

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

Joshua Schneider Dmitriy Traytel

April 19, 2024

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	2
1.1	Infinite traces	2
1.2	Finite trace prefixes	4
2	Finite tables	9
3	Abstract monitors and slicing	15
3.1	First-order specifications	15
3.2	Monitor function	16
3.3	Slicing	17
4	Intervals	20
5	Metric first-order temporal logic	21
5.1	Formulas and satisfiability	21
5.2	Defined connectives	25
5.3	Safe formulas	25
5.4	Slicing traces	26
6	Monitor implementation	28
6.1	Monitorable formulas	28
6.2	The executable monitor	29
6.3	Progress	32
6.4	Specification	36
6.5	Correctness	37
6.5.1	Invariants	37
6.5.2	Initialisation	39
6.5.3	Evaluation	40

6.5.4	Monitor step	53
6.5.5	Monitor function	53
6.6	Collected correctness results	55
7	Slicing framework	56
7.1	Abstract slicing	56
7.1.1	Definition 1	56
7.1.2	Definition 2	56
7.1.3	Definition 3	56
7.1.4	Lemma 1	56
7.2	Joint data slicer	56
7.2.1	Definition 4	57
7.2.2	Lemma 2	57
7.2.3	Theorem 1	57
7.2.4	Corollary 1	58
7.2.5	Definition 5	58
7.2.6	Theorem 2	58
7.2.7	Towards Theorem 3	59
7.2.8	Lemma 3	63
7.2.9	Definition of J'	63
7.2.10	Theorem 3	63

1 Traces and trace prefixes

1.1 Infinite traces

coinductive *ssorted* :: 'a :: linorder stream \Rightarrow bool **where**
shd $s \leq \text{shd } (stl\ s) \Longrightarrow \text{ssorted } (stl\ s) \Longrightarrow \text{ssorted } s$

lemma *ssorted_siterate[simp]*: $(\bigwedge n. n \leq f\ n) \Longrightarrow \text{ssorted } (\text{siterate } f\ n)$
by (*coinduction arbitrary: n*) *auto*

lemma *ssortedD*: $\text{ssorted } s \Longrightarrow s\ !!\ i \leq stl\ s\ !!\ i$
by (*induct i arbitrary: s*) (*auto elim: ssorted.cases*)

lemma *ssorted_sdrop*: $\text{ssorted } s \Longrightarrow \text{ssorted } (\text{sdrop } i\ s)$
by (*coinduction arbitrary: i s*) (*auto elim: ssorted.cases ssortedD*)

lemma *ssorted_monoD*: $\text{ssorted } s \Longrightarrow i \leq j \Longrightarrow s\ !!\ i \leq s\ !!\ j$

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*: $\text{ssorted } s \Longrightarrow \text{sorted } (\text{stake } i\ s)$

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 *sorted_monoI*: $\forall i\ j. i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j \Longrightarrow \text{ssorted } s$

by (*coinduction arbitrary: s*)

(*auto dest: spec2[of _ Suc _ Suc _] spec2[of _ 0 Suc 0]*)

lemma *sorted_iff_mono*: $\text{ssorted } s \longleftrightarrow (\forall i\ j. i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j)$

using *ssorted_monoI ssorted_monoD by metis*

lemma *sorted_iff_le_Suc*: $\text{sorted } s \longleftrightarrow (\forall i. s !! i \leq s !! \text{Suc } i)$
using *mono_iff_le_Suc*[of *snth s*] **by** (*simp add: mono_def sorted_iff_mono*)

definition *sincreasing* $s = (\forall x. \exists i. x < s !! i)$

lemma *sincreasingI*: $(\bigwedge x. \exists i. x < s !! i) \implies \text{sincreasing } s$
by (*simp add: increasing_def*)

lemma *sincreasing_grD*:

fixes $x :: 'a :: \text{semilattice_sup}$

assumes *sincreasing* s

shows $\exists j > i. x < s !! j$

proof –

let $?A = \text{insert } x \{s !! n \mid n. n \leq i\}$

from *assms* **obtain** j **where** $*$: $\text{Sup_fin } ?A < s !! j$

by (*auto simp: increasing_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 $\neg i < j$

then have $s !! j \in ?A$ **by** (*auto simp: not_less*)

then have $s !! j \leq \text{Sup_fin } ?A$

by (*auto intro: Sup_fin.coboundedI*)

with $*$ **show** *False* **by** *simp*

qed

ultimately show *?thesis* **by** *blast*

qed

lemma *sincreasing_siterate_nat*[*simp*]:

fixes $n :: \text{nat}$

assumes $(\bigwedge n. n < f n)$

shows *sincreasing* $(\text{siterate } f n)$

unfolding *increasing_def* **proof**

fix x

show $\exists i. x < \text{siterate } f n !! i$

proof (*induction x*)

case 0

have $0 < \text{siterate } f n !! 1$

using *order.strict_trans1*[*OF le0 assms*] **by** *simp*

then show *?case ..*

next

case $(\text{Suc } x)$

then obtain i **where** $x < \text{siterate } f n !! i ..$

then have $\text{Suc } x < \text{siterate } f n !! \text{Suc } i$

using *order.strict_trans1*[*OF _ assms*] **by** (*simp del: snth.simps*)

then show *?case ..*

qed

qed

lemma *sincreasing_stl*: *sincreasing* $s \implies \text{sincreasing } (\text{stl } s)$ **for** $s :: 'a :: \text{semilattice_sup}$ *stream*

by (*auto 0 4 simp: gr0_conv_Suc intro!: increasingI dest: increasing_grD*[of $s 0$])

typedef $'a$ *trace* = $\{s :: ('a \text{ set} \times \text{nat}) \text{ stream}. \text{sorted } (\text{smap snd } s) \wedge \text{increasing } (\text{smap snd } s)\}$

by (*intro exI*[of *_ smap*] $(\lambda i. (\{\}, i)) \text{ nats}$)

(*auto simp: stream.map_comp stream.map_ident cong: stream.map_cong*)

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **is**
 $\lambda s i. \text{fst } (s !! i) .$

lift_definition $\tau :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **is**
 $\lambda s i. \text{snd } (s !! i) .$

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

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

lemma $\tau_mono[simp]$: $i \leq j \implies \tau s i \leq \tau s j$
by *transfer (auto simp: sorted_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*: $\tau \sigma i \leq \tau \sigma j \implies j < i \implies \tau \sigma i = \tau \sigma j$
by (*simp add: antisym*)

lemma *less_τD*: $\tau \sigma i < \tau \sigma j \implies i < j$
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* :: $('a \text{ set} \Rightarrow 'b \text{ set}) \Rightarrow 'a \text{ trace} \Rightarrow 'b \text{ trace}$ **is**
 $\lambda f s. \text{smap } (\lambda(x, i). (f x, i)) s$
by (*auto simp: stream.map_comp prod.case_eq_if cong: stream.map_cong*)

lemma $\Gamma_map_Γ[simp]$: $\Gamma (\text{map}_\Gamma f s) i = f (\Gamma s i)$
by *transfer (simp add: prod.case_eq_if)*

lemma $\tau_map_Γ[simp]$: $\tau (\text{map}_\Gamma f s) i = \tau s i$
by *transfer (simp add: prod.case_eq_if)*

lemma *map_Γ_id[simp]*: $\text{map}_\Gamma \text{id } s = s$
by *transfer (simp add: stream.map_id)*

lemma *map_Γ_comp*: $\text{map}_\Gamma g (\text{map}_\Gamma f s) = \text{map}_\Gamma (g \circ f) s$
by *transfer (simp add: stream.map_comp comp_def prod.case_eq_if case_prod_beta')*

lemma *map_Γ_cong*: $\sigma_1 = \sigma_2 \implies (\bigwedge x. f_1 x = f_2 x) \implies \text{map}_\Gamma f_1 \sigma_1 = \text{map}_\Gamma f_2 \sigma_2$
by *transfer (auto intro!: stream.map_cong)*

1.2 Finite trace prefixes

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

setup_lifting *type_definition_prefix*

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

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

$\lambda p. (\text{case } p \text{ of } [] \Rightarrow 0 \mid _ \Rightarrow \text{snd } (\text{last } p)) .$

lift_definition *first_ts* :: *nat* \Rightarrow *'a prefix* \Rightarrow *nat* **is**
 $\lambda n p. (\text{case } p \text{ of } [] \Rightarrow n \mid _ \Rightarrow \text{snd } (\text{hd } p)) .$

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

lift_definition *plen* :: *'a prefix* \Rightarrow *nat* **is** *length* .

lift_definition *psnoc* :: *'a prefix* \Rightarrow *'a set* \times *nat* \Rightarrow *'a prefix* **is**
 $\lambda p x. \text{if } (\text{case } p \text{ of } [] \Rightarrow 0 \mid _ \Rightarrow \text{snd } (\text{last } p)) \leq \text{snd } x \text{ then } p @ [x] \text{ else } []$

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* \Rightarrow *'a prefix* \Rightarrow *bool* **is**
 $\lambda p q. \exists r. q = p @ r .$

definition *less_prefix* :: *'a prefix* \Rightarrow *'a prefix* \Rightarrow *bool* **where**
less_prefix x y = (x \leq y \wedge \neg y \leq 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 \leq snd x \implies last_ts q \leq snd y \implies psnoc p x = psnoc q y \longleftrightarrow (p = q \wedge x = y)

by *transfer auto*

lift_definition *prefix_of* :: *'a prefix* \Rightarrow *'a trace* \Rightarrow *bool* **is** $\lambda p s. \text{stake } (\text{length } p) s = p .$

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 $\pi \sigma \implies$ prefix_of (pmap_Γ f π) (map_Γ f σ)*

by *transfer auto*

lemma *plen_mono*: $\pi \leq \pi' \implies \text{plen } \pi \leq \text{plen } \pi'$

by transfer auto

lemma *prefix_of_psnocE*: $\text{prefix_of } (\text{psnoc } p \ x) \ s \implies \text{last_ts } p \leq \text{snd } x \implies$
 $(\text{prefix_of } p \ s \implies \Gamma \ s \ (\text{plen } p) = \text{fst } x \implies \tau \ s \ (\text{plen } p) = \text{snd } x \implies P) \implies P$
by transfer (simp del: stake.simps add: stake_Suc)

lemma *le_pnil[simp]*: $\text{pnil} \leq \pi$
by transfer auto

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

lemma *plen_take_prefix[simp]*: $\text{plen } (\text{take_prefix } i \ \sigma) = i$
by transfer auto

lemma *plen_psnoc[simp]*: $\text{last_ts } \pi \leq \text{snd } x \implies \text{plen } (\text{psnoc } \pi \ x) = \text{plen } \pi + 1$
by transfer auto

lemma *prefix_of_take_prefix[simp]*: $\text{prefix_of } (\text{take_prefix } i \ \sigma) \ \sigma$
by transfer auto

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

lemma *pdrop_0[simp]*: $\text{pdrop } 0 \ \pi = \pi$
by transfer auto

lemma *prefix_of_antimono*: $\pi \leq \pi' \implies \text{prefix_of } \pi' \ s \implies \text{prefix_of } \pi \ s$
by transfer (auto simp del: stake_add simp add: stake_add[symmetric])

lemma *prefix_of_imp_linear*: $\text{prefix_of } \pi \ \sigma \implies \text{prefix_of } \pi' \ \sigma \implies \pi \leq \pi' \vee \pi' \leq \pi$
proof transfer

fix $\pi \ \pi' \ \text{and } \sigma :: ('a \ \text{set} \times \text{nat}) \ \text{stream}$

assume *assms*: $\text{stake } (\text{length } \pi) \ \sigma = \pi \ \text{stake } (\text{length } \pi') \ \sigma = \pi'$

show $(\exists r. \pi' = \pi \ @ \ r) \vee (\exists r. \pi = \pi' \ @ \ r)$

proof (cases $\text{length } \pi \ \text{length } \pi'$ rule: *le_cases*)

case *le*

then have $\pi' = \text{take } (\text{length } \pi) \ \pi' \ @ \ \text{drop } (\text{length } \pi) \ \pi'$

by *simp*

moreover have $\text{take } (\text{length } \pi) \ \pi' = \pi$

using *assms le* by (metis *min.absorb1 take_stake*)

ultimately show *?thesis* by *auto*

next

case *ge*

then have $\pi = \text{take } (\text{length } \pi') \ \pi \ @ \ \text{drop } (\text{length } \pi') \ \pi$

by *simp*

moreover have $\text{take } (\text{length } \pi') \ \pi = \pi'$

using *assms ge* by (metis *min.absorb1 take_stake*)

ultimately show *?thesis* by *auto*

qed

qed

lemma *ex_prefix_of*: $\exists s. \text{prefix_of } p \ s$

proof (transfer, intro *best CollectI conjI*)

fix $p :: ('a \ \text{set} \times \text{nat}) \ \text{list}$

assume ***: $\text{sorted } (\text{map } \text{snd } p)$

let $?s = p \ @- \ \text{smap } (\text{Pair } \text{undefined}) \ (\text{fromN } (\text{snd } (\text{last } p)))$

show $\text{stake } (\text{length } p) \ ?s = p$ 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 sorted_shift:
  sorted (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 sorted (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 $x < (xs @- s) !! (\text{length } xs + i)$
by *simp*
then show $\exists i. x < (xs @- s) !! i ..$

qed

lift_definition *replace_prefix* :: 'a prefix \Rightarrow 'a trace \Rightarrow 'a trace **is**

$\lambda \pi \sigma. \text{if } \text{ssorted } (\text{smap } \text{snd } (\pi @- \text{sdrop } (\text{length } \pi) \sigma)) \text{ then}$
 $\pi @- \text{sdrop } (\text{length } \pi) \sigma \text{ else } \text{smap } (\lambda i. (\{\}, i)) \text{ nats}$

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_comp)

lemma *prefix_of_replace_prefix*:

prefix_of (*pmap* $_{\Gamma}$ f π) $\sigma \Longrightarrow \text{prefix_of } \pi (\text{replace_prefix } \pi \sigma)$

proof (*transfer; safe; goal_cases*)

case ($1 f \pi \sigma$)

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_ $_{\Gamma}$ _replace_prefix*:

$\forall x. f (f x) = f x \Longrightarrow \text{prefix_of } (\text{pmap_}\Gamma f \pi) \sigma \Longrightarrow \text{map_}\Gamma f (\text{replace_prefix } \pi \sigma) = \text{map_}\Gamma f \sigma$

proof (*transfer; safe; goal_cases*)

case ($1 f \pi \sigma$)

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_comp)

qed

lemma *prefix_of_pmap_ $_{\Gamma}$ _D*:

assumes *prefix_of* (*pmap* $_{\Gamma}$ f π) σ

shows $\exists \sigma'. \text{prefix_of } \pi \sigma' \wedge \text{prefix_of } (\text{pmap_}\Gamma f \pi) (\text{map_}\Gamma f \sigma')$

proof –

from *assms*(1) **obtain** σ' **where** $1: \text{prefix_of } \pi \sigma'$

using *ex_prefix_of* **by** *blast*

then have *prefix_of* (*pmap* $_{\Gamma}$ f π) (*map* $_{\Gamma}$ f σ')

by *transfer simp*

with 1 **show** ?*thesis* **by** *blast*

qed

lemma *prefix_of_map_ $_{\Gamma}$ _D*:

assumes *prefix_of* $\pi' (\text{map_}\Gamma f \sigma)$

shows $\exists \pi''. \pi' = \text{pmap_}\Gamma f \pi'' \wedge \text{prefix_of } \pi'' \sigma$

using *assms*

by *transfer (auto intro!: exI[of _ stake (length _) _] elim: sym dest: sorted_stake)*

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

lemma *pts_pmap_ $_{\Gamma}$ [simp]*: *pts* (*pmap* $_{\Gamma}$ 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]*: $k < n \implies \text{tabulate } f \ x \ n \ ! \ k = f \ (x + k)$
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 ($\forall i < n. x!i = \text{None} \longleftrightarrow i \notin V$)

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

definition *empty_table* = {}

definition *unit_table* n = {replicate n None}

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

lemma *in_empty_table[simp]*: $\neg x \in \text{empty_table}$
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]*: $V = \{\}$ $\implies x \in \text{unit_table } n \implies \text{wf_tuple } n \ V \ x$
unfolding *unit_table_def* *wf_tuple_def* **by** simp

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

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

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

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

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

lemma *wf_tuple_length*: $wf_tuple\ n\ V\ x \implies length\ x = n$
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]: $join\ A\ True\ B = (\bigcup a \in A. \bigcup b \in B. set_option\ (join1\ (a, b)))$
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*: $join1\ (xs, ys) \neq Some\ xs \implies join1\ (zs, ys) \neq Some\ xs$
by (induct (zs, ys) arbitrary: zs ys xs rule: *join1.induct*; auto; auto)

lemma *join_False_code*[code]: $join\ A\ False\ B = \{a \in A. \forall b \in B. join1\ (a, b) \neq Some\ a\}$
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'*: $Suc\ (x - Suc\ 0) = (case\ x\ of\ 0 \Rightarrow Suc\ 0\ |\ _ \Rightarrow x)$
by (auto *split*: nat.splits)

lemma *wf_tuple_Cons*[*simp*]:
 $wf_tuple\ n\ A\ (x \# xs) \longleftrightarrow ((if\ x = None\ then\ 0 \notin A\ else\ 0 \in A) \wedge$
 $(\exists m. n = Suc\ m \wedge wf_tuple\ m\ ((\lambda x. x - 1) ' (A - \{0\}))\ xs))$
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*:
 $join1\ (v1, v2) = Some\ v \implies wf_tuple\ n\ A\ v1 \implies wf_tuple\ n\ B\ v2 \implies wf_tuple\ n\ (A \cup B)\ v$
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*: $x \in join\ X\ b\ Y \implies$
 $\forall v \in X. wf_tuple\ n\ A\ v \implies \forall v \in Y. wf_tuple\ n\ B\ v \implies (\neg b \implies B \subseteq A) \implies A \cup B = C \implies$
 $wf_tuple\ n\ C\ x$
unfolding *join_def*
by (fastforce *simp*: Option.these_def image_iff sup_absorb1 dest: *join1_wf_tuple* *split*: if_splits)

lemma *join_table*: $table\ n\ A\ X \implies table\ n\ B\ Y \implies (\neg b \implies B \subseteq A) \implies A \cup B = C \implies$
 $table\ n\ C\ (join\ X\ b\ Y)$
unfolding *table_def* **by** (auto *elim!*: *join_wf_tuple*)

lemma *wf_tuple_Suc*: $wf_tuple (Suc\ m)\ A\ a \longleftrightarrow a \neq [] \wedge$
 $wf_tuple\ m\ ((\lambda x. x - 1)\ ' (A - \{0\}))\ (tl\ a) \wedge (0 \in A \longleftrightarrow hd\ a \neq None)$
by (*cases a*) (*auto simp: nth_Cons image_iff split: nat.splits*)

lemma *table_project*: $table (Suc\ n)\ A\ X \implies table\ n\ ((\lambda x. x - Suc\ 0)\ ' (A - \{0\}))\ (tl\ ' X)$
unfolding *table_def*
by (*auto simp: wf_tuple_Suc*)

definition *restrict where*

restrict A v = map (\lambda i. if i \in 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 \in A then x \# restrict ((\lambda x. x - 1)\ ' (A - \{0\})) xs else None \# restrict ((\lambda x. 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*: $wf_tuple\ n\ B\ v \implies A \cap B = C \implies wf_tuple\ n\ C\ (restrict\ A\ v)$
unfolding *restrict_def wf_tuple_def* **by** *auto*

lemma *wf_tuple_restrict_simple*: $wf_tuple\ n\ B\ v \implies A \subseteq B \implies wf_tuple\ n\ A\ (restrict\ A\ v)$
unfolding *restrict_def wf_tuple_def* **by** *auto*

lemma *nth_restrict*: $i \in A \implies i < length\ v \implies restrict\ A\ v ! i = v ! i$
unfolding *restrict_def* **by** *auto*

lemma *restrict_eq_Nil[simp]*: $restrict\ A\ v = [] \longleftrightarrow v = []$
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 $join1\ (x, y) = Some\ z \longleftrightarrow wf_tuple\ n\ (A \cup B)\ z \wedge restrict\ A\ z = x \wedge restrict\ B\ z = y$

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: $wf_tuple\ n\ A\ v \implies restrict\ A\ v = v$
by (induct v arbitrary: n A) (auto split: if_splits)

lemma map_the_restrict:

$i \in A \implies map\ the\ (restrict\ A\ v)\ !\ i = map\ the\ v\ !\ i$
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 $\bigwedge v. v \in X \implies wf_tuple\ n\ A\ v \wedge \bigwedge v. v \in Y \implies wf_tuple\ n\ B\ v \neg b \implies B \subseteq A$

shows $v \in join\ X\ b\ Y \longleftrightarrow$

$wf_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$

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 $\neg b \implies B \subseteq A$

shows $v \in join\ X\ b\ Y \longleftrightarrow$

$wf_tuple\ n\ (A \cup B)\ v \wedge restrict\ A\ v \in X \wedge (if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y)$

using assms **unfolding** table_def

by (simp add: join_restrict)

lemma join_restrict_annotated:

fixes X Y :: 'a tuple set

assumes $\neg b = simp \implies B \subseteq A$

shows $join\ \{v. wf_tuple\ n\ A\ v \wedge P\ v\}\ b\ \{v. wf_tuple\ n\ B\ v \wedge Q\ v\} =$

$\{v. wf_tuple\ n\ (A \cup B)\ v \wedge P\ (restrict\ A\ v) \wedge (if\ b\ then\ Q\ (restrict\ B\ v)\ else\ \neg\ Q\ (restrict\ B\ v))\}$

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 \implies table n B Y \implies ($\neg b \implies B \subseteq A$) \implies $wf_tuple\ n\ (A \cup B)\ v \implies$

$restrict\ A\ v \in X \implies (b \implies restrict\ B\ v \in Y) \implies (\neg b \implies restrict\ B\ v \notin Y) \implies v \in join\ X\ b\ Y$

unfolding table_def

by (subst join_restrict) (auto)

lemma in_joinE: $v \in join\ X\ b\ Y \implies$ table n A X \implies table n B Y \implies ($\neg b \implies B \subseteq A$) \implies

$(wf_tuple\ n\ (A \cup B)\ v \implies restrict\ A\ v \in X \implies if\ b\ then\ restrict\ B\ v \in Y\ else\ restrict\ B\ v \notin Y \implies$

$P) \implies P$

unfolding table_def

by (subst (asm) join_restrict) (auto)

definition qtable :: nat \Rightarrow nat set \Rightarrow ('a tuple \Rightarrow bool) \Rightarrow ('a tuple \Rightarrow bool) \Rightarrow

'a table \Rightarrow bool **where**

qtable n A P Q X \longleftrightarrow table n A X \wedge ($\forall x. (x \in X \wedge P\ x \longrightarrow Q\ x) \wedge (wf_tuple\ n\ A\ x \wedge P\ x \wedge Q\ x \longrightarrow x \in X)$)

abbreviation wf_table **where**

$wf_table\ n\ A\ Q\ X \equiv qtable\ n\ A\ (\lambda_.\ True)\ Q\ X$

lemma wf_table_iff: $wf_table\ n\ A\ Q\ X \longleftrightarrow (\forall x. x \in X \longleftrightarrow (Q\ x \wedge wf_tuple\ n\ A\ x))$

unfolding qtable_def table_def **by** auto

lemma table_wf_table: table n A X = $wf_table\ n\ A\ (\lambda v. v \in X)\ X$

unfolding table_def wf_table_iff **by** auto

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

lemma *in_qtableI*: $qtable\ n\ A\ P\ Q\ X \implies wf_tuple\ n\ A\ x \implies P\ x \implies Q\ x \implies x \in X$
unfolding *qtable_def* **by** *blast*

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

lemma *qtable_empty*: $(\bigwedge x. wf_tuple\ n\ A\ x \implies P\ x \implies Q\ x \implies False) \implies qtable\ n\ A\ P\ Q\ empty_table$
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*: $(\bigwedge x. wf_tuple\ n\ \{\} x \implies P\ x \implies Q\ x) \implies qtable\ n\ \{\} P\ Q\ (unit_table\ n)$
unfolding *qtable_def table_def in_unit_table* **by** *auto*

lemma *qtable_union*: $qtable\ n\ A\ P\ Q1\ X \implies qtable\ n\ A\ P\ Q2\ Y \implies$
 $(\bigwedge x. wf_tuple\ n\ A\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ x \vee Q2\ x) \implies qtable\ n\ A\ P\ Q\ (X \cup Y)$
unfolding *qtable_def table_def* **by** *blast*

lemma *qtable_Union*: $finite\ I \implies (\bigwedge i. i \in I \implies qtable\ n\ A\ P\ (Qi\ i)\ (Xi\ i)) \implies$
 $(\bigwedge x. wf_tuple\ n\ A\ x \implies P\ x \implies Q\ x \longleftrightarrow (\exists i \in I. Qi\ i\ x)) \implies qtable\ n\ A\ P\ Q\ (\bigcup i \in I. Xi\ i)$
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 $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y \neg b \implies B \subseteq A\ C = A \cup B$
 $\bigwedge x. wf_tuple\ n\ C\ x \implies P\ x \implies P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$
 $\bigwedge x. b \implies wf_tuple\ n\ C\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge Q2\ (restrict\ B\ x)$
 $\bigwedge x. \neg b \implies wf_tuple\ n\ C\ x \implies P\ x \implies Q\ x \longleftrightarrow Q1\ (restrict\ A\ x) \wedge \neg Q2\ (restrict\ B\ x)$
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 $qtable\ n\ A\ P\ Q1\ X\ qtable\ n\ B\ P\ Q2\ Y \neg b \implies B \subseteq A\ C = A \cup B$
 $\bigwedge x. wf_tuple\ n\ C\ x \implies P\ x \implies P\ (restrict\ A\ x) \wedge P\ (restrict\ B\ x)$
shows $qtable\ n\ C\ P\ (\lambda x. Q1\ (restrict\ A\ x) \wedge (if\ b\ then\ Q2\ (restrict\ B\ x)\ else\ \neg\ Q2\ (restrict\ B\ x)))$

(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 $\forall x \in A. \text{map the } v ! x = \text{map the } w ! x$
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 = ($\lambda x. x - 1$) ‘ (A - {0})

have *: map the xs ! z = map the ys ! z **if** z \in ?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 \Rightarrow 'a tuple \Rightarrow bool **where**

mem_restr A x \longleftrightarrow ($\exists y \in A. \text{list_all2 } (\lambda a b. a \neq \text{None} \longrightarrow a = \text{Some } b) x y$)

lemma mem_restrI: $y \in A \Longrightarrow \text{length } y = n \Longrightarrow \text{wf_tuple } n V x \Longrightarrow \forall i \in V. x ! i = \text{Some } (y ! i) \Longrightarrow$
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 $\Longrightarrow \text{wf_tuple } n V x \Longrightarrow \forall i \in V. i < n \Longrightarrow$

($\bigwedge y. y \in A \Longrightarrow \forall i \in V. x ! i = \text{Some } (y ! i) \Longrightarrow P$) $\Longrightarrow P$

unfolding mem_restr_def wf_tuple_def **by** (fastforce simp add: list_all2_conv_all_nth)

lemma mem_restr_IntD: mem_restr (A \cap B) v $\Longrightarrow \text{mem_restr } A v \wedge \text{mem_restr } B v$

unfolding mem_restr_def **by** auto

lemma mem_restr_Un_iff: mem_restr (A \cup B) x $\longleftrightarrow \text{mem_restr } A x \vee \text{mem_restr } B x$

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 $\Longrightarrow \text{mem_restr } A (\text{restrict } V x)$

unfolding mem_restr_def restrict_def

by (auto simp: list_all2_conv_all_nth elim!: bexI[rotated])

definition lift_envs :: 'a list set \Rightarrow 'a list set **where**

lift_envs R = ($\lambda(a,b). a \# b$) ‘ (UNIV \times R)

lemma lift_envs_mem_restr[simp]: mem_restr A x $\Longrightarrow \text{mem_restr } (\text{lift_envs } A) (a \# x)$

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 (($\lambda x. x - \text{Suc } 0$) ‘ (A - {0})) (mem_restr R)

($\lambda v. \exists x. P ((\text{if } 0 \in A \text{ then Some } x \text{ else None}) \# v)$) (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  $\implies$  A = B  $\implies$  ( $\bigwedge$ v. P v  $\implies$  Q v  $\longleftrightarrow$  Q' v)  $\implies$  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  $\implies$  x < nfv and
    sat_fv_cong: ( $\bigwedge$ x. x  $\in$  fv  $\implies$  v!x = v!x)  $\implies$  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  $\implies$   $\exists$   $\pi$ . 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:  $\pi \leq \pi' \Longrightarrow M \pi \subseteq M \pi'$  and
    wf_monitor:  $(i, v) \in M \pi \Longrightarrow wf\_tuple \ nfv \ fv \ v$  and
    sound_monitor:  $(i, v) \in M \pi \Longrightarrow prefix\_of \ \pi \ \sigma \Longrightarrow sat \ \sigma \ (map \ the \ v) \ i$  and
    complete_monitor:  $prefix\_of \ \pi \ \sigma \Longrightarrow wf\_tuple \ nfv \ fv \ v \Longrightarrow$ 
       $(\bigwedge \sigma. prefix\_of \ \pi \ \sigma \Longrightarrow sat \ \sigma \ (map \ the \ v) \ i) \Longrightarrow \exists \pi'. prefix\_of \ \pi' \ \sigma \wedge (i, v) \in M \pi'$ 

```

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  $\sigma = verdicts \ \sigma$ 

```

```

proof

```

```

  show  $\bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\} \subseteq verdicts \ \sigma$ 
  by (auto simp: verdicts_def wf_monitor sound_monitor)

```

```

next

```

```

  show  $\bigcup \{M \pi \mid \pi. prefix\_of \ \pi \ \sigma\} \supseteq verdicts \ \sigma$ 
  unfolding verdicts_def

```

```

proof safe

```

```

  fix i v

```

```

  assume wf_tuple nfv fv v and sat  $\sigma \ (map \ the \ v) \ i$ 

```

```

  then obtain  $\pi$  where prefix_of  $\pi \ \sigma \wedge (\forall \sigma'. prefix\_of \ \pi \ \sigma' \longrightarrow sat \ \sigma' \ (map \ the \ v) \ i)$ 

```

```

  using cosafety_lr by blast

```

```

  with  $\langle wf\_tuple \ nfv \ fv \ v \rangle$  obtain  $\pi'$  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:  $\pi \leq \pi' \Longrightarrow progress \ \pi \leq progress \ \pi'$  and

```

```

  ex_progress_ge:  $\exists \pi. prefix\_of \ \pi \ \sigma \wedge x \leq progress \ \pi$  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

```


$M \pi = \{(i, v). i < \text{progress } \pi \wedge \text{wf_tuple } \text{nfv } \text{fv } v \wedge$
 $(\forall \sigma. \text{prefix_of } \pi \sigma \longrightarrow \text{sat } \sigma (\text{map the } v) i)\}$

lemma M_alt : $M \pi = \{(i, v). i < \text{progress } \pi \wedge \text{wf_tuple } \text{nfv } \text{fv } v \wedge$
 $(\exists \sigma. \text{prefix_of } \pi \sigma \wedge \text{sat } \sigma (\text{map the } v) i)\}$
using ex_prefix_of [of π]
by ($auto simp$: $M_def cong$: $progress_sat_cong$)

end

sublocale $progress \subseteq \text{cosafety_monitor } _ _ _ M$

proof

fix $i v$ **and** $\sigma :: 'a \text{ trace}$
assume $\text{sat } \sigma v i$
moreover obtain π **where** $*$: $\text{prefix_of } \pi \sigma i < \text{progress } \pi$
using $ex_progress_ge$ **by** ($auto simp$: $less_eq_Suc_le$)
ultimately have $\text{sat } \sigma' v i$ **if** $\text{prefix_of } \pi \sigma'$ **for** σ'
using $that$ **by** ($simp cong$: $progress_sat_cong$)
with $*$ **show** $\exists \pi. \text{prefix_of } \pi \sigma \wedge (\forall \sigma'. \text{prefix_of } \pi \sigma' \longrightarrow \text{sat } \sigma' v i)$
by $blast$

next

fix $\pi \pi' :: 'a \text{ prefix}$
assume $\pi \leq \pi'$
then show $M \pi \subseteq M \pi'$
by ($auto simp$: $M_def intro$: $progress_mono prefix_of_antimono$
 $elim$: $order.strict_trans2$)

next

fix $i v \pi$ **and** $\sigma :: 'a \text{ trace}$
assume $*$: $(i, v) \in M \pi$
then show $\text{wf_tuple } \text{nfv } \text{fv } v$
by ($simp add$: M_def)
assume $\text{prefix_of } \pi \sigma$
with $*$ **show** $\text{sat } \sigma (\text{map the } v) i$
by ($simp add$: M_def)

next

fix $i v \pi$ **and** $\sigma :: 'a \text{ trace}$
assume $*$: $\text{prefix_of } \pi \sigma \text{wf_tuple } \text{nfv } \text{fv } v \wedge \sigma'. \text{prefix_of } \pi \sigma' \Longrightarrow \text{sat } \sigma' (\text{map the } v) i$
show $\exists \pi'. \text{prefix_of } \pi' \sigma \wedge (i, v) \in M \pi'$
proof ($cases i < \text{progress } \pi$)

case $True$

with $*$ **show** $?thesis$ **by** ($auto simp$: M_def)

next

case $False$

obtain π' **where** $**$: $\text{prefix_of } \pi' \sigma \wedge i < \text{progress } \pi'$
using $ex_progress_ge$ **by** ($auto simp$: $less_eq_Suc_le$)

then have $\pi \leq \pi'$

using $\langle \text{prefix_of } \pi \sigma \rangle \text{prefix_of_imp_linear } False \text{progress_mono}$
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 $relevant_events :: 'b \text{ list set} \Rightarrow 'a \text{ set}$

begin

abbreviation *slice* :: 'b list set \Rightarrow 'a trace \Rightarrow 'a trace **where**
slice *S* \equiv *map_Γ* ($\lambda D. D \cap$ *relevant_events* *S*)

abbreviation *pslice* :: 'b list set \Rightarrow 'a prefix \Rightarrow 'a prefix **where**
pslice *S* \equiv *pmap_Γ* ($\lambda D. D \cap$ *relevant_events* *S*)

lemma *prefix_of_psliceI*: *prefix_of* π $\sigma \implies$ *prefix_of* (*pslice* *S* π) (*slice* *S* σ)
by (*transfer fixing*: *S*) *auto*

lemma *plen_pslice[simp]*: *plen* (*pslice* *S* π) = *plen* π
by (*transfer fixing*: *S*) *simp*

lemma *pslice_pnil[simp]*: *pslice* *S* *pnil* = *pnil*
by (*transfer fixing*: *S*) *simp*

lemma *last_ts_pslice[simp]*: *last_ts* (*pslice* *S* π) = *last_ts* π
by (*transfer fixing*: *S*) (*simp add*: *last_map case_prod_beta split*: *list.split*)

abbreviation *verdict_filter* :: 'b list set \Rightarrow (*nat* \times 'b tuple) set \Rightarrow (*nat* \times 'b tuple) set **where**
verdict_filter *S* *V* \equiv $\{(i, v) \in V. \text{mem_restr } S \ v\}$

end

locale *sliceable_fo_spec* = *fo_spec* __ *sat* + *abstract_slicer* *relevant_events*
for *relevant_events* :: 'b list set \Rightarrow 'a set **and** *sat* :: 'a trace \Rightarrow 'b list \Rightarrow *nat* \Rightarrow *bool* +
assumes *sliceable*: $v \in S \implies \text{sat} (\text{slice } S \ \sigma) \ v \ i \longleftrightarrow \text{sat } \sigma \ v \ i$
begin

lemma *union_verdicts_slice*:

assumes *part*: $\bigcup S = UNIV$

shows $\bigcup ((\lambda S. \text{verdict_filter } S (\text{verdicts } (\text{slice } S \ \sigma))) \ 'S) = \text{verdicts } \sigma$

proof *safe*

fix *S* *i* **and** *v* :: 'b tuple

assume $(i, v) \in \text{verdicts } (\text{slice } S \ \sigma)$

then have *tuple*: *wf_tuple* *nfv* *fv* *v* **and** *sat* (*slice* *S* σ) (*map the* *v*) *i*

by (*auto simp*: *verdicts_def*)

assume *mem_restr* *S* *v*

then obtain *v'* **where** $v' \in S$ **and** *1*: $\forall i \in \text{fv}. v \ ! \ i = \text{Some } (v' \ ! \ i)$

using *tuple* **by** (*auto simp*: *fv_less_nfv elim*: *mem_restrE*)

then have *sat* (*slice* *S* σ) *v'* *i*

using $\langle \text{sat } (\text{slice } S \ \sigma) (\text{map the } v) \ i \rangle$ *tuple*

by (*auto simp*: *wf_tuple_length fv_less_nfv cong*: *sat_fv_cong*)

then have *sat* σ *v'* *i*

using *sliceable*[*OF* $\langle v' \in S \rangle$] **by** *simp*

then have *sat* σ (*map the* *v*) *i*

using *tuple* *1*

by (*auto simp*: *wf_tuple_length fv_less_nfv cong*: *sat_fv_cong*)

then show $(i, v) \in \text{verdicts } \sigma$

using *tuple* **by** (*simp add*: *verdicts_def*)

next

fix *i* **and** *v* :: 'b tuple

assume $(i, v) \in \text{verdicts } \sigma$

then have *tuple*: *wf_tuple* *nfv* *fv* *v* **and** *sat* σ (*map the* *v*) *i*

by (*auto simp*: *verdicts_def*)

from *part* **obtain** *S* **where** $S \in S$ **and** *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  $\sigma$ ) (map the v) i
  using  $\langle \text{sat } \sigma \text{ (map the v) } i \rangle$  sliceable[OF  $\langle \text{map the v } \in S \rangle$ ] by simp
then have  $(i, v) \in \text{verdicts } (\text{slice } S \ \sigma)$ 
  using tuple by (simp add: verdicts_def)
ultimately show  $(i, v) \in (\bigcup S \in \mathcal{S}. \text{verdict\_filter } S \ (\text{verdicts } (\text{slice } S \ \sigma)))$ 
  using  $\langle S \in \mathcal{S} \rangle$  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  $\_ \_ \text{sat } M + \text{abstract\_slicer } \text{relevant\_events}$ 
  for relevant_events :: 'b list set  $\Rightarrow$  'a set and sat :: 'a trace  $\Rightarrow$  'b list  $\Rightarrow$  nat  $\Rightarrow$  bool and M +
  assumes sliceable_M: mem_restr S v  $\Longrightarrow$   $(i, v) \in M \ (\text{pslice } S \ \pi) \longleftrightarrow (i, v) \in M \ \pi$ 
begin

```

```

lemma union_M_pslice:
  assumes part:  $\bigcup \mathcal{S} = \text{UNIV}$ 
  shows  $\bigcup ((\lambda S. \text{verdict\_filter } S \ (M \ (\text{pslice } S \ \pi))) \ \mathcal{S}) = M \ \pi$ 
proof safe

```

```

  fix S i and v :: 'b tuple
  assume mem_restr S v and  $(i, v) \in M \ (\text{pslice } S \ \pi)$ 
  then show  $(i, v) \in M \ \pi$  using sliceable_M by blast
next
  fix i and v :: 'b tuple
  assume  $(i, v) \in M \ \pi$ 
  then have tuple: wf_tuple nfv fv v
    by (rule wf_monitor)
  from part obtain S where  $S \in \mathcal{S}$  and 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)
  then have  $(i, v) \in M \ (\text{pslice } S \ \pi)$ 
    using  $\langle (i, v) \in M \ \pi \rangle$  sliceable_M by blast
  then show  $(i, v) \in (\bigcup S \in \mathcal{S}. \text{verdict\_filter } S \ (M \ (\text{pslice } S \ \pi)))$ 
    using  $\langle S \in \mathcal{S} \rangle$   $\langle \text{mem\_restr } S \ v \rangle$  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  $\pi = \text{pts } \pi' \Longrightarrow \text{progress } \pi = \text{progress } \pi'$ 

```

```

locale sliceable_timed_progress = sliceable_fo_spec + timed_progress
begin

```

```

lemma progress_pslice[simp]: progress (pslice S  $\pi$ ) = progress  $\pi$ 
  by (simp cong: progress_time_conv)

```

end

```

sublocale sliceable_timed_progress  $\subseteq$  sliceable_monitor _ _ _ _ M
proof
  fix S :: 'a list set and v i  $\pi$ 
  assume *: mem_restr S v
  show (i, v)  $\in$  M (pslice S  $\pi$ )  $\longleftrightarrow$  (i, v)  $\in$  M  $\pi$  (is ?L  $\longleftrightarrow$  ?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  $\mathcal{I} = \{(i :: \text{nat}, j :: \text{enat}). i \leq j\}$ 
  by (intro exI[of _ (0,  $\infty$ )] auto)

```

```

setup_lifting type_definition  $\mathcal{I}$ 

```

```

instantiation  $\mathcal{I} :: \text{equal}$  begin
lift_definition equal_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow \text{bool}$  is (=) .
instance by standard (transfer, auto)
end

```

```

instantiation  $\mathcal{I} :: \text{linorder}$  begin
lift_definition less_eq_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow \text{bool}$  is ( $\leq$ ) .
lift_definition less_ $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow \text{bool}$  is ( $<$ ) .
instance by standard (transfer, auto)+
end

```

```

lift_definition all ::  $\mathcal{I}$  is (0,  $\infty$ ) by simp
lift_definition left ::  $\mathcal{I} \Rightarrow \text{nat}$  is fst .
lift_definition right ::  $\mathcal{I} \Rightarrow \text{enat}$  is snd .
lift_definition point ::  $\text{nat} \Rightarrow \mathcal{I}$  is  $\lambda n. (n, \text{enat } n)$  by simp
lift_definition init ::  $\text{nat} \Rightarrow \mathcal{I}$  is  $\lambda n. (0, \text{enat } n)$  by auto
abbreviation mem where mem n I  $\equiv$  (left I  $\leq$  n  $\wedge$  n  $\leq$  right I)
lift_definition subtract ::  $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is
   $\lambda n (i, j). (i - n, j - \text{enat } n)$  by (auto simp: diff_enat_def split: enat.splits)
lift_definition add ::  $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is
   $\lambda n (a, b). (a, b + \text{enat } n)$  by (auto simp add: add_increasing2)

```

```

lemma left_right: left I  $\leq$  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* = ($\lambda n.$ *subtract* *n* *I*) ‘ {0 .. (case *right* *I* of $\infty \Rightarrow$ *left* *I* | *enat* *n* \Rightarrow *n*)}

lemma *subtract_too_much*: $i > (\text{case } \text{right } I \text{ of } \infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n) \implies$

subtract *i* *I* = *subtract* (case *right* *I* of $\infty \Rightarrow$ *left* *I* | *enat* *n* \Rightarrow *n*) *I*
by *transfer* (*auto split*: *enat.splits*)

lemma *subtract_shifted*: *subtract* *n* *I* \in *shifted* *I*

by (cases $n \leq (\text{case } \text{right } I \text{ of } \infty \Rightarrow \text{left } I \mid \text{enat } n \Rightarrow n)$)
(*auto simp*: *shifted_def subtract_too_much*)

lemma *finite_shifted*: *finite* (*shifted* *I*)

unfolding *shifted_def* **by** *auto*

definition *interval* :: $\text{nat} \Rightarrow \text{enat} \Rightarrow \mathcal{I}$ **where**

interval *l* *r* = (if $l \leq r$ then *Abs* \mathcal{I} (*l*, *r*) else *undefined*)

lemma [*code abstract*]: *Rep* \mathcal{I} (*interval* *l* *r*) = (if $l \leq r$ 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* \times *'a list*)

qualified type_synonym *'a database* = *'a event set*

qualified type_synonym *'a prefix* = (*name* \times *'a list*) *prefix*

qualified type_synonym *'a trace* = (*name* \times *'a list*) *trace*

qualified type_synonym *'a env* = *'a list*

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

qualified primrec *fvi_trm* :: $\text{nat} \Rightarrow 'a \text{ trm} \Rightarrow \text{nat set}$ **where**

fvi_trm *b* (*Var* *x*) = (if $b \leq x$ then {*x* - *b*} else {})

| *fvi_trm* *b* (*Const* _) = {}

abbreviation *fv_trm* \equiv *fvi_trm* 0

qualified primrec *eval_trm* :: *'a env* \Rightarrow *'a trm* \Rightarrow *'a* **where**

eval_trm *v* (*Var* *x*) = *v* ! *x*

| $eval_trm\ v\ (Const\ x) = x$

lemma $eval_trm_cong$: $\forall x \in fv_trm\ t.\ v!\ x = v'!\ x \implies eval_trm\ v\ t = eval_trm\ v'\ t$
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**

$fvi\ b\ (Pred\ r\ ts) = (\bigcup t \in set\ ts.\ fvi_trm\ b\ t)$
 $fvi\ b\ (Eq\ t1\ t2) = fvi_trm\ b\ t1 \cup fvi_trm\ b\ t2$
 $fvi\ b\ (Neg\ \varphi) = fvi\ b\ \varphi$
 $fvi\ b\ (Or\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
 $fvi\ b\ (Exists\ \varphi) = fvi\ (Suc\ b)\ \varphi$
 $fvi\ b\ (Prev\ I\ \varphi) = fvi\ b\ \varphi$
 $fvi\ b\ (Next\ I\ \varphi) = fvi\ b\ \varphi$
 $fvi\ b\ (Since\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
 $fvi\ b\ (Until\ \varphi\ I\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$

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*: $\forall i \in \text{fv } \varphi. i < \text{nfv } \varphi$
unfolding *nfv_def*
by (*auto simp add: Max_gr_iff intro: max.strict_coboundedI2*)

qualified primrec *future_reach* :: 'a formula \Rightarrow enat where

future_reach (Pred $_ _$) = 0
| *future_reach* (Eq $_ _$) = 0
| *future_reach* (Neg φ) = *future_reach* φ
| *future_reach* (Or $\varphi \psi$) = max (*future_reach* φ) (*future_reach* ψ)
| *future_reach* (Exists φ) = *future_reach* φ
| *future_reach* (Prev $I \varphi$) = *future_reach* φ - left I
| *future_reach* (Next $I \varphi$) = *future_reach* φ + right I + 1
| *future_reach* (Since $\varphi I \psi$) = max (*future_reach* φ) (*future_reach* ψ - left I)
| *future_reach* (Until $\varphi I \psi$) = max (*future_reach* φ) (*future_reach* ψ) + right I + 1

qualified primrec *sat* :: 'a trace \Rightarrow 'a env \Rightarrow nat \Rightarrow 'a formula \Rightarrow bool where

sat $\sigma v i$ (Pred $r ts$) = ((r , map (*eval_trm* v) ts) $\in \Gamma \sigma i$)
| *sat* $\sigma v i$ (Eq $t1 t2$) = (*eval_trm* $v t1$ = *eval_trm* $v t2$)
| *sat* $\sigma v i$ (Neg φ) = (\neg *sat* $\sigma v i \varphi$)
| *sat* $\sigma v i$ (Or $\varphi \psi$) = (*sat* $\sigma v i \varphi \vee$ *sat* $\sigma v i \psi$)
| *sat* $\sigma v i$ (Exists φ) = ($\exists z. \text{sat } \sigma (z \# v) i \varphi$)
| *sat* $\sigma v i$ (Prev $I \varphi$) = (case i of 0 \Rightarrow False | Suc $j \Rightarrow$ mem ($\tau \sigma i - \tau \sigma j$) $I \wedge$ *sat* $\sigma v j \varphi$)
| *sat* $\sigma v i$ (Next $I \varphi$) = (mem ($\tau \sigma$ (Suc i) - $\tau \sigma i$) $I \wedge$ *sat* σv (Suc i) φ)
| *sat* $\sigma v i$ (Since $\varphi I \psi$) = ($\exists j \leq i. \text{mem } (\tau \sigma i - \tau \sigma j) I \wedge$ *sat* $\sigma v j \psi \wedge$ ($\forall k \in \{j <.. i\}. \text{sat } \sigma v k \varphi$))
| *sat* $\sigma v i$ (Until $\varphi I \psi$) = ($\exists j \geq i. \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge$ *sat* $\sigma v j \psi \wedge$ ($\forall k \in \{i ..< j\}. \text{sat } \sigma v k \varphi$))

lemma *sat_Until_rec*: *sat* $\sigma v i$ (Until $\varphi I \psi$) \longleftrightarrow

mem 0 $I \wedge$ *sat* $\sigma v i \psi \vee$
 $(\Delta \sigma (i + 1) \leq$ right $I \wedge$ *sat* $\sigma v i \varphi \wedge$ *sat* $\sigma v (i + 1)$ (Until φ (subtract ($\Delta \sigma (i + 1)$) I) ψ))
(is ?L \longleftrightarrow ?R)

proof (*rule iffI; (elim disjE conjE)?*)

assume ?L

then obtain j **where** $j: i \leq j$ mem ($\tau \sigma j - \tau \sigma i$) I *sat* $\sigma v j \psi \forall k \in \{i ..< j\}. \text{sat } \sigma v k \varphi$

by *auto*

then show ?R

proof (*cases* $i = j$)

case False

with $j(1,2)$ **have** $\Delta \sigma (i + 1) \leq$ right I

by (*auto elim: order_trans[rotated] simp: diff_le_mono*)

moreover from False $j(1,4)$ **have** *sat* $\sigma v i \varphi$ **by** *auto*

moreover from False j **have** *sat* $\sigma v (i + 1)$ (Until φ (subtract ($\Delta \sigma (i + 1)$) 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 $\Delta: \Delta \sigma (i + 1) \leq$ right I **and now:** *sat* $\sigma v i \varphi$ **and**

next: *sat* $\sigma v (i + 1)$ (Until φ (subtract ($\Delta \sigma (i + 1)$) I) ψ)

from *next* **obtain** j **where** $j: i + 1 \leq j$ mem ($\tau \sigma j - \tau \sigma (i + 1)$) ((subtract ($\Delta \sigma (i + 1)$) I))

sat $\sigma v j \psi \forall k \in \{i + 1 ..< j\}. \text{sat } \sigma v k \varphi$

by *auto*

from $\Delta j(1,2)$ **have** mem ($\tau \sigma j - \tau \sigma i$) 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*: $\text{sat } \sigma v i (\text{Since } \varphi I \psi) \longleftrightarrow$
 $\text{mem } 0 I \wedge \text{sat } \sigma v i \psi \vee$
 $(i > 0 \wedge \Delta \sigma i \leq \text{right } I \wedge \text{sat } \sigma v i \varphi \wedge \text{sat } \sigma v (i - 1) (\text{Since } \varphi (\text{subtract } (\Delta \sigma i) I) \psi))$
 $(\text{is } ?L \longleftrightarrow ?R)$

proof (*rule iffI*; (*elim disjE conjE*)?)
assume *?L*
then obtain *j* **where** $j: j \leq i \text{ mem } (\tau \sigma i - \tau \sigma j) I \text{ sat } \sigma v j \psi \forall k \in \{j <.. i\}. \text{sat } \sigma v k \varphi$
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** $\Delta \sigma i \leq \text{right } I$
by (*auto elim: order_trans[rotated] simp: diff_le_mono2 le_Suc_eq*)
moreover from *False j(1,4)* **have** $\text{sat } \sigma v i \varphi$ **by** *auto*
moreover from *False j* **have** $\text{sat } \sigma v (i - 1) (\text{Since } \varphi (\text{subtract } (\Delta \sigma i) I) \psi)$
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** $\Delta: \Delta \sigma i \leq \text{right } I$ **and now:** $\text{sat } \sigma v i \varphi$ **and**
 $\text{prev: sat } \sigma v (i - 1) (\text{Since } \varphi (\text{subtract } (\Delta \sigma i) I) \psi)$
from *prev* **obtain** *j* **where** $j: j \leq i - 1 \text{ mem } (\tau \sigma (i - 1) - \tau \sigma j) ((\text{subtract } (\Delta \sigma i) I))$
 $\text{sat } \sigma v j \psi \forall k \in \{j <.. i - 1\}. \text{sat } \sigma v k \varphi$
by *auto*
from $\Delta i j(1,2)$ **have** $\text{mem } (\tau \sigma i - \tau \sigma j) I$
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*: $\text{sat } \sigma v 0 (\text{Since } \varphi I \psi) \longleftrightarrow \text{mem } 0 I \wedge \text{sat } \sigma v 0 \psi$
by *auto*

lemma *sat_Since_point*: $\text{sat } \sigma v i (\text{Since } \varphi I \psi) \implies$
 $(\bigwedge j. j \leq i \implies \text{mem } (\tau \sigma i - \tau \sigma j) I \implies \text{sat } \sigma v i (\text{Since } \varphi (\text{point } (\tau \sigma i - \tau \sigma j)) \psi) \implies P) \implies P$
by (*auto intro: diff_le_self*)

lemma *sat_Since_pointD*: $\text{sat } \sigma v i (\text{Since } \varphi (\text{point } t) \psi) \implies \text{mem } t I \implies \text{sat } \sigma v i (\text{Since } \varphi I \psi)$
by *auto*

lemma *eval_trm_fvi_cong*: $\forall x \in \text{fv_trm } t. v!x = v'!x \implies \text{eval_trm } v t = \text{eval_trm } v' t$
by (*cases t*) *simp_all*

lemma *sat_fvi_cong*: $\forall x \in \text{fv } \varphi. v!x = v'!x \implies \text{sat } \sigma v i \varphi = \text{sat } \sigma v' i \varphi$
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 $And\ \varphi\ \psi = Neg\ (Or\ (Neg\ \varphi)\ (Neg\ \psi))$

lemma fvi_And : $fvi\ b\ (And\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
unfolding And_def **by** $simp$

lemma $nfv_And[simp]$: $nfv\ (And\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$
unfolding nfv_def **by** ($simp\ add$: $fvi_And\ image_Un\ Max_Un[symmetric]$)

lemma $future_reach_And$: $future_reach\ (And\ \varphi\ \psi) = max\ (future_reach\ \varphi)\ (future_reach\ \psi)$
unfolding And_def **by** $simp$

lemma sat_And : $sat\ \sigma\ v\ i\ (And\ \varphi\ \psi) = (sat\ \sigma\ v\ i\ \varphi \wedge sat\ \sigma\ v\ i\ \psi)$
unfolding And_def **by** $simp$

qualified definition $And_Not\ \varphi\ \psi = Neg\ (Or\ (Neg\ \varphi)\ \psi)$

lemma fvi_And_Not : $fvi\ b\ (And_Not\ \varphi\ \psi) = fvi\ b\ \varphi \cup fvi\ b\ \psi$
unfolding And_Not_def **by** $simp$

lemma $nfv_And_Not[simp]$: $nfv\ (And_Not\ \varphi\ \psi) = max\ (nfv\ \varphi)\ (nfv\ \psi)$
unfolding nfv_def **by** ($simp\ add$: $fvi_And_Not\ image_Un\ Max_Un[symmetric]$)

lemma $future_reach_And_Not$: $future_reach\ (And_Not\ \varphi\ \psi) = max\ (future_reach\ \varphi)\ (future_reach\ \psi)$
unfolding And_Not_def **by** $simp$

lemma sat_And_Not : $sat\ \sigma\ v\ i\ (And_Not\ \varphi\ \psi) = (sat\ \sigma\ v\ i\ \varphi \wedge \neg sat\ \sigma\ v\ i\ \psi)$
unfolding And_Not_def **by** $simp$

5.3 Safe formulas

fun $safe_formula :: 'a\ MFOTL.formula \Rightarrow bool$ **where**

$safe_formula\ (MFOTL.Eq\ t1\ t2) = (MFOTL.is_Const\ t1 \vee MFOTL.is_Const\ t2)$
 $| safe_formula\ (MFOTL.Neg\ (MFOTL.Eq\ (MFOTL.Const\ x)\ (MFOTL.Const\ y))) = True$
 $| safe_formula\ (MFOTL.Neg\ (MFOTL.Eq\ (MFOTL.Var\ x)\ (MFOTL.Var\ y))) = (x = y)$
 $| safe_formula\ (MFOTL.Pred\ e\ ts) = True$
 $| safe_formula\ (MFOTL.Neg\ (MFOTL.Or\ (MFOTL.Neg\ \varphi)\ \psi)) = (safe_formula\ \varphi \wedge$
 $(safe_formula\ \psi \wedge MFOTL.fv\ \psi \subseteq MFOTL.fv\ \varphi \vee (case\ \psi\ of\ MFOTL.Neg\ \psi' \Rightarrow safe_formula\ \psi' |$
 $_ \Rightarrow False)))$
 $| safe_formula\ (MFOTL.Or\ \varphi\ \psi) = (MFOTL.fv\ \psi = MFOTL.fv\ \varphi \wedge safe_formula\ \varphi \wedge safe_formula$
 $\psi)$
 $| safe_formula\ (MFOTL.Exists\ \varphi) = (safe_formula\ \varphi)$
 $| safe_formula\ (MFOTL.Prev\ I\ \varphi) = (safe_formula\ \varphi)$
 $| safe_formula\ (MFOTL.Next\ I\ \varphi) = (safe_formula\ \varphi)$
 $| safe_formula\ (MFOTL.Since\ \varphi\ I\ \psi) = (MFOTL.fv\ \varphi \subseteq MFOTL.fv\ \psi \wedge$
 $(safe_formula\ \varphi \vee (case\ \varphi\ of\ MFOTL.Neg\ \varphi' \Rightarrow safe_formula\ \varphi' | _ \Rightarrow False)) \wedge safe_formula\ \psi)$
 $| safe_formula\ (MFOTL.Until\ \varphi\ I\ \psi) = (MFOTL.fv\ \varphi \subseteq MFOTL.fv\ \psi \wedge$
 $(safe_formula\ \varphi \vee (case\ \varphi\ of\ MFOTL.Neg\ \varphi' \Rightarrow safe_formula\ \varphi' | _ \Rightarrow False)) \wedge safe_formula\ \psi)$
 $| safe_formula\ _ = False$

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 $safe_formula\ \varphi$
and $\bigwedge t1\ t2. MFOTL.is_Const\ t1 \Longrightarrow P\ (MFOTL.Eq\ t1\ t2)$
and $\bigwedge t1\ t2. MFOTL.is_Const\ t2 \Longrightarrow P\ (MFOTL.Eq\ t1\ t2)$

```

and  $\bigwedge x y. P (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))$ 
and  $\bigwedge x y. x = y \implies P (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y)))$ 
and  $\bigwedge e ts. P (MFOTL.Pred e ts)$ 
and  $\bigwedge \varphi \psi. \neg (safe\_formula (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \implies P \varphi \implies P \psi$ 
 $\implies P (MFOTL.And \varphi \psi)$ 
and  $\bigwedge \varphi \psi. safe\_formula \psi \implies MFOTL.fv \psi \subseteq MFOTL.fv \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.And\_Not$ 
 $\varphi \psi)$ 
and  $\bigwedge \varphi \psi. MFOTL.fv \psi = MFOTL.fv \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Or \varphi \psi)$ 
and  $\bigwedge \varphi. P \varphi \implies P (MFOTL.Exists \varphi)$ 
and  $\bigwedge I \varphi. P \varphi \implies P (MFOTL.Prev I \varphi)$ 
and  $\bigwedge I \varphi. P \varphi \implies P (MFOTL.Next I \varphi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \implies safe\_formula \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Since$ 
 $\varphi I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \implies$ 
 $\neg safe\_formula (MFOTL.Neg \varphi) \implies P \varphi \implies P \psi \implies P (MFOTL.Since (MFOTL.Neg \varphi) I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \implies safe\_formula \varphi \implies P \varphi \implies P \psi \implies P (MFOTL.Until$ 
 $\varphi I \psi)$ 
and  $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \implies$ 
 $\neg safe\_formula (MFOTL.Neg \varphi) \implies P \varphi \implies P \psi \implies P (MFOTL.Until (MFOTL.Neg \varphi) I \psi)$ 
shows  $P \varphi$ 
using  $assms(1)$ 
proof (induction rule: safe_formula.induct)
case ( $5 \varphi \psi$ )
then show  $?case$ 
by ( $cases \psi$ )
 $(auto 0 3 elim!: disjE\_Not2 intro: assms[unfolded MFOTL.And\_def MFOTL.And\_Not\_def])$ 
next
case ( $10 \varphi I \psi$ )
then show  $?case$ 
by ( $cases \varphi$ ) ( $auto 0 3 elim!: disjE\_Not2 intro: assms$ )
next
case ( $11 \varphi I \psi$ )
then show  $?case$ 
by ( $cases \varphi$ ) ( $auto 0 3 elim!: disjE\_Not2 intro: assms$ )
qed ( $auto intro: assms$ )

```

5.4 Slicing traces

qualified primrec $matches :: 'a env \Rightarrow 'a formula \Rightarrow name \times 'a list \Rightarrow bool$ **where**

```

 $matches v (Pred r ts) e = (r = fst e \wedge map (eval\_trm v) ts = snd e)$ 
 $| matches v (Eq \_ \_) e = False$ 
 $| matches v (Neg \varphi) e = matches v \varphi e$ 
 $| matches v (Or \varphi \psi) e = (matches v \varphi e \vee matches v \psi e)$ 
 $| matches v (Exists \varphi) e = (\exists z. matches (z \# v) \varphi e)$ 
 $| matches v (Prev I \varphi) e = matches v \varphi e$ 
 $| matches v (Next I \varphi) e = matches v \varphi e$ 
 $| matches v (Since \varphi I \psi) e = (matches v \varphi e \vee matches v \psi e)$ 
 $| matches v (Until \varphi I \psi) e = (matches v \varphi e \vee matches v \psi e)$ 

```

lemma $matches_fvi_cong: \forall x \in fv \varphi. v!x = v'!x \implies matches v \varphi e = matches v' \varphi e$

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 ( $Exists \varphi$ )
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* $\varphi S \equiv \{e. S \cap \{v. \text{matches } v \varphi e\} \neq \{\}\}$

lemma *sat_slice_strong*: *relevant_events* $\varphi S \subseteq E \implies v \in S \implies$

$\text{sat } \sigma v i \varphi \longleftrightarrow \text{sat } (\text{map_}\Gamma (\lambda D. D \cap E) \sigma) v i \varphi$

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* $\varphi \psi$)

show *?case* **using** *Or.IH[of S] Or.prem*s

by (*auto simp: Collect_disj_eq Int_Un_distrib subset_iff*)

next

case (*Exists* φ)

have $\text{sat } \sigma (z \# v) i \varphi = \text{sat } (\text{map_}\Gamma (\lambda D. D \cap E) \sigma) (z \# v) i \varphi$ **for** z

using *Exists.prem*s **by** (*auto intro!: Exists.IH[of {z # v | v. v \in S}]*)

then show *?case* **by** *simp*

next

case (*Prev* $I \varphi$)

then show *?case* **by** (*auto cong: nat.case_cong*)

next

case (*Next* $I \varphi$)

then show *?case* **by** *simp*

next

case (*Since* $\varphi I \psi$)

show *?case* **using** *Since.IH[of S] Since.prem*s

by (*auto simp: Collect_disj_eq Int_Un_distrib subset_iff*)

next

case (*Until* $\varphi I \psi$)

show *?case* **using** *Until.IH[of S] Until.prem*s

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 $\text{MFOTL.sat } \sigma v i \varphi \longleftrightarrow \text{MFOTL.sat } (\text{MFOTL_slicer.slice } \varphi S \sigma) v i \varphi$

by (*rule sat_slice_strong[of S, OF subset_refl assms]*)

lemma *slice_replace_prefix*:

prefix_of (*MFOTL_slicer.pslice* $\varphi R \pi$) $\sigma \implies$

$\text{MFOTL_slicer.slice } \varphi R (\text{replace_prefix } \pi \sigma) = \text{MFOTL_slicer.slice } \varphi R \sigma$

by (*rule map_Γ_replace_prefix*) *auto*

6 Monitor implementation

6.1 Monitorable formulas

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

```

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  $\varphi$ )  $\psi$ )) = (mmonitorable_exec  $\varphi$   $\wedge$ 
  (mmonitorable_exec  $\psi$   $\wedge$  MFOTL.fv  $\psi$   $\subseteq$  MFOTL.fv  $\varphi$   $\vee$  (case  $\psi$  of MFOTL.Neg  $\psi'$   $\Rightarrow$  mmoni-
  torable_exec  $\psi'$  | _  $\Rightarrow$  False)))
| mmonitorable_exec (MFOTL.Or  $\varphi$   $\psi$ ) = (MFOTL.fv  $\psi$  = MFOTL.fv  $\varphi$   $\wedge$  mmonitorable_exec  $\varphi$   $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec (MFOTL.Exists  $\varphi$ ) = (mmonitorable_exec  $\varphi$ )
| mmonitorable_exec (MFOTL.Prev I  $\varphi$ ) = (mmonitorable_exec  $\varphi$ )
| mmonitorable_exec (MFOTL.Next I  $\varphi$ ) = (mmonitorable_exec  $\varphi$   $\wedge$  right I  $\neq \infty$ )
| mmonitorable_exec (MFOTL.Since  $\varphi$  I  $\psi$ ) = (MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$   $\wedge$ 
  (mmonitorable_exec  $\varphi$   $\vee$  (case  $\varphi$  of MFOTL.Neg  $\varphi'$   $\Rightarrow$  mmonitorable_exec  $\varphi'$  | _  $\Rightarrow$  False))  $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec (MFOTL.Until  $\varphi$  I  $\psi$ ) = (MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$   $\wedge$  right I  $\neq \infty$   $\wedge$ 
  (mmonitorable_exec  $\varphi$   $\vee$  (case  $\varphi$  of MFOTL.Neg  $\varphi'$   $\Rightarrow$  mmonitorable_exec  $\varphi'$  | _  $\Rightarrow$  False))  $\wedge$ 
  mmonitorable_exec  $\psi$ )
| mmonitorable_exec _ = False

```

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

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

lemma safe_formula_mmonitorable_exec: $safe_formula\ \varphi \implies MFOTL.future_reach\ \varphi \neq \infty \implies mmoni-$
 $torable_exec\ \varphi$

proof (induct φ rule: safe_formula.induct)

case (5 $\varphi\ \psi$)

then show ?case

unfolding safe_formula.simps future_reach.simps mmonitorable_exec.simps

by (fastforce split: formula.splits)

next

case (6 $\varphi\ \psi$)

then show ?case

unfolding safe_formula.simps future_reach.simps mmonitorable_exec.simps

by (fastforce split: formula.splits)

next

case (10 $\varphi\ I\ \psi$)

then show ?case

unfolding safe_formula.simps future_reach.simps mmonitorable_exec.simps

by (fastforce split: formula.splits)

next

case (11 $\varphi\ I\ \psi$)

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: $mmonitorable_exec\ \varphi \implies mmonitorable\ \varphi$

proof (induct φ rule: mmonitorable_exec.induct)

```

case (5  $\varphi$   $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce split: formula.splits)
next
case (10  $\varphi$   $I$   $\psi$ )
then show ?case
  unfolding mmonitorable_def mmonitorable_exec.simps safe_formula.simps
  by (fastforce split: formula.splits)
next
case (11  $\varphi$   $I$   $\psi$ )
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  $\varphi$  = mmonitorable_exec  $\varphi$ 
  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  $\times$  'a table list
type_synonym 'a msaux = (ts  $\times$  'a table) list
type_synonym 'a mauaux = (ts  $\times$  'a table  $\times$  '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  $\mathcal{I}$  'a mformula bool 'a table list ts list
| MNext  $\mathcal{I}$  'a mformula bool ts list
| MSince bool 'a mformula  $\mathcal{I}$  'a mformula 'a mbuf2 ts list 'a msaux
| MUntil bool 'a mformula  $\mathcal{I}$  'a mformula 'a mbuf2 ts list 'a mauaux

```

```

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

```

```

fun eq_rel :: nat  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  '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  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  'a MFOTL.trm  $\Rightarrow$  '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  $\Rightarrow$  'a MFOTL.formula  $\Rightarrow$  'a mformula where
  minit0 n (MFOTL.Neg  $\varphi$ ) = (case  $\varphi$  of
    MFOTL.Eq t1 t2  $\Rightarrow$  MRel (neq_rel n t1 t2)

```

```

| MFOTL.Or (MFOTL.Neg  $\varphi$ )  $\psi$   $\Rightarrow$  (if safe_formula  $\psi \wedge$  MFOTL.fv  $\psi \subseteq$  MFOTL.fv  $\varphi$ 
  then MAnd (minit0 n  $\varphi$ ) False (minit0 n  $\psi$ ) ([], [])
  else (case  $\psi$  of MFOTL.Neg  $\psi \Rightarrow$  MAnd (minit0 n  $\varphi$ ) True (minit0 n  $\psi$ ) ([], []) | _  $\Rightarrow$  undefined))
| _  $\Rightarrow$  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  $\varphi$   $\psi$ ) = MOr (minit0 n  $\varphi$ ) (minit0 n  $\psi$ ) ([], [])
| minit0 n (MFOTL.Exists  $\varphi$ ) = MExists (minit0 (Suc n)  $\varphi$ )
| minit0 n (MFOTL.Prev I  $\varphi$ ) = MPrev I (minit0 n  $\varphi$ ) True [] []
| minit0 n (MFOTL.Next I  $\varphi$ ) = MNext I (minit0 n  $\varphi$ ) True []
| minit0 n (MFOTL.Since  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MSince True (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
  else (case  $\varphi$  of
    MFOTL.Neg  $\varphi \Rightarrow$  MSince False (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
    | _  $\Rightarrow$  undefined))
| minit0 n (MFOTL.Until  $\varphi$  I  $\psi$ ) = (if safe_formula  $\varphi$ 
  then MUntil True (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
  else (case  $\varphi$  of
    MFOTL.Neg  $\varphi \Rightarrow$  MUntil False (minit0 n  $\varphi$ ) I (minit0 n  $\psi$ ) ([], []) [] []
    | _  $\Rightarrow$  undefined))

```

definition minit :: 'a MFOTL.formula \Rightarrow '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 :: $\mathcal{I} \Rightarrow$ 'a table list \Rightarrow ts list \Rightarrow 'a table list \times 'a table list \times 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 \Rightarrow 'a table list \Rightarrow 'a mbuf2 \Rightarrow 'a mbuf2 **where**

mbuf2_add xs' ys' (xs, ys) = (xs @ xs', ys @ ys')

fun mbuf2_take :: ('a table \Rightarrow 'a table \Rightarrow 'b) \Rightarrow 'a mbuf2 \Rightarrow 'b list \times '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 \Rightarrow 'a table \Rightarrow ts \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow

```

'a mbuf2  $\Rightarrow$  ts list  $\Rightarrow$  'b  $\times$  'a mbuf2  $\times$  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 \Rightarrow 'a list \Rightarrow (nat \rightarrow '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  $\Rightarrow$  None
  | Some f  $\Rightarrow$  (case f x of
    None  $\Rightarrow$  Some (f(x  $\mapsto$  y))
    | Some z  $\Rightarrow$  if y = z then Some f else None))
| match _ _ = None

```

definition update_since :: $\mathcal{I} \Rightarrow$ bool \Rightarrow 'a table \Rightarrow 'a table \Rightarrow ts \Rightarrow

'a msaux \Rightarrow 'a table \times 'a msaux **where**

```

update_since I pos rel1 rel2 nt aux =
  (let aux = (case [(t, join rel pos rel1). (t, rel)  $\leftarrow$  aux, nt - t  $\leq$  right I] of
    []  $\Rightarrow$  [(nt, rel2)]

```

$| x \# aux' \Rightarrow (if \text{fst } x = nt \text{ then } (\text{fst } x, \text{snd } x \cup \text{rel2}) \# aux' \text{ else } (nt, \text{rel2}) \# x \# aux')$
 $in (\text{foldr } (\cup) [\text{rel}. (t, \text{rel}) \leftarrow aux, \text{left } I \leq nt - t] \{\}, aux)$

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

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

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

definition $mstep :: 'a \text{ MFOTL.database} \times ts \Rightarrow 'a \text{ mstate} \Rightarrow (\text{nat} \times 'a \text{ tuple}) \text{ set} \times 'a \text{ mstate}$ **where**
 $mstep \text{tdb} st =$
 $(\text{let } (xs, m) = meval (mstate_n st) (\text{snd } \text{tdb}) (\text{fst } \text{tdb}) (mstate_m st)$
 $\text{in } (\bigcup (\text{set } (\text{map } (\lambda (i, X). (\lambda v. (i, v)) ' X) (\text{List.enumerate } (mstate_i st) xs))),$
 $(\{mstate_i = mstate_i st + \text{length } xs, mstate_m = m, mstate_n = mstate_n st\}))$

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

```

    (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)  $\longrightarrow \tau \sigma i + \text{right } I \geq \tau \sigma k$ }

```

```

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  $\leq j' \implies$  progress  $\sigma$   $\varphi$  j  $\leq$  progress  $\sigma$   $\varphi$  j'

```

```

proof (induction  $\varphi$ )
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    by (cases right I)
    (auto dest: trans_le_add1[OF  $\tau$ _mono] intro!: cInf_superset_mono)
qed auto

```

```

lemma progress_le: progress  $\sigma$   $\varphi$  j  $\leq$  j

```

```

proof (induction  $\varphi$ )
  case (Until  $\varphi$  I  $\psi$ )
  then show ?case
    by (cases right I)
    (auto intro: trans_le_add1[OF  $\tau$ _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  $\varphi \neq \infty \implies \exists j. i \leq$  progress  $\sigma$   $\varphi$  j

```

```

proof (induction  $\varphi$  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  $\varphi$ 1  $\varphi$ 2)
  from Or.prem1 have MFOTL.future_reach  $\varphi$ 1  $\neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ 1) (auto)

```



```

moreover from Or.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where  $i \leq \text{progress } \sigma\ \varphi 1\ j1$  and  $i \leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Or.IH[of  $i$ ] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Or  $\varphi 1\ \varphi 2$ ) (max  $j1\ j2$ )
  by (cases  $j1 \leq j2$ ) (auto elim!: order.trans[OF _ progress_mono])
then show ?case by blast
next
case (Exists  $\varphi$ )
then show ?case by simp
next
case (Prev  $I\ \varphi$ )
from Prev.prems have MFOTL.future_reach  $\varphi \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ ) (auto)
then obtain  $j$  where  $i \leq \text{progress } \sigma\ \varphi\ j$ 
  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  $I\ \varphi$ )
from Next.prems have MFOTL.future_reach  $\varphi \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi$ ) (auto)
then obtain  $j$  where Suc  $i \leq \text{progress } \sigma\ \varphi\ j$ 
  using Next.IH[of Suc  $i$ ] by blast
then show ?case by (auto intro!: exI[of  $_ j$ ])
next
case (Since  $\varphi 1\ I\ \varphi 2$ )
from Since.prems have MFOTL.future_reach  $\varphi 1 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 1$ ) (auto)
moreover from Since.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where  $i \leq \text{progress } \sigma\ \varphi 1\ j1$  and  $i \leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Since.IH[of  $i$ ] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Since  $\varphi 1\ I\ \varphi 2$ ) (max  $j1\ j2$ )
  by (cases  $j1 \leq j2$ ) (auto elim!: order.trans[OF _ progress_mono])
then show ?case by blast
next
case (Until  $\varphi 1\ I\ \varphi 2$ )
from Until.prems obtain  $b$  where [simp]: right  $I = \text{enat } b$ 
  by (cases right  $I$ ) (auto)
obtain  $i'$  where  $i < i'$  and  $\tau\ \sigma\ i + b + 1 \leq \tau\ \sigma\ i'$ 
  using ex_le_τ[where  $x = \tau\ \sigma\ i + b + 1$ ] by (auto simp add: less_eq_Suc_le)
then have  $1: \tau\ \sigma\ i + b < \tau\ \sigma\ i'$  by simp
from Until.prems have MFOTL.future_reach  $\varphi 1 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 1$ ) (auto)
moreover from Until.prems have MFOTL.future_reach  $\varphi 2 \neq \infty$ 
  by (cases MFOTL.future_reach  $\varphi 2$ ) (auto)
ultimately obtain  $j1\ j2$  where Suc  $i' \leq \text{progress } \sigma\ \varphi 1\ j1$  and Suc  $i' \leq \text{progress } \sigma\ \varphi 2\ j2$ 
  using Until.IH[of Suc  $i'$ ] by blast
then have  $i \leq \text{progress } \sigma$  (MFOTL.Until  $\varphi 1\ I\ \varphi 2$ ) (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  $\tau\_mono$ ] intro!: exI[of  $_ \text{max } j1\ j2$ ])
next
case (greatest  $x$ )
with  $\langle i < i' \rangle 1$  show ?case
  by (cases  $j1 \leq j2$ )

```

```

(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  $\varphi$ )
  then show ?case by simp
next
  case (Prev I  $\varphi$ )
  with  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case
  by (cases i) (auto split: if_splits)
next
  case (Next I  $\varphi$ )
  then have  $Suc\ i < plen\ \pi$ 
  by (auto intro: order.strict_trans2[OF __progress_le[of  $\sigma\ \varphi$ ]])
  with Next  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case by simp
next
  case (Since  $\varphi1\ I\ \varphi2$ )
  then have  $i < plen\ \pi$ 
  by (auto elim!: order.strict_trans2[OF __progress_le])
  with Since  $\tau\_prefix\_conv[OF\ assms(1,2)]$  show ?case by auto
next
  case (Until  $\varphi1\ I\ \varphi2$ )
  from Until.prem1 obtain b where [simp]:  $right\ I = enat\ b$ 
  by (cases right I) (auto simp add: Inf_UNIV_nat)
  from Until.prem2 obtain j where  $\tau\ \sigma\ i + b + 1 \leq \tau\ \sigma\ j$ 
   $j \leq progress\ \sigma\ \varphi1\ (plen\ \pi)$   $j \leq progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  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:  $k < progress\ \sigma\ \varphi1\ (plen\ \pi)$  and 2:  $k < progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  if  $\tau\ \sigma\ k \leq \tau\ \sigma\ i + b$  for k
  using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of  $\sigma$ ])+
  have 3:  $k < plen\ \pi$  if  $\tau\ \sigma\ k \leq \tau\ \sigma\ i + b$  for k
  using 1[OF that] by (simp add: less_eq_Suc_le order.trans[OF __progress_le])

  from Until.prem3 have  $i < progress\ \sigma'\ (MFOTL.Until\ \varphi1\ I\ \varphi2)\ (plen\ \pi)$ 
  unfolding progress_prefix_conv[OF assms(1,2)] .
  then obtain j where  $\tau\ \sigma'\ i + b + 1 \leq \tau\ \sigma'\ j$ 
   $j \leq progress\ \sigma'\ \varphi1\ (plen\ \pi)$   $j \leq progress\ \sigma'\ \varphi2\ (plen\ \pi)$ 
  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:  $k < progress\ \sigma\ \varphi1\ (plen\ \pi)$  and 21:  $k < progress\ \sigma\ \varphi2\ (plen\ \pi)$ 
  if  $\tau\ \sigma'\ k \leq \tau\ \sigma'\ i + b$  for k
  unfolding progress_prefix_conv[OF assms(1,2)]
  using that by (fastforce elim!: order.strict_trans2[rotated] intro: less_τD[of  $\sigma'$ ])+
  have 31:  $k < plen\ \pi$  if  $\tau\ \sigma'\ k \leq \tau\ \sigma'\ i + b$  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]  $\tau\_prefix\_conv[OF\ assms(1,2)\ 3]$  show ?case
  by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF  $\tau\_mono$ , rotated])
  next
  case (RL j)
  with Until.IH(1)[OF 11] Until.IH(2)[OF 21]  $\tau\_prefix\_conv[OF\ assms(1,2)\ 31]$  show ?case
  by (auto 0 4 simp: le_diff_conv add.commute dest: less_imp_le order.trans[OF  $\tau\_mono$ , rotated])
qed
qed

```

6.4 Specification

definition $pprogress :: 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.prefix} \Rightarrow \text{nat}$ **where**
 $pprogress \varphi \pi = (\text{THE } n. \forall \sigma. \text{prefix_of } \pi \sigma \longrightarrow \text{progress } \sigma \varphi (\text{plen } \pi) = n)$

lemma $pprogress_eq: \text{prefix_of } \pi \sigma \Longrightarrow pprogress \varphi \pi = \text{progress } \sigma \varphi (\text{plen } \pi)$
unfolding $pprogress_def$ **using** $\text{progress_prefix_conv}$
by blast

locale $\text{future_bounded_mfotl} =$
fixes $\varphi :: 'a \text{ MFOTL.formula}$
assumes $\text{future_bounded}: \text{MFOTL.future_reach } \varphi \neq \infty$

sublocale $\text{future_bounded_mfotl} \subseteq \text{sliceable_timed_progress } \text{MFOTL.nfv } \varphi \text{ MFOTL.fv } \varphi \text{ relevant_events}$

φ
 $\lambda \sigma v i. \text{MFOTL.sat } \sigma v i \varphi pprogress \varphi$
proof ($\text{unfold_locales}, \text{goal_cases}$)
case (1 x)
then show $?case$ **by** ($\text{simp add: fvi_less_nfv}$)
next
case (2 $v v' \sigma i$)
then show $?case$ **by** ($\text{simp cong: sat_fvi_cong}[\text{rule_format}]$)
next
case (3 $v S \sigma i$)
then show $?case$ **using** $\text{sat_slice_iff}[\text{of } v, \text{symmetric}]$ **by** simp
next
case (4 $\pi \pi'$)
moreover obtain σ **where** $\text{prefix_of } \pi' \sigma$
using $\text{ex_prefix_of } ..$
moreover have $\text{prefix_of } \pi \sigma$
using $\text{prefix_of_antimono}[\text{OF } \langle \pi \leq \pi' \rangle \langle \text{prefix_of } \pi' \sigma \rangle]$.
ultimately show $?case$
by ($\text{simp add: pprogress_eq plen_mono progress_mono}$)
next
case (5 σx)
obtain j **where** $x \leq \text{progress } \sigma \varphi j$
using $\text{future_bounded progress_ge}$ **by** blast
then have $x \leq pprogress \varphi (\text{take_prefix } j \sigma)$
by ($\text{simp add: pprogress_eq}[\text{of } _ \sigma]$)
then show $?case$ **by** force
next
case (6 $\pi \sigma \sigma' i v$)
then have $i < \text{progress } \sigma \varphi (\text{plen } \pi)$
by ($\text{simp add: pprogress_eq}$)
with 6 **show** $?case$
using sat_prefix_conv **by** blast
next
case (7 $\pi \pi'$)
then have $\text{plen } \pi = \text{plen } \pi'$
by $\text{transfer } (\text{simp add: list_eq_iff_nth_eq})$
moreover obtain $\sigma \sigma'$ **where** $\text{prefix_of } \pi \sigma \text{ prefix_of } \pi' \sigma'$
using ex_prefix_of **by** blast+
moreover have $\forall i < \text{plen } \pi. \tau \sigma i = \tau \sigma' i$
using 7 calculation
by $\text{transfer } (\text{simp add: list_eq_iff_nth_eq})$
ultimately show $?case$
by ($\text{simp add: pprogress_eq progress_time_conv}$)
qed

locale *monitorable_mfotl* =
fixes $\varphi :: 'a \text{ MFOTL.formula}$
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* :: $\text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow (\text{nat} \Rightarrow 'a \text{ table} \Rightarrow \text{bool}) \Rightarrow (\text{nat} \Rightarrow 'a \text{ table} \Rightarrow \text{bool}) \Rightarrow 'a \text{ mbuf2} \Rightarrow \text{bool}$ **where**
wf_mbuf2 $i \ j \ P \ Q \ \text{buf} \longleftrightarrow i \leq j \wedge i \leq j \wedge (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow \text{list_all2 } P \ [i..<j] \ xs \wedge \text{list_all2 } Q \ [i..<j] \ ys)$

definition *wf_mbuf2'* :: $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ mbuf2} \Rightarrow \text{bool}$ **where**
wf_mbuf2' $\sigma \ j \ n \ R \ \varphi \ \psi \ \text{buf} \longleftrightarrow \text{wf_mbuf2} \ (\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)) \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $(\lambda i. \text{qtable } n \ (\text{MFOTL.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi))$
 $(\lambda i. \text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \psi)) \ \text{buf}$

lemma *wf_mbuf2'_UNIV_alt*: *wf_mbuf2'* $\sigma \ j \ n \ \text{UNIV } \varphi \ \psi \ \text{buf} \longleftrightarrow (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow \text{list_all2} \ (\lambda i. \text{wf_table } n \ (\text{MFOTL.fv } \varphi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi)) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j) \ ..< \ (\text{progress } \sigma \ \varphi \ j)] \ xs \wedge \text{list_all2} \ (\lambda i. \text{wf_table } n \ (\text{MFOTL.fv } \psi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \psi)) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j) \ ..< \ (\text{progress } \sigma \ \psi \ j)] \ ys)$

unfolding *wf_mbuf2'_def* *wf_mbuf2_def*

by (*simp add*: *mem_restr_UNIV*[*THEN eqTrueI*, *abs_def*] *split*: *prod.split*)

definition *wf_ts* :: $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow \text{ts list} \Rightarrow \text{bool}$ **where**

wf_ts $\sigma \ j \ \varphi \ \psi \ ts \longleftrightarrow \text{list_all2} \ (\lambda i \ t. t = \tau \ \sigma \ i) \ [\min \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j) \ ..< j] \ ts$

abbreviation *Since_pos* $\varphi \ I \ \psi \equiv \text{MFOTL.Since} \ (\text{if } \text{pos} \ \text{then } \varphi \ \text{else } \text{MFOTL.Neg } \varphi) \ I \ \psi$

definition *wf_since_aux* :: $'a \text{ MFOTL.trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ list set} \Rightarrow \text{bool} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow \mathcal{I} \Rightarrow 'a \text{ MFOTL.formula} \Rightarrow 'a \text{ msaux} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
wf_since_aux $\sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ \text{aux} \ ne \longleftrightarrow \text{sorted_wrt} \ (\lambda x \ y. \text{fst } x > \text{fst } y) \ \text{aux} \wedge (\forall t \ X. (t, X) \in \text{set } \text{aux} \longrightarrow ne \neq 0 \wedge t \leq \tau \ \sigma \ (ne-1) \wedge \tau \ \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \ \sigma \ i = t)) \wedge \text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne-1) \ (\text{Since_pos } \varphi \ (\text{point } (\tau \ \sigma \ (ne-1) - t)) \ \psi)) \ X) \wedge (\forall t. ne \neq 0 \wedge t \leq \tau \ \sigma \ (ne-1) \wedge \tau \ \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \ \sigma \ i = t) \longrightarrow (\exists X. (t, X) \in \text{set } \text{aux}))$

lemma *qtable_mem_restr_UNIV*: *qtable* $n \ A \ (\text{mem_restr } \text{UNIV}) \ Q \ X = \text{wf_table } n \ A \ Q \ X$
unfolding *qtable_def* **by** *auto*

lemma *wf_since_aux_UNIV_alt*:

wf_since_aux $\sigma \ n \ \text{UNIV } \text{pos } \varphi \ I \ \psi \ \text{aux} \ ne \longleftrightarrow \text{sorted_wrt} \ (\lambda x \ y. \text{fst } x > \text{fst } y) \ \text{aux} \wedge (\forall t \ X. (t, X) \in \text{set } \text{aux} \longrightarrow ne \neq 0 \wedge t \leq \tau \ \sigma \ (ne-1) \wedge \tau \ \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \ \sigma \ i = t)) \wedge \text{wf_table } n \ (\text{MFOTL.fv } \psi) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne-1) \ (\text{Since_pos } \varphi \ (\text{point } (\tau \ \sigma \ (ne-1) - t)) \ \psi)) \ X) \wedge (\forall t. ne \neq 0 \wedge t \leq \tau \ \sigma \ (ne-1) \wedge \tau \ \sigma \ (ne-1) - t \leq \text{right } I \wedge (\exists i. \tau \ \sigma \ i = t) \longrightarrow (\exists X. (t, X) \in \text{set } \text{aux}))$

unfolding *wf_since_aux_def* *qtable_mem_restr_UNIV* ..

definition $wf_until_aux :: 'a MFOTL.trace \Rightarrow nat \Rightarrow 'a list set \Rightarrow bool \Rightarrow 'a MFOTL.formula \Rightarrow \mathcal{I} \Rightarrow 'a MFOTL.formula \Rightarrow 'a muaux \Rightarrow nat \Rightarrow bool$ **where**
 $wf_until_aux \sigma n R pos \varphi I \psi aux ne \longleftrightarrow list_all2 (\lambda x i. case x of (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$
 $qtable n (MFOTL.fv \varphi) (mem_restr R) (\lambda v. if pos then (\forall k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma$
 $(map\ the\ v) k \varphi)$
 $else (\exists k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)) r1 \wedge$
 $qtable n (MFOTL.fv \psi) (mem_restr R) (\lambda v. (\exists j. i \leq j \wedge j < ne + length\ aux \wedge mem (\tau \sigma j - \tau \sigma$
 $i) I \wedge$
 $MFOTL.sat \sigma (map\ the\ v) j \psi \wedge$
 $(\forall k \in \{i..<j\}. if pos then MFOTL.sat \sigma (map\ the\ v) k \varphi else \neg MFOTL.sat \sigma (map\ the\ v) k \varphi)))$
 $r2)$
 $aux [ne..<ne+length\ aux]$

lemma $wf_until_aux_UNIV_alt:$

$wf_until_aux \sigma n UNIV pos \varphi I \psi aux ne \longleftrightarrow list_all2 (\lambda x i. case x of (t, r1, r2) \Rightarrow t = \tau \sigma i \wedge$
 $wf_table n (MFOTL.fv \varphi) (\lambda v. if pos$
 $then (\forall k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)$
 $else (\exists k \in \{i..<ne+length\ aux\}. MFOTL.sat \sigma (map\ the\ v) k \varphi)) r1 \wedge$
 $wf_table n (MFOTL.fv \psi) (\lambda v. \exists j. i \leq j \wedge j < ne + length\ aux \wedge mem (\tau \sigma j - \tau \sigma i) I \wedge$
 $MFOTL.sat \sigma (map\ the\ v) j \psi \wedge$
 $(\forall k \in \{i..<j\}. if pos then MFOTL.sat \sigma (map\ the\ v) k \varphi else \neg MFOTL.sat \sigma (map\ the\ v) k \varphi)))$
 $r2)$
 $aux [ne..<ne+length\ aux]$
unfolding $wf_until_aux_def\ qtable_mem_restr_UNIV ..$

inductive $wf_mformula :: 'a MFOTL.trace \Rightarrow nat \Rightarrow$

$nat \Rightarrow 'a list set \Rightarrow 'a mformula \Rightarrow 'a MFOTL.formula \Rightarrow bool$
for σj **where**
 $Eq: MFOTL.is_Const t1 \vee MFOTL.is_Const t2 \Longrightarrow$
 $\forall x \in MFOTL.fv_trm t1. x < n \Longrightarrow \forall x \in MFOTL.fv_trm t2. x < n \Longrightarrow$
 $wf_mformula \sigma j n R (MRel (eq_rel n t1 t2)) (MFOTL.Eq t1 t2)$
 $| neq_Const: \varphi = MRel (neq_rel n (MFOTL.Const x) (MFOTL.Const y)) \Longrightarrow$
 $wf_mformula \sigma j n R \varphi (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))$
 $| neq_Var: x < n \Longrightarrow$
 $wf_mformula \sigma j n R (MRel empty_table) (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var$
 $x)))$
 $| Pred: \forall x \in MFOTL.fv (MFOTL.Pred e ts). x < n \Longrightarrow$
 $wf_mformula \sigma j n R (MPred e ts) (MFOTL.Pred e ts)$
 $| And: wf_mformula \sigma j n R \varphi \varphi' \Longrightarrow wf_mformula \sigma j n R \psi \psi' \Longrightarrow$
 $if pos then \chi = MFOTL.And \varphi' \psi' \wedge \neg (safe_formula (MFOTL.Neg \psi') \wedge MFOTL.fv \psi' \subseteq MFOTL.fv$
 $\varphi')$
 $else \chi = MFOTL.And_Not \varphi' \psi' \wedge safe_formula \psi' \wedge MFOTL.fv \psi' \subseteq MFOTL.fv \varphi' \Longrightarrow$
 $wf_mbuf2' \sigma j n R \varphi' \psi' buf \Longrightarrow$
 $wf_mformula \sigma j n R (MAnd \varphi pos \psi buf) \chi$
 $| Or: wf_mformula \sigma j n R \varphi \varphi' \Longrightarrow wf_mformula \sigma j n R \psi \psi' \Longrightarrow$
 $MFOTL.fv \varphi' = MFOTL.fv \psi' \Longrightarrow$
 $wf_mbuf2' \sigma j n R \varphi' \psi' buf \Longrightarrow$
 $wf_mformula \sigma j n R (MOr \varphi \psi buf) (MFOTL.Or \varphi' \psi')$
 $| Exists: wf_mformula \sigma j (Suc n) (lift_envs R) \varphi \varphi' \Longrightarrow$
 $wf_mformula \sigma j n R (MExists \varphi) (MFOTL.Exists \varphi')$
 $| Prev: wf_mformula \sigma j n R \varphi \varphi' \Longrightarrow$
 $first \longleftrightarrow j = 0 \Longrightarrow$
 $list_all2 (\lambda i. qtable n (MFOTL.fv \varphi') (mem_restr R) (\lambda v. MFOTL.sat \sigma (map\ the\ v) i \varphi'))$
 $[min (progress \sigma \varphi' j) (j-1)..<progress \sigma \varphi' j] buf \Longrightarrow$
 $list_all2 (\lambda i t. t = \tau \sigma i) [min (progress \sigma \varphi' j) (j-1)..<j] nts \Longrightarrow$
 $wf_mformula \sigma j n R (MPrev I \varphi first buf nts) (MFOTL.Prev I \varphi')$

| *Next*: $wf_mformula \sigma j n R \varphi \varphi' \implies$
 $first \longleftrightarrow progress \sigma \varphi' j = 0 \implies$
 $list_all2 (\lambda i t. t = \tau \sigma i) [progress \sigma \varphi' j - 1..<j] nts \implies$
 $wf_mformula \sigma j n R (MNext I \varphi first nts) (MFOTL.Next I \varphi')$
| *Since*: $wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $if\ pos\ then\ \varphi'' = \varphi' \ else\ \varphi'' = MFOTL.Neg\ \varphi' \implies$
 $safe_formula \varphi'' = pos \implies$
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' buf \implies$
 $wf_ts \sigma j \varphi' \psi' nts \implies$
 $wf_since_aux \sigma n R pos \varphi' I \psi' aux (progress \sigma (MFOTL.Since \varphi'' I \psi') j) \implies$
 $wf_mformula \sigma j n R (MSince pos \varphi I \psi buf nts aux) (MFOTL.Since \varphi'' I \psi')$
| *Until*: $wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $if\ pos\ then\ \varphi'' = \varphi' \ else\ \varphi'' = MFOTL.Neg\ \varphi' \implies$
 $safe_formula \varphi'' = pos \implies$
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' buf \implies$
 $wf_ts \sigma j \varphi' \psi' nts \implies$
 $wf_until_aux \sigma n R pos \varphi' I \psi' aux (progress \sigma (MFOTL.Until \varphi'' I \psi') j) \implies$
 $progress \sigma (MFOTL.Until \varphi'' I \psi') j + length\ aux = \min (progress \sigma \varphi' j) (progress \sigma \psi' j) \implies$
 $wf_mformula \sigma j n R (MUntil pos \varphi I \psi buf nts aux) (MFOTL.Until \varphi'' I \psi')$

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 $minit0_And: \neg (safe_formula (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \implies$
 $minit0\ n (MFOTL.And \varphi \psi) = MAnd (minit0\ n \varphi) True (minit0\ n \psi) ([], [])$
unfolding $MFOTL.And_def$ **by** $simp$

lemma $minit0_And_Not: safe_formula \psi \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi \implies$
 $minit0\ n (MFOTL.And_Not \varphi \psi) = (MAnd (minit0\ n \varphi) False (minit0\ n \psi) ([], []))$
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 $wf_minit0: safe_formula \varphi \implies \forall x \in MFOTL.fv \varphi. x < n \implies$
 $wf_mformula \sigma 0 n R (minit0\ n \varphi) \varphi$
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 $wf_mstate_minit: safe_formula \varphi \implies wf_mstate \varphi pnil R (minit \varphi)$
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 \implies wf_tuple n ($\bigcup t \in \text{set } ts. \text{MFOTL.fv_trm } t$) (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 $\implies \forall t \in \text{set } ts. x \notin \text{MFOTL.fv_trm } t \implies f x = \text{None}$*

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 $\implies t \in \text{set } ts \implies x \in \text{MFOTL.fv_trm } t \implies f x \neq \text{None}$*

by (*induction ts xs arbitrary; f rule: match.induct*) (*auto split: if_splits option.splits*)

lemma *match_eval_trm*: $\forall t \in \text{set } ts. \forall i \in \text{MFOTL.fv_trm } t. i < n \implies \text{Some } f = \text{match } ts \text{ xs} \implies \text{map } (\text{MFOTL.eval_trm } (\text{tabulate } (\lambda i. \text{the } (f \ i)) \ 0 \ n)) \ ts = \text{xs}$

proof (*induction ts xs arbitrary; f rule: match.induct*)

case ($\exists x \ ts \ y \ ys$)

from $\exists(1)[\text{symmetric}] \exists(2,3)$ **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) $\implies x \in A \implies x < n \implies \exists y. f x = \text{Some } y$*

unfolding *wf_tuple_def* **by** *auto*

lemma *ex_match*: *wf_tuple n ($\bigcup t \in \text{set } ts. \text{MFOTL.fv_trm } t$) v $\implies \forall t \in \text{set } ts. \forall x \in \text{MFOTL.fv_trm } t. x < n \implies$*

$\exists f. \text{match } ts \ (\text{map } (\text{MFOTL.eval_trm } (\text{map the } v)) \ ts) = \text{Some } f \wedge v = \text{tabulate } f \ 0 \ n$

proof (*induction ts map (MFOTL.eval_trm (map the v)) ts arbitrary; v rule: match.induct*)

case ($\exists x \ ts \ y \ ys$)

then show *?case*

proof (*cases x \in ($\bigcup t \in \text{set } ts. \text{MFOTL.fv_trm } t$)*)

case *True*

with \exists **show** *?thesis*

by (*auto simp: insert_absorb dest!: wf_tuple_tabulate_Some meta_spec[of _ v]*)

next

case *False*

with $\exists(3,4)$ **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* $\exists(2-4)$ **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* $\exists(1)[\text{of } v[x := \text{None}]]$)

(*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* $\exists(3)$

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 \in eq_rel n t1 t2 $\implies \text{MFOTL.is_Const } t1 \vee \text{MFOTL.is_Const } t2 \implies \forall x \in \text{MFOTL.fv_trm } t1 \cup \text{MFOTL.fv_trm } t2. x < n \implies$*

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*: $wf_tuple\ n\ (MFOTL.fv_trm\ t1\ \cup\ MFOTL.fv_trm\ t2)\ v\ \Longrightarrow$
 $MFOTL.is_Const\ t1\ \vee\ MFOTL.is_Const\ t2\ \Longrightarrow$
 $MFOTL.eval_trm\ (map\ the\ v)\ t1\ =\ MFOTL.eval_trm\ (map\ the\ v)\ t2\ \Longrightarrow$
 $v\ \in\ eq_rel\ n\ t1\ t2$
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*: $MFOTL.is_Const\ t1\ \vee\ MFOTL.is_Const\ t2\ \Longrightarrow$
 $table\ n\ (MFOTL.fv_trm\ t1\ \cup\ MFOTL.fv_trm\ t2)\ (eq_rel\ n\ t1\ t2)$
by (*cases* *t1*; *cases* *t2*; *simp*)

lemma *wf_tuple_Suc_fviD*: $wf_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ v\ \Longrightarrow\ wf_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ (tl\ v)$
unfolding *wf_tuple_def* **by** (*simp* *add: fvi_Suc_nth_tl*)

lemma *table_fvi_tl*: $table\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ X\ \Longrightarrow\ table\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ (tl\ 'X)$
unfolding *table_def* **by** (*auto* *intro: wf_tuple_Suc_fviD*)

lemma *wf_tuple_Suc_fvi_SomeI*: $0\ \in\ MFOTL.fvi\ b\ \varphi\ \Longrightarrow\ wf_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ v\ \Longrightarrow$
 $wf_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ (Some\ x\ \# v)$
unfolding *wf_tuple_def*
by (*auto* *simp: fvi_Suc_less_Suc_eq_0_disj*)

lemma *wf_tuple_Suc_fvi_NoneI*: $0\ \notin\ MFOTL.fvi\ b\ \varphi\ \Longrightarrow\ wf_tuple\ n\ (MFOTL.fvi\ (Suc\ b)\ \varphi)\ v\ \Longrightarrow$
 $wf_tuple\ (Suc\ n)\ (MFOTL.fvi\ b\ \varphi)\ (None\ \# v)$
unfolding *wf_tuple_def*
by (*auto* *simp: fvi_Suc_less_Suc_eq_0_disj*)

lemma *qtable_project_fv*: $qtable\ (Suc\ n)\ (fv\ \varphi)\ (mem_restr\ (lift_envs\ R))\ P\ X\ \Longrightarrow$
 $qtable\ n\ (MFOTL.fvi\ (Suc\ 0)\ \varphi)\ (mem_restr\ R)$
($\lambda v. \exists x. P\ ((if\ 0\ \in\ fv\ \varphi\ then\ Some\ x\ else\ None)\ \# v))\ (tl\ 'X)$)

using *neq0_conv* **by** (*fastforce* *simp: image_iff Bex_def fvi_Suc_elim!: qtable_cong dest!: qtable_project*)

lemma *mprev*: $mprev_next\ I\ xs\ ts\ =\ (ys,\ xs',\ ts')\ \Longrightarrow$
 $list_all2\ P\ [i..<j]\ xs\ \Longrightarrow\ list_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts\ \Longrightarrow\ i \leq j' \Longrightarrow i < j \Longrightarrow$
 $list_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i))\ -\ \tau\ \sigma\ i)\ I\ then\ P\ i\ X\ else\ X = empty_table)$
 $[i..<min\ j'\ (j-1)]\ ys\ \wedge$
 $list_all2\ P\ [min\ j'\ (j-1)..<j]\ xs'\ \wedge$
 $list_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ j'\ (j-1)..<j]\ ts'$

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*: $mprev_next\ I\ xs\ ts\ =\ (ys,\ xs',\ ts')\ \Longrightarrow$
 $list_all2\ P\ [Suc\ i..<j]\ xs\ \Longrightarrow\ list_all2\ (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts\ \Longrightarrow\ Suc\ i \leq j' \Longrightarrow i < j \Longrightarrow$
 $list_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i))\ -\ \tau\ \sigma\ i)\ I\ then\ P\ (Suc\ i)\ X\ else\ X = empty_table)$
 $[i..<min\ (j'-1)\ (j-1)]\ ys\ \wedge$

```

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 ≠ 0 t ≤ τ σ (ne-1) τ σ ne - t ≤ right I ∃ i. τ σ i = t
then obtain X where (t, X) ∈ 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 ⟨τ σ ne - t ≤ right I⟩ have (t, join X pos rel1) ∈ set aux0
  unfolding aux0_def by (auto elim!: beX[rotated] intro!: exI[of _ X])
then show ∃ X. (t, X) ∈ set aux0
  by blast
qed
have aux0_Nil: aux0 = [] ⇒ ne = 0 ∨ ne ≠ 0 ∧ (∀ t. t ≤ τ σ (ne-1) ∧ τ σ ne - t ≤ right I →
  (∃ i. τ σ i = t))
  using in_aux0_2 by (cases ne = 0) (auto)

have aux'_eq: aux' = (case aux0 of
  [] ⇒ [(τ σ ne, rel2)]
  | x # aux' ⇒ (if fst x = τ σ ne then (fst x, snd x ∪ rel2) # aux' else (τ σ ne, rel2) # x # aux'))
  using result_eq unfolding aux0_def update_since_def Let_def by simp
have sorted_aux': sorted_wrt (λx 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 ≤ τ σ ne ∧ 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 (Sincep pos φ (point (τ
σ ne - t)) ψ)) X
  if aux': (t, X) ∈ 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 _ τ σ (ne-1)] elim!: qtable_cong[OF _ refl])
next
case (Cons a as)
show ?thesis
proof (cases t = τ σ ne)
  case t: True
  show ?thesis
  proof (cases fst a = τ σ ne)
  case True
  with aux' Cons t have X = snd a ∪ rel2
  unfolding aux'_eq using sorted_aux0 by auto
  moreover from in_aux0_1[of fst a snd a] Cons have ne ≠ 0
  fst a ≤ τ σ (ne - 1) τ σ ne - fst a ≤ right I ∃ i. τ σ i = fst a
  qtable n (fv ψ) (mem_restr R) (λv. MFOTL.sat σ (map the v) (ne - 1)
  (Sincep pos φ (point (τ σ (ne - 1) - fst a)) ψ) ∧ (if pos then MFOTL.sat σ (map the v) ne φ
  else ¬ 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 τ σ (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) (Since 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 (Since 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 (Since 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  $\tau$   $\sigma$  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  $\sigma$  n R pos  $\varphi$  I  $\psi$  aux ne
  and qtable1: qtable n (MFOTL.fv  $\varphi$ ) (mem_restr R) ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the v) (ne + length
aux)  $\varphi$ ) rel1
  and qtable2: qtable n (MFOTL.fv  $\psi$ ) (mem_restr R) ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the v) (ne + length
aux)  $\psi$ ) rel2
  and fvi_subset: MFOTL.fv  $\varphi$   $\subseteq$  MFOTL.fv  $\psi$ 
shows wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  (update_until I pos rel1 rel2 ( $\tau$   $\sigma$  (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  $\exists$  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  $\sigma$  n R pos  $\varphi$  I  $\psi$  (a # aux) ne  $\implies$ 
wf_until_aux  $\sigma$  n R pos  $\varphi$  I  $\psi$  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  $\sigma$  n R pos  $\varphi$  I  $\psi$  ((t, a1, a2) # aux) ne  $\implies$  t =  $\tau$   $\sigma$  ne
unfolding wf_until_aux_def
by (simp add: upt_conv_Cons del: upt_Suc)

```

lemma *wf_until_aux_Cons3*: $wf_until_aux\ \sigma\ n\ R\ pos\ \varphi\ I\ \psi\ ((t, a1, a2) \# aux)\ ne \implies$
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem_restr\ R)\ (\lambda v. (\exists j. ne \leq j \wedge j < Suc\ (ne + length\ aux) \wedge mem\ (\tau\ \sigma\ j -$
 $\tau\ \sigma\ ne)\ I \wedge$
 $MFOTL.sat\ \sigma\ (map\ the\ v)\ j\ \psi \wedge (\forall k \in \{ne..<j\}. if\ pos\ then\ MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi\ else\ \neg$
 $MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi))\ a2$
unfolding *wf_until_aux_def*
by (*simp add: upt_conv_Cons del: upt_Suc*)

lemma *upt_append*: $a \leq b \implies b \leq c \implies [a..<b] @ [b..<c] = [a..<c]$
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* $(\lambda i\ z. \exists x\ y. P\ i\ x \wedge Q\ i\ y \wedge z = f\ x\ y)\ [i..<min\ ja\ jb]\ xs$
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 $\bigwedge k\ x\ y\ t\ z. i \leq k \implies Suc\ k \leq ja \implies Suc\ k \leq jb \implies$
 $P\ k\ x \implies Q\ k\ y \implies R\ k\ t \implies U\ k\ z \implies U\ (Suc\ k)\ (f\ x\ y\ t\ z)$
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 $j \leq j'$
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 $(force\ intro!:\ mbuf2_take_eqD[OF\ eq]\ wf_mbuf2_add[OF\ _]\ xs\ ys]\ progress_mono[OF\ \langle j \leq j' \rangle])+$

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 $j \leq j'$
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 $(blast\ intro!:\ mbuf2t_take_eqD[OF\ eq]\ wf_mbuf2_add[OF\ _]\ xs\ ys]\ progress_mono[OF\ \langle j \leq j' \rangle]\ progress_le)+$

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 $j \leq j'$
and $base: U\ (min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j))\ z$
and $step: \bigwedge k\ X\ Y\ z. min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j) \leq k \implies$
 $Suc\ k \leq progress\ \sigma\ \varphi\ j' \implies Suc\ k \leq progress\ \sigma\ \psi\ j' \implies$
 $qtable\ n\ (MFOTL.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \varphi)\ X \implies$
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ k\ \psi)\ Y \implies$
 $U\ k\ z \implies U\ (Suc\ k)\ (f\ X\ Y\ (\tau\ \sigma\ k)\ z)$
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 $(blast\ intro!:\ mbuf2t_take_induct[OF\ eq]\ wf_mbuf2_add[OF\ _]\ xs\ ys]\ progress_mono[OF\ \langle j \leq j' \rangle]\ progress_le\ base\ step)$

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

lemma $list_all2_upt_Cons$: $P\ a\ x \implies list_all2\ P\ [Suc\ a..<b]\ xs \implies Suc\ a \leq b \implies$
 $list_all2\ P\ [a..<b]\ (x\ \#\ xs)$
by $(simp\ add:\ list_all2_Cons2\ upt_eq_Cons_conv)$

lemma *list_all2_upt_append*: $list_all2\ P\ [a..<b]\ xs \implies list_all2\ P\ [b..<c]\ ys \implies a \leq b \implies b \leq c \implies list_all2\ P\ [a..<c]\ (xs\ @\ ys)$
by (*induction xs arbitrary: a*) (*auto simp add: list_all2_Cons2_upt_eq_Cons_conv*)

lemma *meval*:

assumes *wf_mformula* $\sigma\ j\ n\ R\ \varphi\ \varphi'$
shows *case meval* $n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi$ of $(xs, \varphi_n) \Rightarrow wf_mformula\ \sigma\ (Suc\ j)\ n\ R\ \varphi_n\ \varphi' \wedge list_all2\ (\lambda i. qtable\ n\ (MFOTL.fv\ \varphi')\ (mem_restr\ R))\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi')$
 $[progress\ \sigma\ \varphi'\ j..<progress\ \sigma\ \varphi'\ (Suc\ j)]\ xs$
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* $\varphi\ pos\ \psi\ buf$)
from *MAnd.prem*s **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* $\varphi\ \psi\ buf$)
from *MOr.prem*s **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.prem*s **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* $I\ \varphi\ first\ buf\ nts$)
from *MPrev.prem*s **show** *?case*
proof (*cases pred: wf_mformula*)
case (*Prev* ψ)
let $?xs = fst\ (meval\ n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi)$
let $? \varphi = snd\ (meval\ n\ (\tau\ \sigma\ j)\ (\Gamma\ \sigma\ j)\ \varphi)$
let $?ls = fst\ (mprev_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j]))$
let $?rs = fst\ (snd\ (mprev_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j])))$
let $?ts = snd\ (snd\ (mprev_next\ I\ (buf\ @\ ?xs)\ (nts\ @\ [\tau\ \sigma\ j])))$
let $?P = \lambda i\ X. qtable\ n\ (fv\ \psi)\ (mem_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi)\ X$
let $?min = min\ (progress\ \sigma\ \psi\ (Suc\ j))\ (Suc\ j - 1)$
from *Prev MPrev.IH*[of $n\ R\ \psi$] **have** *IH*: *wf_mformula* $\sigma\ (Suc\ j)\ n\ R\ ? \varphi\ \psi$ **and**
 $list_all2\ ?P\ [progress\ \sigma\ \psi\ j..<progress\ \sigma\ \psi\ (Suc\ j)]\ ?xs$ **by** *auto*
with *Prev*(4,5) **have** $list_all2\ (\lambda i\ X. if\ mem\ (\tau\ \sigma\ (Suc\ i)) - \tau\ \sigma\ i\ I\ then\ ?P\ i\ X\ else\ X = empty_table)$
 $[min\ (progress\ \sigma\ \psi\ j)\ (j - 1)..<?min)]\ ?ls \wedge$


```

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.premis 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.premis 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) (mbuf2t_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.prem 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_mbuf2t' 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  $\sigma$   $\varphi''$   $j$ ) (progress  $\sigma$   $\psi''$   $j$ )
  by (cases pred: wf_mformula) (auto)
have  $\varphi'''$ : progress  $\sigma$   $\varphi'''$   $j$  = progress  $\sigma$   $\varphi''$   $j$  for  $j$ 
  using pos by (simp_all add: progress.simps split: if_splits)
have nts_snoc: list_all2 ( $\lambda i$   $t$ .  $t = \tau \sigma i$ )
  [ $\min$  (progress  $\sigma$   $\varphi''$   $j$ ) (progress  $\sigma$   $\psi''$   $j$ ).. $\text{Suc } j$ ] (nts @ [ $\tau \sigma j$ ])
  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  $n$  ( $\tau \sigma j$ ) ( $\Gamma \sigma j$ )  $\varphi$ ) = xs
  and eval_ψ: fst (meval  $n$  ( $\tau \sigma j$ ) ( $\Gamma \sigma j$ )  $\psi$ ) = ys
  and eq1: mbuf2t_take (update_until  $I$  pos) aux (mbuf2_add xs ys buf) (nts @ [ $\tau \sigma j$ ]) =
    (aux', buf', nts')
  and eq2: eval_until  $I$  (case nts' of []  $\Rightarrow \tau \sigma j$  | nt # _  $\Rightarrow nt$ ) aux' = (zs, aux'')
have update1: wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi'''$   $I$   $\psi''$  aux' (progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ )  $j$ )  $\wedge$ 
  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ )  $j$  + length aux' =
  min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))
  using MUntil.IH(1)[OF  $\varphi$ ] eval_φ MUntil.IH(2)[OF  $\psi$ ]
  eval_ψ nts_snoc nts_snoc length_aux aux fvi_subset
  unfolding  $\varphi'''$ 
  by (elim mbuf2t_take_add_induct'[where  $j' = \text{Suc } j$ , OF eq1 buf])
  (auto simp: length_update_until elim: wf_update_until)
have nts': wf_ts  $\sigma$  ( $\text{Suc } j$ )  $\varphi''$   $\psi''$  nts'
  using MUntil.IH(1)[OF  $\varphi$ ] eval_φ MUntil.IH(2)[OF  $\psi$ ] 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  $i \leq$  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )  $\implies$ 
  wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux'  $i \implies$ 
   $i$  + length aux' = min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))  $\implies$ 
  wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux'' (progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ ))  $\wedge$ 
   $i$  + length zs = progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )  $\wedge$ 
   $i$  + length zs + length aux'' = min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))  $\wedge$ 
  list_all2 ( $\lambda i$ . qtable  $n$  (MFOTL.fv  $\psi''$ ) (mem_restr  $R$ )
    ( $\lambda v$ . MFOTL.sat  $\sigma$  (map the  $v$ )  $i$  (formula.Until (if pos then  $\varphi''$  else MFOTL.Neg  $\varphi''$ )  $I$   $\psi''$ )))
  [ $i$ .. $i$  + 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.prem1(2) have aux': wf_until_aux  $\sigma$   $n$   $R$  pos  $\varphi''$   $I$   $\psi''$  aux' ( $\text{Suc } i$ )
    by (rule wf_until_aux_Cons)
  from Cons.prem1(2) have 1:  $t = \tau \sigma i$ 
    unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons1)
  from Cons.prem1(2) have 3: qtable  $n$  (MFOTL.fv  $\psi''$ ) (mem_restr  $R$ ) ( $\lambda v$ .
    ( $\exists j \geq i$ .  $j < \text{Suc } (i + \text{length } aux') \wedge \text{mem } (\tau \sigma j - \tau \sigma i) I \wedge \text{MFOTL.sat } \sigma$  (map the  $v$ )  $j$   $\psi'' \wedge$ 
    ( $\forall k \in \{i..<j\}$ . if pos then MFOTL.sat  $\sigma$  (map the  $v$ )  $k$   $\varphi''$  else  $\neg \text{MFOTL.sat } \sigma$  (map the  $v$ )  $k$   $\varphi''$ )))
a2
    unfolding  $\langle a = (t, a1, a2) \rangle$  by (rule wf_until_aux_Cons3)
  from Cons.prem1(3) have Suc_i_aux':  $\text{Suc } i + \text{length } aux' =$ 
    min (progress  $\sigma$   $\varphi'''$  ( $\text{Suc } j$ )) (progress  $\sigma$   $\psi''$  ( $\text{Suc } j$ ))
    by simp
  have  $i \geq$  progress  $\sigma$  (formula.Until  $\varphi'''$   $I$   $\psi''$ ) ( $\text{Suc } j$ )
    if enat (case nts' of []  $\Rightarrow \tau \sigma j$  | nt #  $x \Rightarrow nt$ )  $\leq$  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  $\varphi'''$ 
  intro!: cInf_lower  $\tau$ _mono elim!: order.trans[rotated] simp del: upt_Suc split: if_splits list.splits)
moreover
have Suc  $i \leq$  progress  $\sigma$  (formula.Until  $\varphi'''$  I  $\psi''$ ) (Suc j)
  if enat t + right I < enat (case nts' of []  $\Rightarrow$   $\tau$   $\sigma$  j | nt # x  $\Rightarrow$  nt)
proof -
  from that obtain m where m: right I = enat m by (cases right I) auto
  have  $\tau$ _min:  $\tau$   $\sigma$  (min j k) = min ( $\tau$   $\sigma$  j) ( $\tau$   $\sigma$  k) for k
    by (simp add: min_of_mono monoI)
  have le_progress_iff[simp]:  $i \leq$  progress  $\sigma$   $\varphi$  i  $\longleftrightarrow$  progress  $\sigma$   $\varphi$  i = i for  $\varphi$  i
    using progress_le[of  $\sigma$   $\varphi$  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  $\sigma$   $\varphi'''$  (Suc j)) (progress  $\sigma$   $\psi''$  (Suc j))  $\longrightarrow$  enat
    ( $\tau$   $\sigma$  k)  $\leq$  enat ( $\tau$   $\sigma$  i) + right I}
  let ?min = min j (min (progress  $\sigma$   $\varphi''$  (Suc j)) (progress  $\sigma$   $\psi''$  (Suc j)))
  have  $\tau$   $\sigma$  ?min  $\leq$   $\tau$   $\sigma$  j
    by (rule  $\tau$ _mono) auto
  from m have ?X  $\neq$  {}
    by (auto dest!:  $\tau$ _mono[of _ progress  $\sigma$   $\varphi''$  (Suc j)  $\sigma$ ]
      simp: not_le not_less  $\varphi'''$  intro!: exI[of _ progress  $\sigma$   $\varphi''$  (Suc j)])
  thm less_le_trans[of  $\tau$   $\sigma$  i + m  $\tau$   $\sigma$  _  $\tau$   $\sigma$  _ + m]
  from m show ?thesis
    using that nts' unfolding wf_ts_def progress.simps
    by (intro cInf_greatest[OF  $\langle$ ?X  $\neq$  {} $\rangle$ ])
      (auto simp: 1  $\varphi'''$  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  $\tau$   $\sigma$  i + m  $\tau$   $\sigma$  _  $\tau$   $\sigma$  _ + m] less_ $\tau$ D)
qed
moreover have *: k < progress  $\sigma$   $\psi$  (Suc j) if
  enat ( $\tau$   $\sigma$  i) + right I < enat (case nts' of []  $\Rightarrow$   $\tau$   $\sigma$  j | nt # x  $\Rightarrow$  nt)
  enat ( $\tau$   $\sigma$  k -  $\tau$   $\sigma$  i)  $\leq$  right I  $\psi$  =  $\psi'' \vee \psi$  =  $\varphi''$  for k  $\psi$ 
proof -
  from that(1,2) obtain m where right I = enat m
     $\tau$   $\sigma$  i + m < (case nts' of []  $\Rightarrow$   $\tau$   $\sigma$  j | nt # x  $\Rightarrow$  nt)  $\tau$   $\sigma$  k -  $\tau$   $\sigma$  i  $\leq$  m
    by (cases right I) auto
  with that(3) nts' progress_le[of  $\sigma$   $\psi''$  Suc j] progress_le[of  $\sigma$   $\varphi''$  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  $\tau$   $\sigma$  k] less_ $\tau$ D)
qed
ultimately show ?case using Cons.premS Suc_i_aux'[simplified]
  unfolding  $\langle$ a = (t, a1, a2) $\rangle$ 
  by (auto simp:  $\varphi'''$  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  $\varphi$ ] MUntil.IH(2)[OF  $\psi$ ] pos_pos_eq fwi_subset show ?case
  by (auto 0 4 simp: Until_eq  $\varphi'''$  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*: $wf_mstate\ \varphi\ \pi\ R\ st \implies last_ts\ \pi \leq snd\ tdb \implies wf_mstate\ \varphi\ (psnoc\ \pi\ tdb)\ R\ (snd\ (mstep\ tdb\ st))$

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 $wf_mstate\ \varphi\ \pi\ R\ st\ last_ts\ \pi \leq snd\ tdb\ prefix_of\ (psnoc\ \pi\ tdb)\ \sigma\ mem_restr\ R\ v$

shows $(i, v) \in fst\ (mstep\ tdb\ st) \iff$

$progress\ \sigma\ \varphi\ (plen\ \pi) \leq i \wedge i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi)) \wedge$

$wf_tuple\ (MFOTL.nfv\ \varphi)\ (MFOTL.fv\ \varphi)\ v \wedge MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi$

proof –

from *prefix_of_psnocE[OF assms(3,2)]* **have** *prefix_of* $\pi\ \sigma$

$\Gamma\ \sigma\ (plen\ \pi) = fst\ tdb\ \tau\ \sigma\ (plen\ \pi) = snd\ tdb$ **by** *auto*

moreover from *assms(1) <prefix_of* $\pi\ \sigma$ **have** *mstate_n st = MFOTL.nfv* φ

mstate_i st = progress $\sigma\ \varphi\ (plen\ \pi)\ wf_mformula\ \sigma\ (plen\ \pi)\ (mstate_n\ st)\ R\ (mstate_m\ st)\ \varphi$

unfolding *wf_mstate_def* **by** *blast+*

moreover from *meval[OF <wf_mformula* $\sigma\ (plen\ \pi)\ (mstate_n\ st)\ R\ (mstate_m\ st)\ \varphi$]

obtain Vs *st'* **where**

meval $(mstate_n\ st)\ (\tau\ \sigma\ (plen\ \pi))\ (\Gamma\ \sigma\ (plen\ \pi))\ (mstate_m\ st) = (Vs, st')$

wf_mformula $\sigma\ (Suc\ (plen\ \pi))\ (mstate_n\ st)\ R\ st'\ \varphi$

list_all2 $(\lambda i. qtable\ (mstate_n\ st)\ (fv\ \varphi)\ (mem_restr\ R)\ (\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi))$

[progress $\sigma\ \varphi\ (plen\ \pi) \leq i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi))$]

 Vs **by** *blast*

moreover from *this assms(4)* **have** *qtable* $(mstate_n\ st)\ (fv\ \varphi)\ (mem_restr\ R)$

$(\lambda v. MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi)\ (Vs!\ (i - progress\ \sigma\ \varphi\ (plen\ \pi)))$

if *progress* $\sigma\ \varphi\ (plen\ \pi) \leq i < progress\ \sigma\ \varphi\ (Suc\ (plen\ \pi))$

using *that* **by** (*auto simp: list_all2_conv_all_nth*

dest!: spec[of _ (i - progress $\sigma\ \varphi\ (plen\ \pi))$])

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 $\sigma\ \varphi\ (plen\ \pi))$])

qed

6.5.5 Monitor function

definition *minit_safe* **where**

minit_safe $\varphi = (if\ mmonitorable_exec\ \varphi\ then\ minit\ \varphi\ else\ undefined)$

lemma *minit_safe_minit*: $mmonitorable\ \varphi \implies minit_safe\ \varphi = minit\ \varphi$

unfolding *minit_safe_def monitorable_formula_code* **by** *simp*

lemma (**in** *monitorable_mfotl*) *mstep_mverdicts*:

assumes *wf*: $wf_mstate\ \varphi\ \pi\ R\ st$

and *le[simp]*: $last_ts\ \pi \leq snd\ tdb$

and *restrict*: $mem_restr\ R\ v$

shows $(i, v) \in fst\ (mstep\ tdb\ st) \iff (i, v) \in M\ (psnoc\ \pi\ tdb) - M\ \pi$

proof –

obtain σ **where** *p2*: *prefix_of* $(psnoc\ \pi\ tdb)\ \sigma$

using *ex_prefix_of* **by** *blast*

with *le* **have** *p1*: *prefix_of* $\pi\ \sigma$ **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 _ \sigma]

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 _  $\pi'$  @  $\pi$ ] 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.prem(1,2) this(2) refl] Suc.prem 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  $\varphi$   $\pi$   $R$   $st$  mem_restr  $R$   $v$   $\pi \leq \pi'$ 
  shows  $(i, v) \in msteps\_stateless$  (pdrop (plen  $\pi$ )  $\pi'$ )  $st \longleftrightarrow (i, v) \in M \pi' - M \pi$ 
  using wf_mstate_msteps[OF assms refl] unfolding msteps_msteps_stateless by simp

lemma (in monitorable_mfotl) wf_mstate_msteps_stateless_UNIV: wf_mstate  $\varphi$   $\pi$  UNIV  $st \implies \pi \leq \pi' \implies$ 
   $msteps\_stateless$  (pdrop (plen  $\pi$ )  $\pi'$ )  $st = M \pi' - M \pi$ 
  by (auto dest: wf_mstate_msteps_stateless[OF _ mem_restr_UNIV])

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

lemma wf_mstate_init_safe: mmonitorable  $\varphi \implies wf\_mstate \varphi \text{pnil} R$  (minit_safe  $\varphi$ )
  using wf_mstate_init minit_safe_minit mmonitorable_def by metis

lemma (in monitorable_mfotl) monitor_mverdicts: monitor  $\varphi$   $\pi = M \pi$ 
  unfolding monitor_def using monitorable
  by (subst wf_mstate_msteps_stateless_UNIV[OF wf_mstate_init_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*: $\pi \leq \pi' \implies M \pi \subseteq M \pi'$
 - *sound_monitor*: $\llbracket (i, v) \in M \pi; \text{prefix_of } \pi \sigma \rrbracket \implies MFOTL.\text{sat } \sigma$ (map the v) $i \varphi$
 - *complete_monitor*: $\llbracket \text{prefix_of } \pi \sigma; wf_tuple (MFOTL.nfv \varphi) (fv \varphi) v; \bigwedge \sigma. \text{prefix_of } \pi \sigma \implies MFOTL.\text{sat } \sigma$ (map the v) $i \varphi \rrbracket \implies \exists \pi'. \text{prefix_of } \pi' \sigma \wedge (i, v) \in M \pi'$
 - *sliceable_M*: $\text{mem_restr } S v \implies ((i, v) \in M (pmap_f (\lambda D. D \cap \text{relevant_events } \varphi S) \pi)) = ((i, v) \in M \pi)$
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_init_safe*: $mmonitorable \varphi' \implies wf_mstate \varphi' \text{pnil} R$ (minit_safe φ')
 - *wf_mstate_mstep*: $\llbracket wf_mstate \varphi' \pi R st; \text{last_ts } \pi \leq \text{snd tdb} \rrbracket \implies wf_mstate \varphi' (psnoc \pi tdb) R (\text{snd } (mstep tdb st))$
 - *mstep_mverdicts*: $\llbracket wf_mstate \varphi \pi R st; \text{last_ts } \pi \leq \text{snd tdb}; \text{mem_restr } R v \rrbracket \implies ((i, v) \in \text{fst } (mstep tdb st)) = ((i, v) \in M (psnoc \pi tdb) - M \pi)$

3. The executable monitor's offline interface *Monitor.monitor* implements *M*:

- *monitor_mverdicts*: *Monitor.monitor* φ $\pi = M$ π

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  $\Rightarrow$  'a prefix  $\Rightarrow$  (nat  $\times$  'b option list) set
  and splitter :: 'a prefix  $\Rightarrow$  'k  $\Rightarrow$  'a prefix
  and joiner :: ('k  $\Rightarrow$  (nat  $\times$  'b option list) set)  $\Rightarrow$  (nat  $\times$  'b option list) set
assumes mono_splitter:  $\pi \leq \pi' \Longrightarrow$  splitter  $\pi$  k  $\leq$  splitter  $\pi'$  k
  and correct_slicer: joiner ( $\lambda$ k. submonitor k (splitter  $\pi$  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  $\lambda$ _. M splitter joiner for nfv fv sat M splitter joiner

```

7.1.3 Definition 3

```

locale event_separable_splitter =
  fixes event_splitter :: 'a  $\Rightarrow$  'k :: finite set
begin

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

```

7.1.4 Lemma 1

```

lemma mono_splitter:  $\pi \leq \pi' \Longrightarrow$  splitter  $\pi$  k  $\leq$  splitter  $\pi'$  k
  by transfer auto

```

end

7.2 Joint data slicer

```

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

```

```

locale splitting_strategy =

```



```

fixes  $\varphi :: 'a \text{ MFOTL.formula}$ 
and  $\text{strategy} :: 'a \text{ option list} \Rightarrow 'k :: \text{finite set}$ 
assumes  $\text{strategy\_nonempty}: \text{ok } \varphi \ v \implies \text{strategy } v \neq \{\}$ 
begin

abbreviation  $\text{slice\_set}$  where
   $\text{slice\_set } k \equiv \{v. \exists v'. \text{map the } v' = v \wedge \text{ok } \varphi \ v' \wedge k \in \text{strategy } v'\}$ 

end

```

7.2.1 Definition 4

```

locale  $\text{MFOTL\_monitor} =$ 
   $\text{monitor MFOTL.nfv } \varphi \ \text{MFOTL.fv } \varphi \ \lambda\sigma \ v \ i. \ \text{MFOTL.sat } \sigma \ v \ i \ \varphi \ M \ \text{for } \varphi \ M$ 

locale  $\text{joint\_data\_slicer} = \text{MFOTL\_monitor } \varphi \ M + \text{splitting\_strategy } \varphi \ \text{strategy}$ 
  for  $\varphi \ M \ \text{strategy}$ 
begin

definition  $\text{event\_splitter}$  where
   $\text{event\_splitter } e = (\bigcup (\text{strategy } ' \{v. \text{ok } \varphi \ v \wedge \text{MFOTL.matches } (\text{map the } v) \ \varphi \ e\}))$ 

sublocale  $\text{event\_separable\_splitter}$  where  $\text{event\_splitter} = \text{event\_splitter} .$ 

```

```

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

```

```

lemma  $\text{splitter\_pslice}: \text{splitter } \pi \ k = \text{MFOTL\_slicer.pslic} \ \varphi \ (\text{slice\_set } k) \ \pi$ 
by  $\text{transfer } (\text{auto simp: event\_splitter\_def})$ 

```

7.2.2 Lemma 2

Corresponds to the following theorem sat_slice_strong proved in theory $\text{MFOTL_Monitor.Abstract_Monitor}$:

$$\llbracket \text{relevant_events } \varphi' \ S \subseteq E; v \in S \rrbracket \implies \text{MFOTL.sat } \sigma \ v \ i \ \varphi' = \text{MFOTL.sat } (\text{map_}\Gamma \ (\lambda D. D \cap E) \ \sigma) \ v \ i \ \varphi'$$

7.2.3 Theorem 1

```

sublocale  $\text{joint\_monitor}: \text{MFOTL\_monitor } \varphi \ \lambda\pi. \ \text{joiner } (\lambda k. M \ (\text{splitter } \pi \ k))$ 
proof  $(\text{unfold\_locales}, \text{goal\_cases } \text{mono wf sound complete})$ 
  case  $(\text{mono } \pi \ \pi')$ 
  show  $?case$ 
  using  $\text{mono\_monitor}[OF \ \text{mono\_splitter}, OF \ \text{mono}]$ 
  by  $(\text{auto simp: joiner\_def})$ 
next
  case  $(\text{wf } i \ v \ \pi)$ 
  then obtain  $k$  where  $\text{in\_}M: (i, v) \in M \ (\text{splitter } \pi \ k)$  and  $k: k \in \text{strategy } v$ 
  unfolding  $\text{joiner\_def}$  by  $(\text{auto split: if\_splits})$ 
  then show  $?case$ 
  using  $\text{wf\_monitor}[OF \ \text{in\_}M]$  by  $\text{auto}$ 
next
  case  $(\text{sound } i \ v \ \pi \ \sigma)$ 
  then obtain  $k$  where  $\text{in\_}M: (i, v) \in M \ (\text{splitter } \pi \ k)$  and  $k: k \in \text{strategy } v$ 
  unfolding  $\text{joiner\_def}$  by  $(\text{auto split: if\_splits})$ 
  have  $\text{wf}: \text{ok } \varphi \ v$  and  $\text{sat}: \bigwedge \sigma. \text{prefix\_of } (\text{splitter } \pi \ k) \ \sigma \implies \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi$ 
  using  $\text{sound\_monitor}[OF \ \text{in\_}M] \ \text{wf\_monitor}[OF \ \text{in\_}M]$  by  $\text{auto}$ 
  then have  $\text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi$  if  $\text{prefix\_of } \pi \ \sigma$  for  $\sigma$ 
  using  $\text{that } k$ 

```

```

  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 ∈ strategy v by blast
  then have (i, v) ∈ 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 ∪ 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 ∪ names β
| names (MFOTL.Until α I β) = names α ∪ 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 ∧ gen_unique β ∧ names α ∩ 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 ∧ gen_unique β ∧ names α ∩ names β = {})
| gen_unique (MFOTL.Until α I β) = (gen_unique α ∧ gen_unique β ∧ names α ∩ names β = {})

```

lemma sat_inter_names_cong: $(\bigwedge e. e \in \text{names } \varphi \implies \{xs. (e, xs) \in E\} = \{xs. (e, xs) \in F\}) \implies$
 $\text{MFOTL.sat } (\text{map_}\Gamma (\lambda D. D \cap E) \sigma) v i \varphi \longleftrightarrow \text{MFOTL.sat } (\text{map_}\Gamma (\lambda D. D \cap F) \sigma) v i \varphi$
by (induction φ arbitrary: v i) (auto split: nat.splits)

lemma matches_in_names: $\text{MFOTL.matches } v \varphi x \implies \text{fst } x \in \text{names } \varphi$
by (induction φ arbitrary: v) (auto)

lemma unique_names_matches_absorb: $\text{fst } x \in \text{names } \alpha \implies \text{names } \alpha \cap \text{names } \beta = \{\} \implies$
 $\text{MFOTL.matches } v \alpha x \vee \text{MFOTL.matches } v \beta x \longleftrightarrow \text{MFOTL.matches } v \alpha x$
 $\text{fst } x \in \text{names } \beta \implies \text{names } \alpha \cap \text{names } \beta = \{\} \implies$
 $\text{MFOTL.matches } v \alpha x \vee \text{MFOTL.matches } v \beta x \longleftrightarrow \text{MFOTL.matches } v \beta x$
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  $\bigwedge v1 v2 v. v1 \in S \implies v2 \in S \implies \text{length } v = n \implies \forall x < n. v ! x = v1 ! x \vee v ! x = v2 ! x$ 
 $\implies v \in S$ 

```

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

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

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

lemma *mergeable_envs_listset*: *mergeable_envs* ($\text{length } As$) (*listset* As)
by (*rule mergeable_envsI*) (*auto intro!*: *all_nth_in_listset elim!*: *in_listset_nth*)

lemma *mergeable_envs_Ext*: *mergeable_envs* n $S \implies \text{MFOTL.nfv } \alpha \leq n \implies \text{MFOTL.nfv } \beta \leq n \implies$
 $(\exists v' \in S. \forall x \in \text{fv } \alpha. v' !\ x = v !\ x) \implies (\exists v' \in S. \forall x \in \text{fv } \beta. v' !\ x = v !\ x) \implies$
 $(\exists v' \in S. \forall x \in \text{fv } \alpha \cup \text{fv } \beta. v' !\ x = v !\ x)$
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*: $xs \in \text{set_Cons } A\ As \implies (\bigwedge y\ ys. xs = y \# ys \implies y \in A \implies ys \in As \implies P) \implies P$
unfolding *set_Cons_def* **by** *blast*

lemma *mergeable_envs_set_Cons*: *mergeable_envs* n $S \implies \text{mergeable_envs } (\text{Suc } n)$ (*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 $?A = (\lambda x. x - 1) \text{ ` } (A - \{0\})$
let $?B = (\lambda x. x - 1) \text{ ` } (B - \{0\})$
from *mergeable*($4-9$) **have** $\exists v \in S. \forall x \in ?A \cup ?B. v !\ x = f (\text{Suc } x)$
by (*auto dest!*: *mergeable*($2,3$)[*rule_format*] *intro!*: *mergeable*(1)[*rule_format*, *of* $ys1$ $ys2$])
then obtain v **where** $v \in S \forall x \in ?A \cup ?B. v !\ x = f (\text{Suc } x)$ **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 $\sigma = \text{MFOTL_slicer.slice } \varphi$ (*set_Cons UNIV* S) σ
by (*auto simp*: *set_Cons_def intro*: *map_Γ_cong*)

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

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

```

lemma set_Cons_empty_iff[simp]: set_Cons A Xs = {}  $\longleftrightarrow$  A = {}  $\vee$  Xs = {}
  unfolding set_Cons_def by auto

lemma unique_sat_slice_mem: safe_formula  $\varphi \implies$  gen_unique  $\varphi \implies S \neq \{\} \implies$ 
  mergeable_envs n S  $\implies$  MFOTL.nfv  $\varphi \leq n \implies$ 
  MFOTL.sat (MFOTL.slicer.slice  $\varphi$  S  $\sigma$ ) v i  $\varphi \implies \exists v' \in S. \forall x \in fv \varphi. v' ! x = v ! x$ 
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'  $\in S$  and eq:  $\forall t \in set \ ts. MFOTL.eval\_trm \ v' \ t = MFOTL.eval\_trm \ v \ t$ 
  by auto
  have  $\forall t \in set \ ts. \forall x \in fv\_trm \ t. v' ! x = v ! x$  proof
    fix t assume t  $\in set \ ts$ 
    with eq have MFOTL.eval_trm v' t = MFOTL.eval_trm v t ..
    then show  $\forall x \in fv\_trm \ t. v' ! x = v ! x$  by (cases t) (simp_all)
  qed
  with  $\langle v' \in S \rangle$  show ?case by auto
next
  case (6  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.And \ \varphi \ \psi) \rangle$ 
  have
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.And  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.And  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\psi = MFOTL.sat (MFOTL.slicer.slice
 $\psi$  S  $\sigma$ ) v i  $\psi$ 
    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  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.And\_Not \ \varphi \ \psi) \rangle$ 
  have MFOTL.sat (MFOTL.slicer.slice (MFOTL.And\_Not  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    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  $\exists v' \in S. \forall x \in fv \ \varphi. v' ! x = v ! x$ 
    unfolding MFOTL.And\_Not_def by auto
  with  $\langle fv \ \psi \subseteq fv \ \varphi \rangle$  show ?case by (auto simp: MFOTL.fvi_And_Not)
next
  case (8  $\varphi \ \psi$ )
  from  $\langle gen\_unique \ (MFOTL.Or \ \varphi \ \psi) \rangle$ 
  have
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.Or  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\varphi = MFOTL.sat (MFOTL.slicer.slice
 $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
    MFOTL.sat (MFOTL.slicer.slice (MFOTL.Or  $\varphi \ \psi$ ) S  $\sigma$ ) v i  $\psi = 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 ∃ v'∈S. ∀ x∈fv φ. 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.prem1 sat_φ have ∃ v'∈set_Cons UNIV S. ∀ x∈fv φ. 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 ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv φ ⊆ 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 ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv (MFOTL.Neg φ) ⊆ 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 ∃ v'∈S. ∀ x∈fv ψ. v' ! x = v ! x using * by auto
with ⟨fv φ ⊆ fv ψ⟩ show ?case by auto

```

```

next
  case (15  $\varphi$  I  $\psi$ )
  from  $\langle \text{gen\_unique } (MFOTL.Until (MFOTL.Neg \varphi) I \psi) \rangle$ 
  have *:
    MFOTL.sat (MFOTL_slicer.slice (MFOTL.Until (MFOTL.Neg  $\varphi$ ) I  $\psi$ ) S  $\sigma$ ) v j  $\psi$  = MFOTL.sat
    (MFOTL_slicer.slice  $\psi$  S  $\sigma$ ) v j  $\psi$  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  $\varphi$ ) I  $\psi$ ) S
   $\sigma$ ) v j  $\psi$ 
  by auto
  with 15 have  $\exists v' \in S. \forall x \in fv \psi. v' ! x = v ! x$  using * by auto
  with  $\langle fv (MFOTL.Neg \varphi) \subseteq fv \psi \rangle$  show ?case by auto
qed

```

```

lemma unique_sat_slice:
  assumes formula: safe_formula  $\varphi$  gen_unique  $\varphi$ 
  and restr:  $S \neq \{\}$  mergeable_envs (MFOTL.nfv  $\varphi$ ) S
  and sat_slice: MFOTL.sat (MFOTL_slicer.slice  $\varphi$  S  $\sigma$ ) v i  $\varphi$ 
  shows MFOTL.sat  $\sigma$  v i  $\varphi$ 
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  $\varphi$  S  $\sigma$ ) v' i  $\varphi$ 
  by (auto iff: sat_fvi_cong)
  then have MFOTL.sat  $\sigma$  v' i  $\varphi$ 
  unfolding sat_slice_iff[OF  $\langle v' \in S \rangle$ , 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  $\varphi$   $\implies$  gen_unique  $\varphi$   $\implies$  slice_set k  $\neq \{\}$   $\implies$ 
  mergeable_envs (MFOTL.nfv  $\varphi$ ) (slice_set k)  $\implies$ 
  MFOTL.sat (MFOTL_slicer.slice  $\varphi$  (slice_set k)  $\sigma$ ) (map the v) i  $\varphi$   $\implies$ 
  ok  $\varphi$  v  $\implies$  k  $\in$  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  $\neq \{\}$ 
  and mergeable: mergeable_envs (MFOTL.nfv  $\varphi$ ) (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  $\varphi$  gen_unique  $\varphi$ 
  shows joiner ( $\lambda k. M$  (splitter  $\pi$  k)) = skip_joiner ( $\lambda k. M$  (splitter  $\pi$  k))
  (is ?L = ?R)
proof safe
  fix i v
  assume (i, v)  $\in$  ?R
  then obtain k where in_M: (i, v)  $\in$  M (splitter  $\pi$  k)
  unfolding skip_joiner_def by blast
  from ex_prefix_of obtain  $\sigma$  where prefix_of  $\pi$   $\sigma$  by blast

```

```

with wf_monitor[OF in_M] sound_monitor[OF in_M] have
  MFOTL.sat (MFOTL slicer.slice  $\varphi$  (slice_set k)  $\sigma$ ) (map the v) i  $\varphi$  ok  $\varphi$  v
by (auto simp: splitter_pslice intro!: prefix_of_pmap_ $\Gamma$ )
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  $\varphi$ 
   $\lambda\pi.$  (if safe_formula  $\varphi \wedge$  gen_unique  $\varphi$  then skip_joiner else joiner) ( $\lambda k. M$  (splitter  $\pi$  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/traytel/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.