds to the intuitive understanding of monitor progress, and it results in a stronger characterisation. In particular, it implies that the specification is co-safety, as we will show below.
begin

definition M :: $'a$ *prefix* $\Rightarrow$ (*nat* $\times$ $'b$ *tuple*) *set* **where**

```
  M π = {(i, v). i < progress π ∧ wf_tuple nfv fv v ∧
    (∀σ. prefix_of π σ ⟶ sat σ (map the v) i)}

lemma M_alt: M π = {(i, v). i < progress π ∧ wf_tuple nfv fv v ∧
    (∃σ. prefix_of π σ ∧ sat σ (map the v) i)}
  using ex_prefix_of[of π]
  by (auto simp: M_def cong: progress_sat_cong)

end

sublocale progress ⊆ cosafety_monitor _ _ _ M
proof
  fix i v and σ :: 'a trace
  assume sat σ v i
  moreover obtain π where ∗: prefix_of π σ i < progress π
    using ex_progress_ge by (auto simp: less_eq_Suc_le)
  ultimately have sat σ' v i if prefix_of π σ' for σ'
    using that by (simp cong: progress_sat_cong)
  with ∗ show ∃π. prefix_of π σ ∧ (∀σ'. prefix_of π σ' ⟶ sat σ' v i)
    by blast
next
  fix π π' :: 'a prefix
  assume π ≤ π'
  then show M π ⊆ M π'
    by (auto simp: M_def intro: progress_mono prefix_of_antimono
        elim: order.strict_trans2)
next
  fix i v π and σ :: 'a trace
  assume ∗: (i, v) ∈ M π
  then show wf_tuple nfv fv v
    by (simp add: M_def)
  assume prefix_of π σ
  with ∗ show sat σ (map the v) i
    by (simp add: M_def)
next
  fix i v π and σ :: 'a trace
  assume ∗: prefix_of π σ wf_tuple nfv fv v ⋀σ'. prefix_of π σ' ⟹ sat σ' (map the v) i
  show ∃π'. prefix_of π' σ ∧ (i, v) ∈ M π'
  proof (cases i < progress π)
    case True
    with ∗ show ?thesis by (auto simp: M_def)
  next
    case False
    obtain π' where ∗∗: prefix_of π' σ ∧ i < progress π'
      using ex_progress_ge by (auto simp: less_eq_Suc_le)
    then have π ≤ π'
      using ‹prefix_of π σ› prefix_of_imp_linear False progress_mono
      by (blast intro: order.strict_trans2)
    with ∗ ∗∗ show ?thesis
      by (auto simp: M_def intro: prefix_of_antimono)
  qed
qed
```

3.3 Slicing

Sliceable specifications can be evaluated meaningfully on a subset of events.

```
locale abstract_slicer =
  fixes relevant_events :: 'b list set ⇒ 'a set
```

begin

abbreviation $\mathit{slice} :: {'b}\ \mathit{list\ set} \Rightarrow {'a}\ \mathit{trace} \Rightarrow {'a}\ \mathit{trace}$ **where**
  $\mathit{slice}\ S \equiv \mathit{map\_\Gamma}\ (\lambda D.\ D \cap \mathit{relevant\_events}\ S)$

abbreviation $\mathit{pslice} :: {'b}\ \mathit{list\ set} \Rightarrow {'a}\ \mathit{prefix} \Rightarrow {'a}\ \mathit{prefix}$ **where**
  $\mathit{pslice}\ S \equiv \mathit{pmap\_\Gamma}\ (\lambda D.\ D \cap \mathit{relevant\_events}\ S)$

lemma $\mathit{prefix\_of\_psliceI}$: $\mathit{prefix\_of}\ \pi\ \sigma \Longrightarrow \mathit{prefix\_of}\ (\mathit{pslice}\ S\ \pi)\ (\mathit{slice}\ S\ \sigma)$
  by $(\mathit{transfer\ fixing}{:}\ S)\ \mathit{auto}$

lemma $\mathit{plen\_pslice}[\mathit{simp}]$: $\mathit{plen}\ (\mathit{pslice}\ S\ \pi) = \mathit{plen}\ \pi$
  by $(\mathit{transfer\ fixing}{:}\ S)\ \mathit{simp}$

lemma $\mathit{pslice\_pnil}[\mathit{simp}]$: $\mathit{pslice}\ S\ \mathit{pnil} = \mathit{pnil}$
  by $(\mathit{transfer\ fixing}{:}\ S)\ \mathit{simp}$

lemma $\mathit{last\_ts\_pslice}[\mathit{simp}]$: $\mathit{last\_ts}\ (\mathit{pslice}\ S\ \pi) = \mathit{last\_ts}\ \pi$
  by $(\mathit{transfer\ fixing}{:}\ S)\ (\mathit{simp\ add}{:}\ \mathit{last\_map\ case\_prod\_beta\ split}{:}\ \mathit{list.split})$

abbreviation $\mathit{verdict\_filter} :: {'b}\ \mathit{list\ set} \Rightarrow (\mathit{nat} \times {'b}\ \mathit{tuple})\ \mathit{set} \Rightarrow (\mathit{nat} \times {'b}\ \mathit{tuple})\ \mathit{set}$ **where**
  $\mathit{verdict\_filter}\ S\ V \equiv \{(i,\ v) \in V.\ \mathit{mem\_restr}\ S\ v\}$

end

locale $\mathit{sliceable\_fo\_spec} = \mathit{fo\_spec}\ \_\ \_\ \mathit{sat} + \mathit{abstract\_slicer}\ \mathit{relevant\_events}$
  for $\mathit{relevant\_events} :: {'b}\ \mathit{list\ set} \Rightarrow {'a}\ \mathit{set}$ **and** $\mathit{sat} :: {'a}\ \mathit{trace} \Rightarrow {'b}\ \mathit{list} \Rightarrow \mathit{nat} \Rightarrow \mathit{bool}$ +
  assumes $\mathit{sliceable}$: $v \in S \Longrightarrow \mathit{sat}\ (\mathit{slice}\ S\ \sigma)\ v\ i \longleftrightarrow \mathit{sat}\ \sigma\ v\ i$
begin

lemma $\mathit{union\_verdicts\_slice}$:
  assumes part: $\bigcup \mathcal{S} = \mathit{UNIV}$
  shows $\bigcup((\lambda S.\ \mathit{verdict\_filter}\ S\ (\mathit{verdicts}\ (\mathit{slice}\ S\ \sigma)))\ `\ \mathcal{S}) = \mathit{verdicts}\ \sigma$
proof *safe*
  fix $S\ i$ **and** $v :: {'b}\ \mathit{tuple}$
  assume $(i,\ v) \in \mathit{verdicts}\ (\mathit{slice}\ S\ \sigma)$
  then have tuple: $\mathit{wf\_tuple}\ \mathit{nfv}\ \mathit{fv}\ v$ **and** $\mathit{sat}\ (\mathit{slice}\ S\ \sigma)\ (\mathit{map\ the}\ v)\ i$
    by $(\mathit{auto\ simp}{:}\ \mathit{verdicts\_def})$
  assume $\mathit{mem\_restr}\ S\ v$
  then obtain v' **where** $v' \in S$ **and** $\mathit{1}{:}\ \forall i{\in}\mathit{fv}.\ v\ !\ i = \mathit{Some}\ (v'\ !\ i)$
    using *tuple* **by** $(\mathit{auto\ simp}{:}\ \mathit{fv\_less\_nfv\ elim}{:}\ \mathit{mem\_restrE})$
  then have $\mathit{sat}\ (\mathit{slice}\ S\ \sigma)\ v'\ i$
    using $\langle \mathit{sat}\ (\mathit{slice}\ S\ \sigma)\ (\mathit{map\ the}\ v)\ i \rangle\ \mathit{tuple}$
    by $(\mathit{auto\ simp}{:}\ \mathit{wf\_tuple\_length}\ \mathit{fv\_less\_nfv}\ \mathit{cong}{:}\ \mathit{sat\_fv\_cong})$
  then have $\mathit{sat}\ \sigma\ v'\ i$
    using $\mathit{sliceable}[\mathit{OF}\ \langle v' \in S \rangle]$ **by** *simp*
  then have $\mathit{sat}\ \sigma\ (\mathit{map\ the}\ v)\ i$
    using *tuple 1*
    by $(\mathit{auto\ simp}{:}\ \mathit{wf\_tuple\_length}\ \mathit{fv\_less\_nfv}\ \mathit{cong}{:}\ \mathit{sat\_fv\_cong})$
  then show $(i,\ v) \in \mathit{verdicts}\ \sigma$
    using *tuple* **by** $(\mathit{simp\ add}{:}\ \mathit{verdicts\_def})$
next
  fix i **and** $v :: {'b}\ \mathit{tuple}$
  assume $(i,\ v) \in \mathit{verdicts}\ \sigma$
  then have tuple: $\mathit{wf\_tuple}\ \mathit{nfv}\ \mathit{fv}\ v$ **and** $\mathit{sat}\ \sigma\ (\mathit{map\ the}\ v)\ i$
    by $(\mathit{auto\ simp}{:}\ \mathit{verdicts\_def})$
  from *part* **obtain** S **where** $S \in \mathcal{S}$ **and** $\mathit{map\ the}\ v \in S$
    by *blast*

```
  then have mem_restr S v
    using mem_restrI[of map the v S nfv fv] tuple
    by (auto simp: wf_tuple_def fv_less_nfv)
  moreover have sat (slice S σ) (map the v) i
    using ‹sat σ (map the v) i› sliceable[OF ‹map the v ∈ S›] by simp
  then have (i, v) ∈ verdicts (slice S σ)
    using tuple by (simp add: verdicts_def)
  ultimately show (i, v) ∈ (⋃S∈𝒮. verdict_filter S (verdicts (slice S σ)))
    using ‹S ∈ 𝒮› by blast
qed

end
```

We define a similar notion for monitors. It is potentially stronger because the time-point at which verdicts are output must not change.

```
locale sliceable_monitor = monitor _ _ sat M + abstract_slicer relevant_events
  for relevant_events :: 'b list set ⇒ 'a set and sat :: 'a trace ⇒ 'b list ⇒ nat ⇒ bool and M +
  assumes sliceable_M: mem_restr S v ⟹ (i, v) ∈ M (pslice S π) ⟷ (i, v) ∈ M π
begin

lemma union_M_pslice:
  assumes part: ⋃𝒮 = UNIV
  shows ⋃((λS. verdict_filter S (M (pslice S π))) ` 𝒮) = M π
proof safe
  fix S i and v :: 'b tuple
  assume mem_restr S v and (i, v) ∈ M (pslice S π)
  then show (i, v) ∈ M π using sliceable_M by blast
next
  fix i and v :: 'b tuple
  assume (i, v) ∈ M π
  then have tuple: wf_tuple nfv fv v
    by (rule wf_monitor)
  from part obtain S where S ∈ 𝒮 and map the v ∈ S
    by blast
  then have mem_restr S v
    using mem_restrI[of map the v S nfv fv] tuple
    by (auto simp: wf_tuple_def fv_less_nfv)
  then have (i, v) ∈ M (pslice S π)
    using ‹(i, v) ∈ M π› sliceable_M by blast
  then show (i, v) ∈ (⋃S∈𝒮. verdict_filter S (M (pslice S π)))
    using ‹S ∈ 𝒮› ‹mem_restr S v› by blast
qed

end
```

If the specification is sliceable and the monitor's progress depends only on time-stamps, then also the monitor itself is sliceable.

```
locale timed_progress = progress +
  assumes progress_time_conv: pts π = pts π' ⟹ progress π = progress π'

locale sliceable_timed_progress = sliceable_fo_spec + timed_progress
begin

lemma progress_pslice[simp]: progress (pslice S π) = progress π
  by (simp cong: progress_time_conv)

end
```

```
sublocale sliceable_timed_progress ⊆ sliceable_monitor _ _ _ _ M
proof
  fix S :: 'a list set and v i π
  assume *: mem_restr S v
  show (i, v) ∈ M (pslice S π) ⟷ (i, v) ∈ M π (is ?L ⟷ ?R)
  proof
    assume ?L
    with * show ?R
      by (auto 0 4 simp: M_def wf_tuple_def elim!: mem_restrE
          box_equals[OF sliceable sat_fv_cong sat_fv_cong, THEN iffD1, rotated -1]
          intro: prefix_of_psliceI dest: fv_less_nfv spec[of _ slice S _])
  next
    assume ?R
    with * show ?L
      by (auto 0 4 simp: M_alt wf_tuple_def elim!: mem_restrE
        box_equals[OF sliceable sat_fv_cong sat_fv_cong, THEN iffD2, rotated -1]
        intro: exI[of _ slice S _] prefix_of_psliceI dest: fv_less_nfv)
  qed
qed
```

4 Intervals

```
typedef ℐ = {(i :: nat, j :: enat). i ≤ j}
  by (intro exI[of _ (0, ∞)]) auto

setup_lifting type_definition_ℐ

instantiation ℐ :: equal begin
lift_definition equal_ℐ :: ℐ ⇒ ℐ ⇒ bool is (=) .
instance by standard (transfer, auto)
end

instantiation ℐ :: linorder begin
lift_definition less_eq_ℐ :: ℐ ⇒ ℐ ⇒ bool is (≤) .
lift_definition less_ℐ :: ℐ ⇒ ℐ ⇒ bool is (<) .
instance by standard (transfer, auto)+
end


lift_definition all :: ℐ is (0, ∞) by simp
lift_definition left :: ℐ ⇒ nat is fst .
lift_definition right :: ℐ ⇒ enat is snd .
lift_definition point :: nat ⇒ ℐ is λn. (n, enat n) by simp
lift_definition init :: nat ⇒ ℐ is λn. (0, enat n) by auto
abbreviation mem where mem n I ≡ (left I ≤ n ∧ n ≤ right I)
lift_definition subtract :: nat ⇒ ℐ ⇒ ℐ is
  λn (i, j). (i − n, j − enat n) by (auto simp: diff_enat_def split: enat.splits)
lift_definition add :: nat ⇒ ℐ ⇒ ℐ is
  λn (a, b). (a, b + enat n) by (auto simp add: add_increasing2)

lemma left_right: left I ≤ right I
  by transfer auto

lemma point_simps[simp]:
  left (point n) = n
  right (point n) = n
  by (transfer; auto)+
```

lemma *init_simps*[*simp*]:
  left (*init n*) = *0*
  right (*init n*) = *n*
  by (*transfer*; *auto*)+

lemma *subtract_simps*[*simp*]:
  left (*subtract n I*) = *left I* − *n*
  right (*subtract n I*) = *right I* − *n*
  subtract 0 I = *I*
  subtract x (*point y*) = *point* (*y* − *x*)
  by (*transfer*; *auto*)+

definition *shifted* :: $\mathcal{I} \Rightarrow \mathcal{I}$ *set* **where**
  shifted I = (λ*n*. *subtract n I*) ‘ {*0* .. (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*)}

lemma *subtract_too_much*: *i* > (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*) ⟹
  subtract i I = *subtract* (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*) *I*
  by *transfer* (*auto split*: *enat.splits*)

lemma *subtract_shifted*: *subtract n I* ∈ *shifted I*
  by (*cases n* ≤ (*case right I of* ∞ ⇒ *left I* | *enat n* ⇒ *n*))
    (*auto simp*: *shifted_def subtract_too_much*)

lemma *finite_shifted*: *finite* (*shifted I*)
  unfolding *shifted_def* **by** *auto*

definition *interval* :: *nat* ⇒ *enat* ⇒ $\mathcal{I}$ **where**
  interval l r = (*if l* ≤ *r then Abs_*$\mathcal{I}$ (*l*, *r*) *else undefined*)

lemma [*code abstract*]: *Rep_*$\mathcal{I}$ (*interval l r*) = (*if l* ≤ *r then* (*l*, *r*) *else Rep_*$\mathcal{I}$ *undefined*)
  unfolding *interval_def* **using** *Abs_*$\mathcal{I}$*_inverse* **by** *simp*

5 Metric first-order temporal logic

context begin

5.1 Formulas and satisfiability

qualified type_synonym *name* = *string*
qualified type_synonym ′*a event* = (*name* × ′*a list*)
qualified type_synonym ′*a database* = ′*a event set*
qualified type_synonym ′*a prefix* = (*name* × ′*a list*) *prefix*
qualified type_synonym ′*a trace* = (*name* × ′*a list*) *trace*

qualified type_synonym ′*a env* = ′*a list*

qualified datatype ′*a trm* = *Var nat* | *is_Const*: *Const* ′*a*

qualified primrec *fvi_trm* :: *nat* ⇒ ′*a trm* ⇒ *nat set* **where**
  fvi_trm b (*Var x*) = (*if b* ≤ *x then* {*x* − *b*} *else* {})
| *fvi_trm b* (*Const _*) = {}

abbreviation *fv_trm* ≡ *fvi_trm 0*

qualified primrec *eval_trm* :: ′*a env* ⇒ ′*a trm* ⇒ ′*a* **where**
  eval_trm v (*Var x*) = *v* ! *x*

| $eval\_trm\ v\ (Const\ x) = x$

lemma $eval\_trm\_cong$: $\forall x \in fv\_trm\ t.\ v\ !\ x = v'\ !\ x \Longrightarrow eval\_trm\ v\ t = eval\_trm\ v'\ t$
by $(cases\ t)\ simp\_all$

qualified datatype $(discs\_sels)\ 'a\ formula = Pred\ name\ 'a\ trm\ list \mid Eq\ 'a\ trm\ 'a\ trm$
$\mid Neg\ 'a\ formula \mid Or\ 'a\ formula\ 'a\ formula \mid Exists\ 'a\ formula$
$\mid Prev\ \mathcal{I}\ 'a\ formula \mid Next\ \mathcal{I}\ 'a\ formula$
$\mid Since\ 'a\ formula\ \mathcal{I}\ 'a\ formula \mid Until\ 'a\ formula\ \mathcal{I}\ 'a\ formula$

qualified primrec $fvi :: nat \Rightarrow 'a\ formula \Rightarrow nat\ set$ **where**
$fvi\ b\ (Pred\ r\ ts) = (\bigcup t \in set\ ts.\ fvi\_trm\ b\ t)$
$\mid fvi\ b\ (Eq\ t1\ t2) = fvi\_trm\ b\ t1 \cup fvi\_trm\ b\ t2$
$\mid fvi\ b\ (Neg\ \varphi) = fvi\ b\ \varphi$
$\mid fvi\ b\ (Or\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
$\mid fvi\ b\ (Exists\ \varphi) = fvi\ (Suc\ b)\ \varphi$
$\mid fvi\ b\ (Prev\ I\ \varphi) = fvi\ b\ \varphi$
$\mid fvi\ b\ (Next\ I\ \varphi) = fvi\ b\ \varphi$
$\mid fvi\ b\ (Since\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
$\mid fvi\ b\ (Until\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$

abbreviation $fv \equiv fvi\ 0$

lemma $finite\_fvi\_trm[simp]$: $finite\ (fvi\_trm\ b\ t)$
by $(cases\ t)\ simp\_all$

lemma $finite\_fvi[simp]$: $finite\ (fvi\ b\ \varphi)$
by $(induction\ \varphi\ arbitrary{:}\ b)\ simp\_all$

lemma fvi_trm_Suc: $x \in fvi\_trm\ (Suc\ b)\ t \longleftrightarrow Suc\ x \in fvi\_trm\ b\ t$
by $(cases\ t)\ auto$

lemma fvi_Suc: $x \in fvi\ (Suc\ b)\ \varphi \longleftrightarrow Suc\ x \in fvi\ b\ \varphi$
by $(induction\ \varphi\ arbitrary{:}\ b)\ (simp\_all\ add{:}\ fvi\_trm\_Suc)$

lemma fvi_Suc_bound:
assumes $\forall i \in fvi\ (Suc\ b)\ \varphi.\ i < n$
shows $\forall i \in fvi\ b\ \varphi.\ i < Suc\ n$
proof
fix i
assume $i \in fvi\ b\ \varphi$
with $assms$ **show** $i < Suc\ n$ **by** $(cases\ i)\ (simp\_all\ add{:}\ fvi\_Suc)$
qed

qualified definition $nfv :: 'a\ formula \Rightarrow nat$ **where**
$nfv\ \varphi = Max\ (insert\ 0\ (Suc\ `\ fv\ \varphi))$

qualified definition $envs :: 'a\ formula \Rightarrow 'a\ env\ set$ **where**
$envs\ \varphi = \{v.\ length\ v = nfv\ \varphi\}$

lemma $nfv\_simps[simp]$:
$nfv\ (Neg\ \varphi) = nfv\ \varphi$
$nfv\ (Or\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$
$nfv\ (Prev\ I\ \varphi) = nfv\ \varphi$
$nfv\ (Next\ I\ \varphi) = nfv\ \varphi$
$nfv\ (Since\ \varphi\ I\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$
$nfv\ (Until\ \varphi\ I\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$
unfolding nfv_def **by** $(simp\_all\ add{:}\ image\_Un\ Max\_Un[symmetric])$

```
lemma fvi_less_nfv: ∀ i∈fv φ. i < nfv φ
  unfolding nfv_def
  by (auto simp add: Max_gr_iff intro: max.strict_coboundedI2)


qualified primrec future_reach :: 'a formula ⇒ enat where
  future_reach (Pred _ _) = 0
| future_reach (Eq _ _) = 0
| future_reach (Neg φ) = future_reach φ
| future_reach (Or φ ψ) = max (future_reach φ) (future_reach ψ)
| future_reach (Exists φ) = future_reach φ
| future_reach (Prev I φ) = future_reach φ − left I
| future_reach (Next I φ) = future_reach φ + right I + 1
| future_reach (Since φ I ψ) = max (future_reach φ) (future_reach ψ − left I)
| future_reach (Until φ I ψ) = max (future_reach φ) (future_reach ψ) + right I + 1


qualified primrec sat :: 'a trace ⇒ 'a env ⇒ nat ⇒ 'a formula ⇒ bool where
  sat σ v i (Pred r ts) = ((r, map (eval_trm v) ts) ∈ Γ σ i)
| sat σ v i (Eq t1 t2) = (eval_trm v t1 = eval_trm v t2)
| sat σ v i (Neg φ) = (¬ sat σ v i φ)
| sat σ v i (Or φ ψ) = (sat σ v i φ ∨ sat σ v i ψ)
| sat σ v i (Exists φ) = (∃ z. sat σ (z # v) i φ)
| sat σ v i (Prev I φ) = (case i of 0 ⇒ False | Suc j ⇒ mem (τ σ i − τ σ j) I ∧ sat σ v j φ)
| sat σ v i (Next I φ) = (mem (τ σ (Suc i) − τ σ i) I ∧ sat σ v (Suc i) φ)
| sat σ v i (Since φ I ψ) = (∃ j≤i. mem (τ σ i − τ σ j) I ∧ sat σ v j ψ ∧ (∀ k ∈ {j <.. i}. sat σ v k φ))
| sat σ v i (Until φ I ψ) = (∃ j≥i. mem (τ σ j − τ σ i) I ∧ sat σ v j ψ ∧ (∀ k ∈ {i ..< j}. sat σ v k φ))

lemma sat_Until_rec: sat σ v i (Until φ I ψ) ⟷
  mem 0 I ∧ sat σ v i ψ ∨
  (Δ σ (i + 1) ≤ right I ∧ sat σ v i φ ∧ sat σ v (i + 1) (Until φ (subtract (Δ σ (i + 1)) I) ψ))
  (is ?L ⟷ ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j: i ≤ j mem (τ σ j − τ σ i) I sat σ v j ψ ∀ k ∈ {i ..< j}. sat σ v k φ
    by auto
  then show ?R
  proof (cases i = j)
    case False
    with j(1,2) have Δ σ (i + 1) ≤ right I
      by (auto elim: order_trans[rotated] simp: diff_le_mono)
    moreover from False j(1,4) have sat σ v i φ by auto
    moreover from False j have sat σ v (i + 1) (Until φ (subtract (Δ σ (i + 1)) I) ψ)
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
    ultimately show ?thesis by blast
  qed simp
next
  assume Δ: Δ σ (i + 1) ≤ right I and now: sat σ v i φ and
   next: sat σ v (i + 1) (Until φ (subtract (Δ σ (i + 1)) I) ψ)
  from next obtain j where j: i + 1 ≤ j mem (τ σ j − τ σ (i + 1)) ((subtract (Δ σ (i + 1)) I))
      sat σ v j ψ ∀ k ∈ {i + 1 ..< j}. sat σ v k φ
    by auto
  from Δ j(1,2) have mem (τ σ j − τ σ i) I
    by (cases right I) (auto simp: le_diff_conv2)
  with now j(1,3,4) show ?L by (auto simp: le_eq_less_or_eq[of i] intro!: exI[of _ j])
qed auto
```

```
lemma sat_Since_rec: sat σ v i (Since φ I ψ) ⟷
  mem 0 I ∧ sat σ v i ψ ∨
  (i > 0 ∧ Δ σ i ≤ right I ∧ sat σ v i φ ∧ sat σ v (i − 1) (Since φ (subtract (Δ σ i) I) ψ))
  (is ?L ⟷ ?R)
proof (rule iffI; (elim disjE conjE)?)
  assume ?L
  then obtain j where j: j ≤ i mem (τ σ i − τ σ j) I sat σ v j ψ ∀k ∈ {j <.. i}. sat σ v k φ
    by auto
  then show ?R
  proof (cases i = j)
    case False
    with j(1) obtain k where [simp]: i = k + 1
      by (cases i) auto
    with j(1,2) False have Δ σ i ≤ right I
      by (auto elim: order_trans[rotated] simp: diff_le_mono2 le_Suc_eq)
    moreover from False j(1,4) have sat σ v i φ by auto
    moreover from False j have sat σ v (i − 1) (Since φ (subtract (Δ σ i) I) ψ)
      by (cases right I) (auto simp: le_diff_conv le_diff_conv2 intro!: exI[of _ j])
    ultimately show ?thesis by auto
  qed simp
next
  assume i: 0 < i and Δ: Δ σ i ≤ right I and now: sat σ v i φ and
   prev: sat σ v (i − 1) (Since φ (subtract (Δ σ i) I) ψ)
  from prev obtain j where j: j ≤ i − 1 mem (τ σ (i − 1) − τ σ j) ((subtract (Δ σ i) I))
      sat σ v j ψ ∀k ∈ {j <.. i − 1}. sat σ v k φ
    by auto
  from Δ i j(1,2) have mem (τ σ i − τ σ j) I
    by (cases right I) (auto simp: le_diff_conv2)
  with now i j(1,3,4) show ?L by (auto simp: le_Suc_eq gr0_conv_Suc intro!: exI[of _ j])
qed auto

lemma sat_Since_0: sat σ v 0 (Since φ I ψ) ⟷ mem 0 I ∧ sat σ v 0 ψ
  by auto

lemma sat_Since_point: sat σ v i (Since φ I ψ) ⟹
    (⋀j. j ≤ i ⟹ mem (τ σ i − τ σ j) I ⟹ sat σ v i (Since φ (point (τ σ i − τ σ j)) ψ) ⟹ P) ⟹ P
  by (auto intro: diff_le_self)

lemma sat_Since_pointD: sat σ v i (Since φ (point t) ψ) ⟹ mem t I ⟹ sat σ v i (Since φ I ψ)
  by auto

lemma eval_trm_fvi_cong: ∀x∈fv_trm t. v!x = v'!x ⟹ eval_trm v t = eval_trm v' t
  by (cases t) simp_all

lemma sat_fvi_cong: ∀x∈fv φ. v!x = v'!x ⟹ sat σ v i φ = sat σ v' i φ
proof (induct φ arbitrary: v v' i)
  case (Pred n ts)
  show ?case by (simp cong: map_cong eval_trm_fvi_cong[OF Pred[simplified, THEN bspec]])
next
  case (Eq x1 x2)
  then show ?case  unfolding fvi.simps sat.simps by (metis UnCI eval_trm_fvi_cong)
next
  case (Exists φ)
  then show ?case unfolding sat.simps by (intro iff_exI) (simp add: fvi_Suc nth_Cons')
qed (auto 8 0 simp add: nth_Cons' split: nat.splits intro!: iff_exI)
```

5.2 Defined connectives

qualified definition *And* φ ψ = *Neg* (*Or* (*Neg* φ) (*Neg* ψ))

lemma *fvi_And*: *fvi b* (*And* φ ψ) = *fvi b* φ ∪ *fvi b* ψ
  unfolding *And_def* **by** *simp*

lemma *nfv_And*[*simp*]: *nfv* (*And* φ ψ) = *max* (*nfv* φ) (*nfv* ψ)
  unfolding *nfv_def* **by** (*simp add*: *fvi_And image_Un Max_Un*[*symmetric*])

lemma *future_reach_And*: *future_reach* (*And* φ ψ) = *max* (*future_reach* φ) (*future_reach* ψ)
  unfolding *And_def* **by** *simp*

lemma *sat_And*: *sat* σ *v i* (*And* φ ψ) = (*sat* σ *v i* φ ∧ *sat* σ *v i* ψ)
  unfolding *And_def* **by** *simp*

qualified definition *And_Not* φ ψ = *Neg* (*Or* (*Neg* φ) ψ)

lemma *fvi_And_Not*: *fvi b* (*And_Not* φ ψ) = *fvi b* φ ∪ *fvi b* ψ
  unfolding *And_Not_def* **by** *simp*

lemma *nfv_And_Not*[*simp*]: *nfv* (*And_Not* φ ψ) = *max* (*nfv* φ) (*nfv* ψ)
  unfolding *nfv_def* **by** (*simp add*: *fvi_And_Not image_Un Max_Un*[*symmetric*])

lemma *future_reach_And_Not*: *future_reach* (*And_Not* φ ψ) = *max* (*future_reach* φ) (*future_reach* ψ)
  unfolding *And_Not_def* **by** *simp*

lemma *sat_And_Not*: *sat* σ *v i* (*And_Not* φ ψ) = (*sat* σ *v i* φ ∧ ¬ *sat* σ *v i* ψ)
  unfolding *And_Not_def* **by** *simp*

5.3 Safe formulas

fun *safe_formula* :: *'a MFOTL.formula* ⇒ *bool* **where**
  safe_formula (*MFOTL.Eq t1 t2*) = (*MFOTL.is_Const t1* ∨ *MFOTL.is_Const t2*)
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Const x*) (*MFOTL.Const y*))) = *True*
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Var x*) (*MFOTL.Var y*))) = (*x* = *y*)
| *safe_formula* (*MFOTL.Pred e ts*) = *True*
| *safe_formula* (*MFOTL.Neg* (*MFOTL.Or* (*MFOTL.Neg* φ) ψ)) = (*safe_formula* φ ∧
  (*safe_formula* ψ ∧ *MFOTL.fv* ψ ⊆ *MFOTL.fv* φ ∨ (*case* ψ *of MFOTL.Neg* ψ' ⇒ *safe_formula* ψ' | _ ⇒ *False*)))
| *safe_formula* (*MFOTL.Or* φ ψ) = (*MFOTL.fv* ψ = *MFOTL.fv* φ ∧ *safe_formula* φ ∧ *safe_formula* ψ)
| *safe_formula* (*MFOTL.Exists* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Prev I* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Next I* φ) = (*safe_formula* φ)
| *safe_formula* (*MFOTL.Since* φ *I* ψ) = (*MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ∧
  (*safe_formula* φ ∨ (*case* φ *of MFOTL.Neg* φ' ⇒ *safe_formula* φ' | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* (*MFOTL.Until* φ *I* ψ) = (*MFOTL.fv* φ ⊆ *MFOTL.fv* ψ ∧
  (*safe_formula* φ ∨ (*case* φ *of MFOTL.Neg* φ' ⇒ *safe_formula* φ' | _ ⇒ *False*)) ∧ *safe_formula* ψ)
| *safe_formula* _ = *False*

lemma *disjE_Not2*: $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (\neg P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$
  by *blast*

lemma *safe_formula_induct*[*consumes 1*]:
  assumes *safe_formula* φ
    and $\bigwedge$*t1 t2*. *MFOTL.is_Const t1* ⟹ *P* (*MFOTL.Eq t1 t2*)
    and $\bigwedge$*t1 t2*. *MFOTL.is_Const t2* ⟹ *P* (*MFOTL.Eq t1 t2*)

and ⋀x y. P (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))
and ⋀x y. x = y ⟹ P (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y)))
and ⋀e ts. P (MFOTL.Pred e ts)
and ⋀φ ψ. ¬ (safe_formula (MFOTL.Neg ψ) ∧ MFOTL.fv ψ ⊆ MFOTL.fv φ) ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.And φ ψ)
and ⋀φ ψ. safe_formula ψ ⟹ MFOTL.fv ψ ⊆ MFOTL.fv φ ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.And_Not φ ψ)
and ⋀φ ψ. MFOTL.fv ψ = MFOTL.fv φ ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.Or φ ψ)
and ⋀φ. P φ ⟹ P (MFOTL.Exists φ)
and ⋀I φ. P φ ⟹ P (MFOTL.Prev I φ)
and ⋀I φ. P φ ⟹ P (MFOTL.Next I φ)
and ⋀φ I ψ. MFOTL.fv φ ⊆ MFOTL.fv ψ ⟹ safe_formula φ ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.Since φ I ψ)
and ⋀φ I ψ. MFOTL.fv (MFOTL.Neg φ) ⊆ MFOTL.fv ψ ⟹
¬ safe_formula (MFOTL.Neg φ) ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.Since (MFOTL.Neg φ) I ψ)
and ⋀φ I ψ. MFOTL.fv φ ⊆ MFOTL.fv ψ ⟹ safe_formula φ ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.Until φ I ψ)
and ⋀φ I ψ. MFOTL.fv (MFOTL.Neg φ) ⊆ MFOTL.fv ψ ⟹
¬ safe_formula (MFOTL.Neg φ) ⟹ P φ ⟹ P ψ ⟹ P (MFOTL.Until (MFOTL.Neg φ) I ψ)
shows P φ
using assms(1)
proof (induction rule: safe_formula.induct)
case (5 φ ψ)
then show ?case
by (cases ψ)
(auto 0 3 elim!: disjE_Not2 intro: assms[unfolded MFOTL.And_def MFOTL.And_Not_def])
next
case (10 φ I ψ)
then show ?case
by (cases φ) (auto 0 3 elim!: disjE_Not2 intro: assms)
next
case (11 φ I ψ)
then show ?case
by (cases φ) (auto 0 3 elim!: disjE_Not2 intro: assms)
qed (auto intro: assms)

5.4 Slicing traces

qualified primrec matches :: 'a env ⇒ 'a formula ⇒ name × 'a list ⇒ bool **where**
matches v (Pred r ts) e = (r = fst e ∧ map (eval_trm v) ts = snd e)
| matches v (Eq _ _) e = False
| matches v (Neg φ) e = matches v φ e
| matches v (Or φ ψ) e = (matches v φ e ∨ matches v ψ e)
| matches v (Exists φ) e = (∃ z. matches (z # v) φ e)
| matches v (Prev I φ) e = matches v φ e
| matches v (Next I φ) e = matches v φ e
| matches v (Since φ I ψ) e = (matches v φ e ∨ matches v ψ e)
| matches v (Until φ I ψ) e = (matches v φ e ∨ matches v ψ e)

lemma matches_fvi_cong: ∀ x∈fv φ. v!x = v'!x ⟹ matches v φ e = matches v' φ e
proof (induct φ arbitrary: v v')
case (Pred n ts)
show ?case **by** (simp cong: map_cong eval_trm_fvi_cong[OF Pred[simplified, THEN bspec]])
next
case (Exists φ)
then show ?case **unfolding** matches.simps **by** (intro iff_exI) (simp add: fvi_Suc nth_Cons')
qed (auto 5 0 simp add: nth_Cons')

abbreviation *relevant_events* **where** *relevant_events* φ $S \equiv \{e.\ S \cap \{v.\ matches\ v\ \varphi\ e\} \neq \{\}\}$

lemma *sat_slice_strong*: *relevant_events* φ $S \subseteq E \Longrightarrow v \in S \Longrightarrow$
  sat σ v i $\varphi \longleftrightarrow$ *sat* (*map_Γ* ($\lambda D.\ D \cap E$) σ) v i φ
proof (*induction* φ *arbitrary*: *v S i*)
  case (*Pred r ts*)
  then show *?case* **by** (*auto simp*: *subset_eq*)
next
  case (*Eq t1 t2*)
  show *?case*
    unfolding *sat.simps*
    by *simp*
next
  case (*Neg* φ)
  then show *?case* **by** *simp*
next
  case (*Or* φ ψ)
  show *?case* **using** *Or.IH*[*of S*] *Or.prems*
    by (*auto simp*: *Collect_disj_eq Int_Un_distrib subset_iff*)
next
  case (*Exists* φ)
  have *sat* σ (z # v) i φ = *sat* (*map_Γ* ($\lambda D.\ D \cap E$) σ) (z # v) i φ **for** z
    using *Exists.prems* **by** (*auto intro*!: *Exists.IH*[*of* $\{z \# v \mid v.\ v \in S\}$])
  then show *?case* **by** *simp*
next
  case (*Prev I* φ)
  then show *?case* **by** (*auto cong*: *nat.case_cong*)
next
  case (*Next I* φ)
  then show *?case* **by** *simp*
next
  case (*Since* φ *I* ψ)
  show *?case* **using** *Since.IH*[*of S*] *Since.prems*
   by (*auto simp*: *Collect_disj_eq Int_Un_distrib subset_iff*)
next
  case (*Until* φ *I* ψ)
  show *?case* **using** *Until.IH*[*of S*] *Until.prems*
   by (*auto simp*: *Collect_disj_eq Int_Un_distrib subset_iff*)
qed

end

interpretation *MFOTL_slicer*: *abstract_slicer relevant_events* φ **for** φ **.**

lemma *sat_slice_iff*:
  assumes $v \in S$
  shows *MFOTL.sat* σ v i $\varphi \longleftrightarrow$ *MFOTL.sat* (*MFOTL_slicer.slice* φ S σ) v i φ
  by (*rule sat_slice_strong*[*of S*, *OF subset_refl assms*])

lemma *slice_replace_prefix*:
  prefix_of (*MFOTL_slicer.pslice* φ R π) $\sigma \Longrightarrow$
    MFOTL_slicer.slice φ R (*replace_prefix* π σ) = *MFOTL_slicer.slice* φ R σ
  by (*rule map_Γ_replace_prefix*) *auto*

6 Monitor implementation

6.1 Monitorable formulas

definition *mmonitorable* $\varphi \longleftrightarrow$ *safe_formula* $\varphi \wedge$ *MFOTL.future_reach* $\varphi \neq \infty$

fun *mmonitorable_exec* :: *'a MFOTL.formula* $\Rightarrow$ *bool* **where**
mmonitorable_exec (*MFOTL.Eq t1 t2*) = (*MFOTL.is_Const t1* $\vee$ *MFOTL.is_Const t2*)
| *mmonitorable_exec* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Const x*) (*MFOTL.Const y*))) = *True*
| *mmonitorable_exec* (*MFOTL.Neg* (*MFOTL.Eq* (*MFOTL.Var x*) (*MFOTL.Var y*))) = ($x = y$)
| *mmonitorable_exec* (*MFOTL.Pred e ts*) = *True*
| *mmonitorable_exec* (*MFOTL.Neg* (*MFOTL.Or* (*MFOTL.Neg* φ) ψ)) = (*mmonitorable_exec* $\varphi \wedge$ (*mmonitorable_exec* $\psi \wedge$ *MFOTL.fv* $\psi \subseteq$ *MFOTL.fv* $\varphi \vee$ (*case* ψ *of MFOTL.Neg* $\psi' \Rightarrow$ *mmonitorable_exec* ψ' | _ $\Rightarrow$ *False*)))
| *mmonitorable_exec* (*MFOTL.Or* φ ψ) = (*MFOTL.fv* ψ = *MFOTL.fv* $\varphi \wedge$ *mmonitorable_exec* $\varphi \wedge$ *mmonitorable_exec* ψ)
| *mmonitorable_exec* (*MFOTL.Exists* φ) = (*mmonitorable_exec* φ)
| *mmonitorable_exec* (*MFOTL.Prev I* φ) = (*mmonitorable_exec* φ)
| *mmonitorable_exec* (*MFOTL.Next I* φ) = (*mmonitorable_exec* $\varphi \wedge$ *right I* $\neq \infty$)
| *mmonitorable_exec* (*MFOTL.Since* φ *I* ψ) = (*MFOTL.fv* $\varphi \subseteq$ *MFOTL.fv* $\psi \wedge$ (*mmonitorable_exec* $\varphi \vee$ (*case* φ *of MFOTL.Neg* $\varphi' \Rightarrow$ *mmonitorable_exec* φ' | _ $\Rightarrow$ *False*)) $\wedge$ *mmonitorable_exec* ψ)
| *mmonitorable_exec* (*MFOTL.Until* φ *I* ψ) = (*MFOTL.fv* $\varphi \subseteq$ *MFOTL.fv* $\psi \wedge$ *right I* $\neq \infty \wedge$ (*mmonitorable_exec* $\varphi \vee$ (*case* φ *of MFOTL.Neg* $\varphi' \Rightarrow$ *mmonitorable_exec* φ' | _ $\Rightarrow$ *False*)) $\wedge$ *mmonitorable_exec* ψ)
| *mmonitorable_exec* _ = *False*

lemma *plus_eq_enat_iff*: $a + b =$ *enat* $i \longleftrightarrow (\exists j\ k.\ a =$ *enat* $j \wedge b =$ *enat* $k \wedge j + k = i)$
by (*cases a*; *cases b*) *auto*

lemma *minus_eq_enat_iff*: $a -$ *enat* $k =$ *enat* $i \longleftrightarrow (\exists j.\ a =$ *enat* $j \wedge j - k = i)$
by (*cases a*) *auto*

lemma *safe_formula_mmonitorable_exec*: *safe_formula* $\varphi \Longrightarrow$ *MFOTL.future_reach* $\varphi \neq \infty \Longrightarrow$ *mmonitorable_exec* φ
proof (*induct* φ *rule*: *safe_formula.induct*)
case (*5* φ ψ)
then show *?case*
unfolding *safe_formula.simps future_reach.simps mmonitorable_exec.simps*
by (*fastforce split*: *formula.splits*)
next
case (*6* φ ψ)
then show *?case*
unfolding *safe_formula.simps future_reach.simps mmonitorable_exec.simps*
by (*fastforce split*: *formula.splits*)
next
case (*10* φ *I* ψ)
then show *?case*
unfolding *safe_formula.simps future_reach.simps mmonitorable_exec.simps*
by (*fastforce split*: *formula.splits*)
next
case (*11* φ *I* ψ)
then show *?case*
unfolding *safe_formula.simps future_reach.simps mmonitorable_exec.simps*
by (*fastforce simp*: *plus_eq_enat_iff split*: *formula.splits*)
qed (*auto simp add*: *plus_eq_enat_iff minus_eq_enat_iff*)

lemma *mmonitorable_exec_mmonitorable*: *mmonitorable_exec* $\varphi \Longrightarrow$ *mmonitorable* φ
proof (*induct* φ *rule*: *mmonitorable_exec.induct*)

```
  case (5 φ ψ)
  then show ?case
    unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
    by (fastforce split: formula.splits)
next
  case (10 φ I ψ)
  then show ?case
    unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
    by (fastforce split: formula.splits)
next
  case (11 φ I ψ)
  then show ?case
    unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
    by (fastforce simp: one_enat_def split: formula.splits)
qed (auto simp add: mmonitorable_def one_enat_def)

lemma monitorable_formula_code[code]: mmonitorable φ = mmonitorable_exec φ
  using mmonitorable_exec_mmonitorable safe_formula_mmonitorable_exec mmonitorable_def
  by blast
```

6.2 The executable monitor

```
type_synonym ts = nat

type_synonym 'a mbuf2 = 'a table list × 'a table list
type_synonym 'a msaux = (ts × 'a table) list
type_synonym 'a muaux = (ts × 'a table × 'a table) list

datatype 'a mformula =
    MRel 'a table
  | MPred MFOTL.name 'a MFOTL.trm list
  | MAnd 'a mformula bool 'a mformula 'a mbuf2
  | MOr 'a mformula 'a mformula 'a mbuf2
  | MExists 'a mformula
  | MPrev ℐ 'a mformula bool 'a table list ts list
  | MNext ℐ 'a mformula bool ts list
  | MSince bool 'a mformula ℐ 'a mformula 'a mbuf2 ts list 'a msaux
  | MUntil bool 'a mformula ℐ 'a mformula 'a mbuf2 ts list 'a muaux

record 'a mstate =
  mstate_i :: nat
  mstate_m :: 'a mformula
  mstate_n :: nat

fun eq_rel :: nat ⇒ 'a MFOTL.trm ⇒ 'a MFOTL.trm ⇒ 'a table where
  eq_rel n (MFOTL.Const x) (MFOTL.Const y) = (if x = y then unit_table n else empty_table)
| eq_rel n (MFOTL.Var x) (MFOTL.Const y) = singleton_table n x y
| eq_rel n (MFOTL.Const x) (MFOTL.Var y) = singleton_table n y x
| eq_rel n (MFOTL.Var x) (MFOTL.Var y) = undefined

fun neq_rel :: nat ⇒ 'a MFOTL.trm ⇒ 'a MFOTL.trm ⇒ 'a table where
  neq_rel n (MFOTL.Const x) (MFOTL.Const y) = (if x = y then empty_table else unit_table n)
| neq_rel n (MFOTL.Var x) (MFOTL.Var y) = (if x = y then empty_table else undefined)
| neq_rel _ _ _ _ = undefined

fun minit0 :: nat ⇒ 'a MFOTL.formula ⇒ 'a mformula where
  minit0 n (MFOTL.Neg φ) = (case φ of
    MFOTL.Eq t1 t2 ⇒ MRel (neq_rel n t1 t2)
```

```
  | MFOTL.Or (MFOTL.Neg φ) ψ ⇒ (if safe_formula ψ ∧ MFOTL.fv ψ ⊆ MFOTL.fv φ
      then MAnd (minit0 n φ) False (minit0 n ψ) ([], [])
      else (case ψ of MFOTL.Neg ψ ⇒ MAnd (minit0 n φ) True (minit0 n ψ) ([], []) | _ ⇒ undefined))
  | _ ⇒ undefined)
| minit0 n (MFOTL.Eq t1 t2) = MRel (eq_rel n t1 t2)
| minit0 n (MFOTL.Pred e ts) = MPred e ts
| minit0 n (MFOTL.Or φ ψ) = MOr (minit0 n φ) (minit0 n ψ) ([], [])
| minit0 n (MFOTL.Exists φ) = MExists (minit0 (Suc n) φ)
| minit0 n (MFOTL.Prev I φ) = MPrev I (minit0 n φ) True [] []
| minit0 n (MFOTL.Next I φ) = MNext I (minit0 n φ) True []
| minit0 n (MFOTL.Since φ I ψ) = (if safe_formula φ
    then MSince True (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    else (case φ of
      MFOTL.Neg φ ⇒ MSince False (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    | _ ⇒ undefined))
| minit0 n (MFOTL.Until φ I ψ) = (if safe_formula φ
    then MUntil True (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    else (case φ of
      MFOTL.Neg φ ⇒ MUntil False (minit0 n φ) I (minit0 n ψ) ([], []) [] []
    | _ ⇒ undefined))

definition minit :: 'a MFOTL.formula ⇒ 'a mstate where
  minit φ = (let n = MFOTL.nfv φ in (|mstate_i = 0, mstate_m = minit0 n φ, mstate_n = n|))

fun mprev_next :: ℐ ⇒ 'a table list ⇒ ts list ⇒ 'a table list × 'a table list × ts list where
  mprev_next I [] ts = ([], [], ts)
| mprev_next I xs [] = ([], xs, [])
| mprev_next I xs [t] = ([], xs, [t])
| mprev_next I (x # xs) (t # t' # ts) = (let (ys, zs) = mprev_next I xs (t' # ts)
    in ((if mem (t' − t) I then x else empty_table) # ys, zs))

fun mbuf2_add :: 'a table list ⇒ 'a table list ⇒ 'a mbuf2 ⇒ 'a mbuf2 where
 mbuf2_add xs' ys' (xs, ys) = (xs @ xs', ys @ ys')

fun mbuf2_take :: ('a table ⇒ 'a table ⇒ 'b) ⇒ 'a mbuf2 ⇒ 'b list × 'a mbuf2 where
  mbuf2_take f (x # xs, y # ys) = (let (zs, buf) = mbuf2_take f (xs, ys) in (f x y # zs, buf))
| mbuf2_take f (xs, ys) = ([], (xs, ys))

fun mbuf2t_take :: ('a table ⇒ 'a table ⇒ ts ⇒ 'b ⇒ 'b) ⇒ 'b ⇒
  'a mbuf2 ⇒ ts list ⇒ 'b × 'a mbuf2 × ts list where
  mbuf2t_take f z (x # xs, y # ys) (t # ts) = mbuf2t_take f (f x y t z) (xs, ys) ts
| mbuf2t_take f z (xs, ys) ts = (z, (xs, ys), ts)

fun match :: 'a MFOTL.trm list ⇒ 'a list ⇒ (nat ⇀ 'a) option where
  match [] [] = Some Map.empty
| match (MFOTL.Const x # ts) (y # ys) = (if x = y then match ts ys else None)
| match (MFOTL.Var x # ts) (y # ys) = (case match ts ys of
      None ⇒ None
    | Some f ⇒ (case f x of
        None ⇒ Some (f(x ↦ y))
      | Some z ⇒ if y = z then Some f else None))
| match _ _ = None

definition update_since :: ℐ ⇒ bool ⇒ 'a table ⇒ 'a table ⇒ ts ⇒
  'a msaux ⇒ 'a table × 'a msaux where
  update_since I pos rel1 rel2 nt aux =
    (let aux = (case [(t, join rel pos rel1). (t, rel) ← aux, nt − t ≤ right I] of
        [] ⇒ [(nt, rel2)]
```

| x # aux' ⇒ (*if fst x = nt then* (*fst x*, *snd x* ∪ *rel2*) # aux' *else* (*nt*, *rel2*) # *x* # aux'))
in (*foldr* (∪) [*rel*. (*t*, *rel*) ← *aux*, *left I* ≤ *nt* − *t*] {}, *aux*))

definition *update_until* :: $\mathcal{I}$ ⇒ *bool* ⇒ *'a table* ⇒ *'a table* ⇒ *ts* ⇒ *'a muaux* ⇒ *'a muaux* **where**
update_until I pos rel1 rel2 nt aux =
(*map* (λ*x*. *case x of* (*t*, *a1*, *a2*) ⇒ (*t*, *if pos then join a1 True rel1 else a1* ∪ *rel1*,
if mem (*nt* − *t*) *I then a2* ∪ *join rel2 pos a1 else a2*)) *aux*) @
[(*nt*, *rel1*, *if left I* = *0 then rel2 else empty_table*)]

fun *eval_until* :: $\mathcal{I}$ ⇒ *ts* ⇒ *'a muaux* ⇒ *'a table list* × *'a muaux* **where**
eval_until I nt [] = ([], [])
| *eval_until I nt* ((*t*, *a1*, *a2*) # *aux*) = (*if t* + *right I* < *nt then*
(*let* (*xs*, *aux*) = *eval_until I nt aux in* (*a2* # *xs*, *aux*)) *else* ([], (*t*, *a1*, *a2*) # *aux*))

primrec *meval* :: *nat* ⇒ *ts* ⇒ *'a MFOTL.database* ⇒ *'a mformula* ⇒ *'a table list* × *'a mformula* **where**
meval n t db (*MRel rel*) = ([*rel*], *MRel rel*)
| *meval n t db* (*MPred e ts*) = ([(λ*f*. *tabulate f 0 n*) ‘ *Option.these*
(*match ts* ‘ (⋃(*e'*, *x*)∈*db*. *if e* = *e'* *then* {*x*} *else* {}))], *MPred e ts*)
| *meval n t db* (*MAnd* φ *pos* ψ *buf*) =
(*let* (*xs*, φ) = *meval n t db* φ; (*ys*, ψ) = *meval n t db* ψ;
(*zs*, *buf*) = *mbuf2_take* (λ*r1 r2*. *join r1 pos r2*) (*mbuf2_add xs ys buf*)
in (*zs*, *MAnd* φ *pos* ψ *buf*))
| *meval n t db* (*MOr* φ ψ *buf*) =
(*let* (*xs*, φ) = *meval n t db* φ; (*ys*, ψ) = *meval n t db* ψ;
(*zs*, *buf*) = *mbuf2_take* (λ*r1 r2*. *r1* ∪ *r2*) (*mbuf2_add xs ys buf*)
in (*zs*, *MOr* φ ψ *buf*))
| *meval n t db* (*MExists* φ) =
(*let* (*xs*, φ) = *meval* (*Suc n*) *t db* φ *in* (*map* (λ*r*. *tl* ‘ *r*) *xs*, *MExists* φ))
| *meval n t db* (*MPrev I* φ *first buf nts*) =
(*let* (*xs*, φ) = *meval n t db* φ;
(*zs*, *buf*, *nts*) = *mprev_next I* (*buf* @ *xs*) (*nts* @ [*t*])
in (*if first then empty_table* # *zs else zs*, *MPrev I* φ *False buf nts*))
| *meval n t db* (*MNext I* φ *first nts*) =
(*let* (*xs*, φ) = *meval n t db* φ;
(*xs*, *first*) = (*case* (*xs*, *first*) *of* (_ # *xs*, *True*) ⇒ (*xs*, *False*) | *a* ⇒ *a*);
(*zs*, _, *nts*) = *mprev_next I xs* (*nts* @ [*t*])
in (*zs*, *MNext I* φ *first nts*))
| *meval n t db* (*MSince pos* φ *I* ψ *buf nts aux*) =
(*let* (*xs*, φ) = *meval n t db* φ; (*ys*, ψ) = *meval n t db* ψ;
((*zs*, *aux*), *buf*, *nts*) = *mbuf2t_take* (λ*r1 r2 t* (*zs*, *aux*).
let (*z*, *aux*) = *update_since I pos r1 r2 t aux*
in (*zs* @ [*z*], *aux*)) ([], *aux*) (*mbuf2_add xs ys buf*) (*nts* @ [*t*])
in (*zs*, *MSince pos* φ *I* ψ *buf nts aux*))
| *meval n t db* (*MUntil pos* φ *I* ψ *buf nts aux*) =
(*let* (*xs*, φ) = *meval n t db* φ; (*ys*, ψ) = *meval n t db* ψ;
(*aux*, *buf*, *nts*) = *mbuf2t_take* (*update_until I pos*) *aux* (*mbuf2_add xs ys buf*) (*nts* @ [*t*]);
(*zs*, *aux*) = *eval_until I* (*case nts of* [] ⇒ *t* | *nt* # _ ⇒ *nt*) *aux*
in (*zs*, *MUntil pos* φ *I* ψ *buf nts aux*))

definition *mstep* :: *'a MFOTL.database* × *ts* ⇒ *'a mstate* ⇒ (*nat* × *'a tuple*) *set* × *'a mstate* **where**
mstep tdb st =
(*let* (*xs*, *m*) = *meval* (*mstate_n st*) (*snd tdb*) (*fst tdb*) (*mstate_m st*)
in (⋃ (*set* (*map* (λ(*i*, *X*). (λ*v*. (*i*, *v*)) ‘ *X*) (*List.enumerate* (*mstate_i st*) *xs*))),
(|*mstate_i* = *mstate_i st* + *length xs*, *mstate_m* = *m*, *mstate_n* = *mstate_n st*|)))

lemma *mstep_alt*: *mstep tdb st* =
(*let* (*xs*, *m*) = *meval* (*mstate_n st*) (*snd tdb*) (*fst tdb*) (*mstate_m st*)
in (⋃(*i*, *X*) ∈ *set* (*List.enumerate* (*mstate_i st*) *xs*). ⋃*v* ∈ *X*. {(*i*,*v*)},

$(\!|mstate\_i = mstate\_i\ st + length\ xs,\ mstate\_m = m,\ mstate\_n = mstate\_n\ st|\!))$
unfolding *mstep_def*
by (*auto split*: *prod.split*)

6.3 Progress

primrec *progress* :: $'a\ MFOTL.trace \Rightarrow 'a\ MFOTL.formula \Rightarrow nat \Rightarrow nat$ **where**
$progress\ \sigma\ (MFOTL.Pred\ e\ ts)\ j = j$
$|\ progress\ \sigma\ (MFOTL.Eq\ t1\ t2)\ j = j$
$|\ progress\ \sigma\ (MFOTL.Neg\ \varphi)\ j = progress\ \sigma\ \varphi\ j$
$|\ progress\ \sigma\ (MFOTL.Or\ \varphi\ \psi)\ j = min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
$|\ progress\ \sigma\ (MFOTL.Exists\ \varphi)\ j = progress\ \sigma\ \varphi\ j$
$|\ progress\ \sigma\ (MFOTL.Prev\ I\ \varphi)\ j = (if\ j = 0\ then\ 0\ else\ min\ (Suc\ (progress\ \sigma\ \varphi\ j))\ j)$
$|\ progress\ \sigma\ (MFOTL.Next\ I\ \varphi)\ j = progress\ \sigma\ \varphi\ j - 1$
$|\ progress\ \sigma\ (MFOTL.Since\ \varphi\ I\ \psi)\ j = min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
$|\ progress\ \sigma\ (MFOTL.Until\ \varphi\ I\ \psi)\ j =$
$Inf\ \{i.\ \forall k.\ k < j \wedge k \leq min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j) \longrightarrow \tau\ \sigma\ i + right\ I \geq \tau\ \sigma\ k\}$

lemma *progress_And*[*simp*]: $progress\ \sigma\ (MFOTL.And\ \varphi\ \psi)\ j = min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
unfolding *MFOTL.And_def* **by** *simp*

lemma *progress_And_Not*[*simp*]: $progress\ \sigma\ (MFOTL.And\_Not\ \varphi\ \psi)\ j = min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
unfolding *MFOTL.And_Not_def* **by** *simp*

lemma *progress_mono*: $j \leq j' \Longrightarrow progress\ \sigma\ \varphi\ j \leq progress\ \sigma\ \varphi\ j'$
proof (*induction* φ)
case ($Until\ \varphi\ I\ \psi$)
then show *?case*
by (*cases right I*)
(*auto dest*: *trans_le_add1*[*OF* τ*_mono*] *intro*!: *cInf_superset_mono*)
qed *auto*

lemma *progress_le*: $progress\ \sigma\ \varphi\ j \leq j$
proof (*induction* φ)
case ($Until\ \varphi\ I\ \psi$)
then show *?case*
by (*cases right I*)
(*auto intro*: *trans_le_add1*[*OF* τ*_mono*] *intro*!: *cInf_lower*)
qed *auto*

lemma *progress_0*[*simp*]: $progress\ \sigma\ \varphi\ 0 = 0$
using *progress_le* **by** *auto*

lemma *progress_ge*: $MFOTL.future\_reach\ \varphi \neq \infty \Longrightarrow \exists j.\ i \leq progress\ \sigma\ \varphi\ j$
proof (*induction* φ *arbitrary*: *i*)
case (*Pred e ts*)
then show *?case* **by** *auto*
next
case (*Eq t1 t2*)
then show *?case* **by** *auto*
next
case ($Neg\ \varphi$)
then show *?case* **by** *simp*
next
case ($Or\ \varphi1\ \varphi2$)
from *Or.prems* **have** $MFOTL.future\_reach\ \varphi1 \neq \infty$
by (*cases* $MFOTL.future\_reach\ \varphi1$) (*auto*)

```
  moreover from Or.prems have MFOTL.future_reach φ2 ≠ ∞
    by (cases MFOTL.future_reach φ2) (auto)
  ultimately obtain j1 j2 where i ≤ progress σ φ1 j1 and i ≤ progress σ φ2 j2
    using Or.IH[of i] by blast
  then have i ≤ progress σ (MFOTL.Or φ1 φ2) (max j1 j2)
    by (cases j1 ≤ j2) (auto elim!: order.trans[OF _ progress_mono])
  then show ?case by blast
next
  case (Exists φ)
  then show ?case by simp
next
  case (Prev I φ)
  from Prev.prems have MFOTL.future_reach φ ≠ ∞
    by (cases MFOTL.future_reach φ) (auto)
  then obtain j where i ≤ progress σ φ j
    using Prev.IH[of i] by blast
  then show ?case by (auto intro!: exI[of _ j] elim!: order.trans[OF _ progress_le])
next
  case (Next I φ)
  from Next.prems have MFOTL.future_reach φ ≠ ∞
    by (cases MFOTL.future_reach φ) (auto)
  then obtain j where Suc i ≤ progress σ φ j
    using Next.IH[of Suc i] by blast
  then show ?case by (auto intro!: exI[of _ j])
next
  case (Since φ1 I φ2)
  from Since.prems have MFOTL.future_reach φ1 ≠ ∞
    by (cases MFOTL.future_reach φ1) (auto)
  moreover from Since.prems have MFOTL.future_reach φ2 ≠ ∞
    by (cases MFOTL.future_reach φ2) (auto)
  ultimately obtain j1 j2 where i ≤ progress σ φ1 j1 and i ≤ progress σ φ2 j2
    using Since.IH[of i] by blast
  then have i ≤ progress σ (MFOTL.Since φ1 I φ2) (max j1 j2)
    by (cases j1 ≤ j2) (auto elim!: order.trans[OF _ progress_mono])
  then show ?case by blast
next
  case (Until φ1 I φ2)
  from Until.prems obtain b where [simp]: right I = enat b
    by (cases right I) (auto)
  obtain i' where i < i' and τ σ i + b + 1 ≤ τ σ i'
    using ex_le_τ[where x=τ σ i + b + 1] by (auto simp add: less_eq_Suc_le)
  then have 1: τ σ i + b < τ σ i' by simp
  from Until.prems have MFOTL.future_reach φ1 ≠ ∞
    by (cases MFOTL.future_reach φ1) (auto)
  moreover from Until.prems have MFOTL.future_reach φ2 ≠ ∞
    by (cases MFOTL.future_reach φ2) (auto)
  ultimately obtain j1 j2 where Suc i' ≤ progress σ φ1 j1 and Suc i' ≤ progress σ φ2 j2
    using Until.IH[of Suc i'] by blast
  then have i ≤ progress σ (MFOTL.Until φ1 I φ2) (max j1 j2)
    unfolding progress.simps
  proof (intro cInf_greatest, goal_cases nonempty greatest)
    case nonempty
    then show ?case
      by (auto simp: trans_le_add1[OF τ_mono] intro!: exI[of _ max j1 j2])
  next
    case (greatest x)
    with ‹i < i'› 1 show ?case
      by (cases j1 ≤ j2)
```

```
    (auto dest!: spec[of _ i'] simp: max_absorb1 max_absorb2 less_eq_Suc_le
      elim: order.trans[OF _ progress_le] order.trans[OF _ progress_mono, rotated]
      dest!: not_le_imp_less[THEN less_imp_le] intro!: less_τD[THEN less_imp_le, of σ i x])
  qed
  then show ?case by blast
qed

lemma cInf_restrict_nat:
  fixes x :: nat
  assumes x ∈ A
  shows Inf A = Inf {y ∈ A. y ≤ x}
  using assms by (auto intro!: antisym intro: cInf_greatest cInf_lower Inf_nat_def1)

lemma progress_time_conv:
  assumes ∀ i<j. τ σ i = τ σ' i
  shows progress σ φ j = progress σ' φ j
proof (induction φ)
  case (Until φ1 I φ2)
  have *: i ≤ j − 1 ⟷ i < j if j ≠ 0 for i
    using that by auto
  with Until show ?case
  proof (cases right I)
    case (enat b)
    then show ?thesis
    proof (cases j)
      case (Suc n)
      with enat * Until show ?thesis
        using assms τ_mono[THEN trans_le_add1]
        by (auto 6 0
          intro!: box_equals[OF arg_cong[where f=Inf]
            cInf_restrict_nat[symmetric, where x=n] cInf_restrict_nat[symmetric, where x=n]])
    qed simp
  qed simp
qed simp_all

lemma Inf_UNIV_nat: (Inf UNIV :: nat) = 0
  by (simp add: cInf_eq_minimum)

lemma progress_prefix_conv:
  assumes prefix_of π σ and prefix_of π σ'
  shows progress σ φ (plen π) = progress σ' φ (plen π)
  using assms by (auto intro: progress_time_conv τ_prefix_conv)

lemma sat_prefix_conv:
  assumes prefix_of π σ and prefix_of π σ' and i < progress σ φ (plen π)
  shows MFOTL.sat σ v i φ ⟷ MFOTL.sat σ' v i φ
using assms(3) proof (induction φ arbitrary: v i)
  case (Pred e ts)
  with Γ_prefix_conv[OF assms(1,2)] show ?case by simp
next
  case (Eq t1 t2)
  show ?case by simp
next
  case (Neg φ)
  then show ?case by simp
next
  case (Or φ1 φ2)
  then show ?case by simp
```

```
next
  case (Exists φ)
  then show ?case by simp
next
  case (Prev I φ)
  with τ_prefix_conv[OF assms(1,2)] show ?case
    by (cases i) (auto split: if_splits)
next
  case (Next I φ)
  then have Suc i < plen π
    by (auto intro: order.strict_trans2[OF _ progress_le[of σ φ]])
  with Next τ_prefix_conv[OF assms(1,2)] show ?case by simp
next
  case (Since φ1 I φ2)
  then have i < plen π
    by (auto elim!: order.strict_trans2[OF _ progress_le])
  with Since τ_prefix_conv[OF assms(1,2)] show ?case by auto
next
  case (Until φ1 I φ2)
  from Until.prems obtain b where [simp]: right I = enat b
    by (cases right I) (auto simp add: Inf_UNIV_nat)
  from Until.prems obtain j where τ σ i + b + 1 ≤ τ σ j
    j ≤ progress σ φ1 (plen π) j ≤ progress σ φ2 (plen π)
    by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of _ i]
      dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 1: k < progress σ φ1 (plen π) and 2: k < progress σ φ2 (plen π)
    if τ σ k ≤ τ σ i + b for k
    using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of σ])+
  have 3: k < plen π if τ σ k ≤ τ σ i + b for k
    using 1[OF that] by (simp add: less_eq_Suc_le order.trans[OF _ progress_le])

  from Until.prems have i < progress σ' (MFOTL.Until φ1 I φ2) (plen π)
    unfolding progress_prefix_conv[OF assms(1,2)] .
  then obtain j where τ σ' i + b + 1 ≤ τ σ' j
    j ≤ progress σ' φ1 (plen π) j ≤ progress σ' φ2 (plen π)
    by atomize_elim (auto 0 4 simp add: less_eq_Suc_le not_le intro: Suc_leI dest: spec[of _ i]
      dest!: le_cInf_iff[THEN iffD1, rotated -1])
  then have 11: k < progress σ φ1 (plen π) and 21: k < progress σ φ2 (plen π)
    if τ σ' k ≤ τ σ' i + b for k
    unfolding progress_prefix_conv[OF assms(1,2)]
    using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of σ'])+
  have 31: k < plen π if τ σ' k ≤ τ σ' i + b for k
    using 11[OF that] by (simp add: less_eq_Suc_le order.trans[OF _ progress_le])

  show ?case unfolding sat.simps
  proof ((intro ex_cong iffI; elim conjE), goal_cases LR RL)
    case (LR j)
    with Until.IH(1)[OF 1] Until.IH(2)[OF 2] τ_prefix_conv[OF assms(1,2) 3] show ?case
      by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF τ_mono, rotated])
  next
    case (RL j)
    with Until.IH(1)[OF 11] Until.IH(2)[OF 21] τ_prefix_conv[OF assms(1,2) 31] show ?case
      by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF τ_mono, rotated])
  qed
qed
```

6.4 Specification

```
definition pprogress :: 'a MFOTL.formula ⇒ 'a MFOTL.prefix ⇒ nat where
  pprogress φ π = (THE n. ∀σ. prefix_of π σ ⟶ progress σ φ (plen π) = n)

lemma pprogress_eq: prefix_of π σ ⟹ pprogress φ π = progress σ φ (plen π)
  unfolding pprogress_def using progress_prefix_conv
  by blast

locale future_bounded_mfotl =
  fixes φ :: 'a MFOTL.formula
  assumes future_bounded: MFOTL.future_reach φ ≠ ∞

sublocale future_bounded_mfotl ⊆ sliceable_timed_progress MFOTL.nfv φ MFOTL.fv φ relevant_events
φ
  λσ v i. MFOTL.sat σ v i φ pprogress φ
proof (unfold_locales, goal_cases)
  case (1 x)
  then show ?case by (simp add: fvi_less_nfv)
next
  case (2 v v' σ i)
  then show ?case by (simp cong: sat_fvi_cong[rule_format])
next
  case (3 v S σ i)
  then show ?case using sat_slice_iff[of v, symmetric] by simp
next
  case (4 π π')
  moreover obtain σ where prefix_of π' σ
    using ex_prefix_of ..
  moreover have prefix_of π σ
    using prefix_of_antimono[OF ‹π ≤ π'› ‹prefix_of π' σ›] .
  ultimately show ?case
    by (simp add: pprogress_eq plen_mono progress_mono)
next
  case (5 σ x)
  obtain j where x ≤ progress σ φ j
    using future_bounded progress_ge by blast
  then have x ≤ pprogress φ (take_prefix j σ)
    by (simp add: pprogress_eq[of _ σ])
  then show ?case by force
next
  case (6 π σ σ' i v)
  then have i < progress σ φ (plen π)
    by (simp add: pprogress_eq)
  with 6 show ?case
    using sat_prefix_conv by blast
next
  case (7 π π')
  then have plen π = plen π'
    by transfer (simp add: list_eq_iff_nth_eq)
  moreover obtain σ σ' where prefix_of π σ prefix_of π' σ'
    using ex_prefix_of by blast+
  moreover have ∀i < plen π. τ σ i = τ σ' i
    using 7 calculation
    by transfer (simp add: list_eq_iff_nth_eq)
  ultimately show ?case
    by (simp add: pprogress_eq progress_time_conv)
qed
```

locale *monitorable_mfotl* =
  fixes φ :: *'a MFOTL.formula*
  assumes *monitorable*: *mmonitorable* φ

sublocale *monitorable_mfotl* $\subseteq$ *future_bounded_mfotl*
  using *monitorable* **by** *unfold_locales* (*simp add*: *mmonitorable_def*)

6.5 Correctness

6.5.1 Invariants

definition *wf_mbuf2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ (*nat* ⇒ *'a table* ⇒ *bool*) ⇒ (*nat* ⇒ *'a table* ⇒ *bool*) ⇒ *'a mbuf2* ⇒ *bool* **where**
  wf_mbuf2 i ja jb P Q buf ⟷ $i \leq ja \wedge i \leq jb \wedge$ (*case buf of* (*xs*, *ys*) ⇒
    list_all2 P [*i*..<*ja*] *xs* ∧ *list_all2 Q* [*i*..<*jb*] *ys*)

definition *wf_mbuf2'* :: *'a MFOTL.trace* ⇒ *nat* ⇒ *nat* ⇒ *'a list set* ⇒
  'a MFOTL.formula ⇒ *'a MFOTL.formula* ⇒ *'a mbuf2* ⇒ *bool* **where**
  wf_mbuf2' σ *j n R* φ ψ *buf* ⟷ *wf_mbuf2* (*min* (*progress* σ φ *j*) (*progress* σ ψ *j*))
    (*progress* σ φ *j*) (*progress* σ ψ *j*)
    (λ*i*. *qtable n* (*MFOTL.fv* φ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) *i* φ))
    (λ*i*. *qtable n* (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) *i* ψ)) *buf*

lemma *wf_mbuf2'_UNIV_alt*: *wf_mbuf2'* σ *j n UNIV* φ ψ *buf* ⟷ (*case buf of* (*xs*, *ys*) ⇒
  list_all2 (λ*i*. *wf_table n* (*MFOTL.fv* φ) (λ*v*. *MFOTL.sat* σ (*map the v*) *i* φ))
    [*min* (*progress* σ φ *j*) (*progress* σ ψ *j*) ..< (*progress* σ φ *j*)] *xs* ∧
  list_all2 (λ*i*. *wf_table n* (*MFOTL.fv* ψ) (λ*v*. *MFOTL.sat* σ (*map the v*) *i* ψ))
    [*min* (*progress* σ φ *j*) (*progress* σ ψ *j*) ..< (*progress* σ ψ *j*)] *ys*)
  unfolding *wf_mbuf2'_def wf_mbuf2_def*
  by (*simp add*: *mem_restr_UNIV*[*THEN eqTrueI*, *abs_def*] *split*: *prod.split*)

definition *wf_ts* :: *'a MFOTL.trace* ⇒ *nat* ⇒ *'a MFOTL.formula* ⇒ *'a MFOTL.formula* ⇒ *ts list* ⇒ *bool* **where**
  wf_ts σ *j* φ ψ *ts* ⟷ *list_all2* (λ*i t*. $t = \tau\ \sigma\ i$) [*min* (*progress* σ φ *j*) (*progress* σ ψ *j*)..<*j*] *ts*

abbreviation *Sincep pos* φ *I* ψ ≡ *MFOTL.Since* (*if pos then* φ *else MFOTL.Neg* φ) *I* ψ

definition *wf_since_aux* :: *'a MFOTL.trace* ⇒ *nat* ⇒ *'a list set* ⇒ *bool* ⇒
    'a MFOTL.formula ⇒ $\mathcal{I}$ ⇒ *'a MFOTL.formula* ⇒ *'a msaux* ⇒ *nat* ⇒ *bool* **where**
  wf_since_aux σ *n R pos* φ *I* ψ *aux ne* ⟷ *sorted_wrt* (λ*x y*. *fst x* > *fst y*) *aux* ∧
    ($\forall$ *t X*. (*t*, *X*) ∈ *set aux* ⟶ *ne* ≠ *0* ∧ $t \leq \tau\ \sigma$ (*ne*−*1*) ∧ $\tau\ \sigma$ (*ne*−*1*) − *t* ≤ *right I* ∧ ($\exists$ *i*. $\tau\ \sigma\ i$ = *t*) ∧
      qtable n (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. *MFOTL.sat* σ (*map the v*) (*ne*−*1*) (*Sincep pos* φ (*point* ($\tau\ \sigma$ (*ne*−*1*) − *t*)) ψ)) *X*) ∧
    ($\forall$ *t*. *ne* ≠ *0* ∧ $t \leq \tau\ \sigma$ (*ne*−*1*) ∧ $\tau\ \sigma$ (*ne*−*1*) − *t* ≤ *right I* ∧ ($\exists$ *i*. $\tau\ \sigma\ i$ = *t*) ⟶
      ($\exists$ *X*. (*t*, *X*) ∈ *set aux*))

lemma *qtable_mem_restr_UNIV*: *qtable n A* (*mem_restr UNIV*) *Q X* = *wf_table n A Q X*
  unfolding *qtable_def* **by** *auto*

lemma *wf_since_aux_UNIV_alt*:
  wf_since_aux σ *n UNIV pos* φ *I* ψ *aux ne* ⟷ *sorted_wrt* (λ*x y*. *fst x* > *fst y*) *aux* ∧
    ($\forall$ *t X*. (*t*, *X*) ∈ *set aux* ⟶ *ne* ≠ *0* ∧ $t \leq \tau\ \sigma$ (*ne*−*1*) ∧ $\tau\ \sigma$ (*ne*−*1*) − *t* ≤ *right I* ∧ ($\exists$ *i*. $\tau\ \sigma\ i$ = *t*) ∧
      wf_table n (*MFOTL.fv* ψ)
        (λ*v*. *MFOTL.sat* σ (*map the v*) (*ne*−*1*) (*Sincep pos* φ (*point* ($\tau\ \sigma$ (*ne*−*1*) − *t*)) ψ)) *X*) ∧
    ($\forall$ *t*. *ne* ≠ *0* ∧ $t \leq \tau\ \sigma$ (*ne*−*1*) ∧ $\tau\ \sigma$ (*ne*−*1*) − *t* ≤ *right I* ∧ ($\exists$ *i*. $\tau\ \sigma\ i$ = *t*) ⟶
      ($\exists$ *X*. (*t*, *X*) ∈ *set aux*))
  unfolding *wf_since_aux_def qtable_mem_restr_UNIV* **..**

definition *wf_until_aux* :: 'a MFOTL.trace ⇒ nat ⇒ 'a list set ⇒ bool ⇒
'a MFOTL.formula ⇒ ℐ ⇒ 'a MFOTL.formula ⇒ 'a muaux ⇒ nat ⇒ bool **where**
wf_until_aux σ n R pos φ I ψ aux ne ⟷ list_all2 (λx i. case x of (t, r1, r2) ⇒ t = τ σ i ∧
qtable n (MFOTL.fv φ) (mem_restr R) (λv. if pos then (∀ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)
else (∃ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)) r1 ∧
qtable n (MFOTL.fv ψ) (mem_restr R) (λv. (∃ j. i ≤ j ∧ j < ne + length aux ∧ mem (τ σ j − τ σ i) I ∧
MFOTL.sat σ (map the v) j ψ ∧
(∀ k∈{i..<j}. if pos then MFOTL.sat σ (map the v) k φ else ¬ MFOTL.sat σ (map the v) k φ))) r2)
aux [ne..<ne+length aux]

lemma *wf_until_aux_UNIV_alt*:
wf_until_aux σ n UNIV pos φ I ψ aux ne ⟷ list_all2 (λx i. case x of (t, r1, r2) ⇒ t = τ σ i ∧
wf_table n (MFOTL.fv φ) (λv. if pos
then (∀ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)
else (∃ k∈{i..<ne+length aux}. MFOTL.sat σ (map the v) k φ)) r1 ∧
wf_table n (MFOTL.fv ψ) (λv. ∃ j. i ≤ j ∧ j < ne + length aux ∧ mem (τ σ j − τ σ i) I ∧
MFOTL.sat σ (map the v) j ψ ∧
(∀ k∈{i..<j}. if pos then MFOTL.sat σ (map the v) k φ else ¬ MFOTL.sat σ (map the v) k φ)) r2)
aux [ne..<ne+length aux]
unfolding wf_until_aux_def qtable_mem_restr_UNIV **..**

inductive *wf_mformula* :: 'a MFOTL.trace ⇒ nat ⇒
nat ⇒ 'a list set ⇒ 'a mformula ⇒ 'a MFOTL.formula ⇒ bool
for σ j **where**
Eq: MFOTL.is_Const t1 ∨ MFOTL.is_Const t2 ⟹
∀ x∈MFOTL.fv_trm t1. x < n ⟹ ∀ x∈MFOTL.fv_trm t2. x < n ⟹
wf_mformula σ j n R (MRel (eq_rel n t1 t2)) (MFOTL.Eq t1 t2)
| neq_Const: φ = MRel (neq_rel n (MFOTL.Const x) (MFOTL.Const y)) ⟹
wf_mformula σ j n R φ (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))
| neq_Var: x < n ⟹
wf_mformula σ j n R (MRel empty_table) (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var x)))
| Pred: ∀ x∈MFOTL.fv (MFOTL.Pred e ts). x < n ⟹
wf_mformula σ j n R (MPred e ts) (MFOTL.Pred e ts)
| And: wf_mformula σ j n R φ φ' ⟹ wf_mformula σ j n R ψ ψ' ⟹
if pos then χ = MFOTL.And φ' ψ' ∧ ¬ (safe_formula (MFOTL.Neg ψ') ∧ MFOTL.fv ψ' ⊆ MFOTL.fv φ')
else χ = MFOTL.And_Not φ' ψ' ∧ safe_formula ψ' ∧ MFOTL.fv ψ' ⊆ MFOTL.fv φ' ⟹
wf_mbuf2' σ j n R φ' ψ' buf ⟹
wf_mformula σ j n R (MAnd φ pos ψ buf) χ
| Or: wf_mformula σ j n R φ φ' ⟹ wf_mformula σ j n R ψ ψ' ⟹
MFOTL.fv φ' = MFOTL.fv ψ' ⟹
wf_mbuf2' σ j n R φ' ψ' buf ⟹
wf_mformula σ j n R (MOr φ ψ buf) (MFOTL.Or φ' ψ')
| Exists: wf_mformula σ j (Suc n) (lift_envs R) φ φ' ⟹
wf_mformula σ j n R (MExists φ) (MFOTL.Exists φ')
| Prev: wf_mformula σ j n R φ φ' ⟹
first ⟷ j = 0 ⟹
list_all2 (λi. qtable n (MFOTL.fv φ') (mem_restr R) (λv. MFOTL.sat σ (map the v) i φ'))
[min (progress σ φ' j) (j−1)..<progress σ φ' j] buf ⟹
list_all2 (λi t. t = τ σ i) [min (progress σ φ' j) (j−1)..<j] nts ⟹
wf_mformula σ j n R (MPrev I φ first buf nts) (MFOTL.Prev I φ')

| *Next*: $wf\_mformula\ \sigma\ j\ n\ R\ \varphi\ \varphi' \Longrightarrow$
  $first \longleftrightarrow progress\ \sigma\ \varphi'\ j = 0 \Longrightarrow$
  $list\_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [progress\ \sigma\ \varphi'\ j - 1..<j]\ nts \Longrightarrow$
  $wf\_mformula\ \sigma\ j\ n\ R\ (MNext\ I\ \varphi\ first\ nts)\ (MFOTL.Next\ I\ \varphi')$
| *Since*: $wf\_mformula\ \sigma\ j\ n\ R\ \varphi\ \varphi' \Longrightarrow wf\_mformula\ \sigma\ j\ n\ R\ \psi\ \psi' \Longrightarrow$
  $if\ pos\ then\ \varphi'' = \varphi'\ else\ \varphi'' = MFOTL.Neg\ \varphi' \Longrightarrow$
  $safe\_formula\ \varphi'' = pos \Longrightarrow$
  $MFOTL.fv\ \varphi' \subseteq MFOTL.fv\ \psi' \Longrightarrow$
  $wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi'\ \psi'\ buf \Longrightarrow$
  $wf\_ts\ \sigma\ j\ \varphi'\ \psi'\ nts \Longrightarrow$
  $wf\_since\_aux\ \sigma\ n\ R\ pos\ \varphi'\ I\ \psi'\ aux\ (progress\ \sigma\ (MFOTL.Since\ \varphi''\ I\ \psi')\ j) \Longrightarrow$
  $wf\_mformula\ \sigma\ j\ n\ R\ (MSince\ pos\ \varphi\ I\ \psi\ buf\ nts\ aux)\ (MFOTL.Since\ \varphi''\ I\ \psi')$
| *Until*: $wf\_mformula\ \sigma\ j\ n\ R\ \varphi\ \varphi' \Longrightarrow wf\_mformula\ \sigma\ j\ n\ R\ \psi\ \psi' \Longrightarrow$
  $if\ pos\ then\ \varphi'' = \varphi'\ else\ \varphi'' = MFOTL.Neg\ \varphi' \Longrightarrow$
  $safe\_formula\ \varphi'' = pos \Longrightarrow$
  $MFOTL.fv\ \varphi' \subseteq MFOTL.fv\ \psi' \Longrightarrow$
  $wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi'\ \psi'\ buf \Longrightarrow$
  $wf\_ts\ \sigma\ j\ \varphi'\ \psi'\ nts \Longrightarrow$
  $wf\_until\_aux\ \sigma\ n\ R\ pos\ \varphi'\ I\ \psi'\ aux\ (progress\ \sigma\ (MFOTL.Until\ \varphi''\ I\ \psi')\ j) \Longrightarrow$
  $progress\ \sigma\ (MFOTL.Until\ \varphi''\ I\ \psi')\ j + length\ aux = min\ (progress\ \sigma\ \varphi'\ j)\ (progress\ \sigma\ \psi'\ j) \Longrightarrow$
  $wf\_mformula\ \sigma\ j\ n\ R\ (MUntil\ pos\ \varphi\ I\ \psi\ buf\ nts\ aux)\ (MFOTL.Until\ \varphi''\ I\ \psi')$

definition $wf\_mstate :: 'a\ MFOTL.formula \Rightarrow 'a\ MFOTL.prefix \Rightarrow 'a\ list\ set \Rightarrow 'a\ mstate \Rightarrow bool$ **where**
  $wf\_mstate\ \varphi\ \pi\ R\ st \longleftrightarrow mstate\_n\ st = MFOTL.nfv\ \varphi \wedge (\forall \sigma.\ prefix\_of\ \pi\ \sigma \longrightarrow$
  $mstate\_i\ st = progress\ \sigma\ \varphi\ (plen\ \pi) \wedge$
  $wf\_mformula\ \sigma\ (plen\ \pi)\ (mstate\_n\ st)\ R\ (mstate\_m\ st)\ \varphi)$

6.5.2 Initialisation

lemma $minit0\_And$: $\neg (safe\_formula\ (MFOTL.Neg\ \psi) \wedge MFOTL.fv\ \psi \subseteq MFOTL.fv\ \varphi) \Longrightarrow$
  $minit0\ n\ (MFOTL.And\ \varphi\ \psi) = MAnd\ (minit0\ n\ \varphi)\ True\ (minit0\ n\ \psi)\ ([], [])$
  unfolding *MFOTL.And_def* **by** *simp*

lemma $minit0\_And\_Not$: $safe\_formula\ \psi \wedge MFOTL.fv\ \psi \subseteq MFOTL.fv\ \varphi \Longrightarrow$
  $minit0\ n\ (MFOTL.And\_Not\ \varphi\ \psi) = (MAnd\ (minit0\ n\ \varphi)\ False\ (minit0\ n\ \psi)\ ([], []))$
  unfolding *MFOTL.And_Not_def MFOTL.is_Neg_def* **by** (*simp split*: *formula.split*)

lemma $wf\_mbuf2'\_0$: $wf\_mbuf2'\ \sigma\ 0\ n\ R\ \varphi\ \psi\ ([], [])$
  unfolding *wf_mbuf2'_def wf_mbuf2_def* **by** *simp*

lemma wf_ts_0: $wf\_ts\ \sigma\ 0\ \varphi\ \psi\ []$
  unfolding *wf_ts_def* **by** *simp*

lemma $wf\_since\_aux\_Nil$: $wf\_since\_aux\ \sigma\ n\ R\ pos\ \varphi'\ I\ \psi'\ []\ 0$
  unfolding *wf_since_aux_def* **by** *simp*

lemma $wf\_until\_aux\_Nil$: $wf\_until\_aux\ \sigma\ n\ R\ pos\ \varphi'\ I\ \psi'\ []\ 0$
  unfolding *wf_until_aux_def* **by** *simp*

lemma wf_minit0: $safe\_formula\ \varphi \Longrightarrow \forall x \in MFOTL.fv\ \varphi.\ x < n \Longrightarrow$
  $wf\_mformula\ \sigma\ 0\ n\ R\ (minit0\ n\ \varphi)\ \varphi$
  by (*induction arbitrary*: *n R rule*: *safe_formula_induct*)
   (*auto simp add*: *minit0_And fvi_And minit0_And_Not fvi_And_Not*
    intro!: *wf_mformula.intros wf_mbuf2'_0 wf_ts_0 wf_since_aux_Nil wf_until_aux_Nil*
    dest: *fvi_Suc_bound*)

lemma wf_mstate_minit: $safe\_formula\ \varphi \Longrightarrow wf\_mstate\ \varphi\ pnil\ R\ (minit\ \varphi)$
  unfolding *wf_mstate_def minit_def Let_def*
  by (*auto intro*!: *wf_minit0 fvi_less_nfv*)

6.5.3 Evaluation

lemma match_wf_tuple: Some f = match ts xs ⟹ wf_tuple n (⋃ t∈set ts. MFOTL.fv_trm t) (tabulate f 0 n)
  by (induction ts xs arbitrary: f rule: match.induct)
    (fastforce simp: wf_tuple_def split: if_splits option.splits)+

lemma match_fvi_trm_None: Some f = match ts xs ⟹ ∀ t∈set ts. x ∉ MFOTL.fv_trm t ⟹ f x = None
  by (induction ts xs arbitrary: f rule: match.induct) (auto split: if_splits option.splits)

lemma match_fvi_trm_Some: Some f = match ts xs ⟹ t ∈ set ts ⟹ x ∈ MFOTL.fv_trm t ⟹ f x ≠ None
  by (induction ts xs arbitrary: f rule: match.induct) (auto split: if_splits option.splits)

lemma match_eval_trm: ∀ t∈set ts. ∀ i∈MFOTL.fv_trm t. i < n ⟹ Some f = match ts xs ⟹
    map (MFOTL.eval_trm (tabulate (λi. the (f i)) 0 n)) ts = xs
proof (induction ts xs arbitrary: f rule: match.induct)
  case (3 x ts y ys)
  from 3(1)[symmetric] 3(2,3) **show** ?case
    by (auto 0 3 dest: match_fvi_trm_Some sym split: option.splits if_splits intro!: eval_trm_cong)
qed (auto split: if_splits)

lemma wf_tuple_tabulate_Some: wf_tuple n A (tabulate f 0 n) ⟹ x ∈ A ⟹ x < n ⟹ ∃ y. f x = Some y
  unfolding wf_tuple_def **by** auto

lemma ex_match: wf_tuple n (⋃ t∈set ts. MFOTL.fv_trm t) v ⟹ ∀ t∈set ts. ∀ x∈MFOTL.fv_trm t. x < n ⟹
    ∃ f. match ts (map (MFOTL.eval_trm (map the v)) ts) = Some f ∧ v = tabulate f 0 n
proof (induction ts map (MFOTL.eval_trm (map the v)) ts arbitrary: v rule: match.induct)
  case (3 x ts y ys)
  then show ?case
  proof (cases x ∈ (⋃ t∈set ts. MFOTL.fv_trm t))
    case True
    with 3 **show** ?thesis
      by (auto simp: insert_absorb dest!: wf_tuple_tabulate_Some meta_spec[of _ v])
  next
    case False
    with 3(3,4) **have**
      ∗: map (MFOTL.eval_trm (map the v)) ts = map (MFOTL.eval_trm (map the (v[x := None]))) ts
      by (auto simp: wf_tuple_def nth_list_update intro!: eval_trm_cong)
    from False 3(2−4) **obtain** f **where**
      match ts (map (MFOTL.eval_trm (map the v)) ts) = Some f v[x := None] = tabulate f 0 n
      unfolding ∗
      by (atomize_elim, intro 3(1)[of v[x := None]])
        (auto simp: wf_tuple_def nth_list_update intro!: eval_trm_cong)
    moreover from False this **have** f x = None length v = n
      by (auto dest: match_fvi_trm_None[OF sym] arg_cong[of _ _ length])
    ultimately show ?thesis **using** 3(3)
      by (auto simp: list_eq_iff_nth_eq wf_tuple_def)
  qed
qed (auto simp: wf_tuple_def intro: nth_equalityI)

lemma eq_rel_eval_trm: v ∈ eq_rel n t1 t2 ⟹ MFOTL.is_Const t1 ∨ MFOTL.is_Const t2 ⟹
  ∀ x∈MFOTL.fv_trm t1 ∪ MFOTL.fv_trm t2. x < n ⟹
  MFOTL.eval_trm (map the v) t1 = MFOTL.eval_trm (map the v) t2
  by (cases t1; cases t2) (simp_all add: singleton_table_def split: if_splits)

lemma *in_eq_rel*: *wf_tuple n* (*MFOTL.fv_trm t1* ∪ *MFOTL.fv_trm t2*) *v* ⟹
MFOTL.is_Const t1 ∨ *MFOTL.is_Const t2* ⟹
MFOTL.eval_trm (*map the v*) *t1* = *MFOTL.eval_trm* (*map the v*) *t2* ⟹
v ∈ *eq_rel n t1 t2*
by (*cases t1*; *cases t2*)
(*auto simp*: *singleton_table_def wf_tuple_def unit_table_def intro*!: *nth_equalityI split*: *if_splits*)

lemma *table_eq_rel*: *MFOTL.is_Const t1* ∨ *MFOTL.is_Const t2* ⟹
table n (*MFOTL.fv_trm t1* ∪ *MFOTL.fv_trm t2*) (*eq_rel n t1 t2*)
by (*cases t1*; *cases t2*; *simp*)

lemma *wf_tuple_Suc_fviD*: *wf_tuple* (*Suc n*) (*MFOTL.fvi b φ*) *v* ⟹ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) *φ*) (*tl v*)
unfolding *wf_tuple_def* **by** (*simp add*: *fvi_Suc nth_tl*)

lemma *table_fvi_tl*: *table* (*Suc n*) (*MFOTL.fvi b φ*) *X* ⟹ *table n* (*MFOTL.fvi* (*Suc b*) *φ*) (*tl ‘ X*)
unfolding *table_def* **by** (*auto intro*: *wf_tuple_Suc_fviD*)

lemma *wf_tuple_Suc_fvi_SomeI*: *0* ∈ *MFOTL.fvi b φ* ⟹ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) *φ*) *v* ⟹
wf_tuple (*Suc n*) (*MFOTL.fvi b φ*) (*Some x* # *v*)
unfolding *wf_tuple_def*
by (*auto simp*: *fvi_Suc less_Suc_eq_0_disj*)

lemma *wf_tuple_Suc_fvi_NoneI*: *0* ∉ *MFOTL.fvi b φ* ⟹ *wf_tuple n* (*MFOTL.fvi* (*Suc b*) *φ*) *v* ⟹
wf_tuple (*Suc n*) (*MFOTL.fvi b φ*) (*None* # *v*)
unfolding *wf_tuple_def*
by (*auto simp*: *fvi_Suc less_Suc_eq_0_disj*)

lemma *qtable_project_fv*: *qtable* (*Suc n*) (*fv φ*) (*mem_restr* (*lift_envs R*)) *P X* ⟹
qtable n (*MFOTL.fvi* (*Suc 0*) *φ*) (*mem_restr R*)
(*λv*. ∃ *x*. *P* ((*if 0* ∈ *fv φ then Some x else None*) # *v*)) (*tl ‘ X*)
using *neq0_conv* **by** (*fastforce simp*: *image_iff Bex_def fvi_Suc elim*!: *qtable_cong dest*!: *qtable_project*)

lemma *mprev*: *mprev_next I xs ts* = (*ys*, *xs′*, *ts′*) ⟹
list_all2 P [*i*..<*j′*] *xs* ⟹ *list_all2* (*λi t*. *t* = *τ σ i*) [*i*..<*j*] *ts* ⟹ *i* ≤ *j′* ⟹ *i* < *j* ⟹
list_all2 (*λi X*. *if mem* (*τ σ* (*Suc i*) − *τ σ i*) *I then P i X else X* = *empty_table*)
[*i*..<*min j′* (*j*−*1*)] *ys* ∧
list_all2 P [*min j′* (*j*−*1*)..<*j′*] *xs′* ∧
list_all2 (*λi t*. *t* = *τ σ i*) [*min j′* (*j*−*1*)..<*j*] *ts′*
proof (*induction I xs ts arbitrary*: *i ys xs′ ts′ rule*: *mprev_next.induct*)
case (*1 I ts*)
then have *min j′* (*j*−*1*) = *i* **by** *auto*
with *1* **show** *?case* **by** *auto*
next
case (*3 I v v′ t*)
then have *min j′* (*j*−*1*) = *i* **by** (*auto simp*: *list_all2_Cons2 upt_eq_Cons_conv*)
with *3* **show** *?case* **by** *auto*
next
case (*4 I x xs t t′ ts*)
from *4*(*1*)[*of tl ys xs′ ts′ Suc i*] *4*(*2*−*6*) **show** *?case*
by (*auto simp add*: *list_all2_Cons2 upt_eq_Cons_conv Suc_less_eq2*
elim!: *list.rel_mono_strong split*: *prod.splits if_splits*)
qed *simp*

lemma *mnext*: *mprev_next I xs ts* = (*ys*, *xs′*, *ts′*) ⟹
list_all2 P [*Suc i*..<*j′*] *xs* ⟹ *list_all2* (*λi t*. *t* = *τ σ i*) [*i*..<*j*] *ts* ⟹ *Suc i* ≤ *j′* ⟹ *i* < *j* ⟹
list_all2 (*λi X*. *if mem* (*τ σ* (*Suc i*) − *τ σ i*) *I then P* (*Suc i*) *X else X* = *empty_table*)
[*i*..<*min* (*j′*−*1*) (*j*−*1*)] *ys* ∧

list_all2 P [*Suc* (*min* (*j*′−*1*) (*j*−*1*))..<*j*′] *xs*′ ∧
list_all2 (*λi t. t* = *τ σ i*) [*min* (*j*′−*1*) (*j*−*1*)..<*j*] *ts*′
proof (*induction I xs ts arbitrary*: *i ys xs*′ *ts*′ *rule*: *mprev_next.induct*)
case (*1 I ts*)
then have *min* (*j*′ − *1*) (*j*−*1*) = *i* **by** *auto*
with *1* **show** *?case* **by** *auto*
next
case (*3 I v v*′ *t*)
then have *min* (*j*′ − *1*) (*j*−*1*) = *i* **by** (*auto simp*: *list_all2_Cons2 upt_eq_Cons_conv*)
with *3* **show** *?case* **by** *auto*
next
case (*4 I x xs t t*′ *ts*)
from *4*(*1*)[*of tl ys xs*′ *ts*′ *Suc i*] *4*(*2*−*6*) **show** *?case*
by (*auto simp add*: *list_all2_Cons2 upt_eq_Cons_conv Suc_less_eq2*
elim!: *list.rel_mono_strong split*: *prod.splits if_splits*)
qed *simp*

lemma *in_foldr_UnI*: *x* ∈ *A* ⟹ *A* ∈ *set xs* ⟹ *x* ∈ *foldr* (∪) *xs* {}
by (*induction xs*) *auto*

lemma *in_foldr_UnE*: *x* ∈ *foldr* (∪) *xs* {} ⟹ (⋀*A*. *A* ∈ *set xs* ⟹ *x* ∈ *A* ⟹ *P*) ⟹ *P*
by (*induction xs*) *auto*

lemma *sat_the_restrict*: *fv φ* ⊆ *A* ⟹ *MFOTL.sat σ* (*map the* (*restrict A v*)) *i φ* = *MFOTL.sat σ* (*map the v*) *i φ*
by (*rule sat_fvi_cong*) (*auto intro*!: *map_the_restrict*)

lemma *update_since*:
assumes *pre*: *wf_since_aux σ n R pos φ I ψ aux ne*
and *qtable1*: *qtable n* (*MFOTL.fv φ*) (*mem_restr R*) (*λv. MFOTL.sat σ* (*map the v*) *ne φ*) *rel1*
and *qtable2*: *qtable n* (*MFOTL.fv ψ*) (*mem_restr R*) (*λv. MFOTL.sat σ* (*map the v*) *ne ψ*) *rel2*
and *result_eq*: (*rel*, *aux*′) = *update_since I pos rel1 rel2* (*τ σ ne*) *aux*
and *fvi_subset*: *MFOTL.fv φ* ⊆ *MFOTL.fv ψ*
shows *wf_since_aux σ n R pos φ I ψ aux*′ (*Suc ne*)
and *qtable n* (*MFOTL.fv ψ*) (*mem_restr R*) (*λv. MFOTL.sat σ* (*map the v*) *ne* (*Sincep pos φ I ψ*)) *rel*
proof −
let *?wf_tuple* = *λv. wf_tuple n* (*MFOTL.fv ψ*) *v*
note *sat.simps*[*simp del*]

define *aux0* **where** *aux0* = [(*t*, *join rel pos rel1*). (*t*, *rel*) ← *aux*, *τ σ ne* − *t* ≤ *right I*]
have *sorted_aux0*: *sorted_wrt* (*λx y. fst x* > *fst y*) *aux0*
using *pre*[*unfolded wf_since_aux_def*, *THEN conjunct1*]
unfolding *aux0_def*
by (*induction aux*) (*auto simp add*: *sorted_wrt_append*)
have *in_aux0_1*: (*t*, *X*) ∈ *set aux0* ⟹ *ne* ≠ *0* ∧ *t* ≤ *τ σ* (*ne*−*1*) ∧ *τ σ ne* − *t* ≤ *right I* ∧
(∃ *i*. *τ σ i* = *t*) ∧
qtable n (*MFOTL.fv ψ*) (*mem_restr R*) (*λv.* (*MFOTL.sat σ* (*map the v*) (*ne*−*1*) (*Sincep pos φ* (*point* (*τ σ* (*ne*−*1*) − *t*)) *ψ*) ∧
(*if pos then MFOTL.sat σ* (*map the v*) *ne φ else* ¬ *MFOTL.sat σ* (*map the v*) *ne φ*))) *X* **for** *t X*
unfolding *aux0_def* **using** *fvi_subset*
by (*auto 0 1 elim*!: *qtable_join*[*OF _ qtable1*] *simp*: *sat_the_restrict*
dest!: *assms*(*1*)[*unfolded wf_since_aux_def*, *THEN conjunct2*, *THEN conjunct1*, *rule_format*])
then have *in_aux0_le_τ*: (*t*, *X*) ∈ *set aux0* ⟹ *t* ≤ *τ σ ne* **for** *t X*
by (*meson τ_mono diff_le_self le_trans*)
have *in_aux0_2*: *ne* ≠ *0* ⟹ *t* ≤ *τ σ* (*ne*−*1*) ⟹ *τ σ ne* − *t* ≤ *right I* ⟹ ∃ *i*. *τ σ i* = *t* ⟹
∃ *X*. (*t*, *X*) ∈ *set aux0* **for** *t*
proof −

fix t
assume $ne \neq 0$ $t \leq \tau\ \sigma\ (ne-1)$ $\tau\ \sigma\ ne - t \leq right\ I$ $\exists i.\ \tau\ \sigma\ i = t$
then obtain X **where** $(t, X) \in set\ aux$
by (*atomize_elim*, *intro assms*(*1*)[*unfolded wf_since_aux_def*, *THEN conjunct2*, *THEN conjunct2*, *rule_format*])
(*auto simp*: *gr0_conv_Suc elim*!: *order_trans*[*rotated*] *intro*!: *diff_le_mono* τ*_mono*)
with ‹$\tau\ \sigma\ ne - t \leq right\ I$› **have** $(t, join\ X\ pos\ rel1) \in set\ aux0$
unfolding *aux0_def* **by** (*auto elim*!: *bexI*[*rotated*] *intro*!: *exI*[*of _ X*])
then show $\exists X.\ (t, X) \in set\ aux0$
by *blast*
qed
have *aux0_Nil*: $aux0 = [] \Longrightarrow ne = 0 \lor ne \neq 0 \land (\forall t.\ t \leq \tau\ \sigma\ (ne-1) \land \tau\ \sigma\ ne - t \leq right\ I \longrightarrow$
$(\nexists i.\ \tau\ \sigma\ i = t))$
using *in_aux0_2* **by** (*cases* $ne = 0$) (*auto*)

have *aux'_eq*: $aux' = ($*case aux0 of*
$[] \Rightarrow [(\tau\ \sigma\ ne, rel2)]$
$\mid x\ \#\ aux' \Rightarrow (if\ fst\ x = \tau\ \sigma\ ne\ then\ (fst\ x, snd\ x \cup rel2)\ \#\ aux'\ else\ (\tau\ \sigma\ ne, rel2)\ \#\ x\ \#\ aux'))$
using *result_eq* **unfolding** *aux0_def update_since_def Let_def* **by** *simp*
have *sorted_aux'*: *sorted_wrt* $(\lambda x\ y.\ fst\ x > fst\ y)$ *aux'*
unfolding *aux'_eq*
using *sorted_aux0 in_aux0_le_*τ **by** (*cases aux0*) (*fastforce*)+
have *in_aux'_1*: $t \leq \tau\ \sigma\ ne \land \tau\ \sigma\ ne - t \leq right\ I \land (\exists i.\ \tau\ \sigma\ i = t) \land$
qtable n (*MFOTL.fv* ψ) (*mem_restr R*) ($\lambda v.$ *MFOTL.sat* σ (*map the v*) *ne* (*Sincep pos* φ (*point* (τ σ $ne - t$)) ψ)) X
if *aux'*: $(t, X) \in set\ aux'$ **for** $t\ X$
proof (*cases aux0*)
case *Nil*
with *aux'* **show** *?thesis*
unfolding *aux'_eq* **using** *qtable2 aux0_Nil*
by (*auto simp*: *zero_enat_def*[*symmetric*] *sat_Since_rec*[**where** i=*ne*]
dest: *spec*[*of _* $\tau\ \sigma\ (ne-1)$] *elim*!: *qtable_cong*[*OF _ refl*])
next
case (*Cons a as*)
show *?thesis*
proof (*cases* $t = \tau\ \sigma\ ne$)
case *t*: *True*
show *?thesis*
proof (*cases* $fst\ a = \tau\ \sigma\ ne$)
case *True*
with *aux' Cons t* **have** $X = snd\ a \cup rel2$
unfolding *aux'_eq* **using** *sorted_aux0* **by** *auto*
moreover from *in_aux0_1*[*of fst a snd a*] *Cons* **have** $ne \neq 0$
$fst\ a \leq \tau\ \sigma\ (ne - 1)$ $\tau\ \sigma\ ne - fst\ a \leq right\ I$ $\exists i.\ \tau\ \sigma\ i = fst\ a$
qtable n (*fv* ψ) (*mem_restr R*) ($\lambda v.$ *MFOTL.sat* σ (*map the v*) $(ne - 1)$
(*Sincep pos* φ (*point* $(\tau\ \sigma\ (ne - 1) - fst\ a)$) ψ) $\land$ (*if pos then MFOTL.sat* σ (*map the v*) *ne* φ
else $\neg$ *MFOTL.sat* σ (*map the v*) *ne* φ)) (*snd a*)
by *auto*
ultimately show *?thesis* **using** *qtable2 t True*
by (*auto simp*: *sat_Since_rec*[**where** i=*ne*] *sat.simps*(*3*) *elim*!: *qtable_union*)
next
case *False*
with *aux' Cons t* **have** $X = rel2$
unfolding *aux'_eq* **using** *sorted_aux0 in_aux0_le_*τ[*of fst a snd a*] **by** *auto*
with *aux' Cons t False* **show** *?thesis*
unfolding *aux'_eq* **using** *qtable2 in_aux0_2*[*of* $\tau\ \sigma\ (ne-1)$] *in_aux0_le_*τ[*of fst a snd a*] *sorted_aux0*
by (*auto simp*: *sat_Since_rec*[**where** i=*ne*] *sat.simps*(*3*) *zero_enat_def*[*symmetric*] *enat_0_iff*

not_le
elim!: *qtable_cong*[*OF* _ *refl*] *dest*!: *le_τ_less meta_mp*)
qed
next
case *False*
with *aux′ Cons* **have** (*t*, *X*) ∈ *set aux0*
unfolding *aux′_eq* **by** (*auto split*: *if_splits*)
then have *ne* ≠ *0 t* ≤ *τ σ* (*ne* − *1*) *τ σ ne* − *t* ≤ *right I* ∃ *i*. *τ σ i* = *t*
qtable n (*fv ψ*) (*mem_restr R*) (*λv*. *MFOTL.sat σ* (*map the v*) (*ne* − *1*) (*Sincep pos φ* (*point* (*τ σ* (*ne* − *1*) − *t*)) *ψ*) ∧
(*if pos then MFOTL.sat σ* (*map the v*) *ne φ else* ¬ *MFOTL.sat σ* (*map the v*) *ne φ*)) *X*
using *in_aux0_1* **by** *blast*+
with *False aux′ Cons* **show** *?thesis*
unfolding *aux′_eq* **using** *qtable2*
by (*fastforce simp*: *sat_Since_rec*[**where** *i*=*ne*] *sat.simps*(*3*)
diff_diff_right[**where** *i*=*τ σ ne* **and** *j*=*τ σ* _ + *τ σ ne* **and** *k*=*τ σ* (*ne* − *1*),
OF trans_le_add2, *simplified*] *elim*!: *qtable_cong*[*OF* _ *refl*] *order_trans dest*: *le_τ_less*)
qed
qed

have *in_aux′_2*: ∃ *X*. (*t*, *X*) ∈ *set aux′* **if** *t* ≤ *τ σ ne τ σ ne* − *t* ≤ *right I* ∃ *i*. *τ σ i* = *t* **for** *t*
proof (*cases t* = *τ σ ne*)
case *True*
then show *?thesis*
proof (*cases aux0*)
case *Nil*
with *True* **show** *?thesis* **unfolding** *aux′_eq* **by** *simp*
next
case (*Cons a as*)
with *True* **show** *?thesis* **unfolding** *aux′_eq*
by (*cases fst a* = *τ σ ne*) *auto*
qed
next
case *False*
with *that* **have** *ne* ≠ *0*
using *le_τ_less neq0_conv* **by** *blast*
moreover from *False that* **have** *t* ≤ *τ σ* (*ne*−*1*)
by (*metis One_nat_def Suc_leI Suc_pred τ_mono diff_is_0_eq′ order.antisym neq0_conv not_le*)
ultimately obtain *X* **where** (*t*, *X*) ∈ *set aux0* **using** ‹*τ σ ne* − *t* ≤ *right I*› ‹∃ *i*. *τ σ i* = *t*›
by *atomize_elim* (*auto intro*!: *in_aux0_2*)
then show *?thesis* **unfolding** *aux′_eq* **using** *False*
by (*auto intro*: *exI*[*of* _ *X*] *split*: *list.split*)
qed

show *wf_since_aux σ n R pos φ I ψ aux′* (*Suc ne*)
unfolding *wf_since_aux_def*
by (*auto dest*: *in_aux′_1 intro*: *sorted_aux′ in_aux′_2*)

have *rel_eq*: *rel* = *foldr* (∪) [*rel*. (*t*, *rel*) ← *aux′*, *left I* ≤ *τ σ ne* − *t*] {}
unfolding *aux′_eq aux0_def*
using *result_eq* **by** (*simp add*: *update_since_def Let_def*)
have *rel_alt*: *rel* = (⋃ (*t*, *rel*) ∈ *set aux′*. *if left I* ≤ *τ σ ne* − *t then rel else empty_table*)
unfolding *rel_eq*
by (*auto elim*!: *in_foldr_UnE bexI*[*rotated*] *intro*!: *in_foldr_UnI*)
show *qtable n* (*fv ψ*) (*mem_restr R*) (*λv*. *MFOTL.sat σ* (*map the v*) *ne* (*Sincep pos φ I ψ*)) *rel*
unfolding *rel_alt*
proof (*rule qtable_Union*[**where** *Qi*=*λ*(*t*, *X*) *v*.
left I ≤ *τ σ ne* − *t* ∧ *MFOTL.sat σ* (*map the v*) *ne* (*Sincep pos φ* (*point* (*τ σ ne* − *t*)) *ψ*)],

```
    goal_cases finite qtable equiv)
    case (equiv v)
    show ?case
    proof (rule iffI, erule sat_Since_point, goal_cases left right)
      case (left j)
      then show ?case using in_aux'_2[of τ σ j, OF _ _ exI, OF _ _ refl] by auto
    next
      case right
      then show ?case by (auto elim!: sat_Since_pointD dest: in_aux'_1)
    qed
  qed (auto dest!: in_aux'_1 intro!: qtable_empty)
qed

lemma length_update_until: length (update_until pos I rel1 rel2 nt aux) = Suc (length aux)
  unfolding update_until_def by simp

lemma wf_update_until:
  assumes pre: wf_until_aux σ n R pos φ I ψ aux ne
    and qtable1: qtable n (MFOTL.fv φ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne + length
aux) φ) rel1
    and qtable2: qtable n (MFOTL.fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne + length
aux) ψ) rel2
    and fvi_subset: MFOTL.fv φ ⊆ MFOTL.fv ψ
  shows wf_until_aux σ n R pos φ I ψ (update_until I pos rel1 rel2 (τ σ (ne + length aux)) aux) ne
  unfolding wf_until_aux_def length_update_until
  unfolding update_until_def list.rel_map add_Suc_right upt.simps eqTrueI[OF le_add1] if_True
proof (rule list_all2_appendI, unfold list.rel_map, goal_cases old new)
  case old
  show ?case
  proof (rule list.rel_mono_strong[OF assms(1)[unfolded wf_until_aux_def]]; safe, goal_cases mono1
mono2)
    case (mono1 i X Y v)
    then show ?case
      by (fastforce simp: sat_the_restrict less_Suc_eq
        elim!: qtable_join[OF _ qtable1] qtable_union[OF _ qtable1])
  next
    case (mono2 i X Y v)
    then show ?case using fvi_subset
      by (auto 0 3 simp: sat_the_restrict less_Suc_eq split: if_splits
        elim!: qtable_union[OF _ qtable_join_fixed[OF qtable2]]
        elim: qtable_cong[OF _ refl] intro: exI[of _ ne + length aux])
  qed
next
  case new
  then show ?case
    by (auto intro!: qtable_empty qtable1 qtable2[THEN qtable_cong] exI[of _ ne + length aux]
      simp: less_Suc_eq zero_enat_def[symmetric])
qed

lemma wf_until_aux_Cons: wf_until_aux σ n R pos φ I ψ (a # aux) ne ⟹
  wf_until_aux σ n R pos φ I ψ aux (Suc ne)
  unfolding wf_until_aux_def
  by (simp add: upt_conv_Cons del: upt_Suc cong: if_cong)

lemma wf_until_aux_Cons1: wf_until_aux σ n R pos φ I ψ ((t, a1, a2) # aux) ne ⟹ t = τ σ ne
  unfolding wf_until_aux_def
  by (simp add: upt_conv_Cons del: upt_Suc)
```

lemma *wf_until_aux_Cons3*: *wf_until_aux* σ *n R pos* φ *I* ψ ((*t*, *a1*, *a2*) # *aux*) *ne* $\Longrightarrow$
qtable n (*MFOTL.fv* ψ) (*mem_restr R*) (λ*v*. ($\exists$ *j*. *ne* $\leq$ *j* $\wedge$ *j* < *Suc* (*ne* + *length aux*) $\wedge$ *mem* (τ σ *j* − τ σ *ne*) *I* $\wedge$
MFOTL.sat σ (*map the v*) *j* ψ $\wedge$ ($\forall$ *k*$\in${*ne*..<*j*}. *if pos then MFOTL.sat* σ (*map the v*) *k* φ *else* $\neg$ *MFOTL.sat* σ (*map the v*) *k* φ))) *a2*
unfolding *wf_until_aux_def*
by (*simp add*: *upt_conv_Cons del*: *upt_Suc*)

lemma *upt_append*: *a* $\leq$ *b* $\Longrightarrow$ *b* $\leq$ *c* $\Longrightarrow$ [*a*..<*b*] @ [*b*..<*c*] = [*a*..<*c*]
by (*metis le_Suc_ex upt_add_eq_append*)

lemma *wf_mbuf2_add*:
assumes *wf_mbuf2 i ja jb P Q buf*
and *list_all2 P* [*ja*..<*ja′*] *xs*
and *list_all2 Q* [*jb*..<*jb′*] *ys*
and *ja* $\leq$ *ja′ jb* $\leq$ *jb′*
shows *wf_mbuf2 i ja′ jb′ P Q* (*mbuf2_add xs ys buf*)
using *assms* **unfolding** *wf_mbuf2_def*
by (*auto 0 3 simp*: *list_all2_append2 upt_append dest*: *list_all2_lengthD*
intro: *exI*[**where** *x*=[*i*..<*ja*]] *exI*[**where** *x*=[*ja*..<*ja′*]]
exI[**where** *x*=[*i*..<*jb*]] *exI*[**where** *x*=[*jb*..<*jb′*]] *split*: *prod.splits*)

lemma *mbuf2_take_eqD*:
assumes *mbuf2_take f buf* = (*xs*, *buf′*)
and *wf_mbuf2 i ja jb P Q buf*
shows *wf_mbuf2* (*min ja jb*) *ja jb P Q buf′*
and *list_all2* (λ*i z*. $\exists$ *x y*. *P i x* $\wedge$ *Q i y* $\wedge$ *z* = *f x y*) [*i*..<*min ja jb*] *xs*
using *assms* **unfolding** *wf_mbuf2_def*
by (*induction f buf arbitrary*: *i xs buf′ rule*: *mbuf2_take.induct*)
(*fastforce simp add*: *list_all2_Cons2 upt_eq_Cons_conv min_absorb1 min_absorb2 split*: *prod.splits*)+

lemma *mbuf2t_take_eqD*:
assumes *mbuf2t_take f z buf nts* = (*z′*, *buf′*, *nts′*)
and *wf_mbuf2 i ja jb P Q buf*
and *list_all2 R* [*i*..<*j*] *nts*
and *ja* $\leq$ *j jb* $\leq$ *j*
shows *wf_mbuf2* (*min ja jb*) *ja jb P Q buf′*
and *list_all2 R* [*min ja jb*..<*j*] *nts′*
using *assms* **unfolding** *wf_mbuf2_def*
by (*induction f z buf nts arbitrary*: *i z′ buf′ nts′ rule*: *mbuf2t_take.induct*)
(*auto simp add*: *list_all2_Cons2 upt_eq_Cons_conv less_eq_Suc_le min_absorb1 min_absorb2 split*: *prod.split*)

lemma *mbuf2t_take_induct*[*consumes 5*, *case_names base step*]:
assumes *mbuf2t_take f z buf nts* = (*z′*, *buf′*, *nts′*)
and *wf_mbuf2 i ja jb P Q buf*
and *list_all2 R* [*i*..<*j*] *nts*
and *ja* $\leq$ *j jb* $\leq$ *j*
and *U i z*
and $\bigwedge$*k x y t z*. *i* $\leq$ *k* $\Longrightarrow$ *Suc k* $\leq$ *ja* $\Longrightarrow$ *Suc k* $\leq$ *jb* $\Longrightarrow$
P k x $\Longrightarrow$ *Q k y* $\Longrightarrow$ *R k t* $\Longrightarrow$ *U k z* $\Longrightarrow$ *U* (*Suc k*) (*f x y t z*)
shows *U* (*min ja jb*) *z′*
using *assms* **unfolding** *wf_mbuf2_def*
by (*induction f z buf nts arbitrary*: *i z′ buf′ nts′ rule*: *mbuf2t_take.induct*)
(*auto simp add*: *list_all2_Cons2 upt_eq_Cons_conv less_eq_Suc_le min_absorb1 min_absorb2 elim!*: *arg_cong2*[*of _ _ _ _ U*, *OF _ refl*, *THEN iffD1*, *rotated*] *split*: *prod.split*)

lemma *mbuf2_take_add′*:

assumes *eq*: $mbuf2\_take\ f\ (mbuf2\_add\ xs\ ys\ buf) = (zs,\ buf')$
and *pre*: $wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$
and *xs*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$
$[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$
and *ys*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$
$[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$
and $j \le j'$
shows $wf\_mbuf2'\ \sigma\ j'\ n\ R\ \varphi\ \psi\ buf'$
and $list\_all2\ (\lambda i\ Z.\ \exists X\ Y.$
$qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi)\ X \wedge$
$qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi)\ Y \wedge$
$Z = f\ X\ Y)$
$[min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j')]\ zs$
using *pre* **unfolding** *wf_mbuf2'_def*
by (*force intro*!: *mbuf2_take_eqD*[*OF eq*] *wf_mbuf2_add*[*OF _ xs ys*] *progress_mono*[*OF* ‹$j \le j'$›])+

lemma *mbuf2t_take_add'*:
assumes *eq*: $mbuf2t\_take\ f\ z\ (mbuf2\_add\ xs\ ys\ buf)\ nts = (z',\ buf',\ nts')$
and *pre_buf*: $wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$
and *pre_nts*: $list\_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<j']\ nts$
and *xs*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$
$[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$
and *ys*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$
$[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$
and $j \le j'$
shows $wf\_mbuf2'\ \sigma\ j'\ n\ R\ \varphi\ \psi\ buf'$
and $wf\_ts\ \sigma\ j'\ \varphi\ \psi\ nts'$
using *pre_buf pre_nts* **unfolding** *wf_mbuf2'_def wf_ts_def*
by (*blast intro*!: *mbuf2t_take_eqD*[*OF eq*] *wf_mbuf2_add*[*OF _ xs ys*] *progress_mono*[*OF* ‹$j \le j'$›]
progress_le)+

lemma *mbuf2t_take_add_induct'*[*consumes 6*, *case_names base step*]:
assumes *eq*: $mbuf2t\_take\ f\ z\ (mbuf2\_add\ xs\ ys\ buf)\ nts = (z',\ buf',\ nts')$
and *pre_buf*: $wf\_mbuf2'\ \sigma\ j\ n\ R\ \varphi\ \psi\ buf$
and *pre_nts*: $list\_all2\ (\lambda i\ t.\ t = \tau\ \sigma\ i)\ [min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<j']\ nts$
and *xs*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$
$[progress\ \sigma\ \varphi\ j..<progress\ \sigma\ \varphi\ j']\ xs$
and *ys*: $list\_all2\ (\lambda i.\ qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi))$
$[progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ j']\ ys$
and $j \le j'$
and *base*: $U\ (min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j))\ z$
and *step*: $\bigwedge k\ X\ Y\ z.\ min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j) \le k \Longrightarrow$
$Suc\ k \le progress\ \sigma\ \varphi\ j' \Longrightarrow Suc\ k \le progress\ \sigma\ \psi\ j' \Longrightarrow$
$qtable\ n\ (MFOTL.fv\ \varphi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi)\ X \Longrightarrow$
$qtable\ n\ (MFOTL.fv\ \psi)\ (mem\_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \psi)\ Y \Longrightarrow$
$U\ k\ z \Longrightarrow U\ (Suc\ k)\ (f\ X\ Y\ (\tau\ \sigma\ k)\ z)$
shows $U\ (min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j'))\ z'$
using *pre_buf pre_nts* **unfolding** *wf_mbuf2'_def wf_ts_def*
by (*blast intro*!: *mbuf2t_take_induct*[*OF eq*] *wf_mbuf2_add*[*OF _ xs ys*] *progress_mono*[*OF* ‹$j \le j'$›]
progress_le base step)

lemma *progress_Until_le*: $progress\ \sigma\ (formula.Until\ \varphi\ I\ \psi)\ j \le min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
by (*cases right I*) (*auto simp*: *trans_le_add1 intro*!: *cInf_lower*)

lemma *list_all2_upt_Cons*: $P\ a\ x \Longrightarrow list\_all2\ P\ [Suc\ a..<b]\ xs \Longrightarrow Suc\ a \le b \Longrightarrow$
$list\_all2\ P\ [a..<b]\ (x\ \#\ xs)$
by (*simp add*: *list_all2_Cons2 upt_eq_Cons_conv*)

```
lemma list_all2_upt_append: list_all2 P [a..<b] xs ⟹ list_all2 P [b..<c] ys ⟹
  a ≤ b ⟹ b ≤ c ⟹ list_all2 P [a..<c] (xs @ ys)
  by (induction xs arbitrary: a) (auto simp add: list_all2_Cons2 upt_eq_Cons_conv)

lemma meval:
  assumes wf_mformula σ j n R φ φ'
  shows case meval n (τ σ j) (Γ σ j) φ of (xs, φₙ) ⇒ wf_mformula σ (Suc j) n R φₙ φ' ∧
    list_all2 (λi. qtable n (MFOTL.fv φ') (mem_restr R) (λv. MFOTL.sat σ (map the v) i φ'))
    [progress σ φ' j..<progress σ φ' (Suc j)] xs
using assms proof (induction φ arbitrary: n R φ')
  case (MRel rel)
  then show ?case
    by (cases pred: wf_mformula)
      (auto simp add: ball_Un intro: wf_mformula.intros qtableI table_eq_rel eq_rel_eval_trm
        in_eq_rel qtable_empty qtable_unit_table)
next
  case (MPred e ts)
  then show ?case
    by (cases pred: wf_mformula)
      (auto 0 4 simp: table_def in_these_eq match_wf_tuple match_eval_trm image_iff dest: ex_match
        split: if_splits intro!: wf_mformula.intros qtableI elim!: bexI[rotated])
next
  case (MAnd φ pos ψ buf)
  from MAnd.prems show ?case
    by (cases pred: wf_mformula)
      (auto simp: fvi_And sat_And fvi_And_Not sat_And_Not sat_the_restrict
       dest!: MAnd.IH split: if_splits prod.splits intro!: wf_mformula.And qtable_join
       dest: mbuf2_take_add' elim!: list.rel_mono_strong)
next
  case (MOr φ ψ buf)
  from MOr.prems show ?case
    by (cases pred: wf_mformula)
      (auto dest!: MOr.IH split: if_splits prod.splits intro!: wf_mformula.Or qtable_union
       dest: mbuf2_take_add' elim!: list.rel_mono_strong)
next
  case (MExists φ)
  from MExists.prems show ?case
    by (cases pred: wf_mformula)
      (force simp: list.rel_map fvi_Suc sat_fvi_cong nth_Cons'
        intro!: wf_mformula.Exists dest!: MExists.IH qtable_project_fv
        elim!: list.rel_mono_strong table_fvi_tl qtable_cong sat_fvi_cong[THEN iffD1, rotated −1]
        split: if_splits)+
next
  case (MPrev I φ first buf nts)
  from MPrev.prems show ?case
  proof (cases pred: wf_mformula)
    case (Prev ψ)
    let ?xs = fst (meval n (τ σ j) (Γ σ j) φ)
    let ?φ = snd (meval n (τ σ j) (Γ σ j) φ)
    let ?ls = fst (mprev_next I (buf @ ?xs) (nts @ [τ σ j]))
    let ?rs = fst (snd (mprev_next I (buf @ ?xs) (nts @ [τ σ j])))
    let ?ts = snd (snd (mprev_next I (buf @ ?xs) (nts @ [τ σ j])))
    let ?P = λi X. qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) i ψ) X
    let ?min = min (progress σ ψ (Suc j)) (Suc j − 1)
    from Prev MPrev.IH[of n R ψ] have IH: wf_mformula σ (Suc j) n R ?φ ψ and
      list_all2 ?P [progress σ ψ j..<progress σ ψ (Suc j)] ?xs by auto
   with Prev(4,5) have list_all2 (λi X. if mem (τ σ (Suc i) − τ σ i) I then ?P i X else X = empty_table)
       [min (progress σ ψ j) (j − 1)..<?min] ?ls ∧
```

```
      list_all2 ?P [?min..<progress σ ψ (Suc j)] ?rs ∧
      list_all2 (λi t. t = τ σ i) [?min..<Suc j] ?ts
     by (intro mprev)
       (auto simp: progress_mono progress_le simp del:
         intro!: list_all2_upt_append list_all2_appendI order.trans[OF min.cobounded1])
   moreover have min (Suc (progress σ ψ j)) j = Suc (min (progress σ ψ j) (j−1)) if j > 0
     using that by auto
   ultimately show ?thesis using progress_mono[of j Suc j σ ψ] Prev(1,3) IH
     by (auto simp: map_Suc_upt[symmetric] upt_Suc[of 0] list.rel_map qtable_empty_iff
       simp del: upt_Suc elim!: wf_mformula.Prev list.rel_mono_strong
       split: prod.split if_split_asm)
 qed simp
next
 case (MNext I φ first nts)
 from MNext.prems show ?case
 proof (cases pred: wf_mformula)
   case (Next ψ)

   have min[simp]:
     min (progress σ ψ j − Suc 0) (j − Suc 0) = progress σ ψ j − Suc 0
     min (progress σ ψ (Suc j) − Suc 0) j = progress σ ψ (Suc j) − Suc 0 for j
     using progress_le[of σ ψ j] progress_le[of σ ψ Suc j] by auto

   let ?xs = fst (meval n (τ σ j) (Γ σ j) φ)
   let ?ys = case (?xs, first) of (_ # xs, True) ⇒ xs | _ ⇒ ?xs
   let ?φ = snd (meval n (τ σ j) (Γ σ j) φ)
   let ?ls = fst (mprev_next I ?ys (nts @ [τ σ j]))
   let ?rs = fst (snd (mprev_next I ?ys (nts @ [τ σ j])))
   let ?ts = snd (snd (mprev_next I ?ys (nts @ [τ σ j])))
   let ?P = λi X. qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) i ψ) X
   let ?min = min (progress σ ψ (Suc j) − 1) (Suc j − 1)
   from Next MNext.IH[of n R ψ] have IH: wf_mformula σ (Suc j) n R ?φ ψ
     list_all2 ?P [progress σ ψ j..<progress σ ψ (Suc j)] ?xs by auto
    with Next have list_all2 (λi X. if mem (τ σ (Suc i) − τ σ i) I then ?P (Suc i) X else X =
empty_table)
       [progress σ ψ j − 1..<?min] ?ls ∧
      list_all2 ?P [Suc ?min..<progress σ ψ (Suc j)] ?rs ∧
      list_all2 (λi t. t = τ σ i) [?min..<Suc j] ?ts if progress σ ψ j < progress σ ψ (Suc j)
     using progress_le[of σ ψ j] that
     by (intro mnext)
       (auto simp: progress_mono list_all2_Cons2 upt_eq_Cons_conv
         intro!: list_all2_upt_append list_all2_appendI split: list.splits)
   then show ?thesis using progress_mono[of j Suc j σ ψ] progress_le[of σ ψ Suc j] Next IH
     by (cases progress σ ψ (Suc j) > progress σ ψ j)
       (auto 0 3 simp: qtable_empty_iff le_Suc_eq le_diff_conv
        elim!: wf_mformula.Next list.rel_mono_strong list_all2_appendI
        split: prod.split list.splits if_split_asm)
 qed simp
next
 case (MSince pos φ I ψ buf nts aux)
 note sat.simps[simp del]
 from MSince.prems obtain φ'' φ''' ψ'' where Since_eq: φ' = MFOTL.Since φ''' I ψ''
   and pos: if pos then φ''' = φ'' else φ''' = MFOTL.Neg φ''
   and pos_eq: safe_formula φ''' = pos
   and φ: wf_mformula σ j n R φ φ''
   and ψ: wf_mformula σ j n R ψ ψ''
   and fvi_subset: MFOTL.fv φ'' ⊆ MFOTL.fv ψ''
   and buf: wf_mbuf2' σ j n R φ'' ψ'' buf
```

```
    and nts: wf_ts σ j φ'' ψ'' nts
    and aux: wf_since_aux σ n R pos φ'' I ψ'' aux (progress σ (formula.Since φ''' I ψ'') j)
    by (cases pred: wf_mformula) (auto)
  have φ''': MFOTL.fv φ''' = MFOTL.fv φ'' progress σ φ''' j = progress σ φ'' j for j
    using pos by (simp_all split: if_splits)
  have nts_snoc: list_all2 (λi t. t = τ σ i)
    [min (progress σ φ'' j) (progress σ ψ'' j)..<Suc j] (nts @ [τ σ j])
    using nts unfolding wf_ts_def
    by (auto simp add: progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
  have update: wf_since_aux σ n R pos φ'' I ψ'' (snd (zs, aux')) (progress σ (formula.Since φ''' I ψ'')
(Suc j)) ∧
    list_all2 (λi. qtable n (MFOTL.fv φ''' ∪ MFOTL.fv ψ'') (mem_restr R)
      (λv. MFOTL.sat σ (map the v) i (formula.Since φ''' I ψ'')))
     [progress σ (formula.Since φ''' I ψ'') j..<progress σ (formula.Since φ''' I ψ'') (Suc j)] (fst (zs, aux'))
    if eval_φ: fst (meval n (τ σ j) (Γ σ j) φ) = xs
      and eval_ψ: fst (meval n (τ σ j) (Γ σ j) ψ) = ys
      and eq: mbuf2t_take (λr1 r2 t (zs, aux).
        case update_since I pos r1 r2 t aux of (z, x) ⇒ (zs @ [z], x))
        ([], aux) (mbuf2_add xs ys buf) (nts @ [τ σ j]) = ((zs, aux'), buf', nts')
    for xs ys zs aux' buf' nts'
    unfolding progress.simps φ'''
  proof (rule mbuf2t_take_add_induct'[where j'=Suc j and z'=(zs, aux'), OF eq buf nts_snoc],
      goal_cases xs ys _ base step)
    case xs
    then show ?case
      using MSince.IH(1)[OF φ] eval_φ by auto
  next
    case ys
    then show ?case
      using MSince.IH(2)[OF ψ] eval_ψ by auto
  next
    case base
    then show ?case
     using aux by (simp add: φ''')
  next
    case (step k X Y z)
    then show ?case
      using fvi_subset pos
      by (auto simp: Un_absorb1 elim!: update_since(1) list_all2_appendI dest!: update_since(2)
        split: prod.split if_splits)
  qed simp
  with MSince.IH(1)[OF φ] MSince.IH(2)[OF ψ] show ?case
    by (auto 0 3 simp: Since_eq split: prod.split
      intro: wf_mformula.Since[OF _ _ pos pos_eq fvi_subset]
      elim: mbuf2t_take_add'(1)[OF _ buf nts_snoc] mbuf2t_take_add'(2)[OF _ buf nts_snoc])
next
  case (MUntil pos φ I ψ buf nts aux)
  note sat.simps[simp del] progress.simps[simp del]
  from MUntil.prems obtain φ'' φ''' ψ'' where Until_eq: φ' = MFOTL.Until φ''' I ψ''
    and pos: if pos then φ''' = φ'' else φ''' = MFOTL.Neg φ''
    and pos_eq: safe_formula φ''' = pos
    and φ: wf_mformula σ j n R φ φ''
    and ψ: wf_mformula σ j n R ψ ψ''
    and fvi_subset: MFOTL.fv φ'' ⊆ MFOTL.fv ψ''
    and buf: wf_mbuf2' σ j n R φ'' ψ'' buf
    and nts: wf_ts σ j φ'' ψ'' nts
    and aux: wf_until_aux σ n R pos φ'' I ψ'' aux (progress σ (formula.Until φ''' I ψ'') j)
    and length_aux: progress σ (formula.Until φ''' I ψ'') j + length aux =
```

min (progress σ φ'' j) (progress σ ψ'' j)
by (cases pred: wf_mformula) (auto)
have φ''': progress σ φ''' j = progress σ φ'' j **for** j
using pos **by** (simp_all add: progress.simps split: if_splits)
have nts_snoc: list_all2 (λi t. t = τ σ i)
[min (progress σ φ'' j) (progress σ ψ'' j)..<Suc j] (nts @ [τ σ j])
using nts **unfolding** wf_ts_def
by (auto simp add: progress_le[THEN min.coboundedI1] intro: list_all2_appendI)
{
fix xs ys zs aux' aux'' buf' nts'
assume eval_φ: fst (meval n (τ σ j) (Γ σ j) φ) = xs
and eval_ψ: fst (meval n (τ σ j) (Γ σ j) ψ) = ys
and eq1: mbuf2t_take (update_until I pos) aux (mbuf2_add xs ys buf) (nts @ [τ σ j]) =
(aux', buf', nts')
and eq2: eval_until I (case nts' of [] ⇒ τ σ j | nt # _ ⇒ nt) aux' = (zs, aux'')
have update1: wf_until_aux σ n R pos φ'' I ψ'' aux' (progress σ (formula.Until φ''' I ψ'') j) ∧
progress σ (formula.Until φ''' I ψ'') j + length aux' =
min (progress σ φ''' (Suc j)) (progress σ ψ'' (Suc j))
using MUntil.IH(1)[OF φ] eval_φ MUntil.IH(2)[OF ψ]
eval_ψ nts_snoc nts_snoc length_aux aux fvi_subset
unfolding φ'''
by (elim mbuf2t_take_add_induct'[**where** j'=Suc j, OF eq1 buf])
(auto simp: length_update_until elim: wf_update_until)
have nts': wf_ts σ (Suc j) φ'' ψ'' nts'
using MUntil.IH(1)[OF φ] eval_φ MUntil.IH(2)[OF ψ] eval_ψ nts_snoc
unfolding wf_ts_def
by (intro mbuf2t_take_eqD(2)[OF eq1]) (auto simp: progress_mono progress_le
intro: wf_mbuf2_add buf[unfolded wf_mbuf2'_def])
have i ≤ progress σ (formula.Until φ''' I ψ'') (Suc j) ⟹
wf_until_aux σ n R pos φ'' I ψ'' aux' i ⟹
i + length aux' = min (progress σ φ''' (Suc j)) (progress σ ψ'' (Suc j)) ⟹
wf_until_aux σ n R pos φ'' I ψ'' aux'' (progress σ (formula.Until φ''' I ψ'') (Suc j)) ∧
i + length zs = progress σ (formula.Until φ''' I ψ'') (Suc j) ∧
i + length zs + length aux'' = min (progress σ φ''' (Suc j)) (progress σ ψ'' (Suc j)) ∧
list_all2 (λi. qtable n (MFOTL.fv ψ'') (mem_restr R)
(λv. MFOTL.sat σ (map the v) i (formula.Until (if pos then φ'' else MFOTL.Neg φ'') I ψ'')))
[i..<i + length zs] zs **for** i
using eq2
proof (induction aux' arbitrary: zs aux'' i)
case Nil
then show ?case **by** (auto dest!: antisym[OF progress_Until_le])
next
case (Cons a aux')
obtain t a1 a2 **where** a = (t, a1, a2) **by** (cases a)
from Cons.prems(2) **have** aux': wf_until_aux σ n R pos φ'' I ψ'' aux' (Suc i)
by (rule wf_until_aux_Cons)
from Cons.prems(2) **have** 1: t = τ σ i
unfolding ‹a = (t, a1, a2)› **by** (rule wf_until_aux_Cons1)
from Cons.prems(2) **have** 3: qtable n (MFOTL.fv ψ'') (mem_restr R) (λv.
(∃j≥i. j < Suc (i + length aux') ∧ mem (τ σ j − τ σ i) I ∧ MFOTL.sat σ (map the v) j ψ'' ∧
(∀k∈{i..<j}. if pos then MFOTL.sat σ (map the v) k φ'' else ¬ MFOTL.sat σ (map the v) k φ'')))
a2
unfolding ‹a = (t, a1, a2)› **by** (rule wf_until_aux_Cons3)
from Cons.prems(3) **have** Suc_i_aux': Suc i + length aux' =
min (progress σ φ''' (Suc j)) (progress σ ψ'' (Suc j))
by simp
have i ≥ progress σ (formula.Until φ''' I ψ'') (Suc j)
if enat (case nts' of [] ⇒ τ σ j | nt # x ⇒ nt) ≤ enat t + right I

using *that nts'* **unfolding** *wf_ts_def progress.simps*
by (*auto simp add*: *1 list_all2_Cons2 upt_eq_Cons_conv* φ'''
intro!: *cInf_lower* τ*_mono elim*!: *order.trans*[*rotated*] *simp del*: *upt_Suc split*: *if_splits list.splits*)
moreover
have *Suc i* $\leq$ *progress* σ (*formula.Until* φ''' *I* ψ'') (*Suc j*)
if *enat t* + *right I* < *enat* (*case nts'* *of* [] $\Rightarrow$ τ σ *j* | *nt* # *x* $\Rightarrow$ *nt*)
proof −
from *that* **obtain** *m* **where** *m*: *right I* = *enat m* **by** (*cases right I*) *auto*
have τ*_min*: τ σ (*min j k*) = *min* (τ σ *j*) (τ σ *k*) **for** *k*
by (*simp add*: *min_of_mono monoI*)
have *le_progress_iff*[*simp*]: *i* $\leq$ *progress* σ φ *i* $\longleftrightarrow$ *progress* σ φ *i* = *i* **for** φ *i*
using *progress_le*[*of* σ φ *i*] **by** *auto*
have *min_Suc*[*simp*]: *min j* (*Suc j*) = *j* **by** *auto*
let *?X* = {*i*. $\forall$ *k*. *k* < *Suc j* $\wedge$ *k* $\leq$*min* (*progress* σ φ''' (*Suc j*)) (*progress* σ ψ'' (*Suc j*)) $\longrightarrow$ *enat* (τ σ *k*) $\leq$ *enat* (τ σ *i*) + *right I*}
let *?min* = *min j* (*min* (*progress* σ φ'' (*Suc j*)) (*progress* σ ψ'' (*Suc j*)))
have τ σ *?min* $\leq$ τ σ *j*
by (*rule* τ*_mono*) *auto*
from *m* **have** *?X* $\neq$ {}
by (*auto dest*!: τ*_mono*[*of _ progress* σ φ'' (*Suc j*) σ]
simp: *not_le not_less* φ''' *intro*!: *exI*[*of _ progress* σ φ'' (*Suc j*)])
thm *less_le_trans*[*of* τ σ *i* + *m* τ σ _ τ σ _ + *m*]
from *m* **show** *?thesis*
using *that nts'* **unfolding** *wf_ts_def progress.simps*
by (*intro cInf_greatest*[*OF* ‹*?X* $\neq$ {}›])
(*auto simp*: *1* φ''' *not_le not_less list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq*
simp del: *upt_Suc split*: *list.splits if_splits*
dest!: *spec*[*of _ ?min*] *less_le_trans*[*of* τ σ *i* + *m* τ σ _ τ σ _ + *m*] *less_*τ*D*)
qed
moreover have ∗: *k* < *progress* σ ψ (*Suc j*) **if**
enat (τ σ *i*) + *right I* < *enat* (*case nts'* *of* [] $\Rightarrow$ τ σ *j* | *nt* # *x* $\Rightarrow$ *nt*)
enat (τ σ *k* − τ σ *i*) $\leq$ *right I* ψ = ψ'' $\vee$ ψ = φ'' **for** *k* ψ
proof −
from *that*(*1*,*2*) **obtain** *m* **where** *right I* = *enat m*
τ σ *i* + *m* < (*case nts'* *of* [] $\Rightarrow$ τ σ *j* | *nt* # *x* $\Rightarrow$ *nt*) τ σ *k* − τ σ *i* $\leq$ *m*
by (*cases right I*) *auto*
with *that*(*3*) *nts' progress_le*[*of* σ ψ'' *Suc j*] *progress_le*[*of* σ φ'' *Suc j*]
show *?thesis*
unfolding *wf_ts_def le_diff_conv*
by (*auto simp*: *not_le list_all2_Cons2 upt_eq_Cons_conv less_Suc_eq add.commute*
simp del: *upt_Suc split*: *list.splits if_splits dest*!: *le_less_trans*[*of* τ σ *k*] *less_*τ*D*)
qed
ultimately show *?case* **using** *Cons.prems Suc_i_aux'*[*simplified*]
unfolding ‹*a* = (*t*, *a1*, *a2*)›
by (*auto simp*: φ''' *1 sat.simps upt_conv_Cons dest*!: *Cons.IH*[*OF _ aux' Suc_i_aux'*]
simp del: *upt_Suc split*: *if_splits prod.splits intro*!: *iff_exI qtable_cong*[*OF 3 refl*])
qed
note *this*[*OF progress_mono*[*OF le_SucI*, *OF order.refl*] *conjunct1*[*OF update1*] *conjunct2*[*OF update1*]]
}
note *update* = *this*
from *MUntil.IH*(*1*)[*OF* φ] *MUntil.IH*(*2*)[*OF* ψ] *pos pos_eq fvi_subset* **show** *?case*
by (*auto 0 4 simp*: *Until_eq* φ''' *progress.simps*(*3*) *split*: *prod.split if_splits*
dest!: *update*[*OF refl refl*, *rotated*]
intro!: *wf_mformula.Until*
elim!: *list.rel_mono_strong qtable_cong*
elim: *mbuf2t_take_add'*(*1*)[*OF _ buf nts_snoc*] *mbuf2t_take_add'*(*2*)[*OF _ buf nts_snoc*])
qed

6.5.4 Monitor step

lemma *wf_mstate_mstep*: *wf_mstate* φ π *R st* $\Longrightarrow$ *last_ts* $\pi \le$ *snd tdb* $\Longrightarrow$
wf_mstate φ (*psnoc* π *tdb*) *R* (*snd* (*mstep tdb st*))
unfolding *wf_mstate_def mstep_def Let_def*
by (*fastforce simp add*: *progress_mono le_imp_diff_is_add split*: *prod.splits*
elim!: *prefix_of_psnocE dest*: *meval list_all2_lengthD*)

lemma *mstep_output_iff*:
assumes *wf_mstate* φ π *R st last_ts* $\pi \le$ *snd tdb prefix_of* (*psnoc* π *tdb*) σ *mem_restr R v*
shows $(i,\ v) \in$ *fst* (*mstep tdb st*) $\longleftrightarrow$
progress σ φ (*plen* π) $\le i \wedge i <$ *progress* σ φ (*Suc* (*plen* π)) $\wedge$
wf_tuple (*MFOTL.nfv* φ) (*MFOTL.fv* φ) *v* $\wedge$ *MFOTL.sat* σ (*map the v*) *i* φ
proof −
from *prefix_of_psnocE*[*OF assms*(*3*,*2*)] **have** *prefix_of* π σ
Γ σ (*plen* π) = *fst tdb* τ σ (*plen* π) = *snd tdb* **by** *auto*
moreover from *assms*(*1*) ‹*prefix_of* π σ› **have** *mstate_n st* = *MFOTL.nfv* φ
mstate_i st = *progress* σ φ (*plen* π) *wf_mformula* σ (*plen* π) (*mstate_n st*) *R* (*mstate_m st*) φ
unfolding *wf_mstate_def* **by** *blast*+
moreover from *meval*[*OF* ‹*wf_mformula* σ (*plen* π) (*mstate_n st*) *R* (*mstate_m st*) φ›] **obtain** *Vs* *st'* **where**
meval (*mstate_n st*) (τ σ (*plen* π)) (Γ σ (*plen* π)) (*mstate_m st*) = (*Vs*, *st'*)
wf_mformula σ (*Suc* (*plen* π)) (*mstate_n st*) *R st'* φ
list_all2 (λi. *qtable* (*mstate_n st*) (*fv* φ) (*mem_restr R*) (λv. *MFOTL.sat* σ (*map the v*) *i* φ))
[*progress* σ φ (*plen* π)..<*progress* σ φ (*Suc* (*plen* π))] *Vs* **by** *blast*
moreover from *this assms*(*4*) **have** *qtable* (*mstate_n st*) (*fv* φ) (*mem_restr R*)
(λv. *MFOTL.sat* σ (*map the v*) *i* φ) (*Vs* ! (*i* − *progress* σ φ (*plen* π)))
if *progress* σ φ (*plen* π) $\le$ *i i* < *progress* σ φ (*Suc* (*plen* π))
using *that* **by** (*auto simp*: *list_all2_conv_all_nth*
dest!: *spec*[*of* _ (*i* − *progress* σ φ (*plen* π))])
ultimately show *?thesis*
using *assms*(*4*) **unfolding** *mstep_def Let_def*
by (*auto simp*: *in_set_enumerate_eq list_all2_conv_all_nth progress_mono le_imp_diff_is_add*
elim!: *in_qtableE in_qtableI intro*!: *bexI*[*of* _ (*i*, *Vs* ! (*i* − *progress* σ φ (*plen* π)))])
qed

6.5.5 Monitor function

definition *minit_safe* **where**
minit_safe φ = (*if mmonitorable_exec* φ *then minit* φ *else undefined*)

lemma *minit_safe_minit*: *mmonitorable* φ $\Longrightarrow$ *minit_safe* φ = *minit* φ
unfolding *minit_safe_def monitorable_formula_code* **by** *simp*

lemma (**in** *monitorable_mfotl*) *mstep_mverdicts*:
assumes *wf*: *wf_mstate* φ π *R st*
and *le*[*simp*]: *last_ts* $\pi \le$ *snd tdb*
and *restrict*: *mem_restr R v*
shows $(i,\ v) \in$ *fst* (*mstep tdb st*) $\longleftrightarrow$ $(i,\ v) \in$ *M* (*psnoc* π *tdb*) − *M* π
proof −
obtain σ **where** *p2*: *prefix_of* (*psnoc* π *tdb*) σ
using *ex_prefix_of* **by** *blast*
with *le* **have** *p1*: *prefix_of* π σ **by** (*blast elim*!: *prefix_of_psnocE*)
show *?thesis*
unfolding *M_def*
by (*auto 0 3 simp*: *p2 progress_prefix_conv*[*OF* _ *p1*] *sat_prefix_conv*[*OF* _ *p1*] *not_less*
pprogress_eq[*OF p1*] *pprogress_eq*[*OF p2*]
dest: *mstep_output_iff*[*OF wf le p2 restrict*, *THEN iffD1*] *spec*[*of* _ σ]
mstep_output_iff[*OF wf le* _ *restrict*, *THEN iffD1*] *progress_sat_cong*[*OF p1*]

intro: *mstep_output_iff*[*OF wf le p2 restrict*, *THEN iffD2*] *p1*)
qed

primrec *msteps0* **where**
msteps0 [] *st* = ({}, *st*)
| *msteps0* (*tdb* # π) *st* =
(*let* (*V*′, *st*′) = *mstep tdb st*; (*V*′′, *st*′′) = *msteps0* π *st*′ *in* (*V*′ ∪ *V*′′, *st*′′))

primrec *msteps0_stateless* **where**
msteps0_stateless [] *st* = {}
| *msteps0_stateless* (*tdb* # π) *st* = (*let* (*V*′, *st*′) = *mstep tdb st in V*′ ∪ *msteps0_stateless* π *st*′)

lemma *msteps0_msteps0_stateless*: *fst* (*msteps0 w st*) = *msteps0_stateless w st*
by (*induct w arbitrary*: *st*) (*auto simp*: *split_beta*)

lift_definition *msteps* :: ′*a MFOTL.prefix* ⇒ ′*a mstate* ⇒ (*nat* × ′*a option list*) *set* × ′*a mstate*
is *msteps0* **.**

lift_definition *msteps_stateless* :: ′*a MFOTL.prefix* ⇒ ′*a mstate* ⇒ (*nat* × ′*a option list*) *set*
is *msteps0_stateless* **.**

lemma *msteps_msteps_stateless*: *fst* (*msteps w st*) = *msteps_stateless w st*
by *transfer* (*rule msteps0_msteps0_stateless*)

lemma *msteps0_snoc*: *msteps0* (π @ [*tdb*]) *st* =
(*let* (*V*′, *st*′) = *msteps0* π *st*; (*V*′′, *st*′′) = *mstep tdb st*′ *in* (*V*′ ∪ *V*′′, *st*′′))
by (*induct* π *arbitrary*: *st*) (*auto split*: *prod.splits*)

lemma *msteps_psnoc*: *last_ts* π ≤ *snd tdb* ⟹ *msteps* (*psnoc* π *tdb*) *st* =
(*let* (*V*′, *st*′) = *msteps* π *st*; (*V*′′, *st*′′) = *mstep tdb st*′ *in* (*V*′ ∪ *V*′′, *st*′′))
by *transfer* (*auto simp*: *msteps0_snoc split*: *list.splits prod.splits if_splits*)

definition *monitor* **where**
monitor φ π = *msteps_stateless* π (*minit_safe* φ)

lemma *Suc_length_conv_snoc*: (*Suc n* = *length xs*) = (∃ *y ys*. *xs* = *ys* @ [*y*] ∧ *length ys* = *n*)
by (*cases xs rule*: *rev_cases*) *auto*

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps*: *wf_mstate* φ π *R st* ⟹ *mem_restr R v* ⟹ π ≤ π′ ⟹
X = *msteps* (*pdrop* (*plen* π) π′) *st* ⟹ *wf_mstate* φ π′ *R* (*snd X*) ∧
((*i*, *v*) ∈ *fst X*) = ((*i*, *v*) ∈ *M* π′ − *M* π)
proof (*induct plen* π′ − *plen* π *arbitrary*: *X st* π π′)
case *0*
from *0*(*1*,*4*,*5*) **have** π = π′ *X* = ({}, *st*)
by (*transfer*; *auto*)+
with *0*(*2*) **show** *?case* **by** *simp*
next
case (*Suc x*)
from *Suc*(*2*,*5*) **obtain** π′′ *tdb* **where** *x* = *plen* π′′ − *plen* π π ≤ π′′
π′ = *psnoc* π′′ *tdb pdrop* (*plen* π) (*psnoc* π′′ *tdb*) = *psnoc* (*pdrop* (*plen* π) π′′) *tdb*
last_ts (*pdrop* (*plen* π) π′′) ≤ *snd tdb last_ts* π′′ ≤ *snd tdb*
π′′ ≤ *psnoc* π′′ *tdb*
proof (*atomize_elim*, *transfer*, *elim exE*, *goal_cases prefix*)
case (*prefix* _ _ π′ _ π_*tdb*)
then show *?case*
proof (*cases* π_*tdb rule*: *rev_cases*)
case (*snoc* π *tdb*)

with *prefix* **show** *?thesis*
by (*intro bexI*[*of* _ π' @ π] *exI*[*of* _ *tdb*])
(*force simp*: *sorted_append append_eq_Cons_conv split*: *list.splits if_splits*)+
qed *simp*
qed
with *Suc*(*1*)[*OF this*(*1*) *Suc.prems*(*1*,*2*) *this*(*2*) *refl*] *Suc.prems* **show** *?case*
unfolding *msteps_msteps_stateless*[*symmetric*]
by (*auto simp*: *msteps_psnoc split_beta mstep_mverdicts*
dest: *mono_monitor*[*THEN set_mp*, *rotated*] *intro*!: *wf_mstate_mstep*)
qed

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless*:
assumes *wf_mstate* φ π *R st mem_restr R v* $\pi \leq \pi'$
shows (*i*, *v*) $\in$ *msteps_stateless* (*pdrop* (*plen* π) π') *st* $\longleftrightarrow$ (*i*, *v*) $\in$ *M* π' $-$ *M* π
using *wf_mstate_msteps*[*OF assms refl*] **unfolding** *msteps_msteps_stateless* **by** *simp*

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless_UNIV*: *wf_mstate* φ π *UNIV st* $\Longrightarrow$ $\pi \leq$ π' $\Longrightarrow$
msteps_stateless (*pdrop* (*plen* π) π') *st* = *M* π' $-$ *M* π
by (*auto dest*: *wf_mstate_msteps_stateless*[*OF* _ *mem_restr_UNIV*])

lemma (**in** *monitorable_mfotl*) *mverdicts_Nil*: *M pnil* = {}
by (*simp add*: *M_def pprogress_eq*)

lemma *wf_mstate_minit_safe*: *mmonitorable* φ $\Longrightarrow$ *wf_mstate* φ *pnil R* (*minit_safe* φ)
using *wf_mstate_minit minit_safe_minit mmonitorable_def* **by** *metis*

lemma (**in** *monitorable_mfotl*) *monitor_mverdicts*: *monitor* φ π = *M* π
unfolding *monitor_def* **using** *monitorable*
by (*subst wf_mstate_msteps_stateless_UNIV*[*OF wf_mstate_minit_safe*, *simplified*])
(*auto simp*: *mmonitorable_def mverdicts_Nil*)

6.6 Collected correctness results

context *monitorable_mfotl*
begin

We summarize the main results proved above.

1. The term *M* describes semantically the monitor's expected behaviour:
   - *mono_monitor*: $\pi \leq \pi' \Longrightarrow M\ \pi \subseteq M\ \pi'$
   - *sound_monitor*: ⟦(*i*, *v*) $\in$ *M* π; *prefix_of* π σ⟧ $\Longrightarrow$ *MFOTL.sat* σ (*map the v*) *i* φ
   - *complete_monitor*: ⟦*prefix_of* π σ; *wf_tuple* (*MFOTL.nfv* φ) (*fv* φ) *v*; $\bigwedge\sigma$. *prefix_of* π σ $\Longrightarrow$ *MFOTL.sat* σ (*map the v*) *i* φ⟧ $\Longrightarrow$ $\exists\pi'$. *prefix_of* π' σ $\wedge$ (*i*, *v*) $\in$ *M* π'
   - *sliceable_M*: *mem_restr S v* $\Longrightarrow$ ((*i*, *v*) $\in$ *M* (*pmap_*Γ (λD. *D* $\cap$ *relevant_events* φ *S*) π)) = ((*i*, *v*) $\in$ *M* π)
2. The executable monitor's online interface *minit_safe* and *mstep* preserves the invariant *wf_mstate* and produces the the verdicts according to *M*:
   - *wf_mstate_minit_safe*: *mmonitorable* φ' $\Longrightarrow$ *wf_mstate* φ' *pnil R* (*minit_safe* φ')
   - *wf_mstate_mstep*: ⟦*wf_mstate* φ' π *R st*; *last_ts* $\pi \leq$ *snd tdb*⟧ $\Longrightarrow$ *wf_mstate* φ' (*psnoc* π *tdb*) *R* (*snd* (*mstep tdb st*))
   - *mstep_mverdicts*: ⟦*wf_mstate* φ π *R st*; *last_ts* $\pi \leq$ *snd tdb*; *mem_restr R v*⟧ $\Longrightarrow$ ((*i*, *v*) $\in$ *fst* (*mstep tdb st*)) = ((*i*, *v*) $\in$ *M* (*psnoc* π *tdb*) $-$ *M* π)

3. The executable monitor's offline interface *Monitor.monitor* implements M:
   - *monitor_mverdicts*: $Monitor.monitor\ \varphi\ \pi = M\ \pi$

end

7 Slicing framework

This section formalizes the abstract slicing framework and the joint data slicer presented in the article [3, Sections 4.2 and 4.3].

7.1 Abstract slicing

7.1.1 Definition 1

Corresponds to locale *monitor* defined in theory *MFOTL_Monitor.Abstract_Monitor*.

7.1.2 Definition 2

locale *slicer* = *monitor* +
  fixes *submonitor* :: $'k$ :: *finite* ⇒ $'a$ *prefix* ⇒ (*nat* × $'b$ *option list*) *set*
  and *splitter* :: $'a$ *prefix* ⇒ $'k$ ⇒ $'a$ *prefix*
  and *joiner* :: ($'k$ ⇒ (*nat* × $'b$ *option list*) *set*) ⇒ (*nat* × $'b$ *option list*) *set*
assumes *mono_splitter*: $\pi \le \pi' \Longrightarrow$ *splitter* π $k \le$ *splitter* π' k
  and *correct_slicer*: *joiner* (λk. *submonitor* k (*splitter* π k)) = $M\ \pi$
begin

lemmas *sound_slicer* = *equalityD1*[*OF correct_slicer*]
lemmas *complete_slicer* = *equalityD2*[*OF correct_slicer*]

end

locale *self_slicer* = *slicer nfv fv sat M* λ_. *M splitter joiner* **for** *nfv fv sat M splitter joiner*

7.1.3 Definition 3

locale *event_separable_splitter* =
  fixes *event_splitter* :: $'a$ ⇒ $'k$ :: *finite set*
begin

lift_definition *splitter* :: $'a$ *prefix* ⇒ $'k$ ⇒ $'a$ *prefix* **is**
  $\lambda\pi\ k.$ *map* ($\lambda(D, t)$. ($\{e \in D.\ k \in$ *event_splitter* $e\}$, t)) π
  by (*auto simp*: *o_def split_beta*)

7.1.4 Lemma 1

lemma *mono_splitter*: $\pi \le \pi' \Longrightarrow$ *splitter* π $k \le$ *splitter* π' k
  by *transfer auto*

end

7.2 Joint data slicer

abbreviation (*input*) *ok* φ $v \equiv$ *wf_tuple* (*MFOTL.nfv* φ) (*MFOTL.fv* φ) v

locale *splitting_strategy* =

fixes φ :: $'a$ *MFOTL.formula*
and *strategy* :: $'a$ *option list* $\Rightarrow$ $'k$ :: *finite set*
assumes *strategy_nonempty*: *ok* φ $v \Longrightarrow$ *strategy* $v \neq \{\}$
begin

abbreviation *slice_set* **where**
slice_set $k \equiv \{v.\ \exists v'.$ *map the* $v' = v \wedge$ *ok* φ $v' \wedge k \in$ *strategy* $v'\}$

end

7.2.1 Definition 4

locale *MFOTL_monitor* =
monitor MFOTL.nfv φ *MFOTL.fv* φ $\lambda\sigma$ v i. *MFOTL.sat* σ v i φ M **for** φ M

locale *joint_data_slicer* = *MFOTL_monitor* φ M + *splitting_strategy* φ *strategy*
for φ M *strategy*
begin

definition *event_splitter* **where**
event_splitter $e = (\bigcup($*strategy* ` $\{v.$ *ok* φ $v \wedge$ *MFOTL.matches* (*map the* v) φ $e\}))$

sublocale *event_separable_splitter* **where** *event_splitter* = *event_splitter* **.**

definition *joiner* **where**
joiner = $(\lambda s. \bigcup k.\ s\ k \cap$ (*UNIV* :: *nat set*) $\times \{v.\ k \in$ *strategy* $v\})$

lemma *splitter_pslice*: *splitter* π k = *MFOTL_slicer.pslice* φ (*slice_set* k) π
by *transfer* (*auto simp*: *event_splitter_def*)

7.2.2 Lemma 2

Corresponds to the following theorem *sat_slice_strong* proved in theory *MFOTL_Monitor.Abstract_Monitor*:
$[\![$*relevant_events* φ' $S \subseteq E$; $v \in S]\!] \Longrightarrow$ *MFOTL.sat* σ v i φ' = *MFOTL.sat* (*map_Γ* $(\lambda D.\ D \cap E)$ σ) v i φ'

7.2.3 Theorem 1

sublocale *joint_monitor*: *MFOTL_monitor* φ $\lambda\pi$. *joiner* (λk. M (*splitter* π k))
proof (*unfold_locales*, *goal_cases mono wf sound complete*)
case (*mono* π π')
show *?case*
using *mono_monitor*[*OF mono_splitter*, *OF mono*]
by (*auto simp*: *joiner_def*)
next
case (*wf* i v π)
then obtain k **where** *in_M*: $(i, v) \in M$ (*splitter* π k) **and** k: $k \in$ *strategy* v
unfolding *joiner_def* **by** (*auto split*: *if_splits*)
then show *?case*
using *wf_monitor*[*OF in_M*] **by** *auto*
next
case (*sound* i v π σ)
then obtain k **where** *in_M*: $(i, v) \in M$ (*splitter* π k) **and** k: $k \in$ *strategy* v
unfolding *joiner_def* **by** (*auto split*: *if_splits*)
have *wf*: *ok* φ v **and** *sat*: $\bigwedge\sigma$. *prefix_of* (*splitter* π k) $\sigma \Longrightarrow$ *MFOTL.sat* σ (*map the* v) i φ
using *sound_monitor*[*OF in_M*] *wf_monitor*[*OF in_M*] **by** *auto*
then have *MFOTL.sat* σ (*map the* v) i φ **if** *prefix_of* π σ **for** σ
using *that* k

```
    by (intro iffD2[OF sat_slice_iff[of map the v slice_set k σ i φ]])
      (auto simp: wf_tuple_def fvi_less_nfv splitter_pslice intro!: exI[of _ v] prefix_of_pmap_Γ)
  then show ?case using sound(2) by blast
next
  case (complete π σ v i)
  with strategy_nonempty obtain k where k: k ∈ strategy v by blast
  have MFOTL.sat σ′ (map the v) i φ if prefix_of (MFOTL_slicer.pslice φ (slice_set k) π) σ′ for σ′
  proof −
    have MFOTL.sat σ′ (map the v) i φ = MFOTL.sat (MFOTL_slicer.slice φ (slice_set k) σ′) (map the v) i φ
      using complete(2) k by (auto intro!: sat_slice_iff)
    also have ... = MFOTL.sat (MFOTL_slicer.slice φ (slice_set k) (replace_prefix π σ′)) (map the v) i φ
      using that complete k by (subst slice_replace_prefix[symmetric]; simp)
    also have ... = MFOTL.sat (replace_prefix π σ′) (map the v) i φ
      using complete(2) k by (auto intro!: sat_slice_iff[symmetric])
    also have ...
      by (rule complete(3)[rule_format], rule prefix_of_replace_prefix[OF that])
    finally show ?thesis .
  qed
  with complete(1−3) obtain π′ where π′:
    prefix_of π′ (MFOTL_slicer.slice φ (slice_set k) σ) (i, v) ∈ M π′
    by (atomize_elim, intro complete_monitor[where π=MFOTL_slicer.pslice φ (slice_set k) π])
      (auto simp: splitter_pslice intro!: prefix_of_pmap_Γ)
  from π′(1) obtain π′′ where π′ = MFOTL_slicer.pslice φ (slice_set k) π′′ prefix_of π′′ σ
    by (atomize_elim, rule prefix_of_map_Γ_D)
  with π′ k show ?case
    by (intro exI[of _ π′′]) (auto simp: joiner_def splitter_pslice intro!: exI[of _ k])
qed
```

7.2.4 Corollary 1

```
sublocale joint_slicer: slicer MFOTL.nfv φ MFOTL.fv φ λσ v i. MFOTL.sat σ v i φ
  λπ. joiner (λk. M (splitter π k)) λ_. M splitter joiner
  by standard (auto simp: mono_splitter)

end
```

7.2.5 Definition 5

Corresponds to locale *sliceable_monitor* defined in theory *MFOTL_Monitor.Abstract_Monitor*.

```
locale slicable_joint_data_slicer =
  sliceable_monitor MFOTL.nfv φ MFOTL.fv φ relevant_events φ λσ v i. MFOTL.sat σ v i φ M +
  joint_data_slicer φ M strategy for φ M strategy
begin

lemma monitor_split: ok φ v ⟹ k ∈ strategy v ⟹ (i, v) ∈ M (splitter π k) ⟷ (i, v) ∈ M π
  unfolding splitter_pslice
  by (rule sliceable_M)
    (auto simp: wf_tuple_def fvi_less_nfv intro!: mem_restrI[rotated 2, where y=map the v])
```

7.2.6 Theorem 2

```
sublocale self_slicer MFOTL.nfv φ MFOTL.fv φ λσ v i. MFOTL.sat σ v i φ M splitter joiner
proof (standard, erule mono_splitter, safe, goal_cases sound complete)
  case (sound π i v)
  have ok φ v using joint_monitor.wf_monitor[OF sound] by auto
  from sound obtain k where (i, v) ∈ M (splitter π k) k ∈ strategy v
```

unfolding *joiner_def* **by** *blast*
with ‹*ok* φ *v*› **show** *?case* **by** (*simp add*: *monitor_split*)
next
case (*complete* π *i v*)
have *ok* φ *v* **using** *wf_monitor*[*OF complete*] **by** *auto*
with *complete strategy_nonempty* **obtain** *k* **where** *k*: $k \in$ *strategy v* **by** *blast*
then have $(i, v) \in M$ (*splitter* π *k*) **using** *complete* ‹*ok* φ *v*› **by** (*simp add*: *monitor_split*)
with *k* **show** *?case* **unfolding** *joiner_def* **by** *blast*
qed

end

7.2.7 Towards Theorem 3

fun *names* :: *'a MFOTL.formula* $\Rightarrow$ *MFOTL.name set* **where**
names (*MFOTL.Pred e _*) = {*e*}
| *names* (*MFOTL.Eq _ _*) = {}
| *names* (*MFOTL.Neg* ψ) = *names* ψ
| *names* (*MFOTL.Or* α β) = *names* α $\cup$ *names* β
| *names* (*MFOTL.Exists* ψ) = *names* ψ
| *names* (*MFOTL.Prev I* ψ) = *names* ψ
| *names* (*MFOTL.Next I* ψ) = *names* ψ
| *names* (*MFOTL.Since* α *I* β) = *names* α $\cup$ *names* β
| *names* (*MFOTL.Until* α *I* β) = *names* α $\cup$ *names* β

fun *gen_unique* :: *'a MFOTL.formula* $\Rightarrow$ *bool* **where**
gen_unique (*MFOTL.Pred _ _*) = *True*
| *gen_unique* (*MFOTL.Eq* (*MFOTL.Var _*) (*MFOTL.Const _*)) = *False*
| *gen_unique* (*MFOTL.Eq* (*MFOTL.Const _*) (*MFOTL.Var _*)) = *False*
| *gen_unique* (*MFOTL.Eq _ _*) = *True*
| *gen_unique* (*MFOTL.Neg* ψ) = *gen_unique* ψ
| *gen_unique* (*MFOTL.Or* α β) = (*gen_unique* α $\wedge$ *gen_unique* β $\wedge$ *names* α $\cap$ *names* β = {})
| *gen_unique* (*MFOTL.Exists* ψ) = *gen_unique* ψ
| *gen_unique* (*MFOTL.Prev I* ψ) = *gen_unique* ψ
| *gen_unique* (*MFOTL.Next I* ψ) = *gen_unique* ψ
| *gen_unique* (*MFOTL.Since* α *I* β) = (*gen_unique* α $\wedge$ *gen_unique* β $\wedge$ *names* α $\cap$ *names* β = {})
| *gen_unique* (*MFOTL.Until* α *I* β) = (*gen_unique* α $\wedge$ *gen_unique* β $\wedge$ *names* α $\cap$ *names* β = {})

lemma *sat_inter_names_cong*: ($\bigwedge$*e*. *e* $\in$ *names* φ $\Longrightarrow$ {*xs*. (*e*, *xs*) $\in$ *E*} = {*xs*. (*e*, *xs*) $\in$ *F*}) $\Longrightarrow$
MFOTL.sat (*map_*Γ (λD. $D \cap E$) σ) *v i* φ $\longleftrightarrow$ *MFOTL.sat* (*map_*Γ (λD. $D \cap F$) σ) *v i* φ
by (*induction* φ *arbitrary*: *v i*) (*auto split*: *nat.splits*)

lemma *matches_in_names*: *MFOTL.matches v* φ *x* $\Longrightarrow$ *fst x* $\in$ *names* φ
by (*induction* φ *arbitrary*: *v*) (*auto*)

lemma *unique_names_matches_absorb*: *fst x* $\in$ *names* α $\Longrightarrow$ *names* α $\cap$ *names* β = {} $\Longrightarrow$
MFOTL.matches v α *x* $\vee$ *MFOTL.matches v* β *x* $\longleftrightarrow$ *MFOTL.matches v* α *x*
fst x $\in$ *names* β $\Longrightarrow$ *names* α $\cap$ *names* β = {} $\Longrightarrow$
MFOTL.matches v α *x* $\vee$ *MFOTL.matches v* β *x* $\longleftrightarrow$ *MFOTL.matches v* β *x*
by (*auto dest*: *matches_in_names*)

definition *mergeable_envs* **where**
mergeable_envs n S $\longleftrightarrow$ ($\forall v1 \in S$. $\forall v2 \in S$. ($\forall$ *A B f*.
($\forall x \in A$. $x < n \wedge v1\ !\ x = f\,x$) $\wedge$ ($\forall x \in B$. $x < n \wedge v2\ !\ x = f\,x$) $\longrightarrow$
($\exists v \in S$. $\forall x \in A \cup B$. $v\ !\ x = f\,x$)))

lemma *mergeable_envsI*:
assumes $\bigwedge$*v1 v2 v*. $v1 \in S \Longrightarrow v2 \in S \Longrightarrow$ *length* $v = n \Longrightarrow \forall x < n.\ v\ !\ x = v1\ !\ x \vee v\ !\ x = v2\ !\ x$
$\Longrightarrow v \in S$

shows *mergeable_envs n S*
unfolding *mergeable_envs_def*
proof (*safe*, *goal_cases mergeable*)
case [*simp*]: (*mergeable v1 v2 A B f*)
let *?v* = *tabulate* (λx. *if* $x \in A \cup B$ *then f x else v1* ! *x*) *0 n*
from *assms*[*of v1 v2 ?v*, *simplified*] **show** *?case*
by (*auto intro*!: *bexI*[*of _ ?v*])
qed

lemma *in_listset_nth*: $x \in$ *listset As* $\Longrightarrow i <$ *length As* $\Longrightarrow x$! $i \in$ *As* ! *i*
by (*induction As arbitrary*: *x i*) (*auto simp*: *set_Cons_def nth_Cons split*: *nat.split*)

lemma *all_nth_in_listset*: *length x* = *length As* $\Longrightarrow$ ($\bigwedge i$. $i <$ *length As* $\Longrightarrow x$! $i \in$ *As* ! *i*) $\Longrightarrow x \in$ *listset As*
by (*induction x As rule*: *list_induct2*) (*fastforce simp*: *set_Cons_def nth_Cons*)+

lemma *mergeable_envs_listset*: *mergeable_envs* (*length As*) (*listset As*)
by (*rule mergeable_envsI*) (*auto intro*!: *all_nth_in_listset elim*!: *in_listset_nth*)

lemma *mergeable_envs_Ex*: *mergeable_envs n S* $\Longrightarrow$ *MFOTL.nfv* $\alpha \le n \Longrightarrow$ *MFOTL.nfv* $\beta \le n \Longrightarrow$
($\exists v' \in S$. $\forall x \in fv\ \alpha$. v' ! $x = v$! x) $\Longrightarrow$ ($\exists v' \in S$. $\forall x \in fv\ \beta$. v' ! $x = v$! x) $\Longrightarrow$
($\exists v' \in S$. $\forall x \in fv\ \alpha \cup fv\ \beta$. v' ! $x = v$! x)
proof (*clarify*, *goal_cases mergeable*)
case (*mergeable v1 v2*)
then show *?case*
by (*auto intro*: *order.strict_trans2*[*OF fvi_less_nfv*[*rule_format*]]
elim!: *mergeable_envs_def*[*THEN iffD1*, *rule_format*, *of _ _ v1 v2*])
qed

lemma *in_set_ConsE*: $xs \in$ *set_Cons A As* $\Longrightarrow$ ($\bigwedge y\ ys$. $xs = y \# ys \Longrightarrow y \in A \Longrightarrow ys \in As \Longrightarrow P$) $\Longrightarrow P$
unfolding *set_Cons_def* **by** *blast*

lemma *mergeable_envs_set_Cons*: *mergeable_envs n S* $\Longrightarrow$ *mergeable_envs* (*Suc n*) (*set_Cons UNIV S*)
unfolding *mergeable_envs_def*
proof (*clarify*, *elim in_set_ConsE*, *goal_cases mergeable*)
case (*mergeable v1 v2 A B f y1 ys1 y2 ys2*)
let *?A* = (λx. $x - 1$) ‘ ($A - \{0\}$)
let *?B* = (λx. $x - 1$) ‘ ($B - \{0\}$)
from *mergeable*(*4*−*9*) **have** $\exists v \in S$. $\forall x \in ?A \cup ?B$. v ! $x = f$ (*Suc x*)
by (*auto dest*!: *mergeable*(*2*,*3*)[*rule_format*] *intro*!: *mergeable*(*1*)[*rule_format*, *of ys1 ys2*])
then obtain *v* **where** $v \in S$ $\forall x \in ?A \cup ?B$. v ! $x = f$ (*Suc x*) **by** *blast*
then show *?case*
by (*intro bexI*[*of _ f 0 # v*]) (*auto simp*: *nth_Cons'* *set_Cons_def*)
qed

lemma *slice_Exists*: *MFOTL_slicer.slice* (*MFOTL.Exists* φ) *S* σ = *MFOTL_slicer.slice* φ (*set_Cons UNIV S*) σ
by (*auto simp*: *set_Cons_def intro*: *map_Γ_cong*)

lemma *image_Suc_fvi*: *Suc* ‘ *MFOTL.fvi* (*Suc b*) φ = *MFOTL.fvi b* $\varphi - \{0\}$
by (*auto simp*: *image_def Bex_def MFOTL.fvi_Suc dest*: *gr0_implies_Suc*)

lemma *nfv_Exists*: *MFOTL.nfv* (*MFOTL.Exists* φ) = *MFOTL.nfv* $\varphi - 1$
unfolding *MFOTL.nfv_def*
by (*cases fv* $\varphi = \{\}$) (*auto simp add*: *image_Suc_fvi mono_Max_commute*[*symmetric*] *mono_def*)

lemma *set_Cons_empty_iff* [*simp*]: *set_Cons A Xs* = {} ⟷ *A* = {} ∨ *Xs* = {}
  unfolding *set_Cons_def* **by** *auto*

lemma *unique_sat_slice_mem*: *safe_formula* φ ⟹ *gen_unique* φ ⟹ *S* ≠ {} ⟹
  mergeable_envs n S ⟹ *MFOTL.nfv* φ ≤ *n* ⟹
  MFOTL.sat (*MFOTL_slicer.slice* φ *S* σ) *v i* φ ⟹ ∃ *v*′∈*S*. ∀ *x*∈*fv* φ. *v*′ ! *x* = *v* ! *x*
proof (*induction arbitrary*: *v i S n rule*: *safe_formula_induct*)
  case (*1 t1 t2*)
  then show *?case* **by** (*cases t2*) (*auto simp*: *MFOTL.is_Const_def*)
next
  case (*2 t1 t2*)
  then show *?case* **by** (*cases t1*) (*auto simp*: *MFOTL.is_Const_def*)
next
  case (*3 x y*)
  then show *?case* **by** *auto*
next
  case (*4 x y*)
  then show *?case* **by** *simp*
next
  case (*5 e ts*)
  then obtain *v*′ **where** *v*′ ∈ *S* **and** *eq*: ∀ *t*∈*set ts*. *MFOTL.eval_trm v*′ *t* = *MFOTL.eval_trm v t*
    by *auto*
  have ∀ *t*∈*set ts*. ∀ *x*∈*fv_trm t*. *v*′ ! *x* = *v* ! *x* **proof**
    fix *t* **assume** *t* ∈ *set ts*
    with *eq* **have** *MFOTL.eval_trm v*′ *t* = *MFOTL.eval_trm v t* **..**
    then show ∀ *x*∈*fv_trm t*. *v*′ ! *x* = *v* ! *x* **by** (*cases t*) (*simp_all*)
  qed
  with ‹*v*′ ∈ *S*› **show** *?case* **by** *auto*
next
  case (*6* φ ψ)
  from ‹*gen_unique* (*MFOTL.And* φ ψ)›
  have
    MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.And* φ ψ) *S* σ) *v i* φ = *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) *v i* φ
    MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.And* φ ψ) *S* σ) *v i* ψ = *MFOTL.sat* (*MFOTL_slicer.slice* ψ *S* σ) *v i* ψ
    unfolding *MFOTL.And_def*
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)+
  with *6*(*1*,*4*−) *6*(*2*,*3*)[**where** *S*=*S*] **show** *?case*
    unfolding *MFOTL.And_def*
    by (*auto intro*!: *mergeable_envs_Ex*)
next
  case (*7* φ ψ)
  from ‹*gen_unique* (*MFOTL.And_Not* φ ψ)›
  have *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.And_Not* φ ψ) *S* σ) *v i* φ = *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) *v i* φ
    unfolding *MFOTL.And_Not_def*
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)
  with *7*(*1*,*2*,*5*−) *7*(*3*)[**where** *S*=*S*] **have** ∃ *v*′∈*S*. ∀ *x*∈*fv* φ. *v*′ ! *x* = *v* ! *x*
    unfolding *MFOTL.And_Not_def* **by** *auto*
  with ‹*fv* ψ ⊆ *fv* φ› **show** *?case* **by** (*auto simp*: *MFOTL.fvi_And_Not*)
next
  case (*8* φ ψ)
  from ‹*gen_unique* (*MFOTL.Or* φ ψ)›
  have
    MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Or* φ ψ) *S* σ) *v i* φ = *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) *v i* φ
    MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Or* φ ψ) *S* σ) *v i* ψ = *MFOTL.sat* (*MFOTL_slicer.slice*

ψ S σ) v i ψ
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)+
  with *8*(*1*,*4*−) *8*(*2*,*3*)[**where** S=S] **have** $\exists\, v'\in S.\ \forall\, x\in fv\ \varphi.\ v'\ !\ x = v\ !\ x$
    by (*auto simp*: ‹*fv* ψ = *fv* φ›)
  then show *?case* **by** (*auto simp*: ‹*fv* ψ = *fv* φ›)
next
  case (*9* φ)
  then obtain *z* **where** *sat_*φ: *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.Exists* φ) S σ) (z # v) i φ
    by *auto*
  from *9.prems sat_*φ **have** $\exists\, v'\in$*set_Cons UNIV S*. $\forall\, x\in fv\ \varphi.\ v'\ !\ x = (z\ \#\ v)\ !\ x$
    unfolding *slice_Exists*
    by (*intro 9.IH*) (*auto simp*: *nfv_Exists intro*!: *mergeable_envs_set_Cons*)
  then show *?case*
    by (*auto simp*: *set_Cons_def fvi_Suc Ball_def nth_Cons split*: *nat.splits*)
next
  case (*10* I φ)
  then obtain *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* φ S σ) v j φ
    by (*auto split*: *nat.splits*)
  with *10* **show** *?case* **by** *simp*
next
  case (*11* I φ)
  then obtain *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* φ S σ) v j φ
    by (*auto split*: *nat.splits*)
  with *11* **show** *?case* **by** *simp*
next
  case (*12* φ I ψ)
  from ‹*gen_unique* (*MFOTL.Since* φ I ψ)›
  have ∗:
   MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Since* φ I ψ) S σ) v j ψ = *MFOTL.sat* (*MFOTL_slicer.slice* ψ S σ) v j ψ **for** *j*
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)
  from *12* **obtain** *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.Since* φ I ψ) S σ) v j ψ
    by *auto*
  with *12* **have** $\exists\, v'\in S.\ \forall\, x\in fv\ \psi.\ v'\ !\ x = v\ !\ x$ **using** ∗ **by** *auto*
  with ‹*fv* $\varphi \subseteq$ *fv* ψ› **show** *?case* **by** *auto*
next
  case (*13* φ I ψ)
  from ‹*gen_unique* (*MFOTL.Since* (*MFOTL.Neg* φ) I ψ)›
  have ∗:
    MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Since* (*MFOTL.Neg* φ) I ψ) S σ) v j ψ = *MFOTL.sat* (*MFOTL_slicer.slice* ψ S σ) v j ψ **for** *j*
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)
  from *13* **obtain** *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.Since* (*MFOTL.Neg* φ) I ψ) S σ) v j ψ
    by *auto*
  with *13* **have** $\exists\, v'\in S.\ \forall\, x\in fv\ \psi.\ v'\ !\ x = v\ !\ x$ **using** ∗ **by** *auto*
  with ‹*fv* (*MFOTL.Neg* φ) $\subseteq$ *fv* ψ› **show** *?case* **by** *auto*
next
  case (*14* φ I ψ)
  from ‹*gen_unique* (*MFOTL.Until* φ I ψ)›
  have ∗:
   MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Until* φ I ψ) S σ) v j ψ = *MFOTL.sat* (*MFOTL_slicer.slice* ψ S σ) v j ψ **for** *j*
    by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)
  from *14* **obtain** *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.Until* φ I ψ) S σ) v j ψ
    by *auto*
  with *14* **have** $\exists\, v'\in S.\ \forall\, x\in fv\ \psi.\ v'\ !\ x = v\ !\ x$ **using** ∗ **by** *auto*
  with ‹*fv* $\varphi \subseteq$ *fv* ψ› **show** *?case* **by** *auto*

next
case (*15* φ *I* ψ)
from ‹*gen_unique* (*MFOTL.Until* (*MFOTL.Neg* φ) *I* ψ)›
have ∗:
MFOTL.sat (*MFOTL_slicer.slice* (*MFOTL.Until* (*MFOTL.Neg* φ) *I* ψ) *S* σ) *v j* ψ = *MFOTL.sat* (*MFOTL_slicer.slice* ψ *S* σ) *v j* ψ **for** *j*
by (*fastforce simp*: *unique_names_matches_absorb intro*!: *sat_inter_names_cong*)
from *15* **obtain** *j* **where** *MFOTL.sat* (*MFOTL_slicer.slice* (*MFOTL.Until* (*MFOTL.Neg* φ) *I* ψ) *S* σ) *v j* ψ
by *auto*
with *15* **have** $\exists\, v' {\in} S.\ \forall\, x {\in} fv\ \psi.\ v'\ !\ x = v\ !\ x$ **using** ∗ **by** *auto*
with ‹*fv* (*MFOTL.Neg* φ) ⊆ *fv* ψ› **show** *?case* **by** *auto*
qed

lemma *unique_sat_slice*:
assumes *formula*: *safe_formula* φ *gen_unique* φ
and *restr*: *S* ≠ {} *mergeable_envs* (*MFOTL.nfv* φ) *S*
and *sat_slice*: *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) *v i* φ
shows *MFOTL.sat* σ *v i* φ
proof −
obtain v' **where** $v' \in S$ **and** *fv_eq*: $\forall\, x {\in} fv\ \varphi.\ v'\ !\ x = v\ !\ x$
using *unique_sat_slice_mem*[*OF formula restr order_refl sat_slice*] **..**
with *sat_slice* **have** *MFOTL.sat* (*MFOTL_slicer.slice* φ *S* σ) v' *i* φ
by (*auto iff*: *sat_fvi_cong*)
then have *MFOTL.sat* σ v' *i* φ
unfolding *sat_slice_iff*[*OF* ‹$v' \in S$›, *symmetric*] **.**
with *fv_eq* **show** *?thesis* **by** (*auto iff*: *sat_fvi_cong*)
qed

7.2.8 Lemma 3

lemma (**in** *splitting_strategy*) *unique_sat_strategy*:
safe_formula φ ⟹ *gen_unique* φ ⟹ *slice_set k* ≠ {} ⟹
mergeable_envs (*MFOTL.nfv* φ) (*slice_set k*) ⟹
MFOTL.sat (*MFOTL_slicer.slice* φ (*slice_set k*) σ) (*map the v*) *i* φ ⟹
ok φ *v* ⟹ *k* ∈ *strategy v*
by (*drule* (*3*) *unique_sat_slice_mem*) (*auto dest*: *wf_tuple_cong*)

locale *skip_inter* = *joint_data_slicer* +
assumes *nonempty*: *slice_set k* ≠ {}
and *mergeable*: *mergeable_envs* (*MFOTL.nfv* φ) (*slice_set k*)
begin

7.2.9 Definition of J'

definition *skip_joiner* = ($\lambda s.\ \bigcup k.\ s\ k$)

7.2.10 Theorem 3

lemma *skip_joiner*:
assumes *safe_formula* φ *gen_unique* φ
shows *joiner* (λk. *M* (*splitter* π *k*)) = *skip_joiner* (λk. *M* (*splitter* π *k*))
(**is** *?L* = *?R*)
proof *safe*
fix *i v*
assume (*i*, *v*) ∈ *?R*
then obtain *k* **where** *in_M*: (*i*, *v*) ∈ *M* (*splitter* π *k*)
unfolding *skip_joiner_def* **by** *blast*
from *ex_prefix_of* **obtain** σ **where** *prefix_of* π σ **by** *blast*

with *wf_monitor*[*OF in_M*] *sound_monitor*[*OF in_M*] **have**
MFOTL.sat (*MFOTL_slicer.slice* φ (*slice_set k*) σ) (*map the v*) *i* φ *ok* φ *v*
by (*auto simp*: *splitter_pslice intro*!: *prefix_of_pmap_*Γ)
note *unique_sat_strategy*[*OF assms nonempty mergeable this*]
with *in_M* **show** (i, v) $\in$ *?L* **unfolding** *joiner_def* **by** *blast*
qed (*auto simp*: *joiner_def skip_joiner_def*)

sublocale *skip_joint_monitor*: *MFOTL_monitor* φ
$\lambda\pi$. (*if safe_formula* φ $\wedge$ *gen_unique* φ *then skip_joiner else joiner*) (λk. *M* (*splitter* π *k*))
using *joint_monitor.mono_monitor joint_monitor.wf_monitor joint_monitor.sound_monitor joint_monitor.complete_mo*
by *unfold_locales* (*auto simp*: *skip_joiner*[*symmetric*] *split*: *if_splits*)

end

References

[1] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2):15:1–15:45, 2015.

[2] D. Basin, F. Klaedtke, and E. Zălinescu. The MonPoly monitoring tool. In G. Reger and K. Havelund, editors, *RV-CuBES 2017*, volume 3 of *Kalpa Publications in Computing*, pages 19–28. EasyChair, 2017.

[3] J. Schneider, D. Basin, F. Brix, S. Krstić, and D. Traytel. Scalable online first-order monitoring. *Int. J. Softw. Tools Technol. Transf.*, 2020. To appear. Preprint at http://people.inf.ethz.ch/trayteld/papers/sttt20-som_long/som_long.pdf.

[4] J. Schneider, D. Basin, S. Krstić, and D. Traytel. A formally verified monitor for metric first-order temporal logic. In B. Finkbeiner and L. Mariani, editors, *RV 2019*, volume 11757 of *LNCS*, pages 310–328. Springer, 2019.