Multi-Head Monitoring of Metric Dynamic Logic

Martin Raszyk

March 17, 2025

Abstract

Runtime monitoring (or runtime verification) is an approach to checking compliance of a system's execution with a specification (e.g., a temporal formula). The system's execution is logged into a *trace*—a sequence of time-points, each consisting of a time-stamp and observed events. A *monitor* is an algorithm that produces *verdicts* on the satisfaction of a temporal formula on a trace.

We formalize the time-stamps as an abstract algebraic structure satisfying certain assumptions. Instances of this structure include natural numbers, real numbers, and lexicographic combinations of them. We also include the formalization of a conversion from the abstract time domain introduced by Koymans [1] to our time-stamps.

We formalize a monitoring algorithm for metric dynamic logic, an extension of metric temporal logic with regular expressions. The monitor computes whether a given formula is satisfied at every position in an input trace of time-stamped events. Our monitor follows the multi-head paradigm: it reads the input simultaneously at multiple positions and moves its reading heads asynchronously. This mode of operation results in unprecedented time and space complexity guarantees for metric dynamic logic: The monitor's amortized time complexity to process a time-point and the monitor's space complexity neither depends on the event-rate, i.e., the number of events within a fixed time-unit, nor on the numeric constants occurring in the quantitative temporal constraints in the given formula.

The multi-head monitoring algorithm for metric dynamic logic is reported in our paper "Multi-Head Monitoring of Metric Dynamic Logic" [2] published at ATVA 2020. We have also formalized unpublished specialized algorithms for the temporal operators of metric temporal logic.

Contents

theory *Timestamp*
imports *HOL−Library.Extended_Nat HOL−Library.Extended_Real*
begin

class *embed_nat* =
fixes $\iota :: nat \Rightarrow 'a$

class *tfin* =
fixes *tfin* :: $'a\ set$

class *timestamp* = *comm_monoid_add* + *semilattice_sup* + *embed_nat* + *tfin* +
assumes *ι_mono*: $\bigwedge i\ j.\ i \leq j \Longrightarrow \iota\ i \leq \iota\ j$

and ι_tfin: $\bigwedge i.\ \iota\ i \in tfin$
and $\iota\_progressing$: $x \in tfin \Longrightarrow \exists j.\ \neg \iota\ j \leq \iota\ i + x$
and $zero\_tfin$: $0 \in tfin$
and $tfin\_closed$: $c \in tfin \Longrightarrow d \in tfin \Longrightarrow c + d \in tfin$
and add_mono: $c \leq d \Longrightarrow a + c \leq a + d$
and add_pos: $a \in tfin \Longrightarrow 0 < c \Longrightarrow a < a + c$
begin

lemma add_mono_comm:
fixes $a :: 'a$
shows $c \leq d \Longrightarrow c + a \leq d + a$
⟨*proof*⟩

end

instantiation $option :: (timestamp)\ timestamp$
begin

definition $tfin\_option :: 'a\ option\ set$ **where**
$tfin\_option = Some\ `\ tfin$

definition $\iota\_option :: nat \Rightarrow 'a\ option$ **where**
$\iota\_option = Some \circ \iota$

definition $zero\_option :: 'a\ option$ **where**
$zero\_option = Some\ 0$

definition $plus\_option :: 'a\ option \Rightarrow 'a\ option \Rightarrow 'a\ option$ **where**
$plus\_option\ x\ y = (case\ x\ of\ None \Rightarrow None \mid Some\ x' \Rightarrow (case\ y\ of\ None \Rightarrow None \mid Some\ y' \Rightarrow Some\ (x' + y')))$

definition $sup\_option :: 'a\ option \Rightarrow 'a\ option \Rightarrow 'a\ option$ **where**
$sup\_option\ x\ y = (case\ x\ of\ None \Rightarrow None \mid Some\ x' \Rightarrow (case\ y\ of\ None \Rightarrow None \mid Some\ y' \Rightarrow Some\ (sup\ x'\ y')))$

definition $less\_option :: 'a\ option \Rightarrow 'a\ option \Rightarrow bool$ **where**
$less\_option\ x\ y = (case\ x\ of\ None \Rightarrow False \mid Some\ x' \Rightarrow (case\ y\ of\ None \Rightarrow True \mid Some\ y' \Rightarrow x' < y'))$

definition $less\_eq\_option :: 'a\ option \Rightarrow 'a\ option \Rightarrow bool$ **where**
$less\_eq\_option\ x\ y = (case\ x\ of\ None \Rightarrow x = y \mid Some\ x' \Rightarrow (case\ y\ of\ None \Rightarrow True \mid Some\ y' \Rightarrow x' \leq y'))$

instance
⟨*proof*⟩

end

instantiation $enat :: timestamp$
begin

definition $tfin\_enat :: enat\ set$ **where**
$tfin\_enat = UNIV - \{\infty\}$

definition $\iota\_enat :: nat \Rightarrow enat$ **where**
$\iota\_enat\ n = n$

instance
⟨*proof*⟩

end

instantiation *ereal* :: *timestamp*
begin

definition *ι_ereal* :: *nat* ⇒ *ereal* **where**
ι_ereal n = *ereal n*

definition *tfin_ereal* :: *ereal set* **where**
tfin_ereal = *UNIV* − {−∞, ∞}

lemma *ereal_add_pos*:
fixes *a* :: *ereal*
shows *a* ∈ *tfin* ⟹ *0* < *c* ⟹ *a* < *a* + *c*
⟨*proof*⟩

instance
⟨*proof*⟩

end

class *timestamp_total* = *timestamp* +
assumes *timestamp_total*: *a* ≤ *b* ∨ *b* ≤ *a*
assumes *timestamp_tfin_le_not_tfin*: *0* ≤ *a* ⟹ *a* ∈ *tfin* ⟹ *0* ≤ *b* ⟹ *b* ∉ *tfin* ⟹ *a* ≤ *b*
begin

lemma *add_not_tfin*: *0* ≤ *a* ⟹ *a* ∈ *tfin* ⟹ *a* ≤ *c* ⟹ *c* ∈ *tfin* ⟹ *0* ≤ *b* ⟹ *b* ∉ *tfin* ⟹ *c* < *a* + *b*
⟨*proof*⟩

end

instantiation *enat* :: *timestamp_total*
begin

instance
⟨*proof*⟩

end

instantiation *ereal* :: *timestamp_total*
begin

instance
⟨*proof*⟩

end

class *timestamp_strict* = *timestamp* +
assumes *add_mono_strict*: *c* < *d* ⟹ *a* + *c* < *a* + *d*

class *timestamp_total_strict* = *timestamp_total* + *timestamp_strict*

instantiation *nat* :: *timestamp_total_strict*
begin

definition *tfin_nat* :: *nat set* **where**
tfin_nat = *UNIV*

definition *ι_nat* :: *nat* ⇒ *nat* **where**
ι_nat n = *n*

instance
⟨*proof*⟩

end

instantiation *real* :: *timestamp_total_strict*
begin

definition *tfin_real* :: *real set* **where** *tfin_real* = *UNIV*

definition *ι_real* :: *nat* ⇒ *real* **where** *ι_real n* = *real n*

instance
⟨*proof*⟩

end

instantiation *prod* :: (*comm_monoid_add*, *comm_monoid_add*) *comm_monoid_add*
begin

definition *zero_prod* :: *'a* × *'b* **where**
zero_prod = (*0*, *0*)

fun *plus_prod* :: *'a* × *'b* ⇒ *'a* × *'b* ⇒ *'a* × *'b* **where**
(*a*, *b*) + (*c*, *d*) = (*a* + *c*, *b* + *d*)

instance
⟨*proof*⟩

end

end

1 Intervals

typedef (**overloaded**) (*'a* :: *timestamp*) *ℐ* = {(*i* :: *'a*, *j* :: *'a*, *lei* :: *bool*, *lej* :: *bool*). *0* ≤ *i* ∧ *i* ≤ *j* ∧ *i* ∈ *tfin* ∧ ¬(*j* = *0* ∧ ¬*lej*)}
⟨*proof*⟩

setup_lifting *type_definition_ℐ*

instantiation *ℐ* :: (*timestamp*) *equal* **begin**

lift_definition *equal_ℐ* :: *'a ℐ* ⇒ *'a ℐ* ⇒ *bool* **is** (=) ⟨*proof*⟩

instance ⟨*proof*⟩

end

lift_definition *right* :: *'a* :: *timestamp ℐ* ⇒ *'a* **is** *fst* ∘ *snd* ⟨*proof*⟩

lift_definition *memL* :: *'a* :: *timestamp* ⇒ *'a* ⇒ *'a ℐ* ⇒ *bool* **is**

$\lambda t\ t'\ (a,\ b,\ lei,\ lej).\ \mathit{if}\ lei\ \mathit{then}\ t + a \leq t'\ \mathit{else}\ t + a < t'\ \langle proof \rangle$

lift_definition $memR :: 'a :: timestamp \Rightarrow 'a \Rightarrow 'a\ \mathcal{I} \Rightarrow bool$ **is**
$\lambda t\ t'\ (a,\ b,\ lei,\ lej).\ \mathit{if}\ lej\ \mathit{then}\ t' \leq t + b\ \mathit{else}\ t' < t + b\ \langle proof \rangle$

definition $mem :: 'a :: timestamp \Rightarrow 'a \Rightarrow 'a\ \mathcal{I} \Rightarrow bool$ **where**
$mem\ t\ t'\ I \longleftrightarrow memL\ t\ t'\ I \wedge memR\ t\ t'\ I$

lemma *memL_mono*: $memL\ t\ t'\ I \Longrightarrow t'' \leq t \Longrightarrow memL\ t''\ t'\ I$
$\langle proof \rangle$

lemma *memL_mono'*: $memL\ t\ t'\ I \Longrightarrow t' \leq t'' \Longrightarrow memL\ t\ t''\ I$
$\langle proof \rangle$

lemma *memR_mono*: $memR\ t\ t'\ I \Longrightarrow t \leq t'' \Longrightarrow memR\ t''\ t'\ I$
$\langle proof \rangle$

lemma *memR_mono'*: $memR\ t\ t'\ I \Longrightarrow t'' \leq t' \Longrightarrow memR\ t\ t''\ I$
$\langle proof \rangle$

lemma *memR_dest*: $memR\ t\ t'\ I \Longrightarrow t' \leq t + right\ I$
$\langle proof \rangle$

lemma *memR_tfin_refl*:
assumes *fin*: $t \in tfin$
shows $memR\ t\ t\ I$
$\langle proof \rangle$

lemma *right_I_add_mono*:
fixes $x :: 'a :: timestamp$
shows $x \leq x + right\ I$
$\langle proof \rangle$

lift_definition $interval :: 'a :: timestamp \Rightarrow 'a \Rightarrow bool \Rightarrow bool \Rightarrow 'a\ \mathcal{I}$ **is**
$\lambda i\ j\ lei\ lej.\ (\mathit{if}\ 0 \leq i \wedge i \leq j \wedge i \in tfin \wedge \neg(j = 0 \wedge \neg lej)\mathit{then}\ (i,\ j,\ lei,\ lej)\ \mathit{else}\ Code.abort\ (STR\ ''malformed\ interval'')\ (\lambda\_.\ (0,\ 0,\ True,\ True)))$
$\langle proof \rangle$

lemma $Rep\_\mathcal{I}\ I = (l,\ r,\ b1,\ b2) \Longrightarrow memL\ 0\ 0\ I \longleftrightarrow l = 0 \wedge b1$
$\langle proof \rangle$

lift_definition $dropL :: 'a :: timestamp\ \mathcal{I} \Rightarrow 'a\ \mathcal{I}$ **is**
$\lambda(l,\ r,\ b1,\ b2).\ (0,\ r,\ True,\ b2)$
$\langle proof \rangle$

lemma *memL_dropL*: $t \leq t' \Longrightarrow memL\ t\ t'\ (dropL\ I)$
$\langle proof \rangle$

lemma *memR_dropL*: $memR\ t\ t'\ (dropL\ I) = memR\ t\ t'\ I$
$\langle proof \rangle$

lift_definition $flipL :: 'a :: timestamp\ \mathcal{I} \Rightarrow 'a\ \mathcal{I}$ **is**
$\lambda(l,\ r,\ b1,\ b2).\ \mathit{if}\ \neg(l = 0 \wedge b1)\ \mathit{then}\ (0,\ l,\ True,\ \neg b1)\ \mathit{else}\ Code.abort\ (STR\ ''invalid\ flipL'')\ (\lambda\_.\ (0,\ 0,\ True,\ True))$
$\langle proof \rangle$

lemma *memL_flipL*: $t \leq t' \Longrightarrow memL\ t\ t'\ (flipL\ I)$
$\langle proof \rangle$

lemma *memR_flipLD*: $\neg memL\ 0\ 0\ I \Longrightarrow memR\ t\ t'\ (flipL\ I) \Longrightarrow \neg memL\ t\ t'\ I$
⟨*proof*⟩

lemma *memR_flipLI*:
fixes $t :: 'a :: timestamp$
shows $(\bigwedge u\ v.\ (u :: 'a :: timestamp) \leq v \vee v \leq u) \Longrightarrow \neg memL\ t\ t'\ I \Longrightarrow memR\ t\ t'\ (flipL\ I)$
⟨*proof*⟩

lemma $t \in tfin \Longrightarrow memL\ 0\ 0\ I \longleftrightarrow memL\ t\ t\ I$
⟨*proof*⟩

definition $full\ (I :: ('a :: timestamp)\ \mathcal{I}) \longleftrightarrow (\forall t\ t'.\ 0 \leq t \wedge t \leq t' \wedge t \in tfin \wedge t' \in tfin \longrightarrow mem\ t\ t'\ I)$

lemma $memL\ 0\ 0\ (I :: ('a :: timestamp\_total)\ \mathcal{I}) \Longrightarrow right\ I \notin tfin \Longrightarrow full\ I$
⟨*proof*⟩

2 Infinite Traces

inductive $sorted\_list :: 'a :: order\ list \Rightarrow bool$ **where**
[*intro*]: $sorted\_list\ []$
| [*intro*]: $sorted\_list\ [x]$
| [*intro*]: $x \leq y \Longrightarrow sorted\_list\ (y\ \#\ ys) \Longrightarrow sorted\_list\ (x\ \#\ y\ \#\ ys)$

lemma *sorted_list_app*: $sorted\_list\ xs \Longrightarrow (\bigwedge x.\ x \in set\ xs \Longrightarrow x \leq y) \Longrightarrow sorted\_list\ (xs\ @\ [y])$
⟨*proof*⟩

lemma *sorted_list_drop*: $sorted\_list\ xs \Longrightarrow sorted\_list\ (drop\ n\ xs)$
⟨*proof*⟩

lemma *sorted_list_ConsD*: $sorted\_list\ (x\ \#\ xs) \Longrightarrow sorted\_list\ xs$
⟨*proof*⟩

lemma *sorted_list_Cons_nth*: $sorted\_list\ (x\ \#\ xs) \Longrightarrow j < length\ xs \Longrightarrow x \leq xs\ !\ j$
⟨*proof*⟩

lemma *sorted_list_atD*: $sorted\_list\ xs \Longrightarrow i \leq j \Longrightarrow j < length\ xs \Longrightarrow xs\ !\ i \leq xs\ !\ j$
⟨*proof*⟩

coinductive $ssorted :: 'a :: order\ stream \Rightarrow bool$ **where**
$shd\ s \leq shd\ (stl\ s) \Longrightarrow ssorted\ (stl\ s) \Longrightarrow ssorted\ s$

lemma *ssorted_siterate*[*simp*]: $(\bigwedge n.\ n \leq f\ n) \Longrightarrow ssorted\ (siterate\ f\ n)$
⟨*proof*⟩

lemma *ssortedD*: $ssorted\ s \Longrightarrow s\ !!\ i \leq stl\ s\ !!\ i$
⟨*proof*⟩

lemma *ssorted_sdrop*: $ssorted\ s \Longrightarrow ssorted\ (sdrop\ i\ s)$
⟨*proof*⟩

lemma *ssorted_monoD*: $ssorted\ s \Longrightarrow i \leq j \Longrightarrow s\ !!\ i \leq s\ !!\ j$
⟨*proof*⟩

lemma *sorted_stake*: $ssorted\ s \Longrightarrow sorted\_list\ (stake\ i\ s)$
⟨*proof*⟩

lemma *ssorted_monoI*: $\forall i\ j.\ i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j \Longrightarrow ssorted\ s$
⟨*proof*⟩

lemma *ssorted_iff_mono*: $ssorted\ s \longleftrightarrow (\forall i\ j.\ i \leq j \longrightarrow s\ !!\ i \leq s\ !!\ j)$
⟨*proof*⟩

typedef (**overloaded**) $('a, 'b :: timestamp)\ trace = \{s :: ('a\ set \times 'b)\ stream.$
$ssorted\ (smap\ snd\ s) \wedge (\forall x.\ x \in snd\ `\ sset\ s \longrightarrow x \in tfin) \wedge (\forall i\ x.\ x \in tfin \longrightarrow (\exists j.\ \neg snd\ (s\ !!\ j) \leq snd\ (s\ !!\ i) + x))\}$
⟨*proof*⟩

setup_lifting *type_definition_trace*

lift_definition $\Gamma :: ('a, 'b :: timestamp)\ trace \Rightarrow nat \Rightarrow 'a\ set$ **is**
$\lambda s\ i.\ fst\ (s\ !!\ i)$ ⟨*proof*⟩
lift_definition $\tau :: ('a, 'b :: timestamp)\ trace \Rightarrow nat \Rightarrow 'b$ **is**
$\lambda s\ i.\ snd\ (s\ !!\ i)$ ⟨*proof*⟩

lemma *τ_mono*[*simp*]: $i \leq j \Longrightarrow \tau\ s\ i \leq \tau\ s\ j$
⟨*proof*⟩

lemma *τ_fin*: $\tau\ \sigma\ i \in tfin$
⟨*proof*⟩

lemma *ex_lt_τ*: $x \in tfin \Longrightarrow \exists j.\ \neg \tau\ s\ j \leq \tau\ s\ i + x$
⟨*proof*⟩

lemma *le_τ_less*: $\tau\ \sigma\ i \leq \tau\ \sigma\ j \Longrightarrow j < i \Longrightarrow \tau\ \sigma\ i = \tau\ \sigma\ j$
⟨*proof*⟩

lemma *less_τD*: $\tau\ \sigma\ i < \tau\ \sigma\ j \Longrightarrow i < j$
⟨*proof*⟩
theory *MDL*
imports *Interval Trace*
begin

3 Formulas and Satisfiability

declare [[*typedef_overloaded*]]

datatype $('a, 't :: timestamp)\ formula = Bool\ bool \mid Atom\ 'a \mid Neg\ ('a, 't)\ formula \mid$
$Bin\ bool \Rightarrow bool \Rightarrow bool\ ('a, 't)\ formula\ ('a, 't)\ formula \mid$
$Prev\ 't\ \mathcal{I}\ ('a, 't)\ formula \mid Next\ 't\ \mathcal{I}\ ('a, 't)\ formula \mid$
$Since\ ('a, 't)\ formula\ 't\ \mathcal{I}\ ('a, 't)\ formula \mid$
$Until\ ('a, 't)\ formula\ 't\ \mathcal{I}\ ('a, 't)\ formula \mid$
$MatchP\ 't\ \mathcal{I}\ ('a, 't)\ regex \mid MatchF\ 't\ \mathcal{I}\ ('a, 't)\ regex$
and $('a, 't)\ regex = Lookahead\ ('a, 't)\ formula \mid Symbol\ ('a, 't)\ formula \mid$
$Plus\ ('a, 't)\ regex\ ('a, 't)\ regex \mid Times\ ('a, 't)\ regex\ ('a, 't)\ regex \mid$
$Star\ ('a, 't)\ regex$

fun $eps :: ('a, 't :: timestamp)\ regex \Rightarrow bool$ **where**
$eps\ (Lookahead\ phi) = True$
$\mid eps\ (Symbol\ phi) = False$
$\mid eps\ (Plus\ r\ s) = (eps\ r \vee eps\ s)$
$\mid eps\ (Times\ r\ s) = (eps\ r \wedge eps\ s)$
$\mid eps\ (Star\ r) = True$

fun $atms :: ('a, 't :: timestamp)\ regex \Rightarrow ('a, 't)\ formula\ set$ **where**

```
  atms (Lookahead phi) = {phi}
| atms (Symbol phi) = {phi}
| atms (Plus r s) = atms r ∪ atms s
| atms (Times r s) = atms r ∪ atms s
| atms (Star r) = atms r

lemma size_atms[termination_simp]: phi ∈ atms r ⟹ size phi < size r
  ⟨proof⟩

fun wf_fmla :: ('a, 't :: timestamp) formula ⇒ bool
  and wf_regex :: ('a, 't) regex ⇒ bool where
  wf_fmla (Bool b) = True
| wf_fmla (Atom a) = True
| wf_fmla (Neg phi) = wf_fmla phi
| wf_fmla (Bin f phi psi) = (wf_fmla phi ∧ wf_fmla psi)
| wf_fmla (Prev I phi) = wf_fmla phi
| wf_fmla (Next I phi) = wf_fmla phi
| wf_fmla (Since phi I psi) = (wf_fmla phi ∧ wf_fmla psi)
| wf_fmla (Until phi I psi) = (wf_fmla phi ∧ wf_fmla psi)
| wf_fmla (MatchP I r) = (wf_regex r ∧ (∀ phi ∈ atms r. wf_fmla phi))
| wf_fmla (MatchF I r) = (wf_regex r ∧ (∀ phi ∈ atms r. wf_fmla phi))
| wf_regex (Lookahead phi) = False
| wf_regex (Symbol phi) = wf_fmla phi
| wf_regex (Plus r s) = (wf_regex r ∧ wf_regex s)
| wf_regex (Times r s) = (wf_regex s ∧ (¬eps s ∨ wf_regex r))
| wf_regex (Star r) = wf_regex r

fun progress :: ('a, 't :: timestamp) formula ⇒ 't list ⇒ nat where
  progress (Bool b) ts = length ts
| progress (Atom a) ts = length ts
| progress (Neg phi) ts = progress phi ts
| progress (Bin f phi psi) ts = min (progress phi ts) (progress psi ts)
| progress (Prev I phi) ts = min (length ts) (Suc (progress phi ts))
| progress (Next I phi) ts = (case progress phi ts of 0 ⇒ 0 | Suc k ⇒ k)
| progress (Since phi I psi) ts = min (progress phi ts) (progress psi ts)
| progress (Until phi I psi) ts = (if length ts = 0 then 0 else
  (let k = min (length ts − 1) (min (progress phi ts) (progress psi ts)) in
   Min {j. 0 ≤ j ∧ j ≤ k ∧ memR (ts ! j) (ts ! k) I}))
| progress (MatchP I r) ts = Min ((λf. progress f ts) ‘ atms r)
| progress (MatchF I r) ts = (if length ts = 0 then 0 else
  (let k = min (length ts − 1) (Min ((λf. progress f ts) ‘ atms r)) in
   Min {j. 0 ≤ j ∧ j ≤ k ∧ memR (ts ! j) (ts ! k) I}))

fun bounded_future_fmla :: ('a, 't :: timestamp) formula ⇒ bool
  and bounded_future_regex :: ('a, 't) regex ⇒ bool where
  bounded_future_fmla (Bool b) ⟷ True
| bounded_future_fmla (Atom a) ⟷ True
| bounded_future_fmla (Neg phi) ⟷ bounded_future_fmla phi
| bounded_future_fmla (Bin f phi psi) ⟷ bounded_future_fmla phi ∧ bounded_future_fmla psi
| bounded_future_fmla (Prev I phi) ⟷ bounded_future_fmla phi
| bounded_future_fmla (Next I phi) ⟷ bounded_future_fmla phi
| bounded_future_fmla (Since phi I psi) ⟷ bounded_future_fmla phi ∧ bounded_future_fmla psi
| bounded_future_fmla (Until phi I psi) ⟷ bounded_future_fmla phi ∧ bounded_future_fmla psi ∧
right I ∈ tfin
| bounded_future_fmla (MatchP I r) ⟷ bounded_future_regex r
| bounded_future_fmla (MatchF I r) ⟷ bounded_future_regex r ∧ right I ∈ tfin
| bounded_future_regex (Lookahead phi) ⟷ bounded_future_fmla phi
| bounded_future_regex (Symbol phi) ⟷ bounded_future_fmla phi
```

| *bounded_future_regex* (*Plus r s*) ⟷ *bounded_future_regex r* ∧ *bounded_future_regex s*
| *bounded_future_regex* (*Times r s*) ⟷ *bounded_future_regex r* ∧ *bounded_future_regex s*
| *bounded_future_regex* (*Star r*) ⟷ *bounded_future_regex r*

lemmas *regex_induct*[*case_names Lookahead Symbol Plus Times Star*, *induct type*: *regex*] =
regex.induct[*of* λ_. *True*, *simplified*]

definition *Once I φ* ≡ *Since* (*Bool True*) *I φ*
definition *Historically I φ* ≡ *Neg* (*Once I* (*Neg φ*))
definition *Eventually I φ* ≡ *Until* (*Bool True*) *I φ*
definition *Always I φ* ≡ *Neg* (*Eventually I* (*Neg φ*))

fun *rderive* :: (′*a*, ′*t* :: *timestamp*) *regex* ⇒ (′*a*, ′*t*) *regex* **where**
rderive (*Lookahead phi*) = *Lookahead* (*Bool False*)
| *rderive* (*Symbol phi*) = *Lookahead phi*
| *rderive* (*Plus r s*) = *Plus* (*rderive r*) (*rderive s*)
| *rderive* (*Times r s*) = (*if eps s then Plus* (*rderive r*) (*Times r* (*rderive s*)) *else Times r* (*rderive s*))
| *rderive* (*Star r*) = *Times* (*Star r*) (*rderive r*)

lemma *atms_rderive*: *phi* ∈ *atms* (*rderive r*) ⟹ *phi* ∈ *atms r* ∨ *phi* = *Bool False*
⟨*proof*⟩

lemma *size_formula_positive*: *size* (*phi* :: (′*a*, ′*t* :: *timestamp*) *formula*) > *0*
⟨*proof*⟩

lemma *size_regex_positive*: *size* (*r* :: (′*a*, ′*t* :: *timestamp*) *regex*) > *Suc 0*
⟨*proof*⟩

lemma *size_rderive*[*termination_simp*]: *phi* ∈ *atms* (*rderive r*) ⟹ *size phi* < *size r*
⟨*proof*⟩

locale *MDL* =
fixes σ :: (′*a*, ′*t* :: *timestamp*) *trace*
begin

fun *sat* :: (′*a*, ′*t*) *formula* ⇒ *nat* ⇒ *bool*
and *match* :: (′*a*, ′*t*) *regex* ⇒ (*nat* × *nat*) *set* **where**
sat (*Bool b*) *i* = *b*
| *sat* (*Atom a*) *i* = (*a* ∈ Γ σ *i*)
| *sat* (*Neg φ*) *i* = (¬ *sat φ i*)
| *sat* (*Bin f φ ψ*) *i* = (*f* (*sat φ i*) (*sat ψ i*))
| *sat* (*Prev I φ*) *i* = (*case i of 0* ⇒ *False* | *Suc j* ⇒ *mem* (τ σ *j*) (τ σ *i*) *I* ∧ *sat φ j*)
| *sat* (*Next I φ*) *i* = (*mem* (τ σ *i*) (τ σ (*Suc i*)) *I* ∧ *sat φ* (*Suc i*))
| *sat* (*Since φ I ψ*) *i* = (∃ *j*≤*i*. *mem* (τ σ *j*) (τ σ *i*) *I* ∧ *sat ψ j* ∧ (∀ *k* ∈ {*j*<..*i*}. *sat φ k*))
| *sat* (*Until φ I ψ*) *i* = (∃ *j*≥*i*. *mem* (τ σ *i*) (τ σ *j*) *I* ∧ *sat ψ j* ∧ (∀ *k* ∈ {*i*..<*j*}. *sat φ k*))
| *sat* (*MatchP I r*) *i* = (∃ *j*≤*i*. *mem* (τ σ *j*) (τ σ *i*) *I* ∧ (*j*, *Suc i*) ∈ *match r*)
| *sat* (*MatchF I r*) *i* = (∃ *j*≥*i*. *mem* (τ σ *i*) (τ σ *j*) *I* ∧ (*i*, *Suc j*) ∈ *match r*)
| *match* (*Lookahead φ*) = {(*i*, *i*) | *i*. *sat φ i*}
| *match* (*Symbol φ*) = {(*i*, *Suc i*) | *i*. *sat φ i*}
| *match* (*Plus r s*) = *match r* ∪ *match s*
| *match* (*Times r s*) = *match r O match s*
| *match* (*Star r*) = *rtrancl* (*match r*)

lemma *sat* (*Prev I* (*Bool False*)) *i* ⟷ *sat* (*Bool False*) *i*
sat (*Next I* (*Bool False*)) *i* ⟷ *sat* (*Bool False*) *i*
sat (*Since φ I* (*Bool False*)) *i* ⟷ *sat* (*Bool False*) *i*
sat (*Until φ I* (*Bool False*)) *i* ⟷ *sat* (*Bool False*) *i*
⟨*proof*⟩

lemma *prev_rewrite*: *sat* (*Prev I* φ) *i* $\longleftrightarrow$ *sat* (*MatchP I* (*Times* (*Symbol* φ) (*Symbol* (*Bool True*)))) *i*
⟨*proof*⟩

lemma *next_rewrite*: *sat* (*Next I* φ) *i* $\longleftrightarrow$ *sat* (*MatchF I* (*Times* (*Symbol* (*Bool True*)) (*Symbol* φ))) *i*
⟨*proof*⟩

lemma *trancl_Base*: $\{(i,\ Suc\ i)\ |i.\ P\ i\}^* = \{(i,\ j).\ i \leq j \wedge (\forall\, k{\in}\{i..{<}j\}.\ P\ k)\}$
⟨*proof*⟩

lemma *Ball_atLeastLessThan_reindex*:
$(\forall\, k{\in}\{j..{<}i\}.\ P\ (Suc\ k)) = (\forall\, k \in \{j{<}..i\}.\ P\ k)$
⟨*proof*⟩

lemma *since_rewrite*: *sat* (*Since* φ *I* ψ) *i* $\longleftrightarrow$ *sat* (*MatchP I* (*Times* (*Symbol* ψ) (*Star* (*Symbol* φ)))) *i*
⟨*proof*⟩

lemma *until_rewrite*: *sat* (*Until* φ *I* ψ) *i* $\longleftrightarrow$ *sat* (*MatchF I* (*Times* (*Star* (*Symbol* φ)) (*Symbol* ψ))) *i*
⟨*proof*⟩

lemma *match_le*: $(i,\ j) \in match\ r \Longrightarrow i \leq j$
⟨*proof*⟩

lemma *match_Times*: $(i,\ i + n) \in match\ (Times\ r\ s) \longleftrightarrow$
$(\exists\, k \leq n.\ (i,\ i + k) \in match\ r \wedge (i + k,\ i + n) \in match\ s)$
⟨*proof*⟩

lemma *rtrancl_unfold*: $(x,\ z) \in rtrancl\ R \Longrightarrow$
$x = z \vee (\exists\, y.\ (x,\ y) \in R \wedge x \neq y \wedge (y,\ z) \in rtrancl\ R)$
⟨*proof*⟩

lemma *rtrancl_unfold′*: $(x,\ z) \in rtrancl\ R \Longrightarrow$
$x = z \vee (\exists\, y.\ (x,\ y) \in rtrancl\ R \wedge y \neq z \wedge (y,\ z) \in R)$
⟨*proof*⟩

lemma *match_Star*: $(i,\ i + Suc\ n) \in match\ (Star\ r) \longleftrightarrow$
$(\exists\, k \leq n.\ (i,\ i + 1 + k) \in match\ r \wedge (i + 1 + k,\ i + Suc\ n) \in match\ (Star\ r))$
⟨*proof*⟩

lemma *match_refl_eps*: $(i,\ i) \in match\ r \Longrightarrow eps\ r$
⟨*proof*⟩

lemma *wf_regex_eps_match*: *wf_regex r* $\Longrightarrow$ *eps r* $\Longrightarrow (i,\ i) \in match\ r$
⟨*proof*⟩

lemma *match_Star_unfold*: $i < j \Longrightarrow (i,\ j) \in match\ (Star\ r) \Longrightarrow \exists\, k \in \{i..{<}j\}.\ (i,\ k) \in match\ (Star\ r) \wedge (k,\ j) \in match\ r$
⟨*proof*⟩

lemma *match_rderive*: *wf_regex r* $\Longrightarrow$ $i \leq j \Longrightarrow (i,\ Suc\ j) \in match\ r \longleftrightarrow (i,\ j) \in match\ (rderive\ r)$
⟨*proof*⟩

end

lemma *atms_nonempty*: *atms r* $\neq \{\}$
⟨*proof*⟩

lemma *atms_finite*: *finite* (*atms r*)

⟨*proof*⟩

lemma *progress_le_ts*:
assumes $\bigwedge t.\ t \in set\ ts \Longrightarrow t \in tfin$
shows *progress phi ts* $\leq$ *length ts*
⟨*proof*⟩

end
theory *Metric_Point_Structure*
imports *Interval*
begin

class *metric_domain* = *plus* + *zero* + *ord* +
assumes $\Delta 1$: $x + x' = x' + x$
and $\Delta 2$: $(x + x') + x'' = x + (x' + x'')$
and $\Delta 3$: $x + 0 = x$
and $\Delta 3'$: $x = 0 + x$
and $\Delta 4$: $x + x' = x + x'' \Longrightarrow x' = x''$
and $\Delta 4'$: $x + x'' = x' + x'' \Longrightarrow x = x'$
and $\Delta 5$: $x + x' = 0 \Longrightarrow x = 0$
and $\Delta 5'$: $x + x' = 0 \Longrightarrow x' = 0$
and $\Delta 6$: $\exists x''.\ x = x' + x'' \vee x' = x + x''$
and *metric_domain_le_def*: $x \leq x' \longleftrightarrow (\exists x''.\ x' = x + x'')$
and *metric_domain_lt_def*: $x < x' \longleftrightarrow (\exists x''.\ x'' \neq 0 \wedge x' = x + x'')$
begin

lemma *metric_domain_pos*: $x \geq 0$
⟨*proof*⟩

lemma *less_eq_le_neq*: $x < x' \longleftrightarrow (x \leq x' \wedge x \neq x')$
⟨*proof*⟩

end

class *metric_domain_timestamp* = *metric_domain* + *sup* + *embed_nat* + *tfin* +
assumes *metric_domain_sup_def*: *sup x x'* = (*if x* $\leq$ *x' then x' else x*)
and *metric_domain_ι_mono*: $\bigwedge i\ j.\ i \leq j \Longrightarrow \iota\ i \leq \iota\ j$
and *metric_domain_ι_progressing*: $\exists j.\ \neg \iota\ j \leq \iota\ i + x$
and *metric_domain_tfin_def*: *tfin* = *UNIV*

subclass (**in** *metric_domain_timestamp*) *timestamp*
⟨*proof*⟩

locale *metric_point_structure* =
fixes $d :: 't :: \{order\} \Rightarrow 't \Rightarrow 'd :: metric\_domain\_timestamp$
assumes *d1*: $d\ t\ t' = 0 \longleftrightarrow t = t'$
and *d2*: $d\ t\ t' = d\ t'\ t$
and *d3*: $t < t' \Longrightarrow t' < t'' \Longrightarrow d\ t\ t'' = d\ t\ t' + d\ t'\ t''$
and *d3'*: $t < t' \Longrightarrow t' < t'' \Longrightarrow d\ t''\ t = d\ t''\ t' + d\ t'\ t$
begin

lemma *metric_point_structure_memL_aux*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow x \leq d\ t\ t' \longleftrightarrow (d\ t0\ t + x \leq d\ t0\ t')$
⟨*proof*⟩

lemma *metric_point_structure_memL_strict_aux*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow x < d\ t\ t' \longleftrightarrow (d\ t0\ t + x < d\ t0\ t')$
⟨*proof*⟩

lemma *metric_point_structure_memR_aux*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow d\ t\ t' \leq x \longleftrightarrow (d\ t0\ t' \leq d\ t0\ t + x)$
⟨*proof*⟩

lemma *metric_point_structure_memR_strict_aux*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow d\ t\ t' < x \longleftrightarrow (d\ t0\ t' < d\ t0\ t + x)$
⟨*proof*⟩

lemma *metric_point_structure_le_mem*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow d\ t\ t' \leq x \longleftrightarrow mem\ (d\ t0\ t)\ (d\ t0\ t')\ (interval\ 0\ x\ True\ True)$
⟨*proof*⟩

lemma *metric_point_structure_lt_mem*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow 0 < x \Longrightarrow d\ t\ t' < x \longleftrightarrow mem\ (d\ t0\ t)\ (d\ t0\ t')\ (interval\ 0\ x\ True\ False)$
⟨*proof*⟩

lemma *metric_point_structure_eq_mem*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow d\ t\ t' = x \longleftrightarrow mem\ (d\ t0\ t)\ (d\ t0\ t')\ (interval\ x\ x\ True\ True)$
⟨*proof*⟩

lemma *metric_point_structure_ge_mem*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow x \leq d\ t\ t' \longleftrightarrow mem\ (Some\ (d\ t0\ t))\ (Some\ (d\ t0\ t'))\ (interval\ (Some\ x)\ None\ True\ True)$
⟨*proof*⟩

lemma *metric_point_structure_gt_mem*: $t0 \leq t \Longrightarrow t \leq t' \Longrightarrow x < d\ t\ t' \longleftrightarrow mem\ (Some\ (d\ t0\ t))\ (Some\ (d\ t0\ t'))\ (interval\ (Some\ x)\ None\ False\ True)$
⟨*proof*⟩

end

instantiation *nat* :: *metric_domain_timestamp*
begin

instance
⟨*proof*⟩

end

interpretation *nat_metric_point_structure*: *metric_point_structure* $\lambda t :: nat.\ \lambda t'.\ if\ t \leq t'\ then\ t' - t\ else\ t - t'$
⟨*proof*⟩

end
theory *NFA*
imports *HOL−Library.IArray*
begin

type_synonym *state* = *nat*

datatype *transition* = *eps_trans state nat* | *symb_trans state* | *split_trans state state*

fun *state_set* :: *transition* ⇒ *state set* **where**
  state_set (*eps_trans s* _) = {*s*}
| *state_set* (*symb_trans s*) = {*s*}
| *state_set* (*split_trans s s'*) = {*s*, *s'*}

fun *fmla_set* :: *transition* ⇒ *nat set* **where**
  fmla_set (*eps_trans* _ *n*) = {*n*}
| *fmla_set* _ = {}

lemma *rtranclp_closed*: *rtranclp R q q'* ⟹ *X* = *X* ∪ {*q'*. ∃ *q* ∈ *X*. *R q q'*} ⟹
  q ∈ *X* ⟹ *q'* ∈ *X*
  ⟨*proof*⟩

lemma *rtranclp_closed_sub*: *rtranclp R q q'* ⟹ {*q'*. ∃ *q* ∈ *X*. *R q q'*} ⊆ *X* ⟹
  q ∈ *X* ⟹ *q'* ∈ *X*
  ⟨*proof*⟩

lemma *rtranclp_closed_sub'*: *rtranclp R q q'* ⟹ *q'* = *q* ∨ (∃ *q''*. *R q q''* ∧ *rtranclp R q'' q'*)
  ⟨*proof*⟩

lemma *rtranclp_step*: *rtranclp R q q''* ⟹ (⋀*q'*. *R q q'* ⟷ *q'* ∈ *X*) ⟹
  q = *q''* ∨ (∃ *q'* ∈ *X*. *R q q'* ∧ *rtranclp R q' q''*)
  ⟨*proof*⟩

lemma *rtranclp_unfold*: *rtranclp R x z* ⟹ *x* = *z* ∨ (∃ *y*. *R x y* ∧ *rtranclp R y z*)
  ⟨*proof*⟩

context fixes
  q0 :: *state* **and**
  qf :: *state* **and**
  transs :: *transition list*
begin

qualified definition *SQ* :: *state set* **where**
  SQ = {*q0*..<*q0* + *length transs*}

lemma *q_in_SQ*[*code_unfold*]: *q* ∈ *SQ* ⟷ *q0* ≤ *q* ∧ *q* < *q0* + *length transs*
  ⟨*proof*⟩

lemma *finite_SQ*: *finite SQ*
  ⟨*proof*⟩

lemma *transs_q_in_set*: *q* ∈ *SQ* ⟹ *transs* ! (*q* − *q0*) ∈ *set transs*
  ⟨*proof*⟩ **definition** *Q* :: *state set* **where**
  Q = *SQ* ∪ {*qf*}

lemma *finite_Q*: *finite Q*
  ⟨*proof*⟩

lemma *SQ_sub_Q*: *SQ* ⊆ *Q*
  ⟨*proof*⟩ **definition** *nfa_fmla_set* :: *nat set* **where**
  nfa_fmla_set = ⋃(*fmla_set* ‘ *set transs*)

qualified definition *step_eps* :: *bool list* ⇒ *state* ⇒ *state* ⇒ *bool* **where**
step_eps bs q q′ ⟷ *q* ∈ *SQ* ∧
(*case transs* ! (*q* − *q0*) *of eps_trans p n* ⇒ *n* < *length bs* ∧ *bs* ! *n* ∧ *p* = *q′*
| *split_trans p p′* ⇒ *p* = *q′* ∨ *p′* = *q′* | _ ⇒ *False*)

lemma *step_eps_dest*: *step_eps bs q q′* ⟹ *q* ∈ *SQ*
⟨*proof*⟩

lemma *step_eps_mono*: *step_eps* [] *q q′* ⟹ *step_eps bs q q′*
⟨*proof*⟩ **definition** *step_eps_sucs* :: *bool list* ⇒ *state* ⇒ *state set* **where**
step_eps_sucs bs q = (*if q* ∈ *SQ then*
(*case transs* ! (*q* − *q0*) *of eps_trans p n* ⇒ *if n* < *length bs* ∧ *bs* ! *n then* {*p*} *else* {}
| *split_trans p p′* ⇒ {*p*, *p′*} | _ ⇒ {}) *else* {})

lemma *step_eps_sucs_sound*: *q′* ∈ *step_eps_sucs bs q* ⟷ *step_eps bs q q′*
⟨*proof*⟩ **definition** *step_eps_set* :: *bool list* ⇒ *state set* ⇒ *state set* **where**
step_eps_set bs R = ⋃(*step_eps_sucs bs* ‘ *R*)

lemma *step_eps_set_sound*: *step_eps_set bs R* = {*q′*. ∃ *q* ∈ *R*. *step_eps bs q q′*}
⟨*proof*⟩

lemma *step_eps_set_mono*: *R* ⊆ *S* ⟹ *step_eps_set bs R* ⊆ *step_eps_set bs S*
⟨*proof*⟩ **definition** *step_eps_closure* :: *bool list* ⇒ *state* ⇒ *state* ⇒ *bool* **where**
step_eps_closure bs = (*step_eps bs*)**

lemma *step_eps_closure_dest*: *step_eps_closure bs q q′* ⟹ *q* ≠ *q′* ⟹ *q* ∈ *SQ*
⟨*proof*⟩ **definition** *step_eps_closure_set* :: *state set* ⇒ *bool list* ⇒ *state set* **where**
step_eps_closure_set R bs = ⋃((λ*q*. {*q′*. *step_eps_closure bs q q′*}) ‘ *R*)

lemma *step_eps_closure_set_refl*: *R* ⊆ *step_eps_closure_set R bs*
⟨*proof*⟩

lemma *step_eps_closure_set_mono*: *R* ⊆ *S* ⟹ *step_eps_closure_set R bs* ⊆ *step_eps_closure_set S bs*
⟨*proof*⟩

lemma *step_eps_closure_set_empty*: *step_eps_closure_set* {} *bs* = {}
⟨*proof*⟩

lemma *step_eps_closure_set_mono′*: *step_eps_closure_set R* [] ⊆ *step_eps_closure_set R bs*
⟨*proof*⟩

lemma *step_eps_closure_set_split*: *step_eps_closure_set* (*R* ∪ *S*) *bs* =
step_eps_closure_set R bs ∪ *step_eps_closure_set S bs*
⟨*proof*⟩

lemma *step_eps_closure_set_Un*: *step_eps_closure_set* (⋃*x* ∈ *X*. *R x*) *bs* =
(⋃*x* ∈ *X*. *step_eps_closure_set* (*R x*) *bs*)
⟨*proof*⟩

lemma *step_eps_closure_set_idem*: *step_eps_closure_set* (*step_eps_closure_set R bs*) *bs* =
step_eps_closure_set R bs
⟨*proof*⟩

lemma *step_eps_closure_set_flip*:
assumes *step_eps_closure_set R bs* = *R* ∪ *S*
shows *step_eps_closure_set S bs* ⊆ *R* ∪ *S*

⟨*proof*⟩

lemma *step_eps_closure_set_unfold*: $(\bigwedge q'.\ step\_eps\ bs\ q\ q' \longleftrightarrow q' \in X) \Longrightarrow$
step_eps_closure_set $\{q\}$ *bs* $= \{q\} \cup$ *step_eps_closure_set X bs*
⟨*proof*⟩

lemma *step_step_eps_closure*: *step_eps bs q q'* $\Longrightarrow q \in R \Longrightarrow q' \in$ *step_eps_closure_set R bs*
⟨*proof*⟩

lemma *step_eps_closure_set_code*[*code*]:
step_eps_closure_set R bs =
(*let* $R' = R \cup$ *step_eps_set bs R in if* $R = R'$ *then R else step_eps_closure_set* R' *bs*)
⟨*proof*⟩

lemma *step_eps_closure_empty*: *step_eps_closure bs q q'* $\Longrightarrow (\bigwedge q'.\ \neg$*step_eps bs q q'*$) \Longrightarrow q = q'$
⟨*proof*⟩

lemma *step_eps_closure_set_step_id*: $(\bigwedge q\ q'.\ q \in R \Longrightarrow \neg$*step_eps bs q q'*$) \Longrightarrow$
step_eps_closure_set R bs $= R$
⟨*proof*⟩ **definition** *step_symb* :: *state* ⇒ *state* ⇒ *bool* **where**
step_symb q q' $\longleftrightarrow q \in SQ \wedge$
(*case transs* ! $(q - q0)$ *of symb_trans p* ⇒ $p = q'$ | _ ⇒ *False*)

lemma *step_symb_dest*: *step_symb q q'* $\Longrightarrow q \in SQ$
⟨*proof*⟩ **definition** *step_symb_sucs* :: *state* ⇒ *state set* **where**
step_symb_sucs q = (*if* $q \in SQ$ *then*
(*case transs* ! $(q - q0)$ *of symb_trans p* ⇒ $\{p\}$ | _ ⇒ $\{\}$) *else* $\{\}$)

lemma *step_symb_sucs_sound*: $q' \in$ *step_symb_sucs q* $\longleftrightarrow$ *step_symb q q'*
⟨*proof*⟩ **definition** *step_symb_set* :: *state set* ⇒ *state set* **where**
step_symb_set R = $\{q'.\ \exists q \in R.$ *step_symb q q'*$\}$

lemma *step_symb_set_mono*: $R \subseteq S \Longrightarrow$ *step_symb_set R* $\subseteq$ *step_symb_set S*
⟨*proof*⟩

lemma *step_symb_set_empty*: *step_symb_set* $\{\} = \{\}$
⟨*proof*⟩

lemma *step_symb_set_proj*: *step_symb_set R* = *step_symb_set* $(R \cap SQ)$
⟨*proof*⟩

lemma *step_symb_set_split*: *step_symb_set* $(R \cup S)$ = *step_symb_set R* $\cup$ *step_symb_set S*
⟨*proof*⟩

lemma *step_symb_set_Un*: *step_symb_set* $(\bigcup x \in X.\ R\ x) = (\bigcup x \in X.$ *step_symb_set* $(R\ x))$
⟨*proof*⟩

lemma *step_symb_set_code*[*code*]: *step_symb_set R* $= \bigcup$ (*step_symb_sucs* ‘ *R*)
⟨*proof*⟩ **definition** *delta* :: *state set* ⇒ *bool list* ⇒ *state set* **where**
delta R bs = *step_symb_set* (*step_eps_closure_set R bs*)

lemma *delta_eps*: *delta* (*step_eps_closure_set R bs*) *bs* = *delta R bs*
⟨*proof*⟩

lemma *delta_eps_split*:
assumes *step_eps_closure_set R bs* $= R \cup S$

shows *delta R bs = step_symb_set R ∪ delta S bs*
⟨*proof*⟩

lemma *delta_split*: *delta (R ∪ S) bs = delta R bs ∪ delta S bs*
⟨*proof*⟩

lemma *delta_Un*: *delta (⋃x ∈ X. R x) bs = (⋃x ∈ X. delta (R x) bs)*
⟨*proof*⟩

lemma *delta_step_symb_set_absorb*: *delta R bs = delta R bs ∪ step_symb_set R*
⟨*proof*⟩

lemma *delta_sub_eps_mono*:
assumes *S ⊆ step_eps_closure_set R bs*
shows *delta S bs ⊆ delta R bs*
⟨*proof*⟩ **definition** *run :: state set ⇒ bool list list ⇒ state set* **where**
run R bss = foldl delta R bss

lemma *run_eps_split*:
assumes *step_eps_closure_set R bs = R ∪ S step_symb_set R = {}*
shows *run R (bs # bss) = run S (bs # bss)*
⟨*proof*⟩

lemma *run_empty*: *run {} bss = {}*
⟨*proof*⟩

lemma *run_Nil*: *run R [] = R*
⟨*proof*⟩

lemma *run_Cons*: *run R (bs # bss) = run (delta R bs) bss*
⟨*proof*⟩

lemma *run_split*: *run (R ∪ S) bss = run R bss ∪ run S bss*
⟨*proof*⟩

lemma *run_Un*: *run (⋃x ∈ X. R x) bss = (⋃x ∈ X. run (R x) bss)*
⟨*proof*⟩

lemma *run_comp*: *run R (bss @ css) = run (run R bss) css*
⟨*proof*⟩ **definition** *accept_eps :: state set ⇒ bool list ⇒ bool* **where**
accept_eps R bs ⟷ (qf ∈ step_eps_closure_set R bs)

lemma *step_eps_accept_eps*: *step_eps bs q qf ⟹ q ∈ R ⟹ accept_eps R bs*
⟨*proof*⟩

lemma *accept_eps_empty*: *accept_eps {} bs ⟷ False*
⟨*proof*⟩

lemma *accept_eps_split*: *accept_eps (R ∪ S) bs ⟷ accept_eps R bs ∨ accept_eps S bs*
⟨*proof*⟩

lemma *accept_eps_Un*: *accept_eps (⋃x ∈ X. R x) bs ⟷ (∃x ∈ X. accept_eps (R x) bs)*
⟨*proof*⟩ **definition** *accept :: state set ⇒ bool* **where**
accept R ⟷ accept_eps R []

qualified definition *run_accept_eps :: state set ⇒ bool list list ⇒ bool list ⇒ bool* **where**

run_accept_eps R bss bs = accept_eps (run R bss) bs

lemma *run_accept_eps_empty*: ¬*run_accept_eps* {} *bss bs*
⟨*proof*⟩

lemma *run_accept_eps_Nil*: *run_accept_eps R* [] *cs* ⟷ *accept_eps R cs*
⟨*proof*⟩

lemma *run_accept_eps_Cons*: *run_accept_eps R* (*bs* # *bss*) *cs* ⟷ *run_accept_eps* (*delta R bs*) *bss cs*
⟨*proof*⟩

lemma *run_accept_eps_Cons_delta_cong*: *delta R bs* = *delta S bs* ⟹
run_accept_eps R (*bs* # *bss*) *cs* ⟷ *run_accept_eps S* (*bs* # *bss*) *cs*
⟨*proof*⟩

lemma *run_accept_eps_Nil_eps*: *run_accept_eps* (*step_eps_closure_set R bs*) [] *bs* ⟷ *run_accept_eps R* [] *bs*
⟨*proof*⟩

lemma *run_accept_eps_Cons_eps*: *run_accept_eps* (*step_eps_closure_set R cs*) (*cs* # *css*) *bs* ⟷
run_accept_eps R (*cs* # *css*) *bs*
⟨*proof*⟩

lemma *run_accept_eps_Nil_eps_split*:
assumes *step_eps_closure_set R bs* = *R* ∪ *S step_symb_set R* = {} *qf* ∉ *R*
shows *run_accept_eps R* [] *bs* = *run_accept_eps S* [] *bs*
⟨*proof*⟩

lemma *run_accept_eps_Cons_eps_split*:
assumes *step_eps_closure_set R cs* = *R* ∪ *S step_symb_set R* = {} *qf* ∉ *R*
shows *run_accept_eps R* (*cs* # *css*) *bs* = *run_accept_eps S* (*cs* # *css*) *bs*
⟨*proof*⟩

lemma *run_accept_eps_split*: *run_accept_eps* (*R* ∪ *S*) *bss bs* ⟷
run_accept_eps R bss bs ∨ *run_accept_eps S bss bs*
⟨*proof*⟩

lemma *run_accept_eps_Un*: *run_accept_eps* (⋃ *x* ∈ *X*. *R x*) *bss bs* ⟷
(∃ *x* ∈ *X*. *run_accept_eps* (*R x*) *bss bs*)
⟨*proof*⟩ **definition** *run_accept* :: *state set* ⇒ *bool list list* ⇒ *bool* **where**
run_accept R bss = *accept* (*run R bss*)

end

definition *iarray_of_list xs* = *IArray xs*

context fixes
transs :: *transition iarray*
and *len* :: *nat*
begin

qualified definition *step_eps′* :: *bool iarray* ⇒ *state* ⇒ *state* ⇒ *bool* **where**
step_eps′ bs q q′ ⟷ *q* < *len* ∧
(*case transs* !! *q of eps_trans p n* ⇒ *n* < *IArray.length bs* ∧ *bs* !! *n* ∧ *p* = *q′*
| *split_trans p p′* ⇒ *p* = *q′* ∨ *p′* = *q′* | _ ⇒ *False*)

qualified definition *step_eps_closure′* :: *bool iarray* ⇒ *state* ⇒ *state* ⇒ *bool* **where**

*step_eps_closure' bs = (step_eps' bs)***

qualified definition *step_eps_sucs' :: bool iarray ⇒ state ⇒ state set* **where**
step_eps_sucs' bs q = (if q < len then
(case transs !! q of eps_trans p n ⇒ if n < IArray.length bs ∧ bs !! n then {p} else {}
| split_trans p p' ⇒ {p, p'} | _ ⇒ {}) else {})

lemma *step_eps_sucs'_sound: q' ∈ step_eps_sucs' bs q ⟷ step_eps' bs q q'*
⟨*proof*⟩ **definition** *step_eps_set' :: bool iarray ⇒ state set ⇒ state set* **where**
step_eps_set' bs R = ⋃(step_eps_sucs' bs ‘ R)

lemma *step_eps_set'_sound: step_eps_set' bs R = {q'. ∃ q ∈ R. step_eps' bs q q'}*
⟨*proof*⟩ **definition** *step_eps_closure_set' :: state set ⇒ bool iarray ⇒ state set* **where**
step_eps_closure_set' R bs = ⋃((λq. {q'. step_eps_closure' bs q q'}) ‘ R)

lemma *step_eps_closure_set'_code*[*code*]:
step_eps_closure_set' R bs =
(let R' = R ∪ step_eps_set' bs R in if R = R' then R else step_eps_closure_set' R' bs)
⟨*proof*⟩ **definition** *step_symb_sucs' :: state ⇒ state set* **where**
step_symb_sucs' q = (if q < len then
(case transs !! q of symb_trans p ⇒ {p} | _ ⇒ {}) else {})

qualified definition *step_symb_set' :: state set ⇒ state set* **where**
step_symb_set' R = ⋃(step_symb_sucs' ‘ R)

qualified definition *delta' :: state set ⇒ bool iarray ⇒ state set* **where**
delta' R bs = step_symb_set' (step_eps_closure_set' R bs)

qualified definition *accept_eps' :: state set ⇒ bool iarray ⇒ bool* **where**
accept_eps' R bs ⟷ (len ∈ step_eps_closure_set' R bs)

qualified definition *accept' :: state set ⇒ bool* **where**
accept' R ⟷ accept_eps' R (iarray_of_list [])

qualified definition *run' :: state set ⇒ bool iarray list ⇒ state set* **where**
run' R bss = foldl delta' R bss

qualified definition *run_accept_eps' :: state set ⇒ bool iarray list ⇒ bool iarray ⇒ bool* **where**
run_accept_eps' R bss bs = accept_eps' (run' R bss) bs

qualified definition *run_accept' :: state set ⇒ bool iarray list ⇒ bool* **where**
run_accept' R bss = accept' (run' R bss)

end

locale *nfa_array =*
fixes *transs :: transition list*
and *transs' :: transition iarray*
and *len :: nat*
assumes *transs_eq: transs' = IArray transs*
and *len_def: len = length transs*
begin

abbreviation *step_eps ≡ NFA.step_eps 0 transs*
abbreviation *step_eps' ≡ NFA.step_eps' transs' len*
abbreviation *step_eps_closure ≡ NFA.step_eps_closure 0 transs*
abbreviation *step_eps_closure' ≡ NFA.step_eps_closure' transs' len*
abbreviation *step_eps_sucs ≡ NFA.step_eps_sucs 0 transs*

abbreviation *step_eps_sucs'* ≡ *NFA.step_eps_sucs' transs' len*
abbreviation *step_eps_set* ≡ *NFA.step_eps_set 0 transs*
abbreviation *step_eps_set'* ≡ *NFA.step_eps_set' transs' len*
abbreviation *step_eps_closure_set* ≡ *NFA.step_eps_closure_set 0 transs*
abbreviation *step_eps_closure_set'* ≡ *NFA.step_eps_closure_set' transs' len*
abbreviation *step_symb_sucs* ≡ *NFA.step_symb_sucs 0 transs*
abbreviation *step_symb_sucs'* ≡ *NFA.step_symb_sucs' transs' len*
abbreviation *step_symb_set* ≡ *NFA.step_symb_set 0 transs*
abbreviation *step_symb_set'* ≡ *NFA.step_symb_set' transs' len*
abbreviation *delta* ≡ *NFA.delta 0 transs*
abbreviation *delta'* ≡ *NFA.delta' transs' len*
abbreviation *accept_eps* ≡ *NFA.accept_eps 0 len transs*
abbreviation *accept_eps'* ≡ *NFA.accept_eps' transs' len*
abbreviation *accept* ≡ *NFA.accept 0 len transs*
abbreviation *accept'* ≡ *NFA.accept' transs' len*
abbreviation *run* ≡ *NFA.run 0 transs*
abbreviation *run'* ≡ *NFA.run' transs' len*
abbreviation *run_accept_eps* ≡ *NFA.run_accept_eps 0 len transs*
abbreviation *run_accept_eps'* ≡ *NFA.run_accept_eps' transs' len*
abbreviation *run_accept* ≡ *NFA.run_accept 0 len transs*
abbreviation *run_accept'* ≡ *NFA.run_accept' transs' len*

lemma *q_in_SQ*: *q* ∈ *NFA.SQ 0 transs* ⟷ *q* < *len*
⟨*proof*⟩

lemma *step_eps'_eq*: *bs'* = *IArray bs* ⟹ *step_eps bs q q'* ⟷ *step_eps' bs' q q'*
⟨*proof*⟩

lemma *step_eps_closure'_eq*: *bs'* = *IArray bs* ⟹ *step_eps_closure bs q q'* ⟷ *step_eps_closure' bs' q q'*
⟨*proof*⟩

lemma *step_eps_sucs'_eq*: *bs'* = *IArray bs* ⟹ *step_eps_sucs bs q* = *step_eps_sucs' bs' q*
⟨*proof*⟩

lemma *step_eps_set'_eq*: *bs'* = *IArray bs* ⟹ *step_eps_set bs R* = *step_eps_set' bs' R*
⟨*proof*⟩

lemma *step_eps_closure_set'_eq*: *bs'* = *IArray bs* ⟹ *step_eps_closure_set R bs* = *step_eps_closure_set' R bs'*
⟨*proof*⟩

lemma *step_symb_sucs'_eq*: *bs'* = *IArray bs* ⟹ *step_symb_sucs R* = *step_symb_sucs' R*
⟨*proof*⟩

lemma *step_symb_set'_eq*: *bs'* = *IArray bs* ⟹ *step_symb_set R* = *step_symb_set' R*
⟨*proof*⟩

lemma *delta'_eq*: *bs'* = *IArray bs* ⟹ *delta R bs* = *delta' R bs'*
⟨*proof*⟩

lemma *accept_eps'_eq*: *bs'* = *IArray bs* ⟹ *accept_eps R bs* = *accept_eps' R bs'*
⟨*proof*⟩

lemma *accept'_eq*: *accept R* = *accept' R*
⟨*proof*⟩

lemma *run'_eq*: *bss'* = *map IArray bss* ⟹ *run R bss* = *run' R bss'*

⟨*proof*⟩

lemma *run_accept_eps'_eq*: $bss' = map\ IArray\ bss \Longrightarrow bs' = IArray\ bs \Longrightarrow$
$run\_accept\_eps\ R\ bss\ bs \longleftrightarrow run\_accept\_eps'\ R\ bss'\ bs'$
⟨*proof*⟩

lemma *run_accept'_eq*: $bss' = map\ IArray\ bss \Longrightarrow$
$run\_accept\ R\ bss \longleftrightarrow run\_accept'\ R\ bss'$
⟨*proof*⟩

end

locale *nfa* =
fixes *q0* :: *nat*
and *qf* :: *nat*
and *transs* :: *transition list*
assumes *state_closed*: $\bigwedge t.\ t \in set\ transs \Longrightarrow state\_set\ t \subseteq NFA.Q\ q0\ qf\ transs$
and *transs_not_Nil*: $transs \neq []$
and *qf_not_in_SQ*: $qf \notin NFA.SQ\ q0\ transs$
begin

abbreviation $SQ \equiv NFA.SQ\ q0\ transs$
abbreviation $Q \equiv NFA.Q\ q0\ qf\ transs$
abbreviation $nfa\_fmla\_set \equiv NFA.nfa\_fmla\_set\ transs$
abbreviation $step\_eps \equiv NFA.step\_eps\ q0\ transs$
abbreviation $step\_eps\_sucs \equiv NFA.step\_eps\_sucs\ q0\ transs$
abbreviation $step\_eps\_set \equiv NFA.step\_eps\_set\ q0\ transs$
abbreviation $step\_eps\_closure \equiv NFA.step\_eps\_closure\ q0\ transs$
abbreviation $step\_eps\_closure\_set \equiv NFA.step\_eps\_closure\_set\ q0\ transs$
abbreviation $step\_symb \equiv NFA.step\_symb\ q0\ transs$
abbreviation $step\_symb\_sucs \equiv NFA.step\_symb\_sucs\ q0\ transs$
abbreviation $step\_symb\_set \equiv NFA.step\_symb\_set\ q0\ transs$
abbreviation $delta \equiv NFA.delta\ q0\ transs$
abbreviation $run \equiv NFA.run\ q0\ transs$
abbreviation $accept\_eps \equiv NFA.accept\_eps\ q0\ qf\ transs$
abbreviation $run\_accept\_eps \equiv NFA.run\_accept\_eps\ q0\ qf\ transs$

lemma *Q_diff_qf_SQ*: $Q - \{qf\} = SQ$
⟨*proof*⟩

lemma *q0_sub_SQ*: $\{q0\} \subseteq SQ$
⟨*proof*⟩

lemma *q0_sub_Q*: $\{q0\} \subseteq Q$
⟨*proof*⟩

lemma *step_eps_closed*: $step\_eps\ bs\ q\ q' \Longrightarrow q' \in Q$
⟨*proof*⟩

lemma *step_eps_set_closed*: $step\_eps\_set\ bs\ R \subseteq Q$
⟨*proof*⟩

lemma *step_eps_closure_closed*: $step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \neq q' \Longrightarrow q' \in Q$
⟨*proof*⟩

lemma *step_eps_closure_set_closed_union*: $step\_eps\_closure\_set\ R\ bs \subseteq R \cup Q$
⟨*proof*⟩

lemma *step_eps_closure_set_closed*: $R \subseteq Q \Longrightarrow$ *step_eps_closure_set* R *bs* $\subseteq Q$
⟨*proof*⟩

lemma *step_symb_closed*: *step_symb* q $q' \Longrightarrow q' \in Q$
⟨*proof*⟩

lemma *step_symb_set_closed*: *step_symb_set* $R \subseteq Q$
⟨*proof*⟩

lemma *step_symb_set_qf*: *step_symb_set* $\{qf\} = \{\}$
⟨*proof*⟩

lemma *delta_closed*: *delta* R *bs* $\subseteq Q$
⟨*proof*⟩

lemma *run_closed_Cons*: *run* R (*bs* # *bss*) $\subseteq Q$
⟨*proof*⟩

lemma *run_closed*: $R \subseteq Q \Longrightarrow$ *run* R *bss* $\subseteq Q$
⟨*proof*⟩

lemma *step_eps_qf*: *step_eps bs qf q* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *step_symb_qf*: *step_symb qf q* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *step_eps_closure_qf*: *step_eps_closure bs q* $q' \Longrightarrow q = qf \Longrightarrow q = q'$
⟨*proof*⟩

lemma *step_eps_closure_set_qf*: *step_eps_closure_set* $\{qf\}$ *bs* $= \{qf\}$
⟨*proof*⟩

lemma *delta_qf*: *delta* $\{qf\}$ *bs* $= \{\}$
⟨*proof*⟩

lemma *run_qf_many*: *run* $\{qf\}$ (*bs* # *bss*) $= \{\}$
⟨*proof*⟩

lemma *run_accept_eps_qf_many*: *run_accept_eps* $\{qf\}$ (*bs* # *bss*) *cs* $\longleftrightarrow$ *False*
⟨*proof*⟩

lemma *run_accept_eps_qf_one*: *run_accept_eps* $\{qf\}$ [] *bs* $\longleftrightarrow$ *True*
⟨*proof*⟩

end

locale *nfa_cong* = *nfa q0 qf transs* + *nfa'*: *nfa q0' qf' transs'*
for *q0 q0' qf qf' transs transs'* +
assumes *SQ_sub*: *nfa'.SQ* $\subseteq$ *SQ* **and**
qf_eq: *qf* $=$ *qf'* **and**
transs_eq: $\bigwedge q.\ q \in$ *nfa'.SQ* $\Longrightarrow$ *transs* ! (*q* − *q0*) $=$ *transs'* ! (*q* − *q0'*)
begin

lemma *q_Q_SQ_nfa'_SQ*: $q \in$ *nfa'.Q* $\Longrightarrow q \in$ *SQ* $\longleftrightarrow q \in$ *nfa'.SQ*
⟨*proof*⟩

lemma *step_eps_cong*: $q \in nfa'.Q \Longrightarrow step\_eps\ bs\ q\ q' \longleftrightarrow nfa'.step\_eps\ bs\ q\ q'$
⟨*proof*⟩

lemma *eps_nfa'_step_eps_closure*: $step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \in nfa'.Q \Longrightarrow$
$q' \in nfa'.Q \wedge nfa'.step\_eps\_closure\ bs\ q\ q'$
⟨*proof*⟩

lemma *nfa'_eps_step_eps_closure*: $nfa'.step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \in nfa'.Q \Longrightarrow$
$q' \in nfa'.Q \wedge step\_eps\_closure\ bs\ q\ q'$
⟨*proof*⟩

lemma *step_eps_closure_set_cong*: $R \subseteq nfa'.Q \Longrightarrow step\_eps\_closure\_set\ R\ bs =$
$nfa'.step\_eps\_closure\_set\ R\ bs$
⟨*proof*⟩

lemma *step_symb_cong*: $q \in nfa'.Q \Longrightarrow step\_symb\ q\ q' \longleftrightarrow nfa'.step\_symb\ q\ q'$
⟨*proof*⟩

lemma *step_symb_set_cong*: $R \subseteq nfa'.Q \Longrightarrow step\_symb\_set\ R = nfa'.step\_symb\_set\ R$
⟨*proof*⟩

lemma *delta_cong*: $R \subseteq nfa'.Q \Longrightarrow delta\ R\ bs = nfa'.delta\ R\ bs$
⟨*proof*⟩

lemma *run_cong*: $R \subseteq nfa'.Q \Longrightarrow run\ R\ bss = nfa'.run\ R\ bss$
⟨*proof*⟩

lemma *accept_eps_cong*: $R \subseteq nfa'.Q \Longrightarrow accept\_eps\ R\ bs \longleftrightarrow nfa'.accept\_eps\ R\ bs$
⟨*proof*⟩

lemma *run_accept_eps_cong*:
assumes $R \subseteq nfa'.Q$
shows $run\_accept\_eps\ R\ bss\ bs \longleftrightarrow nfa'.run\_accept\_eps\ R\ bss\ bs$
⟨*proof*⟩

end

fun *list_split* :: $'a\ list \Rightarrow ('a\ list \times 'a\ list)\ set$ **where**
$list\_split\ [] = \{\}$
| $list\_split\ (x\ \#\ xs) = \{([], x\ \#\ xs)\} \cup (\bigcup (ys, zs) \in list\_split\ xs.\ \{(x\ \#\ ys, zs)\})$

lemma *list_split_unfold*: $(\bigcup (ys, zs) \in list\_split\ (x\ \#\ xs).\ f\ ys\ zs) =$
$f\ []\ (x\ \#\ xs) \cup (\bigcup (ys, zs) \in list\_split\ xs.\ f\ (x\ \#\ ys)\ zs)$
⟨*proof*⟩

lemma *list_split_def*: $list\_split\ xs = (\bigcup n < length\ xs.\ \{(take\ n\ xs, drop\ n\ xs)\})$
⟨*proof*⟩

locale *nfa_cong'* = *nfa q0 qf transs* + *nfa'*: *nfa q0' qf' transs'*
for *q0 q0' qf qf' transs transs'* +
assumes *SQ_sub*: $nfa'.SQ \subseteq SQ$ **and**
qf'_in_SQ: $qf' \in SQ$ **and**
transs_eq: $\bigwedge q.\ q \in nfa'.SQ \Longrightarrow transs\ !\ (q - q0) = transs'\ !\ (q - q0')$
begin

lemma *nfa'_Q_sub_Q*: $nfa'.Q \subseteq Q$
⟨*proof*⟩

lemma *q_SQ_SQ_nfa'_SQ*: $q \in nfa'.SQ \Longrightarrow q \in SQ \longleftrightarrow q \in nfa'.SQ$
⟨*proof*⟩

lemma *step_eps_cong_SQ*: $q \in nfa'.SQ \Longrightarrow step\_eps\ bs\ q\ q' \longleftrightarrow nfa'.step\_eps\ bs\ q\ q'$
⟨*proof*⟩

lemma *step_eps_cong_Q*: $q \in nfa'.Q \Longrightarrow nfa'.step\_eps\ bs\ q\ q' \Longrightarrow step\_eps\ bs\ q\ q'$
⟨*proof*⟩

lemma *nfa'_step_eps_closure_cong*: $nfa'.step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \in nfa'.Q \Longrightarrow$
$step\_eps\_closure\ bs\ q\ q'$
⟨*proof*⟩

lemma *nfa'_step_eps_closure_set_sub*: $R \subseteq nfa'.Q \Longrightarrow nfa'.step\_eps\_closure\_set\ R\ bs \subseteq$
$step\_eps\_closure\_set\ R\ bs$
⟨*proof*⟩

lemma *eps_nfa'_step_eps_closure_cong*: $step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \in nfa'.Q \Longrightarrow$
$(q' \in nfa'.Q \wedge nfa'.step\_eps\_closure\ bs\ q\ q') \vee$
$(nfa'.step\_eps\_closure\ bs\ q\ qf' \wedge step\_eps\_closure\ bs\ qf'\ q')$
⟨*proof*⟩

lemma *nfa'_eps_step_eps_closure_cong*: $nfa'.step\_eps\_closure\ bs\ q\ q' \Longrightarrow q \in nfa'.Q \Longrightarrow$
$q' \in nfa'.Q \wedge step\_eps\_closure\ bs\ q\ q'$
⟨*proof*⟩

lemma *step_eps_closure_set_cong_reach*: $R \subseteq nfa'.Q \Longrightarrow qf' \in nfa'.step\_eps\_closure\_set\ R\ bs \Longrightarrow$
$step\_eps\_closure\_set\ R\ bs = nfa'.step\_eps\_closure\_set\ R\ bs \cup step\_eps\_closure\_set\ \{qf'\}\ bs$
⟨*proof*⟩

lemma *step_eps_closure_set_cong_unreach*: $R \subseteq nfa'.Q \Longrightarrow qf' \notin nfa'.step\_eps\_closure\_set\ R\ bs \Longrightarrow$
$step\_eps\_closure\_set\ R\ bs = nfa'.step\_eps\_closure\_set\ R\ bs$
⟨*proof*⟩

lemma *step_symb_cong_SQ*: $q \in nfa'.SQ \Longrightarrow step\_symb\ q\ q' \longleftrightarrow nfa'.step\_symb\ q\ q'$
⟨*proof*⟩

lemma *step_symb_cong_Q*: $nfa'.step\_symb\ q\ q' \Longrightarrow step\_symb\ q\ q'$
⟨*proof*⟩

lemma *step_symb_set_cong_SQ*: $R \subseteq nfa'.SQ \Longrightarrow step\_symb\_set\ R = nfa'.step\_symb\_set\ R$
⟨*proof*⟩

lemma *step_symb_set_cong_Q*: $nfa'.step\_symb\_set\ R \subseteq step\_symb\_set\ R$
⟨*proof*⟩

lemma *delta_cong_unreach*:
assumes $R \subseteq nfa'.Q$ $\neg nfa'.accept\_eps\ R\ bs$
shows $delta\ R\ bs = nfa'.delta\ R\ bs$
⟨*proof*⟩

lemma *nfa'_delta_sub_delta*:
assumes $R \subseteq nfa'.Q$
shows $nfa'.delta\ R\ bs \subseteq delta\ R\ bs$
⟨*proof*⟩

lemma *delta_cong_reach*:

assumes $R \subseteq nfa'.Q$ $nfa'.accept\_eps\ R\ bs$
shows $delta\ R\ bs = nfa'.delta\ R\ bs \cup delta\ \{qf'\}\ bs$
⟨*proof*⟩

lemma *run_cong*:
assumes $R \subseteq nfa'.Q$
shows $run\ R\ bss = nfa'.run\ R\ bss \cup (\bigcup (css,\ css') \in list\_split\ bss.$
if $nfa'.run\_accept\_eps\ R\ css\ (hd\ css')$ *then* $run\ \{qf'\}\ css'$ *else* $\{\})$
⟨*proof*⟩

lemma *run_cong_Cons_sub*:
assumes $R \subseteq nfa'.Q$ $delta\ \{qf'\}\ bs \subseteq nfa'.delta\ R\ bs$
shows $run\ R\ (bs\ \#\ bss) = nfa'.run\ R\ (bs\ \#\ bss) \cup$
$(\bigcup (css,\ css') \in list\_split\ bss.$
if $nfa'.run\_accept\_eps\ (nfa'.delta\ R\ bs)\ css\ (hd\ css')$ *then* $run\ \{qf'\}\ css'$ *else* $\{\})$
⟨*proof*⟩

lemma *accept_eps_nfa'_run*:
assumes $R \subseteq nfa'.Q$
shows $accept\_eps\ (nfa'.run\ R\ bss)\ bs \longleftrightarrow$
$nfa'.accept\_eps\ (nfa'.run\ R\ bss)\ bs \wedge accept\_eps\ (run\ \{qf'\}\ []\ bs)$
⟨*proof*⟩

lemma *run_accept_eps_cong*:
assumes $R \subseteq nfa'.Q$
shows $run\_accept\_eps\ R\ bss\ bs \longleftrightarrow (nfa'.run\_accept\_eps\ R\ bss\ bs \wedge run\_accept\_eps\ \{qf'\}\ []\ bs) \vee$
$(\exists (css,\ css') \in list\_split\ bss.\ nfa'.run\_accept\_eps\ R\ css\ (hd\ css') \wedge$
$run\_accept\_eps\ \{qf'\}\ css'\ bs)$
⟨*proof*⟩

lemma *run_accept_eps_cong_Cons_sub*:
assumes $R \subseteq nfa'.Q$ $delta\ \{qf'\}\ bs \subseteq nfa'.delta\ R\ bs$
shows $run\_accept\_eps\ R\ (bs\ \#\ bss)\ cs \longleftrightarrow$
$(nfa'.run\_accept\_eps\ R\ (bs\ \#\ bss)\ cs \wedge run\_accept\_eps\ \{qf'\}\ []\ cs) \vee$
$(\exists (css,\ css') \in list\_split\ bss.\ nfa'.run\_accept\_eps\ (nfa'.delta\ R\ bs)\ css\ (hd\ css') \wedge$
$run\_accept\_eps\ \{qf'\}\ css'\ cs)$
⟨*proof*⟩

lemmas *run_accept_eps_cong_Cons_sub_simp* =
run_accept_eps_cong_Cons_sub[*unfolded list_split_def*, *simplified*,
unfolded run_accept_eps_Cons[*symmetric*] *take_Suc_Cons*[*symmetric*]]

end

locale *nfa_cong_Plus* = *nfa_cong q0 q0' qf qf' transs transs'* +
right: *nfa_cong q0 q0'' qf qf'' transs transs''*
for *q0 q0' q0'' qf qf' qf'' transs transs' transs''* +
assumes *step_eps_q0*: $step\_eps\ bs\ q0\ q \longleftrightarrow q \in \{q0',\ q0''\}$ **and**
step_symb_q0: $\neg step\_symb\ q0\ q$
begin

lemma *step_symb_set_q0*: $step\_symb\_set\ \{q0\} = \{\}$
⟨*proof*⟩

lemma *qf_not_q0*: $qf \notin \{q0\}$
⟨*proof*⟩

lemma *step_eps_closure_set_q0*: $step\_eps\_closure\_set\ \{q0\}\ bs = \{q0\} \cup$

(*nfa'.step_eps_closure_set {q0'} bs ∪ right.nfa'.step_eps_closure_set {q0''} bs*)
⟨*proof*⟩

lemmas *run_accept_eps_Nil_cong* =
run_accept_eps_Nil_eps_split[*OF step_eps_closure_set_q0 step_symb_set_q0 qf_not_q0*,
unfolded run_accept_eps_split
run_accept_eps_cong[*OF nfa'.step_eps_closure_set_closed*[*OF nfa'.q0_sub_Q*]]
right.run_accept_eps_cong[*OF right.nfa'.step_eps_closure_set_closed*[*OF right.nfa'.q0_sub_Q*]]
run_accept_eps_Nil_eps]

lemmas *run_accept_eps_Cons_cong* =
run_accept_eps_Cons_eps_split[*OF step_eps_closure_set_q0 step_symb_set_q0 qf_not_q0*,
unfolded run_accept_eps_split
run_accept_eps_cong[*OF nfa'.step_eps_closure_set_closed*[*OF nfa'.q0_sub_Q*]]
right.run_accept_eps_cong[*OF right.nfa'.step_eps_closure_set_closed*[*OF right.nfa'.q0_sub_Q*]]
run_accept_eps_Cons_eps]

lemma *run_accept_eps_cong*: *run_accept_eps {q0} bss bs* ⟷
(*nfa'.run_accept_eps {q0'} bss bs* ∨ *right.nfa'.run_accept_eps {q0''} bss bs*)
⟨*proof*⟩

end

locale *nfa_cong_Times* = *nfa_cong' q0 q0 qf q0' transs transs'* +
right: *nfa_cong q0 q0' qf qf transs transs''*
for *q0 q0' qf transs transs' transs''*
begin

lemmas *run_accept_eps_cong* =
run_accept_eps_cong[*OF nfa'.q0_sub_Q*, *unfolded*
right.run_accept_eps_cong[*OF right.nfa'.q0_sub_Q*], *unfolded list_split_def*, *simplified*]

end

locale *nfa_cong_Star* = *nfa_cong' q0 q0' qf q0 transs transs'*
for *q0 q0' qf transs transs'* +
assumes *step_eps_q0*: *step_eps bs q0 q* ⟷ *q* ∈ {*q0'*, *qf*} **and**
step_symb_q0: ¬*step_symb q0 q*
begin

lemma *step_symb_set_q0*: *step_symb_set {q0}* = {}
⟨*proof*⟩

lemma *run_accept_eps_Nil*: *run_accept_eps {q0}* [] *bs*
⟨*proof*⟩

lemma *rtranclp_step_eps_q0_q0'*: (*step_eps bs*)** *q q'* ⟹ *q* = *q0* ⟹
q' ∈ {*q0*, *qf*} ∨ (*q'* ∈ *nfa'.SQ* ∧ (*nfa'.step_eps bs*)** *q0' q'*)
⟨*proof*⟩

lemma *step_eps_closure_set_q0*: *step_eps_closure_set {q0} bs* ⊆ {*q0*, *qf*} ∪
(*nfa'.step_eps_closure_set {q0'} bs* ∩ *nfa'.SQ*)
⟨*proof*⟩

lemma *delta_sub_nfa'_delta*: *delta {q0} bs* ⊆ *nfa'.delta {q0'} bs*
⟨*proof*⟩

lemma *step_eps_closure_set_q0_split*: *step_eps_closure_set {q0} bs* = {*q0*, *qf*} ∪

step_eps_closure_set {*q0'*} *bs*
⟨*proof*⟩

lemma *delta_q0_q0'*: *delta* {*q0*} *bs* = *delta* {*q0'*} *bs*
⟨*proof*⟩

lemmas *run_accept_eps_cong_Cons* =
run_accept_eps_cong_Cons_sub_simp[*OF nfa'.q0_sub_Q delta_sub_nfa'_delta*,
unfolded run_accept_eps_Cons_delta_cong[*OF delta_q0_q0'*, *symmetric*]]

end

end
theory *Window*
imports *HOL−Library.AList HOL−Library.Mapping HOL−Library.While_Combinator Timestamp*
begin

type_synonym ('*a*, '*b*) *mmap* = ('*a* × '*b*) *list*

inductive *chain_le* :: '*d* :: *timestamp list* ⇒ *bool* **where**
chain_le_Nil: *chain_le* []
| *chain_le_singleton*: *chain_le* [*x*]
| *chain_le_cons*: *chain_le* (*y* # *xs*) ⟹ *x* ≤ *y* ⟹ *chain_le* (*x* # *y* # *xs*)

lemma *chain_le_app*: *chain_le* (*zs* @ [*z*]) ⟹ *z* ≤ *w* ⟹ *chain_le* ((*zs* @ [*z*]) @ [*w*])
⟨*proof*⟩

inductive *reaches_on* :: ('*e* ⇒ ('*e* × '*f*) *option*) ⇒ '*e* ⇒ '*f list* ⇒ '*e* ⇒ *bool*
for *run* :: '*e* ⇒ ('*e* × '*f*) *option* **where**
reaches_on run s [] *s*
| *run s* = *Some* (*s'*, *v*) ⟹ *reaches_on run s' vs s''* ⟹ *reaches_on run s* (*v* # *vs*) *s''*

lemma *reaches_on_init_Some*: *reaches_on r s xs s'* ⟹ *r s'* ≠ *None* ⟹ *r s* ≠ *None*
⟨*proof*⟩

lemma *reaches_on_split*: *reaches_on run s vs s'* ⟹ *i* < *length vs* ⟹
∃ *s'' s'''*. *reaches_on run s* (*take i vs*) *s''* ∧ *run s''* = *Some* (*s'''*, *vs* ! *i*) ∧ *reaches_on run s'''* (*drop* (*Suc i*) *vs*) *s'*
⟨*proof*⟩

lemma *reaches_on_split'*: *reaches_on run s vs s'* ⟹ *i* ≤ *length vs* ⟹
∃ *s''* . *reaches_on run s* (*take i vs*) *s''* ∧ *reaches_on run s''* (*drop i vs*) *s'*
⟨*proof*⟩

lemma *reaches_on_split_app*: *reaches_on run s* (*vs* @ *vs'*) *s'* ⟹
∃ *s''*. *reaches_on run s vs s''* ∧ *reaches_on run s'' vs' s'*
⟨*proof*⟩

lemma *reaches_on_inj*: *reaches_on run s vs t* ⟹ *reaches_on run s vs' t'* ⟹
length vs = *length vs'* ⟹ *vs* = *vs'* ∧ *t* = *t'*
⟨*proof*⟩

lemma *reaches_on_split_last*: *reaches_on run s* (*xs* @ [*x*]) *s''* ⟹
∃ *s'*. *reaches_on run s xs s'* ∧ *run s'* = *Some* (*s''*, *x*)
⟨*proof*⟩

lemma *reaches_on_rev_induct*[*consumes 1*]: *reaches_on run s vs s'* $\Longrightarrow$
(⋀*s*. *P s* [] *s*) $\Longrightarrow$
(⋀*s s' v vs s''*. *reaches_on run s vs s'* $\Longrightarrow$ *P s vs s'* $\Longrightarrow$ *run s'* = *Some* (*s''*, *v*) $\Longrightarrow$
P s (*vs* @ [*v*]) *s''*) $\Longrightarrow$
P s vs s'
⟨*proof*⟩

lemma *reaches_on_app*: *reaches_on run s vs s'* $\Longrightarrow$ *run s'* = *Some* (*s''*, *v*) $\Longrightarrow$
reaches_on run s (*vs* @ [*v*]) *s''*
⟨*proof*⟩

lemma *reaches_on_trans*: *reaches_on run s vs s'* $\Longrightarrow$ *reaches_on run s' vs' s''* $\Longrightarrow$
reaches_on run s (*vs* @ *vs'*) *s''*
⟨*proof*⟩

lemma *reaches_onD*: *reaches_on run s* ((*t*, *b*) # *vs*) *s'* $\Longrightarrow$
∃ *s''*. *run s* = *Some* (*s''*, (*t*, *b*)) ∧ *reaches_on run s'' vs s'*
⟨*proof*⟩

lemma *reaches_on_setD*: *reaches_on run s vs s'* $\Longrightarrow$ *x* ∈ *set vs* $\Longrightarrow$
∃ *vs' vs'' s''*. *reaches_on run s* (*vs'* @ [*x*]) *s''* ∧ *reaches_on run s'' vs'' s'* ∧ *vs* = *vs'* @ *x* # *vs''*
⟨*proof*⟩

lemma *reaches_on_len*: ∃ *vs s'*. *reaches_on run s vs s'* ∧ (*length vs* = *n* ∨ *run s'* = *None*)
⟨*proof*⟩

lemma *reaches_on_NilD*: *reaches_on run q* [] *q'* $\Longrightarrow$ *q* = *q'*
⟨*proof*⟩

lemma *reaches_on_ConsD*: *reaches_on run q* (*x* # *xs*) *q'* $\Longrightarrow$ ∃ *q''*. *run q* = *Some* (*q''*, *x*) ∧ *reaches_on run q'' xs q'*
⟨*proof*⟩

inductive *reaches* :: ('*e* ⇒ ('*e* × '*f*) *option*) ⇒ '*e* ⇒ *nat* ⇒ '*e* ⇒ *bool*
for *run* :: '*e* ⇒ ('*e* × '*f*) *option* **where**
reaches run s 0 s
| *run s* = *Some* (*s'*, *v*) $\Longrightarrow$ *reaches run s' n s''* $\Longrightarrow$ *reaches run s* (*Suc n*) *s''*

lemma *reaches_Suc_split_last*: *reaches run s* (*Suc n*) *s'* $\Longrightarrow$ ∃ *s'' x*. *reaches run s n s''* ∧ *run s''* = *Some* (*s'*, *x*)
⟨*proof*⟩

lemma *reaches_invar*: *reaches f x n y* $\Longrightarrow$ *P x* $\Longrightarrow$ (⋀*z z' v*. *P z* $\Longrightarrow$ *f z* = *Some* (*z'*, *v*) $\Longrightarrow$ *P z'*) $\Longrightarrow$ *P y*
⟨*proof*⟩

lemma *reaches_cong*: *reaches f x n y* $\Longrightarrow$ *P x* $\Longrightarrow$ (⋀*z z' v*. *P z* $\Longrightarrow$ *f z* = *Some* (*z'*, *v*) $\Longrightarrow$ *P z'*) $\Longrightarrow$ (⋀*z*. *P z* $\Longrightarrow$ *f'* (*g z*) = *map_option* (*apfst g*) (*f z*)) $\Longrightarrow$ *reaches f'* (*g x*) *n* (*g y*)
⟨*proof*⟩

lemma *reaches_on_n*: *reaches_on run s vs s'* $\Longrightarrow$ *reaches run s* (*length vs*) *s'*
⟨*proof*⟩

lemma *reaches_on*: *reaches run s n s'* $\Longrightarrow$ ∃ *vs*. *reaches_on run s vs s'* ∧ *length vs* = *n*
⟨*proof*⟩

definition *ts_at* :: ('*d* × '*b*) *list* ⇒ *nat* ⇒ '*d* **where**
ts_at rho i = *fst* (*rho* ! *i*)

definition *bs_at* :: $('d \times 'b)$ *list* $\Rightarrow$ *nat* $\Rightarrow$ $'b$ **where**
bs_at rho i = *snd* (*rho* ! *i*)

fun *sub_bs* :: $('d \times 'b)$ *list* $\Rightarrow$ *nat* $\times$ *nat* $\Rightarrow$ $'b$ *list* **where**
sub_bs rho (*i*, *j*) = *map* (*bs_at rho*) [*i*..<*j*]

definition *steps* :: $('c \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('d \times 'b)$ *list* $\Rightarrow$ $'c \Rightarrow$ *nat* $\times$ *nat* $\Rightarrow$ $'c$ **where**
steps step rho q ij = *foldl step q* (*sub_bs rho ij*)

definition *acc* :: $('c \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('c \Rightarrow$ *bool*$) \Rightarrow ('d \times 'b)$ *list* $\Rightarrow$
$'c \Rightarrow$ *nat* $\times$ *nat* $\Rightarrow$ *bool* **where**
acc step accept rho q ij = *accept* (*steps step rho q ij*)

definition *sup_acc* :: $('c \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('c \Rightarrow$ *bool*$) \Rightarrow ('d \times 'b)$ *list* $\Rightarrow$
$'c \Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ($'d \times$ *nat*) *option* **where**
sup_acc step accept rho q i j =
(*let* $L' = \{l \in \{i..<j\}.$ *acc step accept rho q* (*i*, *Suc l*)$\}$; *m* = *Max* L' *in*
if $L' = \{\}$ *then None else Some* (*ts_at rho m*, *m*))

definition *sup_leadsto* :: $'c \Rightarrow ('c \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('d \times 'b)$ *list* $\Rightarrow$
nat $\Rightarrow$ *nat* $\Rightarrow$ $'c \Rightarrow$ $'d$ *option* **where**
sup_leadsto init step rho i j q =
(*let* $L' = \{l.\ l < i \wedge$ *steps step rho init* (*l*, *j*) = *q*$\}$; *m* = *Max* L' *in*
if $L' = \{\}$ *then None else Some* (*ts_at rho m*))

definition *mmap_keys* :: ($'a$, $'b$) *mmap* $\Rightarrow$ $'a$ *set* **where**
mmap_keys kvs = *set* (*map fst kvs*)

definition *mmap_lookup* :: ($'a$, $'b$) *mmap* $\Rightarrow$ $'a \Rightarrow$ $'b$ *option* **where**
mmap_lookup = *map_of*

definition *valid_s* :: $'c \Rightarrow ('c \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('c \times 'b, 'c)$ *mapping* $\Rightarrow ('c \Rightarrow$ *bool*$) \Rightarrow$
($'d \times 'b$) *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* $\Rightarrow$ ($'c$, $'c \times$ ($'d \times$ *nat*) *option*) *mmap* $\Rightarrow$ *bool* **where**
valid_s init step st accept rho u i j s $\equiv$
($\forall$ *q bs. case Mapping.lookup st* (*q*, *bs*) *of None* $\Rightarrow$ *True* | *Some v* $\Rightarrow$ *step q bs* = *v*) $\wedge$
(*mmap_keys s* = $\{q.\ (\exists\, l \leq u.$ *steps step rho init* (*l*, *i*) = *q*$)\}$ $\wedge$ *distinct* (*map fst s*) $\wedge$
($\forall$ *q. case mmap_lookup s q of None* $\Rightarrow$ *True*
| *Some* (q', *tstp*) $\Rightarrow$ *steps step rho q* (*i*, *j*) = $q' \wedge$ *tstp* = *sup_acc step accept rho q i j*))

record ($'b$, $'c$, $'d$, $'t$, $'e$) *args* =
w_init :: $'c$
w_step :: $'c \Rightarrow 'b \Rightarrow 'c$
w_accept :: $'c \Rightarrow$ *bool*
w_run_t :: $'t \Rightarrow ('t \times 'd)$ *option*
w_read_t :: $'t \Rightarrow 'd$ *option*
w_run_sub :: $'e \Rightarrow ('e \times 'b)$ *option*

record ($'b$, $'c$, $'d$, $'t$, $'e$) *window* =
w_st :: ($'c \times 'b$, $'c$) *mapping*
w_ac :: ($'c$, *bool*) *mapping*
w_i :: *nat*
w_ti :: $'t$
w_si :: $'e$
w_j :: *nat*
w_tj :: $'t$
w_sj :: $'e$
w_s :: ($'c$, $'c \times$ ($'d \times$ *nat*) *option*) *mmap*

$w\_e :: ('c, 'd)\ mmap$

copy_bnf $(dead\ 'b, dead\ 'c, dead\ 'd, dead\ 't, 'e, dead\ 'ext)\ window\_ext$

fun $reach\_window :: ('b, 'c, 'd, 't, 'e)\ args \Rightarrow 't \Rightarrow 'e \Rightarrow$
$('d \times 'b)\ list \Rightarrow nat \times 't \times 'e \times nat \times 't \times 'e \Rightarrow bool$ **where**
$reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj) \longleftrightarrow i \leq j \wedge length\ rho = j \wedge$
$reaches\_on\ (w\_run\_t\ args)\ t0\ (take\ i\ (map\ fst\ rho))\ ti \wedge$
$reaches\_on\ (w\_run\_t\ args)\ ti\ (drop\ i\ (map\ fst\ rho))\ tj \wedge$
$reaches\_on\ (w\_run\_sub\ args)\ sub\ (take\ i\ (map\ snd\ rho))\ si \wedge$
$reaches\_on\ (w\_run\_sub\ args)\ si\ (drop\ i\ (map\ snd\ rho))\ sj$

lemma $reach\_windowI: reaches\_on\ (w\_run\_t\ args)\ t0\ (take\ i\ (map\ fst\ rho))\ ti \Longrightarrow$
$reaches\_on\ (w\_run\_sub\ args)\ sub\ (take\ i\ (map\ snd\ rho))\ si \Longrightarrow$
$reaches\_on\ (w\_run\_t\ args)\ t0\ (map\ fst\ rho)\ tj \Longrightarrow$
$reaches\_on\ (w\_run\_sub\ args)\ sub\ (map\ snd\ rho)\ sj \Longrightarrow$
$i \leq length\ rho \Longrightarrow length\ rho = j \Longrightarrow$
$reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj)$
⟨*proof*⟩

lemma *reach_window_shift*:
assumes $reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj)\ i < j$
$w\_run\_t\ args\ ti = Some\ (ti', t)\ w\_run\_sub\ args\ si = Some\ (si', s)$
shows $reach\_window\ args\ t0\ sub\ rho\ (Suc\ i, ti', si', j, tj, sj)$
⟨*proof*⟩

lemma $reach\_window\_run\_ti: reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj) \Longrightarrow$
$i < j \Longrightarrow \exists ti'.\ reaches\_on\ (w\_run\_t\ args)\ t0\ (take\ i\ (map\ fst\ rho))\ ti \wedge$
$w\_run\_t\ args\ ti = Some\ (ti', ts\_at\ rho\ i) \wedge$
$reaches\_on\ (w\_run\_t\ args)\ ti'\ (drop\ (Suc\ i)\ (map\ fst\ rho))\ tj$
⟨*proof*⟩

lemma $reach\_window\_run\_si: reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj) \Longrightarrow$
$i < j \Longrightarrow \exists si'.\ reaches\_on\ (w\_run\_sub\ args)\ sub\ (take\ i\ (map\ snd\ rho))\ si \wedge$
$w\_run\_sub\ args\ si = Some\ (si', bs\_at\ rho\ i) \wedge$
$reaches\_on\ (w\_run\_sub\ args)\ si'\ (drop\ (Suc\ i)\ (map\ snd\ rho))\ sj$
⟨*proof*⟩

lemma $reach\_window\_run\_tj: reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj) \Longrightarrow$
$reaches\_on\ (w\_run\_t\ args)\ t0\ (map\ fst\ rho)\ tj$
⟨*proof*⟩

lemma $reach\_window\_run\_sj: reach\_window\ args\ t0\ sub\ rho\ (i, ti, si, j, tj, sj) \Longrightarrow$
$reaches\_on\ (w\_run\_sub\ args)\ sub\ (map\ snd\ rho)\ sj$
⟨*proof*⟩

lemma $reach\_window\_shift\_all: reach\_window\ args\ t0\ sub\ rho\ (i, si, ti, j, sj, tj) \Longrightarrow$
$reach\_window\ args\ t0\ sub\ rho\ (j, sj, tj, j, sj, tj)$
⟨*proof*⟩

lemma $reach\_window\_app: reach\_window\ args\ t0\ sub\ rho\ (i, si, ti, j, tj, sj) \Longrightarrow$
$w\_run\_t\ args\ tj = Some\ (tj', x) \Longrightarrow w\_run\_sub\ args\ sj = Some\ (sj', y) \Longrightarrow$
$reach\_window\ args\ t0\ sub\ (rho\ @\ [(x, y)])\ (i, si, ti, Suc\ j, tj', sj')$
⟨*proof*⟩

fun $init\_args :: ('c \times ('c \Rightarrow 'b \Rightarrow 'c) \times ('c \Rightarrow bool)) \Rightarrow$
$(('t \Rightarrow ('t \times 'd)\ option) \times ('t \Rightarrow 'd\ option)) \Rightarrow$
$('e \Rightarrow ('e \times 'b)\ option) \Rightarrow ('b, 'c, 'd, 't, 'e)\ args$ **where**

init_args (*init*, *step*, *accept*) (*run_t*, *read_t*) *run_sub* =
(|*w_init* = *init*, *w_step* = *step*, *w_accept* = *accept*, *w_run_t* = *run_t*, *w_read_t* = *read_t*, *w_run_sub* = *run_sub*|)

fun *init_window* :: ('*b*, '*c*, '*d*, '*t*, '*e*) *args* ⇒ '*t* ⇒ '*e* ⇒ ('*b*, '*c*, '*d*, '*t*, '*e*) *window* **where**
init_window args t0 sub = (|*w_st* = *Mapping.empty*, *w_ac* = *Mapping.empty*,
w_i = *0*, *w_ti* = *t0*, *w_si* = *sub*, *w_j* = *0*, *w_tj* = *t0*, *w_sj* = *sub*,
w_s =[(*w_init args*, (*w_init args*, *None*))], *w_e* = []|)

definition *valid_window* :: ('*b*, '*c*, '*d* :: *timestamp*, '*t*, '*e*) *args* ⇒ '*t* ⇒ '*e* ⇒ ('*d* × '*b*) *list* ⇒
('*b*, '*c*, '*d*, '*t*, '*e*) *window* ⇒ *bool* **where**
valid_window args t0 sub rho w ⟷
(*let init* = *w_init args*; *step* = *w_step args*; *accept* = *w_accept args*;
run_t = *w_run_t args*; *run_sub* = *w_run_sub args*;
st = *w_st w*; *ac* = *w_ac w*;
i = *w_i w*; *ti* = *w_ti w*; *si* = *w_si w*; *j* = *w_j w*; *tj* = *w_tj w*; *sj* = *w_sj w*;
s = *w_s w*; *e* = *w_e w in*
(*reach_window args t0 sub rho* (*i*, *ti*, *si*, *j*, *tj*, *sj*) ∧
(∀ *i j*. *i* ≤ *j* ∧ *j* < *length rho* ⟶ *ts_at rho i* ≤ *ts_at rho j*) ∧
(∀ *q*. *case Mapping.lookup ac q of None* ⇒ *True* | *Some v* ⇒ *accept q* = *v*) ∧
(∀ *q*. *mmap_lookup e q* = *sup_leadsto init step rho i j q*) ∧ *distinct* (*map fst e*) ∧
valid_s init step st accept rho i i j s))

lemma *valid_init_window*: *valid_window args t0 sub* [] (*init_window args t0 sub*)
⟨*proof*⟩

lemma *steps_app_cong*: *j* ≤ *length rho* ⟹ *steps step* (*rho* @ [*x*]) *q* (*i*, *j*) =
steps step rho q (*i*, *j*)
⟨*proof*⟩

lemma *acc_app_cong*: *j* < *length rho* ⟹ *acc step accept* (*rho* @ [*x*]) *q* (*i*, *j*) =
acc step accept rho q (*i*, *j*)
⟨*proof*⟩

lemma *sup_acc_app_cong*: *j* ≤ *length rho* ⟹ *sup_acc step accept* (*rho* @ [*x*]) *q i j* =
sup_acc step accept rho q i j
⟨*proof*⟩

lemma *sup_acc_concat_cong*: *j* ≤ *length rho* ⟹ *sup_acc step accept* (*rho* @ *rho'*) *q i j* =
sup_acc step accept rho q i j
⟨*proof*⟩

lemma *sup_leadsto_app_cong*: *i* ≤ *j* ⟹ *j* ≤ *length rho* ⟹
sup_leadsto init step (*rho* @ [*x*]) *i j q* = *sup_leadsto init step rho i j q*
⟨*proof*⟩

lemma *chain_le*:
fixes *xs* :: '*d* :: *timestamp list*
shows *chain_le xs* ⟹ *i* ≤ *j* ⟹ *j* < *length xs* ⟹ *xs* ! *i* ≤ *xs* ! *j*
⟨*proof*⟩

lemma *steps_refl*[*simp*]: *steps step rho q* (*i*, *i*) = *q*
⟨*proof*⟩

lemma *steps_split*: *i* < *j* ⟹ *steps step rho q* (*i*, *j*) =
steps step rho (*step q* (*bs_at rho i*)) (*Suc i*, *j*)
⟨*proof*⟩

lemma *steps_app*: $i \leq j \Longrightarrow$ *steps step rho q* $(i, j + 1) =$
step (*steps step rho q* (i, j)) (*bs_at rho j*)
⟨*proof*⟩

lemma *steps_appE*: $i \leq j \Longrightarrow$ *steps step rho q* $(i,$ *Suc j*$) = q' \Longrightarrow$
$\exists\, q''.$ *steps step rho q* $(i, j) = q'' \wedge q' =$ *step* q'' (*bs_at rho j*)
⟨*proof*⟩

lemma *steps_comp*: $i \leq l \Longrightarrow l \leq j \Longrightarrow$ *steps step rho q* $(i, l) = q' \Longrightarrow$
steps step rho q' $(l, j) = q'' \Longrightarrow$ *steps step rho q* $(i, j) = q''$
⟨*proof*⟩

lemma *sup_acc_SomeI*: *acc step accept rho q* $(i,$ *Suc l*$) \Longrightarrow l \in \{i..<j\} \Longrightarrow$
$\exists$ *tp*. *sup_acc step accept rho q i j* = *Some* (*ts_at rho tp*, *tp*) $\wedge\ l \leq tp \wedge tp < j$
⟨*proof*⟩

lemma *sup_acc_Some_ts*: *sup_acc step accept rho q i j* = *Some* $(ts,$ *tp*$) \Longrightarrow ts =$ *ts_at rho tp*
⟨*proof*⟩

lemma *sup_acc_SomeE*: *sup_acc step accept rho q i j* = *Some* $(ts,$ *tp*$) \Longrightarrow$
$tp \in \{i..<j\} \wedge$ *acc step accept rho q* $(i,$ *Suc tp*$)$
⟨*proof*⟩

lemma *sup_acc_NoneE*: $l \in \{i..<j\} \Longrightarrow$ *sup_acc step accept rho q i j* = *None* $\Longrightarrow$
$\neg$*acc step accept rho q* $(i,$ *Suc l*$)$
⟨*proof*⟩

lemma *sup_acc_same*: *sup_acc step accept rho q i i* = *None*
⟨*proof*⟩

lemma *sup_acc_None_restrict*: $i \leq j \Longrightarrow$ *sup_acc step accept rho q i j* = *None* $\Longrightarrow$
sup_acc step accept rho (*step q* (*bs_at rho i*)) (*Suc i*) *j* = *None*
⟨*proof*⟩

lemma *sup_acc_ext_idle*: $i \leq j \Longrightarrow \neg$*acc step accept rho q* $(i,$ *Suc j*$) \Longrightarrow$
sup_acc step accept rho q i (*Suc j*) = *sup_acc step accept rho q i j*
⟨*proof*⟩

lemma *sup_acc_comp_Some_ge*: $i \leq l \Longrightarrow l \leq j \Longrightarrow tp \geq l \Longrightarrow$
sup_acc step accept rho (*steps step rho q* (i, l)) *l j* = *Some* $(ts,$ *tp*$) \Longrightarrow$
sup_acc step accept rho q i j = *sup_acc step accept rho* (*steps step rho q* (i, l)) *l j*
⟨*proof*⟩

lemma *sup_acc_comp_None*: $i \leq l \Longrightarrow l \leq j \Longrightarrow$
sup_acc step accept rho (*steps step rho q* (i, l)) *l j* = *None* $\Longrightarrow$
sup_acc step accept rho q i j = *sup_acc step accept rho q i l*
⟨*proof*⟩

lemma *sup_acc_ext*: $i \leq j \Longrightarrow$ *acc step accept rho q* $(i,$ *Suc j*$) \Longrightarrow$
sup_acc step accept rho q i (*Suc j*) = *Some* (*ts_at rho j*, *j*)
⟨*proof*⟩

lemma *sup_acc_None*: $i < j \Longrightarrow$ *sup_acc step accept rho q i j* = *None* $\Longrightarrow$
sup_acc step accept rho (*step q* (*bs_at rho i*)) $(i + 1)$ *j* = *None*
⟨*proof*⟩

lemma *sup_acc_i*: $i < j \Longrightarrow$ *sup_acc step accept rho q i j* = *Some* $(ts,$ *i*$) \Longrightarrow$
sup_acc step accept rho (*step q* (*bs_at rho i*)) (*Suc i*) *j* = *None*

⟨*proof*⟩

lemma *sup_acc_l*: $i < j \Longrightarrow i \neq l \Longrightarrow$ *sup_acc step accept rho q i j* = *Some* $(ts, l) \Longrightarrow$
sup_acc step accept rho q i j = *sup_acc step accept rho* (*step q* (*bs_at rho i*)) (*Suc i*) *j*
⟨*proof*⟩

lemma *sup_leadsto_idle*: $i < j \Longrightarrow$ *steps step rho init* $(i, j) \neq q \Longrightarrow$
sup_leadsto init step rho i j q = *sup_leadsto init step rho* $(i + 1)$ *j q*
⟨*proof*⟩

lemma *sup_leadsto_SomeI*: $l < i \Longrightarrow$ *steps step rho init* $(l, j) = q \Longrightarrow$
$\exists\, l'$. *sup_leadsto init step rho i j q* = *Some* (*ts_at rho l'*) $\wedge\ l \leq l' \wedge l' < i$
⟨*proof*⟩

lemma *sup_leadsto_SomeE*: $i \leq j \Longrightarrow$ *sup_leadsto init step rho i j q* = *Some ts* $\Longrightarrow$
$\exists\, l < i$. *steps step rho init* $(l, j) = q \wedge$ *ts_at rho l* = *ts*
⟨*proof*⟩

lemma *Mapping_keys_dest*: $x \in$ *mmap_keys f* $\Longrightarrow \exists\, y$. *mmap_lookup f x* = *Some y*
⟨*proof*⟩

lemma *Mapping_keys_intro*: *mmap_lookup f x* $\neq$ *None* $\Longrightarrow x \in$ *mmap_keys f*
⟨*proof*⟩

lemma *Mapping_not_keys_intro*: *mmap_lookup f x* = *None* $\Longrightarrow x \notin$ *mmap_keys f*
⟨*proof*⟩

lemma *Mapping_lookup_None_intro*: $x \notin$ *mmap_keys f* $\Longrightarrow$ *mmap_lookup f x* = *None*
⟨*proof*⟩

primrec *mmap_combine* :: *'key* $\Rightarrow$ *'val* $\Rightarrow$ (*'val* $\Rightarrow$ *'val* $\Rightarrow$ *'val*) $\Rightarrow$ (*'key* $\times$ *'val*) *list* $\Rightarrow$
(*'key* $\times$ *'val*) *list*
where
mmap_combine k v c [] = [(k, v)]
| *mmap_combine k v c* (*p* # *ps*) = (*case p of* $(k', v') \Rightarrow$
if k = *k' then* $(k, c\ v'\ v)$ # *ps else p* # *mmap_combine k v c ps*)

lemma *mmap_combine_distinct_set*: *distinct* (*map fst r*) $\Longrightarrow$
distinct (*map fst* (*mmap_combine k v c r*)) $\wedge$
set (*map fst* (*mmap_combine k v c r*)) = *set* (*map fst r*) $\cup$ $\{k\}$
⟨*proof*⟩

lemma *mmap_combine_lookup*: *distinct* (*map fst r*) $\Longrightarrow$ *mmap_lookup* (*mmap_combine k v c r*) *z* =
(*if k* = *z then* (*case mmap_lookup r k of None* $\Rightarrow$ *Some v* | *Some v'* $\Rightarrow$ *Some* $(c\ v'\ v)$)
else mmap_lookup r z)
⟨*proof*⟩

definition *mmap_fold* :: $('c, 'd)$ *mmap* $\Rightarrow$ $(('c \times 'd) \Rightarrow ('c \times 'd)) \Rightarrow ('d \Rightarrow 'd \Rightarrow 'd) \Rightarrow$
$('c, 'd)$ *mmap* $\Rightarrow$ $('c, 'd)$ *mmap* **where**
mmap_fold m f c r = *foldl* $(\lambda r\ p$. *case f p of* $(k, v) \Rightarrow$ *mmap_combine k v c r*$)$ *r m*

definition *mmap_fold'* :: $('c, 'd)$ *mmap* $\Rightarrow$ *'e* $\Rightarrow$ $(('c \times 'd) \times 'e \Rightarrow ('c \times 'd) \times 'e) \Rightarrow$
$('d \Rightarrow 'd \Rightarrow 'd) \Rightarrow ('c, 'd)$ *mmap* $\Rightarrow ('c, 'd)$ *mmap* $\times$ *'e* **where**
mmap_fold' m e f c r = *foldl* $(\lambda(r, e)\ p$. *case f* (p, e) *of* $((k, v), e') \Rightarrow$
(*mmap_combine k v c r*, *e'*)) (r, e) *m*

lemma *mmap_fold'_eq*: *mmap_fold' m e f' c r* = $(m', e') \Longrightarrow$ *P e* $\Longrightarrow$
$(\bigwedge p\ e\ p'\ e'$. *P e* $\Longrightarrow$ *f'* $(p, e) = (p', e') \Longrightarrow p' = f\ p \wedge P\ e') \Longrightarrow$

$m' = mmap\_fold\ m\ f\ c\ r \wedge P\ e'$
⟨*proof*⟩

lemma *foldl_mmap_combine_distinct_set*: $distinct\ (map\ fst\ r) \Longrightarrow$
$distinct\ (map\ fst\ (mmap\_fold\ m\ f\ c\ r)) \wedge$
$set\ (map\ fst\ (mmap\_fold\ m\ f\ c\ r)) = set\ (map\ fst\ r) \cup set\ (map\ (fst \circ f)\ m)$
⟨*proof*⟩

lemma *mmap_fold_lookup_rec*: $distinct\ (map\ fst\ r) \Longrightarrow mmap\_lookup\ (mmap\_fold\ m\ f\ c\ r)\ z =$
$(case\ map\ (snd \circ f)\ (filter\ (\lambda(k, v).\ fst\ (f\ (k, v)) = z)\ m)\ of\ [] \Rightarrow mmap\_lookup\ r\ z$
$\mid v\ \#\ vs \Rightarrow (case\ mmap\_lookup\ r\ z\ of\ None \Rightarrow Some\ (foldl\ c\ v\ vs)$
$\mid Some\ w \Rightarrow Some\ (foldl\ c\ w\ (v\ \#\ vs))))$
⟨*proof*⟩

lemma *mmap_fold_distinct*: $distinct\ (map\ fst\ m) \Longrightarrow distinct\ (map\ fst\ (mmap\_fold\ m\ f\ c\ []))$
⟨*proof*⟩

lemma *mmap_fold_set*: $distinct\ (map\ fst\ m) \Longrightarrow$
$set\ (map\ fst\ (mmap\_fold\ m\ f\ c\ [])) = (fst \circ f)\ `\ set\ m$
⟨*proof*⟩

lemma *mmap_lookup_empty*: $mmap\_lookup\ []\ z = None$
⟨*proof*⟩

lemma *mmap_fold_lookup*: $distinct\ (map\ fst\ m) \Longrightarrow mmap\_lookup\ (mmap\_fold\ m\ f\ c\ [])\ z =$
$(case\ map\ (snd \circ f)\ (filter\ (\lambda(k, v).\ fst\ (f\ (k, v)) = z)\ m)\ of\ [] \Rightarrow None$
$\mid v\ \#\ vs \Rightarrow Some\ (foldl\ c\ v\ vs))$
⟨*proof*⟩

definition *fold_sup* :: $('c, 'd :: timestamp)\ mmap \Rightarrow ('c \Rightarrow 'c) \Rightarrow ('c, 'd)\ mmap$ **where**
$fold\_sup\ m\ f = mmap\_fold\ m\ (\lambda(x, y).\ (f\ x, y))\ sup\ []$

lemma *mmap_lookup_distinct*: $distinct\ (map\ fst\ m) \Longrightarrow (k, v) \in set\ m \Longrightarrow$
$mmap\_lookup\ m\ k = Some\ v$
⟨*proof*⟩

lemma *fold_sup_distinct*: $distinct\ (map\ fst\ m) \Longrightarrow distinct\ (map\ fst\ (fold\_sup\ m\ f))$
⟨*proof*⟩

lemma *fold_sup*:
fixes $v :: 'd :: timestamp$
shows $foldl\ sup\ v\ vs = fold\ sup\ vs\ v$
⟨*proof*⟩

lemma *lookup_fold_sup*:
assumes *distinct*: $distinct\ (map\ fst\ m)$
shows $mmap\_lookup\ (fold\_sup\ m\ f)\ z =$
$(let\ Z = \{x \in mmap\_keys\ m.\ f\ x = z\}\ in$
$if\ Z = \{\}\ then\ None\ else\ Some\ (Sup\_fin\ ((the \circ mmap\_lookup\ m)\ `\ Z)))$
⟨*proof*⟩

definition *mmap_map* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a, 'b)\ mmap \Rightarrow ('a, 'c)\ mmap$ **where**
$mmap\_map\ f\ m = map\ (\lambda(k, v).\ (k, f\ k\ v))\ m$

lemma *mmap_map_keys*: $mmap\_keys\ (mmap\_map\ f\ m) = mmap\_keys\ m$
⟨*proof*⟩

lemma *mmap_map_fst*: $map\ fst\ (mmap\_map\ f\ m) = map\ fst\ m$

⟨*proof*⟩

definition *cstep* :: ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*c* × '*b*, '*c*) *mapping* ⇒
'*c* ⇒ '*b* ⇒ ('*c* × ('*c* × '*b*, '*c*) *mapping*) **where**
cstep step st q bs = (*case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ (*let res* = *step q bs in*
(*res*, *Mapping.update* (*q*, *bs*) *res st*)) | *Some v* ⇒ (*v*, *st*))

definition *cac* :: ('*c* ⇒ *bool*) ⇒ ('*c*, *bool*) *mapping* ⇒ '*c* ⇒ (*bool* × ('*c*, *bool*) *mapping*) **where**
cac accept ac q = (*case Mapping.lookup ac q of None* ⇒ (*let res* = *accept q in*
(*res*, *Mapping.update q res ac*)) | *Some v* ⇒ (*v*, *ac*))

fun *mmap_fold_s* :: ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*c* × '*b*, '*c*) *mapping* ⇒
('*c* ⇒ *bool*) ⇒ ('*c*, *bool*) *mapping* ⇒
'*b* ⇒ '*d* ⇒ *nat* ⇒ ('*c*, '*c* × ('*d* × *nat*) *option*) *mmap* ⇒
(('*c*, '*c* × ('*d* × *nat*) *option*) *mmap* × ('*c* × '*b*, '*c*) *mapping* × ('*c*, *bool*) *mapping*) **where**
mmap_fold_s step st accept ac bs t j [] = ([], *st*, *ac*)
| *mmap_fold_s step st accept ac bs t j* ((*q*, (*q'*, *tstp*)) # *qbss*) =
(*let* (*q''*, *st'*) = *cstep step st q' bs*;
(*β*, *ac'*) = *cac accept ac q''*;
(*qbss'*, *st''*, *ac''*) = *mmap_fold_s step st' accept ac' bs t j qbss in*
((*q*, (*q''*, *if β then Some* (*t*, *j*) *else tstp*)) # *qbss'*, *st''*, *ac''*))

lemma *mmap_fold_s_sound*: *mmap_fold_s step st accept ac bs t j qbss* = (*qbss'*, *st'*, *ac'*) ⟹
(⋀*q bs*. *case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*) ⟹
(⋀*q bs*. *case Mapping.lookup ac q of None* ⇒ *True* | *Some v* ⇒ *accept q* = *v*) ⟹
qbss' = *mmap_map* (λ*q* (*q'*, *tstp*). (*step q' bs*, *if accept* (*step q' bs*) *then Some* (*t*, *j*) *else tstp*)) *qbss* ∧
(∀ *q bs*. *case Mapping.lookup st'* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*) ∧
(∀ *q bs*. *case Mapping.lookup ac' q of None* ⇒ *True* | *Some v* ⇒ *accept q* = *v*)
⟨*proof*⟩

definition *adv_end* :: ('*b*, '*c*, '*d* :: *timestamp*, '*t*, '*e*) *args* ⇒
('*b*, '*c*, '*d*, '*t*, '*e*) *window* ⇒ ('*b*, '*c*, '*d*, '*t*, '*e*) *window option* **where**
adv_end args w = (*let step* = *w_step args*; *accept* = *w_accept args*;
run_t = *w_run_t args*; *run_sub* = *w_run_sub args*; *st* = *w_st w*; *ac* = *w_ac w*;
j = *w_j w*; *tj* = *w_tj w*; *sj* = *w_sj w*; *s* = *w_s w*; *e* = *w_e w in*
(*case run_t tj of None* ⇒ *None* | *Some* (*tj'*, *t*) ⇒ (*case run_sub sj of None* ⇒ *None* | *Some* (*sj'*, *bs*)
⇒
let (*s'*, *st'*, *ac'*) = *mmap_fold_s step st accept ac bs t j s*;
(*e'*, *st''*) = *mmap_fold' e st'* (λ((*x*, *y*), *st*). *let* (*q'*, *st'*) = *cstep step st x bs in* ((*q'*, *y*), *st'*)) *sup* [] *in*
Some (*w*(|*w_st* := *st''*, *w_ac* := *ac'*, *w_j* := *Suc j*, *w_tj* := *tj'*, *w_sj* := *sj'*, *w_s* := *s'*, *w_e* :=
e'|)))))

lemma *map_values_lookup*: *mmap_lookup* (*mmap_map f m*) *z* = *Some v'* ⟹
∃ *v*. *mmap_lookup m z* = *Some v* ∧ *v'* = *f z v*
⟨*proof*⟩

lemma *acc_app*:
assumes *i* ≤ *j steps step rho q* (*i*, *Suc j*) = *q' accept q'*
shows *sup_acc step accept rho q i* (*Suc j*) = *Some* (*ts_at rho j*, *j*)
⟨*proof*⟩

lemma *acc_app_idle*:
assumes *i* ≤ *j steps step rho q* (*i*, *Suc j*) = *q'* ¬*accept q'*
shows *sup_acc step accept rho q i* (*Suc j*) = *sup_acc step accept rho q i j*
⟨*proof*⟩

lemma *sup_fin_closed*: *finite A* ⟹ *A* ≠ {} ⟹
(⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *sup x y* ∈ {*x*, *y*}) ⟹ ⨆$_{fin}$ *A* ∈ *A*

⟨*proof*⟩

lemma *valid_adv_end*:
assumes *valid_window args t0 sub rho w w_run_t args* (*w_tj w*) = *Some* (*tj'*, *t*)
w_run_sub args (*w_sj w*) = *Some* (*sj'*, *bs*)
⋀*t'*. *t'* ∈ *set* (*map fst rho*) ⟹ *t'* ≤ *t*
shows *case adv_end args w of None* ⇒ *False* | *Some w'* ⇒ *valid_window args t0 sub* (*rho* @ [(*t*, *bs*)]) *w'*
⟨*proof*⟩

lemma *adv_end_bounds*:
assumes *w_run_t args* (*w_tj w*) = *Some* (*tj'*, *t*)
w_run_sub args (*w_sj w*) = *Some* (*sj'*, *bs*)
adv_end args w = *Some w'*
shows *w_i w'* = *w_i w w_ti w'* = *w_ti w w_si w'* = *w_si w*
w_j w' = *Suc* (*w_j w*) *w_tj w'* = *tj' w_sj w'* = *sj'*
⟨*proof*⟩

definition *drop_cur* :: *nat* ⇒ (*'c* × (*'d* × *nat*) *option*) ⇒ (*'c* × (*'d* × *nat*) *option*) **where**
drop_cur i = (λ(*q'*, *tstp*). (*q'*, *case tstp of Some* (*ts*, *tp*) ⇒
if tp = *i then None else tstp* | *None* ⇒ *tstp*))

definition *adv_d* :: (*'c* ⇒ *'b* ⇒ *'c*) ⇒ (*'c* × *'b*, *'c*) *mapping* ⇒ *nat* ⇒ *'b* ⇒
(*'c*, *'c* × (*'d* × *nat*) *option*) *mmap* ⇒
((*'c*, *'c* × (*'d* × *nat*) *option*) *mmap* × (*'c* × *'b*, *'c*) *mapping*) **where**
adv_d step st i b s = (*mmap_fold' s st* (λ((*x*, *v*), *st*). *case cstep step st x b of* (*x'*, *st'*) ⇒
((*x'*, *drop_cur i v*), *st'*)) (λ*x y*. *x*) []))

lemma *adv_d_mmap_fold*:
assumes *inv*: ⋀*q bs*. *case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*
and *fold'*: *mmap_fold' s st* (λ((*x*, *v*), *st*). *case cstep step st x bs of* (*x'*, *st'*) ⇒
((*x'*, *drop_cur i v*), *st'*)) (λ*x y*. *x*) *r* = (*s'*, *st'*)
shows *s'* = *mmap_fold s* (λ(*x*, *v*). (*step x bs*, *drop_cur i v*)) (λ*x y*. *x*) *r* ∧
(∀ *q bs*. *case Mapping.lookup st'* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*)
⟨*proof*⟩

definition *keys_idem* :: (*'c* ⇒ *'b* ⇒ *'c*) ⇒ *nat* ⇒ *'b* ⇒
(*'c*, *'c* × (*'d* × *nat*) *option*) *mmap* ⇒ *bool* **where**
keys_idem step i b s = (∀ *x* ∈ *mmap_keys s*. ∀ *x'* ∈ *mmap_keys s*.
step x b = *step x' b* ⟶ *drop_cur i* (*the* (*mmap_lookup s x*)) =
drop_cur i (*the* (*mmap_lookup s x'*)))

lemma *adv_d_keys*:
assumes *inv*: ⋀*q bs*. *case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*
and *distinct*: *distinct* (*map fst s*)
and *adv_d*: *adv_d step st i bs s* = (*s'*, *st'*)
shows *mmap_keys s'* = (λ*q*. *step q bs*) ‘ (*mmap_keys s*)
⟨*proof*⟩

lemma *lookup_adv_d_None*:
assumes *inv*: ⋀*q bs*. *case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*
and *distinct*: *distinct* (*map fst s*)
and *adv_d*: *adv_d step st i bs s* = (*s'*, *st'*)
and *Z_empty*: {*x* ∈ *mmap_keys s*. *step x bs* = *z*} = {}
shows *mmap_lookup s' z* = *None*
⟨*proof*⟩

lemma *lookup_adv_d_Some*:

assumes *inv*: $\bigwedge q\ bs.\ case\ Mapping.lookup\ st\ (q,\ bs)\ of\ None \Rightarrow True \mid Some\ v \Rightarrow step\ q\ bs = v$
and *distinct*: *distinct* (*map fst s*) **and** *idem*: *keys_idem step i bs s*
and *wit*: $x \in mmap\_keys\ s\ step\ x\ bs = z$
and *adv_d*: *adv_d step st i bs s* = $(s',\ st')$
shows *mmap_lookup* $s'\ z = Some\ (drop\_cur\ i\ (the\ (mmap\_lookup\ s\ x)))$
⟨*proof*⟩

definition *loop_cond j* = $(\lambda(st,\ ac,\ i,\ ti,\ si,\ q,\ s,\ tstp).\ i < j \wedge q \notin mmap\_keys\ s)$
definition *loop_body step accept run_t run_sub* =
$(\lambda(st,\ ac,\ i,\ ti,\ si,\ q,\ s,\ tstp).\ case\ run\_t\ ti\ of\ Some\ (ti',\ t) \Rightarrow$
case run_sub si of Some $(si',\ b) \Rightarrow case\ adv\_d\ step\ st\ i\ b\ s\ of\ (s',\ st') \Rightarrow$
case cstep step $st'\ q\ b\ of\ (q',\ st'') \Rightarrow case\ cac\ accept\ ac\ q'\ of\ (\beta,\ ac') \Rightarrow$
$(st'',\ ac',\ Suc\ i,\ ti',\ si',\ q',\ s',\ if\ \beta\ then\ Some\ (t,\ i)\ else\ tstp))$
definition *loop_inv init step accept args t0 sub rho u j tj sj* =
$(\lambda(st,\ ac,\ i,\ ti,\ si,\ q,\ s,\ tstp).\ u + 1 \le i \wedge$
reach_window args t0 sub rho $(i,\ ti,\ si,\ j,\ tj,\ sj) \wedge$
steps step rho init $(u + 1,\ i) = q \wedge$
$(\forall q.\ case\ Mapping.lookup\ ac\ q\ of\ None \Rightarrow True \mid Some\ v \Rightarrow accept\ q = v) \wedge$
valid_s init step st accept rho u i j s $\wedge$ *tstp* = *sup_acc step accept rho init* $(u + 1)\ i)$

definition *mmap_update* :: $'a \Rightarrow 'b \Rightarrow ('a,\ 'b)\ mmap \Rightarrow ('a,\ 'b)\ mmap$ **where**
mmap_update = *AList.update*

lemma *mmap_update_distinct*: *distinct* (*map fst m*) $\Longrightarrow$ *distinct* (*map fst* (*mmap_update k v m*))
⟨*proof*⟩

definition *adv_start* :: $('b,\ 'c,\ 'd :: timestamp,\ 't,\ 'e)\ args \Rightarrow$
$('b,\ 'c,\ 'd,\ 't,\ 'e)\ window \Rightarrow ('b,\ 'c,\ 'd,\ 't,\ 'e)\ window$ **where**
adv_start args w = (*let init* = *w_init args*; *step* = *w_step args*; *accept* = *w_accept args*;
run_t = *w_run_t args*; *run_sub* = *w_run_sub args*; *st* = *w_st w*; *ac* = *w_ac w*;
i = *w_i w*; *ti* = *w_ti w*; *si* = *w_si w*; *j* = *w_j w*;
s = *w_s w*; *e* = *w_e w in*
(*case run_t ti of Some* $(ti',\ t) \Rightarrow$ (*case run_sub si of Some* $(si',\ bs) \Rightarrow$
let $(s',\ st')$ = *adv_d step st i bs s*;
e' = *mmap_update* (*fst* (*the* (*mmap_lookup s init*))) *t e*;
$(st\_cur,\ ac\_cur,\ i\_cur,\ ti\_cur,\ si\_cur,\ q\_cur,\ s\_cur,\ tstp\_cur)$ =
while (*loop_cond j*) (*loop_body step accept run_t run_sub*)
$(st',\ ac,\ Suc\ i,\ ti',\ si',\ init,\ s',\ None)$;
s'' = *mmap_update init* (*case mmap_lookup s_cur q_cur of Some* $(q',\ tstp') \Rightarrow$
(*case tstp' of Some* $(ts,\ tp) \Rightarrow (q',\ tstp') \mid None \Rightarrow (q',\ tstp\_cur)$)
$\mid None \Rightarrow (q\_cur,\ tstp\_cur))\ s'$ *in*
$w(\!|w\_st := st\_cur,\ w\_ac := ac\_cur,\ w\_i := Suc\ i,\ w\_ti := ti',\ w\_si := si',$
$w\_s := s'',\ w\_e := e'|\!)))$

lemma *valid_adv_d*:
assumes *valid_before*: *valid_s init step st accept rho u i j s*
and u_le_i: $u \le i$ **and** i_lt_j: $i < j$ **and** *b_def*: $b = bs\_at\ rho\ i$
and *adv_d*: *adv_d step st i b s* = $(s',\ st')$
shows *valid_s init step* st' *accept rho u* $(i + 1)\ j\ s'$
⟨*proof*⟩

lemma *mmap_lookup_update'*:
mmap_lookup (*mmap_update k v kvs*) z = (*if* $k = z$ *then Some v else mmap_lookup kvs z*)
⟨*proof*⟩

lemma *mmap_keys_update*: *mmap_keys* (*mmap_update k v kvs*) = *mmap_keys kvs* $\cup \{k\}$
⟨*proof*⟩

lemma *valid_adv_start*:
assumes *valid_window args t0 sub rho w w_i w < w_j w*
shows *valid_window args t0 sub rho (adv_start args w)*
⟨*proof*⟩

lemma *valid_adv_start_bounds*:
assumes *valid_window args t0 sub rho w w_i w < w_j w*
shows *w_i (adv_start args w) = Suc (w_i w) w_j (adv_start args w) = w_j w*
w_tj (adv_start args w) = w_tj w w_sj (adv_start args w) = w_sj w
⟨*proof*⟩

lemma *valid_adv_start_bounds'*:
assumes *valid_window args t0 sub rho w w_run_t args (w_ti w) = Some (ti', t)*
w_run_sub args (w_si w) = Some (si', bs)
shows *w_ti (adv_start args w) = ti' w_si (adv_start args w) = si'*
⟨*proof*⟩

end
theory *Temporal*
imports *MDL NFA Window*
begin

fun *state_cnt* :: ('*a*, '*b* :: *timestamp*) *regex* ⇒ *nat* **where**
state_cnt (Lookahead phi) = 1
| *state_cnt (Symbol phi) = 2*
| *state_cnt (Plus r s) = 1 + state_cnt r + state_cnt s*
| *state_cnt (Times r s) = state_cnt r + state_cnt s*
| *state_cnt (Star r) = 1 + state_cnt r*

lemma *state_cnt_pos*: *state_cnt r > 0*
⟨*proof*⟩

fun *collect_subfmlas* :: ('*a*, '*b* :: *timestamp*) *regex* ⇒ ('*a*, '*b*) *formula list* ⇒
('*a*, '*b*) *formula list* **where**
collect_subfmlas (Lookahead φ) phis = (if φ ∈ set phis then phis else phis @ [φ])
| *collect_subfmlas (Symbol φ) phis = (if φ ∈ set phis then phis else phis @ [φ])*
| *collect_subfmlas (Plus r s) phis = collect_subfmlas s (collect_subfmlas r phis)*
| *collect_subfmlas (Times r s) phis = collect_subfmlas s (collect_subfmlas r phis)*
| *collect_subfmlas (Star r) phis = collect_subfmlas r phis*

lemma *bf_collect_subfmlas*: *bounded_future_regex r ⟹ phi ∈ set (collect_subfmlas r phis) ⟹*
phi ∈ set phis ∨ bounded_future_fmla phi
⟨*proof*⟩

lemma *collect_subfmlas_atms*: *set (collect_subfmlas r phis) = set phis ∪ atms r*
⟨*proof*⟩

lemma *collect_subfmlas_set*: *set (collect_subfmlas r phis) = set (collect_subfmlas r []) ∪ set phis*
⟨*proof*⟩

lemma *collect_subfmlas_size*: *x ∈ set (collect_subfmlas r []) ⟹ size x < size r*
⟨*proof*⟩

lemma *collect_subfmlas_app*: *∃ phis'. collect_subfmlas r phis = phis @ phis'*
⟨*proof*⟩

lemma *length_collect_subfmlas*: *length (collect_subfmlas r phis) ≥ length phis*
⟨*proof*⟩

fun *pos* :: *'a* ⇒ *'a list* ⇒ *nat option* **where**
  pos a [] = *None*
| *pos a* (*x* # *xs*) =
    (*if a* = *x then Some 0 else* (*case pos a xs of Some n* ⇒ *Some* (*Suc n*) | _ ⇒ *None*))

lemma *pos_sound*: *pos a xs* = *Some i* ⟹ *i* < *length xs* ∧ *xs* ! *i* = *a*
  ⟨*proof*⟩

lemma *pos_complete*: *pos a xs* = *None* ⟹ *a* ∉ *set xs*
  ⟨*proof*⟩

fun *build_nfa_impl* :: (*'a*, *'b* :: *timestamp*) *regex* ⇒ (*state* × *state* × (*'a*, *'b*) *formula list*) ⇒
  transition list **where**
  build_nfa_impl (*Lookahead* φ) (*q0*, *qf*, *phis*) = (*case pos* φ *phis of*
    Some n ⇒ [*eps_trans qf n*]
  | *None* ⇒ [*eps_trans qf* (*length phis*)])
| *build_nfa_impl* (*Symbol* φ) (*q0*, *qf*, *phis*) = (*case pos* φ *phis of*
    Some n ⇒ [*eps_trans* (*Suc q0*) *n*, *symb_trans qf*]
  | *None* ⇒ [*eps_trans* (*Suc q0*) (*length phis*), *symb_trans qf*])
| *build_nfa_impl* (*Plus r s*) (*q0*, *qf*, *phis*) = (
    let ts_r = *build_nfa_impl r* (*q0* + *1*, *qf*, *phis*);
        ts_s = *build_nfa_impl s* (*q0* + *1* + *state_cnt r*, *qf*, *collect_subfmlas r phis*) *in*
    split_trans (*q0* + *1*) (*q0* + *1* + *state_cnt r*) # *ts_r* @ *ts_s*)
| *build_nfa_impl* (*Times r s*) (*q0*, *qf*, *phis*) = (
    let ts_r = *build_nfa_impl r* (*q0*, *q0* + *state_cnt r*, *phis*);
        ts_s = *build_nfa_impl s* (*q0* + *state_cnt r*, *qf*, *collect_subfmlas r phis*) *in*
    ts_r @ *ts_s*)
| *build_nfa_impl* (*Star r*) (*q0*, *qf*, *phis*) = (
    let ts_r = *build_nfa_impl r* (*q0* + *1*, *q0*, *phis*) *in*
    split_trans (*q0* + *1*) *qf* # *ts_r*)

lemma *build_nfa_impl_state_cnt*: *length* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) = *state_cnt r*
  ⟨*proof*⟩

lemma *build_nfa_impl_not_Nil*: *build_nfa_impl r* (*q0*, *qf*, *phis*) ≠ []
  ⟨*proof*⟩

lemma *build_nfa_impl_state_set*: *t* ∈ *set* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) ⟹
  state_set t ⊆ {*q0*..<*q0* + *length* (*build_nfa_impl r* (*q0*, *qf*, *phis*))} ∪ {*qf*}
  ⟨*proof*⟩

lemma *build_nfa_impl_fmla_set*: *t* ∈ *set* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) ⟹
  n ∈ *fmla_set t* ⟹ *n* < *length* (*collect_subfmlas r phis*)
⟨*proof*⟩

context *MDL*
begin

definition *IH r q0 qf phis transs bss bs i* ≡
  let n = *length* (*collect_subfmlas r phis*) *in*
  transs = *build_nfa_impl r* (*q0*, *qf*, *phis*) ∧ (∀ *cs* ∈ *set bss*. *length cs* ≥ *n*) ∧ *length bs* ≥ *n* ∧
  qf ∉ *NFA.SQ q0* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) ∧
  (∀ *k* < *n*. (*bs* ! *k* ⟷ *sat* (*collect_subfmlas r phis* ! *k*) (*i* + *length bss*))) ∧
  (∀ *j* < *length bss*. ∀ *k* < *n*. ((*bss* ! *j*) ! *k* ⟷ *sat* (*collect_subfmlas r phis* ! *k*) (*i* + *j*)))

lemma *nfa_correct*: *IH r q0 qf phis transs bss bs i* ⟹
  NFA.run_accept_eps q0 qf transs {*q0*} *bss bs* ⟷ (*i*, *i* + *length bss*) ∈ *match r*

⟨*proof*⟩

lemma *step_eps_closure_set_empty_list*:
assumes *wf_regex r IH r q0 qf phis transs bss bs i NFA.step_eps_closure q0 transs bs q qf*
shows *NFA.step_eps_closure q0 transs [] q qf*
⟨*proof*⟩

lemma *accept_eps_iff_accept*:
assumes *wf_regex r IH r q0 qf phis transs bss bs i*
shows *NFA.accept_eps q0 qf transs R bs = NFA.accept q0 qf transs R*
⟨*proof*⟩

lemma *run_accept_eps_iff_run_accept*:
assumes *wf_regex r IH r q0 qf phis transs bss bs i*
shows *NFA.run_accept_eps q0 qf transs {q0} bss bs ⟷ NFA.run_accept q0 qf transs {q0} bss*
⟨*proof*⟩

end

definition *pred_option'* :: (*'a* ⇒ *bool*) ⇒ *'a option* ⇒ *bool* **where**
pred_option' P z = (*case z of Some z'* ⇒ *P z'* | *None* ⇒ *False*)

definition *map_option'* :: (*'b* ⇒ *'c option*) ⇒ *'b option* ⇒ *'c option* **where**
map_option' f z = (*case z of Some z'* ⇒ *f z'* | *None* ⇒ *None*)

definition *while_break* :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a option*) ⇒ *'a* ⇒ *'a option* **where**
while_break P f x = *while* (*pred_option' P*) (*map_option' f*) (*Some x*)

lemma *wf_while_break*:
assumes *wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *Some t* = *c s*}
shows *wf* {(*t*, *s*). *pred_option P s* ∧ *pred_option' b s* ∧ *t* = *map_option' c s*}
⟨*proof*⟩

lemma *wf_while_break'*:
assumes *wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *Some t* = *c s*}
shows *wf* {(*t*, *s*). *pred_option' P s* ∧ *pred_option' b s* ∧ *t* = *map_option' c s*}
⟨*proof*⟩

lemma *while_break_sound*:
assumes ⋀*s s'*. *P s* ⟹ *b s* ⟹ *c s* = *Some s'* ⟹ *P s'* ⋀*s*. *P s* ⟹ ¬ *b s* ⟹ *Q s wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *Some t* = *c s*} *P s*
shows *pred_option Q* (*while_break b c s*)
⟨*proof*⟩

lemma *while_break_complete*: (⋀*s*. *P s* ⟹ *b s* ⟹ *pred_option' P* (*c s*)) ⟹ (⋀*s*. *P s* ⟹ ¬ *b s* ⟹ *Q s*) ⟹ *wf* {(*t*, *s*). *P s* ∧ *b s* ∧ *Some t* = *c s*} ⟹ *P s* ⟹
pred_option' Q (*while_break b c s*)
⟨*proof*⟩

context
fixes *args* :: (*bool iarray*, *nat set*, *'d* :: *timestamp*, *'t*, *'e*) *args*
begin

abbreviation *reach_w* ≡ *reach_window args*

qualified definition *in_win* = *init_window args*

definition *valid_window_matchP* :: *'d* ℐ ⇒ *'t* ⇒ *'e* ⇒

```
  ('d × bool iarray) list ⇒ nat ⇒ (bool iarray, nat set, 'd, 't, 'e) window ⇒ bool where
  valid_window_matchP I t0 sub rho j w ⟷ j = w_j w ∧
    valid_window args t0 sub rho w ∧
    reach_w t0 sub rho (w_i w, w_ti w, w_si w, w_j w, w_tj w, w_sj w) ∧
    (case w_read_t args (w_tj w) of None ⇒ True
    | Some t ⇒ (∀ l < w_i w. memL (ts_at rho l) t I))

lemma valid_window_matchP_reach_tj: valid_window_matchP I t0 sub rho i w ⟹
  reaches_on (w_run_t args) t0 (map fst rho) (w_tj w)
  ⟨proof⟩

lemma valid_window_matchP_reach_sj: valid_window_matchP I t0 sub rho i w ⟹
  reaches_on (w_run_sub args) sub (map snd rho) (w_sj w)
  ⟨proof⟩

lemma valid_window_matchP_len_rho: valid_window_matchP I t0 sub rho i w ⟹ length rho = i
  ⟨proof⟩

definition matchP_loop_cond I t = (λw. w_i w < w_j w ∧ memL (the (w_read_t args (w_ti w))) t I)

definition matchP_loop_inv I t0 sub rho j0 tj0 sj0 t =
  (λw. valid_window args t0 sub rho w ∧
    w_j w = j0 ∧ w_tj w = tj0 ∧ w_sj w = sj0 ∧ (∀ l < w_i w. memL (ts_at rho l) t I))

fun ex_key :: ('c, 'd) mmap ⇒ ('d ⇒ bool) ⇒
  ('c ⇒ bool) ⇒ ('c, bool) mapping ⇒ (bool × ('c, bool) mapping) where
  ex_key [] time accept ac = (False, ac)
| ex_key ((q, t) # qts) time accept ac = (if time t then
    (case cac accept ac q of (β, ac') ⇒
    if β then (True, ac') else ex_key qts time accept ac')
  else ex_key qts time accept ac)

lemma ex_key_sound:
  assumes inv: ⋀q. case Mapping.lookup ac q of None ⇒ True | Some v ⇒ accept q = v
  and distinct: distinct (map fst qts)
  and eval: ex_key qts time accept ac = (b, ac')
  shows b = (∃ q ∈ mmap_keys qts. time (the (mmap_lookup qts q)) ∧ accept q) ∧
    (∀ q. case Mapping.lookup ac' q of None ⇒ True | Some v ⇒ accept q = v)
  ⟨proof⟩

fun eval_matchP :: 'd I ⇒ (bool iarray, nat set, 'd, 't, 'e) window ⇒
  (('d × bool) × (bool iarray, nat set, 'd, 't, 'e) window) option where
  eval_matchP I w =
    (case w_read_t args (w_tj w) of None ⇒ None | Some t ⇒
      (case adv_end args w of None ⇒ None | Some w' ⇒
        let w'' = while (matchP_loop_cond I t) (adv_start args) w';
        (β, ac') = ex_key (w_e w'') (λt'. memR t' t I) (w_accept args) (w_ac w'') in
        Some ((t, β), w''(|w_ac := ac'|))))

definition valid_window_matchF :: 'd I ⇒ 't ⇒ 'e ⇒ ('d × bool iarray) list ⇒ nat ⇒
  (bool iarray, nat set, 'd, 't, 'e) window ⇒ bool where
  valid_window_matchF I t0 sub rho i w ⟷ i = w_i w ∧
    valid_window args t0 sub rho w ∧
    reach_w t0 sub rho (w_i w, w_ti w, w_si w, w_j w, w_tj w, w_sj w) ∧
    (∀ l ∈ {w_i w..<w_j w}. memR (ts_at rho i) (ts_at rho l) I)

lemma valid_window_matchF_reach_tj: valid_window_matchF I t0 sub rho i w ⟹
  reaches_on (w_run_t args) t0 (map fst rho) (w_tj w)
```

⟨*proof*⟩

lemma *valid_window_matchF_reach_sj*: *valid_window_matchF I t0 sub rho i w* ⟹
reaches_on (*w_run_sub args*) *sub* (*map snd rho*) (*w_sj w*)
⟨*proof*⟩

definition *matchF_loop_cond I t* =
(λ*w*. *case w_read_t args* (*w_tj w*) *of None* ⇒ *False* | *Some t'* ⇒ *memR t t' I*)

definition *matchF_loop_inv I t0 sub rho i ti si tjm sjm* =
(λ*w*. *valid_window args t0 sub* (*take* (*w_j w*) *rho*) *w* ∧
w_i w = *i* ∧ *w_ti w* = *ti* ∧ *w_si w* = *si* ∧
reach_window args t0 sub rho (*w_j w*, *w_tj w*, *w_sj w*, *length rho*, *tjm*, *sjm*) ∧
(∀ *l* ∈ {*w_i w*..<*w_j w*}. *memR* (*ts_at rho i*) (*ts_at rho l*) *I*))

definition *matchF_loop_inv' t0 sub rho i ti si j tj sj* =
(λ*w*. *w_i w* = *i* ∧ *w_ti w* = *ti* ∧ *w_si w* = *si* ∧
(∃ *rho'*. *valid_window args t0 sub* (*rho* @ *rho'*) *w* ∧
reach_window args t0 sub (*rho* @ *rho'*) (*j*, *tj*, *sj*, *w_j w*, *w_tj w*, *w_sj w*)))

fun *eval_matchF* :: *'d* ℐ ⇒ (*bool iarray*, *nat set*, *'d*, *'t*, *'e*) *window* ⇒
((*'d* × *bool*) × (*bool iarray*, *nat set*, *'d*, *'t*, *'e*) *window*) *option* **where**
eval_matchF I w =
(*case w_read_t args* (*w_ti w*) *of None* ⇒ *None* | *Some t* ⇒
(*case while_break* (*matchF_loop_cond I t*) (*adv_end args*) *w of None* ⇒ *None* | *Some w'* ⇒
(*case w_read_t args* (*w_tj w'*) *of None* ⇒ *None* | *Some t'* ⇒
let β = (*case snd* (*the* (*mmap_lookup* (*w_s w'*) {*0*})) *of None* ⇒ *False*
| *Some tstp* ⇒ *memL t* (*fst tstp*) *I*) *in*
Some ((*t*, β), *adv_start args w'*))))

end

locale *MDL_window* = *MDL* σ
for σ :: (*'a*, *'d* :: *timestamp*) *trace* +
fixes *r* :: (*'a*, *'d* :: *timestamp*) *regex*
and *t0* :: *'t*
and *sub* :: *'e*
and *args* :: (*bool iarray*, *nat set*, *'d*, *'t*, *'e*) *args*
assumes *init_def*: *w_init args* = {*0* :: *nat*}
and *step_def*: *w_step args* =
NFA.delta' (*IArray* (*build_nfa_impl r* (*0*, *state_cnt r*, []))) (*state_cnt r*)
and *accept_def*: *w_accept args* = *NFA.accept'* (*IArray* (*build_nfa_impl r* (*0*, *state_cnt r*, []))) (*state_cnt r*)
and *run_t_sound*: *reaches_on* (*w_run_t args*) *t0 ts t* ⟹
w_run_t args t = *Some* (*t'*, *x*) ⟹ *x* = τ σ (*length ts*)
and *run_sub_sound*: *reaches_on* (*w_run_sub args*) *sub bs s* ⟹
w_run_sub args s = *Some* (*s'*, *b*) ⟹
b = *IArray* (*map* (λ*phi*. *sat phi* (*length bs*)) (*collect_subfmlas r* []))
and *run_t_read*: *w_run_t args t* = *Some* (*t'*, *x*) ⟹ *w_read_t args t* = *Some x*
and *read_t_run*: *w_read_t args t* = *Some x* ⟹ ∃ *t'*. *w_run_t args t* = *Some* (*t'*, *x*)
begin

definition *qf* = *state_cnt r*
definition *transs* = *build_nfa_impl r* (*0*, *qf*, [])

abbreviation *init* ≡ *w_init args*
abbreviation *step* ≡ *w_step args*
abbreviation *accept* ≡ *w_accept args*

abbreviation *run* ≡ *NFA.run′* (*IArray transs*) *qf*
abbreviation *wacc* ≡ *Window.acc* (*w_step args*) (*w_accept args*)
abbreviation *rw* ≡ *reach_window args*

abbreviation *valid_matchP* ≡ *valid_window_matchP args*
abbreviation *eval_mP* ≡ *eval_matchP args*
abbreviation *matchP_inv* ≡ *matchP_loop_inv args*
abbreviation *matchP_cond* ≡ *matchP_loop_cond args*

abbreviation *valid_matchF* ≡ *valid_window_matchF args*
abbreviation *eval_mF* ≡ *eval_matchF args*
abbreviation *matchF_inv* ≡ *matchF_loop_inv args*
abbreviation *matchF_inv′* ≡ *matchF_loop_inv′ args*
abbreviation *matchF_cond* ≡ *matchF_loop_cond args*

lemma *run_t_sound′*:
  assumes *reaches_on* (*w_run_t args*) *t0 ts t i* < *length ts*
  shows *ts* ! *i* = *τ σ i*
⟨*proof*⟩

lemma *run_sub_sound′*:
  assumes *reaches_on* (*w_run_sub args*) *sub bs s i* < *length bs*
  shows *bs* ! *i* = *IArray* (*map* (*λphi. sat phi i*) (*collect_subfmlas r* []))
⟨*proof*⟩

lemma *run_ts*: *reaches_on* (*w_run_t args*) *t ts t′* ⟹ *t* = *t0* ⟹ *chain_le ts*
⟨*proof*⟩

lemma *ts_at_tau*: *reaches_on* (*w_run_t args*) *t0* (*map fst rho*) *t* ⟹ *i* < *length rho* ⟹
  ts_at rho i = *τ σ i*
  ⟨*proof*⟩

lemma *length_bs_at*: *reaches_on* (*w_run_sub args*) *sub* (*map snd rho*) *s* ⟹ *i* < *length rho* ⟹
  IArray.length (*bs_at rho i*) = *length* (*collect_subfmlas r* [])
  ⟨*proof*⟩

lemma *bs_at_nth*: *reaches_on* (*w_run_sub args*) *sub* (*map snd rho*) *s* ⟹ *i* < *length rho* ⟹
  n < *IArray.length* (*bs_at rho i*) ⟹
  IArray.sub (*bs_at rho i*) *n* ⟷ *sat* (*collect_subfmlas r* [] ! *n*) *i*
  ⟨*proof*⟩

lemma *ts_at_mono*: ⋀*i j*. *reaches_on* (*w_run_t args*) *t0* (*map fst rho*) *t* ⟹
  i ≤ *j* ⟹ *j* < *length rho* ⟹ *ts_at rho i* ≤ *ts_at rho j*
  ⟨*proof*⟩

lemma *steps_is_run*: *steps* (*w_step args*) *rho q ij* = *run q* (*sub_bs rho ij*)
  ⟨*proof*⟩

lemma *acc_is_accept*: *wacc rho q* (*i*, *j*) = *w_accept args* (*run q* (*sub_bs rho* (*i*, *j*)))
  ⟨*proof*⟩

lemma *iarray_list_of*: *IArray* (*IArray.list_of xs*) = *xs*
  ⟨*proof*⟩

lemma *map_iarray_list_of*: *map IArray* (*map IArray.list_of bss*) = *bss*
  ⟨*proof*⟩

lemma *acc_match*:

fixes *ts* :: *'d list*
assumes *reaches_on* (*w_run_sub args*) *sub* (*map snd rho*) *s i* ≤ *j j* ≤ *length rho wf_regex r*
shows *wacc rho* {*0*} (*i*, *j*) ⟷ (*i*, *j*) ∈ *match r*
⟨*proof*⟩

lemma *accept_match*:
fixes *ts* :: *'d list*
shows *reaches_on* (*w_run_sub args*) *sub* (*map snd rho*) *s* ⟹ *i* ≤ *j* ⟹ *j* ≤ *length rho* ⟹ *wf_regex r* ⟹
w_accept args (*steps* (*w_step args*) *rho* {*0*} (*i*, *j*)) ⟷ (*i*, *j*) ∈ *match r*
⟨*proof*⟩

lemma *drop_take_drop*: *i* ≤ *j* ⟹ *j* ≤ *length rho* ⟹ *drop i* (*take j rho*) @ *drop j rho* = *drop i rho*
⟨*proof*⟩

lemma *take_Suc*: *drop n xs* = *y* # *ys* ⟹ *take n xs* @ [*y*] = *take* (*Suc n*) *xs*
⟨*proof*⟩

lemma *valid_init_matchP*: *valid_matchP I t0 sub* [] *0* (*init_window args t0 sub*)
⟨*proof*⟩

lemma *valid_init_matchF*: *valid_matchF I t0 sub* [] *0* (*init_window args t0 sub*)
⟨*proof*⟩

lemma *valid_eval_matchP*:
assumes *valid_before'*: *valid_matchP I t0 sub rho j w*
and *before_end*: *w_run_t args* (*w_tj w*) = *Some* (*tj'''*, *t*) *w_run_sub args* (*w_sj w*) = *Some* (*sj'''*, *b*)
and *wf*: *wf_regex r*
shows ∃ *w'*. *eval_mP I w* = *Some* ((*τ σ j*, *sat* (*MatchP I r*) *j*), *w'*) ∧
t = *τ σ j* ∧ *valid_matchP I t0 sub* (*rho* @ [(*t*, *b*)]) (*Suc j*) *w'*
⟨*proof*⟩

lemma *valid_eval_matchF_Some*:
assumes *valid_before'*: *valid_matchF I t0 sub rho i w*
and *eval*: *eval_mF I w* = *Some* ((*t*, *b*), *w''*)
and *bounded*: *right I* ∈ *tfin*
shows ∃ *rho' tm*. *reaches_on* (*w_run_t args*) (*w_tj w*) (*map fst rho'*) (*w_tj w''*) ∧
reaches_on (*w_run_sub args*) (*w_sj w*) (*map snd rho'*) (*w_sj w''*) ∧
(*w_read_t args*) (*w_ti w*) = *Some t* ∧
(*w_read_t args*) (*w_tj w''*) = *Some tm* ∧
¬*memR t tm I*
⟨*proof*⟩

lemma *valid_eval_matchF_complete*:
assumes *valid_before'*: *valid_matchF I t0 sub rho i w*
and *before_end*: *reaches_on* (*w_run_t args*) (*w_tj w*) (*map fst rho'*) *tj'''*
reaches_on (*w_run_sub args*) (*w_sj w*) (*map snd rho'*) *sj'''*
w_read_t args (*w_ti w*) = *Some t w_read_t args tj'''* = *Some tm* ¬*memR t tm I*
and *wf*: *wf_regex r*
shows ∃ *w'*. *eval_mF I w* = *Some* ((*τ σ i*, *sat* (*MatchF I r*) *i*), *w'*) ∧
valid_matchF I t0 sub (*take* (*w_j w'*) (*rho* @ *rho'*)) (*Suc i*) *w'*
⟨*proof*⟩

lemma *valid_eval_matchF_sound*:
assumes *valid_before*: *valid_matchF I t0 sub rho i w*
and *eval*: *eval_mF I w* = *Some* ((*t*, *b*), *w''*)
and *bounded*: *right I* ∈ *tfin*

and *wf*: *wf_regex r*
shows $t = \tau\ \sigma\ i \wedge b = sat\ (MatchF\ I\ r)\ i \wedge (\exists\, rho'.\ valid\_matchF\ I\ t0\ sub\ rho'\ (Suc\ i)\ w'')$
⟨*proof*⟩

thm *valid_eval_matchP*
thm *valid_eval_matchF_sound*
thm *valid_eval_matchF_complete*

end

end
theory *Monitor*
imports *MDL Temporal*
begin

type_synonym $('h, 't)\ time = ('h \times 't)\ option$

datatype $(dead\ 'a, dead\ 't :: timestamp, dead\ 'h)\ vydra\_aux =$
VYDRA_None
| *VYDRA_Bool bool 'h*
| *VYDRA_Atom 'a 'h*
| *VYDRA_Neg ('a, 't, 'h) vydra_aux*
| *VYDRA_Bin bool ⇒ bool ⇒ bool ('a, 't, 'h) vydra_aux ('a, 't, 'h) vydra_aux*
| *VYDRA_Prev 't ℐ ('a, 't, 'h) vydra_aux 'h ('t × bool) option*
| *VYDRA_Next 't ℐ ('a, 't, 'h) vydra_aux 'h 't option*
| *VYDRA_Since 't ℐ ('a, 't, 'h) vydra_aux ('a, 't, 'h) vydra_aux ('h, 't) time nat nat nat option 't option*
| *VYDRA_Until 't ℐ ('h, 't) time ('a, 't, 'h) vydra_aux ('a, 't, 'h) vydra_aux ('h, 't) time nat ('t × bool × bool) option*
| *VYDRA_MatchP 't ℐ transition iarray nat*
(bool iarray, nat set, 't, ('h, 't) time, ('a, 't, 'h) vydra_aux list) window
| *VYDRA_MatchF 't ℐ transition iarray nat*
(bool iarray, nat set, 't, ('h, 't) time, ('a, 't, 'h) vydra_aux list) window

type_synonym $('a, 't, 'h)\ vydra = nat \times ('a, 't, 'h)\ vydra\_aux$

fun *msize_vydra* :: *nat* ⇒ $('a, 't :: timestamp, 'h)\ vydra\_aux \Rightarrow nat$ **where**
msize_vydra n VYDRA_None = 0
| *msize_vydra n (VYDRA_Bool b e) = 0*
| *msize_vydra n (VYDRA_Atom a e) = 0*
| *msize_vydra (Suc n) (VYDRA_Bin f v1 v2) = msize_vydra n v1 + msize_vydra n v2 + 1*
| *msize_vydra (Suc n) (VYDRA_Neg v) = msize_vydra n v + 1*
| *msize_vydra (Suc n) (VYDRA_Prev I v e tb) = msize_vydra n v + 1*
| *msize_vydra (Suc n) (VYDRA_Next I v e to) = msize_vydra n v + 1*
| *msize_vydra (Suc n) (VYDRA_Since I vphi vpsi e cphi cpsi cppsi tppsi) = msize_vydra n vphi + msize_vydra n vpsi + 1*
| *msize_vydra (Suc n) (VYDRA_Until I e vphi vpsi epsi c zo) = msize_vydra n vphi + msize_vydra n vpsi + 1*
| *msize_vydra (Suc n) (VYDRA_MatchP I transs qf w) = size_list (msize_vydra n) (w_si w) + size_list (msize_vydra n) (w_sj w) + 1*
| *msize_vydra (Suc n) (VYDRA_MatchF I transs qf w) = size_list (msize_vydra n) (w_si w) + size_list (msize_vydra n) (w_sj w) + 1*
| *msize_vydra _ _ = 0*

fun *next_vydra* :: $('a, 't :: timestamp, 'h)\ vydra\_aux \Rightarrow nat$ **where**
next_vydra (VYDRA_Next I v e None) = 1
| *next_vydra _ = 0*

context
  fixes *init_hd* :: *'h*
    and *run_hd* :: *'h* ⇒ (*'h* × (*'t* :: *timestamp* × *'a set*)) *option*
begin

definition *t0* :: (*'h*, *'t*) *time* **where**
  t0 = (*case run_hd init_hd of None* ⇒ *None* | *Some* (*e'*, (*t*, *X*)) ⇒ *Some* (*e'*, *t*))

fun *run_t* :: (*'h*, *'t*) *time* ⇒ ((*'h*, *'t*) *time* × *'t*) *option* **where**
  run_t None = *None*
| *run_t* (*Some* (*e*, *t*)) = (*case run_hd e of None* ⇒ *Some* (*None*, *t*)
  | *Some* (*e'*, (*t'*, *X*)) ⇒ *Some* (*Some* (*e'*, *t'*), *t*))

fun *read_t* :: (*'h*, *'t*) *time* ⇒ *'t option* **where**
  read_t None = *None*
| *read_t* (*Some* (*e*, *t*)) = *Some t*

lemma *run_t_read*: *run_t x* = *Some* (*x'*, *t*) ⟹ *read_t x* = *Some t*
  ⟨*proof*⟩

lemma *read_t_run*: *read_t x* = *Some t* ⟹ ∃ *x'*. *run_t x* = *Some* (*x'*, *t*)
  ⟨*proof*⟩

lemma *reach_event_t*: *reaches_on run_hd e vs e''* ⟹ *run_hd e* = *Some* (*e'*, (*t*, *X*)) ⟹
  run_hd e'' = *Some* (*e'''*, (*t'*, *X'*)) ⟹
  reaches_on run_t (*Some* (*e'*, *t*)) (*map fst vs*) (*Some* (*e'''*, *t'*))
⟨*proof*⟩

lemma *reach_event_t0_t*:
  assumes *reaches_on run_hd init_hd vs e'' run_hd e''* = *Some* (*e'''*, (*t'*, *X'*))
  shows *reaches_on run_t t0* (*map fst vs*) (*Some* (*e'''*, *t'*))
⟨*proof*⟩

lemma *reaches_on_run_hd_t*:
  assumes *reaches_on run_hd init_hd vs e*
  shows ∃ *x*. *reaches_on run_t t0* (*map fst vs*) *x*
⟨*proof*⟩

definition *run_subs run* = (λ*vs*. *let vs'* = *map run vs in*
  (*if* (∃ *x* ∈ *set vs'*. *Option.is_none x*) *then None*
   else Some (*map* (*fst* ∘ *the*) *vs'*, *iarray_of_list* (*map* (*snd* ∘ *snd* ∘ *the*) *vs'*))))

lemma *run_subs_lD*: *run_subs run vs* = *Some* (*vs'*, *bs*) ⟹
  length vs' = *length vs* ∧ *IArray.length bs* = *length vs*
  ⟨*proof*⟩

lemma *run_subs_vD*: *run_subs run vs* = *Some* (*vs'*, *bs*) ⟹ *j* < *length vs* ⟹
  ∃ *vj' tj bj*. *run* (*vs* ! *j*) = *Some* (*vj'*, (*tj*, *bj*)) ∧ *vs'* ! *j* = *vj'* ∧ *IArray.sub bs j* = *bj*
  ⟨*proof*⟩

fun *msize_fmla* :: (*'a*, *'b* :: *timestamp*) *formula* ⇒ *nat*
  and *msize_regex* :: (*'a*, *'b*) *regex* ⇒ *nat* **where**
  msize_fmla (*Bool b*) = *0*
| *msize_fmla* (*Atom a*) = *0*
| *msize_fmla* (*Neg phi*) = *Suc* (*msize_fmla phi*)
| *msize_fmla* (*Bin f phi psi*) = *Suc* (*msize_fmla phi* + *msize_fmla psi*)
| *msize_fmla* (*Prev I phi*) = *Suc* (*msize_fmla phi*)
| *msize_fmla* (*Next I phi*) = *Suc* (*msize_fmla phi*)

| *msize_fmla* (*Since phi I psi*) = *Suc* (*max* (*msize_fmla phi*) (*msize_fmla psi*))
| *msize_fmla* (*Until phi I psi*) = *Suc* (*max* (*msize_fmla phi*) (*msize_fmla psi*))
| *msize_fmla* (*MatchP I r*) = *Suc* (*msize_regex r*)
| *msize_fmla* (*MatchF I r*) = *Suc* (*msize_regex r*)
| *msize_regex* (*Lookahead phi*) = *msize_fmla phi*
| *msize_regex* (*Symbol phi*) = *msize_fmla phi*
| *msize_regex* (*Plus r s*) = *max* (*msize_regex r*) (*msize_regex s*)
| *msize_regex* (*Times r s*) = *max* (*msize_regex r*) (*msize_regex s*)
| *msize_regex* (*Star r*) = *msize_regex r*

lemma *collect_subfmlas_msize*: *x* ∈ *set* (*collect_subfmlas r* []) ⟹
msize_fmla x ≤ *msize_regex r*
⟨*proof*⟩

definition *until_ready I t c zo* = (*case* (*c*, *zo*) *of* (*Suc* _, *Some* (*t′*, *b1*, *b2*)) ⇒ (*b2* ∧ *memL t t′ I*) ∨ ¬*b1* | _ ⇒ *False*)

definition *while_since_cond I t* = (λ(*vpsi*, *e*, *cpsi* :: *nat*, *cppsi*, *tppsi*). *cpsi* > *0* ∧ *memL* (*the* (*read_t e*)) *t I*)
definition *while_since_body run* =
(λ(*vpsi*, *e*, *cpsi* :: *nat*, *cppsi*, *tppsi*).
case run vpsi of Some (*vpsi′*, (*t′*, *b′*)) ⇒
Some (*vpsi′*, *fst* (*the* (*run_t e*)), *cpsi* − *1*, *if b′ then Some cpsi else cppsi*, *if b′ then Some t′ else tppsi*)
| _ ⇒ *None*
)

definition *while_until_cond I t* = (λ(*vphi*, *vpsi*, *epsi*, *c*, *zo*). ¬*until_ready I t c zo* ∧ (*case read_t epsi of Some t′* ⇒ *memR t t′ I* | *None* ⇒ *False*))
definition *while_until_body run* =
(λ(*vphi*, *vpsi*, *epsi*, *c*, *zo*). *case run_t epsi of Some* (*epsi′*, *t′*) ⇒
(*case run vphi of Some* (*vphi′*, (_, *b1*)) ⇒
(*case run vpsi of Some* (*vpsi′*, (_, *b2*)) ⇒ *Some* (*vphi′*, *vpsi′*, *epsi′*, *Suc c*, *Some* (*t′*, *b1*, *b2*))
| _ ⇒ *None*)
| _ ⇒ *None*))

function (*sequential*) *run* :: *nat* ⇒ ('*a*, '*t*, '*h*) *vydra_aux* ⇒ (('*a*, '*t*, '*h*) *vydra_aux* × ('*t* × *bool*)) *option*
where
run n (*VYDRA_None*) = *None*
| *run n* (*VYDRA_Bool b e*) = (*case run_hd e of None* ⇒ *None*
| *Some* (*e′*, (*t*, _)) ⇒ *Some* (*VYDRA_Bool b e′*, (*t*, *b*)))
| *run n* (*VYDRA_Atom a e*) = (*case run_hd e of None* ⇒ *None*
| *Some* (*e′*, (*t*, *X*)) ⇒ *Some* (*VYDRA_Atom a e′*, (*t*, *a* ∈ *X*)))
| *run* (*Suc n*) (*VYDRA_Neg v*) = (*case run n v of None* ⇒ *None*
| *Some* (*v′*, (*t*, *b*)) ⇒ *Some* (*VYDRA_Neg v′*, (*t*, ¬*b*)))
| *run* (*Suc n*) (*VYDRA_Bin f vl vr*) = (*case run n vl of None* ⇒ *None*
| *Some* (*vl′*, (*t*, *bl*)) ⇒ (*case run n vr of None* ⇒ *None*
| *Some* (*vr′*, (_, *br*)) ⇒ *Some* (*VYDRA_Bin f vl′ vr′*, (*t*, *f bl br*))))
| *run* (*Suc n*) (*VYDRA_Prev I v e tb*) = (*case run_hd e of Some* (*e′*, (*t*, _)) ⇒
(*let* β = (*case tb of Some* (*t′*, *b′*) ⇒ *b′* ∧ *mem t′ t I* | *None* ⇒ *False*) *in*
case run n v of Some (*v′*, _, *b′*) ⇒ *Some* (*VYDRA_Prev I v′ e′* (*Some* (*t*, *b′*)), (*t*, β))
| *None* ⇒ *Some* (*VYDRA_None*, (*t*, β)))
| *None* ⇒ *None*)
| *run* (*Suc n*) (*VYDRA_Next I v e to*) = (*case run_hd e of Some* (*e′*, (*t*, _)) ⇒
(*case to of None* ⇒
(*case run n v of Some* (*v′*, _, _) ⇒ *run* (*Suc n*) (*VYDRA_Next I v′ e′* (*Some t*))
| *None* ⇒ *None*)
| *Some t′* ⇒

```
      (case run n v of Some (v', _, b) ⇒ Some (VYDRA_Next I v' e' (Some t), (t', b ∧ mem t' t I))
      | None ⇒ if mem t' t I then None else Some (VYDRA_None, (t', False))))
    | None ⇒ None)
| run (Suc n) (VYDRA_Since I vphi vpsi e cphi cpsi cppsi tppsi) = (case run n vphi of
    Some (vphi', (t, b1)) ⇒
      let cphi = (if b1 then Suc cphi else 0) in
      let cpsi = Suc cpsi in
      let cppsi = map_option Suc cppsi in
      (case while_break (while_since_cond I t) (while_since_body (run n)) (vpsi, e, cpsi, cppsi, tppsi) of
Some (vpsi', e', cpsi', cppsi', tppsi') ⇒
        (let β = (case cppsi' of Some k ⇒ k − 1 ≤ cphi ∧ memR (the tppsi') t I | _ ⇒ False) in
         Some (VYDRA_Since I vphi' vpsi' e' cphi cpsi' cppsi' tppsi', (t, β)))
      | _ ⇒ None)
    | _ ⇒ None)
| run (Suc n) (VYDRA_Until I e vphi vpsi epsi c zo) = (case run_t e of Some (e', t) ⇒
     (case while_break (while_until_cond I t) (while_until_body (run n)) (vphi, vpsi, epsi, c, zo) of Some
(vphi', vpsi', epsi', c', zo') ⇒
        if c' = 0 then None else
        (case zo' of Some (t', b1, b2) ⇒
          (if b2 ∧ memL t t' I then Some (VYDRA_Until I e' vphi' vpsi' epsi' (c' − 1) zo', (t, True))
          else if ¬b1 then Some (VYDRA_Until I e' vphi' vpsi' epsi' (c' − 1) zo', (t, False))
          else (case read_t epsi' of Some t' ⇒ Some (VYDRA_Until I e' vphi' vpsi' epsi' (c' − 1) zo', (t,
False)) | _ ⇒ None))
        | _ ⇒ None)
      | _ ⇒ None)
    | _ ⇒ None)
| run (Suc n) (VYDRA_MatchP I transs qf w) =
    (case eval_matchP (init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
      (run_t, read_t) (run_subs (run n))) I w of None ⇒ None
    | Some ((t, b), w') ⇒ Some (VYDRA_MatchP I transs qf w', (t, b)))
| run (Suc n) (VYDRA_MatchF I transs qf w) =
    (case eval_matchF (init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
      (run_t, read_t) (run_subs (run n))) I w of None ⇒ None
    | Some ((t, b), w') ⇒ Some (VYDRA_MatchF I transs qf w', (t, b)))
| run _ _ = undefined
  ⟨proof⟩
termination
  ⟨proof⟩

lemma wf_since: wf {(t, s). while_since_cond I tt s ∧ Some t = while_since_body (run n) s}
⟨proof⟩

definition run_vydra :: ('a, 't, 'h) vydra ⇒ (('a, 't, 'h) vydra × ('t × bool)) option where
  run_vydra v = (case v of (n, w) ⇒ map_option (apfst (Pair n)) (run n w))

fun sub :: nat ⇒ ('a, 't) formula ⇒ ('a, 't, 'h) vydra_aux where
  sub n (Bool b) = VYDRA_Bool b init_hd
| sub n (Atom a) = VYDRA_Atom a init_hd
| sub (Suc n) (Neg phi) = VYDRA_Neg (sub n phi)
| sub (Suc n) (Bin f phi psi) = VYDRA_Bin f (sub n phi) (sub n psi)
| sub (Suc n) (Prev I phi) = VYDRA_Prev I (sub n phi) init_hd None
| sub (Suc n) (Next I phi) = VYDRA_Next I (sub n phi) init_hd None
| sub (Suc n) (Since phi I psi) = VYDRA_Since I (sub n phi) (sub n psi) t0 0 0 None None
| sub (Suc n) (Until phi I psi) = VYDRA_Until I t0 (sub n phi) (sub n psi) t0 0 None
| sub (Suc n) (MatchP I r) = (let qf = state_cnt r;
    transs = iarray_of_list (build_nfa_impl r (0, qf, [])) in
    VYDRA_MatchP I transs qf (init_window (init_args
      ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
```

```
      (run_t, read_t) (run_subs (run n)))
      t0 (map (sub n) (collect_subfmlas r [])))))
| sub (Suc n) (MatchF I r) = (let qf = state_cnt r;
    transs = iarray_of_list (build_nfa_impl r (0, qf, [])) in
    VYDRA_MatchF I transs qf (init_window (init_args
      ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
      (run_t, read_t) (run_subs (run n)))
      t0 (map (sub n) (collect_subfmlas r [])))))
| sub _ _ = undefined

definition init_vydra :: ('a, 't) formula ⇒ ('a, 't, 'h) vydra where
  init_vydra φ = (let n = msize_fmla φ in (n, sub n φ))

end

locale VYDRA_MDL = MDL σ
  for σ :: ('a, 't :: timestamp) trace +
  fixes init_hd :: 'h
    and run_hd :: 'h ⇒ ('h × ('t × 'a set)) option
  assumes run_hd_sound: reaches run_hd init_hd n s ⟹ run_hd s = Some (s', (t, X)) ⟹ (t, X) =
(τ σ n, Γ σ n)
begin

lemma reaches_on_run_hd: reaches_on run_hd init_hd es s ⟹ run_hd s = Some (s', (t, X)) ⟹ t
= τ σ (length es) ∧ X = Γ σ (length es)
  ⟨proof⟩

abbreviation ru_t ≡ run_t run_hd
abbreviation l_t0 ≡ t0 init_hd run_hd
abbreviation ru ≡ run run_hd
abbreviation su ≡ sub init_hd run_hd

lemma ru_t_event: reaches_on ru_t t ts t' ⟹ t = l_t0 ⟹ ru_t t' = Some (t'', x) ⟹
  ∃ rho e tt. t' = Some (e, tt) ∧ reaches_on run_hd init_hd rho e ∧ length rho = Suc (length ts) ∧
  x = τ σ (length ts)
⟨proof⟩

lemma ru_t_tau: reaches_on ru_t l_t0 ts t' ⟹ ru_t t' = Some (t'', x) ⟹ x = τ σ (length ts)
  ⟨proof⟩

lemma ru_t_Some_tau:
  assumes reaches_on ru_t l_t0 ts (Some (e, t))
  shows t = τ σ (length ts)
⟨proof⟩

lemma ru_t_tau_in:
  assumes reaches_on ru_t l_t0 ts t j < length ts
  shows ts ! j = τ σ j
⟨proof⟩

lemmas run_hd_tau_in = ru_t_tau_in[OF reach_event_t0_t, simplified]

fun last_before :: (nat ⇒ bool) ⇒ nat ⇒ nat option where
  last_before P 0 = None
| last_before P (Suc n) = (if P n then Some n else last_before P n)

lemma last_before_None: last_before P n = None ⟹ m < n ⟹ ¬P m
⟨proof⟩
```

lemma *last_before_Some*: *last_before P n = Some m ⟹ m < n ∧ P m ∧ (∀ k ∈ {m<..<n}. ¬P k)*
⟨*proof*⟩

inductive *wf_vydra* :: (*'a*, *'t* :: *timestamp*) *formula* ⇒ *nat* ⇒ *nat* ⇒ (*'a*, *'t*, *'h*) *vydra_aux* ⇒ *bool* **where**
wf_vydra phi i n w ⟹ ru n w = None ⟹ wf_vydra (Prev I phi) (Suc i) (Suc n) VYDRA_None
| *wf_vydra phi i n w ⟹ ru n w = None ⟹ wf_vydra (Next I phi) i (Suc n) VYDRA_None*
| *reaches_on run_hd init_hd es sub' ⟹ length es = i ⟹ wf_vydra (Bool b) i n (VYDRA_Bool b sub')*
| *reaches_on run_hd init_hd es sub' ⟹ length es = i ⟹ wf_vydra (Atom a) i n (VYDRA_Atom a sub')*
| *wf_vydra phi i n v ⟹ wf_vydra (Neg phi) i (Suc n) (VYDRA_Neg v)*
| *wf_vydra phi i n v ⟹ wf_vydra psi i n v' ⟹ wf_vydra (Bin f phi psi) i (Suc n) (VYDRA_Bin f v v')*
| *wf_vydra phi i n v ⟹ reaches_on run_hd init_hd es sub' ⟹ length es = i ⟹*
wf_vydra (Prev I phi) i (Suc n) (VYDRA_Prev I v sub' (case i of 0 ⇒ None | Suc j ⇒ Some (τ σ j, sat phi j)))
| *wf_vydra phi i n v ⟹ reaches_on run_hd init_hd es sub' ⟹ length es = i ⟹*
wf_vydra (Next I phi) (i − 1) (Suc n) (VYDRA_Next I v sub' (case i of 0 ⇒ None | Suc j ⇒ Some (τ σ j)))
| *wf_vydra phi i n vphi ⟹ wf_vydra psi j n vpsi ⟹ j ≤ i ⟹*
reaches_on ru_t l_t0 es sub' ⟹ length es = j ⟹ (⋀t. t ∈ set es ⟹ memL t (τ σ i) I) ⟹
cphi = i − (case last_before (λk. ¬sat phi k) i of None ⇒ 0 | Some k ⇒ Suc k) ⟹ cpsi = i − j ⟹
cppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (i − k)) ⟹
tppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (τ σ k)) ⟹
wf_vydra (Since phi I psi) i (Suc n) (VYDRA_Since I vphi vpsi sub' cphi cpsi cppsi tppsi)
| *wf_vydra phi j n vphi ⟹ wf_vydra psi j n vpsi ⟹ i ≤ j ⟹*
reaches_on ru_t l_t0 es back ⟹ length es = i ⟹
reaches_on ru_t l_t0 es' front ⟹ length es' = j ⟹ (⋀t. t ∈ set es' ⟹ memR (τ σ i) t I) ⟹
c = j − i ⟹ z = (case j of 0 ⇒ None | Suc k ⇒ Some (τ σ k, sat phi k, sat psi k)) ⟹
(⋀k. k ∈ {i..<j − 1} ⟹ sat phi k ∧ (memL (τ σ i) (τ σ k) I ⟶ ¬sat psi k)) ⟹
wf_vydra (Until phi I psi) i (Suc n) (VYDRA_Until I back vphi vpsi front c z)
| *valid_window_matchP args I l_t0 (map (su n) (collect_subfmlas r [])) xs i w ⟹*
n ≥ msize_regex r ⟹ qf = state_cnt r ⟹
transs = iarray_of_list (build_nfa_impl r (0, qf, [])) ⟹
args = init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
(ru_t, read_t) (run_subs (ru n)) ⟹
wf_vydra (MatchP I r) i (Suc n) (VYDRA_MatchP I transs qf w)
| *valid_window_matchF args I l_t0 (map (su n) (collect_subfmlas r [])) xs i w ⟹*
n ≥ msize_regex r ⟹ qf = state_cnt r ⟹
transs = iarray_of_list (build_nfa_impl r (0, qf, [])) ⟹
args = init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
(ru_t, read_t) (run_subs (ru n)) ⟹
wf_vydra (MatchF I r) i (Suc n) (VYDRA_MatchF I transs qf w)

lemma *reach_run_subs_len*:
assumes *reaches_ons*: *reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) rho vs*
shows *length vs = length (collect_subfmlas r [])*
⟨*proof*⟩

lemma *reach_run_subs_run*:
assumes *reaches_ons*: *reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) rho vs*
and *subfmla*: *j < length (collect_subfmlas r []) phi = collect_subfmlas r [] ! j*
shows *∃ rho'. reaches_on (ru n) (su n phi) rho' (vs ! j) ∧ length rho' = length rho*
⟨*proof*⟩

lemma *IArray_nth_equalityI*: *IArray.length xs = length ys ⟹*
(⋀i. i < IArray.length xs ⟹ IArray.sub xs i = ys ! i) ⟹ xs = IArray ys
⟨*proof*⟩

lemma *bs_sat*:
assumes *IH*: $\bigwedge phi\ i\ v\ v'\ b.\ phi \in set\ (collect\_subfmlas\ r\ []) \Longrightarrow wf\_vydra\ phi\ i\ n\ v \Longrightarrow ru\ n\ v = Some\ (v',\ b) \Longrightarrow snd\ b = sat\ phi\ i$
and *reaches_ons*: $\bigwedge j.\ j < length\ (collect\_subfmlas\ r\ []) \Longrightarrow wf\_vydra\ (collect\_subfmlas\ r\ []\ !\ j)\ i\ n\ (vs\ !\ j)$
and *run_subs*: *run_subs* $(ru\ n)\ vs = Some\ (vs',\ bs)$ *length* $vs = length\ (collect\_subfmlas\ r\ [])$
shows $bs = iarray\_of\_list\ (map\ (\lambda phi.\ sat\ phi\ i)\ (collect\_subfmlas\ r\ []))$
⟨*proof*⟩

lemma *run_induct*[*case_names Bool Atom Neg Bin Prev Next Since Until MatchP MatchF*, *consumes 1*]:
fixes *phi* :: $('a,\ 't)\ formula$
assumes *msize_fmla phi* $\le n$ $(\bigwedge b\ n.\ P\ n\ (Bool\ b))$ $(\bigwedge a\ n.\ P\ n\ (Atom\ a))$
$(\bigwedge n\ phi.\ msize\_fmla\ phi \le n \Longrightarrow P\ n\ phi \Longrightarrow P\ (Suc\ n)\ (Neg\ phi))$
$(\bigwedge n\ f\ phi\ psi.\ msize\_fmla\ (Bin\ f\ phi\ psi) \le Suc\ n \Longrightarrow P\ n\ phi \Longrightarrow P\ n\ psi \Longrightarrow P\ (Suc\ n)\ (Bin\ f\ phi\ psi))$
$(\bigwedge n\ I\ phi.\ msize\_fmla\ phi \le n \Longrightarrow P\ n\ phi \Longrightarrow P\ (Suc\ n)\ (Prev\ I\ phi))$
$(\bigwedge n\ I\ phi.\ msize\_fmla\ phi \le n \Longrightarrow P\ n\ phi \Longrightarrow P\ (Suc\ n)\ (Next\ I\ phi))$
$(\bigwedge n\ I\ phi\ psi.\ msize\_fmla\ phi \le n \Longrightarrow msize\_fmla\ psi \le n \Longrightarrow P\ n\ phi \Longrightarrow P\ n\ psi \Longrightarrow P\ (Suc\ n)\ (Since\ phi\ I\ psi))$
$(\bigwedge n\ I\ phi\ psi.\ msize\_fmla\ phi \le n \Longrightarrow msize\_fmla\ psi \le n \Longrightarrow P\ n\ phi \Longrightarrow P\ n\ psi \Longrightarrow P\ (Suc\ n)\ (Until\ phi\ I\ psi))$
$(\bigwedge n\ I\ r.\ msize\_fmla\ (MatchP\ I\ r) \le Suc\ n \Longrightarrow (\bigwedge x.\ msize\_fmla\ x \le n \Longrightarrow P\ n\ x) \Longrightarrow P\ (Suc\ n)\ (MatchP\ I\ r))$
$(\bigwedge n\ I\ r.\ msize\_fmla\ (MatchF\ I\ r) \le Suc\ n \Longrightarrow (\bigwedge x.\ msize\_fmla\ x \le n \Longrightarrow P\ n\ x) \Longrightarrow P\ (Suc\ n)\ (MatchF\ I\ r))$
shows *P n phi*
⟨*proof*⟩

lemma *wf_vydra_sub*: $msize\_fmla\ \varphi \le n \Longrightarrow wf\_vydra\ \varphi\ 0\ n\ (su\ n\ \varphi)$
⟨*proof*⟩

lemma *ru_t_Some*: $\exists e'\ et.\ ru\_t\ e = Some\ (e',\ et)$ **if** *reaches_Suc_i*: *reaches_on run_hd init_hd fs f length fs* $= Suc\ i$
and *aux*: *reaches_on ru_t l_t0 es e length es* $\le i$ **for** *es e*
⟨*proof*⟩

lemma *vydra_sound_aux*:
assumes $msize\_fmla\ \varphi \le n\ wf\_vydra\ \varphi\ i\ n\ v\ ru\ n\ v = Some\ (v',\ t,\ b)\ bounded\_future\_fmla\ \varphi\ wf\_fmla\ \varphi$
shows $wf\_vydra\ \varphi\ (Suc\ i)\ n\ v' \wedge (\exists es\ e.\ reaches\_on\ run\_hd\ init\_hd\ es\ e \wedge length\ es = Suc\ i) \wedge t = \tau\ \sigma\ i \wedge b = sat\ \varphi\ i$
⟨*proof*⟩

lemma *reaches_ons_run_lD*: *reaches_on* $(run\_subs\ (ru\ n))\ vs\ ws\ vs' \Longrightarrow$
$length\ vs = length\ vs'$
⟨*proof*⟩

lemma *reaches_ons_run_vD*: *reaches_on* $(run\_subs\ (ru\ n))\ vs\ ws\ vs' \Longrightarrow$
$i < length\ vs \Longrightarrow (\exists ys.\ reaches\_on\ (ru\ n)\ (vs\ !\ i)\ ys\ (vs'\ !\ i) \wedge length\ ys = length\ ws)$
⟨*proof*⟩

lemma *reaches_ons_runI*:
assumes $\bigwedge phi.\ phi \in set\ (collect\_subfmlas\ r\ []) \Longrightarrow \exists ws\ v.\ reaches\_on\ (ru\ n)\ (su\ n\ phi)\ ws\ v \wedge length\ ws = i$
shows $\exists ws\ v.\ reaches\_on\ (run\_subs\ (ru\ n))\ (map\ (su\ n)\ (collect\_subfmlas\ r\ []))\ ws\ v \wedge length\ ws = i$
⟨*proof*⟩

lemma *reaches_on_takeWhile*: *reaches_on r s vs s'* $\Longrightarrow$ *r s' = Some (s'', v)* $\Longrightarrow$ $\neg$*f v* $\Longrightarrow$
vs' = takeWhile f vs $\Longrightarrow$
$\exists$ *t' t'' v'. reaches_on r s vs' t'* $\wedge$ *r t' = Some (t'', v')* $\wedge$ $\neg$*f v'* $\wedge$
reaches_on r t' (drop (length vs') vs) s'
⟨*proof*⟩

lemma *reaches_on_suffix*:
assumes *reaches_on r s vs s' reaches_on r s vs' s'' length vs'* $\leq$ *length vs*
shows $\exists$ *vs''. reaches_on r s'' vs'' s'* $\wedge$ *vs = vs' @ vs''*
⟨*proof*⟩

lemma *vydra_wf_reaches_on*:
assumes $\bigwedge$*j v. j < i* $\Longrightarrow$ *wf_vydra* φ *j n v* $\Longrightarrow$ *ru n v = None* $\Longrightarrow$ *False bounded_future_fmla* φ *wf_fmla* φ *msize_fmla* φ $\leq$ *n*
shows $\exists$ *vs v. reaches_on (ru n) (su n* φ*) vs v* $\wedge$ *wf_vydra* φ *i n v* $\wedge$ *length vs = i*
⟨*proof*⟩

lemma *reaches_on_Some*:
assumes *reaches_on r s vs s' reaches_on r s vs' s'' length vs < length vs'*
shows $\exists$ *s''' x. r s' = Some (s''', x)*
⟨*proof*⟩

lemma *reaches_on_progress*:
assumes *reaches_on run_hd init_hd vs e*
shows *progress phi (map fst vs)* $\leq$ *length vs*
⟨*proof*⟩

lemma *vydra_complete_aux*:
assumes *prefix*: *reaches_on run_hd init_hd vs e*
and *run*: *wf_vydra* φ *i n v ru n v = None i < progress* φ *(map fst vs) bounded_future_fmla* φ *wf_fmla* φ
and *msize*: *msize_fmla* φ $\leq$ *n*
shows *False*
⟨*proof*⟩

definition *ru'* φ = *ru (msize_fmla* φ*)*
definition *su'* φ = *su (msize_fmla* φ*)* φ

lemma *vydra_wf*:
assumes *reaches (ru n) (su n* φ*) i v bounded_future_fmla* φ *wf_fmla* φ *msize_fmla* φ $\leq$ *n*
shows *wf_vydra* φ *i n v*
⟨*proof*⟩

lemma *vydra_sound'*:
assumes *reaches (ru'* φ*) (su'* φ*) n v ru'* φ *v = Some (v', (t, b)) bounded_future_fmla* φ *wf_fmla* φ
shows *(t, b) =* (τ σ *n, sat* φ *n*)
⟨*proof*⟩

lemma *vydra_complete'*:
assumes *prefix*: *reaches_on run_hd init_hd vs e*
and *prog*: *n < progress* φ *(map fst vs) bounded_future_fmla* φ *wf_fmla* φ
shows $\exists$ *v v'. reaches (ru'* φ*) (su'* φ*) n v* $\wedge$ *ru'* φ *v = Some (v', (*τ σ *n, sat* φ *n))*
⟨*proof*⟩

lemma *map_option_apfst_idle*: *map_option (apfst snd) (map_option (apfst (Pair n)) x) = x*
⟨*proof*⟩

lemma *vydra_sound*:
assumes *reaches* (*run_vydra run_hd*) (*init_vydra init_hd run_hd* φ) *n v run_vydra run_hd v* = *Some* (v', (*t*, *b*)) *bounded_future_fmla* φ *wf_fmla* φ
shows (*t*, *b*) = (τ σ *n*, *sat* φ *n*)
⟨*proof*⟩

lemma *vydra_complete*:
assumes *prefix*: *reaches_on run_hd init_hd vs e*
and *prog*: *n* < *progress* φ (*map fst vs*) *bounded_future_fmla* φ *wf_fmla* φ
shows ∃ *v* v'. *reaches* (*run_vydra run_hd*) (*init_vydra init_hd run_hd* φ) *n v* ∧ *run_vydra run_hd v* = *Some* (v', (τ σ *n*, *sat* φ *n*))
⟨*proof*⟩

end

context *MDL*
begin

lemma *reach_elem*:
assumes *reaches* (λ*i*. *if P i then Some* (*Suc i*, (τ σ *i*, Γ σ *i*)) *else None*) *s n s'* *s* = *0*
shows s' = *n*
⟨*proof*⟩

interpretation *default_vydra*: *VYDRA_MDL* σ *0* λ*i*. *Some* (*Suc i*, (τ σ *i*, Γ σ *i*))
⟨*proof*⟩

end

lemma *reaches_inj*: *reaches r s i t* ⟹ *reaches r s i* t' ⟹ *t* = t'
⟨*proof*⟩

lemma *progress_sound*:
assumes
⋀*n*. *n* < *length ts* ⟹ *ts* ! *n* = τ σ *n*
⋀*n*. *n* < *length ts* ⟹ τ σ *n* = τ σ' *n*
⋀*n*. *n* < *length ts* ⟹ Γ σ *n* = Γ σ' *n*
n < *progress phi ts*
bounded_future_fmla phi
wf_fmla phi
shows *MDL.sat* σ *phi n* ⟷ *MDL.sat* σ' *phi n*
⟨*proof*⟩

end
theory *Preliminaries*
imports *MDL*
begin

4 Formulas and Satisfiability

declare [[*typedef_overloaded*]]

context
begin

qualified datatype ('*a*, '*t* :: *timestamp*) *formula* = *Bool bool* | *Atom* '*a* | *Neg* ('*a*, '*t*) *formula* |
Bin bool ⇒ *bool* ⇒ *bool* ('*a*, '*t*) *formula* ('*a*, '*t*) *formula* |
Prev '*t* $\mathcal{I}$ ('*a*, '*t*) *formula* | *Next* '*t* $\mathcal{I}$ ('*a*, '*t*) *formula* |
Since ('*a*, '*t*) *formula* '*t* $\mathcal{I}$ ('*a*, '*t*) *formula* |

```
  Until ('a, 't) formula 't \<I> ('a, 't) formula |
  MatchP 't \<I> ('a, 't) regex | MatchF 't \<I> ('a, 't) regex
  and ('a, 't) regex = Test ('a, 't) formula | Wild |
  Plus ('a, 't) regex ('a, 't) regex | Times ('a, 't) regex ('a, 't) regex |
  Star ('a, 't) regex

end

fun mdl2mdl :: ('a, 't :: timestamp) Preliminaries.formula ⇒ ('a, 't) formula
  and embed :: ('a, 't) Preliminaries.regex ⇒ ('a, 't) regex where
  mdl2mdl (Preliminaries.Bool b) = Bool b
| mdl2mdl (Preliminaries.Atom a) = Atom a
| mdl2mdl (Preliminaries.Neg phi) = Neg (mdl2mdl phi)
| mdl2mdl (Preliminaries.Bin f phi psi) = Bin f (mdl2mdl phi) (mdl2mdl psi)
| mdl2mdl (Preliminaries.Prev I phi) = Prev I (mdl2mdl phi)
| mdl2mdl (Preliminaries.Next I phi) = Next I (mdl2mdl phi)
| mdl2mdl (Preliminaries.Since phi I psi) = Since (mdl2mdl phi) I (mdl2mdl psi)
| mdl2mdl (Preliminaries.Until phi I psi) = Until (mdl2mdl phi) I (mdl2mdl psi)
| mdl2mdl (Preliminaries.MatchP I r) = MatchP I (Times (embed r) (Symbol (Bool True)))
| mdl2mdl (Preliminaries.MatchF I r) = MatchF I (Times (embed r) (Symbol (Bool True)))
| embed (Preliminaries.Test phi) = Lookahead (mdl2mdl phi)
| embed Preliminaries.Wild = Symbol (Bool True)
| embed (Preliminaries.Plus r s) = Plus (embed r) (embed s)
| embed (Preliminaries.Times r s) = Times (embed r) (embed s)
| embed (Preliminaries.Star r) = Star (embed r)

lemma mdl2mdl_wf:
  fixes phi :: ('a, 't :: timestamp) Preliminaries.formula
  shows wf_fmla (mdl2mdl phi)
  ⟨proof⟩

fun embed' :: (('a, 't :: timestamp) formula ⇒ ('a, 't) Preliminaries.formula) ⇒ ('a, 't) regex ⇒ ('a, 't)
Preliminaries.regex where
  embed' f (Lookahead phi) = Preliminaries.Test (f phi)
| embed' f (Symbol phi) = Preliminaries.Times (Preliminaries.Test (f phi)) Preliminaries.Wild
| embed' f (Plus r s) = Preliminaries.Plus (embed' f r) (embed' f s)
| embed' f (Times r s) = Preliminaries.Times (embed' f r) (embed' f s)
| embed' f (Star r) = Preliminaries.Star (embed' f r)

lemma embed'_cong[fundef_cong]: (⋀phi. phi ∈ atms r ⟹ f phi = f' phi) ⟹ embed' f r = embed' f'
r
  ⟨proof⟩

fun mdl2mdl' :: ('a, 't :: timestamp) formula ⇒ ('a, 't) Preliminaries.formula where
  mdl2mdl' (Bool b) = Preliminaries.Bool b
| mdl2mdl' (Atom a) = Preliminaries.Atom a
| mdl2mdl' (Neg phi) = Preliminaries.Neg (mdl2mdl' phi)
| mdl2mdl' (Bin f phi psi) = Preliminaries.Bin f (mdl2mdl' phi) (mdl2mdl' psi)
| mdl2mdl' (Prev I phi) = Preliminaries.Prev I (mdl2mdl' phi)
| mdl2mdl' (Next I phi) = Preliminaries.Next I (mdl2mdl' phi)
| mdl2mdl' (Since phi I psi) = Preliminaries.Since (mdl2mdl' phi) I (mdl2mdl' psi)
| mdl2mdl' (Until phi I psi) = Preliminaries.Until (mdl2mdl' phi) I (mdl2mdl' psi)
| mdl2mdl' (MatchP I r) = Preliminaries.MatchP I (embed' mdl2mdl' (rderive r))
| mdl2mdl' (MatchF I r) = Preliminaries.MatchF I (embed' mdl2mdl' (rderive r))

context MDL
begin
```

fun *rvsat* :: ('*a*, '*t*) *Preliminaries.formula* ⇒ *nat* ⇒ *bool*
  and *rvmatch* :: ('*a*, '*t*) *Preliminaries.regex* ⇒ (*nat* × *nat*) *set* **where**
  rvsat (*Preliminaries.Bool b*) *i* = *b*
| *rvsat* (*Preliminaries.Atom a*) *i* = (*a* ∈ Γ *σ i*)
| *rvsat* (*Preliminaries.Neg φ*) *i* = (¬ *rvsat φ i*)
| *rvsat* (*Preliminaries.Bin f φ ψ*) *i* = (*f* (*rvsat φ i*) (*rvsat ψ i*))
| *rvsat* (*Preliminaries.Prev I φ*) *i* = (*case i of 0* ⇒ *False* | *Suc j* ⇒ *mem* (*τ σ j*) (*τ σ i*) *I* ∧ *rvsat φ j*)
| *rvsat* (*Preliminaries.Next I φ*) *i* = (*mem* (*τ σ i*) (*τ σ* (*Suc i*)) *I* ∧ *rvsat φ* (*Suc i*))
| *rvsat* (*Preliminaries.Since φ I ψ*) *i* = (∃ *j*≤*i*. *mem* (*τ σ j*) (*τ σ i*) *I* ∧ *rvsat ψ j* ∧ (∀ *k* ∈ {*j*<..*i*}. *rvsat φ k*))
| *rvsat* (*Preliminaries.Until φ I ψ*) *i* = (∃ *j*≥*i*. *mem* (*τ σ i*) (*τ σ j*) *I* ∧ *rvsat ψ j* ∧ (∀ *k* ∈ {*i*..<*j*}. *rvsat φ k*))
| *rvsat* (*Preliminaries.MatchP I r*) *i* = (∃ *j*≤*i*. *mem* (*τ σ j*) (*τ σ i*) *I* ∧ (*j*, *i*) ∈ *rvmatch r*)
| *rvsat* (*Preliminaries.MatchF I r*) *i* = (∃ *j*≥*i*. *mem* (*τ σ i*) (*τ σ j*) *I* ∧ (*i*, *j*) ∈ *rvmatch r*)
| *rvmatch* (*Preliminaries.Test φ*) = {(*i*, *i*) | *i*. *rvsat φ i*}
| *rvmatch Preliminaries.Wild* = {(*i*, *i* + *1*) | *i*. *True*}
| *rvmatch* (*Preliminaries.Plus r s*) = *rvmatch r* ∪ *rvmatch s*
| *rvmatch* (*Preliminaries.Times r s*) = *rvmatch r O rvmatch s*
| *rvmatch* (*Preliminaries.Star r*) = *rtrancl* (*rvmatch r*)

lemma *mdl2mdl_equivalent*:
  fixes *phi* :: ('*a*, '*t* :: *timestamp*) *Preliminaries.formula*
  shows ⋀*i*. *sat* (*mdl2mdl phi*) *i* ⟷ *rvsat phi i*
  ⟨*proof*⟩

lemma *mdlstar2mdl*:
  fixes *phi* :: ('*a*, '*t* :: *timestamp*) *Preliminaries.formula*
  shows *wf_fmla* (*mdl2mdl phi*) ⋀*i*. *sat* (*mdl2mdl phi*) *i* ⟷ *rvsat phi i*
  ⟨*proof*⟩

lemma *rvmatch_embed'*:
  assumes ⋀*phi i*. *phi* ∈ *atms r* ⟹ *rvsat* (*mdl2mdl' phi*) *i* ⟷ *sat phi i*
  shows *rvmatch* (*embed' mdl2mdl' r*) = *match r*
  ⟨*proof*⟩

lemma *mdl2mdlstar*:
  fixes *phi* :: ('*a*, '*t* :: *timestamp*) *formula*
  assumes *wf_fmla phi*
  shows ⋀*i*. *rvsat* (*mdl2mdl' phi*) *i* ⟷ *sat phi i*
  ⟨*proof*⟩

end

end
theory *Monitor_Code*
  imports *HOL−Library.Code_Target_Nat Containers.Containers Monitor Preliminaries*
begin

derive (*eq*) *ceq enat*

instantiation *enat* :: *ccompare* **begin**

definition *ccompare_enat* :: *enat comparator option* **where**
  ccompare_enat = *Some* (λ*x y*. *if x* = *y then order.Eq else if x* < *y then order.Lt else order.Gt*)

instance ⟨*proof*⟩

end

```
code_printing
  code_module IArray ⇀ (OCaml)
‹module IArray : sig
  val length' : 'a array -> Z.t
  val sub' : 'a array * Z.t -> 'a
end = struct

let length' xs = Z.of_int (Array.length xs);;

let sub' (xs, i) = Array.get xs (Z.to_int i);;

end› for type_constructor iarray constant IArray.length' IArray.sub'

code_reserved (OCaml) IArray

code_printing
  type_constructor iarray ⇀ (OCaml) _ array
| constant iarray_of_list ⇀ (OCaml) Array.of'_list
| constant IArray.list_of ⇀ (OCaml) Array.to'_list
| constant IArray.length' ⇀ (OCaml) IArray.length'
| constant IArray.sub' ⇀ (OCaml) IArray.sub'

lemma iarray_list_of_inj: IArray.list_of xs = IArray.list_of ys ⟹ xs = ys
  ⟨proof⟩

instantiation iarray :: (ccompare) ccompare
begin

definition ccompare_iarray :: ('a iarray ⇒ 'a iarray ⇒ order) option where
  ccompare_iarray = (case ID CCOMPARE('a list) of None ⇒ None
  | Some c ⇒ Some (λxs ys. c (IArray.list_of xs) (IArray.list_of ys)))

instance
  ⟨proof⟩

end

derive (rbt) mapping_impl iarray

definition mk_db :: String.literal list ⇒ String.literal set where mk_db = set

definition init_vydra_string_enat :: _ ⇒ _ ⇒ _ ⇒ (String.literal, enat, 'e) vydra where
  init_vydra_string_enat = init_vydra
definition run_vydra_string_enat :: _ ⇒ (String.literal, enat, 'e) vydra ⇒ _ where
  run_vydra_string_enat = run_vydra
definition progress_enat :: (String.literal, enat) formula ⇒ enat list ⇒ nat where
  progress_enat = progress
definition bounded_future_fmla_enat :: (String.literal, enat) formula ⇒ bool where
  bounded_future_fmla_enat = bounded_future_fmla
definition wf_fmla_enat :: (String.literal, enat) formula ⇒ bool where
  wf_fmla_enat = wf_fmla
definition mdl2mdl'_enat :: (String.literal, enat) formula ⇒ (String.literal, enat) Preliminaries.formula
where
  mdl2mdl'_enat = mdl2mdl'
definition interval_enat :: enat ⇒ enat ⇒ bool ⇒ bool ⇒ enat ℐ where
  interval_enat = interval
definition rep_interval_enat :: enat ℐ ⇒ enat × enat × bool × bool where
```

rep_interval_enat = Rep_ℐ

definition *init_vydra_string_ereal* :: _ ⇒ _ ⇒ _ ⇒ (*String.literal*, *ereal*, *'e*) *vydra* **where**
init_vydra_string_ereal = init_vydra
definition *run_vydra_string_ereal* :: _ ⇒ (*String.literal*, *ereal*, *'e*) *vydra* ⇒ _ **where**
run_vydra_string_ereal = run_vydra
definition *progress_ereal* :: (*String.literal*, *ereal*) *formula* ⇒ *ereal list* ⇒ *real* **where**
progress_ereal = progress
definition *bounded_future_fmla_ereal* :: (*String.literal*, *ereal*) *formula* ⇒ *bool* **where**
bounded_future_fmla_ereal = bounded_future_fmla
definition *wf_fmla_ereal* :: (*String.literal*, *ereal*) *formula* ⇒ *bool* **where**
wf_fmla_ereal = wf_fmla
definition *mdl2mdl'_ereal* :: (*String.literal*, *ereal*) *formula* ⇒ (*String.literal*, *ereal*) *Preliminaries.formula*
where
mdl2mdl'_ereal = mdl2mdl'
definition *interval_ereal* :: *ereal* ⇒ *ereal* ⇒ *bool* ⇒ *bool* ⇒ *ereal ℐ* **where**
interval_ereal = interval
definition *rep_interval_ereal* :: *ereal ℐ* ⇒ *ereal* × *ereal* × *bool* × *bool* **where**
rep_interval_ereal = Rep_ℐ

lemma *tfin_enat_code*[*code*]: (*tfin* :: *enat set*) = *Collect_set* (λ*x*. *x* ≠ ∞)
⟨*proof*⟩

lemma *tfin_ereal_code*[*code*]: (*tfin* :: *ereal set*) = *Collect_set* (λ*x*. *x* ≠ −∞ ∧ *x* ≠ ∞)
⟨*proof*⟩

lemma *Ball_atms*[*code_unfold*]: *Ball* (*atms r*) *P* = *list_all P* (*collect_subfmlas r* [])
⟨*proof*⟩

lemma *MIN_fold*: (*MIN x*∈*set* (*z* # *zs*). *f x*) = *fold min* (*map f zs*) (*f z*)
⟨*proof*⟩

declare *progress.simps*(1−8)[*code*]

lemma *progress_matchP_code*[*code*]:
progress (*MatchP I r*) *ts* = (*case collect_subfmlas r* [] *of x* # *xs* ⇒ *fold min* (*map* (λ*f*. *progress f ts*)
xs) (*progress x ts*))
⟨*proof*⟩

lemma *progress_matchF_code*[*code*]:
progress (*MatchF I r*) *ts* = (*if length ts* = *0 then 0 else*
(*let k* = *min* (*length ts* − *1*) (*case collect_subfmlas r* [] *of x* # *xs* ⇒ *fold min* (*map* (λ*f*. *progress f ts*)
xs) (*progress x ts*)) *in*
Min {*j* ∈ {..*k*}. *memR* (*ts* ! *j*) (*ts* ! *k*) *I*}))
⟨*proof*⟩

export_code *init_vydra_string_enat run_vydra_string_enat progress_enat bounded_future_fmla_enat*
wf_fmla_enat mdl2mdl'_enat
Bool Preliminaries.Bool enat interval_enat rep_interval_enat nat_of_integer integer_of_nat mk_db
in *OCaml* **module_name** *VYDRA* **file_prefix** *verified*

end
theory *Timestamp_Lex*
imports *Timestamp*
begin

instantiation *prod* :: (*timestamp_total_strict*, *timestamp_total_strict*) *timestamp_total_strict*
begin

definition *tfin_prod* :: $('a \times 'b)$ *set* **where**
tfin_prod = *tfin* × *UNIV*

definition ι*_prod* :: *nat* ⇒ $'a \times 'b$ **where**
ι*_prod n* = (ι *n*, ι *n*)

fun *sup_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ **where**
sup_prod (*a*, *b*) (*c*, *d*) = (*if a* < *c then* (*c*, *d*) *else if c* < *a then* (*a*, *b*) *else* (*a*, *sup b d*))

fun *less_eq_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less_eq_prod (*a*, *b*) (*c*, *d*) ⟷ *a* < *c* ∨ (*a* = *c* ∧ *b* ≤ *d*)

definition *less_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less_prod x y ⟷ *x* ≤ *y* ∧ *x* ≠ *y*

instance
⟨*proof*⟩

end

end
theory *Timestamp_Prod*
imports *Timestamp*
begin

instantiation *prod* :: (*timestamp*, *timestamp*) *timestamp*
begin

definition *tfin_prod* :: $('a \times 'b)$ *set* **where**
tfin_prod = *tfin* × *tfin*

definition ι*_prod* :: *nat* ⇒ $'a \times 'b$ **where**
ι*_prod n* = (ι *n*, ι *n*)

fun *sup_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow 'a \times 'b$ **where**
sup_prod (*a*, *b*) (*c*, *d*) = (*sup a c*, *sup b d*)

fun *less_eq_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less_eq_prod (*a*, *b*) (*c*, *d*) ⟷ *a* ≤ *c* ∧ *b* ≤ *d*

definition *less_prod* :: $'a \times 'b \Rightarrow 'a \times 'b \Rightarrow$ *bool* **where**
less_prod x y ⟷ *x* ≤ *y* ∧ *x* ≠ *y*

instance
⟨*proof*⟩

end

end

References

[1] R. Koymans. Specifying real-time properties with metric temporal logic. *Real Time Syst.*, 2(4):255–299, 1990.

[2] M. Raszyk, D. A. Basin, and D. Traytel. Multi-head monitoring of metric dynamic logic. In

D. V. Hung and O. Sokolsky, editors, *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, volume 12302 of *Lecture Notes in Computer Science*, pages 233–250. Springer, 2020.