

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

Joshua Schneider Dmitriy Traytel

December 14, 2021

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	6
3	Abstract monitors and slicing	11
3.1	First-order specifications	11
3.2	Monitor function	12
3.3	Slicing	13
4	Intervals	14
5	Metric first-order temporal logic	15
5.1	Formulas and satisfiability	15
5.2	Defined connectives	17
5.3	Safe formulas	18
5.4	Slicing traces	19
6	Monitor implementation	19
6.1	Monitorable formulas	19
6.2	The executable monitor	20
6.3	Progress	23
6.4	Specification	24
6.5	Correctness	24
6.5.1	Invariants	24
6.5.2	Initialisation	26
6.5.3	Evaluation	27

6.5.4	Monitor step	31
6.5.5	Monitor function	31
6.6	Collected correctness results	32
7	Slicing framework	33
7.1	Abstract slicing	33
7.1.1	Definition 1	33
7.1.2	Definition 2	33
7.1.3	Definition 3	33
7.1.4	Lemma 1	34
7.2	Joint data slicer	34
7.2.1	Definition 4	34
7.2.2	Lemma 2	34
7.2.3	Theorem 1	34
7.2.4	Corollary 1	34
7.2.5	Definition 5	35
7.2.6	Theorem 2	35
7.2.7	Towards Theorem 3	35
7.2.8	Lemma 3	37
7.2.9	Definition of J'	37
7.2.10	Theorem 3	37

1 Traces and trace prefixes

1.1 Infinite traces

coinductive *sorted* :: 'a :: linorder stream \Rightarrow bool **where**
shd s \leq *shd* (stl s) \Longrightarrow sorted (stl s) \Longrightarrow sorted s

lemma *sorted_siterate*[simp]: $(\bigwedge n. n \leq f n) \Longrightarrow$ sorted (siterate f n)
 <proof>

lemma *sortedD*: sorted s \Longrightarrow s !! i \leq stl s !! i
 <proof>

lemma *sorted_sdrop*: sorted s \Longrightarrow sorted (sdrop i s)
 <proof>

lemma *sorted_monoD*: sorted s \Longrightarrow i \leq j \Longrightarrow s !! i \leq s !! j
 <proof>

lemma *sorted_stake*: sorted s \Longrightarrow sorted (stake i s)
 <proof>

lemma *sorted_monoI*: $\forall i j. i \leq j \longrightarrow$ s !! i \leq s !! j \Longrightarrow sorted s
 <proof>

lemma *sorted_iff_mono*: sorted s \longleftrightarrow $(\forall i j. i \leq j \longrightarrow$ s !! i \leq s !! j)
 <proof>

lemma *sorted_iff_le_Suc*: sorted s \longleftrightarrow $(\forall i. s !! i \leq s !! \text{Suc } i)$
 <proof>

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

lemma *sincreasingI*: $(\bigwedge x. \exists i. x < s !! i) \Longrightarrow$ increasing s

<proof>

lemma *sincreasing_grD*:

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

assumes *sincreasing s*

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

<proof>

lemma *sincreasing_siterate_nat[simp]*:

fixes $n :: \text{nat}$

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

shows *sincreasing (siterate f n)*

<proof>

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

<proof>

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

<proof>

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow 'a \text{ set}$ **is**

$\lambda s i. \text{fst } (s !! i)$ *<proof>*

lift_definition $\tau :: 'a \text{ trace} \Rightarrow \text{nat} \Rightarrow \text{nat}$ **is**

$\lambda s i. \text{snd } (s !! i)$ *<proof>*

lemma *stream_eq_iff*: $s = s' \iff (\forall n. s !! n = s' !! n)$

<proof>

lemma *trace_eqI*: $(\bigwedge i. \Gamma \sigma i = \Gamma \sigma' i) \implies (\bigwedge i. \tau \sigma i = \tau \sigma' i) \implies \sigma = \sigma'$

<proof>

lemma *τ _mono[simp]*: $i \leq j \implies \tau s i \leq \tau s j$

<proof>

lemma *ex_le_ τ* : $\exists j \geq i. x \leq \tau s j$

<proof>

lemma *le_ τ _less*: $\tau \sigma i \leq \tau \sigma j \implies j < i \implies \tau \sigma i = \tau \sigma j$

<proof>

lemma *less_ τ D*: $\tau \sigma i < \tau \sigma j \implies i < j$

<proof>

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**

$\lambda f s. \text{smap } (\lambda(x, i). (f x, i)) s$

<proof>

lemma *Γ _map_ Γ [simp]*: $\Gamma (\text{map}_\Gamma f s) i = f (\Gamma s i)$

<proof>

lemma *τ _map_ Γ [simp]*: $\tau (\text{map}_\Gamma f s) i = \tau s i$

<proof>

lemma *map_ Γ _id[simp]*: *map_ Γ id s = s*

<proof>

lemma *map_Γ_comp*: $\text{map}_\Gamma g (\text{map}_\Gamma f s) = \text{map}_\Gamma (g \circ f) s$
<proof>

lemma *map_Γ_cong*: $\sigma_1 = \sigma_2 \implies (\bigwedge x. f_1 x = f_2 x) \implies \text{map}_\Gamma f_1 \sigma_1 = \text{map}_\Gamma f_2 \sigma_2$
<proof>

1.2 Finite trace prefixes

typedef *'a prefix* = $\{p :: ('a \text{ set} \times \text{nat}) \text{ list. sorted (map snd } p)\}$
<proof>

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**
 $\lambda f. \text{map } (\lambda(x, i). (f x, i))$
<proof>

lift_definition *last_ts* :: $'a \text{ prefix} \Rightarrow \text{nat}$ **is**
 $\lambda p. (\text{case } p \text{ of } [] \Rightarrow 0 \mid _ \Rightarrow \text{snd (last } p))$ *<proof>*

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

lift_definition *pnil* :: $'a \text{ prefix}$ **is** $[]$ *<proof>*

lift_definition *plen* :: $'a \text{ prefix} \Rightarrow \text{nat}$ **is** *length* *<proof>*

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

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

lift_definition *less_eq_prefix* :: $'a \text{ prefix} \Rightarrow 'a \text{ prefix} \Rightarrow \text{bool}$ **is**
 $\lambda p q. \exists r. q = p @ r$ *<proof>*

definition *less_prefix* :: $'a \text{ prefix} \Rightarrow 'a \text{ prefix} \Rightarrow \text{bool}$ **where**
 $\text{less_prefix } x y = (x \leq y \wedge \neg y \leq x)$

instance
<proof>

end

lemma *psnoc_inject[simp]*:
 $\text{last_ts } p \leq \text{snd } x \implies \text{last_ts } q \leq \text{snd } y \implies \text{psnoc } p x = \text{psnoc } q y \iff (p = q \wedge x = y)$
<proof>

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

lemma *prefix_of_pnil[simp]*: *prefix_of* *pnil* σ
<proof>

lemma *plen_pnil[simp]*: *plen* *pnil* = 0
<proof>

lemma *prefix_of_pmap_Γ[simp]*: $\text{prefix_of } \pi \sigma \implies \text{prefix_of } (\text{pmap_}\Gamma f \pi) (\text{map_}\Gamma f \sigma)$
 ⟨proof⟩

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

lemma *prefix_of_psnocE*: $\text{prefix_of } (\text{psnoc } p x) s \implies \text{last_ts } p \leq \text{snd } x \implies$
 $(\text{prefix_of } p s \implies \Gamma s (\text{plen } p) = \text{fst } x \implies \tau s (\text{plen } p) = \text{snd } x \implies P) \implies P$
 ⟨proof⟩

lemma *le_pnil[simp]*: $\text{pnil} \leq \pi$
 ⟨proof⟩

lift_definition *take_prefix* :: $\text{nat} \Rightarrow 'a \text{ trace} \Rightarrow 'a \text{ prefix is stake}$
 ⟨proof⟩

lemma *plen_take_prefix[simp]*: $\text{plen } (\text{take_prefix } i \sigma) = i$
 ⟨proof⟩

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

lemma *prefix_of_take_prefix[simp]*: $\text{prefix_of } (\text{take_prefix } i \sigma) \sigma$
 ⟨proof⟩

lift_definition *pdrop* :: $\text{nat} \Rightarrow 'a \text{ prefix} \Rightarrow 'a \text{ prefix is drop}$
 ⟨proof⟩

lemma *pdrop_0[simp]*: $\text{pdrop } 0 \pi = \pi$
 ⟨proof⟩

lemma *prefix_of_antimono*: $\pi \leq \pi' \implies \text{prefix_of } \pi' s \implies \text{prefix_of } \pi s$
 ⟨proof⟩

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⟩

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

lemma *τ_prefix_conv*: $\text{prefix_of } p s \implies \text{prefix_of } p s' \implies i < \text{plen } p \implies \tau s i = \tau s' i$
 ⟨proof⟩

lemma *Γ_prefix_conv*: $\text{prefix_of } p s \implies \text{prefix_of } p s' \implies i < \text{plen } p \implies \Gamma s i = \Gamma s' i$
 ⟨proof⟩

lemma *sincreasing_sdrop*:
fixes $s :: ('a :: \text{semilattice_sup}) \text{ stream}$
assumes *sincreasing s*
shows *sincreasing (sdrop n s)*
 ⟨proof⟩

lemma *ssorted_shift*:
 $\text{ssorted } (xs @- s) = (\text{sorted } xs \wedge \text{ssorted } s \wedge (\forall x \in \text{set } xs. \forall y \in \text{sset } s. x \leq y))$
 ⟨proof⟩

lemma *sincreasing_shift*:
assumes *sincreasing s*

shows *sincreasing* ($xs @- s$)
 ⟨*proof*⟩

lift_definition *replace_prefix* :: 'a prefix \Rightarrow 'a trace \Rightarrow 'a trace **is**
 $\lambda \pi \sigma.$ if *sorted* ($smap\ snd (\pi @- sdrop (length\ \pi)\ \sigma)$) then
 $\pi @- sdrop (length\ \pi)\ \sigma$ else $smap (\lambda i. (\{\}, i))\ nats$
 ⟨*proof*⟩

lemma *prefix_of_replace_prefix*:
 $prefix_of (pmap_ \Gamma f \pi)\ \sigma \implies prefix_of\ \pi (replace_prefix\ \pi\ \sigma)$
 ⟨*proof*⟩

lemma *map_Γ_replace_prefix*:
 $\forall x. f (f x) = f x \implies prefix_of (pmap_ \Gamma f \pi)\ \sigma \implies map_ \Gamma f (replace_prefix\ \pi\ \sigma) = map_ \Gamma f \sigma$
 ⟨*proof*⟩

lemma *prefix_of_pmap_Γ_D*:
assumes $prefix_of (pmap_ \Gamma f \pi)\ \sigma$
shows $\exists \sigma'. prefix_of\ \pi\ \sigma' \wedge prefix_of (pmap_ \Gamma f \pi) (map_ \Gamma f \sigma')$
 ⟨*proof*⟩

lemma *prefix_of_map_Γ_D*:
assumes $prefix_of\ \pi' (map_ \Gamma f \sigma)$
shows $\exists \pi''. \pi' = pmap_ \Gamma f \pi'' \wedge prefix_of\ \pi'' \sigma$
 ⟨*proof*⟩

lift_definition *pts* :: 'a prefix \Rightarrow nat list **is** $map\ snd$ ⟨*proof*⟩

lemma *pts_pmap_Γ[simp]*: $pts (pmap_ \Gamma f \pi) = pts\ \pi$
 ⟨*proof*⟩

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*: $tabulate\ f\ x\ n = map\ f [x ..< x + n]$
 ⟨*proof*⟩

lemma *length_tabulate[simp]*: $length (tabulate\ f\ x\ n) = n$
 ⟨*proof*⟩

lemma *map_tabulate[simp]*: $map\ f (tabulate\ g\ x\ n) = tabulate (\lambda x. f (g\ x))\ x\ n$
 ⟨*proof*⟩

lemma *nth_tabulate[simp]*: $k < n \implies tabulate\ f\ x\ n ! k = f (x + k)$
 ⟨*proof*⟩

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

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

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

definition *empty_table* = {}

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

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

lemma *in_empty_table[simp]*: $\neg x \in \text{empty_table}$
<proof>

lemma *empty_table[simp]*: *table* *n* *V* *empty_table*
<proof>

lemma *unit_table_wf_tuple[simp]*: $V = \{\}$ $\implies x \in \text{unit_table } n \implies \text{wf_tuple } n \ V \ x$
<proof>

lemma *unit_table[simp]*: $V = \{\}$ $\implies \text{table } n \ V \ (\text{unit_table } n)$
<proof>

lemma *in_unit_table*: $v \in \text{unit_table } n \iff \text{wf_tuple } n \ \{\} \ v$
<proof>

lemma *singleton_table_wf_tuple[simp]*: $V = \{i\} \implies x \in \text{singleton_table } n \ i \ z \implies \text{wf_tuple } n \ V \ x$
<proof>

lemma *singleton_table[simp]*: $V = \{i\} \implies \text{table } n \ V \ (\text{singleton_table } n \ i \ z)$
<proof>

lemma *table_Un[simp]*: *table* *n* *V* *X* $\implies \text{table } n \ V \ Y \implies \text{table } n \ V \ (X \cup Y)$
<proof>

lemma *wf_tuple_length*: $\text{wf_tuple } n \ V \ x \implies \text{length } x = n$
<proof>

fun *join1* :: 'a tuple \times 'a tuple \Rightarrow '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 \Rightarrow bool \Rightarrow 'a table \Rightarrow 'a table **where**

join *A* *pos* *B* = (if *pos* then Option.these (*join1* ' (*A* \times *B*))
else *A* - Option.these (*join1* ' (*A* \times *B*)))

lemma *join_True_code[code]*: *join* *A* True *B* = ($\bigcup a \in A. \bigcup b \in B. \text{set_option } (\text{join1 } (a, b))$)
<proof>

lemma *join_False_alt*: *join* *X* False *Y* = *X* - *join* *X* True *Y*
<proof>

lemma *self_join1*: *join1* (*xs*, *ys*) \neq Some *xs* $\implies \text{join1 } (zs, ys) \neq$ Some *xs*
<proof>

lemma *join_False_code[code]*: *join* *A* False *B* = { $a \in A. \forall b \in B. \text{join1 } (a, b) \neq$ Some *a*}

<proof>

lemma *wf_tuple_Nil[simp]*: $wf_tuple\ n\ A\ [] = (n = 0)$
<proof>

lemma *Suc_pred'*: $Suc\ (x - Suc\ 0) = (case\ x\ of\ 0 \Rightarrow Suc\ 0\ |\ _ \Rightarrow x)$
<proof>

lemma *wf_tuple_Cons[simp]*:
 $wf_tuple\ n\ A\ (x\ \#\ xs) \longleftrightarrow ((if\ x = None\ then\ 0 \notin A\ else\ 0 \in A) \wedge$
 $(\exists\ m.\ n = Suc\ m \wedge wf_tuple\ m\ ((\lambda x.\ x - 1)\ ' (A - \{0\}))\ xs))$
<proof>

lemma *join1_wf_tuple*:
 $join1\ (v1,\ v2) = Some\ v \Longrightarrow wf_tuple\ n\ A\ v1 \Longrightarrow wf_tuple\ n\ B\ v2 \Longrightarrow wf_tuple\ n\ (A \cup B)\ v$
<proof>

lemma *join_wf_tuple*: $x \in join\ X\ b\ Y \Longrightarrow$
 $\forall v \in X.\ wf_tuple\ n\ A\ v \Longrightarrow \forall v \in Y.\ wf_tuple\ n\ B\ v \Longrightarrow (\neg b \Longrightarrow B \subseteq A) \Longrightarrow A \cup B = C \Longrightarrow$
 $wf_tuple\ n\ C\ x$
<proof>

lemma *join_table*: $table\ n\ A\ X \Longrightarrow table\ n\ B\ Y \Longrightarrow (\neg b \Longrightarrow B \subseteq A) \Longrightarrow A \cup B = C \Longrightarrow$
 $table\ n\ C\ (join\ X\ b\ Y)$
<proof>

lemma *wf_tuple_Suc*: $wf_tuple\ (Suc\ m)\ A\ a \longleftrightarrow a \neq [] \wedge$
 $wf_tuple\ m\ ((\lambda x.\ x - 1)\ ' (A - \{0\}))\ (tl\ a) \wedge (0 \in A \longleftrightarrow hd\ a \neq None)$
<proof>

lemma *table_project*: $table\ (Suc\ n)\ A\ X \Longrightarrow table\ n\ ((\lambda x.\ x - Suc\ 0)\ ' (A - \{0\}))\ (tl\ ' X)$
<proof>

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]*: $restrict\ A\ [] = []$
<proof>

lemma *restrict_Cons[simp]*: $restrict\ A\ (x\ \#\ xs) =$
 $(if\ 0 \in A\ then\ x\ \#\ restrict\ ((\lambda x.\ x - 1)\ ' (A - \{0\}))\ xs\ else\ None\ \#\ restrict\ ((\lambda x.\ x - 1)\ ' A)\ xs)$
<proof>

lemma *wf_tuple_restrict*: $wf_tuple\ n\ B\ v \Longrightarrow A \cap B = C \Longrightarrow wf_tuple\ n\ C\ (restrict\ A\ v)$
<proof>

lemma *wf_tuple_restrict_simple*: $wf_tuple\ n\ B\ v \Longrightarrow A \subseteq B \Longrightarrow wf_tuple\ n\ A\ (restrict\ A\ v)$
<proof>

lemma *nth_restrict*: $i \in A \Longrightarrow i < length\ v \Longrightarrow restrict\ A\ v\ !\ i = v\ !\ i$
<proof>

lemma *restrict_eq_Nil[simp]*: $restrict\ A\ v = [] \longleftrightarrow v = []$
<proof>

lemma *length_restrict[simp]*: $length\ (restrict\ A\ v) = length\ v$
<proof>

lemma *join1_Some_restrict*:

fixes $x\ y :: 'a\ tuple$

assumes $wf_tuple\ n\ A\ x\ wf_tuple\ n\ B\ y$

shows $join1\ (x,\ y) = Some\ z \longleftrightarrow wf_tuple\ n\ (A \cup B)\ z \wedge restrict\ A\ z = x \wedge restrict\ B\ z = y$

<proof>

lemma *restrict_idle*: $wf_tuple\ n\ A\ v \Longrightarrow restrict\ A\ v = v$

<proof>

lemma *map_the_restrict*:

$i \in A \Longrightarrow map\ the\ (restrict\ A\ v)\ !\ i = map\ the\ v\ !\ i$

<proof>

lemma *join_restrict*:

fixes $X\ Y :: 'a\ tuple\ set$

assumes $\bigwedge v. v \in X \Longrightarrow wf_tuple\ n\ A\ v \wedge \bigwedge v. v \in Y \Longrightarrow wf_tuple\ n\ B\ v \neg b \Longrightarrow B \subseteq A$

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

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

<proof>

lemma *join_restrict_table*:

assumes $table\ n\ A\ X\ table\ n\ B\ Y \neg b \Longrightarrow B \subseteq A$

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

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

<proof>

lemma *join_restrict_annotated*:

fixes $X\ Y :: 'a\ tuple\ set$

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

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

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

<proof>

lemma *in_joinI*: $table\ n\ A\ X \Longrightarrow table\ n\ B\ Y \Longrightarrow (\neg b \Longrightarrow B \subseteq A) \Longrightarrow wf_tuple\ n\ (A \cup B)\ v \Longrightarrow$

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

<proof>

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

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

$P) \Longrightarrow P$

<proof>

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

<proof>

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

<proof>

lemma *qtableI*: $table\ n\ A\ X \Longrightarrow$

$(\bigwedge x. x \in X \Longrightarrow wf_tuple\ n\ A\ x \Longrightarrow P\ x \Longrightarrow Q\ x) \Longrightarrow$

$(\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \implies x \in X) \implies$
 $\text{qtable } n \ A \ P \ Q \ X$
 $\langle \text{proof} \rangle$

lemma *in_qtableI*: $\text{qtable } n \ A \ P \ Q \ X \implies \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \implies x \in X$
 $\langle \text{proof} \rangle$

lemma *in_qtableE*: $\text{qtable } n \ A \ P \ Q \ X \implies x \in X \implies P \ x \implies (\text{wf_tuple } n \ A \ x \implies Q \ x \implies R) \implies R$
 $\langle \text{proof} \rangle$

lemma *qtable_empty*: $(\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \implies \text{False}) \implies \text{qtable } n \ A \ P \ Q \ \text{empty_table}$
 $\langle \text{proof} \rangle$

lemma *qtable_empty_iff*: $\text{qtable } n \ A \ P \ Q \ \text{empty_table} = (\forall x. \text{wf_tuple } n \ A \ x \longrightarrow P \ x \longrightarrow Q \ x \longrightarrow \text{False})$
 $\langle \text{proof} \rangle$

lemma *qtable_unit_table*: $(\bigwedge x. \text{wf_tuple } n \ \{\} \ x \implies P \ x \implies Q \ x) \implies \text{qtable } n \ \{\} \ P \ Q \ (\text{unit_table } n)$
 $\langle \text{proof} \rangle$

lemma *qtable_union*: $\text{qtable } n \ A \ P \ Q1 \ X \implies \text{qtable } n \ A \ P \ Q2 \ Y \implies$
 $(\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \longleftrightarrow Q1 \ x \vee Q2 \ x) \implies \text{qtable } n \ A \ P \ Q \ (X \cup Y)$
 $\langle \text{proof} \rangle$

lemma *qtable_Union*: $\text{finite } I \implies (\bigwedge i. i \in I \implies \text{qtable } n \ A \ P \ (Qi \ i) \ (Xi \ i)) \implies$
 $(\bigwedge x. \text{wf_tuple } n \ A \ x \implies P \ x \implies Q \ x \longleftrightarrow (\exists i \in I. Qi \ i \ x)) \implies \text{qtable } n \ A \ P \ Q \ (\bigcup i \in I. Xi \ i)$
 $\langle \text{proof} \rangle$

lemma *qtable_join*:

assumes $\text{qtable } n \ A \ P \ Q1 \ X \ \text{qtable } n \ B \ P \ Q2 \ Y \ \neg b \implies B \subseteq A \ C = A \cup B$
 $\bigwedge x. \text{wf_tuple } n \ C \ x \implies P \ x \implies P \ (\text{restrict } A \ x) \wedge P \ (\text{restrict } B \ x)$
 $\bigwedge x. b \implies \text{wf_tuple } n \ C \ x \implies P \ x \implies Q \ x \longleftrightarrow Q1 \ (\text{restrict } A \ x) \wedge Q2 \ (\text{restrict } B \ x)$
 $\bigwedge x. \neg b \implies \text{wf_tuple } n \ C \ x \implies P \ x \implies Q \ x \longleftrightarrow Q1 \ (\text{restrict } A \ x) \wedge \neg Q2 \ (\text{restrict } B \ x)$
shows $\text{qtable } n \ C \ P \ Q \ (\text{join } X \ b \ Y)$

$\langle \text{proof} \rangle$

lemma *qtable_join_fixed*:

assumes $\text{qtable } n \ A \ P \ Q1 \ X \ \text{qtable } n \ B \ P \ Q2 \ Y \ \neg b \implies B \subseteq A \ C = A \cup B$
 $\bigwedge x. \text{wf_tuple } n \ C \ x \implies P \ x \implies P \ (\text{restrict } A \ x) \wedge P \ (\text{restrict } B \ x)$
shows $\text{qtable } n \ C \ P \ (\lambda x. Q1 \ (\text{restrict } A \ x) \wedge (\text{if } b \ \text{then } Q2 \ (\text{restrict } B \ x) \ \text{else } \neg Q2 \ (\text{restrict } B \ x)))$
 $(\text{join } X \ b \ Y)$

$\langle \text{proof} \rangle$

lemma *wf_tuple_cong*:

assumes $\text{wf_tuple } n \ A \ v \ \text{wf_tuple } n \ A \ w \ \forall x \in A. \text{map the } v \ ! \ x = \text{map the } w \ ! \ x$
shows $v = w$

$\langle \text{proof} \rangle$

definition *mem_restr* :: $'a \ \text{list set} \Rightarrow 'a \ \text{tuple} \Rightarrow \text{bool}$ **where**

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

lemma *mem_restrI*: $y \in A \implies \text{length } y = n \implies \text{wf_tuple } n \ V \ x \implies \forall i \in V. x \ ! \ i = \text{Some } (y \ ! \ i) \implies$
 $\text{mem_restr } A \ x$
 $\langle \text{proof} \rangle$

lemma *mem_restrE*: $\text{mem_restr } A \ x \implies \text{wf_tuple } n \ V \ x \implies \forall i \in V. i < n \implies$
 $(\bigwedge y. y \in A \implies \forall i \in V. x \ ! \ i = \text{Some } (y \ ! \ i) \implies P) \implies P$
 $\langle \text{proof} \rangle$

lemma *mem_restr_IntD*: $mem_restr (A \cap B) v \implies mem_restr A v \wedge mem_restr B v$
 ⟨*proof*⟩

lemma *mem_restr_Un_iff*: $mem_restr (A \cup B) x \iff mem_restr A x \vee mem_restr B x$
 ⟨*proof*⟩

lemma *mem_restr_UNIV [simp]*: $mem_restr UNIV x$
 ⟨*proof*⟩

lemma *restrict_mem_restr [simp]*: $mem_restr A x \implies mem_restr A (restrict V x)$
 ⟨*proof*⟩

definition *lift_envs* :: 'a list set \Rightarrow 'a list set **where**
lift_envs R = $(\lambda(a,b). a \# b) \text{ ' } (UNIV \times R)$

lemma *lift_envs_mem_restr [simp]*: $mem_restr A x \implies mem_restr (lift_envs A) (a \# x)$
 ⟨*proof*⟩

lemma *qtable_project*:

assumes *qtable* (Suc n) A (mem_restr (lift_envs R)) P X

shows *qtable* n (($\lambda x. x - Suc 0$) ' (A - {0})) (mem_restr R)

($\lambda v. \exists x. P ((if 0 \in A \text{ then } Some x \text{ else } None) \# v)$) (tl ' X)

(**is** *qtable* n ?A (mem_restr R) ?P ?X)

⟨*proof*⟩

lemma *qtable_cong*: $qtable n A P Q X \implies A = B \implies (\bigwedge v. P v \implies Q v \iff Q' v) \implies qtable n B P Q' X$
 ⟨*proof*⟩

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_br*: $sat \sigma v i \implies \exists \pi. prefix_of \pi \sigma \wedge (\forall \sigma'. prefix_of \pi \sigma' \longrightarrow sat \sigma' v i)$

begin

lemma *cosafety*: $\text{sat } \sigma \ v \ i \longleftrightarrow (\exists \pi. \text{prefix_of } \pi \ \sigma \wedge (\forall \sigma'. \text{prefix_of } \pi \ \sigma' \longrightarrow \text{sat } \sigma' \ v \ i))$
 ⟨*proof*⟩

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 $M :: 'a \text{ prefix} \Rightarrow (\text{nat} \times 'b \text{ tuple}) \text{ set}$

assumes

mono_monitor: $\pi \leq \pi' \Longrightarrow M \ \pi \subseteq M \ \pi'$ **and**

wf_monitor: $(i, v) \in M \ \pi \Longrightarrow \text{wf_tuple } \text{nfv } \text{fv } v$ **and**

sound_monitor: $(i, v) \in M \ \pi \Longrightarrow \text{prefix_of } \pi \ \sigma \Longrightarrow \text{sat } \sigma \ (\text{map the } v) \ i$ **and**

complete_monitor: $\text{prefix_of } \pi \ \sigma \Longrightarrow \text{wf_tuple } \text{nfv } \text{fv } v \Longrightarrow$

$(\bigwedge \sigma. \text{prefix_of } \pi \ \sigma \Longrightarrow \text{sat } \sigma \ (\text{map the } v) \ i) \Longrightarrow \exists \pi'. \text{prefix_of } \pi' \ \sigma \wedge (i, v) \in M \ \pi'$

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

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

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

begin

lemma *M_limit_eq*: $M_limit \ \sigma = \text{verdicts } \sigma$

⟨*proof*⟩

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** $\text{sat} :: 'a \text{ trace} \Rightarrow 'b \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool} +$

fixes $\text{progress} :: 'a \text{ prefix} \Rightarrow \text{nat}$

assumes

progress_mono: $\pi \leq \pi' \Longrightarrow \text{progress } \pi \leq \text{progress } \pi'$ **and**

ex_progress_ge: $\exists \pi. \text{prefix_of } \pi \ \sigma \wedge x \leq \text{progress } \pi$ **and**

progress_sat_cong: $\text{prefix_of } \pi \ \sigma \Longrightarrow \text{prefix_of } \pi \ \sigma' \Longrightarrow i < \text{progress } \pi \Longrightarrow$

$\text{sat } \sigma \ v \ i \longleftrightarrow \text{sat } \sigma' \ v \ i$

— 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 $M :: 'a \text{ prefix} \Rightarrow (\text{nat} \times 'b \text{ tuple}) \text{ set}$ **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*: $M \ \pi = \{(i, v). i < \text{progress } \pi \wedge \text{wf_tuple } \text{nfv } \text{fv } v \wedge$

$(\exists \sigma. \text{prefix_of } \pi \ \sigma \wedge \text{sat } \sigma \ (\text{map the } v) \ i)\}$

⟨*proof*⟩

end

sublocale $progress \subseteq cosafety_monitor _ _ _ M$
 ⟨*proof*⟩

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 $slice :: 'b \text{ list set} \Rightarrow 'a \text{ trace} \Rightarrow 'a \text{ trace}$ **where**
 $slice \ S \equiv map_ \Gamma (\lambda D. D \cap relevant_events \ S)$

abbreviation $pslice :: 'b \text{ list set} \Rightarrow 'a \text{ prefix} \Rightarrow 'a \text{ prefix}$ **where**
 $pslice \ S \equiv pmap_ \Gamma (\lambda D. D \cap relevant_events \ S)$

lemma $prefix_of_psliceI: prefix_of \ \pi \ \sigma \Longrightarrow prefix_of \ (pslice \ S \ \pi) \ (slice \ S \ \sigma)$
 ⟨*proof*⟩

lemma $plen_pslice[simp]: plen \ (pslice \ S \ \pi) = plen \ \pi$
 ⟨*proof*⟩

lemma $pslice_pnil[simp]: pslice \ S \ pnil = pnil$
 ⟨*proof*⟩

lemma $last_ts_pslice[simp]: last_ts \ (pslice \ S \ \pi) = last_ts \ \pi$
 ⟨*proof*⟩

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

end

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

lemma $union_verdicts_slice:$
assumes $part: \bigcup S = UNIV$
shows $\bigcup ((\lambda S. verdict_filter \ S \ (verdicts \ (slice \ S \ \sigma))) \ 'S) = verdicts \ \sigma$
 ⟨*proof*⟩

end

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

locale $sliceable_monitor = monitor _ _ _ sat \ M + abstract_slicer \ relevant_events$
for $relevant_events :: 'b \text{ list set} \Rightarrow 'a \text{ set}$ **and** $sat :: 'a \text{ trace} \Rightarrow 'b \text{ list} \Rightarrow nat \Rightarrow bool$ **and** $M +$
assumes $sliceable_M: mem_restr \ S \ v \Longrightarrow (i, v) \in M \ (pslice \ S \ \pi) \longleftrightarrow (i, v) \in M \ \pi$
begin

lemma $union_M_pslice:$
assumes $part: \bigcup S = UNIV$
shows $\bigcup ((\lambda S. verdict_filter \ S \ (M \ (pslice \ S \ \pi))) \ 'S) = M \ \pi$
 ⟨*proof*⟩

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  $\pi = \text{pts } \pi' \implies \text{progress } \pi = \text{progress } \pi'$ 
```

```
locale sliceable_timed_progress = sliceable_fo_spec + timed_progress  
begin
```

```
lemma progress_pslice[simp]: progress (pslice S  $\pi$ ) = progress  $\pi$   
  <proof>
```

end

```
sublocale sliceable_timed_progress  $\subseteq$  sliceable_monitor _ _ _ _ M  
<proof>
```

4 Intervals

```
typedef  $\mathcal{I} = \{(i :: \text{nat}, j :: \text{enat}). i \leq j\}$   
<proof>
```

```
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 (=) <proof>  
instance <proof>  
end
```

```
instantiation  $\mathcal{I} :: \text{linorder}$  begin  
lift_definition less_eq  $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow \text{bool}$  is ( $\leq$ ) <proof>  
lift_definition less  $\mathcal{I} :: \mathcal{I} \Rightarrow \mathcal{I} \Rightarrow \text{bool}$  is ( $<$ ) <proof>  
instance <proof>  
end
```

```
lift_definition all ::  $\mathcal{I}$  is ( $(0, \infty)$ ) <proof>  
lift_definition left ::  $\mathcal{I} \Rightarrow \text{nat}$  is fst <proof>  
lift_definition right ::  $\mathcal{I} \Rightarrow \text{enat}$  is snd <proof>  
lift_definition point ::  $\text{nat} \Rightarrow \mathcal{I}$  is  $\lambda n. (n, \text{enat } n)$  <proof>  
lift_definition init ::  $\text{nat} \Rightarrow \mathcal{I}$  is  $\lambda n. (0, \text{enat } n)$  <proof>  
abbreviation mem where mem  $n I \equiv (\text{left } I \leq n \wedge n \leq \text{right } I)$   
lift_definition subtract ::  $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is  
   $\lambda n (i, j). (i - n, j - \text{enat } n)$  <proof>  
lift_definition add ::  $\text{nat} \Rightarrow \mathcal{I} \Rightarrow \mathcal{I}$  is  
   $\lambda n (a, b). (a, b + \text{enat } n)$  <proof>
```

```
lemma left_right: left  $I \leq \text{right } I$   
<proof>
```

```
lemma point_simps[simp]:  
  left (point  $n$ ) =  $n$   
  right (point  $n$ ) =  $n$   
<proof>
```

```
lemma init_simps[simp]:  
  left (init  $n$ ) =  $0$ 
```

right (*init* *n*) = *n*
 ⟨*proof*⟩

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*)
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *subtract_shifted*: *subtract* *n* *I* \in *shifted* *I*
 ⟨*proof*⟩

lemma *finite_shifted*: *finite* (*shifted* *I*)
 ⟨*proof*⟩

definition *interval* :: $\text{nat} \Rightarrow \text{enat} \Rightarrow \mathcal{I}$ **where**
interval *l* *r* = (if $l \leq r$ then *Abs* $_I$ (*l*, *r*) else *undefined*)

lemma [*code abstract*]: *Rep* $_I$ (*interval* *l* *r*) = (if $l \leq r$ then (*l*, *r*) else *Rep* $_I$ *undefined*)
 ⟨*proof*⟩

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*: $\forall x \in \text{fv_trm } t. v ! x = v' ! x \implies \text{eval_trm } v t = \text{eval_trm } v' t$
 ⟨*proof*⟩ **datatype** (*discs_sels*) 'a *formula* = *Pred* *name* 'a *trm* *list* | *Eq* 'a *trm* 'a *trm*
 | *Neg* 'a *formula* | *Or* 'a *formula* 'a *formula* | *Exists* 'a *formula*

| *Prev* \mathcal{I} 'a formula | *Next* \mathcal{I} 'a formula
 | *Since* 'a formula \mathcal{I} 'a formula | *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* φ) = *fvi* b φ
 | *fvi* b (*Or* φ ψ) = *fvi* b φ \cup *fvi* b ψ
 | *fvi* b (*Exists* φ) = *fvi* (Suc b) φ
 | *fvi* b (*Prev* I φ) = *fvi* b φ
 | *fvi* b (*Next* I φ) = *fvi* b φ
 | *fvi* b (*Since* φ I ψ) = *fvi* b φ \cup *fvi* b ψ
 | *fvi* b (*Until* φ I ψ) = *fvi* b φ \cup *fvi* b ψ

abbreviation *fv* \equiv *fvi* 0

lemma *finite_fvi_trm[simp]*: finite (*fvi*_trm b t)
 <proof>

lemma *finite_fvi[simp]*: finite (*fvi* b φ)
 <proof>

lemma *fvi_trm_Suc*: $x \in$ *fvi*_trm (Suc b) t \longleftrightarrow Suc x \in *fvi*_trm b t
 <proof>

lemma *fvi_Suc*: $x \in$ *fvi* (Suc b) $\varphi \longleftrightarrow$ Suc x \in *fvi* b φ
 <proof>

lemma *fvi_Suc_bound*:

assumes $\forall i \in$ *fvi* (Suc b) φ . $i < n$

shows $\forall i \in$ *fvi* b φ . $i < \text{Suc } n$

<proof> **definition** *nfv* :: 'a formula \Rightarrow nat **where**
nfv $\varphi = \text{Max} (\text{insert } 0 (\text{Suc } 'fv \varphi))$

qualified definition *envs* :: 'a formula \Rightarrow 'a env set **where**

envs $\varphi = \{v. \text{length } v = \text{nfv } \varphi\}$

lemma *nfv_simps[simp]*:

nfv (*Neg* φ) = *nfv* φ
nfv (*Or* φ ψ) = max (*nfv* φ) (*nfv* ψ)
nfv (*Prev* I φ) = *nfv* φ
nfv (*Next* I φ) = *nfv* φ
nfv (*Since* φ I ψ) = max (*nfv* φ) (*nfv* ψ)
nfv (*Until* φ I ψ) = max (*nfv* φ) (*nfv* ψ)
 <proof>

lemma *fvi_less_nfv*: $\forall i \in$ *fv* φ . $i < \text{nfv } \varphi$

<proof> **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* φ ψ) = max (*future_reach* φ) (*future_reach* ψ)
 | *future_reach* (*Exists* φ) = *future_reach* φ
 | *future_reach* (*Prev* I φ) = *future_reach* φ - left I
 | *future_reach* (*Next* I φ) = *future_reach* φ + right I + 1
 | *future_reach* (*Since* φ I ψ) = max (*future_reach* φ) (*future_reach* ψ - left I)
 | *future_reach* (*Until* φ I ψ) = max (*future_reach* φ) (*future_reach* ψ) + right I + 1

qualified primrec $\text{sat} :: 'a \text{ trace} \Rightarrow 'a \text{ env} \Rightarrow \text{nat} \Rightarrow 'a \text{ formula} \Rightarrow \text{bool}$ where

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

lemma sat_Until_rec : $\text{sat } \sigma \ v \ i \ (\text{Until } \varphi \ I \ \psi) \longleftrightarrow$
 $\text{mem } 0 \ I \wedge \text{sat } \sigma \ v \ i \ \psi \vee$
 $(\Delta \ \sigma \ (i + 1) \leq \text{right } I \wedge \text{sat } \sigma \ v \ i \ \varphi \wedge \text{sat } \sigma \ v \ (i + 1) \ (\text{Until } \varphi \ (\text{subtract } (\Delta \ \sigma \ (i + 1)) \ I) \ \psi))$
 $(\text{is } ?L \longleftrightarrow ?R)$
 $\langle \text{proof} \rangle$

lemma sat_Since_rec : $\text{sat } \sigma \ v \ i \ (\text{Since } \varphi \ I \ \psi) \longleftrightarrow$
 $\text{mem } 0 \ I \wedge \text{sat } \sigma \ v \ i \ \psi \vee$
 $(i > 0 \wedge \Delta \ \sigma \ i \leq \text{right } I \wedge \text{sat } \sigma \ v \ i \ \varphi \wedge \text{sat } \sigma \ v \ (i - 1) \ (\text{Since } \varphi \ (\text{subtract } (\Delta \ \sigma \ i) \ I) \ \psi))$
 $(\text{is } ?L \longleftrightarrow ?R)$
 $\langle \text{proof} \rangle$

lemma sat_Since_0 : $\text{sat } \sigma \ v \ 0 \ (\text{Since } \varphi \ I \ \psi) \longleftrightarrow \text{mem } 0 \ I \wedge \text{sat } \sigma \ v \ 0 \ \psi$
 $\langle \text{proof} \rangle$

lemma sat_Since_point : $\text{sat } \sigma \ v \ i \ (\text{Since } \varphi \ I \ \psi) \Longrightarrow$
 $(\bigwedge j. j \leq i \Longrightarrow \text{mem } (\tau \ \sigma \ i - \tau \ \sigma \ j) \ I \Longrightarrow \text{sat } \sigma \ v \ i \ (\text{Since } \varphi \ (\text{point } (\tau \ \sigma \ i - \tau \ \sigma \ j)) \ \psi) \Longrightarrow P) \Longrightarrow P$
 $\langle \text{proof} \rangle$

lemma sat_Since_pointD : $\text{sat } \sigma \ v \ i \ (\text{Since } \varphi \ (\text{point } t) \ \psi) \Longrightarrow \text{mem } t \ I \Longrightarrow \text{sat } \sigma \ v \ i \ (\text{Since } \varphi \ I \ \psi)$
 $\langle \text{proof} \rangle$

lemma eval_trm_fvi_cong : $\forall x \in \text{fv_trm } t. v!x = v'!x \Longrightarrow \text{eval_trm } v \ t = \text{eval_trm } v' \ t$
 $\langle \text{proof} \rangle$

lemma sat_fvi_cong : $\forall x \in \text{fv } \varphi. v!x = v'!x \Longrightarrow \text{sat } \sigma \ v \ i \ \varphi = \text{sat } \sigma \ v' \ i \ \varphi$
 $\langle \text{proof} \rangle$

5.2 Defined connectives

qualified definition $\text{And } \varphi \ \psi = \text{Neg } (\text{Or } (\text{Neg } \varphi) \ (\text{Neg } \psi))$

lemma fvi_And : $\text{fvi } b \ (\text{And } \varphi \ \psi) = \text{fvi } b \ \varphi \cup \text{fvi } b \ \psi$
 $\langle \text{proof} \rangle$

lemma nfv_And[simp] : $\text{nfv } (\text{And } \varphi \ \psi) = \max (\text{nfv } \varphi) \ (\text{nfv } \psi)$
 $\langle \text{proof} \rangle$

lemma future_reach_And : $\text{future_reach } (\text{And } \varphi \ \psi) = \max (\text{future_reach } \varphi) \ (\text{future_reach } \psi)$
 $\langle \text{proof} \rangle$

lemma sat_And : $\text{sat } \sigma \ v \ i \ (\text{And } \varphi \ \psi) = (\text{sat } \sigma \ v \ i \ \varphi \wedge \text{sat } \sigma \ v \ i \ \psi)$
 $\langle \text{proof} \rangle$ **definition** $\text{And_Not } \varphi \ \psi = \text{Neg } (\text{Or } (\text{Neg } \varphi) \ \psi)$

lemma fvi_And_Not : $\text{fvi } b \ (\text{And_Not } \varphi \ \psi) = \text{fvi } b \ \varphi \cup \text{fvi } b \ \psi$
 $\langle \text{proof} \rangle$

lemma *nfv_And_Not[simp]*: $nfv (And_Not \varphi \psi) = max (nfv \varphi) (nfv \psi)$
 ⟨proof⟩

lemma *future_reach_And_Not*: $future_reach (And_Not \varphi \psi) = max (future_reach \varphi) (future_reach \psi)$
 ⟨proof⟩

lemma *sat_And_Not*: $sat \sigma v i (And_Not \varphi \psi) = (sat \sigma v i \varphi \wedge \neg sat \sigma v i \psi)$
 ⟨proof⟩

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 φ) ψ)) = (*safe_formula* φ \wedge
 (safe_formula ψ \wedge MFOTL.fv ψ \subseteq MFOTL.fv φ \vee (case ψ of MFOTL.Neg ψ' \Rightarrow safe_formula ψ' |
 _ \Rightarrow False)))
 | *safe_formula* (MFOTL.Or φ ψ) = (MFOTL.fv ψ = MFOTL.fv φ \wedge *safe_formula* φ \wedge *safe_formula*
 ψ)
 | *safe_formula* (MFOTL.Exists φ) = (*safe_formula* φ)
 | *safe_formula* (MFOTL.Prev I φ) = (*safe_formula* φ)
 | *safe_formula* (MFOTL.Next I φ) = (*safe_formula* φ)
 | *safe_formula* (MFOTL.Since φ I ψ) = (MFOTL.fv φ \subseteq MFOTL.fv ψ \wedge
 (safe_formula φ \vee (case φ of MFOTL.Neg φ' \Rightarrow safe_formula φ' | _ \Rightarrow False)) \wedge *safe_formula* ψ)
 | *safe_formula* (MFOTL.Until φ I ψ) = (MFOTL.fv φ \subseteq MFOTL.fv ψ \wedge
 (safe_formula φ \vee (case φ of MFOTL.Neg φ' \Rightarrow safe_formula φ' | _ \Rightarrow False)) \wedge *safe_formula* ψ)
 | *safe_formula* _ = False

lemma *disjE_Not2*: $P \vee Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (\neg P \Longrightarrow Q \Longrightarrow R) \Longrightarrow R$
 ⟨proof⟩

lemma *safe_formula_induct[consumes 1]*:

assumes *safe_formula* φ
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 $\bigwedge x y. x = y \Longrightarrow P (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var y)))$
and $\bigwedge e ts. P (MFOTL.Pred e ts)$
and $\bigwedge \varphi \psi. \neg (safe_formula (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \Longrightarrow P \varphi \Longrightarrow P \psi$
 $\Longrightarrow P (MFOTL.And \varphi \psi)$
and $\bigwedge \varphi \psi. safe_formula \psi \Longrightarrow MFOTL.fv \psi \subseteq MFOTL.fv \varphi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.And_Not$
 $\varphi \psi)$
and $\bigwedge \varphi \psi. MFOTL.fv \psi = MFOTL.fv \varphi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.Or \varphi \psi)$
and $\bigwedge \varphi. P \varphi \Longrightarrow P (MFOTL.Exists \varphi)$
and $\bigwedge I \varphi. P \varphi \Longrightarrow P (MFOTL.Prev I \varphi)$
and $\bigwedge I \varphi. P \varphi \Longrightarrow P (MFOTL.Next I \varphi)$
and $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \Longrightarrow safe_formula \varphi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.Since$
 $\varphi I \psi)$
and $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \Longrightarrow$
 $\neg safe_formula (MFOTL.Neg \varphi) \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.Since (MFOTL.Neg \varphi) I \psi)$
and $\bigwedge \varphi I \psi. MFOTL.fv \varphi \subseteq MFOTL.fv \psi \Longrightarrow safe_formula \varphi \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.Until$
 $\varphi I \psi)$
and $\bigwedge \varphi I \psi. MFOTL.fv (MFOTL.Neg \varphi) \subseteq MFOTL.fv \psi \Longrightarrow$
 $\neg safe_formula (MFOTL.Neg \varphi) \Longrightarrow P \varphi \Longrightarrow P \psi \Longrightarrow P (MFOTL.Until (MFOTL.Neg \varphi) I \psi)$

shows $P \varphi$
 ⟨proof⟩

5.4 Slicing traces

qualified primrec $\text{matches} :: 'a \text{ env} \Rightarrow 'a \text{ formula} \Rightarrow \text{name} \times 'a \text{ list} \Rightarrow \text{bool}$ **where**
 $\text{matches } v \text{ (Pred } r \text{ ts)} e = (r = \text{fst } e \wedge \text{map } (\text{eval_trm } v) \text{ ts} = \text{snd } e)$
 $\text{matches } v \text{ (Eq } _ _ _) e = \text{False}$
 $\text{matches } v \text{ (Neg } \varphi) e = \text{matches } v \varphi e$
 $\text{matches } v \text{ (Or } \varphi \psi) e = (\text{matches } v \varphi e \vee \text{matches } v \psi e)$
 $\text{matches } v \text{ (Exists } \varphi) e = (\exists z. \text{matches } (z \# v) \varphi e)$
 $\text{matches } v \text{ (Prev } I \varphi) e = \text{matches } v \varphi e$
 $\text{matches } v \text{ (Next } I \varphi) e = \text{matches } v \varphi e$
 $\text{matches } v \text{ (Since } \varphi I \psi) e = (\text{matches } v \varphi e \vee \text{matches } v \psi e)$
 $\text{matches } v \text{ (Until } \varphi I \psi) e = (\text{matches } v \varphi e \vee \text{matches } v \psi e)$

lemma $\text{matches_fvi_cong}: \forall x \in \text{fv } \varphi. v!x = v'!x \implies \text{matches } v \varphi e = \text{matches } v' \varphi e$
 ⟨proof⟩

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

lemma $\text{sat_slice_strong}: \text{relevant_events } \varphi S \subseteq E \implies v \in S \implies$
 $\text{sat } \sigma v i \varphi \longleftrightarrow \text{sat } (\text{map_}\Gamma (\lambda D. D \cap E) \sigma) v i \varphi$
 ⟨proof⟩

end

interpretation $\text{MFOTL_slicer}: \text{abstract_slicer relevant_events } \varphi$ **for** φ ⟨proof⟩

lemma $\text{sat_slice_iff}: \text{assumes } v \in S$
shows $\text{MFOTL.sat } \sigma v i \varphi \longleftrightarrow \text{MFOTL.sat } (\text{MFOTL_slicer.slice } \varphi S \sigma) v i \varphi$
 ⟨proof⟩

lemma $\text{slice_replace_prefix}: \text{prefix_of } (\text{MFOTL_slicer.pslic} \varphi R \pi) \sigma \implies$
 $\text{MFOTL_slicer.slice } \varphi R (\text{replace_prefix } \pi \sigma) = \text{MFOTL_slicer.slice } \varphi R \sigma$
 ⟨proof⟩

6 Monitor implementation

6.1 Monitorable formulas

definition $\text{mmonitorable } \varphi \longleftrightarrow \text{safe_formula } \varphi \wedge \text{MFOTL.future_reach } \varphi \neq \infty$

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

```

| 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: $a + b = \text{enat } i \iff (\exists j k. a = \text{enat } j \wedge b = \text{enat } k \wedge j + k = i)$
 <proof>

lemma minus_eq_enat_iff: $a - \text{enat } k = \text{enat } i \iff (\exists j. a = \text{enat } j \wedge j - k = i)$
 <proof>

lemma safe_formula_mmonitorable_exec: $\text{safe_formula } \varphi \implies \text{MFOTL.future_reach } \varphi \neq \infty \implies \text{mmonitorable_exec } \varphi$
 <proof>

lemma mmonitorable_exec_mmonitorable: $\text{mmonitorable_exec } \varphi \implies \text{mmonitorable } \varphi$
 <proof>

lemma monitorable_formula_code[code]: $\text{mmonitorable } \varphi = \text{mmonitorable_exec } \varphi$
 <proof>

6.2 The executable monitor

type_synonym $ts = \text{nat}$

type_synonym $'a \text{ mbuf2} = 'a \text{ table list} \times 'a \text{ table list}$
type_synonym $'a \text{ msaux} = (ts \times 'a \text{ table}) \text{ list}$
type_synonym $'a \text{ muaux} = (ts \times 'a \text{ table} \times 'a \text{ table}) \text{ list}$

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

record $'a \text{ mstate} =$
 mstate_i :: nat
 mstate_m :: $'a \text{ mformula}$
 mstate_n :: nat

fun $\text{eq_rel} :: \text{nat} \Rightarrow 'a \text{ MFOTL.trm} \Rightarrow 'a \text{ MFOTL.trm} \Rightarrow 'a \text{ table}$ **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 $\text{neq_rel} :: \text{nat} \Rightarrow 'a \text{ MFOTL.trm} \Rightarrow 'a \text{ MFOTL.trm} \Rightarrow 'a \text{ table}$ **where**
 neq_rel n (MFOTL.Const x) (MFOTL.Const y) = (if $x = y$ then empty_table else unit_table n)
 | neq_rel n (MFOTL.Var x) (MFOTL.Var y) = (if $x = y$ then empty_table else undefined)
 | neq_rel $_ _ _ =$ undefined

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

definition update_since :: ℐ ⇒ bool ⇒ 'a table ⇒ 'a table ⇒ ts ⇒

```

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

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

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

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

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

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

6.3 Progress

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

lemma *progress_And[simp]*: $progress\ \sigma\ (MFOTL.And\ \varphi\ \psi)\ j = \min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
 <proof>

lemma *progress_And_Not[simp]*: $progress\ \sigma\ (MFOTL.And_Not\ \varphi\ \psi)\ j = \min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
 <proof>

lemma *progress_mono*: $j \leq j' \implies progress\ \sigma\ \varphi\ j \leq progress\ \sigma\ \varphi\ j'$
 <proof>

lemma *progress_le*: $progress\ \sigma\ \varphi\ j \leq j$
 <proof>

lemma *progress_0[simp]*: $progress\ \sigma\ \varphi\ 0 = 0$
 <proof>

lemma *progress_ge*: $MFOTL.future_reach\ \varphi \neq \infty \implies \exists j. i \leq progress\ \sigma\ \varphi\ j$
 <proof>

lemma *cInf_restrict_nat*:
 fixes $x :: nat$
 assumes $x \in A$
 shows $Inf\ A = Inf\ \{y \in A. y \leq x\}$
 <proof>

lemma *progress_time_conv*:
 assumes $\forall i < j. \tau\ \sigma\ i = \tau\ \sigma'\ i$
 shows $progress\ \sigma\ \varphi\ j = progress\ \sigma'\ \varphi\ j$
 <proof>

lemma *Inf_UNIV_nat*: $(Inf\ UNIV :: nat) = 0$
 <proof>

lemma *progress_prefix_conv*:
 assumes *prefix_of* $\pi\ \sigma$ and *prefix_of* $\pi\ \sigma'$
 shows $progress\ \sigma\ \varphi\ (plen\ \pi) = progress\ \sigma'\ \varphi\ (plen\ \pi)$
 <proof>

lemma *sat_prefix_conv*:
assumes *prefix_of* π σ **and** *prefix_of* π σ' **and** $i < \text{progress } \sigma \ \varphi$ (*plen* π)
shows $\text{MFOTL.sat } \sigma \ v \ i \ \varphi \longleftrightarrow \text{MFOTL.sat } \sigma' \ v \ i \ \varphi$
<proof>

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*: $\text{prefix_of } \pi \ \sigma \Longrightarrow \text{pprogress } \varphi \ \pi = \text{progress } \sigma \ \varphi \ (\text{plen } \pi)$
<proof>

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

sublocale *future_bounded_mfotl* \subseteq *sliceable_timed_progress* $\text{MFOTL.nfv } \varphi \ \text{MFOTL.fv } \varphi \ \text{relevant_events } \varphi$
 $\lambda \sigma \ v \ i. \ \text{MFOTL.sat } \sigma \ v \ i \ \varphi \ \text{pprogress } \varphi$
<proof>

locale *monitorable_mfotl* =
fixes $\varphi :: 'a \text{ MFOTL.formula}$
assumes *monitorable*: $\text{mmonitorable } \varphi$

sublocale *monitorable_mfotl* \subseteq *future_bounded_mfotl*
<proof>

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 $i \ j_a \ j_b \ P \ Q \ \text{buf} \longleftrightarrow i \leq j_a \wedge i \leq j_b \wedge (\text{case } \text{buf} \text{ of } (xs, ys) \Rightarrow \text{list_all2 } P \ [i..<j_a] \ xs \wedge \text{list_all2 } Q \ [i..<j_b] \ ys)$

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' $\sigma \ j \ n \ R \ \varphi \ \psi \ \text{buf} \longleftrightarrow \text{wf_mbuf2} \ (\text{min } (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)) \ (\text{progress } \sigma \ \varphi \ j) \ (\text{progress } \sigma \ \psi \ j)$
 $(\lambda i. \ \text{qtable } n \ (\text{MFOTL.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \varphi))$
 $(\lambda i. \ \text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \ \text{MFOTL.sat } \sigma \ (\text{map the } v) \ i \ \psi)) \ \text{buf}$

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

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

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

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

lemma $qtable_mem_restr_UNIV: qtable n A (mem_restr UNIV) Q X = wf_table n A Q X$
 $\langle proof \rangle$

lemma $wf_since_aux_UNIV_alt:$

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

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 $wf_until_aux_UNIV_alt:$

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

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**

$Eq: MFOTL.is_Const t1 \vee MFOTL.is_Const t2 \Longrightarrow$

$\forall x \in MFOTL.fv_trm t1. x < n \Longrightarrow \forall x \in MFOTL.fv_trm t2. x < n \Longrightarrow$

$wf_mformula \sigma j n R (MRel (eq_rel n t1 t2)) (MFOTL.Eq t1 t2)$

$| neq_Const: \varphi = MRel (neq_rel n (MFOTL.Const x) (MFOTL.Const y)) \Longrightarrow$

$wf_mformula \sigma j n R \varphi (MFOTL.Neg (MFOTL.Eq (MFOTL.Const x) (MFOTL.Const y)))$

$| neq_Var: x < n \Longrightarrow$

$wf_mformula \sigma j n R (MRel \text{ empty_table}) (MFOTL.Neg (MFOTL.Eq (MFOTL.Var x) (MFOTL.Var x)))$
 $| Pred: \forall x \in MFOTL.fv (MFOTL.Pred e ts). x < n \implies$
 $wf_mformula \sigma j n R (MPred e ts) (MFOTL.Pred e ts)$
 $| And: wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $if \text{ pos then } \chi = MFOTL.And \varphi' \psi' \wedge \neg (\text{safe_formula } (MFOTL.Neg \psi') \wedge MFOTL.fv \psi' \subseteq MFOTL.fv \varphi')$
 $\varphi')$
 $else \chi = MFOTL.And_Not \varphi' \psi' \wedge \text{safe_formula } \psi' \wedge MFOTL.fv \psi' \subseteq MFOTL.fv \varphi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' \text{ buf} \implies$
 $wf_mformula \sigma j n R (MAnd \varphi \text{ pos } \psi \text{ buf}) \chi$
 $| Or: wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $MFOTL.fv \varphi' = MFOTL.fv \psi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' \text{ buf} \implies$
 $wf_mformula \sigma j n R (MOr \varphi \psi \text{ buf}) (MFOTL.Or \varphi' \psi')$
 $| Exists: wf_mformula \sigma j (Suc n) (\text{lift_envs } R) \varphi \varphi' \implies$
 $wf_mformula \sigma j n R (MExists \varphi) (MFOTL.Exists \varphi')$
 $| Prev: wf_mformula \sigma j n R \varphi \varphi' \implies$
 $first \longleftrightarrow j = 0 \implies$
 $list_all2 (\lambda i. \text{qtable } n (MFOTL.fv \varphi') (\text{mem_restr } R) (\lambda v. MFOTL.sat \sigma (\text{map the } v) i \varphi'))$
 $[\text{min } (\text{progress } \sigma \varphi' j) (j-1)..<\text{progress } \sigma \varphi' j] \text{ buf} \implies$
 $list_all2 (\lambda i t. t = \tau \sigma i) [\text{min } (\text{progress } \sigma \varphi' j) (j-1)..<j] \text{ nts} \implies$
 $wf_mformula \sigma j n R (MPrev I \varphi \text{ first } \text{buf } \text{nts}) (MFOTL.Prev I \varphi')$
 $| Next: wf_mformula \sigma j n R \varphi \varphi' \implies$
 $first \longleftrightarrow \text{progress } \sigma \varphi' j = 0 \implies$
 $list_all2 (\lambda i t. t = \tau \sigma i) [\text{progress } \sigma \varphi' j - 1..<j] \text{ nts} \implies$
 $wf_mformula \sigma j n R (MNext I \varphi \text{ first } \text{nts}) (MFOTL.Next I \varphi')$
 $| Since: wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $if \text{ pos then } \varphi'' = \varphi' \text{ else } \varphi'' = MFOTL.Neg \varphi' \implies$
 $\text{safe_formula } \varphi'' = \text{pos} \implies$
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' \text{ buf} \implies$
 $wf_ts \sigma j \varphi' \psi' \text{ nts} \implies$
 $wf_since_aux \sigma n R \text{pos } \varphi' I \psi' \text{aux } (\text{progress } \sigma (MFOTL.Since \varphi'' I \psi') j) \implies$
 $wf_mformula \sigma j n R (MSince \text{pos } \varphi I \psi \text{buf } \text{nts } \text{aux}) (MFOTL.Since \varphi'' I \psi')$
 $| Until: wf_mformula \sigma j n R \varphi \varphi' \implies wf_mformula \sigma j n R \psi \psi' \implies$
 $if \text{ pos then } \varphi'' = \varphi' \text{ else } \varphi'' = MFOTL.Neg \varphi' \implies$
 $\text{safe_formula } \varphi'' = \text{pos} \implies$
 $MFOTL.fv \varphi' \subseteq MFOTL.fv \psi' \implies$
 $wf_mbuf2' \sigma j n R \varphi' \psi' \text{ buf} \implies$
 $wf_ts \sigma j \varphi' \psi' \text{ nts} \implies$
 $wf_until_aux \sigma n R \text{pos } \varphi' I \psi' \text{aux } (\text{progress } \sigma (MFOTL.Until \varphi'' I \psi') j) \implies$
 $\text{progress } \sigma (MFOTL.Until \varphi'' I \psi') j + \text{length } \text{aux} = \text{min } (\text{progress } \sigma \varphi' j) (\text{progress } \sigma \psi' j) \implies$
 $wf_mformula \sigma j n R (MUntil \text{pos } \varphi I \psi \text{buf } \text{nts } \text{aux}) (MFOTL.Until \varphi'' I \psi')$

definition $wf_mstate :: 'a MFOTL.formula \Rightarrow 'a MFOTL.prefix \Rightarrow 'a \text{ list set} \Rightarrow 'a \text{ mstate} \Rightarrow \text{bool}$ **where**
 $wf_mstate \varphi \pi R st \longleftrightarrow \text{mstate_n } st = MFOTL.nfv \varphi \wedge (\forall \sigma. \text{prefix_of } \pi \sigma \longrightarrow$
 $\text{mstate_i } st = \text{progress } \sigma \varphi (\text{plen } \pi) \wedge$
 $wf_mformula \sigma (\text{plen } \pi) (\text{mstate_n } st) R (\text{mstate_m } st) \varphi)$

6.5.2 Initialisation

lemma $\text{minit0_And}: \neg (\text{safe_formula } (MFOTL.Neg \psi) \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi) \implies$
 $\text{minit0 } n (MFOTL.And \varphi \psi) = MAnd (\text{minit0 } n \varphi) \text{ True } (\text{minit0 } n \psi) ([], [])$
 $\langle \text{proof} \rangle$

lemma $\text{minit0_And_Not}: \text{safe_formula } \psi \wedge MFOTL.fv \psi \subseteq MFOTL.fv \varphi \implies$
 $\text{minit0 } n (MFOTL.And_Not \varphi \psi) = (MAnd (\text{minit0 } n \varphi) \text{ False } (\text{minit0 } n \psi) ([], []))$
 $\langle \text{proof} \rangle$

lemma *wf_mbuf2'_0*: $wf_mbuf2' \sigma 0 n R \varphi \psi (\[], \[])$
 ⟨proof⟩

lemma *wf_ts_0*: $wf_ts \sigma 0 \varphi \psi \[]$
 ⟨proof⟩

lemma *wf_since_aux_Nil*: $wf_since_aux \sigma n R pos \varphi' I \psi' \[] 0$
 ⟨proof⟩

lemma *wf_until_aux_Nil*: $wf_until_aux \sigma n R pos \varphi' I \psi' \[] 0$
 ⟨proof⟩

lemma *wf_minit0*: $safe_formula \varphi \implies \forall x \in MFOTL.fv \varphi. x < n \implies$
 $wf_mformula \sigma 0 n R (minit0 n \varphi) \varphi$
 ⟨proof⟩

lemma *wf_mstate_minit*: $safe_formula \varphi \implies wf_mstate \varphi pnil R (minit \varphi)$
 ⟨proof⟩

6.5.3 Evaluation

lemma *match_wf_tuple*: $Some f = match\ ts\ xs \implies wf_tuple\ n (\bigcup_{t \in set\ ts} MFOTL.fv_trm\ t)$ (tabulate
 $f\ 0\ n$)
 ⟨proof⟩

lemma *match_fvi_trm_None*: $Some f = match\ ts\ xs \implies \forall t \in set\ ts. x \notin MFOTL.fv_trm\ t \implies f\ x =$
 $None$
 ⟨proof⟩

lemma *match_fvi_trm_Some*: $Some f = match\ ts\ xs \implies t \in set\ ts \implies x \in MFOTL.fv_trm\ t \implies f\ x =$
 $\neq None$
 ⟨proof⟩

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

lemma *wf_tuple_tabulate_Some*: $wf_tuple\ n\ A\ (tabulate\ f\ 0\ n) \implies x \in A \implies x < n \implies \exists y. f\ x =$
 $Some\ y$
 ⟨proof⟩

lemma *ex_match*: $wf_tuple\ n (\bigcup_{t \in set\ ts} MFOTL.fv_trm\ t)\ v \implies \forall t \in set\ ts. \forall x \in MFOTL.fv_trm\ t. x$
 $< n \implies$
 $\exists f. match\ ts\ (map\ (MFOTL.eval_trm\ (map\ the\ v))\ ts) = Some\ f \wedge v = tabulate\ f\ 0\ n$
 ⟨proof⟩

lemma *eq_rel_eval_trm*: $v \in eq_rel\ n\ t1\ t2 \implies MFOTL.is_Const\ t1 \vee MFOTL.is_Const\ t2 \implies$
 $\forall x \in MFOTL.fv_trm\ t1 \cup MFOTL.fv_trm\ t2. x < n \implies$
 $MFOTL.eval_trm\ (map\ the\ v)\ t1 = MFOTL.eval_trm\ (map\ the\ v)\ t2$
 ⟨proof⟩

lemma *in_eq_rel*: $wf_tuple\ n\ (MFOTL.fv_trm\ t1 \cup MFOTL.fv_trm\ t2)\ v \implies$
 $MFOTL.is_Const\ t1 \vee MFOTL.is_Const\ t2 \implies$
 $MFOTL.eval_trm\ (map\ the\ v)\ t1 = MFOTL.eval_trm\ (map\ the\ v)\ t2 \implies$
 $v \in eq_rel\ n\ t1\ t2$
 ⟨proof⟩

lemma *table_eq_rel*: $MFOTL.is_Const\ t1 \vee MFOTL.is_Const\ t2 \implies$
 $table\ n\ (MFOTL.fv_trm\ t1 \cup MFOTL.fv_trm\ t2)\ (eq_rel\ n\ t1\ t2)$

<proof>

lemma *wf_tuple_Suc_fviD*: $wf_tuple (Suc\ n) (MFOTL.fvi\ b\ \varphi)\ v \implies wf_tuple\ n (MFOTL.fvi (Suc\ b)\ \varphi) (tl\ v)$

<proof>

lemma *table_fvi_tl*: $table (Suc\ n) (MFOTL.fvi\ b\ \varphi)\ X \implies table\ n (MFOTL.fvi (Suc\ b)\ \varphi) (tl\ 'X)$

<proof>

lemma *wf_tuple_Suc_fvi_SomeI*: $0 \in MFOTL.fvi\ b\ \varphi \implies wf_tuple\ n (MFOTL.fvi (Suc\ b)\ \varphi)\ v \implies wf_tuple (Suc\ n) (MFOTL.fvi\ b\ \varphi) (Some\ x\ \# v)$

<proof>

lemma *wf_tuple_Suc_fvi_NoneI*: $0 \notin MFOTL.fvi\ b\ \varphi \implies wf_tuple\ n (MFOTL.fvi (Suc\ b)\ \varphi)\ v \implies wf_tuple (Suc\ n) (MFOTL.fvi\ b\ \varphi) (None\ \# v)$

<proof>

lemma *qtable_project_fv*: $qtable (Suc\ n) (fv\ \varphi) (mem_restr (lift_envs\ R))\ P\ X \implies$

$qtable\ n (MFOTL.fvi (Suc\ 0)\ \varphi) (mem_restr\ R)$

$(\lambda v. \exists x. P ((if\ 0 \in fv\ \varphi\ then\ Some\ x\ else\ None)\ \# v)) (tl\ 'X)$

<proof>

lemma *mprev*: $mprev_next\ I\ xs\ ts = (ys,\ xs',\ ts') \implies$

$list_all2\ P\ [i..<j]\ xs \implies list_all2 (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts \implies i \leq j' \implies i < j \implies$

$list_all2 (\lambda i\ X. if\ mem (\tau\ \sigma (Suc\ i) - \tau\ \sigma\ i)\ I\ then\ P\ i\ X\ else\ X = empty_table)$

$[i..<min\ j'\ (j-1)]\ ys \wedge$

$list_all2\ P\ [min\ j'\ (j-1)..<j']\ xs' \wedge$

$list_all2 (\lambda i\ t. t = \tau\ \sigma\ i)\ [min\ j'\ (j-1)..<j]\ ts'$

<proof>

lemma *mnext*: $mprev_next\ I\ xs\ ts = (ys,\ xs',\ ts') \implies$

$list_all2\ P\ [Suc\ i..<j']\ xs \implies list_all2 (\lambda i\ t. t = \tau\ \sigma\ i)\ [i..<j]\ ts \implies Suc\ i \leq j' \implies i < j \implies$

$list_all2 (\lambda i\ X. if\ mem (\tau\ \sigma (Suc\ i) - \tau\ \sigma\ i)\ I\ then\ P (Suc\ i)\ X\ else\ X = empty_table)$

$[i..<min\ (j'-1)\ (j-1)]\ ys \wedge$

$list_all2\ P\ [Suc (min (j'-1) (j-1))..<j']\ xs' \wedge$

$list_all2 (\lambda i\ t. t = \tau\ \sigma\ i)\ [min (j'-1) (j-1)..<j]\ ts'$

<proof>

lemma *in_foldr_UnI*: $x \in A \implies A \in set\ xs \implies x \in foldr (\cup)\ xs\ \{\}$

<proof>

lemma *in_foldr_UnE*: $x \in foldr (\cup)\ xs\ \{\} \implies (\bigwedge A. A \in set\ xs \implies x \in A \implies P) \implies P$

<proof>

lemma *sat_the_restrict*: $fv\ \varphi \subseteq A \implies MFOTL.sat\ \sigma (map\ the (restrict\ A\ v))\ i\ \varphi = MFOTL.sat\ \sigma (map\ the\ v)\ i\ \varphi$

<proof>

lemma *update_since*:

assumes *pre*: $wf_since_aux\ \sigma\ n\ R\ pos\ \varphi\ I\ \psi\ aux\ ne$

and *qtable1*: $qtable\ n (MFOTL.fv\ \varphi) (mem_restr\ R) (\lambda v. MFOTL.sat\ \sigma (map\ the\ v)\ ne\ \varphi)\ rel1$

and *qtable2*: $qtable\ n (MFOTL.fv\ \psi) (mem_restr\ R) (\lambda v. MFOTL.sat\ \sigma (map\ the\ v)\ ne\ \psi)\ rel2$

and *result_eq*: $(rel,\ aux') = update_since\ I\ pos\ rel1\ rel2 (\tau\ \sigma\ ne)\ aux$

and *fvi_subset*: $MFOTL.fv\ \varphi \subseteq MFOTL.fv\ \psi$

shows $wf_since_aux\ \sigma\ n\ R\ pos\ \varphi\ I\ \psi\ aux' (Suc\ ne)$

and $qtable\ n (MFOTL.fv\ \psi) (mem_restr\ R) (\lambda v. MFOTL.sat\ \sigma (map\ the\ v)\ ne (Sincep\ pos\ \varphi\ I\ \psi))$

rel

<proof>

lemma *length_update_until*: $\text{length } (\text{update_until } \text{pos } I \text{ rel1 } \text{rel2 } \text{nt } \text{aux}) = \text{Suc } (\text{length } \text{aux})$
 ⟨proof⟩

lemma *wf_update_until*:

assumes *pre*: $\text{wf_until_aux } \sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ \text{aux} \ ne$
and *qtable1*: $\text{qtable } n \ (\text{MFOTL.fv } \varphi) \ (\text{mem_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne + \text{length } \text{aux}) \ \varphi) \ \text{rel1}$
and *qtable2*: $\text{qtable } n \ (\text{MFOTL.fv } \psi) \ (\text{mem_restr } R) \ (\lambda v. \text{MFOTL.sat } \sigma \ (\text{map the } v) \ (ne + \text{length } \text{aux}) \ \psi) \ \text{rel2}$
and *fv_subset*: $\text{MFOTL.fv } \varphi \subseteq \text{MFOTL.fv } \psi$
shows $\text{wf_until_aux } \sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ (\text{update_until } I \ \text{pos } \text{rel1 } \text{rel2 } (\tau \ \sigma \ (ne + \text{length } \text{aux})) \ \text{aux}) \ ne$
 ⟨proof⟩

lemma *wf_until_aux_Cons*: $\text{wf_until_aux } \sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ (a \ \# \ \text{aux}) \ ne \implies$
 $\text{wf_until_aux } \sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ \text{aux} \ (\text{Suc } ne)$
 ⟨proof⟩

lemma *wf_until_aux_Cons1*: $\text{wf_until_aux } \sigma \ n \ R \ \text{pos } \varphi \ I \ \psi \ ((t, a1, a2) \ \# \ \text{aux}) \ ne \implies t = \tau \ \sigma \ ne$
 ⟨proof⟩

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

lemma *upt_append*: $a \leq b \implies b \leq c \implies [a..<b] \ @ \ [b..<c] = [a..<c]$
 ⟨proof⟩

lemma *wf_mbuf2_add*:

assumes $\text{wf_mbuf2 } i \ ja \ jb \ P \ Q \ \text{buf}$
and $\text{list_all2 } P \ [ja..<ja'] \ xs$
and $\text{list_all2 } Q \ [jb..<jb'] \ ys$
and $ja \leq ja' \ jb \leq jb'$
shows $\text{wf_mbuf2 } i \ ja' \ jb' \ P \ Q \ (\text{mbuf2_add } xs \ ys \ \text{buf})$
 ⟨proof⟩

lemma *mbuf2_take_eqD*:

assumes $\text{mbuf2_take } f \ \text{buf} = (xs, \text{buf}')$
and $\text{wf_mbuf2 } i \ ja \ jb \ P \ Q \ \text{buf}$
shows $\text{wf_mbuf2 } (\text{min } ja \ jb) \ ja \ jb \ P \ Q \ \text{buf}'$
and $\text{list_all2 } (\lambda i \ z. \exists x \ y. P \ i \ x \wedge Q \ i \ y \wedge z = f \ x \ y) \ [i..<\text{min } ja \ jb] \ xs$
 ⟨proof⟩

lemma *mbuf2t_take_eqD*:

assumes $\text{mbuf2t_take } f \ z \ \text{buf} \ \text{nts} = (z', \text{buf}', \text{nts}')$
and $\text{wf_mbuf2 } i \ ja \ jb \ P \ Q \ \text{buf}$
and $\text{list_all2 } R \ [i..<j] \ \text{nts}$
and $ja \leq j \ jb \leq j$
shows $\text{wf_mbuf2 } (\text{min } ja \ jb) \ ja \ jb \ P \ Q \ \text{buf}'$
and $\text{list_all2 } R \ [\text{min } ja \ jb..<j] \ \text{nts}'$
 ⟨proof⟩

lemma *mbuf2t_take_induct[consumes 5, case_names base step]*:

assumes $\text{mbuf2t_take } f \ z \ \text{buf} \ \text{nts} = (z', \text{buf}', \text{nts}')$
and $\text{wf_mbuf2 } i \ ja \ jb \ P \ Q \ \text{buf}$

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 $U\ (min\ ja\ jb)\ z'$
 $\langle proof \rangle$

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 $list_all2\ (\lambda i.\ Z.\ \exists X\ Y.$
 $qtable\ n\ (MFOTL.fv\ \varphi)\ (mem_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \varphi)\ X \wedge$
 $qtable\ n\ (MFOTL.fv\ \psi)\ (mem_restr\ R)\ (\lambda v.\ MFOTL.sat\ \sigma\ (map\ the\ v)\ i\ \psi)\ Y \wedge$
 $Z = f\ X\ Y)$
 $[min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)..<min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j')]\ zs$
 $\langle proof \rangle$

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 $wf_ts\ \sigma\ j'\ \varphi\ \psi\ nts'$
 $\langle proof \rangle$

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 $U\ (min\ (progress\ \sigma\ \varphi\ j')\ (progress\ \sigma\ \psi\ j'))\ z'$
 $\langle proof \rangle$

lemma $progress_Until_le: progress\ \sigma\ (formula.Until\ \varphi\ I\ \psi)\ j \leq min\ (progress\ \sigma\ \varphi\ j)\ (progress\ \sigma\ \psi\ j)$
 $\langle proof \rangle$

lemma *list_all2_upt_Cons*: $P a x \implies \text{list_all2 } P [\text{Suc } a..<b] xs \implies \text{Suc } a \leq b \implies \text{list_all2 } P [a..<b] (x \# xs)$
 ⟨proof⟩

lemma *list_all2_upt_append*: $\text{list_all2 } P [a..<b] xs \implies \text{list_all2 } P [b..<c] ys \implies a \leq b \implies b \leq c \implies \text{list_all2 } P [a..<c] (xs @ ys)$
 ⟨proof⟩

lemma *meval*:

assumes *wf_mformula* $\sigma j n R \varphi \varphi'$
shows *case meval* $n (\tau \sigma j) (\Gamma \sigma j) \varphi$ of $(xs, \varphi_n) \Rightarrow \text{wf_mformula } \sigma (\text{Suc } j) n R \varphi_n \varphi' \wedge \text{list_all2 } (\lambda i. \text{qtable } n (\text{MFOTL.fv } \varphi') (\text{mem_restr } R) (\lambda v. \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi')) [\text{progress } \sigma \varphi' j..<\text{progress } \sigma \varphi' (\text{Suc } j)] xs$
 ⟨proof⟩

6.5.4 Monitor step

lemma *wf_mstate_mstep*: $\text{wf_mstate } \varphi \pi R st \implies \text{last_ts } \pi \leq \text{snd tdb} \implies \text{wf_mstate } \varphi (\text{psnoc } \pi \text{ tdb}) R (\text{snd } (\text{mstep tdb } st))$
 ⟨proof⟩

lemma *mstep_output_iff*:

assumes *wf_mstate* $\varphi \pi R st \text{last_ts } \pi \leq \text{snd tdb}$ *prefix_of* $(\text{psnoc } \pi \text{ tdb}) \sigma \text{mem_restr } R v$
shows $(i, v) \in \text{fst } (\text{mstep tdb } st) \iff \text{progress } \sigma \varphi (\text{plen } \pi) \leq i \wedge i < \text{progress } \sigma \varphi (\text{Suc } (\text{plen } \pi)) \wedge \text{wf_tuple } (\text{MFOTL.nfv } \varphi) (\text{MFOTL.fv } \varphi) v \wedge \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi$
 ⟨proof⟩

6.5.5 Monitor function

definition *minit_safe* **where**

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

lemma *minit_safe_minit*: $\text{mmonitorable } \varphi \implies \text{minit_safe } \varphi = \text{minit } \varphi$
 ⟨proof⟩

lemma (in *monitorable_mfotl*) *mstep_mverdicts*:

assumes *wf*: *wf_mstate* $\varphi \pi R st$
and *le[simp]*: $\text{last_ts } \pi \leq \text{snd tdb}$
and *restrict*: *mem_restr* $R v$
shows $(i, v) \in \text{fst } (\text{mstep tdb } st) \iff (i, v) \in M (\text{psnoc } \pi \text{ tdb}) - M \pi$
 ⟨proof⟩

primrec *msteps0* **where**

msteps0 $\square st = (\{\}, st)$
 | *msteps0* $(\text{tdb } \# \pi) st = (\text{let } (V', st') = \text{mstep tdb } st; (V'', st'') = \text{msteps0 } \pi st' \text{ in } (V' \cup V'', st''))$

primrec *msteps0_stateless* **where**

msteps0_stateless $\square st = \{\}$
 | *msteps0_stateless* $(\text{tdb } \# \pi) st = (\text{let } (V', st') = \text{mstep tdb } st \text{ in } V' \cup \text{msteps0_stateless } \pi st')$

lemma *msteps0_msteps0_stateless*: $\text{fst } (\text{msteps0 } w st) = \text{msteps0_stateless } w st$
 ⟨proof⟩

lift_definition *msteps* :: $'a \text{MFOTL.prefix} \Rightarrow 'a \text{mstate} \Rightarrow (\text{nat} \times 'a \text{option list}) \text{set} \times 'a \text{mstate}$
is *msteps0* ⟨proof⟩

lift_definition *msteps_stateless* :: $'a \text{MFOTL.prefix} \Rightarrow 'a \text{mstate} \Rightarrow (\text{nat} \times 'a \text{option list}) \text{set}$

is *msteps0_stateless* $\langle \text{proof} \rangle$

lemma *msteps_msteps_stateless*: $\text{fst} (\text{msteps } w \text{ st}) = \text{msteps_stateless } w \text{ st}$
 $\langle \text{proof} \rangle$

lemma *msteps0_snoc*: $\text{msteps0} (\pi @ [\text{tdb}]) \text{ st} =$
 $(\text{let } (V', \text{st}') = \text{msteps0 } \pi \text{ st}; (V'', \text{st}'') = \text{mstep tdb st' in } (V' \cup V'', \text{st}''))$
 $\langle \text{proof} \rangle$

lemma *msteps_psnoc*: $\text{last_ts } \pi \leq \text{snd tdb} \implies \text{msteps} (\text{psnoc } \pi \text{ tdb}) \text{ st} =$
 $(\text{let } (V', \text{st}') = \text{msteps } \pi \text{ st}; (V'', \text{st}'') = \text{mstep tdb st' in } (V' \cup V'', \text{st}''))$
 $\langle \text{proof} \rangle$

definition *monitor where*

monitor $\varphi \pi = \text{msteps_stateless } \pi (\text{minit_safe } \varphi)$

lemma *Suc_length_conv_snoc*: $(\text{Suc } n = \text{length } xs) = (\exists y \text{ ys. } xs = \text{ys} @ [y] \wedge \text{length } \text{ys} = n)$
 $\langle \text{proof} \rangle$

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps*: $\text{wf_mstate } \varphi \pi R \text{ st} \implies \text{mem_restr } R v \implies \pi \leq$
 $\pi' \implies$
 $X = \text{msteps} (\text{pdrop} (\text{plen } \pi) \pi') \text{ st} \implies \text{wf_mstate } \varphi \pi' R (\text{snd } X) \wedge$
 $((i, v) \in \text{fst } X) = ((i, v) \in M \pi' - M \pi)$
 $\langle \text{proof} \rangle$

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless*:
assumes $\text{wf_mstate } \varphi \pi R \text{ st mem_restr } R v \pi \leq \pi'$
shows $(i, v) \in \text{msteps_stateless} (\text{pdrop} (\text{plen } \pi) \pi') \text{ st} \longleftrightarrow (i, v) \in M \pi' - M \pi$
 $\langle \text{proof} \rangle$

lemma (**in** *monitorable_mfotl*) *wf_mstate_msteps_stateless_UNIV*: $\text{wf_mstate } \varphi \pi \text{ UNIV st} \implies \pi \leq$
 $\pi' \implies$
 $\text{msteps_stateless} (\text{pdrop} (\text{plen } \pi) \pi') \text{ st} = M \pi' - M \pi$
 $\langle \text{proof} \rangle$

lemma (**in** *monitorable_mfotl*) *mverdicts_Nil*: $M \text{ pnil} = \{\}$
 $\langle \text{proof} \rangle$

lemma *wf_mstate_minit_safe*: $\text{mmonitorable } \varphi \implies \text{wf_mstate } \varphi \text{ pnil } R (\text{minit_safe } \varphi)$
 $\langle \text{proof} \rangle$

lemma (**in** *monitorable_mfotl*) *monitor_mverdicts*: $\text{monitor } \varphi \pi = M \pi$
 $\langle \text{proof} \rangle$

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*: $\llbracket (i, v) \in M \pi; \text{prefix_of } \pi \sigma \rrbracket \implies \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi$
- *complete_monitor*: $\llbracket \text{prefix_of } \pi \sigma; \text{wf_tuple} (\text{MFOTL.nfv } \varphi) (\text{fv } \varphi) v; \bigwedge \sigma. \text{prefix_of } \pi \sigma \rrbracket \implies \text{MFOTL.sat } \sigma (\text{map the } v) i \varphi \implies \exists \pi'. \text{prefix_of } \pi' \sigma \wedge (i, v) \in M \pi'$

- $\text{sliceable_}M: \text{mem_restr } S \ v \implies ((i, v) \in M \ (\text{pmap_}\Gamma \ (\lambda D. D \cap \text{relevant_events } \varphi \ S) \ \pi)) = ((i, v) \in M \ \pi)$
2. The executable monitor's online interface minit_safe and mstep preserves the invariant wf_mstate and produces the the verdicts according to M :
- $\text{wf_mstate_minit_safe}: \text{mmonitorable } \varphi' \implies \text{wf_mstate } \varphi' \ \text{pnil } R \ (\text{minit_safe } \varphi')$
 - $\text{wf_mstate_mstep}: \llbracket \text{wf_mstate } \varphi' \ \pi \ R \ st; \text{last_ts } \pi \leq \text{snd } tdb \rrbracket \implies \text{wf_mstate } \varphi' \ (\text{psnoc } \pi \ tdb) \ R \ (\text{snd } (\text{mstep } tdb \ st))$
 - $\text{mstep_mverdicts}: \llbracket \text{wf_mstate } \varphi \ \pi \ R \ st; \text{last_ts } \pi \leq \text{snd } tdb; \text{mem_restr } R \ v \rrbracket \implies ((i, v) \in \text{fst } (\text{mstep } tdb \ st)) = ((i, v) \in M \ (\text{psnoc } \pi \ tdb) - M \ \pi)$
3. The executable monitor's offline interface Monitor.monitor implements M :
- $\text{monitor_mverdicts}: \text{Monitor.monitor } \varphi \ \pi = M \ \pi$

end

7 Slicing framework

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

7.1 Abstract slicing

7.1.1 Definition 1

Corresponds to locale monitor defined in theory $\text{MFOTL_Monitor.Abstract_Monitor}$.

7.1.2 Definition 2

locale $\text{slicer} = \text{monitor} +$
fixes $\text{submonitor} :: 'k :: \text{finite} \Rightarrow 'a \ \text{prefix} \Rightarrow (\text{nat} \times 'b \ \text{option list}) \ \text{set}$
and $\text{splitter} :: 'a \ \text{prefix} \Rightarrow 'k \Rightarrow 'a \ \text{prefix}$
and $\text{joiner} :: ('k \Rightarrow (\text{nat} \times 'b \ \text{option list}) \ \text{set}) \Rightarrow (\text{nat} \times 'b \ \text{option list}) \ \text{set}$
assumes $\text{mono_splitter}: \pi \leq \pi' \implies \text{splitter } \pi \ k \leq \text{splitter } \pi' \ k$
and $\text{correct_slicer}: \text{joiner } (\lambda k. \text{submonitor } k \ (\text{splitter } \pi \ k)) = M \ \pi$
begin

lemmas $\text{sound_slicer} = \text{equalityD1}[\text{OF } \text{correct_slicer}]$
lemmas $\text{complete_slicer} = \text{equalityD2}[\text{OF } \text{correct_slicer}]$

end

locale $\text{self_slicer} = \text{slicer} \ \text{nfv} \ \text{fv} \ \text{sat} \ M \ \lambda _ . M \ \text{splitter} \ \text{joiner} \ \text{for} \ \text{nfv} \ \text{fv} \ \text{sat} \ M \ \text{splitter} \ \text{joiner}$

7.1.3 Definition 3

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

lift_definition $\text{splitter} :: 'a \ \text{prefix} \Rightarrow 'k \Rightarrow 'a \ \text{prefix}$ **is**
 $\lambda \pi \ k. \text{map } (\lambda (D, t). (\{e \in D. k \in \text{event_splitter } e\}, t)) \ \pi$
 $\langle \text{proof} \rangle$

7.1.4 Lemma 1

lemma *mono_splitter*: $\pi \leq \pi' \implies \text{splitter } \pi \ k \leq \text{splitter } \pi' \ k$
<proof>

end

7.2 Joint data slicer

abbreviation *(input) ok* $\varphi \ v \equiv \text{wf_tuple } (MFOTL.nfv \ \varphi) \ (MFOTL.fv \ \varphi) \ v$

locale *splitting_strategy* =
 fixes $\varphi :: 'a \ MFOTL.formula$
 and *strategy* $:: 'a \ \text{option list} \implies 'k :: \text{finite set}$
 assumes *strategy_nonempty*: $ok \ \varphi \ v \implies \text{strategy } v \neq \{\}$
begin

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

end

7.2.1 Definition 4

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

locale *joint_data_slicer* = *MFOTL_monitor* $\varphi \ M + \text{splitting_strategy } \varphi \ \text{strategy}$
 for $\varphi \ M \ \text{strategy}$
begin

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

sublocale *event_separable_splitter* **where** *event_splitter* = *event_splitter* *<proof>*

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

lemma *splitter_pslice*: $\text{splitter } \pi \ k = MFOTL_slicer.pslice \ \varphi \ (\text{slice_set } k) \ \pi$
<proof>

7.2.2 Lemma 2

Corresponds to the following theorem *sat_slice_strong* proved in theory *MFOTL_Monitor.Abstract_Monitor*:
 $\llbracket \text{relevant_events } \varphi' \ S \subseteq E; v \in S \rrbracket \implies MFOTL.sat \ \sigma \ v \ i \ \varphi' = MFOTL.sat \ (\text{map_}\Gamma \ (\lambda D. D \cap E) \ \sigma) \ v \ i \ \varphi'$

7.2.3 Theorem 1

sublocale *joint_monitor*: *MFOTL_monitor* $\varphi \ \lambda \pi. \ \text{joiner} \ (\lambda k. \ M \ (\text{splitter } \pi \ k))$
<proof>

7.2.4 Corollary 1

sublocale *joint_slicer*: *slicer* $MFOTL.nfv \ \varphi \ MFOTL.fv \ \varphi \ \lambda \sigma \ v \ i. \ MFOTL.sat \ \sigma \ v \ i \ \varphi$
 $\lambda \pi. \ \text{joiner} \ (\lambda k. \ M \ (\text{splitter } \pi \ k)) \ \lambda _ . \ M \ \text{splitter } \text{joiner}$
<proof>

end

7.2.5 Definition 5

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

```
locale sliceable_joint_data_slicer =  
  sliceable_monitor MFOTL.nfv  $\varphi$  MFOTL.fv  $\varphi$  relevant_events  $\varphi$   $\lambda\sigma v i.$  MFOTL.sat  $\sigma v i \varphi M$  +  
  joint_data_slicer  $\varphi M$  strategy for  $\varphi M$  strategy  
begin
```

```
lemma monitor_split: ok  $\varphi v \implies k \in$  strategy  $v \implies (i, v) \in M$  (splitter  $\pi k$ )  $\longleftrightarrow (i, v) \in M \pi$   
<proof>
```

7.2.6 Theorem 2

```
sublocale self_slicer MFOTL.nfv  $\varphi$  MFOTL.fv  $\varphi$   $\lambda\sigma v i.$  MFOTL.sat  $\sigma v i \varphi M$  splitter joiner  
<proof>
```

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  $\psi$ ) = names  $\psi$   
| names (MFOTL.Or  $\alpha \beta$ ) = names  $\alpha \cup$  names  $\beta$   
| names (MFOTL.Exists  $\psi$ ) = names  $\psi$   
| names (MFOTL.Prev  $I \psi$ ) = names  $\psi$   
| names (MFOTL.Next  $I \psi$ ) = names  $\psi$   
| names (MFOTL.Since  $\alpha I \beta$ ) = names  $\alpha \cup$  names  $\beta$   
| names (MFOTL.Until  $\alpha I \beta$ ) = names  $\alpha \cup$  names  $\beta$ 
```

```
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  $\psi$ ) = gen_unique  $\psi$   
| gen_unique (MFOTL.Or  $\alpha \beta$ ) = (gen_unique  $\alpha \wedge$  gen_unique  $\beta \wedge$  names  $\alpha \cap$  names  $\beta =$  {})   
| gen_unique (MFOTL.Exists  $\psi$ ) = gen_unique  $\psi$   
| gen_unique (MFOTL.Prev  $I \psi$ ) = gen_unique  $\psi$   
| gen_unique (MFOTL.Next  $I \psi$ ) = gen_unique  $\psi$   
| gen_unique (MFOTL.Since  $\alpha I \beta$ ) = (gen_unique  $\alpha \wedge$  gen_unique  $\beta \wedge$  names  $\alpha \cap$  names  $\beta =$  {})   
| gen_unique (MFOTL.Until  $\alpha I \beta$ ) = (gen_unique  $\alpha \wedge$  gen_unique  $\beta \wedge$  names  $\alpha \cap$  names  $\beta =$  {})
```

```
lemma sat_inter_names_cong: ( $\bigwedge e. e \in$  names  $\varphi \implies \{xs. (e, xs) \in E\} = \{xs. (e, xs) \in F\}$ )  $\implies$   
  MFOTL.sat (map_ $\Gamma$  ( $\lambda D. D \cap E$ )  $\sigma$ )  $v i \varphi \longleftrightarrow$  MFOTL.sat (map_ $\Gamma$  ( $\lambda D. D \cap F$ )  $\sigma$ )  $v i \varphi$   
<proof>
```

```
lemma matches_in_names: MFOTL.matches  $v \varphi x \implies$  fst  $x \in$  names  $\varphi$   
<proof>
```

```
lemma unique_names_matches_absorb: fst  $x \in$  names  $\alpha \implies$  names  $\alpha \cap$  names  $\beta =$  {}  $\implies$   
  MFOTL.matches  $v \alpha x \vee$  MFOTL.matches  $v \beta x \longleftrightarrow$  MFOTL.matches  $v \alpha x$   
fst  $x \in$  names  $\beta \implies$  names  $\alpha \cap$  names  $\beta =$  {}  $\implies$   
  MFOTL.matches  $v \alpha x \vee$  MFOTL.matches  $v \beta x \longleftrightarrow$  MFOTL.matches  $v \beta x$   
<proof>
```

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

<proof>

lemma *in_listset_nth*: $x \in \text{listset } As \implies i < \text{length } As \implies x ! i \in As ! i$

<proof>

lemma *all_nth_in_listset*: $\text{length } x = \text{length } As \implies (\bigwedge i. i < \text{length } As \implies x ! i \in As ! i) \implies x \in \text{listset } As$

<proof>

lemma *mergeable_envs_listset*: *mergeable_envs* $(\text{length } As)$ $(\text{listset } As)$

<proof>

lemma *mergeable_envs_Exists*: *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>

lemma *in_set_ConsE*: $xs \in \text{set_Cons } A \ As \implies (\bigwedge y \ ys. xs = y \# \ ys \implies y \in A \implies ys \in As \implies P)$

$\implies P$

<proof>

lemma *mergeable_envs_set_Cons*: *mergeable_envs* n $S \implies \text{mergeable_envs } (\text{Suc } n)$ $(\text{set_Cons UNIV } S)$

<proof>

lemma *slice_Exists*: $\text{MFOTL.slicer.slice } (\text{MFOTL.Exists } \varphi) \ S \ \sigma = \text{MFOTL.slicer.slice } \varphi \ (\text{set_Cons UNIV } S) \ \sigma$

<proof>

lemma *image_Suc_fvi*: $\text{Suc } \text{' MFOTL.fvi } (\text{Suc } b) \ \varphi = \text{MFOTL.fvi } b \ \varphi - \{0\}$

<proof>

lemma *nfv_Exists*: $\text{MFOTL.nfv } (\text{MFOTL.Exists } \varphi) = \text{MFOTL.nfv } \varphi - 1$

<proof>

lemma *set_Cons_empty_iff[simp]*: $\text{set_Cons } A \ Xs = \{\} \longleftrightarrow A = \{\} \vee Xs = \{\}$

<proof>

lemma *unique_sat_slice_mem*: $\text{safe_formula } \varphi \implies \text{gen_unique } \varphi \implies S \neq \{\} \implies$

$\text{mergeable_envs } n \ S \implies \text{MFOTL.nfv } \varphi \leq n \implies$

$\text{MFOTL.sat } (\text{MFOTL.slicer.slice } \varphi \ S \ \sigma) \ v \ i \ \varphi \implies \exists v' \in S. \forall x \in \text{fv } \varphi. v' ! x = v ! x$

<proof>

lemma *unique_sat_slice*:

assumes *formula*: $\text{safe_formula } \varphi \ \text{gen_unique } \varphi$

and *restr*: $S \neq \{\} \ \text{mergeable_envs } (\text{MFOTL.nfv } \varphi) \ S$

and *sat_slice*: $\text{MFOTL.sat } (\text{MFOTL.slicer.slice } \varphi \ S \ \sigma) \ v \ i \ \varphi$

shows $\text{MFOTL.sat } \sigma \ v \ i \ \varphi$

<proof>

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 φ) (slice_set k) \implies
MFOTL.sat (MFOTL.slicer.slice φ (slice_set k) σ) (map the v) $i \varphi \implies$
ok $\varphi v \implies k \in$ strategy v
<proof>

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

7.2.9 Definition of J'

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

7.2.10 Theorem 3

lemma skip_joiner:
assumes safe_formula φ gen_unique φ
shows joiner ($\lambda k. M$ (splitter πk)) = skip_joiner ($\lambda k. M$ (splitter πk))
(is ?L = ?R)
<proof>

sublocale skip_joint_monitor: MFOTL_monitor φ
 $\lambda \pi. (if$ safe_formula $\varphi \wedge$ gen_unique φ $then$ skip_joiner $else$ joiner) ($\lambda k. M$ (splitter πk))
<proof>

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.