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 We have also formalized unpublished specialized algorithms for the temporal operators of metric temporal logic.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

1 Intervals **5**

2 Infinite Traces **6**

3 Formulas and Satisfiability **8**

4 Formulas and Satisfiability **136**

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 $c \leq d \Longrightarrow c + a \leq d + a$
by (*auto simp*: *add.commute add_mono*)

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
apply *standard*
apply (*auto simp*: *plus_option_def add.assoc split*: *option.splits*)[*1*]
apply (*auto simp*: *plus_option_def add.commute split*: *option.splits*)[*1*]
apply (*auto simp*: *zero_option_def plus_option_def split*: *option.splits*)[*1*]
apply (*auto simp*: *less_option_def less_eq_option_def split*: *option.splits*)[*1*]
apply (*auto simp*: *less_eq_option_def split*: *option.splits*)[*3*]
apply (*auto simp*: *sup_option_def less_eq_option_def split*: *option.splits*)[*3*]
apply (*auto simp*: $\iota\_option\_def\ less\_eq\_option\_def\ intro$: ι_mono)[*1*]
apply (*auto simp*: $tfin\_option\_def\ \iota\_option\_def\ intro$: ι_tfin)[*1*]
apply (*auto simp*: $tfin\_option\_def\ \iota\_option\_def\ plus\_option\_def\ less\_eq\_option\_def\ intro$: $\iota\_progressing$)[*1*]
apply (*auto simp*: $tfin\_option\_def\ zero\_option\_def\ intro$: $zero\_tfin$)[*1*]
apply (*auto simp*: $tfin\_option\_def\ plus\_option\_def\ intro$: $tfin\_closed$)[*1*]
apply (*auto simp*: *plus_option_def less_eq_option_def intro*: *add_mono split*: *option.splits*)[*1*]

apply (*auto simp*: *tfin_option_def zero_option_def plus_option_def less_option_def intro*: *add_pos split*: *option.splits*)
done

end

instantiation *enat* :: *timestamp*
begin

definition *tfin_enat* :: *enat set* **where**
tfin_enat = *UNIV* − {∞}

definition *ι_enat* :: *nat* ⇒ *enat* **where**
ι_enat n = *n*

instance
by *standard* (*auto simp add*: *ι_enat_def tfin_enat_def dest*!: *leD*)

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*
by (*auto simp*: *tfin_ereal_def*) (*metis add.right_neutral ereal_add_cancel_left ereal_le_add_self order_less_le*)

instance
by *standard* (*auto simp add*: *ι_ereal_def tfin_ereal_def add.commute ereal_add_le_add_iff2 not_le less_PInf_Ex_of_nat ereal_less_ereal_Ex reals_Archimedean2 intro*: *ereal_add_pos*)

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*
by (*metis add_0_left local.add_mono_comm timestamp_tfin_le_not_tfin dual_order.order_iff_strict dual_order.strict_trans1*)

end

instantiation *enat* :: *timestamp_total*
begin

instance
by *standard* (*auto simp*: *tfin_enat_def*)

end

instantiation *ereal* :: *timestamp_total*
begin

instance
by *standard* (*auto simp*: *tfin_ereal_def*)

end

class *timestamp_strict* = *timestamp* +
assumes *add_mono_strict*: $c < d \Longrightarrow a + c < a + d$

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* $\Rightarrow$ *nat* **where**
ι*_nat n* = *n*

instance
by *standard* (*auto simp*: *tfin_nat_def* ι*_nat_def dest*!: *leD*)

end

instantiation *real* :: *timestamp_total_strict*
begin

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

definition ι*_real* :: *nat* $\Rightarrow$ *real* **where** ι*_real n* = *real n*

instance
by *standard* (*auto simp*: *tfin_real_def* ι*_real_def not_le reals_Archimedean2*)

end

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

definition *zero_prod* :: $'a \times 'b$ **where**
zero_prod = (*0*, *0*)

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

instance
by *standard* (*auto simp*: *zero_prod_def ac_simps*)

end

end

1 Intervals

typedef (**overloaded**) ('*a* :: *timestamp*) $\mathcal{I}$ = {(*i* :: '*a*, *j* :: '*a*, *lei* :: *bool*, *lej* :: *bool*). $0 \leq i \wedge i \leq j \wedge i \in tfin \wedge \neg(j = 0 \wedge \neg lej)$}
by (*intro exI*[*of* _ (0, 0, *True*, *True*)]) (*auto intro*: *zero_tfin*)

setup_lifting *type_definition_*$\mathcal{I}$

instantiation $\mathcal{I}$ :: (*timestamp*) *equal* **begin**

lift_definition *equal_*$\mathcal{I}$ :: '*a* $\mathcal{I}$ ⇒ '*a* $\mathcal{I}$ ⇒ *bool* **is** (=) .

instance by *standard* (*transfer*, *auto*)

end

lift_definition *right* :: '*a* :: *timestamp* $\mathcal{I}$ ⇒ '*a* **is** *fst* ∘ *snd* .

lift_definition *memL* :: '*a* :: *timestamp* ⇒ '*a* ⇒ '*a* $\mathcal{I}$ ⇒ *bool* **is**
$\lambda t\ t'$ (*a*, *b*, *lei*, *lej*). *if lei then* $t + a \leq t'$ *else* $t + a < t'$.

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

definition *mem* :: '*a* :: *timestamp* ⇒ '*a* ⇒ '*a* $\mathcal{I}$ ⇒ *bool* **where**
mem t t' I ⟷ *memL t t' I* ∧ *memR t t' I*

lemma *memL_mono*: *memL t t' I* ⟹ $t'' \leq t$ ⟹ *memL t'' t' I*
by *transfer* (*auto simp*: *add.commute order_le_less_subst2 order_subst2 add_mono split*: *if_splits*)

lemma *memL_mono'*: *memL t t' I* ⟹ $t' \leq t''$ ⟹ *memL t t'' I*
by *transfer* (*auto split*: *if_splits*)

lemma *memR_mono*: *memR t t' I* ⟹ $t \leq t''$ ⟹ *memR t'' t' I*
apply *transfer*
apply (*simp split*: *prod.splits*)
apply (*meson add_mono_comm dual_order.trans order_less_le_trans*)
done

lemma *memR_mono'*: *memR t t' I* ⟹ $t'' \leq t'$ ⟹ *memR t t'' I*
by *transfer* (*auto split*: *if_splits*)

lemma *memR_dest*: *memR t t' I* ⟹ $t' \leq t +$ *right I*
by *transfer* (*auto split*: *if_splits*)

lemma *memR_tfin_refl*:
assumes *fin*: $t \in tfin$
shows *memR t t I*
by (*transfer fixing*: *t*) (*force split*: *if_splits intro*: *order_trans*[*OF* _ *add_mono*, **where** *?x=t* **and** *?a1=t* **and** *?c1=0*] *add_pos*[*OF fin*])

lemma *right_I_add_mono*:
fixes *x* :: '*a* :: *timestamp*
shows $x \leq x +$ *right I*
by *transfer* (*auto split*: *if_splits intro*: *order_trans*[*OF* _ *add_mono*, *of* _ _ 0])

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

by (*auto intro*: *zero_tfin*)

lemma *Rep_ℐ I* = (*l*, *r*, *b1*, *b2*) ⟹ *memL 0 0 I* ⟷ *l* = *0* ∧ *b1*
by *transfer auto*

lift_definition *dropL* :: *'a* :: *timestamp ℐ* ⇒ *'a ℐ* **is**
λ(*l*, *r*, *b1*, *b2*). (*0*, *r*, *True*, *b2*)
by (*auto intro*: *zero_tfin*)

lemma *memL_dropL*: *t* ≤ *t'* ⟹ *memL t t'* (*dropL I*)
by *transfer auto*

lemma *memR_dropL*: *memR t t'* (*dropL I*) = *memR t t' I*
by *transfer auto*

lift_definition *flipL* :: *'a* :: *timestamp ℐ* ⇒ *'a ℐ* **is**
λ(*l*, *r*, *b1*, *b2*). *if* ¬(*l* = *0* ∧ *b1*) *then* (*0*, *l*, *True*, ¬*b1*) *else Code.abort* (*STR ''invalid flipL''*) (λ_. (*0*, *0*, *True*, *True*))
by (*auto intro*: *zero_tfin split*: *if_splits*)

lemma *memL_flipL*: *t* ≤ *t'* ⟹ *memL t t'* (*flipL I*)
by *transfer* (*auto split*: *if_splits*)

lemma *memR_flipLD*: ¬*memL 0 0 I* ⟹ *memR t t'* (*flipL I*) ⟹ ¬*memL t t' I*
by *transfer* (*auto split*: *if_splits*)

lemma *memR_flipLI*:
fixes *t* :: *'a* :: *timestamp*
shows (⋀*u v*. (*u* :: *'a* :: *timestamp*) ≤ *v* ∨ *v* ≤ *u*) ⟹ ¬*memL t t' I* ⟹ *memR t t'* (*flipL I*)
by *transfer* (*force split*: *if_splits*)

lemma *t* ∈ *tfin* ⟹ *memL 0 0 I* ⟷ *memL t t I*
apply *transfer*
apply (*simp split*: *prod.splits*)
apply (*metis add.right_neutral add_pos antisym_conv2 dual_order.eq_iff order_less_imp_not_less*)
done

definition *full* (*I* :: (*'a* :: *timestamp*) *ℐ*) ⟷ (∀ *t t'*. *0* ≤ *t* ∧ *t* ≤ *t'* ∧ *t* ∈ *tfin* ∧ *t'* ∈ *tfin* ⟶ *mem t t' I*)

lemma *memL 0 0* (*I* :: (*'a* :: *timestamp_total*) *ℐ*) ⟹ *right I* ∉ *tfin* ⟹ *full I*
unfolding *full_def mem_def*
by *transfer* (*fastforce split*: *if_splits dest*: *add_not_tfin*)

2 Infinite Traces

inductive *sorted_list* :: *'a* :: *order list* ⇒ *bool* **where**
[*intro*]: *sorted_list* []
| [*intro*]: *sorted_list* [*x*]
| [*intro*]: *x* ≤ *y* ⟹ *sorted_list* (*y* # *ys*) ⟹ *sorted_list* (*x* # *y* # *ys*)

lemma *sorted_list_app*: *sorted_list xs* ⟹ (⋀*x*. *x* ∈ *set xs* ⟹ *x* ≤ *y*) ⟹ *sorted_list* (*xs* @ [*y*])
by (*induction xs rule*: *sorted_list.induct*) *auto*

lemma *sorted_list_drop*: *sorted_list xs* ⟹ *sorted_list* (*drop n xs*)
proof (*induction xs arbitrary*: *n rule*: *sorted_list.induct*)
case (*2 x n*)

```
  then show ?case
    by (cases n) auto
next
  case (3 x y ys n)
  then show ?case
    by (cases n) auto
qed auto

lemma sorted_list_ConsD: sorted_list (x # xs) ⟹ sorted_list xs
  by (auto elim: sorted_list.cases)

lemma sorted_list_Cons_nth: sorted_list (x # xs) ⟹ j < length xs ⟹ x ≤ xs ! j
  by (induction x # xs arbitrary: x xs j rule: sorted_list.induct)
     (fastforce simp: nth_Cons split: nat.splits)+

lemma sorted_list_atD: sorted_list xs ⟹ i ≤ j ⟹ j < length xs ⟹ xs ! i ≤ xs ! j
proof (induction xs arbitrary: i j rule: sorted_list.induct)
  case (2 x i j)
  then show ?case
    by (cases i) auto
next
  case (3 x y ys i j)
  have x ≤ (x # y # ys) ! j
    using 3(5) sorted_list_Cons_nth[OF sorted_list.intros(3)[OF 3(1,2)]]
    by (auto simp: nth_Cons split: nat.splits)
  then show ?case
    using 3
    by (cases i) auto
qed auto

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

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

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

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

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

lemma sorted_stake: ssorted s ⟹ sorted_list (stake i s)
proof (induct i arbitrary: s)
  case (Suc i)
  then show ?case
    by (cases i) (auto elim: ssorted.cases)
qed auto

lemma ssorted_monoI: ∀ i j. i ≤ j ⟶ s !! i ≤ s !! j ⟹ ssorted s
  by (coinduction arbitrary: s)
```

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

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

typedef (**overloaded**) ('*a*, '*b* :: *timestamp*) *trace* = {*s* :: ('*a set* × '*b*) *stream*.
ssorted (*smap snd s*) ∧ (∀ *x*. *x* ∈ *snd* ‘ *sset s* ⟶ *x* ∈ *tfin*) ∧ (∀ *i x*. *x* ∈ *tfin* ⟶ (∃ *j*. ¬*snd* (*s* !! *j*) ≤ *snd* (*s* !! *i*) + *x*))}
by (*auto simp*: *ι_mono ι_tfin ι_progressing stream.set_map*
intro!: *exI*[*of* _ *smap* (λ*n*. ({}, *ι n*)) *nats*] *ssorted_monoI*)

setup_lifting *type_definition_trace*

lift_definition Γ :: ('*a*, '*b* :: *timestamp*) *trace* ⇒ *nat* ⇒ '*a set* **is**
λ*s i*. *fst* (*s* !! *i*) .
lift_definition τ :: ('*a*, '*b* :: *timestamp*) *trace* ⇒ *nat* ⇒ '*b* **is**
λ*s i*. *snd* (*s* !! *i*) .

lemma τ_*mono*[*simp*]: *i* ≤ *j* ⟹ τ *s i* ≤ τ *s j*
by *transfer* (*auto simp*: *ssorted_iff_mono*)

lemma τ_*fin*: τ *σ i* ∈ *tfin*
by *transfer auto*

lemma *ex_lt_*τ: *x* ∈ *tfin* ⟹ ∃ *j*. ¬τ *s j* ≤ τ *s i* + *x*
by *transfer auto*

lemma *le_*τ_*less*: τ *σ i* ≤ τ *σ j* ⟹ *j* < *i* ⟹ τ *σ i* = τ *σ j*
by (*simp add*: *antisym*)

lemma *less_*τ*D*: τ *σ i* < τ *σ j* ⟹ *i* < *j*
by (*meson* τ_*mono less_le_not_le not_le_imp_less*)

theory *MDL*
imports *Interval Trace*
begin

3 Formulas and Satisfiability

declare [[*typedef_overloaded*]]

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* *I* ('*a*, '*t*) *formula* | *Next* '*t* *I* ('*a*, '*t*) *formula* |
Since ('*a*, '*t*) *formula* '*t* *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* = *Lookahead* ('*a*, '*t*) *formula* | *Symbol* ('*a*, '*t*) *formula* |
Plus ('*a*, '*t*) *regex* ('*a*, '*t*) *regex* | *Times* ('*a*, '*t*) *regex* ('*a*, '*t*) *regex* |
Star ('*a*, '*t*) *regex*

fun *eps* :: ('*a*, '*t* :: *timestamp*) *regex* ⇒ *bool* **where**
eps (*Lookahead phi*) = *True*
| *eps* (*Symbol phi*) = *False*
| *eps* (*Plus r s*) = (*eps r* ∨ *eps s*)
| *eps* (*Times r s*) = (*eps r* ∧ *eps s*)
| *eps* (*Star r*) = *True*

fun *atms* :: ('*a*, '*t* :: *timestamp*) *regex* ⇒ ('*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*
  by (*induction r*) *auto*

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* ((λ*f*. *progress f ts*) ‘ *atms r*)
| *progress* (*MatchF I r*) *ts* = (*if length ts* = *0 then 0 else*
  (*let k* = *min* (*length ts* − *1*) (*Min* ((λ*f*. *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*
by (*induction r*) (*auto split*: *if_splits*)

lemma *size_formula_positive*: *size* (*phi* :: (′*a*, ′*t* :: *timestamp*) *formula*) > *0*
by (*induction phi*) *auto*

lemma *size_regex_positive*: *size* (*r* :: (′*a*, ′*t* :: *timestamp*) *regex*) > *Suc 0*
by (*induction r*) (*auto intro*: *size_formula_positive*)

lemma *size_rderive*[*termination_simp*]: *phi* ∈ *atms* (*rderive r*) ⟹ *size phi* < *size r*
by (*drule atms_rderive*) (*auto intro*: *size_atms size_regex_positive*)

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*

```
  apply (auto split: nat.splits)
  done

lemma prev_rewrite: sat (Prev I φ) i ⟷ sat (MatchP I (Times (Symbol φ) (Symbol (Bool True)))) i
  apply (auto split: nat.splits)
  subgoal for j
    by (fastforce intro: exI[of _ j])
  done

lemma next_rewrite: sat (Next I φ) i ⟷ sat (MatchF I (Times (Symbol (Bool True)) (Symbol φ))) i
  by (fastforce intro: exI[of _ Suc i])

lemma trancl_Base: {(i, Suc i) |i. P i}* = {(i, j). i ≤ j ∧ (∀k∈{i..<j}. P k)}
proof −
  have (x, y) ∈ {(i, j). i ≤ j ∧ (∀k∈{i..<j}. P k)}
    if (x, y) ∈ {(i, Suc i) |i. P i}* for x y
    using that by (induct rule: rtrancl_induct) (auto simp: less_Suc_eq)
  moreover have (x, y) ∈ {(i, Suc i) |i. P i}*
    if (x, y) ∈ {(i, j). i ≤ j ∧ (∀k∈{i..<j}. P k)} for x y
    using that unfolding mem_Collect_eq prod.case Ball_def
    by (induct y arbitrary: x)
       (auto 0 3 simp: le_Suc_eq intro: rtrancl_into_rtrancl[rotated])
  ultimately show ?thesis by blast
qed

lemma Ball_atLeastLessThan_reindex:
  (∀k∈{j..<i}. P (Suc k)) = (∀k ∈ {j<..i}. P k)
  by (auto simp: less_eq_Suc_le less_eq_nat.simps split: nat.splits)

lemma since_rewrite: sat (Since φ I ψ) i ⟷ sat (MatchP I (Times (Symbol ψ) (Star (Symbol φ)))) i
proof (rule iffI)
  assume sat (Since φ I ψ) i
  then obtain j where j_def: j ≤ i mem (τ σ j) (τ σ i) I sat ψ j
    ∀k ∈ {j..<i}. sat φ (Suc k)
    by auto
  have k ∈ {Suc j..<Suc i} ⟹ (k, Suc k) ∈ match (Symbol φ) for k
    using j_def(4)
    by (cases k) auto
  then have (Suc j, Suc i) ∈ (match (Symbol φ))*
    using j_def(1) trancl_Base
    by auto
  then show sat (MatchP I (Times (Symbol ψ) (Star (Symbol φ)))) i
    using j_def(1,2,3)
    by auto
next
  assume sat (MatchP I (Times (Symbol ψ) (Star (Symbol φ)))) i
  then obtain j where j_def: j ≤ i mem (τ σ j) (τ σ i) I (Suc j, Suc i) ∈ (match (Symbol φ))* sat ψ j
    by auto
  have ⋀k. k ∈ {Suc j..<Suc i} ⟹ (k, Suc k) ∈ match (Symbol φ)
    using j_def(3) trancl_Base[of λk. (k, Suc k) ∈ match (Symbol φ)]
    by simp
  then have ∀k ∈ {j..<i}. sat φ (Suc k)
    by auto
  then show sat (Since φ I ψ) i
    using j_def(1,2,4) Ball_atLeastLessThan_reindex[of j i sat φ]
    by auto
qed
```

lemma *until_rewrite*: *sat* (*Until* φ *I* ψ) *i* $\longleftrightarrow$ *sat* (*MatchF I* (*Times* (*Star* (*Symbol* φ)) (*Symbol* ψ))) *i*
proof (*rule iffI*)
  assume *sat* (*Until* φ *I* ψ) *i*
  then obtain *j* **where** *j_def*: $j \geq i$ *mem* (τ σ *i*) (τ σ *j*) *I sat* ψ *j*
    $\forall k \in \{i..<j\}$. *sat* φ *k*
    by *auto*
  have $k \in \{i..<j\} \Longrightarrow$ (*k*, *Suc k*) $\in$ *match* (*Symbol* φ) **for** *k*
    using *j_def*(*4*)
    by *auto*
  then have (*i*, *j*) $\in$ (*match* (*Symbol* φ))*
    using *j_def*(*1*) *trancl_Base*
    by *simp*
  then show *sat* (*MatchF I* (*Times* (*Star* (*Symbol* φ)) (*Symbol* ψ))) *i*
    using *j_def*(*1*,*2*,*3*)
    by *auto*
next
  assume *sat* (*MatchF I* (*Times* (*Star* (*Symbol* φ)) (*Symbol* ψ))) *i*
  then obtain *j* **where** *j_def*: $j \geq i$ *mem* (τ σ *i*) (τ σ *j*) *I* (*i*, *j*) $\in$ (*match* (*Symbol* φ))* *sat* ψ *j*
    by *auto*
  have $\bigwedge k.\ k \in \{i..<j\} \Longrightarrow$ (*k*, *Suc k*) $\in$ *match* (*Symbol* φ)
    using *j_def*(*3*) *trancl_Base*[*of* $\lambda k.$ (*k*, *Suc k*) $\in$ *match* (*Symbol* φ)]
    by *auto*
  then have $\forall k \in \{i..<j\}$. *sat* φ *k*
    by *simp*
  then show *sat* (*Until* φ *I* ψ) *i*
    using *j_def*(*1*,*2*,*4*)
    by *auto*
qed

lemma *match_le*: (*i*, *j*) $\in$ *match r* $\Longrightarrow$ $i \leq j$
proof (*induction r arbitrary*: *i j*)
  case (*Times r s*)
  then show *?case* **using** *order.trans* **by** *fastforce*
next
  case (*Star r*)
  from *Star.prems* **show** *?case*
    unfolding *match.simps*
    by (*induct i j rule*: *rtrancl.induct*) (*force dest*: *Star.IH*)+
qed *auto*

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*)
  using *match_le* **by** *auto* (*metis le_iff_add nat_add_left_cancel_le*)

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*)
  by (*induction x z rule*: *rtrancl.induct*) *auto*

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*)
  by (*induction x z rule*: *rtrancl.induct*) *auto*

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 (*rule iffI*)
  assume *assms*: (*i*, *i* + *Suc n*) $\in$ *match* (*Star r*)
  obtain *k* **where** *k_def*: (*i*, *k*) $\in$ *local.match r i* $\leq$ *k i* $\neq$ *k*
    (*k*, *i* + *Suc n*) $\in$ (*local.match r*)*

```
    using rtrancl_unfold[OF assms[unfolded match.simps]] match_le by auto
  from k_def(4) have (k, i + Suc n) ∈ match (Star r)
    unfolding match.simps by simp
  then have k_le: k ≤ i + Suc n
    using match_le by blast
  from k_def(2,3) obtain k' where k'_def: k = i + Suc k'
    by (metis Suc_diff_Suc le_add_diff_inverse le_neq_implies_less)
  show ∃k ≤ n. (i, i + 1 + k) ∈ match r ∧ (i + 1 + k, i + Suc n) ∈ match (Star r)
    using k_def k_le unfolding k'_def by auto
next
  assume assms: ∃k ≤ n. (i, i + 1 + k) ∈ match r ∧
    (i + 1 + k, i + Suc n) ∈ match (Star r)
  then show (i, i + Suc n) ∈ match (Star r)
    by (induction n) auto
qed

lemma match_refl_eps: (i, i) ∈ match r ⟹ eps r
proof (induction r)
  case (Times r s)
  then show ?case
    using match_Times[where ?i=i and ?n=0]
    by auto
qed auto

lemma wf_regex_eps_match: wf_regex r ⟹ eps r ⟹ (i, i) ∈ match r
  by (induction r arbitrary: i) auto

lemma match_Star_unfold: i < j ⟹ (i, j) ∈ match (Star r) ⟹ ∃k ∈ {i..<j}. (i, k) ∈ match (Star
r) ∧ (k, j) ∈ match r
  using rtrancl_unfold'[of i j match r] match_le[of _ j r] match_le[of i _ Star r]
  by auto (meson atLeastLessThan_iff order_le_less)

lemma match_rderive: wf_regex r ⟹ i ≤ j ⟹ (i, Suc j) ∈ match r ⟷ (i, j) ∈ match (rderive r)
proof (induction r arbitrary: i j)
  case (Times r1 r2)
  then show ?case
    using match_refl_eps[of Suc j r2] match_le[of _ Suc j r2]
    apply (auto)
         apply (metis le_Suc_eq relcomp.simps)
        apply (meson match_le relcomp.simps)
       apply (metis le_SucE relcomp.simps)
      apply (meson relcomp.relcompI wf_regex_eps_match)
     apply (meson match_le relcomp.simps)
    apply (metis le_SucE relcomp.simps)
    apply (meson match_le relcomp.simps)
    done
next
  case (Star r)
  then show ?case
    using match_Star_unfold[of i Suc j r]
    by auto (meson match_le rtrancl.simps)
qed auto

end

lemma atms_nonempty: atms r ≠ {}
  by (induction r) auto
```

lemma *atms_finite*: *finite* (*atms r*)
by (*induction r*) *auto*

lemma *progress_le_ts*:
assumes $\bigwedge t.\ t \in set\ ts \Longrightarrow t \in tfin$
shows *progress phi ts* $\leq$ *length ts*
using *assms*
proof (*induction phi ts rule*: *progress.induct*)
case (*8 phi I psi ts*)
have *ts* $\neq$ [] $\Longrightarrow$ *Min* {*j*. *j* $\leq$ *min* (*length ts* − *Suc 0*) (*min* (*progress phi ts*) (*progress psi ts*)) ∧
memR (*ts* ! *j*) (*ts* ! *min* (*length ts* − *Suc 0*) (*min* (*progress phi ts*) (*progress psi ts*))) *I*}
$\leq$ *length ts*
apply (*rule le_trans*[*OF Min_le*[**where** *?x*=*min* (*length ts* − *Suc 0*) (*min* (*progress phi ts*) (*progress psi ts*))]])
apply (*auto simp*: *in_set_conv_nth intro*!: *memR_tfin_refl 8*(*3*))
apply (*metis One_nat_def diff_less length_greater_0_conv less_numeral_extra*(*1*) *min.commute min.strict_coboundedI2*)
done
then show *?case*
by *auto*
next
case (*9 I r ts*)
then show *?case*
using *atms_nonempty*[*of r*] *atms_finite*[*of r*]
by *auto* (*meson Min_le dual_order.trans finite_imageI image_iff*)
next
case (*10 I r ts*)
have *ts* $\neq$ [] $\Longrightarrow$ *Min* {*j*. *j* $\leq$ *min* (*length ts* − *Suc 0*) (*MIN f*∈*atms r*. *progress f ts*) ∧
memR (*ts* ! *j*) (*ts* ! *min* (*length ts* − *Suc 0*) (*MIN f*∈*atms r*. *progress f ts*)) *I*}
$\leq$ *length ts*
apply (*rule le_trans*[*OF Min_le*[**where** *?x*=*min* (*length ts* − *Suc 0*) (*Min* ((λ*f*. *progress f ts*) ‘ *atms r*))]])
apply (*auto simp*: *in_set_conv_nth intro*!: *memR_tfin_refl 10*(*2*))
apply (*metis One_nat_def diff_less length_greater_0_conv less_numeral_extra*(*1*) *min.commute min.strict_coboundedI2*)
done
then show *?case*
by *auto*
qed (*auto split*: *nat.splits*)

end
theory *Metric_Point_Structure*
imports *Interval*
begin

class *metric_domain* = *plus* + *zero* + *ord* +
assumes Δ*1*: $x + x' = x' + x$
and Δ*2*: $(x + x') + x'' = x + (x' + x'')$
and Δ*3*: $x + 0 = x$
and Δ*3′*: $x = 0 + x$
and Δ*4*: $x + x' = x + x'' \Longrightarrow x' = x''$
and Δ*4′*: $x + x'' = x' + x'' \Longrightarrow x = x'$
and Δ*5*: $x + x' = 0 \Longrightarrow x = 0$
and Δ*5′*: $x + x' = 0 \Longrightarrow x' = 0$

```
    and Δ6: ∃x''. x = x' + x'' ∨ x' = x + x''
    and metric_domain_le_def: x ≤ x' ⟷ (∃x''. x' = x + x'')
    and metric_domain_lt_def: x < x' ⟷ (∃x''. x'' ≠ 0 ∧ x' = x + x'')
begin

lemma metric_domain_pos: x ≥ 0
  using Δ3' local.metric_domain_le_def by auto

lemma less_eq_le_neq: x < x' ⟷ (x ≤ x' ∧ x ≠ x')
  apply (auto simp: metric_domain_le_def metric_domain_lt_def)
   apply (metis Δ3 Δ4)
  apply (metis Δ3)
  done

end



class metric_domain_timestamp = metric_domain + sup + embed_nat + tfin +
  assumes metric_domain_sup_def: sup x x' = (if x ≤ x' then x' else x)
    and metric_domain_ι_mono: ⋀i j. i ≤ j ⟹ ι i ≤ ι j
    and metric_domain_ι_progressing: ∃j. ¬ι j ≤ ι i + x
    and metric_domain_tfin_def: tfin = UNIV

subclass (in metric_domain_timestamp) timestamp
  apply unfold_locales
                apply (auto simp: Δ2)[1]
               apply (auto simp: Δ1)[1]
              apply (auto simp: Δ3'[symmetric])[1]
  subgoal for x y
    apply (auto simp: metric_domain_le_def metric_domain_lt_def)
     apply (metis Δ2 Δ3 Δ4 Δ5)
    apply (metis Δ2 Δ3)
    done
  using Δ6 apply (auto simp: metric_domain_le_def)[1]
  using Δ2 apply (auto simp: metric_domain_le_def)[1]
  subgoal for x y
    apply (auto simp: metric_domain_le_def metric_domain_lt_def)
    apply (metis Δ2 Δ3 Δ4 Δ5)
    done
  using Δ6 apply (fastforce simp: metric_domain_le_def metric_domain_sup_def)
  using Δ6 apply (fastforce simp: metric_domain_le_def metric_domain_sup_def)
        apply (auto simp: metric_domain_le_def metric_domain_sup_def)[1]
  using metric_domain_ι_mono apply (auto simp: metric_domain_le_def)[1]
      apply (auto simp: metric_domain_tfin_def)[1]
  using metric_domain_ι_progressing apply (auto simp: metric_domain_le_def)[1]
     apply (auto simp: metric_domain_tfin_def)[2]
  using Δ2 apply (auto simp: metric_domain_le_def)[1]
  using Δ1 Δ3 apply (auto simp: metric_domain_lt_def)
  done



locale metric_point_structure =
  fixes d :: 't :: {order} ⇒ 't ⇒ 'd :: metric_domain_timestamp
  assumes d1: d t t' = 0 ⟷ t = t'
    and d2: d t t' = d t' t
    and d3: t < t' ⟹ t' < t'' ⟹ d t t'' = d t t' + d t' t''
```

```
    and d3': t < t' ⟹ t' < t'' ⟹ d t'' t = d t'' t' + d t' t
begin

lemma metric_point_structure_memL_aux: t0 ≤ t ⟹ t ≤ t' ⟹ x ≤ d t t' ⟷ (d t0 t + x ≤ d t0
t')
  apply (rule iffI)
   apply (metis Δ1 add_0 add_mono_comm d1 d3 order_le_less)
  apply (cases t0 < t; cases t < t')
     apply (auto simp: metric_domain_le_def)
     apply (metis Δ4 ab_semigroup_add_class.add_ac(1) d3)
    apply (metis comm_monoid_add_class.add_0 group_cancel.add1 metric_domain_lt_def nless_le)
   apply (metis Δ3' d1)
  apply (metis add_0 d1)
  done

lemma metric_point_structure_memL_strict_aux: t0 ≤ t ⟹ t ≤ t' ⟹ x < d t t' ⟷ (d t0 t + x <
d t0 t')
  using metric_point_structure_memL_aux[of t0 t t' x]
  apply auto
   apply (metis (no_types, lifting) Δ1 Δ3 Δ4 antisym_conv2 d1 d3)
  apply (metis Δ1 add_0 d1 d3 order.order_iff_strict order_less_irrefl)
  done

lemma metric_point_structure_memR_aux: t0 ≤ t ⟹ t ≤ t' ⟹ d t t' ≤ x ⟷ (d t0 t' ≤ d t0 t +
x)
  apply auto
   apply (metis Δ1 Δ3 d1 d3 order_le_less add_mono)
  apply (smt (verit, ccfv_threshold) Δ1 Δ2 Δ3 Δ4 d1 d3 metric_domain_le_def order_le_less)
  done

lemma metric_point_structure_memR_strict_aux: t0 ≤ t ⟹ t ≤ t' ⟹ d t t' < x ⟷ (d t0 t' < d
t0 t + x)
  by (auto simp add: metric_point_structure_memL_aux metric_point_structure_memR_aux less_le_not_le)

lemma metric_point_structure_le_mem: t0 ≤ t ⟹ t ≤ t' ⟹ d t t' ≤ x ⟷ mem (d t0 t) (d t0 t')
(interval 0 x True True)
  unfolding mem_def
  apply (transfer fixing: d)
  using metric_point_structure_memR_aux
  apply (auto simp: metric_domain_le_def metric_domain_tfin_def)
  apply (metis add.right_neutral d3 order.order_iff_strict)
  done

lemma metric_point_structure_lt_mem: t0 ≤ t ⟹ t ≤ t' ⟹ 0 < x ⟹ d t t' < x ⟷ mem (d t0
t) (d t0 t') (interval 0 x True False)
  unfolding mem_def
  apply (transfer fixing: d)
  using metric_point_structure_memR_strict_aux
  apply (auto simp: metric_domain_tfin_def)
  apply (metis Δ3 metric_domain_pos metric_point_structure_memL_aux)
  done

lemma metric_point_structure_eq_mem: t0 ≤ t ⟹ t ≤ t' ⟹ d t t' = x ⟷ mem (d t0 t) (d t0 t')
(interval x x True True)
  unfolding mem_def
  apply (transfer fixing: d)
  subgoal for t0 t t' x
    using metric_point_structure_memL_aux[of t0 t t' x] metric_point_structure_memR_aux[of t0 t t'
```

```
x] metric_domain_pos
    by (auto simp: metric_domain_tfin_def)
  done

lemma metric_point_structure_ge_mem: t0 ≤ t ⟹ t ≤ t' ⟹ x ≤ d t t' ⟷ mem (Some (d t0 t))
(Some (d t0 t')) (interval (Some x) None True True)
  unfolding mem_def
  apply (transfer fixing: d)
  using metric_point_structure_memL_aux by (auto simp: tfin_option_def zero_option_def plus_option_def
less_eq_option_def metric_domain_le_def metric_domain_tfin_def split: option.splits)

lemma metric_point_structure_gt_mem: t0 ≤ t ⟹ t ≤ t' ⟹ x < d t t' ⟷ mem (Some (d t0 t))
(Some (d t0 t')) (interval (Some x) None False True)
  unfolding mem_def
  apply (transfer fixing: d)
   using metric_point_structure_memL_strict_aux by (auto simp: tfin_option_def zero_option_def
plus_option_def less_option_def less_eq_option_def metric_domain_le_def metric_domain_tfin_def
split: option.splits)

end

instantiation nat :: metric_domain_timestamp
begin

instance
  apply standard
            apply auto[8]
       apply (meson less_eqE timestamp_total)
  using nat_le_iff_add apply blast
  using less_imp_add_positive apply auto[1]
    apply (auto simp: sup_max)[1]
   apply (auto simp: ι_nat_def)[1]
  subgoal for i x
    apply (auto simp: ι_nat_def)
    using add_le_same_cancel1 by blast
  apply (auto simp: tfin_nat_def)
  done

end

interpretation nat_metric_point_structure: metric_point_structure λt :: nat. λt'. if t ≤ t' then t' − t
else t − t'
  by unfold_locales auto

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*
  by (*induction q q' rule*: *rtranclp.induct*) *auto*

lemma *rtranclp_closed_sub*: *rtranclp R q q'* ⟹ {*q'*. ∃ *q* ∈ *X*. *R q q'*} ⊆ *X* ⟹
  q ∈ *X* ⟹ *q'* ∈ *X*
  by (*induction q q' rule*: *rtranclp.induct*) *auto*

lemma *rtranclp_closed_sub'*: *rtranclp R q q'* ⟹ *q'* = *q* ∨ (∃ *q''*. *R q q''* ∧ *rtranclp R q'' q'*)
  using *converse_rtranclpE* **by** *force*

lemma *rtranclp_step*: *rtranclp R q q''* ⟹ (⋀*q'*. *R q q'* ⟷ *q'* ∈ *X*) ⟹
  q = *q''* ∨ (∃ *q'* ∈ *X*. *R q q'* ∧ *rtranclp R q' q''*)
  by (*induction q q'' rule*: *rtranclp.induct*)
    (*auto intro*: *rtranclp.rtrancl_into_rtrancl*)

lemma *rtranclp_unfold*: *rtranclp R x z* ⟹ *x* = *z* ∨ (∃ *y*. *R x y* ∧ *rtranclp R y z*)
  by (*induction x z rule*: *rtranclp.induct*) *auto*

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*
  by (*auto simp*: *SQ_def*)

lemma *finite_SQ*: *finite SQ*
  by (*auto simp add*: *SQ_def*)

lemma *transs_q_in_set*: *q* ∈ *SQ* ⟹ *transs* ! (*q* − *q0*) ∈ *set transs*
  by (*auto simp add*: *SQ_def*)

qualified definition *Q* :: *state set* **where**
  Q = *SQ* ∪ {*qf*}

lemma *finite_Q*: *finite Q*
  by (*auto simp add*: *Q_def SQ_def*)

lemma *SQ_sub_Q*: *SQ* ⊆ *Q*
  by (*auto simp add*: *SQ_def Q_def*)

qualified 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*
by (*auto simp add*: *step_eps_def*)

lemma *step_eps_mono*: *step_eps* [] *q q′* ⟹ *step_eps bs q q′*
by (*auto simp*: *step_eps_def split*: *transition.splits*)

qualified 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′*
by (*auto simp add*: *step_eps_sucs_def step_eps_def split*: *transition.splits*)

qualified 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′*}
using *step_eps_sucs_sound* **by** (*auto simp add*: *step_eps_set_def*)

lemma *step_eps_set_mono*: *R* ⊆ *S* ⟹ *step_eps_set bs R* ⊆ *step_eps_set bs S*
by (*auto simp add*: *step_eps_set_def*)

qualified 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*
unfolding *step_eps_closure_def*
apply (*induction q q′ rule*: *rtranclp.induct*) **using** *step_eps_dest* **by** *auto*

qualified 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*
by (*auto simp add*: *step_eps_closure_set_def step_eps_closure_def*)

lemma *step_eps_closure_set_mono*: *R* ⊆ *S* ⟹ *step_eps_closure_set R bs* ⊆ *step_eps_closure_set S bs*
by (*auto simp add*: *step_eps_closure_set_def*)

lemma *step_eps_closure_set_empty*: *step_eps_closure_set* {} *bs* = {}
by (*auto simp add*: *step_eps_closure_set_def*)

lemma *step_eps_closure_set_mono′*: *step_eps_closure_set R* [] ⊆ *step_eps_closure_set R bs*
by (*auto simp*: *step_eps_closure_set_def step_eps_closure_def*) (*metis mono_rtranclp step_eps_mono*)

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*

by (*auto simp add*: *step_eps_closure_set_def*)

lemma *step_eps_closure_set_Un*: *step_eps_closure_set* ($\bigcup x \in X.\ R\ x$) *bs* =
($\bigcup x \in X.$ *step_eps_closure_set* (*R x*) *bs*)
by (*auto simp add*: *step_eps_closure_set_def*)

lemma *step_eps_closure_set_idem*: *step_eps_closure_set* (*step_eps_closure_set R bs*) *bs* =
step_eps_closure_set R bs
unfolding *step_eps_closure_set_def step_eps_closure_def* **by** *auto*

lemma *step_eps_closure_set_flip*:
assumes *step_eps_closure_set R bs* = *R* ∪ *S*
shows *step_eps_closure_set S bs* ⊆ *R* ∪ *S*
using *step_eps_closure_set_idem*[*of R bs*, *unfolded assms*, *unfolded step_eps_closure_set_split*]
by *auto*

lemma *step_eps_closure_set_unfold*: ($\bigwedge q'.$ *step_eps bs q q'* ⟷ *q'* ∈ *X*) ⟹
step_eps_closure_set {*q*} *bs* = {*q*} ∪ *step_eps_closure_set X bs*
unfolding *step_eps_closure_set_def step_eps_closure_def*
using *rtranclp_step*[*of step_eps bs q*]
by (*auto simp add*: *converse_rtranclp_into_rtranclp*)

lemma *step_step_eps_closure*: *step_eps bs q q'* ⟹ *q* ∈ *R* ⟹ *q'* ∈ *step_eps_closure_set R bs*
unfolding *step_eps_closure_set_def step_eps_closure_def* **by** *auto*

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*)
using *rtranclp_closed*
by (*auto simp add*: *step_eps_closure_set_def step_eps_closure_def step_eps_set_sound Let_def*)

lemma *step_eps_closure_empty*: *step_eps_closure bs q q'* ⟹ ($\bigwedge q'.$ ¬*step_eps bs q q'*) ⟹ *q* = *q'*
unfolding *step_eps_closure_def* **by** (*induction q q' rule*: *rtranclp.induct*) *auto*

lemma *step_eps_closure_set_step_id*: ($\bigwedge q\ q'.$ *q* ∈ *R* ⟹ ¬*step_eps bs q q'*) ⟹
step_eps_closure_set R bs = *R*
using *step_eps_closure_empty step_eps_closure_set_refl* **unfolding** *step_eps_closure_set_def* **by** *blast*

qualified definition *step_symb* :: *state* ⇒ *state* ⇒ *bool* **where**
step_symb q q' ⟷ *q* ∈ *SQ* ∧
(*case transs* ! (*q* − *q0*) *of symb_trans p* ⇒ *p* = *q'* | _ ⇒ *False*)

lemma *step_symb_dest*: *step_symb q q'* ⟹ *q* ∈ *SQ*
by (*auto simp add*: *step_symb_def*)

qualified definition *step_symb_sucs* :: *state* ⇒ *state set* **where**
step_symb_sucs q = (*if q* ∈ *SQ then*
(*case transs* ! (*q* − *q0*) *of symb_trans p* ⇒ {*p*} | _ ⇒ {}) *else* {})

lemma *step_symb_sucs_sound*: *q'* ∈ *step_symb_sucs q* ⟷ *step_symb q q'*
by (*auto simp add*: *step_symb_sucs_def step_symb_def split*: *transition.splits*)

qualified definition *step_symb_set* :: *state set* ⇒ *state set* **where**
step_symb_set R = {*q'*. ∃ *q* ∈ *R*. *step_symb q q'*}

lemma *step_symb_set_mono*: *R* ⊆ *S* ⟹ *step_symb_set R* ⊆ *step_symb_set S*
by (*auto simp add*: *step_symb_set_def*)

lemma *step_symb_set_empty*: *step_symb_set* {} = {}
by (*auto simp add*: *step_symb_set_def*)

lemma *step_symb_set_proj*: *step_symb_set R* = *step_symb_set* (*R* ∩ *SQ*)
using *step_symb_dest* **by** (*auto simp add*: *step_symb_set_def*)

lemma *step_symb_set_split*: *step_symb_set* (*R* ∪ *S*) = *step_symb_set R* ∪ *step_symb_set S*
by (*auto simp add*: *step_symb_set_def*)

lemma *step_symb_set_Un*: *step_symb_set* (⋃ *x* ∈ *X*. *R x*) = (⋃ *x* ∈ *X*. *step_symb_set* (*R x*))
by (*auto simp add*: *step_symb_set_def*)

lemma *step_symb_set_code*[*code*]: *step_symb_set R* = ⋃(*step_symb_sucs* ‘ *R*)
using *step_symb_sucs_sound* **by** (*auto simp add*: *step_symb_set_def*)

qualified 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*
unfolding *delta_def step_eps_closure_set_idem* **by** (*rule refl*)

lemma *delta_eps_split*:
assumes *step_eps_closure_set R bs* = *R* ∪ *S*
shows *delta R bs* = *step_symb_set R* ∪ *delta S bs*
unfolding *delta_def assms step_symb_set_split*
using *step_symb_set_mono*[*OF step_eps_closure_set_flip*[*OF assms*], *unfolded step_symb_set_split*]
step_symb_set_mono[*OF step_eps_closure_set_refl*] **by** *auto*

lemma *delta_split*: *delta* (*R* ∪ *S*) *bs* = *delta R bs* ∪ *delta S bs*
by (*auto simp add*: *delta_def step_symb_set_split step_eps_closure_set_split*)

lemma *delta_Un*: *delta* (⋃ *x* ∈ *X*. *R x*) *bs* = (⋃ *x* ∈ *X*. *delta* (*R x*) *bs*)
unfolding *delta_def step_eps_closure_set_Un step_symb_set_Un* **by** *simp*

lemma *delta_step_symb_set_absorb*: *delta R bs* = *delta R bs* ∪ *step_symb_set R*
using *step_eps_closure_set_refl* **by** (*auto simp add*: *delta_def step_symb_set_def*)

lemma *delta_sub_eps_mono*:
assumes *S* ⊆ *step_eps_closure_set R bs*
shows *delta S bs* ⊆ *delta R bs*
unfolding *delta_def*
using *step_symb_set_mono*[*OF step_eps_closure_set_mono*[*OF assms*, *of bs*,
unfolded step_eps_closure_set_idem]] **by** *simp*

qualified 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*)
unfolding *run_def foldl.simps delta_eps_split*[*OF assms*(*1*), *unfolded assms*(*2*)]
by *auto*

lemma *run_empty*: *run* {} *bss* = {}
unfolding *run_def*
by (*induction bss*)
(*auto simp add*: *delta_def step_symb_set_empty step_eps_closure_set_empty*)

lemma *run_Nil*: *run R* [] = *R*
by (*auto simp add*: *run_def*)

lemma *run_Cons*: *run R* (*bs* # *bss*) = *run* (*delta R bs*) *bss*
unfolding *run_def* **by** *simp*

lemma *run_split*: *run* (*R* ∪ *S*) *bss* = *run R bss* ∪ *run S bss*
unfolding *run_def*
by (*induction bss arbitrary*: *R S*) (*auto simp add*: *delta_split*)

lemma *run_Un*: *run* (⋃ *x* ∈ *X*. *R x*) *bss* = (⋃ *x* ∈ *X*. *run* (*R x*) *bss*)
unfolding *run_def*
by (*induction bss arbitrary*: *R*) (*auto simp add*: *delta_Un*)

lemma *run_comp*: *run R* (*bss* @ *css*) = *run* (*run R bss*) *css*
unfolding *run_def* **by** *simp*

qualified 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*
unfolding *accept_eps_def* **using** *step_step_eps_closure* **by** *simp*

lemma *accept_eps_empty*: *accept_eps* {} *bs* ⟷ *False*
by (*auto simp add*: *accept_eps_def step_eps_closure_set_def*)

lemma *accept_eps_split*: *accept_eps* (*R* ∪ *S*) *bs* ⟷ *accept_eps R bs* ∨ *accept_eps S bs*
by (*auto simp add*: *accept_eps_def step_eps_closure_set_split*)

lemma *accept_eps_Un*: *accept_eps* (⋃ *x* ∈ *X*. *R x*) *bs* ⟷ (∃ *x* ∈ *X*. *accept_eps* (*R x*) *bs*)
by (*auto simp add*: *accept_eps_def step_eps_closure_set_def*)

qualified 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*
unfolding *run_accept_eps_def run_empty accept_eps_empty* **by** *simp*

lemma *run_accept_eps_Nil*: *run_accept_eps R* [] *cs* ⟷ *accept_eps R cs*
by (*auto simp add*: *run_accept_eps_def run_Nil*)

lemma *run_accept_eps_Cons*: *run_accept_eps R* (*bs* # *bss*) *cs* ⟷ *run_accept_eps* (*delta R bs*) *bss cs*
by (*auto simp add*: *run_accept_eps_def run_Cons*)

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*
unfolding *run_accept_eps_Cons* **by** *auto*

lemma *run_accept_eps_Nil_eps*: *run_accept_eps* (*step_eps_closure_set R bs*) [] *bs* ⟷ *run_accept_eps R* [] *bs*
unfolding *run_accept_eps_Nil accept_eps_def step_eps_closure_set_idem* **by** (*rule refl*)

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*
unfolding *run_accept_eps_Cons delta_eps* **by** (*rule refl*)

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*
unfolding *Nil run_accept_eps_Nil accept_eps_def assms*(*1*)
using *assms*(*3*) *step_eps_closure_set_refl step_eps_closure_set_flip*[*OF assms*(*1*)] **by** *auto*

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*
unfolding *run_accept_eps_def Cons run_eps_split*[*OF assms*(*1*,*2*)] **by** (*rule refl*)

lemma *run_accept_eps_split*: *run_accept_eps* (*R* ∪ *S*) *bss bs* ⟷
run_accept_eps R bss bs ∨ *run_accept_eps S bss bs*
unfolding *run_accept_eps_def run_split accept_eps_split* **by** *auto*

lemma *run_accept_eps_Un*: *run_accept_eps* (⋃ *x* ∈ *X*. *R x*) *bss bs* ⟷
(∃ *x* ∈ *X*. *run_accept_eps* (*R x*) *bss bs*)
unfolding *run_accept_eps_def run_Un accept_eps_Un* **by** *simp*

qualified 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'*
by (*auto simp add*: *step_eps_sucs'_def step_eps'_def split*: *transition.splits*)

qualified 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'*}
using *step_eps_sucs'_sound* **by** (*auto simp add*: *step_eps_set'_def*)

qualified definition *step_eps_closure_set'* :: *state set ⇒ bool iarray ⇒ state set* **where**
step_eps_closure_set' R bs = ⋃((λ*q*. {*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*)
using *rtranclp_closed*
by (*auto simp add*: *step_eps_closure_set'_def step_eps_closure'_def step_eps_set'_sound Let_def*)

qualified 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*
using *len_def*
by (*auto simp*: *NFA.SQ_def*)

lemma *step_eps′_eq*: *bs′* = *IArray bs* ⟹ *step_eps bs q q′* ⟷ *step_eps′ bs′ q q′*
by (*auto simp*: *NFA.step_eps_def NFA.step_eps′_def q_in_SQ transs_eq split*: *transition.splits*)

lemma *step_eps_closure′_eq*: *bs′* = *IArray bs* ⟹ *step_eps_closure bs q q′* ⟷ *step_eps_closure′ bs′ q q′*
proof −
assume *lassm*: *bs′* = *IArray bs*
have *step_eps_eq_folded*: *step_eps bs* = *step_eps′ bs′*
using *step_eps′_eq*[*OF lassm*]
by *auto*
show *?thesis*
by (*auto simp*: *NFA.step_eps_closure_def NFA.step_eps_closure′_def step_eps_eq_folded*)
qed

lemma *step_eps_sucs′_eq*: *bs′* = *IArray bs* ⟹ *step_eps_sucs bs q* = *step_eps_sucs′ bs′ q*
by (*auto simp*: *NFA.step_eps_sucs_def NFA.step_eps_sucs′_def q_in_SQ transs_eq*
split: *transition.splits*)

lemma *step_eps_set′_eq*: *bs′* = *IArray bs* ⟹ *step_eps_set bs R* = *step_eps_set′ bs′ R*
by (*auto simp*: *NFA.step_eps_set_def NFA.step_eps_set′_def step_eps_sucs′_eq*)

lemma *step_eps_closure_set′_eq*: *bs′* = *IArray bs* ⟹ *step_eps_closure_set R bs* = *step_eps_closure_set′ R bs′*
by (*auto simp*: *NFA.step_eps_closure_set_def NFA.step_eps_closure_set′_def step_eps_closure′_eq*)

lemma *step_symb_sucs′_eq*: *bs′* = *IArray bs* ⟹ *step_symb_sucs R* = *step_symb_sucs′ R*
by (*auto simp*: *NFA.step_symb_sucs_def NFA.step_symb_sucs′_def q_in_SQ transs_eq*
split: *transition.splits*)

lemma *step_symb_set'_eq*: $bs' = IArray\ bs \Longrightarrow step\_symb\_set\ R = step\_symb\_set'\ R$
by (*auto simp*: *step_symb_set_code NFA.step_symb_set'_def step_symb_sucs'_eq*)

lemma *delta'_eq*: $bs' = IArray\ bs \Longrightarrow delta\ R\ bs = delta'\ R\ bs'$
by (*auto simp*: *NFA.delta_def NFA.delta'_def step_eps_closure_set'_eq step_symb_set'_eq*)

lemma *accept_eps'_eq*: $bs' = IArray\ bs \Longrightarrow accept\_eps\ R\ bs = accept\_eps'\ R\ bs'$
by (*auto simp*: *NFA.accept_eps_def NFA.accept_eps'_def step_eps_closure_set'_eq*)

lemma *accept'_eq*: $accept\ R = accept'\ R$
by (*auto simp*: *NFA.accept_def NFA.accept'_def accept_eps'_eq iarray_of_list_def*)

lemma *run'_eq*: $bss' = map\ IArray\ bss \Longrightarrow run\ R\ bss = run'\ R\ bss'$
by (*induction bss arbitrary*: *R bss'*) (*auto simp*: *NFA.run_def NFA.run'_def delta'_eq*)

lemma *run_accept_eps'_eq*: $bss' = map\ IArray\ bss \Longrightarrow bs' = IArray\ bs \Longrightarrow$
$run\_accept\_eps\ R\ bss\ bs \longleftrightarrow run\_accept\_eps'\ R\ bss'\ bs'$
by (*auto simp*: *NFA.run_accept_eps_def NFA.run_accept_eps'_def accept_eps'_eq run'_eq*)

lemma *run_accept'_eq*: $bss' = map\ IArray\ bss \Longrightarrow$
$run\_accept\ R\ bss \longleftrightarrow run\_accept'\ R\ bss'$
by (*auto simp*: *NFA.run_accept_def NFA.run_accept'_def run'_eq accept'_eq*)

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*: $Q - \{qf\} = SQ$
using *qf_not_in_SQ* **by** (*auto simp add*: *NFA.Q_def*)

lemma *q0_sub_SQ*: $\{q0\} \subseteq SQ$
using *transs_not_Nil* **by** (*auto simp add*: *NFA.SQ_def*)

lemma *q0_sub_Q*: $\{q0\} \subseteq Q$
using *q0_sub_SQ SQ_sub_Q* **by** *auto*

lemma *step_eps_closed*: *step_eps bs q q'* $\Longrightarrow$ *q'* $\in$ *Q*
using *transs_q_in_set state_closed*
by (*fastforce simp add*: *NFA.step_eps_def split*: *transition.splits*)

lemma *step_eps_set_closed*: *step_eps_set bs R* $\subseteq$ *Q*
using *step_eps_closed* **by** (*auto simp add*: *step_eps_set_sound*)

lemma *step_eps_closure_closed*: *step_eps_closure bs q q'* $\Longrightarrow$ *q* $\neq$ *q'* $\Longrightarrow$ *q'* $\in$ *Q*
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*) **using** *step_eps_closed* **by** *auto*

lemma *step_eps_closure_set_closed_union*: *step_eps_closure_set R bs* $\subseteq$ *R* $\cup$ *Q*
using *step_eps_closure_closed* **by** (*auto simp add*: *NFA.step_eps_closure_set_def NFA.step_eps_closure_def*)

lemma *step_eps_closure_set_closed*: *R* $\subseteq$ *Q* $\Longrightarrow$ *step_eps_closure_set R bs* $\subseteq$ *Q*
using *step_eps_closure_set_closed_union* **by** *auto*

lemma *step_symb_closed*: *step_symb q q'* $\Longrightarrow$ *q'* $\in$ *Q*
using *transs_q_in_set state_closed*
by (*fastforce simp add*: *NFA.step_symb_def split*: *transition.splits*)

lemma *step_symb_set_closed*: *step_symb_set R* $\subseteq$ *Q*
using *step_symb_closed* **by** (*auto simp add*: *NFA.step_symb_set_def*)

lemma *step_symb_set_qf*: *step_symb_set* {*qf*} = {}
using *qf_not_in_SQ step_symb_set_proj*[*of* _ _ {*qf*}] *step_symb_set_empty* **by** *auto*

lemma *delta_closed*: *delta R bs* $\subseteq$ *Q*
using *step_symb_set_closed* **by** (*auto simp add*: *NFA.delta_def*)

lemma *run_closed_Cons*: *run R* (*bs* # *bss*) $\subseteq$ *Q*
unfolding *NFA.run_def*
using *delta_closed* **by** (*induction bss arbitrary*: *R bs*) *auto*

lemma *run_closed*: *R* $\subseteq$ *Q* $\Longrightarrow$ *run R bss* $\subseteq$ *Q*
using *run_Nil run_closed_Cons* **by** (*cases bss*) *auto*

lemma *step_eps_qf*: *step_eps bs qf q* $\longleftrightarrow$ *False*
using *qf_not_in_SQ step_eps_dest* **by** *force*

lemma *step_symb_qf*: *step_symb qf q* $\longleftrightarrow$ *False*
using *qf_not_in_SQ step_symb_dest* **by** *force*

lemma *step_eps_closure_qf*: *step_eps_closure bs q q'* $\Longrightarrow$ *q* = *qf* $\Longrightarrow$ *q* = *q'*
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*) **using** *step_eps_qf* **by** *auto*

lemma *step_eps_closure_set_qf*: *step_eps_closure_set* {*qf*} *bs* = {*qf*}
using *step_eps_closure_qf* **unfolding** *NFA.step_eps_closure_set_def NFA.step_eps_closure_def* **by** *auto*

lemma *delta_qf*: *delta* {*qf*} *bs* = {}
using *step_eps_closure_qf step_symb_qf*
by (*auto simp add*: *NFA.delta_def NFA.step_symb_set_def NFA.step_eps_closure_set_def*)

lemma *run_qf_many*: *run* {*qf*} (*bs* # *bss*) = {}
unfolding *run_Cons delta_qf run_empty* **by** (*rule refl*)

lemma *run_accept_eps_qf_many*: *run_accept_eps* {*qf*} (*bs* # *bss*) *cs* ⟷ *False*
unfolding *NFA.run_accept_eps_def* **using** *run_qf_many accept_eps_empty* **by** *simp*

lemma *run_accept_eps_qf_one*: *run_accept_eps* {*qf*} [] *bs* ⟷ *True*
unfolding *NFA.run_accept_eps_def NFA.accept_eps_def* **using** *run_Nil step_eps_closure_set_refl*
by *force*

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* ⊆ *SQ* **and**
qf_eq: *qf* = *qf'* **and**
transs_eq: ⋀*q*. *q* ∈ *nfa'.SQ* ⟹ *transs* ! (*q* − *q0*) = *transs'* ! (*q* − *q0'*)
begin

lemma *q_Q_SQ_nfa'_SQ*: *q* ∈ *nfa'.Q* ⟹ *q* ∈ *SQ* ⟷ *q* ∈ *nfa'.SQ*
using *SQ_sub qf_not_in_SQ qf_eq* **by** (*auto simp add*: *NFA.Q_def*)

lemma *step_eps_cong*: *q* ∈ *nfa'.Q* ⟹ *step_eps bs q q'* ⟷ *nfa'.step_eps bs q q'*
using *q_Q_SQ_nfa'_SQ transs_eq* **by** (*auto simp add*: *NFA.step_eps_def*)

lemma *eps_nfa'_step_eps_closure*: *step_eps_closure bs q q'* ⟹ *q* ∈ *nfa'.Q* ⟹
q' ∈ *nfa'.Q* ∧ *nfa'.step_eps_closure bs q q'*
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*)
using *nfa'.step_eps_closure_closed step_eps_cong* **by** (*auto simp add*: *NFA.step_eps_closure_def*)

lemma *nfa'_eps_step_eps_closure*: *nfa'.step_eps_closure bs q q'* ⟹ *q* ∈ *nfa'.Q* ⟹
q' ∈ *nfa'.Q* ∧ *step_eps_closure bs q q'*
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*)
using *nfa'.step_eps_closed step_eps_cong*
by (*auto simp add*: *NFA.step_eps_closure_def intro*: *rtranclp.intros*(*2*))

lemma *step_eps_closure_set_cong*: *R* ⊆ *nfa'.Q* ⟹ *step_eps_closure_set R bs* =
nfa'.step_eps_closure_set R bs
using *eps_nfa'_step_eps_closure nfa'_eps_step_eps_closure*
by (*fastforce simp add*: *NFA.step_eps_closure_set_def*)

lemma *step_symb_cong*: *q* ∈ *nfa'.Q* ⟹ *step_symb q q'* ⟷ *nfa'.step_symb q q'*
using *q_Q_SQ_nfa'_SQ transs_eq* **by** (*auto simp add*: *NFA.step_symb_def*)

lemma *step_symb_set_cong*: *R* ⊆ *nfa'.Q* ⟹ *step_symb_set R* = *nfa'.step_symb_set R*
using *step_symb_cong* **by** (*auto simp add*: *NFA.step_symb_set_def*)

lemma *delta_cong*: *R* ⊆ *nfa'.Q* ⟹ *delta R bs* = *nfa'.delta R bs*
using *step_symb_set_cong nfa'.step_eps_closure_set_closed*
by (*auto simp add*: *NFA.delta_def step_eps_closure_set_cong*)

lemma *run_cong*: *R* ⊆ *nfa'.Q* ⟹ *run R bss* = *nfa'.run R bss*
unfolding *NFA.run_def*
using *nfa'.delta_closed delta_cong* **by** (*induction bss arbitrary*: *R*) *auto*

lemma *accept_eps_cong*: *R* ⊆ *nfa'.Q* ⟹ *accept_eps R bs* ⟷ *nfa'.accept_eps R bs*

unfolding *NFA.accept_eps_def* **using** *step_eps_closure_set_cong qf_eq* **by** *auto*

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*
unfolding *NFA.run_accept_eps_def run_cong*[*OF assms*]
accept_eps_cong[*OF nfa'.run_closed*[*OF assms*]] **by** *simp*

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*)
by (*induction xs*) *auto*

lemma *list_split_def*: *list_split xs* = ($\bigcup n <$ *length xs*. {(*take n xs*, *drop n xs*)})
using *less_Suc_eq_0_disj* **by** (*induction xs rule*: *list_split.induct*) *auto*+

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*
unfolding *NFA.Q_def* **using** *SQ_sub qf'_in_SQ* **by** *auto*

lemma *q_SQ_SQ_nfa'_SQ*: $q \in nfa'.SQ \Longrightarrow q \in SQ \longleftrightarrow q \in nfa'.SQ$
using *SQ_sub* **by** *auto*

lemma *step_eps_cong_SQ*: $q \in nfa'.SQ \Longrightarrow$ *step_eps bs q q'* $\longleftrightarrow$ *nfa'.step_eps bs q q'*
using *q_SQ_SQ_nfa'_SQ transs_eq* **by** (*auto simp add*: *NFA.step_eps_def*)

lemma *step_eps_cong_Q*: $q \in nfa'.Q \Longrightarrow$ *nfa'.step_eps bs q q'* $\Longrightarrow$ *step_eps bs q q'*
using *SQ_sub transs_eq* **by** (*auto simp add*: *NFA.step_eps_def*)

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'
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*)
using *NFA.Q_def NFA.step_eps_closure_def*
by (*auto simp add*: *rtranclp.rtrancl_into_rtrancl step_eps_cong_SQ step_eps_dest*)

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
unfolding *NFA.step_eps_closure_set_def*
using *nfa'_step_eps_closure_cong* **by** *fastforce*

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'*)
unfolding *NFA.step_eps_closure_def*
apply (*induction q q' rule*: *rtranclp.induct*)
using *nfa'.step_eps_closure_closed nfa'.step_eps_closed step_eps_cong_SQ NFA.Q_def*

by (*auto simp add*: *intro*: *rtranclp.rtrancl_into_rtrancl*) *fastforce+*

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′*
unfolding *NFA.step_eps_closure_def*
apply (*induction q q′ rule*: *rtranclp.induct*)
using *nfa′.step_eps_closed step_eps_cong_Q*
by (*auto intro*: *rtranclp.intros*(*2*))

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*
using *eps_nfa′_step_eps_closure_cong nfa′_eps_step_eps_closure_cong*
rtranclp_trans[*of step_eps bs*]
unfolding *NFA.step_eps_closure_set_def NFA.step_eps_closure_def*
by *auto blast+*

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*
using *eps_nfa′_step_eps_closure_cong nfa′_eps_step_eps_closure_cong*
unfolding *NFA.step_eps_closure_set_def NFA.step_eps_closure_def*
by *auto blast+*

lemma *step_symb_cong_SQ*: *q* $\in$ *nfa′.SQ* $\Longrightarrow$ *step_symb q q′* $\longleftrightarrow$ *nfa′.step_symb q q′*
using *q_SQ_SQ_nfa′_SQ transs_eq* **by** (*auto simp add*: *NFA.step_symb_def*)

lemma *step_symb_cong_Q*: *nfa′.step_symb q q′* $\Longrightarrow$ *step_symb q q′*
using *SQ_sub transs_eq* **by** (*auto simp add*: *NFA.step_symb_def*)

lemma *step_symb_set_cong_SQ*: *R* $\subseteq$ *nfa′.SQ* $\Longrightarrow$ *step_symb_set R* = *nfa′.step_symb_set R*
using *step_symb_cong_SQ* **by** (*auto simp add*: *NFA.step_symb_set_def*)

lemma *step_symb_set_cong_Q*: *nfa′.step_symb_set R* $\subseteq$ *step_symb_set R*
using *step_symb_cong_Q* **by** (*auto simp add*: *NFA.step_symb_set_def*)

lemma *delta_cong_unreach*:
assumes *R* $\subseteq$ *nfa′.Q* $\neg$*nfa′.accept_eps R bs*
shows *delta R bs* = *nfa′.delta R bs*
proof −
have *nfa′.step_eps_closure_set R bs* $\subseteq$ *nfa′.SQ*
using *nfa′.step_eps_closure_set_closed*[*OF assms*(*1*), *unfolded NFA.Q_def*]
assms(*2*)[*unfolded NFA.accept_eps_def*] **by** *auto*
then show *?thesis*
unfolding *NFA.accept_eps_def NFA.delta_def* **using** *step_symb_set_cong_SQ*
step_eps_closure_set_cong_unreach[*OF assms*(*1*) *assms*(*2*)[*unfolded NFA.accept_eps_def*]]
by *auto*
qed

lemma *nfa′_delta_sub_delta*:
assumes *R* $\subseteq$ *nfa′.Q*
shows *nfa′.delta R bs* $\subseteq$ *delta R bs*
unfolding *NFA.delta_def*
using *step_symb_set_mono*[*OF nfa′_step_eps_closure_set_sub*[*OF assms*]] *step_symb_set_cong_Q*
by *fastforce*

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 (*rule set_eqI*, *rule iffI*)

```
  fix q
  assume assm: q ∈ delta R bs
  have nfa'_eps_diff_Un: nfa'.step_eps_closure_set R bs =
    nfa'.step_eps_closure_set R bs − {qf'} ∪ {qf'}
    using assms(2)[unfolded NFA.accept_eps_def] by auto
  from assm have q ∈ step_symb_set (nfa'.step_eps_closure_set R bs − {qf'}) ∪
    step_symb_set {qf'} ∪ delta {qf'} bs
    unfolding NFA.delta_def step_eps_closure_set_cong_reach[OF assms(1)
      assms(2)[unfolded NFA.accept_eps_def]] step_symb_set_split[symmetric]
      nfa'_eps_diff_Un[symmetric] by simp
  then have q ∈ step_symb_set (nfa'.step_eps_closure_set R bs − {qf'}) ∪ delta {qf'} bs
    using step_symb_set_mono[of {qf'} step_eps_closure_set {qf'} bs,
      OF step_eps_closure_set_refl, unfolded NFA.delta_def[symmetric]]
    delta_step_symb_set_absorb by blast
  then show q ∈ nfa'.delta R bs ∪ delta {qf'} bs
    unfolding NFA.delta_def
    using nfa'.step_eps_closure_set_closed[OF assms(1), unfolded NFA.Q_def]
      step_symb_set_cong_SQ[of nfa'.step_eps_closure_set R bs − {qf'}]
      step_symb_set_mono by blast
next
  fix q
  assume q ∈ nfa'.delta R bs ∪ delta {qf'} bs
  then show q ∈ delta R bs
    using nfa'_delta_sub_delta[OF assms(1)] delta_sub_eps_mono[of {qf'} _ _ R bs]
      assms(2)[unfolded NFA.accept_eps_def] nfa'_step_eps_closure_set_sub[OF assms(1)]
    by fastforce
qed

lemma run_cong:
  assumes R ⊆ nfa'.Q
  shows run R bss = nfa'.run R bss ∪ (⋃(css, css') ∈ list_split bss.
    if nfa'.run_accept_eps R css (hd css') then run {qf'} css' else {})
  using assms
proof (induction bss arbitrary: R rule: list_split.induct)
  case 1
  then show ?case
    using run_Nil by simp
next
  case (2 x xs)
  show ?case
    apply (cases nfa'.accept_eps R x)
    unfolding run_Cons delta_cong_reach[OF 2(2)]
      delta_cong_unreach[OF 2(2)] run_split run_accept_eps_Nil run_accept_eps_Cons
      list_split_unfold[of λys zs. if nfa'.run_accept_eps R ys (hd zs)
      then run {qf'} zs else {} x xs] using 2(1)[of nfa'.delta R x,
    OF nfa'.delta_closed, unfolded run_accept_eps_Nil] by auto
qed

lemma run_cong_Cons_sub:
  assumes R ⊆ nfa'.Q delta {qf'} bs ⊆ nfa'.delta R bs
  shows run R (bs # bss) = nfa'.run R (bs # bss) ∪
    (⋃(css, css') ∈ list_split bss.
    if nfa'.run_accept_eps (nfa'.delta R bs) css (hd css') then run {qf'} css' else {})
  unfolding run_Cons using run_cong[OF nfa'.delta_closed]
    delta_cong_reach[OF assms(1)] delta_cong_unreach[OF assms(1)]
  by (cases nfa'.accept_eps R bs) (auto simp add: Un_absorb2[OF assms(2)])

lemma accept_eps_nfa'_run:
```

assumes $R \subseteq nfa'.Q$
shows $accept\_eps\ (nfa'.run\ R\ bss)\ bs \longleftrightarrow$
$nfa'.accept\_eps\ (nfa'.run\ R\ bss)\ bs \wedge accept\_eps\ (run\ \{qf'\}\ []))\ bs$
unfolding *NFA.accept_eps_def run_Nil*
using *step_eps_closure_set_cong_reach*[*OF nfa'.run_closed*[*OF assms*]]
step_eps_closure_set_cong_unreach[*OF nfa'.run_closed*[*OF assms*]] *qf_not_in_SQ*
qf'_in_SQ nfa'.step_eps_closure_set_closed[*OF nfa'.run_closed*[*OF assms*],
unfolded NFA.Q_def] *SQ_sub*
by (*cases qf' ∈ nfa'.step_eps_closure_set* (*nfa'.run R bss*) *bs*) *fastforce*+

lemma *run_accept_eps_cong*:
assumes $R \subseteq nfa'.Q$
shows $run\_accept\_eps\ R\ bss\ bs \longleftrightarrow (nfa'.run\_accept\_eps\ R\ bss\ bs \wedge run\_accept\_eps\ \{qf'\}\ []\ bs) \vee$
$(\exists (css, css') \in list\_split\ bss.\ nfa'.run\_accept\_eps\ R\ css\ (hd\ css') \wedge$
$run\_accept\_eps\ \{qf'\}\ css'\ bs)$
unfolding *NFA.run_accept_eps_def run_cong*[*OF assms*] *accept_eps_split*
accept_eps_Un accept_eps_nfa'_run[*OF assms*]
using *accept_eps_empty* **by** (*auto split*: *if_splits*)+

lemma *run_accept_eps_cong_Cons_sub*:
assumes $R \subseteq nfa'.Q\ delta\ \{qf'\}\ bs \subseteq nfa'.delta\ R\ bs$
shows $run\_accept\_eps\ R\ (bs\ \#\ bss)\ cs \longleftrightarrow$
$(nfa'.run\_accept\_eps\ R\ (bs\ \#\ bss)\ cs \wedge run\_accept\_eps\ \{qf'\}\ []\ cs) \vee$
$(\exists (css, css') \in list\_split\ bss.\ nfa'.run\_accept\_eps\ (nfa'.delta\ R\ bs)\ css\ (hd\ css') \wedge$
$run\_accept\_eps\ \{qf'\}\ css'\ cs)$
unfolding *NFA.run_accept_eps_def run_cong_Cons_sub*[*OF assms*]
accept_eps_split accept_eps_Un accept_eps_nfa'_run[*OF assms*(*1*)]
using *accept_eps_empty* **by** (*auto split*: *if_splits*)+

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*: $step\_symb\_set\ \{q0\} = \{\}$
unfolding *NFA.step_symb_set_def* **using** *step_symb_q0* **by** *simp*

lemma *qf_not_q0*: $qf \notin \{q0\}$
using *qf_not_in_SQ q0_sub_SQ* **by** *auto*

lemma *step_eps_closure_set_q0*: $step\_eps\_closure\_set\ \{q0\}\ bs = \{q0\} \cup$
$(nfa'.step\_eps\_closure\_set\ \{q0'\}\ bs \cup right.nfa'.step\_eps\_closure\_set\ \{q0''\}\ bs)$
using *step_eps_closure_set_unfold*[*OF step_eps_q0*]
insert_is_Un[*of q0' {q0''}*]
step_eps_closure_set_split[*of _ _ {q0'} {q0''}*]
step_eps_closure_set_cong[*OF nfa'.q0_sub_Q*]
right.step_eps_closure_set_cong[*OF right.nfa'.q0_sub_Q*]
by *auto*

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*)
using *run_accept_eps_Nil_cong run_accept_eps_Cons_cong* **by** (*cases bss*) *auto*

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*} = {}
unfolding *NFA.step_symb_set_def* **using** *step_symb_q0* **by** *simp*

lemma *run_accept_eps_Nil*: *run_accept_eps* {*q0*} [] *bs*
unfolding *NFA.run_accept_eps_def NFA.run_def* **using** *step_eps_accept_eps step_eps_q0* **by** *fastforce*

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'*)
apply (*induction q q' rule*: *rtranclp.induct*)
using *step_eps_q0 step_eps_dest qf_not_in_SQ step_eps_cong_SQ nfa'.q0_sub_SQ*
nfa'.step_eps_closed[*unfolded NFA.Q_def*] **by** *fastforce*+

lemma *step_eps_closure_set_q0*: *step_eps_closure_set* {*q0*} *bs* ⊆ {*q0*, *qf*} ∪
(*nfa'.step_eps_closure_set* {*q0'*} *bs* ∩ *nfa'.SQ*)
unfolding *NFA.step_eps_closure_set_def NFA.step_eps_closure_def*
using *rtranclp_step_eps_q0_q0'* **by** *auto*

lemma *delta_sub_nfa'_delta*: *delta* {*q0*} *bs* ⊆ *nfa'.delta* {*q0'*} *bs*
unfolding *NFA.delta_def*
using *step_symb_set_mono*[*OF step_eps_closure_set_q0*, *unfolded step_symb_set_q0*
step_symb_set_qf step_symb_set_split insert_is_Un[*of q0* {*qf*}]]

```
    step_symb_set_cong_SQ
  by (metis boolean_algebra_cancel.sup0 inf_le2 step_symb_set_proj step_symb_set_q0
      step_symb_set_qf sup_commute)

lemma step_eps_closure_set_q0_split: step_eps_closure_set {q0} bs = {q0, qf} ∪
  step_eps_closure_set {q0'} bs
  unfolding NFA.step_eps_closure_set_def NFA.step_eps_closure_def
  using step_eps_qf step_eps_q0
  apply (auto)
   apply (metis rtranclp_unfold)
  by (metis r_into_rtranclp rtranclp.rtrancl_into_rtrancl rtranclp_idemp)

lemma delta_q0_q0': delta {q0} bs = delta {q0'} bs
  unfolding NFA.delta_def step_eps_closure_set_q0_split step_symb_set_split
  unfolding NFA.delta_def[symmetric]
  unfolding NFA.step_symb_set_def
  using step_symb_q0 step_symb_dest qf_not_in_SQ
  by fastforce

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])
  apply (induction zs @ [z] arbitrary: zs rule: chain_le.induct)
    apply (auto intro: chain_le.intros)[2]
  subgoal for y xs x zs
    apply (cases zs)
     apply (auto)
    apply (metis append.assoc append_Cons append_Nil chain_le_cons)
    done
  done

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
  by (auto elim: reaches_on.cases)

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 (induction s vs s' arbitrary: i rule: reaches_on.induct)
  case (2 s s' v vs s'')
  show ?case
    using 2(1,2)
  proof (cases i)
    case (Suc n)
    show ?thesis
      using 2
      by (fastforce simp: Suc intro: reaches_on.intros)
  qed (auto intro: reaches_on.intros)
qed auto

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 (induction s vs s' arbitrary: i rule: reaches_on.induct)
  case (2 s s' v vs s'')
  show ?case
    using 2(1,2)
  proof (cases i)
    case (Suc n)
    show ?thesis
      using 2
      by (fastforce simp: Suc intro: reaches_on.intros)
  qed (auto intro: reaches_on.intros)
qed (auto intro: reaches_on.intros)

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'
  using reaches_on_split'[where i=length vs, of run s vs @ vs' s']
  by auto

lemma reaches_on_inj: reaches_on run s vs t ⟹ reaches_on run s vs' t' ⟹
  length vs = length vs' ⟹ vs = vs' ∧ t = t'
  apply (induction s vs t arbitrary: vs' t' rule: reaches_on.induct)
   apply (auto elim: reaches_on.cases)[1]
  subgoal for s s' v vs s'' vs' t'
    apply (rule reaches_on.cases[of run s' vs s'']; rule reaches_on.cases[of run s vs' t'])
          apply assumption+
       apply auto[2]
      apply fastforce
     apply (metis length_0_conv list.discI)
    apply (metis Pair_inject length_Cons nat.inject option.inject)
    done
  done

lemma reaches_on_split_last: reaches_on run s (xs @ [x]) s'' ⟹
  ∃ s'. reaches_on run s xs s' ∧ run s' = Some (s'', x)
  apply (induction s xs @ [x] s'' arbitrary: xs x rule: reaches_on.induct)
   apply simp
  subgoal for s s' v vs s'' xs x
    by (cases vs rule: rev_cases) (fastforce elim: reaches_on.cases intro: reaches_on.intros)+
  done

lemma reaches_on_rev_induct[consumes 1]: reaches_on run s vs s' ⟹
  (⋀s. P s [] s) ⟹
  (⋀s s' v vs s''. reaches_on run s vs s' ⟹ P s vs s' ⟹ run s' = Some (s'', v) ⟹
```

```
    P s (vs @ [v]) s'') ⟹
  P s vs s'
proof (induction vs arbitrary: s s' rule: rev_induct)
  case (snoc x xs)
  from snoc(2) obtain s'' where s''_def: reaches_on run s xs s'' run s'' = Some (s', x)
    using reaches_on_split_last
    by fast
  show ?case
    using snoc(4)[OF s''_def(1) _ s''_def(2)] snoc(1)[OF s''_def(1) snoc(3,4)]
    by auto
qed (auto elim: reaches_on.cases)

lemma reaches_on_app: reaches_on run s vs s' ⟹ run s' = Some (s'', v) ⟹
  reaches_on run s (vs @ [v]) s''
  by (induction s vs s' rule: reaches_on.induct) (auto intro: reaches_on.intros)

lemma reaches_on_trans: reaches_on run s vs s' ⟹ reaches_on run s' vs' s'' ⟹
  reaches_on run s (vs @ vs') s''
  by (induction s vs s' rule: reaches_on.induct) (auto intro: reaches_on.intros)

lemma reaches_onD: reaches_on run s ((t, b) # vs) s' ⟹
  ∃ s''. run s = Some (s'', (t, b)) ∧ reaches_on run s'' vs s'
  by (auto elim: reaches_on.cases)

lemma reaches_on_setD: reaches_on run s vs s' ⟹ x ∈ set vs ⟹
  ∃ vs' vs'' s''. reaches_on run s (vs' @ [x]) s'' ∧ reaches_on run s'' vs'' s' ∧ vs = vs' @ x # vs''
proof (induction s vs s' rule: reaches_on_rev_induct)
  case (2 s s' v vs s'')
  show ?case
  proof (cases x ∈ set vs)
    case True
    obtain vs' vs'' s''' where split_def: reaches_on run s (vs' @ [x]) s'''
      reaches_on run s''' vs'' s' vs = vs' @ x # vs''
      using 2(3)[OF True]
      by auto
    show ?thesis
      using split_def(1,3) reaches_on_app[OF split_def(2) 2(2)]
      by auto
  next
    case False
    have x_v: x = v
      using 2(4) False
      by auto
    show ?thesis
      unfolding x_v
      using reaches_on_app[OF 2(1,2)] reaches_on.intros(1)[of run s'']
      by auto
  qed
qed auto

lemma reaches_on_len: ∃ vs s'. reaches_on run s vs s' ∧ (length vs = n ∨ run s' = None)
proof (induction n arbitrary: s)
  case (Suc n)
  show ?case
  proof (cases run s)
    case (Some x)
    obtain s' v where x_def: x = (s', v)
      by (cases x) auto
```

obtain *vs s''* **where** *s''_def*: *reaches_on run s' vs s'' length vs = n ∨ run s'' = None*
using *Suc*[*of s'*]
by *auto*
show *?thesis*
using *reaches_on.intros*(*2*)[*OF Some*[*unfolded x_def*] *s''_def*(*1*)] *s''_def*(*2*)
by *fastforce*
qed (*auto intro*: *reaches_on.intros*)
qed (*auto intro*: *reaches_on.intros*)

lemma *reaches_on_NilD*: *reaches_on run q* [] *q' ⟹ q = q'*
by (*auto elim*: *reaches_on.cases*)

lemma *reaches_on_ConsD*: *reaches_on run q* (*x # xs*) *q' ⟹* ∃ *q''*. *run q = Some* (*q''*, *x*) ∧ *reaches_on run q'' xs q'*
by (*auto elim*: *reaches_on.cases*)

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*) *⟹ reaches run s' n s'' ⟹ reaches run s* (*Suc n*) *s''*

lemma *reaches_Suc_split_last*: *reaches run s* (*Suc n*) *s' ⟹* ∃ *s''* *x*. *reaches run s n s'' ∧ run s'' = Some* (*s'*, *x*)
proof (*induction n arbitrary*: *s*)
case (*Suc n*)
obtain *s''* *x* **where** *s''_def*: *run s = Some* (*s''*, *x*) *reaches run s''* (*Suc n*) *s'*
using *Suc*(*2*)
by (*auto elim*: *reaches.cases*)
show *?case*
using *s''_def*(*1*) *Suc*(*1*)[*OF s''_def*(*2*)]
by (*auto intro*: *reaches.intros*)
qed (*auto elim*!: *reaches.cases intro*: *reaches.intros*)

lemma *reaches_invar*: *reaches f x n y ⟹ P x ⟹* (⋀*z z' v*. *P z ⟹ f z = Some* (*z'*, *v*) *⟹ P z'*) *⟹ P y*
by (*induction x n y rule*: *reaches.induct*) *auto*

lemma *reaches_cong*: *reaches f x n y ⟹ P x ⟹* (⋀*z z' v*. *P z ⟹ f z = Some* (*z'*, *v*) *⟹ P z'*) *⟹* (⋀*z*. *P z ⟹ f'* (*g z*) = *map_option* (*apfst g*) (*f z*)) *⟹ reaches f'* (*g x*) *n* (*g y*)
by (*induction x n y rule*: *reaches.induct*) (*auto intro*: *reaches.intros*)

lemma *reaches_on_n*: *reaches_on run s vs s' ⟹ reaches run s* (*length vs*) *s'*
by (*induction s vs s' rule*: *reaches_on.induct*) (*auto intro*: *reaches.intros*)

lemma *reaches_on*: *reaches run s n s' ⟹* ∃ *vs*. *reaches_on run s vs s' ∧ length vs = n*
by (*induction s n s' rule*: *reaches.induct*) (*auto intro*: *reaches_on.intros*)

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

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

fun *sub_bs* :: (*'d × 'b*) *list ⇒ nat × nat ⇒ 'b list* **where**
sub_bs rho (*i*, *j*) = *map* (*bs_at rho*) [*i*..<*j*]

definition *steps* :: (*'c ⇒ 'b ⇒ 'c*) *⇒* (*'d × 'b*) *list ⇒ 'c ⇒ nat × nat ⇒ 'c* **where**
steps step rho q ij = foldl step q (*sub_bs rho ij*)

definition *acc* :: ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*c* ⇒ *bool*) ⇒ ('*d* × '*b*) *list* ⇒
'*c* ⇒ *nat* × *nat* ⇒ *bool* **where**
acc step accept rho q ij = *accept* (*steps step rho q ij*)

definition *sup_acc* :: ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*c* ⇒ *bool*) ⇒ ('*d* × '*b*) *list* ⇒
'*c* ⇒ *nat* ⇒ *nat* ⇒ ('*d* × *nat*) *option* **where**
sup_acc step accept rho q i j =
(*let L'* = {*l* ∈ {*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* ⇒ ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*d* × '*b*) *list* ⇒
nat ⇒ *nat* ⇒ '*c* ⇒ '*d option* **where**
sup_leadsto init step rho i j q =
(*let L'* = {*l*. *l* < *i* ∧ *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* ⇒ '*a set* **where**
mmap_keys kvs = *set* (*map fst kvs*)

definition *mmap_lookup* :: ('*a*, '*b*) *mmap* ⇒ '*a* ⇒ '*b option* **where**
mmap_lookup = *map_of*

definition *valid_s* :: '*c* ⇒ ('*c* ⇒ '*b* ⇒ '*c*) ⇒ ('*c* × '*b*, '*c*) *mapping* ⇒ ('*c* ⇒ *bool*) ⇒
('*d* × '*b*) *list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ ('*c*, '*c* × ('*d* × *nat*) *option*) *mmap* ⇒ *bool* **where**
valid_s init step st accept rho u i j s ≡
(∀ *q bs*. *case Mapping.lookup st* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*) ∧
(*mmap_keys s* = {*q*. (∃ *l* ≤ *u*. *steps step rho init* (*l*, *i*) = *q*)} ∧ *distinct* (*map fst s*) ∧
(∀ *q*. *case mmap_lookup s q of None* ⇒ *True*
| *Some* (*q'*, *tstp*) ⇒ *steps step rho q* (*i*, *j*) = *q'* ∧ *tstp* = *sup_acc step accept rho q i j*))

record ('*b*, '*c*, '*d*, '*t*, '*e*) *args* =
w_init :: '*c*
w_step :: '*c* ⇒ '*b* ⇒ '*c*
w_accept :: '*c* ⇒ *bool*
w_run_t :: '*t* ⇒ ('*t* × '*d*) *option*
w_read_t :: '*t* ⇒ '*d option*
w_run_sub :: '*e* ⇒ ('*e* × '*b*) *option*

record ('*b*, '*c*, '*d*, '*t*, '*e*) *window* =
w_st :: ('*c* × '*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* × ('*d* × *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* ⇒ '*t* ⇒ '*e* ⇒
('*d* × '*b*) *list* ⇒ *nat* × '*t* × '*e* × *nat* × '*t* × '*e* ⇒ *bool* **where**
reach_window args t0 sub rho (*i*, *ti*, *si*, *j*, *tj*, *sj*) ⟷ *i* ≤ *j* ∧ *length rho* = *j* ∧
reaches_on (*w_run_t args*) *t0* (*take i* (*map fst rho*)) *ti* ∧
reaches_on (*w_run_t args*) *ti* (*drop i* (*map fst rho*)) *tj* ∧

```
    reaches_on (w_run_sub args) sub (take i (map snd rho)) si ∧
    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 ⟹
  reaches_on (w_run_sub args) sub (take i (map snd rho)) si ⟹
  reaches_on (w_run_t args) t0 (map fst rho) tj ⟹
  reaches_on (w_run_sub args) sub (map snd rho) sj ⟹
  i ≤ length rho ⟹ length rho = j ⟹
  reach_window args t0 sub rho (i, ti, si, j, tj, sj)
  by auto (metis reaches_on_split'[of _ _ _ _ i] length_map reaches_on_inj)+

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)
  using reaches_on_app[of w_run_t args t0 take i (map fst rho) ti ti' t]
    reaches_on_app[of w_run_sub args sub take i (map snd rho) si si' s] assms
  apply (auto)
     apply (smt append_take_drop_id id_take_nth_drop length_map list.discI list.inject
      option.inject reaches_on.cases same_append_eq snd_conv take_Suc_conv_app_nth)
    apply (smt Cons_nth_drop_Suc fst_conv length_map list.discI list.inject option.inject
      reaches_on.cases)
   apply (smt append_take_drop_id id_take_nth_drop length_map list.discI list.inject
      option.inject reaches_on.cases same_append_eq snd_conv take_Suc_conv_app_nth)
  apply (smt Cons_nth_drop_Suc fst_conv length_map list.discI list.inject option.inject
      reaches_on.cases)
  done

lemma reach_window_run_ti: reach_window args t0 sub rho (i, ti, si, j, tj, sj) ⟹
  i < j ⟹ ∃ ti'. reaches_on (w_run_t args) t0 (take i (map fst rho)) ti ∧
  w_run_t args ti = Some (ti', ts_at rho i) ∧
  reaches_on (w_run_t args) ti' (drop (Suc i) (map fst rho)) tj
  apply (auto simp: ts_at_def elim!: reaches_on.cases[of w_run_t args ti drop i (map fst rho)])
  using nth_via_drop apply fastforce
  by (metis Cons_nth_drop_Suc length_map list.inject)

lemma reach_window_run_si: reach_window args t0 sub rho (i, ti, si, j, tj, sj) ⟹
  i < j ⟹ ∃ si'. reaches_on (w_run_sub args) sub (take i (map snd rho)) si ∧
  w_run_sub args si = Some (si', bs_at rho i) ∧
  reaches_on (w_run_sub args) si' (drop (Suc i) (map snd rho)) sj
  apply (auto simp: bs_at_def elim!: reaches_on.cases[of w_run_sub args si drop i (map snd rho)])
  using nth_via_drop apply fastforce
  by (metis Cons_nth_drop_Suc length_map list.inject)

lemma reach_window_run_tj: reach_window args t0 sub rho (i, ti, si, j, tj, sj) ⟹
  reaches_on (w_run_t args) t0 (map fst rho) tj
  using reaches_on_trans
  by fastforce

lemma reach_window_run_sj: reach_window args t0 sub rho (i, ti, si, j, tj, sj) ⟹
  reaches_on (w_run_sub args) sub (map snd rho) sj
  using reaches_on_trans
  by fastforce

lemma reach_window_shift_all: reach_window args t0 sub rho (i, si, ti, j, sj, tj) ⟹
  reach_window args t0 sub rho (j, sj, tj, j, sj, tj)
  using reach_window_run_tj[of args t0 sub rho] reach_window_run_sj[of args t0 sub rho]
  by (auto intro: reaches_on.intros)
```

lemma *reach_window_app*: *reach_window args t0 sub rho* (*i*, *si*, *ti*, *j*, *tj*, *sj*) ⟹
w_run_t args tj = *Some* (*tj'*, *x*) ⟹ *w_run_sub args sj* = *Some* (*sj'*, *y*) ⟹
reach_window args t0 sub (*rho* @ [(*x*, *y*)]) (*i*, *si*, *ti*, *Suc j*, *tj'*, *sj'*)
by (*fastforce simp add*: *reaches_on_app*)

fun *init_args* :: ('*c* × ('*c* ⇒ '*b* ⇒ '*c*) × ('*c* ⇒ *bool*)) ⇒
(('*t* ⇒ ('*t* × '*d*) *option*) × ('*t* ⇒ '*d option*)) ⇒
('*e* ⇒ ('*e* × '*b*) *option*) ⇒ ('*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*)
by (*auto simp*: *valid_window_def mmap_keys_def mmap_lookup_def sup_leadsto_def*
valid_s_def steps_def sup_acc_def intro: *reaches_on.intros split*: *option.splits*)

lemma *steps_app_cong*: *j* ≤ *length rho* ⟹ *steps step* (*rho* @ [*x*]) *q* (*i*, *j*) =
steps step rho q (*i*, *j*)
proof −
assume *j* ≤ *length rho*
then have *map_cong*: *map* (*bs_at* (*rho* @ [*x*])) [*i*..<*j*] = *map* (*bs_at rho*) [*i*..<*j*]
by (*auto simp*: *bs_at_def nth_append*)
show *?thesis*
by (*auto simp*: *steps_def map_cong*)
qed

lemma *acc_app_cong*: *j* < *length rho* ⟹ *acc step accept* (*rho* @ [*x*]) *q* (*i*, *j*) =
acc step accept rho q (*i*, *j*)
by (*auto simp*: *acc_def bs_at_def nth_append steps_app_cong*)

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
apply (*auto simp*: *sup_acc_def Let_def ts_at_def nth_append acc_def*)
apply (*metis* (*mono_tags*, *opaque_lifting*) *less_eq_Suc_le order_less_le_trans steps_app_cong*)+
done

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

```
  apply (induction rho' rule: rev_induct)
   apply auto
  apply (smt append.assoc le_add1 le_trans length_append sup_acc_app_cong)
  done

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 −
  assume assms: i ≤ j j ≤ length rho
  define L' where L' = {l. l < i ∧ steps step rho init (l, j) = q}
  define L'' where L'' = {l. l < i ∧ steps step (rho @ [x]) init (l, j) = q}
  show ?thesis
    using assms
    by (cases L' = {})
       (auto simp: sup_leadsto_def L'_def L''_def ts_at_def nth_append steps_app_cong)
qed

lemma chain_le:
  fixes xs :: 'd :: timestamp list
  shows chain_le xs ⟹ i ≤ j ⟹ j < length xs ⟹ xs ! i ≤ xs ! j
proof (induction xs arbitrary: i j rule: chain_le.induct)
  case (chain_le_cons y xs x)
  then show ?case
  proof (cases i)
    case 0
    then show ?thesis
      using chain_le_cons
      apply (cases j)
      apply auto
     apply (metis (no_types, lifting) le_add1 le_add_same_cancel1 le_less less_le_trans nth_Cons_0)
      done
  qed auto
qed auto

lemma steps_refl[simp]: steps step rho q (i, i) = q
  unfolding steps_def by auto

lemma steps_split: i < j ⟹ steps step rho q (i, j) =
  steps step rho (step q (bs_at rho i)) (Suc i, j)
  unfolding steps_def by (simp add: upt_rec)

lemma steps_app: i ≤ j ⟹ steps step rho q (i, j + 1) =
  step (steps step rho q (i, j)) (bs_at rho j)
  unfolding steps_def by auto

lemma steps_appE: i ≤ j ⟹ steps step rho q (i, Suc j) = q' ⟹
  ∃ q''. steps step rho q (i, j) = q'' ∧ q' = step q'' (bs_at rho j)
  unfolding steps_def sub_bs.simps by auto

lemma steps_comp: i ≤ l ⟹ l ≤ j ⟹ steps step rho q (i, l) = q' ⟹
  steps step rho q' (l, j) = q'' ⟹ steps step rho q (i, j) = q''
proof −
  assume assms: i ≤ l l ≤ j steps step rho q (i, l) = q' steps step rho q' (l, j) = q''
  have range_app: [i..<l] @ [l..<j] = [i..<j]
    using assms(1,2)
    by (metis le_Suc_ex upt_add_eq_append)
  have q' = foldl step q (map (bs_at rho) [i..<l])
    using assms(3) unfolding steps_def by auto
```

```
  moreover have q'' = foldl step q' (map (bs_at rho) [l..<j])
    using assms(4) unfolding steps_def by auto
  ultimately have q'' = foldl step q (map (bs_at rho) ([i..<l] @ [l..<j]))
    by auto
  then show ?thesis
    unfolding steps_def range_app by auto
qed

lemma sup_acc_SomeI: acc step accept rho q (i, Suc l) ⟹ l ∈ {i..<j} ⟹
  ∃ tp. sup_acc step accept rho q i j = Some (ts_at rho tp, tp) ∧ l ≤ tp ∧ tp < j
proof −
  assume assms: acc step accept rho q (i, Suc l) l ∈ {i..<j}
  define L where L = {l ∈ {i..<j}. acc step accept rho q (i, Suc l)}
  have L_props: finite L L ≠ {} l ∈ L
    using assms unfolding L_def by auto
  then show ∃ tp. sup_acc step accept rho q i j = Some (ts_at rho tp, tp) ∧ l ≤ tp ∧ tp < j
    using L_def L_props
    by (auto simp add: sup_acc_def)
      (smt L_props(1) L_props(2) Max_ge Max_in mem_Collect_eq)
qed

lemma sup_acc_Some_ts: sup_acc step accept rho q i j = Some (ts, tp) ⟹ ts = ts_at rho tp
  by (auto simp add: sup_acc_def Let_def split: if_splits)

lemma sup_acc_SomeE: sup_acc step accept rho q i j = Some (ts, tp) ⟹
  tp ∈ {i..<j} ∧ acc step accept rho q (i, Suc tp)
proof −
  assume assms: sup_acc step accept rho q i j = Some (ts, tp)
  define L where L  = {l ∈ {i..<j}. acc step accept rho q (i, Suc l)}
  have L_props: finite L L ≠ {} Max L = tp
    unfolding L_def using assms
    by (auto simp add: sup_acc_def Let_def split: if_splits)
  show ?thesis
    using Max_in[OF L_props(1,2)] unfolding L_props(3) unfolding L_def by auto
qed

lemma sup_acc_NoneE: l ∈ {i..<j} ⟹ sup_acc step accept rho q i j = None ⟹
  ¬acc step accept rho q (i, Suc l)
  by (auto simp add: sup_acc_def Let_def split: if_splits)

lemma sup_acc_same: sup_acc step accept rho q i i = None
  by (auto simp add: sup_acc_def)

lemma sup_acc_None_restrict: i ≤ j ⟹ sup_acc step accept rho q i j = None ⟹
  sup_acc step accept rho (step q (bs_at rho i)) (Suc i) j = None
  using steps_split
  apply (auto simp add: sup_acc_def Let_def acc_def split: if_splits)
  apply (smt (z3) lessI less_imp_le_nat order_less_le_trans steps_split)
  done

lemma sup_acc_ext_idle: i ≤ j ⟹ ¬acc step accept rho q (i, Suc j) ⟹
  sup_acc step accept rho q i (Suc j) = sup_acc step accept rho q i j
proof −
  assume assms: i ≤ j ¬acc step accept rho q (i, Suc j)
  define L where L = {l ∈ {i..<j}. acc step accept rho q (i, Suc l)}
  define L' where L' = {l ∈ {i..<Suc j}. acc step accept rho q (i, Suc l)}
  have L_L': L = L'
    unfolding L_def L'_def using assms(2) less_antisym by fastforce
```

```
  show sup_acc step accept rho q i (Suc j) = sup_acc step accept rho q i j
    using L_def L'_def L_L' by (auto simp add: sup_acc_def)
qed

lemma sup_acc_comp_Some_ge: i ≤ l ⟹ l ≤ j ⟹ tp ≥ l ⟹
  sup_acc step accept rho (steps step rho q (i, l)) l j = Some (ts, tp) ⟹
  sup_acc step accept rho q i j = sup_acc step accept rho (steps step rho q (i, l)) l j
proof −
  assume assms: i ≤ l l ≤ j sup_acc step accept rho (steps step rho q (i, l)) l j =
    Some (ts, tp) tp ≥ l
  define L where L = {l ∈ {i..<j}. acc step accept rho q (i, Suc l)}
  define L' where L' = {l' ∈ {l..<j}. acc step accept rho (steps step rho q (i, l)) (l, Suc l')}
  have L'_props: finite L' L' ≠ {} tp = Max L' ts = ts_at rho tp
    using assms(3) unfolding L'_def
    by (auto simp add: sup_acc_def Let_def split: if_splits)
  have tp_in_L': tp ∈ L'
    using Max_in[OF L'_props(1,2)] unfolding L'_props(3) .
  then have tp_in_L: tp ∈ L
    unfolding L_def L'_def using assms(1) steps_comp[OF assms(1,2), of step rho]
    apply (auto simp add: acc_def)
    using steps_comp
    by (metis le_SucI)
  have L_props: finite L L ≠ {}
    using L_def tp_in_L by auto
  have ⋀l'. l' ∈ L ⟹ l' ≤ tp
  proof −
    fix l'
    assume assm: l' ∈ L
    show l' ≤ tp
    proof (cases l' < l)
      case True
      then show ?thesis
        using assms(4) by auto
    next
      case False
      then have l' ∈ L'
        using assm
        unfolding L_def L'_def
        by (auto simp add: acc_def) (metis assms(1) less_imp_le_nat not_less_eq steps_comp)
      then show ?thesis
        using Max_eq_iff[OF L'_props(1,2)] L'_props(3) by auto
    qed
  qed
  then have Max L = tp
    using Max_eq_iff[OF L_props] tp_in_L by auto
  then have sup_acc step accept rho q i j = Some (ts, tp)
    using L_def L_props(2) unfolding L'_props(4)
    by (auto simp add: sup_acc_def)
  then show sup_acc step accept rho q i j = sup_acc step accept rho (steps step rho q (i, l)) l j
    using assms(3) by auto
qed

lemma sup_acc_comp_None: i ≤ l ⟹ l ≤ j ⟹
  sup_acc step accept rho (steps step rho q (i, l)) l j = None ⟹
  sup_acc step accept rho q i j = sup_acc step accept rho q i l
proof (induction j − l arbitrary: l)
  case (Suc n)
  have i_lt_j: i < j
```

```
    using Suc by auto
  have l_lt_j: l < j
    using Suc by auto
  have ¬acc step accept rho q (i, Suc l)
    using sup_acc_NoneE[of l l j step accept rho steps step rho q (i, l)] Suc(2−)
    by (auto simp add: acc_def steps_def)
  then have sup_acc step accept rho q i (l + 1) = sup_acc step accept rho q i l
    using sup_acc_ext_idle[OF Suc(3)] by auto
  moreover have sup_acc step accept rho (steps step rho q (i, l + 1)) (l + 1) j = None
    using sup_acc_None_restrict[OF Suc(4,5)] steps_app[OF Suc(3), of step rho]
    by auto
  ultimately show ?case
    using Suc(1)[of l + 1] Suc(2,3,4,5)
    by auto
qed (auto simp add: sup_acc_same)

lemma sup_acc_ext: i ≤ j ⟹ acc step accept rho q (i, Suc j) ⟹
  sup_acc step accept rho q i (Suc j) = Some (ts_at rho j, j)
proof −
  assume assms: i ≤ j acc step accept rho q (i, Suc j)
  define L' where L' = {l ∈ {i..<j + 1}. acc step accept rho q (i, Suc l)}
  have j_in_L': finite L' L' ≠ {} j ∈ L'
    using assms unfolding L'_def by auto
  have j_is_Max: Max L' = j
    using Max_eq_iff[OF j_in_L'(1,2)] j_in_L'(3)
    by (auto simp add: L'_def)
  show sup_acc step accept rho q i (Suc j) = Some (ts_at rho j, j)
    using L'_def j_is_Max j_in_L'(2)
    by (auto simp add: sup_acc_def)
qed

lemma sup_acc_None: i < j ⟹ sup_acc step accept rho q i j = None ⟹
  sup_acc step accept rho (step q (bs_at rho i)) (i + 1) j = None
  using steps_split[of _ _ step rho]
  by (auto simp add: sup_acc_def Let_def acc_def split: if_splits)

lemma sup_acc_i: i < j ⟹ sup_acc step accept rho q i j = Some (ts, i) ⟹
  sup_acc step accept rho (step q (bs_at rho i)) (Suc i) j = None
proof (rule ccontr)
  assume assms: i < j sup_acc step accept rho q i j = Some (ts, i)
    sup_acc step accept rho (step q (bs_at rho i)) (Suc i) j ≠ None
  from assms(3) obtain l where l_def: l ∈ {Suc i..<j}
    acc step accept rho (step q (bs_at rho i)) (Suc i, Suc l)
    by (auto simp add: sup_acc_def Let_def split: if_splits)
  define L' where L' = {l ∈ {i..<j}. acc step accept rho q (i, Suc l)}
  from assms(2) have L'_props: finite L' L' ≠ {} Max L' = i
    by (auto simp add: sup_acc_def L'_def Let_def split: if_splits)
  have i_lt_l: i < l
    using l_def(1) by auto
  from l_def have l ∈ L'
    unfolding L'_def acc_def using steps_split[OF i_lt_l, of step rho] by (auto simp: steps_def)
  then show False
    using l_def(1) L'_props Max_ge i_lt_l not_le by auto
qed

lemma sup_acc_l: i < j ⟹ i ≠ l ⟹ sup_acc step accept rho q i j = Some (ts, l) ⟹
  sup_acc step accept rho q i j = sup_acc step accept rho (step q (bs_at rho i)) (Suc i) j
proof −
```

assume *assms*: $i < j$ $i \neq l$ *sup_acc step accept rho q i j* = *Some* (*ts*, *l*)
define L **where** $L = \{l \in \{i..<j\}.$ *acc step accept rho q* (*i*, *Suc l*)$\}$
define L' **where** $L' = \{l \in \{Suc\ i..<j\}.$ *acc step accept rho* (*step q* (*bs_at rho i*)) (*Suc i*, *Suc l*)$\}$
from *assms*(*3*) **have** *L_props*: *finite L* $L \neq \{\}$ l = *Max L*
sup_acc step accept rho q i j = *Some* (*ts_at rho l*, *l*)
by (*auto simp add*: *sup_acc_def L_def Let_def split*: *if_splits*)
have *l_in_L*: $l \in L$
using *Max_in*[*OF L_props*(*1*,*2*)] *L_props*(*3*) **by** *auto*
then have *i_lt_l*: $i < l$
unfolding *L_def* **using** *assms*(*2*) **by** *auto*
have *l_in_L'*: *finite* L' $L' \neq \{\}$ $l \in L'$
using *steps_split*[*OF i_lt_l*, *of step rho q*] *l_in_L assms*(*2*)
unfolding *L_def L'_def acc_def* **by** (*auto simp*: *steps_def*)
have $\bigwedge l'.\ l' \in L' \Longrightarrow l' \leq l$
proof −
fix l'
assume *assms*: $l' \in L'$
have *i_lt_l'*: $i < l'$
using *assms* **unfolding** *L'_def* **by** *auto*
have $l' \in L$
using *steps_split*[*OF i_lt_l'*, *of step rho*] *assms* **unfolding** *L_def L'_def acc_def* **by** (*auto simp*: *steps_def*)
then show $l' \leq l$
using *L_props* **by** *simp*
qed
then have *l_sup_L'*: *Max* $L' = l$
using *Max_eq_iff*[*OF l_in_L'*(*1*,*2*)] *l_in_L'*(*3*) **by** *auto*
then show *sup_acc step accept rho q i j* =
sup_acc step accept rho (*step q* (*bs_at rho i*)) (*Suc i*) *j*
unfolding *L_props*(*4*)
unfolding *sup_acc_def Let_def*
using *L'_def l_in_L'*(*2*,*3*) *L_props*
unfolding *Suc_eq_plus1* **by** *auto*
qed

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 −
assume *assms*: $i < j$ *steps step rho init* (*i*, *j*) $\neq q$
define L **where** $L = \{l.\ l < i \wedge$ *steps step rho init* $(l, j) = q\}$
define L' **where** $L' = \{l.\ l < i + 1 \wedge$ *steps step rho init* $(l, j) = q\}$
have *L_L'*: $L = L'$
unfolding *L_def L'_def* **using** *assms*(*2*) *less_antisym*
by *fastforce*
show *sup_leadsto init step rho i j q* = *sup_leadsto init step rho* (*i* + *1*) *j q*
using *L_def L'_def L_L'*
by (*auto simp add*: *sup_leadsto_def*)
qed

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 −
assume *assms*: $l < i$ *steps step rho init* $(l, j) = q$
define L' **where** $L' = \{l.\ l < i \wedge$ *steps step rho init* $(l, j) = q\}$
have *fin_L'*: *finite* L'
unfolding *L'_def* **by** *auto*
moreover have *L_nonempty*: $L' \neq \{\}$
using *assms* **unfolding** *L'_def*

```
    by (auto simp add: sup_leadsto_def split: if_splits)
  ultimately have Max L' ∈ L'
    using Max_in by auto
  then show ∃ l'. sup_leadsto init step rho i j q = Some (ts_at rho l') ∧ l ≤ l' ∧ l' < i
    using L'_def L_nonempty assms
    by (fastforce simp add: sup_leadsto_def split: if_splits)
qed

lemma sup_leadsto_SomeE: i ≤ j ⟹ sup_leadsto init step rho i j q = Some ts ⟹
  ∃ l < i. steps step rho init (l, j) = q ∧ ts_at rho l = ts
proof −
  assume assms: i ≤ j sup_leadsto init step rho i j q = Some ts
  define L' where L' = {l. l < i ∧ steps step rho init (l, j) = q}
  have fin_L': finite L'
    unfolding L'_def by auto
  moreover have L_nonempty: L' ≠ {}
    using assms(2) unfolding L'_def
    by (auto simp add: sup_leadsto_def split: if_splits)
  ultimately have Max L' ∈ L'
    using Max_in by auto
  then show ∃ l < i. steps step rho init (l, j) = q ∧ ts_at rho l = ts
    using assms(2) L'_def
    apply (auto simp add: sup_leadsto_def split: if_splits)
    using ‹Max L' ∈ L'› by blast
qed

lemma Mapping_keys_dest: x ∈ mmap_keys f ⟹ ∃ y. mmap_lookup f x = Some y
  by (auto simp add: mmap_keys_def mmap_lookup_def weak_map_of_SomeI)

lemma Mapping_keys_intro: mmap_lookup f x ≠ None ⟹ x ∈ mmap_keys f
  by (auto simp add: mmap_keys_def mmap_lookup_def)
     (metis map_of_eq_None_iff option.distinct(1))

lemma Mapping_not_keys_intro: mmap_lookup f x = None ⟹ x ∉ mmap_keys f
  unfolding mmap_lookup_def mmap_keys_def
  using weak_map_of_SomeI by force

lemma Mapping_lookup_None_intro: x ∉ mmap_keys f ⟹ mmap_lookup f x = None
  unfolding mmap_lookup_def mmap_keys_def
  by (simp add: map_of_eq_None_iff)

primrec mmap_combine :: 'key ⇒ 'val ⇒ ('val ⇒ 'val ⇒ 'val) ⇒ ('key × 'val) list ⇒
  ('key × 'val) list
  where
  mmap_combine k v c [] = [(k, v)]
| mmap_combine k v c (p # ps) = (case p of (k', v') ⇒
    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) ⟹
  distinct (map fst (mmap_combine k v c r)) ∧
  set (map fst (mmap_combine k v c r)) = set (map fst r) ∪ {k}
  by (induction r) force+

lemma mmap_combine_lookup: distinct (map fst r) ⟹ mmap_lookup (mmap_combine k v c r) z =
  (if k = z then (case mmap_lookup r k of None ⇒ Some v | Some v' ⇒ Some (c v' v))
  else mmap_lookup r z)
  using eq_key_imp_eq_value
  by (induction r) (fastforce simp: mmap_lookup_def split: option.splits)+
```

```
definition mmap_fold :: ('c, 'd) mmap ⇒ (('c × 'd) ⇒ ('c × 'd)) ⇒ ('d ⇒ 'd ⇒ 'd) ⇒
  ('c, 'd) mmap ⇒ ('c, 'd) mmap where
  mmap_fold m f c r = foldl (λr p. case f p of (k, v) ⇒ mmap_combine k v c r) r m

definition mmap_fold' :: ('c, 'd) mmap ⇒ 'e ⇒ (('c × 'd) × 'e ⇒ ('c × 'd) × 'e) ⇒
  ('d ⇒ 'd ⇒ 'd) ⇒ ('c, 'd) mmap ⇒ ('c, 'd) mmap × 'e where
  mmap_fold' m e f c r = foldl (λ(r, e) p. case f (p, e) of ((k, v), e') ⇒
    (mmap_combine k v c r, e')) (r, e) m

lemma mmap_fold'_eq: mmap_fold' m e f' c r = (m', e') ⟹ P e ⟹
  (⋀p e p' e'. P e ⟹ f' (p, e) = (p', e') ⟹ p' = f p ∧ P e') ⟹
  m' = mmap_fold m f c r ∧ P e'
proof (induction m arbitrary: e r m' e')
  case (Cons p m)
  obtain k v e'' where kv_def: f' (p, e) = ((k, v), e'') P e''
    using Cons
    by (cases f' (p, e)) fastforce
  have mmap_fold: mmap_fold m f c (mmap_combine k v c r) = mmap_fold (p # m) f c r
    using Cons(1)[OF _ kv_def(2), where ?r=mmap_combine k v c r] Cons(2,3,4)
    by (simp add: mmap_fold_def mmap_fold'_def kv_def(1))
  have mmap_fold': mmap_fold' m e'' f' c (mmap_combine k v c r) = (m', e')
    using Cons(2)
    by (auto simp: mmap_fold'_def kv_def)
  show ?case
    using Cons(1)[OF mmap_fold' kv_def(2) Cons(4)]
    unfolding mmap_fold
    by auto
qed (auto simp: mmap_fold_def mmap_fold'_def)

lemma foldl_mmap_combine_distinct_set: distinct (map fst r) ⟹
  distinct (map fst (mmap_fold m f c r)) ∧
  set (map fst (mmap_fold m f c r)) = set (map fst r) ∪ set (map (fst ∘ f) m)
  apply (induction m arbitrary: r)
  using mmap_combine_distinct_set
   apply (auto simp: mmap_fold_def split: prod.splits)
     apply force
    apply (smt Un_iff fst_conv imageI insert_iff)
  using mk_disjoint_insert
   apply fastforce+
  done

lemma mmap_fold_lookup_rec: distinct (map fst r) ⟹ mmap_lookup (mmap_fold m f c r) z =
  (case map (snd ∘ f) (filter (λ(k, v). fst (f (k, v)) = z) m) of [] ⇒ mmap_lookup r z
  | v # vs ⇒ (case mmap_lookup r z of None ⇒ Some (foldl c v vs)
    | Some w ⇒ Some (foldl c w (v # vs))))
proof (induction m arbitrary: r)
  case (Cons p ps)
  obtain k v where kv_def: f p = (k, v)
    by fastforce
  have distinct: distinct (map fst (mmap_combine k v c r))
    using mmap_combine_distinct_set[OF Cons(2)]
    by auto
  show ?case
    using Cons(1)[OF distinct, unfolded mmap_combine_lookup[OF Cons(2)]]
    by (auto simp: mmap_lookup_def kv_def mmap_fold_def split: list.splits option.splits)
qed (auto simp: mmap_fold_def)
```

lemma *mmap_fold_distinct*: *distinct* (*map fst m*) ⟹ *distinct* (*map fst* (*mmap_fold m f c* [])) **using** *foldl_mmap_combine_distinct_set*[*of* []] **by** *auto*

lemma *mmap_fold_set*: *distinct* (*map fst m*) ⟹ *set* (*map fst* (*mmap_fold m f c* [])) = (*fst* ∘ *f*) ‘ *set m* **using** *foldl_mmap_combine_distinct_set*[*of* []] **by** *force*

lemma *mmap_lookup_empty*: *mmap_lookup* [] *z* = *None* **by** (*auto simp*: *mmap_lookup_def*)

lemma *mmap_fold_lookup*: *distinct* (*map fst m*) ⟹ *mmap_lookup* (*mmap_fold m f c* []) *z* = (*case map* (*snd* ∘ *f*) (*filter* (λ(*k*, *v*). *fst* (*f* (*k*, *v*)) = *z*) *m*) *of* [] ⇒ *None* | *v* # *vs* ⇒ *Some* (*foldl c v vs*)) **using** *mmap_fold_lookup_rec*[*of* [] _ *f c*] **by** (*auto simp*: *mmap_lookup_empty split*: *list.splits*)

definition *fold_sup* :: (′*c*, ′*d* :: *timestamp*) *mmap* ⇒ (′*c* ⇒ ′*c*) ⇒ (′*c*, ′*d*) *mmap* **where** *fold_sup m f* = *mmap_fold m* (λ(*x*, *y*). (*f x*, *y*)) *sup* []

lemma *mmap_lookup_distinct*: *distinct* (*map fst m*) ⟹ (*k*, *v*) ∈ *set m* ⟹ *mmap_lookup m k* = *Some v* **by** (*auto simp*: *mmap_lookup_def*)

lemma *fold_sup_distinct*: *distinct* (*map fst m*) ⟹ *distinct* (*map fst* (*fold_sup m f*)) **using** *mmap_fold_distinct* **by** (*auto simp*: *fold_sup_def*)

lemma *fold_sup*: **fixes** *v* :: ′*d* :: *timestamp* **shows** *foldl sup v vs* = *fold sup vs v* **by** (*induction vs arbitrary*: *v*) (*auto simp*: *sup.commute*)

lemma *lookup_fold_sup*: **assumes** *distinct*: *distinct* (*map fst m*) **shows** *mmap_lookup* (*fold_sup m f*) *z* = (*let Z* = {*x* ∈ *mmap_keys m*. *f x* = *z*} *in* *if Z* = {} *then None else Some* (*Sup_fin* ((*the* ∘ *mmap_lookup m*) ‘ *Z*)))
proof (*cases* {*x* ∈ *mmap_keys m*. *f x* = *z*} = {})
case *True*
have *z* ∉ *mmap_keys* (*mmap_fold m* (λ(*x*, *y*). (*f x*, *y*)) *sup* [])
using *True*[*unfolded mmap_keys_def*] *mmap_fold_set*[*OF distinct*]
by (*auto simp*: *mmap_keys_def*)
then have *mmap_lookup* (*fold_sup m f*) *z* = *None*
unfolding *fold_sup_def*
by (*meson Mapping_keys_intro*)
then show *?thesis*
unfolding *True*
by *auto*
next
case *False*
have *z_in_keys*: *z* ∈ *mmap_keys* (*mmap_fold m* (λ(*x*, *y*). (*f x*, *y*)) *sup* [])
using *False*[*unfolded mmap_keys_def*] *mmap_fold_set*[*OF distinct*]
by (*force simp*: *mmap_keys_def*)
obtain *v vs* **where** *vs_def*: *mmap_lookup* (*fold_sup m f*) *z* = *Some* (*foldl sup v vs*) *v* # *vs* = *map snd* (*filter* (λ(*k*, *v*). *f k* = *z*) *m*)
using *mmap_fold_lookup*[*OF distinct*, *of* (λ(*x*, *y*). (*f x*, *y*)) *sup z*]

```
    Mapping_keys_dest[OF z_in_keys]
   by (force simp: fold_sup_def mmap_keys_def comp_def split: list.splits prod.splits)
  have set (v # vs) = (the ∘ mmap_lookup m) ‘ {x ∈ mmap_keys m. f x = z}
  proof (rule set_eqI, rule iffI)
   fix w
   assume w ∈ set (v # vs)
   then obtain x where x_def: x ∈ mmap_keys m f x = z (x, w) ∈ set m
    using vs_def(2)
    by (auto simp add: mmap_keys_def rev_image_eqI)
   show w ∈ (the ∘ mmap_lookup m) ‘ {x ∈ mmap_keys m. f x = z}
    using x_def(1,2) mmap_lookup_distinct[OF distinct x_def(3)]
    by force
  next
   fix w
   assume w ∈ (the ∘ mmap_lookup m) ‘ {x ∈ mmap_keys m. f x = z}
   then obtain x where x_def: x ∈ mmap_keys m f x = z (x, w) ∈ set m
    using mmap_lookup_distinct[OF distinct]
    by (auto simp add: Mapping_keys_intro distinct mmap_lookup_def dest: Mapping_keys_dest)
   show w ∈ set (v # vs)
    using x_def
    by (force simp: vs_def(2))
  qed
  then have foldl sup v vs = Sup_fin ((the ∘ mmap_lookup m) ‘ {x ∈ mmap_keys m. f x = z})
   unfolding fold_sup
   by (metis Sup_fin.set_eq_fold)
  then show ?thesis
   using False
   by (auto simp: vs_def(1))
qed

definition mmap_map :: ('a ⇒ 'b ⇒ 'c) ⇒ ('a, 'b) mmap ⇒ ('a, 'c) mmap where
  mmap_map f m = map (λ(k, v). (k, f k v)) m

lemma mmap_map_keys: mmap_keys (mmap_map f m) = mmap_keys m
  by (force simp: mmap_map_def mmap_keys_def)

lemma mmap_map_fst: map fst (mmap_map f m) = map fst m
  by (auto simp: mmap_map_def)

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 (*induction qbss arbitrary*: *st ac qbss'*)
case (*Cons a qbss*)
obtain *q q' tstp* **where** *a_def*: *a* = (*q*, (*q'*, *tstp*))
by (*cases a*) *auto*
obtain *q''* *st''* **where** *q''_def*: *cstep step st q' bs* = (*q''*, *st''*)
q'' = *step q' bs*
using *Cons*(*3*)
by (*cases cstep step st q' bs*)
(*auto simp*: *cstep_def Let_def option.case_eq_if split*: *option.splits if_splits*)
obtain *b ac''* **where** *b_def*: *cac accept ac q''* = (*b*, *ac''*)
b = *accept q''*
using *Cons*(*4*)
by (*cases cac accept ac q''*)
(*auto simp*: *cac_def Let_def option.case_eq_if split*: *option.splits if_splits*)
obtain *qbss''* **where** *qbss''_def*: *mmap_fold_s step st'' accept ac'' bs t j qbss* =
(*qbss''*, *st'*, *ac'*) *qbss'* = (*q*, *q''*, *if b then Some* (*t*, *j*) *else tstp*) # *qbss''*
using *Cons*(*2*)[*unfolded a_def mmap_fold_s.simps q''_def*(*1*) *b_def*(*1*)]
unfolding *Let_def*
by (*auto simp*: *b_def*(*1*) *split*: *prod.splits*)
have *ih*: ⋀*q bs*. *case Mapping.lookup st''* (*q*, *bs*) *of None* ⇒ *True* | *Some a* ⇒ *step q bs* = *a*
⋀*q bs*. *case Mapping.lookup ac''* *q of None* ⇒ *True* | *Some a* ⇒ *accept q* = *a*
using *q''_def b_def Cons*(*3*,*4*)
by (*auto simp*: *cstep_def cac_def Let_def Mapping.lookup_update' option.case_eq_if*
split: *option.splits if_splits*)
show *?case*
using *Cons*(*1*)[*OF qbss''_def*(*1*) *ih*]
unfolding *a_def q''_def*(*2*) *b_def*(*2*) *qbss''_def*(*2*)
by (*auto simp*: *mmap_map_def*)
qed (*auto simp*: *mmap_map_def*)

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*
by (*induction m*) (*auto simp*: *mmap_lookup_def mmap_map_def*)

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 −
define *L* **where** *L* = {*l* ∈ {*i*..<*Suc j*}. *accept* (*steps step rho q* (*i*, *Suc l*))}

```
  have nonempty: finite L L ≠ {}
    using assms unfolding L_def by auto
  moreover have Max {l ∈ {i..<Suc j}. accept (steps step rho q (i, Suc l))} = j
    unfolding L_def[symmetric] Max_eq_iff[OF nonempty, of j]
    unfolding L_def using assms by auto
  ultimately show ?thesis
    by (auto simp add: sup_acc_def acc_def L_def)
qed

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
  using assms
  by (auto simp add: sup_acc_def Let_def acc_def elim: less_SucE) (metis less_Suc_eq)+

lemma sup_fin_closed: finite A ⟹ A ≠ {} ⟹
  (⋀x y. x ∈ A ⟹ y ∈ A ⟹ sup x y ∈ {x, y}) ⟹ ⨆fin A ∈ A
  apply (induct A rule: finite.induct)
  using Sup_fin.insert
  by auto fastforce

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 −
  define init where init = w_init args
  define step where step = w_step args
  define accept where accept = w_accept args
  define run_t where run_t = w_run_t args
  define run_sub where run_sub = w_run_sub args
  define st where st = w_st w
  define ac where ac = w_ac w
  define i where i = w_i w
  define ti where ti = w_ti w
  define si where si = w_si w
  define j where j = w_j w
  define tj where tj = w_tj w
  define sj where sj = w_sj w
  define s where s = w_s w
  define e where e = w_e w
  have valid_before: 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 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)
    ∀ 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
    using assms(1)
    unfolding valid_window_def valid_s_def Let_def init_def step_def accept_def run_t_def
      run_sub_def st_def ac_def i_def ti_def si_def j_def tj_def sj_def s_def e_def
    by auto
  have i_j: i ≤ j
    using valid_before(1)
    by auto
  have distinct_before: distinct (map fst s) distinct (map fst e)
    using valid_before
```

by (*auto simp*: *valid_s_def*)
note *run_tj* = *assms*(2)[*folded run_t_def tj_def*]
note *run_sj* = *assms*(3)[*folded run_sub_def sj_def*]
define *rho'* **where** *rho'* = *rho* @ [(*t*, *bs*)]
have *ts_at_mono*: ⋀*i j*. *i* ≤ *j* ⟹ *j* < *length rho'* ⟹ *ts_at rho' i* ≤ *ts_at rho' j*
using *valid_before*(2) *assms*(4)
by (*auto simp*: *rho'_def ts_at_def nth_append split*: *option.splits list.splits if_splits*)
obtain *s' st' ac'* **where** *s'_def*: *mmap_fold_s step st accept ac bs t j s* = (*s'*, *st'*, *ac'*)
apply (*cases mmap_fold_s step st accept ac bs t j s*)
apply (*auto*)
done
have *s'_mmap_map*: *s'* = *mmap_map* (λ*q* (*q'*, *tstp*).
(*step q' bs*, *if accept* (*step q' bs*) *then Some* (*t*, *j*) *else tstp*)) *s*
(⋀*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*)
using *mmap_fold_s_sound*[*OF s'_def valid_before*(3,4)]
by *auto*
obtain *e' st''* **where** *e'_def*: *mmap_fold' e st'* (λ((*x*, *y*), *st*).
let (*q'*, *st'*) = *cstep step st x bs in* ((*q'*, *y*), *st'*)) *sup* [] = (*e'*, *st''*)
by (*metis old.prod.exhaust*)
define *inv* **where** *inv* ≡ λ*st'*. ∀ *q bs*. *case Mapping.lookup st'* (*q*, *bs*) *of None* ⇒ *True*
| *Some v* ⇒ *step q bs* = *v*
have *inv_st'*: *inv st'*
using *s'_mmap_map*(2)
by (*auto simp*: *inv_def*)
have ⋀*p e p' e'*. *inv e* ⟹ (*case* (*p*, *e*) *of* (*x*, *xa*) ⇒ (*case x of* (*x*, *y*) ⇒
λ*st*. *let* (*q'*, *st'*) = *cstep step st x bs in* ((*q'*, *y*), *st'*)) *xa*) = (*p'*, *e'*) ⟹
p' = (*case p of* (*x*, *y*) ⇒ (*step x bs*, *y*)) ∧ *inv e'*
by (*auto simp*: *inv_def cstep_def Let_def Mapping.lookup_update' split*: *option.splits if_splits*)
then have *e'_fold_sup_st''*: *e'* = *fold_sup e* (λ*q*. *step q bs*)
(⋀*q bs*. *case Mapping.lookup st''* (*q*, *bs*) *of None* ⇒ *True* | *Some v* ⇒ *step q bs* = *v*)
using *mmap_fold'_eq*[*OF e'_def*, *of inv* λ(*x*, *y*). (*step x bs*, *y*), *OF inv_st'*]
by (*fastforce simp*: *fold_sup_def inv_def*)+
have *adv_end*: *adv_end args w* = *Some* (*w*(|*w_st* := *st''*, *w_ac* := *ac'*,
w_j := *Suc j*, *w_tj* := *tj'*, *w_sj* := *sj'*, *w_s* := *s'*, *w_e* := *e'*|))
using *run_tj run_sj e'_def*[*unfolded st_def*]
unfolding *adv_end_def init_def step_def accept_def run_t_def run_sub_def*
i_def ti_def si_def j_def tj_def sj_def s_def e_def s'_def e'_def
by (*auto simp*: *Let_def s'_def*[*unfolded step_def st_def accept_def ac_def j_def s_def*])
have *keys_s'*: *mmap_keys s'* = *mmap_keys s*
by (*force simp*: *mmap_keys_def mmap_map_def s'_mmap_map*(1))
have *lookup_s*: ⋀*q q' tstp*. *mmap_lookup s q* = *Some* (*q'*, *tstp*) ⟹
steps step rho' q (*i*, *j*) = *q'* ∧ *tstp* = *sup_acc step accept rho' q i j*
using *valid_before Mapping_keys_intro*
by (*force simp add*: *Let_def rho'_def valid_s_def steps_app_cong sup_acc_app_cong*
split: *option.splits*)
have *bs_at_rho'_j*: *bs_at rho' j* = *bs*
using *valid_before*
by (*auto simp*: *rho'_def bs_at_def nth_append*)
have *ts_at_rho'_j*: *ts_at rho' j* = *t*
using *valid_before*
by (*auto simp*: *rho'_def ts_at_def nth_append*)
have *lookup_s'*: ⋀*q q' tstp*. *mmap_lookup s' q* = *Some* (*q'*, *tstp*) ⟹
steps step rho' q (*i*, *Suc j*) = *q'* ∧ *tstp* = *sup_acc step accept rho' q i* (*Suc j*)
proof −
fix *q q'' tstp'*
assume *assm*: *mmap_lookup s' q* = *Some* (*q''*, *tstp'*)
obtain *q' tstp* **where** *mmap_lookup s q* = *Some* (*q'*, *tstp*) *q''* = *step q' bs*

```
    tstp' = (if accept (step q' bs) then Some (t, j) else tstp)
    using map_values_lookup[OF assm[unfolded s'_mmap_map]] by auto
  then show steps step rho' q (i, Suc j) = q'' ∧ tstp' = sup_acc step accept rho' q i (Suc j)
    using lookup_s
    apply (auto simp: bs_at_rho'_j ts_at_rho'_j)
      apply (metis Suc_eq_plus1 bs_at_rho'_j i_j steps_app)
     apply (metis acc_app bs_at_rho'_j i_j steps_appE ts_at_rho'_j)
    apply (metis Suc_eq_plus1 bs_at_rho'_j i_j steps_app)
    apply (metis (no_types, lifting) acc_app_idle bs_at_rho'_j i_j steps_appE)
    done
qed
have lookup_e: ⋀q. mmap_lookup e q = sup_leadsto init step rho' i j q
  using valid_before sup_leadsto_app_cong[of _ _ rho init step]
  by (auto simp: rho'_def)
have keys_e_alt: mmap_keys e = {q. ∃ l < i. steps step rho' init (l, j) = q}
  using valid_before
  apply (auto simp add: sup_leadsto_def rho'_def)
   apply (metis (no_types, lifting) Mapping_keys_dest lookup_e rho'_def sup_leadsto_SomeE)
  apply (metis (no_types, lifting) Mapping_keys_intro option.simps(3) order_refl steps_app_cong)
  done
have finite_keys_e: finite (mmap_keys e)
  unfolding keys_e_alt
  by (rule finite_surj[of {l. l < i}]) auto
have reaches_on run_sub sub (map snd rho) sj
  using valid_before reaches_on_trans
  unfolding run_sub_def sub_def
  by fastforce
then have reaches_on': reaches_on run_sub sub (map snd rho @ [bs]) sj'
  using reaches_on_app run_sj
  by fast
have reaches_on run_t t0 (map fst rho) tj
  using valid_before reaches_on_trans
  unfolding run_t_def
  by fastforce
then have reach_t': reaches_on run_t t0 (map fst rho') tj'
  using reaches_on_app run_tj
  unfolding rho'_def
  by fastforce
have lookup_e': ⋀q. mmap_lookup e' q = sup_leadsto init step rho' i (Suc j) q
proof −
  fix q
  define Z where Z = {x ∈ mmap_keys e. step x bs = q}
  show mmap_lookup e' q = sup_leadsto init step rho' i (Suc j) q
  proof (cases Z = {})
    case True
    then have mmap_lookup e' q = None
      using Z_def lookup_fold_sup[OF distinct_before(2)]
      unfolding e'_fold_sup_st''
      by (auto simp: Let_def)
    moreover have sup_leadsto init step rho' i (Suc j) q = None
    proof (rule ccontr)
      assume assm: sup_leadsto init step rho' i (Suc j) q ≠ None
      obtain l where l_def: l < i steps step rho' init (l, Suc j) = q
        using i_j sup_leadsto_SomeE[of i Suc j] assm
        by force
      have l_j: l ≤ j
        using less_le_trans[OF l_def(1) i_j] by auto
      obtain q'' where q''_def: steps step rho' init (l, j) = q'' step q'' bs = q
```

```
    using steps_appE[OF _ l_def(2)] l_j
    by (auto simp: bs_at_rho'_j)
  then have q'' ∈ mmap_keys e
    using keys_e_alt l_def(1)
    by auto
  then show False
    using Z_def q''_def(2) True
    by auto
 qed
 ultimately show ?thesis
  by auto
next
 case False
 then have lookup_e': mmap_lookup e' q = Some (Sup_fin ((the ∘ mmap_lookup e) ‘ Z))
  using Z_def lookup_fold_sup[OF distinct_before(2)]
  unfolding e'_fold_sup_st''
  by (auto simp: Let_def)
 define L where L = {l. l < i ∧ steps step rho' init (l, Suc j) = q}
 have fin_L: finite L
  unfolding L_def by auto
 have Z_alt: Z = {x. ∃ l < i. steps step rho' init (l, j) = x ∧ step x bs = q}
  using Z_def[unfolded keys_e_alt] by auto
 have fin_Z: finite Z
  unfolding Z_alt by auto
 have L_nonempty: L ≠ {}
  using L_def Z_alt False i_j steps_app[of _ _ step rho q]
  by (auto simp: bs_at_rho'_j)
   (smt Suc_eq_plus1 bs_at_rho'_j less_irrefl_nat less_le_trans nat_le_linear steps_app)
 have sup_leadsto: sup_leadsto init step rho' i (Suc j) q = Some (ts_at rho' (Max L))
  using L_nonempty L_def
  by (auto simp add: sup_leadsto_def)
 have j_lt_rho': j < length rho'
  using valid_before
  by (auto simp: rho'_def)
 have Sup_fin ((the ∘ mmap_lookup e) ‘ Z) = ts_at rho' (Max L)
 proof (rule antisym)
  obtain z ts where zts_def: z ∈ Z (the ∘ mmap_lookup e) z = ts
   Sup_fin ((the ∘ mmap_lookup e) ‘ Z) = ts
  proof −
   assume lassm: ⋀z ts. z ∈ Z ⟹ (the ∘ mmap_lookup e) z = ts ⟹
   ⨆fin ((the ∘ mmap_lookup e) ‘ Z) = ts ⟹ thesis
   define T where T = (the ∘ mmap_lookup e) ‘ Z
   have T_sub: T ⊆ ts_at rho' ‘ {..j}
    using lookup_e keys_e_alt i_j
    by (auto simp add: T_def Z_def sup_leadsto_def)
   have finite T T ≠ {}
    using fin_Z False
    by (auto simp add: T_def)
   then have sup_in: ⨆fin T ∈ T
   proof (rule sup_fin_closed)
    fix x y
    assume xy: x ∈ T y ∈ T
    then obtain a c where x = ts_at rho' a y = ts_at rho' c a ≤ j c ≤ j
     using T_sub
     by (meson atMost_iff imageE subsetD)
    then show sup x y ∈ {x, y}
     using ts_at_mono j_lt_rho'
     by (cases a ≤ c) (auto simp add: sup.absorb1 sup.absorb2)
```

```
      qed
      then show ?thesis
        using lassm
        by (auto simp add: T_def)
    qed
    from zts_def(2) have lookup_e_z: mmap_lookup e z = Some ts
      using zts_def(1) Z_def by (auto dest: Mapping_keys_dest)
    have sup_leadsto init step rho' i j z = Some ts
      using lookup_e_z lookup_e
      by auto
    then obtain l where l_def: l < i steps step rho' init (l, j) = z ts_at rho' l = ts
      using sup_leadsto_SomeE[OF i_j]
      by (fastforce simp: rho'_def ts_at_def nth_append)
    have l_j: l ≤ j
      using less_le_trans[OF l_def(1) i_j] by auto
    have l ∈ L
      unfolding L_def using l_def zts_def(1) Z_alt
      by auto (metis (no_types, lifting) Suc_eq_plus1 bs_at_rho'_j l_j steps_app)
    then have l ≤ Max L Max L < i
      using L_nonempty fin_L
      by (auto simp add: L_def)
    then show Sup_fin ((the ∘ mmap_lookup e) ‘ Z) ≤ ts_at rho' (Max L)
      unfolding zts_def(3) l_def(3)[symmetric]
      using ts_at_mono i_j j_lt_rho'
      by (auto simp: rho'_def)
  next
    obtain l where l_def: Max L = l l < i steps step rho' init (l, Suc j) = q
      using Max_in[OF fin_L L_nonempty] L_def by auto
    obtain z where z_def: steps step rho' init (l, j) = z step z bs = q
      using l_def(2,3) i_j bs_at_rho'_j
      by (metis less_imp_le_nat less_le_trans steps_appE)
    have z_in_Z: z ∈ Z
      unfolding Z_alt
      using l_def(2) z_def i_j
      by fastforce
    have lookup_e_z: mmap_lookup e z = sup_leadsto init step rho' i j z
      using lookup_e z_in_Z Z_alt
      by auto
    obtain l' where l'_def: sup_leadsto init step rho' i j z = Some (ts_at rho' l')
      l ≤ l' l' < i
      using sup_leadsto_SomeI[OF l_def(2) z_def(1)] by auto
    have ts_at rho' l' ∈ (the ∘ mmap_lookup e) ‘ Z
      using lookup_e_z l'_def(1) z_in_Z
      by force
    then have ts_at rho' l' ≤ Sup_fin ((the ∘ mmap_lookup e) ‘ Z)
      using Inf_fin_le_Sup_fin fin_Z z_in_Z
      by (simp add: Sup_fin.coboundedI)
    then show ts_at rho' (Max L) ≤ Sup_fin ((the ∘ mmap_lookup e) ‘ Z)
    unfolding l_def(1)
    using ts_at_mono l'_def(2,3) i_j j_lt_rho'
    by (fastforce simp: rho'_def)
  qed
  then show ?thesis
    unfolding lookup_e' sup_leadsto by auto
 qed
qed
have distinct (map fst s') distinct (map fst e')
 using distinct_before mmap_fold_distinct
```

```
    unfolding s'_mmap_map mmap_map_fst e'_fold_sup_st'' fold_sup_def
    by auto
  moreover have mmap_keys s' = {q. ∃ l≤i. steps step rho' init (l, i) = q}
    unfolding keys_s' rho'_def
    using valid_before(1,7) valid_s_def[of init step st accept rho i i j s]
    by (auto simp: steps_app_cong[of _ rho step])
  moreover have reaches_on run_t ti (drop i (map fst rho')) tj'
    reaches_on run_sub si (drop i (map snd rho')) sj'
    using valid_before reaches_on_app run_tj run_sj
    by (auto simp: rho'_def run_t_def run_sub_def)
  ultimately show ?thesis
    unfolding adv_end
    using valid_before lookup_e' lookup_s' ts_at_mono s'_mmap_map(3) e'_fold_sup_st''(2)
    by (fastforce simp: valid_window_def Let_def init_def step_def accept_def run_t_def
        run_sub_def i_def ti_def si_def j_def tj_def sj_def s_def e'_def
        rho'_def valid_s_def intro!: exI[of _ rho'] split: option.splits)
qed

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'
  using assms
  by (auto simp: adv_end_def Let_def split: prod.splits)

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')) (λx 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')) (λx y. x) r = (s', st')
  shows s' = mmap_fold s (λ(x, v). (step x bs, drop_cur i v)) (λx y. x) r ∧
    (∀ q bs. case Mapping.lookup st' (q, bs) of None ⇒ True | Some v ⇒ step q bs = v)
proof −
  define inv where inv ≡ λst. ∀ q bs. case Mapping.lookup st (q, bs) of None ⇒ True
    | Some v ⇒ step q bs = v
  have inv_st: inv st
    using inv
    by (auto simp: inv_def)
  show ?thesis
    by (rule mmap_fold'_eq[OF fold', of inv λ(x, v). (step x bs, drop_cur i v),
          OF inv_st, unfolded inv_def])
       (auto simp: cstep_def Let_def Mapping.lookup_update'
        split: prod.splits option.splits if_splits)
qed

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' = (λq. step q bs) ‘ (mmap_keys s)
  using adv_d_mmap_fold[OF inv adv_d[unfolded adv_d_def]]
    mmap_fold_set[OF distinct]
  unfolding mmap_keys_def
  by fastforce

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 −
  have z ∉ mmap_keys (mmap_fold s (λ(x, v). (step x bs, drop_cur i v)) (λx y. x) [])
    using Z_empty[unfolded mmap_keys_def] mmap_fold_set[OF distinct]
    by (auto simp: mmap_keys_def)
  then show ?thesis
    using adv_d adv_d_mmap_fold[OF inv adv_d[unfolded adv_d_def]]
    unfolding adv_d_def
    by (simp add: Mapping_lookup_None_intro)
qed

lemma lookup_adv_d_Some:
  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 idem: keys_idem step i bs s
    and wit: x ∈ 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 −
  have z_in_keys: z ∈ mmap_keys (mmap_fold s (λ(x, v). (step x bs, drop_cur i v)) (λx y. x) [])
    using wit(1,2)[unfolded mmap_keys_def] mmap_fold_set[OF distinct]
    by (force simp: mmap_keys_def)
  obtain v vs where vs_def: mmap_lookup s' z = Some (foldl (λx y. x) v vs)
    v # vs = map (λ(x, v). drop_cur i v) (filter (λ(k, v). step k bs = z) s)
    using adv_d adv_d_mmap_fold[OF inv adv_d[unfolded adv_d_def]]
    unfolding adv_d_def
    using mmap_fold_lookup[OF distinct, of (λ(x, v). (step x bs, drop_cur i v)) λx y. x z]
      Mapping_keys_dest[OF z_in_keys]
    by (force simp: adv_d_def mmap_keys_def split: list.splits)
  have set (v # vs) = drop_cur i ‘ (the ∘ mmap_lookup s) ‘ {x ∈ mmap_keys s. step x bs = z}
  proof (rule set_eqI, rule iffI)
    fix w
    assume w ∈ set (v # vs)
    then obtain x y where xy_def: x ∈ mmap_keys s step x bs = z (x, y) ∈ set s
      w = drop_cur i y
      using vs_def(2)
      by (auto simp add: mmap_keys_def rev_image_eqI)
    show w ∈ drop_cur i ‘ (the ∘ mmap_lookup s) ‘ {x ∈ mmap_keys s. step x bs = z}
      using xy_def(1,2,4) mmap_lookup_distinct[OF distinct xy_def(3)]
      by force
```

```
  next
    fix w
    assume w ∈ drop_cur i ‘ (the ∘ mmap_lookup s) ‘ {x ∈ mmap_keys s. step x bs = z}
    then obtain x y where xy_def: x ∈ mmap_keys s step x bs = z (x, y) ∈ set s
      w = drop_cur i y
      using mmap_lookup_distinct[OF distinct]
      by (auto simp add: Mapping_keys_intro distinct mmap_lookup_def dest: Mapping_keys_dest)
    show w ∈ set (v # vs)
      using xy_def
      by (force simp: vs_def(2))
  qed
  then have foldl (λx y. x) v vs = drop_cur i (the (mmap_lookup s x))
    using wit
    apply (induction vs arbitrary: v)
     apply (auto)
    apply (metis (mono_tags, lifting) emptyE imageI insertE mem_Collect_eq)
    apply (smt Collect_cong idem imageE insert_compr keys_idem_def mem_Collect_eq)
    done
  then show ?thesis
    using wit
    by (auto simp: vs_def(1))
qed

definition loop_cond j = (λ(st, ac, i, ti, si, q, s, tstp). i < j ∧ q ∉ mmap_keys s)
definition loop_body step accept run_t run_sub =
  (λ(st, ac, i, ti, si, q, s, tstp). case run_t ti of Some (ti', t) ⇒
  case run_sub si of Some (si', b) ⇒ case adv_d step st i b s of (s', st') ⇒
  case cstep step st' q b of (q', st'') ⇒ case cac accept ac q' of (β, ac') ⇒
  (st'', ac', Suc i, ti', si', q', s', if β then Some (t, i) else tstp))
definition loop_inv init step accept args t0 sub rho u j tj sj =
  (λ(st, ac, i, ti, si, q, s, tstp). u + 1 ≤ i ∧
  reach_window args t0 sub rho (i, ti, si, j, tj, sj) ∧
  steps step rho init (u + 1, i) = q ∧
  (∀ q. case Mapping.lookup ac q of None ⇒ True | Some v ⇒ accept q = v) ∧
  valid_s init step st accept rho u i j s ∧ tstp = sup_acc step accept rho init (u + 1) i)

definition mmap_update :: 'a ⇒ 'b ⇒ ('a, 'b) mmap ⇒ ('a, 'b) mmap where
  mmap_update = AList.update

lemma mmap_update_distinct: distinct (map fst m) ⟹ distinct (map fst (mmap_update k v m))
  by (auto simp: mmap_update_def distinct_update)

definition adv_start :: ('b, 'c, 'd :: timestamp, 't, 'e) args ⇒
  ('b, 'c, 'd, 't, 'e) window ⇒ ('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) ⇒ (case run_sub si of Some (si', bs) ⇒
    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') ⇒
      (case tstp' of Some (ts, tp) ⇒ (q', tstp') | None ⇒ (q', tstp_cur))
      | None ⇒ (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 ≤ 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 −
  have inv_st: ⋀q bs. case Mapping.lookup st (q, bs) of None ⇒ True | Some v ⇒ step q bs = v
    using valid_before by (auto simp add: valid_s_def)
  have keys_s: mmap_keys s = {q. (∃ l ≤ u. steps step rho init (l, i) = q)}
    using valid_before by (auto simp add: valid_s_def)
  have fin_keys_s: finite (mmap_keys s)
    using valid_before by (auto simp add: valid_s_def)
  have lookup_s: ⋀q q' tstp. mmap_lookup s q = Some (q', tstp) ⟹
    steps step rho q (i, j) = q' ∧ tstp = sup_acc step accept rho q i j
    using valid_before Mapping_keys_intro
    by (auto simp add: valid_s_def) (smt case_prodD option.simps(5))+
  have drop_cur_i: ⋀x. x ∈ mmap_keys s ⟹ drop_cur i (the (mmap_lookup s x)) =
    (steps step rho (step x (bs_at rho i)) (i + 1, j),
    sup_acc step accept rho (step x (bs_at rho i)) (i + 1) j)
    proof −
      fix x
      assume assms: x ∈ mmap_keys s
      obtain q tstp where q_def: mmap_lookup s x = Some (q, tstp)
        using assms(1) by (auto dest: Mapping_keys_dest)
      have q_q': q = steps step rho (step x (bs_at rho i)) (i + 1, j)
        tstp = sup_acc step accept rho x i j
        using lookup_s[OF q_def] steps_split[OF i_lt_j] assms(1) by auto
      show drop_cur i (the (mmap_lookup s x)) =
        (steps step rho (step x (bs_at rho i)) (i + 1, j),
        sup_acc step accept rho (step x (bs_at rho i)) (i + 1) j)
        using q_def sup_acc_None[OF i_lt_j, of step accept rho]
          sup_acc_i[OF i_lt_j, of step accept rho] sup_acc_l[OF i_lt_j, of _ step accept rho]
        unfolding q_q'
        by (auto simp add: drop_cur_def split: option.splits)
    qed
  have valid_drop_cur: ⋀x x'. x ∈ mmap_keys s ⟹ x' ∈ mmap_keys s ⟹
    step x (bs_at rho i) = step x' (bs_at rho i) ⟹ drop_cur i (the (mmap_lookup s x)) =
    drop_cur i (the (mmap_lookup s x'))
    using drop_cur_i by auto
  then have keys_idem: keys_idem step i b s
    unfolding keys_idem_def b_def
    by blast
  have distinct: distinct (map fst s)
    using valid_before
    by (auto simp: valid_s_def)
  have (λq. step q (bs_at rho i)) ‘ {q. ∃ l≤u. steps step rho init (l, i) = q} =
    {q. ∃ l≤u. steps step rho init (l, i + 1) = q}
    using steps_app[of _ i step rho init] u_le_i
    by auto
  then have keys_s': mmap_keys s' = {q. ∃ l≤u. steps step rho init (l, i + 1) = q}
    using adv_d_keys[OF _ distinct adv_d] inv_st
    unfolding keys_s b_def
    by auto
  have lookup_s': ⋀q q' tstp. mmap_lookup s' q = Some (q', tstp) ⟹
    steps step rho q (i + 1, j) = q' ∧ tstp = sup_acc step accept rho q (i + 1) j
  proof −
```

```
    fix q q' tstp
    assume assm: mmap_lookup s' q = Some (q', tstp)
    obtain x where wit: x ∈ mmap_keys s step x (bs_at rho i) = q
      using assm lookup_adv_d_None[OF _ distinct adv_d] inv_st
      by (fastforce simp: b_def)
    have lookup_s'_q: mmap_lookup s' q = Some (drop_cur i (the (mmap_lookup s x)))
      using lookup_adv_d_Some[OF _ distinct keys_idem wit[folded b_def] adv_d] inv_st
      by auto
    then show steps step rho q (i + 1, j) = q' ∧ tstp = sup_acc step accept rho q (i + 1) j
      using assm
      by (simp add: drop_cur_i wit)
  qed
  have distinct (map fst s')
    using mmap_fold_distinct[OF distinct] adv_d_mmap_fold[OF inv_st adv_d[unfolded adv_d_def]]
    unfolding adv_d_def mmap_map_fst
    by auto
  then show valid_s init step st' accept rho u (i + 1) j s'
    unfolding valid_s_def
    using keys_s' lookup_s' u_le_i inv_st adv_d[unfolded adv_d_def]
      adv_d_mmap_fold[OF inv_st adv_d[unfolded adv_d_def]]
    by (auto split: option.splits dest: Mapping_keys_dest)
qed

lemma mmap_lookup_update':
  mmap_lookup (mmap_update k v kvs) z = (if k = z then Some v else mmap_lookup kvs z)
  unfolding mmap_lookup_def mmap_update_def
  by (auto simp add: update_conv')

lemma mmap_keys_update: mmap_keys (mmap_update k v kvs) = mmap_keys kvs ∪ {k}
  by (induction kvs) (auto simp: mmap_keys_def mmap_update_def)

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 −
  define init where init = w_init args
  define step where step = w_step args
  define accept where accept = w_accept args
  define run_t where run_t = w_run_t args
  define run_sub where run_sub = w_run_sub args
  define st where st = w_st w
  define ac where ac = w_ac w
  define i where i = w_i w
  define ti where ti = w_ti w
  define si where si = w_si w
  define j where j = w_j w
  define tj where tj = w_tj w
  define sj where sj = w_sj w
  define s where s = w_s w
  define e where e = w_e w
  have valid_before: 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 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)
    ∀ 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
    using assms(1)
    unfolding valid_window_def valid_s_def Let_def init_def step_def accept_def run_t_def
```

```
    run_sub_def st_def ac_def i_def ti_def si_def j_def tj_def sj_def s_def e_def
  by auto
have distinct_before: distinct (map fst s) distinct (map fst e)
  using valid_before
  by (auto simp: valid_s_def)
note i_lt_j = assms(2)[folded i_def j_def]
obtain ti' si' t b where tb_def: run_t ti = Some (ti', t)
  run_sub si = Some (si', b)
  reaches_on run_t ti' (drop (Suc i) (map fst rho)) tj
  reaches_on run_sub si' (drop (Suc i) (map snd rho)) sj
  t = ts_at rho i b = bs_at rho i
  using valid_before i_lt_j
  apply (auto simp: ts_at_def bs_at_def run_t_def[symmetric] run_sub_def[symmetric]
      elim!: reaches_on.cases[of run_t ti drop i (map fst rho) tj]
      reaches_on.cases[of run_sub si drop i (map snd rho) sj])
  by (metis Cons_nth_drop_Suc length_map list.inject nth_map)
have reaches_on_si': reaches_on run_sub sub (take (Suc i) (map snd rho)) si'
  using valid_before tb_def(2,3,4) i_lt_j reaches_on_app tb_def(1)
 by (auto simp: run_sub_def sub_def bs_at_def take_Suc_conv_app_nth reaches_on_app tb_def(6))
have reaches_on_ti': reaches_on run_t t0 (take (Suc i) (map fst rho)) ti'
  using valid_before tb_def(2,3,4) i_lt_j reaches_on_app tb_def(1)
  by (auto simp: run_t_def ts_at_def take_Suc_conv_app_nth reaches_on_app tb_def(5))
define e' where e' = mmap_update (fst (the (mmap_lookup s init))) t e
obtain st' s' where s'_def: adv_d step st i b s = (s', st')
  by (metis old.prod.exhaust)
obtain st_cur ac_cur i_cur ti_cur si_cur q_cur s_cur tstp_cur where loop_def:
  (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)
  by (cases while (loop_cond j) (loop_body step accept run_t run_sub)
      (st', ac, Suc i, ti', si', init, s', None)) auto
define s'' where s'' = mmap_update init (case mmap_lookup s_cur q_cur of
  Some (q', tstp') ⇒ (case tstp' of Some (ts, tp) ⇒ (q', tstp') | None ⇒ (q', tstp_cur))
  | None ⇒ (q_cur, tstp_cur)) s'
have i_le_j: i ≤ j
  using i_lt_j by auto
have length_rho: length rho = j
  using valid_before by auto
have lookup_s: ⋀q q' tstp. mmap_lookup s q = Some (q', tstp) ⟹
  steps step rho q (i, j) = q' ∧ tstp = sup_acc step accept rho q i j
  using valid_before Mapping_keys_intro
  by (auto simp: valid_s_def) (smt case_prodD option.simps(5))+
have init_in_keys_s: init ∈ mmap_keys s
  using valid_before by (auto simp add: valid_s_def)
then have run_init_i_j: steps step rho init (i, j) = fst (the (mmap_lookup s init))
  using lookup_s by (auto dest: Mapping_keys_dest)
have lookup_e: ⋀q. mmap_lookup e q = sup_leadsto init step rho i j q
  using valid_before by auto
have lookup_e': ⋀q. mmap_lookup e' q = sup_leadsto init step rho (i + 1) j q
proof −
  fix q
  show mmap_lookup e' q = sup_leadsto init step rho (i + 1) j q
  proof (cases steps step rho init (i, j) = q)
    case True
    have Max {l. l < Suc i ∧ steps step rho init (l, j) = steps step rho init (i, j)} = i
      by (rule iffD2[OF Max_eq_iff]) auto
    then have sup_leadsto init step rho (i + 1) j q = Some (ts_at rho i)
      by (auto simp add: sup_leadsto_def True)
```

```
    then show ?thesis
      unfolding e'_def using run_init_i_j tb_def
      by (auto simp add: mmap_lookup_update' True)
  next
    case False
    show ?thesis
      using run_init_i_j sup_leadsto_idle[OF i_lt_j False] lookup_e[of q] False
      by (auto simp add: e'_def mmap_lookup_update')
  qed
qed
have reach_split: {q. ∃ l≤i + 1. steps step rho init (l, i + 1) = q} =
  {q. ∃ l≤i. steps step rho init (l, i + 1) = q} ∪ {init}
  using le_Suc_eq by auto
have valid_s_i: valid_s init step st accept rho i i j s
  using valid_before by auto
have valid_s'_Suc_i: valid_s init step st' accept rho i (i + 1) j s'
  using valid_adv_d[OF valid_s_i order.refl i_lt_j, OF tb_def(6) s'_def] unfolding s'_def .
have loop: loop_inv init step accept args t0 sub rho i j tj sj
  (st_cur, ac_cur, i_cur, ti_cur, si_cur, q_cur, s_cur, tstp_cur) ∧
  ¬loop_cond j (st_cur, ac_cur, i_cur, ti_cur, si_cur, q_cur, s_cur, tstp_cur)
  unfolding loop_def
proof (rule while_rule_lemma[of loop_inv init step accept args t0 sub rho i j tj sj
      loop_cond j loop_body step accept run_t run_sub
      λs. loop_inv init step accept args t0 sub rho i j tj sj s ∧ ¬ loop_cond j s])
  show loop_inv init step accept args t0 sub rho i j tj sj
    (st', ac, Suc i, ti', si', init, s', None)
    unfolding loop_inv_def
    using i_lt_j valid_s'_Suc_i sup_acc_same[of step accept rho]
      length_rho reaches_on_si' reaches_on_ti' tb_def(3,4) valid_before(4)
    by (auto simp: run_t_def run_sub_def split: prod.splits)
next
  have {(t, s). loop_inv init step accept args t0 sub rho i j tj sj s ∧
    loop_cond j s ∧ t = loop_body step accept run_t run_sub s} ⊆
    measure (λ(st, ac, i_cur, ti, si, q, s, tstp). j − i_cur)
    unfolding loop_inv_def loop_cond_def loop_body_def
    apply (auto simp: run_t_def run_sub_def split: option.splits)
    apply (metis drop_eq_Nil length_map not_less option.distinct(1) reaches_on.simps)
    apply (metis (no_types, lifting) drop_eq_Nil length_map not_less option.distinct(1)
        reaches_on.simps)
    apply (auto split: prod.splits)
    done
  then show wf {(t, s). loop_inv init step accept args t0 sub rho i j tj sj s ∧
    loop_cond j s ∧ t = loop_body step accept run_t run_sub s}
    using wf_measure wf_subset by auto
next
  fix state
  assume assms: loop_inv init step accept args t0 sub rho i j tj sj state
    loop_cond j state
  obtain st_cur ac_cur i_cur ti_cur si_cur q_cur s_cur tstp_cur
    where state_def: state = (st_cur, ac_cur, i_cur, ti_cur, si_cur, q_cur, s_cur, tstp_cur)
    by (cases state) auto
  obtain ti'_cur si'_cur t_cur b_cur where tb_cur_def: run_t ti_cur = Some (ti'_cur, t_cur)
    run_sub si_cur = Some (si'_cur, b_cur)
    reaches_on run_t ti'_cur (drop (Suc i_cur) (map fst rho)) tj
    reaches_on run_sub si'_cur (drop (Suc i_cur) (map snd rho)) sj
    t_cur = ts_at rho i_cur b_cur = bs_at rho i_cur
    using assms
    unfolding loop_inv_def loop_cond_def state_def
```

apply (*auto simp*: *ts_at_def bs_at_def run_t_def*[*symmetric*] *run_sub_def*[*symmetric*]
elim!: *reaches_on.cases*[*of run_t ti_cur drop i_cur* (*map fst rho*) *tj*]
reaches_on.cases[*of run_sub si_cur drop i_cur* (*map snd rho*) *sj*])
by (*metis Cons_nth_drop_Suc length_map list.inject nth_map*)
obtain *s′_cur st′_cur* **where** *s′_cur_def*: *adv_d step st_cur i_cur b_cur s_cur* = (*s′_cur*, *st′_cur*)
by *fastforce*
have *valid_s′_cur*: *valid_s init step st′_cur accept rho i* (*i_cur* + *1*) *j s′_cur*
using *assms valid_adv_d*[*of init step st_cur accept rho*] *tb_cur_def*(*6*) *s′_cur_def*
unfolding *loop_inv_def loop_cond_def state_def*
by *auto*
obtain *q′ st′′_cur* **where** *q′_def*: *cstep step st′_cur q_cur b_cur* = (*q′*, *st′′_cur*)
by *fastforce*
obtain *β ac′_cur* **where** *b_def*: *cac accept ac_cur q′* = (*β*, *ac′_cur*)
by *fastforce*
have *step*: *q′* = *step q_cur b_cur* ⋀*q bs*. *case Mapping.lookup st′′_cur* (*q*, *bs*) *of*
None ⇒ *True* | *Some v* ⇒ *step q bs* = *v*
using *valid_s′_cur q′_def*
unfolding *valid_s_def*
by (*auto simp*: *cstep_def Let_def Mapping.lookup_update′ split*: *option.splits if_splits*)
have *accept*: *β* = *accept q′* ⋀*q*. *case Mapping.lookup ac′_cur q of*
None ⇒ *True* | *Some v* ⇒ *accept q* = *v*
using *assms b_def*
unfolding *loop_inv_def state_def*
by (*auto simp*: *cac_def Let_def Mapping.lookup_update′ split*: *option.splits if_splits*)
have *steps_q′*: *steps step rho init* (*i* + *1*, *Suc i_cur*) = *q′*
using *assms*
unfolding *loop_inv_def state_def*
by *auto* (*metis local.step*(*1*) *steps_appE tb_cur_def*(*6*))
have *b_acc*: *β* = *acc step accept rho init* (*i* + *1*, *Suc i_cur*)
unfolding *accept*(*1*) *acc_def steps_q′*
by (*auto simp*: *tb_cur_def*)
have *valid_s′′_cur*: *valid_s init step st′′_cur accept rho i* (*i_cur* + *1*) *j s′_cur*
using *valid_s′_cur step*(*2*)
unfolding *valid_s_def*
by *auto*
have *reaches_on_si′*: *reaches_on run_sub sub* (*take* (*Suc i_cur*) (*map snd rho*)) *si′_cur*
using *assms*
unfolding *loop_inv_def loop_cond_def state_def*
by (*auto simp*: *run_sub_def sub_def bs_at_def take_Suc_conv_app_nth reaches_on_app*
tb_cur_def(*2*,*4*,*6*))
(*metis bs_at_def reaches_on_app run_sub_def tb_cur_def*(*2*) *tb_cur_def*(*6*))
have *reaches_on_ti′*: *reaches_on run_t t0* (*take* (*Suc i_cur*) (*map fst rho*)) *ti′_cur*
using *assms*
unfolding *loop_inv_def loop_cond_def state_def*
by (*auto simp*: *run_t_def ts_at_def take_Suc_conv_app_nth reaches_on_app tb_cur_def*(*1*,*3*,*5*))
(*metis reaches_on_app run_t_def tb_cur_def*(*1*) *tb_cur_def*(*5*) *ts_at_def*)
have *reach_window args t0 sub rho* (*Suc i_cur*, *ti′_cur*, *si′_cur*, *j*, *tj*, *sj*)
using *reaches_on_si′ reaches_on_ti′ tb_cur_def*(*3*,*4*) *length_rho assms*(*2*)
unfolding *loop_cond_def state_def*
by (*auto simp*: *run_t_def run_sub_def*)
moreover have *steps step rho init* (*i* + *1*, *Suc i_cur*) = *q′*
using *assms steps_app*
unfolding *loop_inv_def state_def step*(*1*)
by (*auto simp*: *tb_cur_def*(*6*))
ultimately show *loop_inv init step accept args t0 sub rho i j tj sj*
(*loop_body step accept run_t run_sub state*)
using *assms accept*(*2*) *valid_s′′_cur sup_acc_ext*[*of _ _ step accept rho*]

```
    sup_acc_ext_idle[of _ _ step accept rho]
   unfolding loop_inv_def loop_body_def state_def
   by (auto simp: tb_cur_def(1,2,5) s'_cur_def q'_def b_def b_acc
     split: option.splits prod.splits)
qed auto
have valid_stac_cur: ∀ q bs. case Mapping.lookup st_cur (q, bs) of None ⇒ True
 | Some v ⇒ step q bs = v ∀ q. case Mapping.lookup ac_cur q of None ⇒ True
 | Some v ⇒ accept q = v
 using loop unfolding loop_inv_def valid_s_def
 by auto
have valid_s'': valid_s init step st_cur accept rho (i + 1) (i + 1) j s''
proof (cases mmap_lookup s_cur q_cur)
 case None
 then have added: steps step rho init (i + 1, j) = q_cur
   tstp_cur = sup_acc step accept rho init (i + 1) j
   using loop unfolding loop_inv_def loop_cond_def
   by (auto dest: Mapping_keys_dest)
 have s''_case: s'' = mmap_update init (q_cur, tstp_cur) s'
   unfolding s''_def using None by auto
 show ?thesis
   using valid_s'_Suc_i reach_split added mmap_update_distinct valid_stac_cur
   unfolding s''_case valid_s_def mmap_keys_update
   by (auto simp add: mmap_lookup_update' split: option.splits)
next
 case (Some p)
 obtain q' tstp' where p_def: p = (q', tstp')
   by (cases p) auto
 note lookup_s_cur = Some[unfolded p_def]
 have i_cur_in: i + 1 ≤ i_cur i_cur ≤ j
   using loop unfolding loop_inv_def by auto
 have q_cur_def: steps step rho init (i + 1, i_cur) = q_cur
   using loop unfolding loop_inv_def by auto
 have valid_s_cur: valid_s init step st_cur accept rho i i_cur j s_cur
   using loop unfolding loop_inv_def by auto
 have q'_steps: steps step rho q_cur (i_cur, j) = q'
   using Some valid_s_cur unfolding valid_s_def p_def
   by (auto intro: Mapping_keys_intro) (smt case_prodD option.simps(5))
 have tstp_cur: tstp_cur = sup_acc step accept rho init (i + 1) i_cur
   using loop unfolding loop_inv_def by auto
 have tstp': tstp' = sup_acc step accept rho q_cur i_cur j
   using loop Some unfolding loop_inv_def p_def valid_s_def
   by (auto intro: Mapping_keys_intro) (smt case_prodD option.simps(5))
 have added: steps step rho init (i + 1, j) = q'
   using steps_comp[OF i_cur_in q_cur_def q'_steps] .
 show ?thesis
 proof (cases tstp')
   case None
   have s''_case: s'' = mmap_update init (q', tstp_cur) s'
     unfolding s''_def lookup_s_cur None by auto
   have tstp_cur_opt: tstp_cur = sup_acc step accept rho init (i + 1) j
     using sup_acc_comp_None[OF i_cur_in, of step accept rho init, unfolded q_cur_def,
       OF tstp'[unfolded None, symmetric]]
     unfolding tstp_cur by auto
   then show ?thesis
     using valid_s'_Suc_i reach_split added mmap_update_distinct valid_stac_cur
     unfolding s''_case valid_s_def mmap_keys_update
     by (auto simp add: mmap_lookup_update' split: option.splits)
 next
```

```
    case (Some p')
    obtain ts tp where p'_def: p' = (ts, tp)
      by (cases p') auto
    have True: tp ≥ i_cur
      using sup_acc_SomeE[OF tstp'[unfolded Some p'_def, symmetric]] by auto
    have s''_case: s'' = mmap_update init (q', tstp') s'
      unfolding s''_def lookup_s_cur Some p'_def using True by auto
    have tstp'_opt: tstp' = sup_acc step accept rho init (i + 1) j
      using sup_acc_comp_Some_ge[OF i_cur_in True
          tstp'[unfolded Some p'_def q_cur_def[symmetric], symmetric]]
      unfolding tstp' by (auto simp: q_cur_def[symmetric])
    then show ?thesis
      using valid_s'_Suc_i reach_split added mmap_update_distinct valid_stac_cur
      unfolding s''_case valid_s_def mmap_keys_update
      by (auto simp add: mmap_lookup_update' split: option.splits)
  qed
 qed
 have distinct (map fst e')
  using mmap_update_distinct[OF distinct_before(2), unfolded e'_def]
  unfolding e'_def .
 then have valid_window args t0 sub rho
  (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'|))
  using i_lt_j lookup_e' valid_s'' length_rho tb_def(3,4) reaches_on_si' reaches_on_ti'
    valid_before[unfolded step_def accept_def] valid_stac_cur(2)[unfolded accept_def]
  by (auto simp: valid_window_def Let_def init_def step_def accept_def run_t_def
      run_sub_def st_def ac_def i_def ti_def si_def j_def tj_def sj_def s_def e_def)
 moreover have adv_start args w = 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'|)
  unfolding adv_start_def Let_def s''_def e'_def
  using tb_def(1,2) s'_def i_lt_j loop_def valid_before(3)
  by (auto simp: valid_window_def Let_def init_def step_def accept_def run_t_def
      run_sub_def st_def ac_def i_def ti_def si_def j_def tj_def sj_def s_def e_def
      split: prod.splits)
 ultimately show ?thesis
  by auto
qed

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
 using assms
 by (auto simp: adv_start_def Let_def valid_window_def split: option.splits prod.splits
    elim: reaches_on.cases)

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'
 using assms
 by (auto simp: adv_start_def Let_def valid_window_def split: option.splits prod.splits)

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
  by (induction r rule: state_cnt.induct) auto

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
  by (induction r phis rule: collect_subfmlas.induct) (auto split: if_splits)

lemma collect_subfmlas_atms: set (collect_subfmlas r phis) = set phis ∪ atms r
  by (induction r phis rule: collect_subfmlas.induct) (auto split: if_splits)

lemma collect_subfmlas_set: set (collect_subfmlas r phis) = set (collect_subfmlas r []) ∪ set phis
proof (induction r arbitrary: phis)
  case (Plus r1 r2)
  show ?case
    using Plus(1)[of phis] Plus(2)[of collect_subfmlas r1 phis]
      Plus(2)[of collect_subfmlas r1 []]
    by auto
next
  case (Times r1 r2)
  show ?case
    using Times(1)[of phis] Times(2)[of collect_subfmlas r1 phis]
      Times(2)[of collect_subfmlas r1 []]
    by auto
next
  case (Star r)
  show ?case
    using Star[of phis]
    by auto
qed auto

lemma collect_subfmlas_size: x ∈ set (collect_subfmlas r []) ⟹ size x < size r
proof (induction r)
  case (Plus r1 r2)
  then show ?case
    by (auto simp: collect_subfmlas_set[of r2 collect_subfmlas r1 []])
next
  case (Times r1 r2)
  then show ?case
    by (auto simp: collect_subfmlas_set[of r2 collect_subfmlas r1 []])
next
  case (Star r)
  then show ?case
    by fastforce
```

qed (*auto split*: *if_splits*)

lemma *collect_subfmlas_app*: $\exists$ *phis'*. *collect_subfmlas r phis* = *phis* @ *phis'*
by (*induction r phis rule*: *collect_subfmlas.induct*) *auto*

lemma *length_collect_subfmlas*: *length* (*collect_subfmlas r phis*) $\geq$ *length phis*
by (*induction r phis rule*: *collect_subfmlas.induct*) *auto*

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

lemma *pos_sound*: *pos a xs* = *Some i* $\Longrightarrow$ *i* < *length xs* $\wedge$ *xs* ! *i* = *a*
by (*induction a xs arbitrary*: *i rule*: *pos.induct*) (*auto split*: *if_splits option.splits*)

lemma *pos_complete*: *pos a xs* = *None* $\Longrightarrow$ *a* $\notin$ *set xs*
by (*induction a xs rule*: *pos.induct*) (*auto split*: *if_splits option.splits*)

fun *build_nfa_impl* :: (*'a*, *'b* :: *timestamp*) *regex* $\Rightarrow$ (*state* $\times$ *state* $\times$ (*'a*, *'b*) *formula list*) $\Rightarrow$
transition list **where**
build_nfa_impl (*Lookahead* φ) (*q0*, *qf*, *phis*) = (*case pos* φ *phis of*
Some n $\Rightarrow$ [*eps_trans qf n*]
| *None* $\Rightarrow$ [*eps_trans qf* (*length phis*)])
| *build_nfa_impl* (*Symbol* φ) (*q0*, *qf*, *phis*) = (*case pos* φ *phis of*
Some n $\Rightarrow$ [*eps_trans* (*Suc q0*) *n*, *symb_trans qf*]
| *None* $\Rightarrow$ [*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*
by (*induction r* (*q0*, *qf*, *phis*) *arbitrary*: *q0 qf phis rule*: *build_nfa_impl.induct*)
(*auto split*: *option.splits*)

lemma *build_nfa_impl_not_Nil*: *build_nfa_impl r* (*q0*, *qf*, *phis*) $\neq$ []
by (*induction r* (*q0*, *qf*, *phis*) *arbitrary*: *q0 qf phis rule*: *build_nfa_impl.induct*)
(*auto split*: *option.splits*)

lemma *build_nfa_impl_state_set*: *t* $\in$ *set* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) $\Longrightarrow$
state_set t $\subseteq$ {*q0*..<*q0* + *length* (*build_nfa_impl r* (*q0*, *qf*, *phis*))} $\cup$ {*qf*}
by (*induction r* (*q0*, *qf*, *phis*) *arbitrary*: *q0 qf phis t rule*: *build_nfa_impl.induct*)
(*fastforce simp add*: *build_nfa_impl_state_cnt state_cnt_pos build_nfa_impl_not_Nil*
split: *option.splits*)+

lemma *build_nfa_impl_fmla_set*: *t* $\in$ *set* (*build_nfa_impl r* (*q0*, *qf*, *phis*)) $\Longrightarrow$
n $\in$ *fmla_set t* $\Longrightarrow$ *n* < *length* (*collect_subfmlas r phis*)
proof (*induction r* (*q0*, *qf*, *phis*) *arbitrary*: *q0 qf phis t rule*: *build_nfa_impl.induct*)
case (*1* φ *q0 qf phis*)
then show *?case*

using *pos_sound pos_complete* **by** (*force split*: *option.splits*)
next
case (*2 φ q0 qf phis*)
then show *?case*
using *pos_sound pos_complete* **by** (*force split*: *option.splits*)
next
case (*3 r s q0 qf phis*)
then show *?case*
using *length_collect_subfmlas dual_order.strict_trans1* **by** *fastforce*
next
case (*4 r s q0 qf phis*)
then show *?case*
using *length_collect_subfmlas dual_order.strict_trans1* **by** *fastforce*
next
case (*5 r q0 qf phis*)
then show *?case*
using *length_collect_subfmlas dual_order.strict_trans1* **by** *fastforce*
qed

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 (*induct r arbitrary*: *q0 qf phis transs bss bs i rule*: *regex_induct*)
case (*Lookahead φ*)
have *qf_not_in_SQ*: *qf* ∉ *NFA.SQ q0 transs*
using *Lookahead* **unfolding** *IH_def* **by** (*auto simp*: *Let_def*)
have *qf_not_q0_Suc_q0*: *qf* ∉ {*q0*}
using *Lookahead* **unfolding** *IH_def*
by (*auto simp*: *NFA.SQ_def split*: *option.splits*)
have *transs_def*: *transs* = *build_nfa_impl* (*Lookahead φ*) (*q0*, *qf*, *phis*)
using *Lookahead*(*1*)
by (*auto simp*: *Let_def IH_def*)
interpret *base*: *nfa q0 qf transs*
apply *unfold_locales*
using *build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ*
unfolding *IH_def NFA.Q_def NFA.SQ_def transs_def*
by (*auto split*: *option.splits*)
define *n* **where** *n* ≡ *case pos φ phis of Some n* ⇒ *n* | _ ⇒ *length phis*
then have *collect*: *n* < *length* (*collect_subfmlas* (*Lookahead φ*) *phis*)
(*collect_subfmlas* (*Lookahead φ*) *phis*) ! *n* = *φ*
using *pos_sound pos_complete* **by** (*force split*: *option.splits*)+
have ⋀*cs q*. *base.step_eps cs q0 q* ⟷ *n* < *length cs* ∧ *cs* ! *n* ∧ *q* = *qf* ⋀*cs q*. ¬*base.step_eps cs qf q*
using *base.q0_sub_SQ qf_not_in_SQ*
by (*auto simp*: *NFA.step_eps_def transs_def n_def split*: *option.splits*)
then have *base_eps*: *base.step_eps_closure_set* {*q0*} *cs* = (*if n* < *length cs* ∧ *cs* ! *n then* {*q0*, *qf*}
else {*q0*}) **for** *cs*
using *NFA.step_eps_closure_set_unfold*[**where** *?X*={*qf*}]
using *NFA.step_eps_closure_set_step_id*[**where** *?R*={*q0*}]
using *NFA.step_eps_closure_set_step_id*[**where** *?R*={*qf*}]

```
    by auto
  have base_delta: base.delta {q0} cs = {} for cs
    unfolding NFA.delta_def NFA.step_symb_set_def base_eps
    by (auto simp: NFA.step_symb_def NFA.SQ_def transs_def split: option.splits)
  show ?case
  proof (cases bss)
    case Nil
    have sat: n < length bs ∧ bs ! n ⟷ sat φ i
      using Lookahead(1) collect
      by (auto simp: Let_def IH_def Nil)
    show ?thesis
      using qf_not_q0_Suc_q0
      unfolding NFA.run_accept_eps_def NFA.run_def NFA.accept_eps_def Nil
      by (auto simp: base_eps sat)
  next
    case bss_def: (Cons cs css)
    show ?thesis
      using NFA.run_accept_eps_empty
      unfolding NFA.run_accept_eps_def NFA.run_def NFA.accept_eps_def bss_def
      by (auto simp: bss_def base_delta)
  qed
next
  case (Symbol φ)
  have qf_not_in_SQ: qf ∉ NFA.SQ q0 transs
    using Symbol unfolding IH_def by (auto simp: Let_def)
  have qf_not_q0_Suc_q0: qf ∉ {q0, Suc q0}
    using Symbol unfolding IH_def
    by (auto simp: NFA.SQ_def split: option.splits)
  have transs_def: transs = build_nfa_impl (Symbol φ) (q0, qf, phis)
    using Symbol(1)
    by (auto simp: Let_def IH_def)
  interpret base: nfa q0 qf transs
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding IH_def NFA.Q_def NFA.SQ_def transs_def
    by (auto split: option.splits)
  define n where n ≡ case pos φ phis of Some n ⇒ n | _ ⇒ length phis
  then have collect: n < length (collect_subfmlas (Symbol φ) phis)
    (collect_subfmlas (Symbol φ) phis) ! n = φ
    using pos_sound pos_complete by (force split: option.splits)+
  have ⋀cs q. base.step_eps cs q0 q ⟷ n < length cs ∧ cs ! n ∧ q = Suc q0 ⋀cs q. ¬base.step_eps cs
(Suc q0) q
    using base.q0_sub_SQ
    by (auto simp: NFA.step_eps_def transs_def n_def split: option.splits)
  then have base_eps: base.step_eps_closure_set {q0} cs = (if n < length cs ∧ cs ! n then {q0, Suc
q0} else {q0}) for cs
    using NFA.step_eps_closure_set_unfold[where ?X={Suc q0}]
    using NFA.step_eps_closure_set_step_id[where ?R={q0}]
    using NFA.step_eps_closure_set_step_id[where ?R={Suc q0}]
    by auto
  have base_delta: base.delta {q0} cs = (if n < length cs ∧ cs ! n then {qf} else {}) for cs
    unfolding NFA.delta_def NFA.step_symb_set_def base_eps
    by (auto simp: NFA.step_symb_def NFA.SQ_def transs_def split: option.splits)
  show ?case
  proof (cases bss)
    case Nil
    show ?thesis
      using qf_not_q0_Suc_q0
```

```
    unfolding NFA.run_accept_eps_def NFA.run_def NFA.accept_eps_def Nil
    by (auto simp: base_eps)
  next
    case bss_def: (Cons cs css)
    have sat: n < length cs ∧ cs ! n ⟷ sat φ i
      using Symbol(1) collect
      by (auto simp: Let_def IH_def bss_def)
    show ?thesis
    proof (cases css)
      case Nil
      show ?thesis
        unfolding NFA.run_accept_eps_def NFA.run_def NFA.accept_eps_def bss_def Nil
        by (auto simp: base_delta sat NFA.step_eps_closure_set_def NFA.step_eps_closure_def)
    next
      case css_def: (Cons ds dss)
      have base.delta {} ds = {} base.delta {qf} ds = {}
        using base.step_eps_closure_qf qf_not_in_SQ step_symb_dest
         by (fastforce simp: NFA.delta_def NFA.step_eps_closure_set_def NFA.step_symb_set_def)+
      then show ?thesis
        using NFA.run_accept_eps_empty
        unfolding NFA.run_accept_eps_def NFA.run_def NFA.accept_eps_def bss_def css_def
        by (auto simp: base_delta)
    qed
  qed
next
  case (Plus r s)
  obtain phis' where collect: collect_subfmlas (Plus r s) phis =
    collect_subfmlas r phis @ phis'
    using collect_subfmlas_app by auto
  have qf_not_in_SQ: qf ∉ NFA.SQ q0 (build_nfa_impl (Plus r s) (q0, qf, phis))
    using Plus unfolding IH_def by auto
  interpret base: nfa q0 qf build_nfa_impl (Plus r s) (q0, qf, phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt by fast+
  interpret left: nfa q0 + 1 qf build_nfa_impl r (q0 + 1, qf, phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
    by fastforce+
  interpret right: nfa q0 + 1 + state_cnt r qf
    build_nfa_impl s (q0 + 1 + state_cnt r, qf, collect_subfmlas r phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
    by fastforce+
  from Plus(3) have IH r (q0 + 1) qf phis (build_nfa_impl r (q0 + 1, qf, phis)) bss bs i
    unfolding Let_def IH_def collect
    using left.qf_not_in_SQ
    by (auto simp: nth_append)
  then have left_IH: left.run_accept_eps {q0 + 1} bss bs ⟷
    (i, i + length bss) ∈ match r
    using Plus(1) build_nfa_impl_state_cnt
    by auto
  have IH s (q0 + 1 + state_cnt r) qf (collect_subfmlas r phis)
    (build_nfa_impl s (q0 + 1 + state_cnt r, qf, collect_subfmlas r phis)) bss bs i
    using right.qf_not_in_SQ IH_def Plus
    by (auto simp: Let_def)
```

```
  then have right_IH: right.run_accept_eps {q0 + 1 + state_cnt r} bss bs ⟷
    (i, i + length bss) ∈ match s
    using Plus(2) build_nfa_impl_state_cnt
    by auto
  interpret cong: nfa_cong_Plus q0 q0 + 1 q0 + 1 + state_cnt r qf qf qf
    build_nfa_impl (Plus r s) (q0, qf, phis) build_nfa_impl r (q0 + 1, qf, phis)
    build_nfa_impl s (q0 + 1 + state_cnt r, qf, collect_subfmlas r phis)
    apply unfold_locales
    unfolding NFA.SQ_def build_nfa_impl_state_cnt
      NFA.step_eps_def NFA.step_symb_def
    by (auto simp add: nth_append build_nfa_impl_state_cnt)
  show ?case
    using cong.run_accept_eps_cong left_IH right_IH Plus
    by (auto simp: Let_def IH_def)
next
  case (Times r s)
  obtain phis' where collect: collect_subfmlas (Times r s) phis =
    collect_subfmlas r phis @ phis'
    using collect_subfmlas_app by auto
  have transs_def: transs = build_nfa_impl (Times r s) (q0, qf, phis)
    using Times unfolding IH_def by (auto simp: Let_def)
  have qf_not_in_SQ: qf ∉ NFA.SQ q0 (build_nfa_impl (Times r s) (q0, qf, phis))
    using Times unfolding IH_def by auto
  interpret base: nfa q0 qf build_nfa_impl (Times r s) (q0, qf, phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt by fast+
  interpret left: nfa q0 q0 + state_cnt r build_nfa_impl r (q0, q0 + state_cnt r, phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
    by fastforce+
  interpret right: nfa q0 + state_cnt r qf
    build_nfa_impl s (q0 + state_cnt r, qf, collect_subfmlas r phis)
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
    by fastforce+
  from Times(3) have left_IH: IH r q0 (q0 + state_cnt r) phis
    (build_nfa_impl r (q0 , q0 + state_cnt r, phis)) bss bs i
    unfolding Let_def IH_def collect
    using left.qf_not_in_SQ
    by (auto simp: nth_append)
  from Times(3) have left_IH_take: ⋀n. n < length bss ⟹
    IH r q0 (q0 + state_cnt r) phis
    (build_nfa_impl r (q0, q0 + state_cnt r, phis)) (take n bss) (hd (drop n bss)) i
    unfolding Let_def IH_def collect
    using left.qf_not_in_SQ
    apply (auto simp: nth_append min_absorb2 hd_drop_conv_nth)
     apply (meson in_set_takeD le_add1 le_trans)
    by (meson le_add1 le_trans nth_mem)
  have left_IH_match: left.run_accept_eps {q0} bss bs ⟷
    (i, i + length bss) ∈ match r
    using Times(1) build_nfa_impl_state_cnt left_IH
    by auto
  have left_IH_match_take: ⋀n. n < length bss ⟹
    left.run_accept_eps {q0} (take n bss) (hd (drop n bss)) ⟷ (i, i + n) ∈ match r
    using Times(1) build_nfa_impl_state_cnt left_IH_take
```

```
    by (fastforce simp add: nth_append min_absorb2)
  have IH s (q0 + state_cnt r) qf (collect_subfmlas r phis)
    (build_nfa_impl s (q0 + state_cnt r, qf, collect_subfmlas r phis)) bss bs i
    using right.qf_not_in_SQ IH_def Times
    by (auto simp: Let_def)
  then have right_IH: ⋀n. n ≤ length bss ⟹ IH s (q0 + state_cnt r) qf (collect_subfmlas r phis)
    (build_nfa_impl s (q0 + state_cnt r, qf, collect_subfmlas r phis)) (drop n bss) bs (i + n)
    unfolding Let_def IH_def
    by (auto simp: nth_append add.assoc) (meson in_set_dropD)
  have right_IH_match: ⋀n. n ≤ length bss ⟹
    right.run_accept_eps {q0 + state_cnt r} (drop n bss) bs ⟷ (i + n, i + length bss) ∈ match s
    using Times(2)[OF right_IH] build_nfa_impl_state_cnt
    by (auto simp: IH_def)
  interpret cong: nfa_cong_Times q0 q0 + state_cnt r qf
    build_nfa_impl (Times r s) (q0, qf, phis)
    build_nfa_impl r (q0, q0 + state_cnt r, phis)
    build_nfa_impl s (q0 + state_cnt r, qf, collect_subfmlas r phis)
    apply unfold_locales
    using NFA.Q_def NFA.SQ_def NFA.step_eps_def NFA.step_symb_def build_nfa_impl_state_set
    by (fastforce simp add: nth_append build_nfa_impl_state_cnt build_nfa_impl_not_Nil
        state_cnt_pos)+
  have right_IH_Nil: right.run_accept_eps {q0 + state_cnt r} [] bs ⟷
    (i + length bss, i + length bss) ∈ match s
    using right_IH_match
    by fastforce
  show ?case
    unfolding match_Times transs_def cong.run_accept_eps_cong left_IH_match right_IH_Nil
    using left_IH_match_take right_IH_match less_imp_le_nat le_eq_less_or_eq
    by auto
next
  case (Star r)
  then show ?case
  proof (induction length bss arbitrary: bss bs i rule: nat_less_induct)
    case 1
    have transs_def: transs = build_nfa_impl (Star r) (q0, qf, phis)
      using 1 unfolding IH_def by (auto simp: Let_def)
    have qf_not_in_SQ: qf ∉ NFA.SQ q0 (build_nfa_impl (Star r) (q0, qf, phis))
      using 1 unfolding IH_def by auto
    interpret base: nfa q0 qf build_nfa_impl (Star r) (q0, qf, phis)
      apply unfold_locales
      using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
      unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
      by fast+
    interpret left: nfa q0 + 1 q0 build_nfa_impl r (q0 + 1, q0, phis)
      apply unfold_locales
      using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
      unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt
      by fastforce+
    from 1(3) have left_IH: IH r (q0 + 1) q0 phis (build_nfa_impl r (q0 + 1, q0, phis)) bss bs i
      using left.qf_not_in_SQ
      unfolding Let_def IH_def
      by (auto simp add: nth_append)
    from 1(3) have left_IH_take: ⋀n. n < length bss ⟹
      IH r (q0 + 1) q0 phis (build_nfa_impl r (q0 + 1, q0, phis)) (take n bss) (hd (drop n bss)) i
      using left.qf_not_in_SQ
      unfolding Let_def IH_def
      by (auto simp add: nth_append min_absorb2 hd_drop_conv_nth) (meson in_set_takeD)
    have left_IH_match: left.run_accept_eps {q0 + 1} bss bs ⟷
```

```
  (i, i + length bss) ∈ match r
  using 1(2) left_IH
  unfolding build_nfa_impl_state_cnt NFA.SQ_def
  by auto
have left_IH_match_take: ⋀n. n < length bss ⟹
  left.run_accept_eps {q0 + 1} (take n bss) (hd (drop n bss)) ⟷
  (i, i + n) ∈ match r
  using 1(2) left_IH_take
  unfolding build_nfa_impl_state_cnt NFA.SQ_def
  by (fastforce simp add: nth_append min_absorb2)
interpret cong: nfa_cong_Star q0 q0 + 1 qf
  build_nfa_impl (Star r) (q0, qf, phis)
  build_nfa_impl r (q0 + 1, q0, phis)
  apply unfold_locales
  unfolding NFA.SQ_def build_nfa_impl_state_cnt NFA.step_eps_def NFA.step_symb_def
  by (auto simp add: nth_append build_nfa_impl_state_cnt)
show ?case
  using cong.run_accept_eps_Nil
proof (cases bss)
  case Nil
  show ?thesis
    unfolding transs_def Nil
    using cong.run_accept_eps_Nil run_Nil run_accept_eps_Nil
    by auto
next
  case (Cons cs css)
  have aux: ⋀n j x P. n < x ⟹ j < x − n ⟹ (∀j < Suc x. P j) ⟹ P (Suc (n + j))
    by auto
  from 1(3) have star_IH: ⋀n. n < length css ⟹
    IH (Star r) q0 qf phis transs (drop n css) bs (i + n + 1)
    unfolding Cons Let_def IH_def
    using aux[of _ _ _ λj. ∀k<length (collect_subfmlas r phis).
      (cs # css) ! j ! k = sat (collect_subfmlas r phis ! k) (i + j)]
    by (auto simp add: nth_append add.assoc dest: in_set_dropD)
  have IH_inst: ⋀xs i. length xs ≤ length css ⟹ IH (Star r) q0 qf phis transs xs bs i ⟶
    (base.run_accept_eps {q0} xs bs ⟷ (i, i + length xs) ∈ match (Star r))
    using 1
    unfolding Cons
    by (auto simp add: nth_append less_Suc_eq_le transs_def)
  have ⋀n. n < length css ⟹ base.run_accept_eps {q0} (drop n css) bs ⟷
    (i + n + 1, i + length (cs # css)) ∈ match (Star r)
  proof −
    fix n
    assume assm: n < length css
    then show base.run_accept_eps {q0} (drop n css) bs ⟷
      (i + n + 1, i + length (cs # css)) ∈ match (Star r)
      using IH_inst[of drop n css i + n + 1] star_IH
      by (auto simp add: nth_append)
  qed
  then show ?thesis
    using match_Star length_Cons Cons cong.run_accept_eps_cong_Cons
    using cong.run_accept_eps_Nil left_IH_match left_IH_match_take
    apply (auto simp add: Cons transs_def)
     apply (metis Suc_less_eq add_Suc_right drop_Suc_Cons less_imp_le_nat take_Suc_Cons)
    apply (metis Suc_less_eq add_Suc_right drop_Suc_Cons le_eq_less_or_eq lessThan_iff
        take_Suc_Cons)
    done
qed
```

```
  qed
qed

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
  using assms
proof (induction r arbitrary: q0 qf phis transs q)
  case (Symbol φ)
  have qf_not_in_SQ: qf ∉ NFA.SQ q0 transs
    using Symbol unfolding IH_def by (auto simp: Let_def)
  have qf_not_q0_Suc_q0: qf ∉ {q0, Suc q0}
    using Symbol unfolding IH_def
    by (auto simp: NFA.SQ_def split: option.splits)
  have transs_def: transs = build_nfa_impl (Symbol φ) (q0, qf, phis)
    using Symbol(2)
    by (auto simp: Let_def IH_def)
  interpret base: nfa q0 qf transs
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding IH_def NFA.Q_def NFA.SQ_def transs_def
    by (auto split: option.splits)
  define n where n ≡ case pos φ phis of Some n ⇒ n | _ ⇒ length phis
  then have collect: n < length (collect_subfmlas (Symbol φ) phis)
    (collect_subfmlas (Symbol φ) phis) ! n = φ
    using pos_sound pos_complete by (force split: option.splits)+
  have SQD: q ∈ NFA.SQ q0 transs ⟹ q = q0 ∨ q = Suc q0 for q
    by (auto simp: NFA.SQ_def transs_def split: option.splits)
  have ¬base.step_eps cs q qf if q ∈ NFA.SQ q0 transs for cs q
    using SQD[OF that] qf_not_q0_Suc_q0
    by (auto simp: NFA.step_eps_def transs_def split: option.splits transition.splits)
  then show ?case
    using Symbol(3)
    by (auto simp: NFA.step_eps_closure_def) (metis rtranclp.simps step_eps_dest)
next
  case (Plus r s)
  have transs_def: transs = build_nfa_impl (Plus r s) (q0, qf, phis)
    using Plus(4)
    by (auto simp: IH_def Let_def)
  define ts_l where ts_l = build_nfa_impl r (q0 + 1, qf, phis)
  define ts_r where ts_r = build_nfa_impl s (q0 + 1 + state_cnt r, qf, collect_subfmlas r phis)
  have len_ts: length ts_l = state_cnt r length ts_r = state_cnt s length transs = Suc (state_cnt r +
state_cnt s)
    by (auto simp: ts_l_def ts_r_def transs_def build_nfa_impl_state_cnt)
  have transs_eq: transs = split_trans (q0 + 1) (q0 + 1 + state_cnt r) # ts_l @ ts_r
    by (auto simp: transs_def ts_l_def ts_r_def)
  have ts_nonempty: ts_l = [] ⟹ False ts_r = [] ⟹ False
    by (auto simp: ts_l_def ts_r_def build_nfa_impl_not_Nil)
  obtain phis' where collect: collect_subfmlas (Plus r s) phis = collect_subfmlas r phis @ phis'
    using collect_subfmlas_app by auto
  have qf_not_in_SQ: qf ∉ NFA.SQ q0 (build_nfa_impl (Plus r s) (q0, qf, phis))
    using Plus unfolding IH_def by auto
  interpret base: nfa q0 qf transs
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt transs_def by fast+
  interpret left: nfa Suc q0 qf ts_l
    apply unfold_locales
```

```
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt ts_l_def
    by fastforce+
  interpret right: nfa Suc (q0 + state_cnt r) qf ts_r
    apply unfold_locales
    using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
    unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt ts_r_def
    by fastforce+
  interpret cong: nfa_cong_Plus q0 Suc q0 Suc (q0 + state_cnt r) qf qf qf transs ts_l ts_r
    apply unfold_locales
    unfolding NFA.SQ_def build_nfa_impl_state_cnt
      NFA.step_eps_def NFA.step_symb_def transs_def ts_l_def ts_r_def
    by (auto simp add: nth_append build_nfa_impl_state_cnt)
  have IH s (Suc (q0 + state_cnt r)) qf (collect_subfmlas r phis) ts_r bss bs i
    using right.qf_not_in_SQ IH_def Plus
    by (auto simp: Let_def ts_r_def)
  then have case_right: base.step_eps_closure [] q qf if base.step_eps_closure bs q qf q ∈ right.Q for q
   using cong.right.eps_nfa'_step_eps_closure[OF that] Plus(2,3) cong.right.nfa'_eps_step_eps_closure[OF
_ that(2)]
    by auto
  from Plus(4) have IH r (Suc q0) qf phis ts_l bss bs i
    using left.qf_not_in_SQ
    unfolding Let_def IH_def collect ts_l_def
    by (auto simp: nth_append)
  then have case_left: base.step_eps_closure [] q qf if base.step_eps_closure bs q qf q ∈ left.Q for q
     using cong.eps_nfa'_step_eps_closure[OF that] Plus(1,3) cong.nfa'_eps_step_eps_closure[OF _
that(2)]
    by auto
  have q = q0 ∨ q ∈ left.Q ∨ q ∈ right.Q
    using Plus(5)
    by (auto simp: NFA.Q_def NFA.SQ_def len_ts dest!: NFA.step_eps_closure_dest)
  moreover have ?case if q_q0: q = q0
  proof −
    have q0 ≠ qf
      using qf_not_in_SQ
      by (auto simp: NFA.SQ_def)
    then obtain q' where q'_def: base.step_eps bs q q' base.step_eps_closure bs q' qf
      using Plus(5)
      by (auto simp: q_q0 NFA.step_eps_closure_def elim: converse_rtranclpE)
    have fst_step_eps: base.step_eps [] q q'
      using q'_def(1)
      by (auto simp: q_q0 NFA.step_eps_def transs_eq)
    have q' ∈ left.Q ∨ q' ∈ right.Q
      using q'_def(1)
     by (auto simp: NFA.step_eps_def NFA.Q_def NFA.SQ_def q_q0 transs_eq dest: ts_nonempty split:
transition.splits)
    then show ?case
      using fst_step_eps case_left[OF q'_def(2)] case_right[OF q'_def(2)]
      by (auto simp: NFA.step_eps_closure_def)
  qed
  ultimately show ?case
    using Plus(5) case_left case_right
    by auto
next
  case (Times r s)
  obtain phis' where collect: collect_subfmlas (Times r s) phis =
    collect_subfmlas r phis @ phis'
    using collect_subfmlas_app by auto
```

have *transs_def*: *transs* = *build_nfa_impl* (*Times r s*) (*q0*, *qf*, *phis*)
using *Times* **unfolding** *IH_def* **by** (*auto simp*: *Let_def*)
define *ts_l* **where** *ts_l* = *build_nfa_impl r* (*q0*, *q0* + *state_cnt r*, *phis*)
define *ts_r* **where** *ts_r* = *build_nfa_impl s* (*q0* + *state_cnt r*, *qf*, *collect_subfmlas r phis*)
have *len_ts*: *length ts_l* = *state_cnt r length ts_r* = *state_cnt s length transs* = *state_cnt r* + *state_cnt s*
by (*auto simp*: *ts_l_def ts_r_def transs_def build_nfa_impl_state_cnt*)
have *transs_eq*: *transs* = *ts_l* @ *ts_r*
by (*auto simp*: *transs_def ts_l_def ts_r_def*)
have *ts_nonempty*: *ts_l* = [] ⟹ *False ts_r* = [] ⟹ *False*
by (*auto simp*: *ts_l_def ts_r_def build_nfa_impl_not_Nil*)
have *qf_not_in_SQ*: *qf* ∉ *NFA.SQ q0* (*build_nfa_impl* (*Times r s*) (*q0*, *qf*, *phis*))
using *Times* **unfolding** *IH_def* **by** *auto*
interpret *base*: *nfa q0 qf transs*
apply *unfold_locales*
using *build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ*
unfolding *NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt transs_def* **by** *fast+*
interpret *left*: *nfa q0 q0* + *state_cnt r ts_l*
apply *unfold_locales*
using *build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ*
unfolding *NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt ts_l_def*
by *fastforce+*
interpret *right*: *nfa q0* + *state_cnt r qf ts_r*
apply *unfold_locales*
using *build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ*
unfolding *NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt ts_r_def*
by *fastforce+*
interpret *cong*: *nfa_cong_Times q0 q0* + *state_cnt r qf transs ts_l ts_r*
apply *unfold_locales*
using *NFA.Q_def NFA.SQ_def NFA.step_eps_def NFA.step_symb_def build_nfa_impl_state_set*
by (*auto simp add*: *nth_append build_nfa_impl_state_cnt build_nfa_impl_not_Nil state_cnt_pos len_ts transs_eq*)
have *qf* ∉ *base.SQ*
using *Times*(*4*)
by (*auto simp*: *IH_def Let_def*)
then have *qf_left_Q*: *qf* ∈ *left.Q* ⟹ *False*
by (*auto simp*: *NFA.Q_def NFA.SQ_def len_ts state_cnt_pos*)
have *left_IH*: *IH r q0* (*q0* + *state_cnt r*) *phis ts_l bss bs i*
using *left.qf_not_in_SQ Times*
unfolding *Let_def IH_def collect*
by (*auto simp*: *nth_append ts_l_def*)
have *case_left*: *base.step_eps_closure* [] *q* (*q0* + *state_cnt r*) **if** *left.step_eps_closure bs q* (*q0* + *state_cnt r*) *q* ∈ *left.Q* **and** *wf*: *wf_regex r* **for** *q*
using *that*(*1*) *Times*(*1*)[*OF wf left_IH*] *cong.nfa'_step_eps_closure_cong*[*OF _ that*(*2*)]
by *auto*
have *left_IH*: *IH s* (*q0* + *state_cnt r*) *qf* (*collect_subfmlas r phis*) *ts_r bss bs i*
using *right.qf_not_in_SQ IH_def Times*
by (*auto simp*: *Let_def ts_r_def*)
then have *case_right*: *base.step_eps_closure* [] *q qf* **if** *base.step_eps_closure bs q qf q* ∈ *right.Q* **for** *q*
using *cong.right.eps_nfa'_step_eps_closure*[*OF that*] *Times*(*2*,*3*) *cong.right.nfa'_eps_step_eps_closure*[*OF _ that*(*2*)]
by *auto*
have *init_right*: *q0* + *state_cnt r* ∈ *right.Q*
by (*auto simp*: *NFA.Q_def NFA.SQ_def dest*: *ts_nonempty*)
{
assume *q_left_Q*: *q* ∈ *left.Q*
then have *split*: *left.step_eps_closure bs q* (*q0* + *state_cnt r*) *base.step_eps_closure bs* (*q0* + *state_cnt r*) *qf*

```
    using cong.eps_nfa'_step_eps_closure_cong[OF Times(5)]
    by (auto dest: qf_left_Q)
  have empty_IH: IH s (q0 + state_cnt r) qf (collect_subfmlas r phis) ts_r [] bs (i + length bss)
    using left_IH
    by (auto simp: IH_def Let_def ts_r_def)
  have right.step_eps_closure bs (q0 + state_cnt r) qf
    using cong.right.eps_nfa'_step_eps_closure[OF split(2) init_right]
    by auto
  then have right.run_accept_eps {q0 + state_cnt r} [] bs
      by (auto simp: NFA.run_accept_eps_def NFA.accept_eps_def NFA.step_eps_closure_set_def
NFA.run_def)
  then have wf: wf_regex r
    using nfa_correct[OF empty_IH] Times(3) match_refl_eps
    by auto
  have ?case
    using case_left[OF split(1) q_left_Q wf] case_right[OF split(2) init_right]
    by (auto simp: NFA.step_eps_closure_def)
 }
 moreover have q ∈ left.Q ∨ q ∈ right.Q
  using Times(5)
  by (auto simp: NFA.Q_def NFA.SQ_def transs_eq len_ts dest!: NFA.step_eps_closure_dest)
 ultimately show ?case
  using case_right[OF Times(5)]
  by auto
next
 case (Star r)
 have transs_def: transs = build_nfa_impl (Star r) (q0, qf, phis)
  using Star unfolding IH_def by (auto simp: Let_def)
 obtain ts_r where ts_r: transs = split_trans (q0 + 1) qf # ts_r ts_r = build_nfa_impl r (Suc q0,
q0, phis)
  using Star(3)
  by (auto simp: Let_def IH_def)
 have qf_not_in_SQ: qf ∉ NFA.SQ q0 transs
  using Star unfolding IH_def transs_def by auto
 interpret base: nfa q0 qf transs
  apply unfold_locales
  using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
  unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt transs_def
  by fast+
 interpret left: nfa Suc q0 q0 ts_r
  apply unfold_locales
  using build_nfa_impl_state_set build_nfa_impl_not_Nil qf_not_in_SQ
  unfolding NFA.Q_def NFA.SQ_def build_nfa_impl_state_cnt ts_r(2)
  by fastforce+
 interpret cong: nfa_cong_Star q0 Suc q0 qf transs ts_r
  apply unfold_locales
  unfolding NFA.SQ_def build_nfa_impl_state_cnt NFA.step_eps_def NFA.step_symb_def
  by (auto simp add: nth_append build_nfa_impl_state_cnt ts_r(1))
 have IH: wf_regex r IH r (Suc q0) q0 phis ts_r bss bs i
  using Star(2,3)
  by (auto simp: Let_def IH_def NFA.SQ_def ts_r(2))
 have step_eps_q'_qf: q' = q0 if base.step_eps bs q' qf for q'
 proof (rule ccontr)
  assume q' ≠ q0
  then have q' ∈ left.SQ
   using that
   by (auto simp: NFA.step_eps_def NFA.SQ_def ts_r(1))
  then have left.step_eps bs q' qf
```

```
    using cong.step_eps_cong_SQ that
    by simp
  then show False
    using qf_not_in_SQ
    by (metis NFA.Q_def UnE base.q0_sub_SQ cong.SQ_sub left.step_eps_closed subset_eq)
 qed
 show ?case
 proof (cases q = qf)
  case False
  then have base_q_q0: base.step_eps_closure bs q q0 base.step_eps bs q0 qf
    using Star(4) step_eps_q'_qf
    by (auto simp: NFA.step_eps_closure_def) (metis rtranclp.cases)+
  have base_Nil_q0_qf: base.step_eps [] q0 qf
    by (auto simp: NFA.step_eps_def NFA.SQ_def ts_r(1))
  have q_left_Q: q ∈ left.Q
    using base_q_q0
    by (auto simp: NFA.Q_def NFA.SQ_def ts_r(1) dest: step_eps_closure_dest)
  have left.step_eps_closure [] q q0
    using cong.eps_nfa'_step_eps_closure_cong[OF base_q_q0(1) q_left_Q] Star(1)[OF IH]
    by auto
  then show ?thesis
    using cong.nfa'_step_eps_closure_cong[OF _ q_left_Q] base_Nil_q0_qf
    by (auto simp: NFA.step_eps_closure_def) (meson rtranclp.rtrancl_into_rtrancl)
 qed (auto simp: NFA.step_eps_closure_def)
qed auto

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
 using step_eps_closure_set_empty_list[OF assms] step_eps_closure_set_mono'
 unfolding NFA.accept_eps_def NFA.accept_def
 by (fastforce simp: NFA.accept_eps_def NFA.accept_def NFA.step_eps_closure_set_def)

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
 unfolding NFA.run_accept_eps_def NFA.run_accept_def accept_eps_iff_accept[OF assms] ..

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 −
 have sub: {(t, s). pred_option P s ∧ pred_option' b s ∧ t = map_option' c s} ⊆
   map_prod Some Some ‘ {(t, s). P s ∧ b s ∧ Some t = c s} ∪ ({None} × (Some ‘ UNIV))
   by (auto simp: pred_option'_def map_option'_def split: option.splits)
     (smt (z3) case_prodI map_prod_imageI mem_Collect_eq not_Some_eq)
 show ?thesis
```

```
    apply (rule wf_subset[OF _ sub])
    apply (rule wf_union_compatible)
     apply (rule wf_map_prod_image)
      apply (fastforce simp: wf_def intro: assms)+
    done
qed

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}
  by (rule wf_subset[OF wf_while_break[OF assms]]) (auto simp: pred_option'_def split: option.splits)

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 −
  have aux: P t ⟹ b t ⟹ pred_option P (c t) for t
    using assms(1)
    by (cases c t) auto
  show ?thesis
    using assms aux
    by (auto simp: while_break_def pred_option'_def map_option'_def split: option.splits
      intro!: while_rule_lemma[where ?P=pred_option P and ?Q=pred_option Q and ?b=pred_option'
b and ?c=map_option' c, OF _ _ wf_while_break])
qed

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)
  unfolding while_break_def
 by (rule while_rule_lemma[where ?P=pred_option' P and ?Q=pred_option' Q and ?b=pred_option'
b and ?c=map_option' c, OF _ _ wf_while_break'])
     (force simp: pred_option'_def map_option'_def split: option.splits elim!: case_optionE)+

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)
  using reach_window_run_tj
  by (fastforce simp: valid_window_matchP_def simp del: reach_window.simps)

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)
```

```
  using reach_window_run_sj
  by (fastforce simp: valid_window_matchP_def simp del: reach_window.simps)

lemma valid_window_matchP_len_rho: valid_window_matchP I t0 sub rho i w ⟹ length rho = i
  by (auto simp: valid_window_matchP_def)

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)
  using assms
proof (induction qts arbitrary: ac)
  case (Cons a qts)
  obtain q t where qt_def: a = (q, t)
    by fastforce
  show ?case
  proof (cases time t)
    case True
    note time_t = True
    obtain β ac'' where ac''_def: cac accept ac q = (β, ac'')
      by fastforce
    have accept: β = accept q ⋀q. case Mapping.lookup ac'' q of None ⇒ True
      | Some v ⇒ accept q = v
      using ac''_def Cons(2)
      by (fastforce simp: cac_def Let_def Mapping.lookup_update' split: option.splits if_splits)+
    show ?thesis
    proof (cases β)
      case True
      then show ?thesis
        using accept(2) time_t Cons(4)
        by (auto simp: qt_def mmap_keys_def accept(1) mmap_lookup_def ac''_def)
    next
      case False
      have ex_key: ex_key qts time accept ac'' = (b, ac')
        using Cons(4) time_t False
        by (auto simp: qt_def ac''_def)
      show ?thesis
        using Cons(1)[OF accept(2) _ ex_key] False[unfolded accept(1)] Cons(3)
        by (auto simp: mmap_keys_def mmap_lookup_def qt_def)
    qed
  next
    case False
```

```
    have ex_key: ex_key qts time accept ac = (b, ac')
      using Cons(4) False
      by (auto simp: qt_def)
    show ?thesis
      using Cons(1)[OF Cons(2) _ ex_key] False Cons(3)
      by (auto simp: mmap_keys_def mmap_lookup_def qt_def)
  qed
qed (auto simp: mmap_keys_def)

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)
  using reach_window_run_tj
  by (fastforce simp: valid_window_matchF_def simp del: reach_window.simps)

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)
  using reach_window_run_sj
  by (fastforce simp: valid_window_matchF_def simp del: reach_window.simps)

definition matchF_loop_cond I t =
  (λw. 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 =
  (λw. 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 =
  (λw. 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 −
  obtain *t' t''* **where** *t'_def*: *reaches_on* (*w_run_t args*) *t0* (*take i ts*) *t'*
    w_run_t args t' = *Some* (*t''*, *ts* ! *i*)
    using *reaches_on_split*[*OF assms*]
    by *auto*
  show *?thesis*
    using *run_t_sound*[*OF t'_def*] *assms*(*2*)
    by *simp*
qed

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 −
  obtain s' s'' where s'_def: reaches_on (w_run_sub args) sub (take i bs) s'
    w_run_sub args s' = Some (s'', bs ! i)
    using reaches_on_split[OF assms]
    by auto
  show ?thesis
    using run_sub_sound[OF s'_def] assms(2)
    by simp
qed

lemma run_ts: reaches_on (w_run_t args) t ts t' ⟹ t = t0 ⟹ chain_le ts
proof (induction t ts t' rule: reaches_on_rev_induct)
  case (2 s s' v vs s'')
  show ?case
  proof (cases vs rule: rev_cases)
    case (snoc zs z)
    show ?thesis
      using 2(3)[OF 2(4)]
      using chain_le_app[OF _ τ_mono[of length zs Suc (length zs) σ]]
        run_t_sound'[OF reaches_on_app[OF 2(1,2), unfolded 2(4)], of length zs]
        run_t_sound'[OF reaches_on_app[OF 2(1,2), unfolded 2(4)], of Suc (length zs)]
      unfolding snoc
      by (auto simp: nth_append)
  qed (auto intro: chain_le.intros)
qed (auto intro: chain_le.intros)

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
  using run_t_sound'
  unfolding ts_at_def
  by fastforce

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 [])
  using run_sub_sound'
  unfolding bs_at_def
  by fastforce

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
  using run_sub_sound'
  unfolding bs_at_def
  by fastforce

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
  using ts_at_tau
  by fastforce

lemma steps_is_run: steps (w_step args) rho q ij = run q (sub_bs rho ij)
  unfolding NFA.run'_def steps_def step_def transs_def qf_def ..

lemma acc_is_accept: wacc rho q (i, j) = w_accept args (run q (sub_bs rho (i, j)))
  unfolding acc_def steps_is_run by auto
```

lemma *iarray_list_of*: *IArray* (*IArray.list_of xs*) = *xs*
by (*cases xs*) *auto*

lemma *map_iarray_list_of*: *map IArray* (*map IArray.list_of bss*) = *bss*
using *iarray_list_of*
by (*induction bss*) *auto*

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 −
have *j_eq*: *j* = *i* + *length* (*sub_bs rho* (*i*, *j*))
using *assms* **by** *auto*
define *bs* **where** *bs* = *map* (λ*phi*. *sat phi j*) (*collect_subfmlas r* [])
have *IH*: *IH r 0 qf* [] *transs* (*map IArray.list_of* (*sub_bs rho* (*i*, *j*))) *bs i*
unfolding *IH_def transs_def qf_def NFA.SQ_def build_nfa_impl_state_cnt bs_def*
using *assms run_sub_sound bs_at_nth length_bs_at* **by** *fastforce*
interpret *NFA_array*: *nfa_array transs IArray transs qf*
by *unfold_locales* (*auto simp*: *qf_def transs_def build_nfa_impl_state_cnt*)
have *run_run'*: *NFA_array.run R* (*map IArray.list_of* (*sub_bs rho* (*i*, *j*))) = *NFA_array.run'* *R* (*sub_bs rho* (*i*, *j*)) **for** *R*
using *NFA_array.run'_eq*[*of sub_bs rho* (*i*, *j*) *map IArray.list_of* (*sub_bs rho* (*i*, *j*))]
unfolding *map_iarray_list_of*
by *auto*
show *?thesis*
using *nfa_correct*[*OF IH*, *unfolded NFA.run_accept_def*]
unfolding *run_accept_eps_iff_run_accept*[*OF assms*(*4*) *IH*] *acc_is_accept NFA.run_accept_def run_run' NFA_array.accept'_eq*
by (*simp add*: *j_eq*[*symmetric*] *accept_def assms*(*2*) *qf_def transs_def*)
qed

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*
using *acc_match acc_is_accept steps_is_run*
by *metis*

lemma *drop_take_drop*: *i* ≤ *j* ⟹ *j* ≤ *length rho* ⟹ *drop i* (*take j rho*) @ *drop j rho* = *drop i rho*
apply (*induction i arbitrary*: *j rho*)
by *auto* (*metis append_take_drop_id diff_add drop_drop drop_take*)

lemma *take_Suc*: *drop n xs* = *y* # *ys* ⟹ *take n xs* @ [*y*] = *take* (*Suc n*) *xs*
by (*metis drop_all list.distinct*(*1*) *list.sel*(*1*) *not_less take_hd_drop*)

lemma *valid_init_matchP*: *valid_matchP I t0 sub* [] *0* (*init_window args t0 sub*)
using *valid_init_window*
by (*fastforce simp*: *valid_window_matchP_def Let_def intro*: *reaches_on.intros split*: *option.splits*)

lemma *valid_init_matchF*: *valid_matchF I t0 sub* [] *0* (*init_window args t0 sub*)
using *valid_init_window*
by (*fastforce simp*: *valid_window_matchF_def Let_def intro*: *reaches_on.intros split*: *option.splits*)

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 −
  obtain w' where w'_def: adv_end args w = Some w'
    using before_end
    by (fastforce simp: adv_end_def Let_def split: prod.splits)
  define st where st = w_st w'
  define i where i = w_i w'
  define ti where ti = w_ti w'
  define si where si = w_si w'
  define tj where tj = w_tj w'
  define sj where sj = w_sj w'
  define s where s = w_s w'
  define e where e = w_e w'
  define rho' where rho' = rho @ [(t, b)]
  have reaches_on': reaches_on (w_run_t args) t0 (map fst rho') tj'''
    using valid_before' reach_window_run_tj[OF reach_window_app[OF _ before_end]]
    by (auto simp: valid_window_matchP_def rho'_def)
  have rho_mono: ⋀t'. t' ∈ set (map fst rho) ⟹ t' ≤ t
    using ts_at_mono[OF reaches_on'] nat_less_le
    by (fastforce simp: rho'_def ts_at_def nth_append in_set_conv_nth split: list.splits)
  have valid_adv_end_w: valid_window args t0 sub rho' w'
    using valid_before' valid_adv_end[OF _ before_end rho_mono]
    by (auto simp: valid_window_matchP_def rho'_def w'_def)
  have w_ij_adv_end: w_i w' = w_i w w_j w' = Suc j
    using valid_before' w'_def
    by (auto simp: valid_window_matchP_def adv_end_def Let_def before_end split: prod.splits)
  have valid_before: rw t0 sub rho' (i, ti, si, Suc j, tj, sj)
    ⋀i j. i ≤ j ⟹ j < length rho' ⟹ ts_at rho' i ≤ ts_at rho' j
    ∀ q. mmap_lookup e q = sup_leadsto init step rho' i (Suc j) q
    valid_s init step st accept rho' i i (Suc j) s
    w_j w' = Suc j i ≤ Suc j
    using valid_adv_end_w
    unfolding valid_window_def Let_def ti_def si_def i_def tj_def sj_def s_def e_def w_ij_adv_end
st_def
    by auto
  note read_t_def = run_t_read[OF before_end(1)]
  have t_props: ∀ l < i. memL (ts_at rho' l) t I
    using valid_before'
     by (auto simp: valid_window_matchP_def i_def w_ij_adv_end read_t_def rho'_def ts_at_def
nth_append)

  note reaches_on_tj = reach_window_run_tj[OF valid_before(1)]
  note reaches_on_sj = reach_window_run_sj[OF valid_before(1)]
  have length_rho': length rho' = Suc j length rho = j
    using valid_before
    by (auto simp: rho'_def)
  have j_len_rho': j < length rho'
    by (auto simp: length_rho')
  have tj_eq: t = τ σ j t = ts_at rho' j
    using run_t_sound'[OF reaches_on_tj, of j]
    by (auto simp: rho'_def length_rho' nth_append ts_at_def)
  have bj_def: b = bs_at rho' j
    using run_sub_sound'[OF reaches_on_sj, of j]
    by (auto simp: rho'_def length_rho' nth_append bs_at_def)
  define w'' where loop_def: w'' = while (matchP_cond I t) (adv_start args) w'
```

have *inv_before*: *matchP_inv I t0 sub rho′ (Suc j) tj sj t w′*
using *valid_adv_end_w valid_before t_props*
unfolding *matchP_loop_inv_def*
by (*auto simp*: *tj_def sj_def i_def*)
have *loop*: *matchP_inv I t0 sub rho′ (Suc j) tj sj t w′′ ∧ ¬matchP_cond I t w′′*
unfolding *loop_def*
proof (*rule while_rule_lemma*[*of matchP_inv I t0 sub rho′ (Suc j) tj sj t*])
fix *w_cur* :: (*bool iarray*, *nat set*, *′d*, *′t*, *′e*) *window*
assume *assms*: *matchP_inv I t0 sub rho′ (Suc j) tj sj t w_cur matchP_cond I t w_cur*
define *st_cur* **where** *st_cur* = *w_st w_cur*
define *i_cur* **where** *i_cur* = *w_i w_cur*
define *ti_cur* **where** *ti_cur* = *w_ti w_cur*
define *si_cur* **where** *si_cur* = *w_si w_cur*
define *s_cur* **where** *s_cur* = *w_s w_cur*
define *e_cur* **where** *e_cur* = *w_e w_cur*
have *valid_loop*: *rw t0 sub rho′* (*i_cur*, *ti_cur*, *si_cur*, *Suc j*, *tj*, *sj*)
⋀*i j*. *i* ≤ *j* ⟹ *j* < *length rho′* ⟹ *ts_at rho′ i* ≤ *ts_at rho′ j*
∀ *q*. *mmap_lookup e_cur q* = *sup_leadsto init step rho′ i_cur* (*Suc j*) *q*
valid_s init step st_cur accept rho′ i_cur i_cur (*Suc j*) *s_cur*
w_j w_cur = *Suc j*
using *assms*(*1*)[*unfolded matchP_loop_inv_def valid_window_matchP_def*] *valid_before*(*6*)
ti_cur_def si_cur_def i_cur_def s_cur_def e_cur_def
by (*auto simp*: *valid_window_def Let_def init_def step_def st_cur_def accept_def*
split: *option.splits*)
obtain *ti′_cur si′_cur t_cur b_cur* **where** *run_si_cur*:
w_run_t args ti_cur = *Some* (*ti′_cur*, *t_cur*) *w_run_sub args si_cur* = *Some* (*si′_cur*, *b_cur*)
t_cur = *ts_at rho′ i_cur b_cur* = *bs_at rho′ i_cur*
using *assms*(*2*) *reach_window_run_si*[*OF valid_loop*(*1*)] *reach_window_run_ti*[*OF valid_loop*(*1*)]
unfolding *matchP_loop_cond_def valid_loop*(*5*) *i_cur_def*
by *auto*
have ⋀*l*. *l* < *i_cur* ⟹ *memL* (*ts_at rho′ l*) *t I*
using *assms*(*1*)
unfolding *matchP_loop_inv_def i_cur_def*
by *auto*
then have ⋀*l*. *l* < *Suc* (*i_cur*) ⟹ *memL* (*ts_at rho′ l*) *t I*
using *assms*(*2*) *run_t_read*[*OF run_si_cur*(*1*), *unfolded run_si_cur*(*3*)]
unfolding *matchP_loop_cond_def i_cur_def ti_cur_def*
by (*auto simp*: *less_Suc_eq*)
then show *matchP_inv I t0 sub rho′* (*Suc j*) *tj sj t* (*adv_start args w_cur*)
using *assms i_cur_def valid_adv_start valid_adv_start_bounds*
unfolding *matchP_loop_inv_def matchP_loop_cond_def*
by *fastforce*
next
{
fix *w1 w2*
assume *lassms*: *matchP_inv I t0 sub rho′* (*Suc j*) *tj sj t w1 matchP_cond I t w1*
w2 = *adv_start args w1*
define *i_cur* **where** *i_cur* = *w_i w1*
define *ti_cur* **where** *ti_cur* = *w_ti w1*
define *si_cur* **where** *si_cur* = *w_si w1*
have *valid_loop*: *rw t0 sub rho′* (*i_cur*, *ti_cur*, *si_cur*, *Suc j*, *tj*, *sj*) *w_j w1* = *Suc j*
using *lassms*(*1*)[*unfolded matchP_loop_inv_def valid_window_matchP_def*] *valid_before*(*6*)
ti_cur_def si_cur_def i_cur_def
by (*auto simp*: *valid_window_def Let_def*)
obtain *ti′_cur si′_cur t_cur b_cur* **where** *run_si_cur*:
w_run_t args ti_cur = *Some* (*ti′_cur*, *t_cur*)
w_run_sub args si_cur = *Some* (*si′_cur*, *b_cur*)
using *lassms*(*2*) *reach_window_run_si*[*OF valid_loop*(*1*)] *reach_window_run_ti*[*OF valid_loop*(*1*)]

```
      unfolding matchP_loop_cond_def valid_loop i_cur_def
      by auto
    have w1_ij: w_i w1 < Suc j w_j w1 = Suc j
      using lassms
      unfolding matchP_loop_inv_def matchP_loop_cond_def
      by auto
    have w2_ij: w_i w2 = Suc (w_i w1) w_j w2 = Suc j
      using w1_ij lassms(3) run_si_cur(1,2)
      unfolding ti_cur_def si_cur_def
      by (auto simp: adv_start_def Let_def split: option.splits prod.splits if_splits)
    have w_j w2 − w_i w2 < w_j w1 − w_i w1
      using w1_ij w2_ij
      by auto
  }
  then have {(s', s). matchP_inv I t0 sub rho' (Suc j) tj sj t s ∧ matchP_cond I t s ∧
    s' = adv_start args s} ⊆ measure (λw. w_j w − w_i w)
    by auto
  then show wf {(s', s). matchP_inv I t0 sub rho' (Suc j) tj sj t s ∧ matchP_cond I t s ∧
    s' = adv_start args s}
    using wf_measure wf_subset by auto
qed (auto simp: inv_before)
have valid_w': valid_window args t0 sub rho' w''
  using conjunct1[OF loop]
  unfolding matchP_loop_inv_def
  by auto
have w_tsj_w': w_tj w'' = tj w_sj w'' = sj w_j w'' = Suc j
  using loop
  by (auto simp: matchP_loop_inv_def)
define st' where st' = w_st w''
define ac where ac = w_ac w''
define i' where i' = w_i w''
define ti' where ti' = w_ti w''
define si' where si' = w_si w''
define s' where s' = w_s w''
define e' where e' = w_e w''
define tj' where tj' = w_tj w''
define sj' where sj' = w_sj w''
have i'_le_Suc_j: i' ≤ Suc j
  using loop
  unfolding matchP_loop_inv_def
  by (auto simp: valid_window_def Let_def i'_def)
have valid_after: rw t0 sub rho' (i', ti', si', Suc j, tj', sj')
  ⋀i j. i ≤ j ⟹ j < length rho' ⟹ ts_at rho' i ≤ ts_at rho' j
  distinct (map fst e')
  ∀ q. mmap_lookup e' q = sup_leadsto init step rho' i' (Suc j) q
  ⋀q. case Mapping.lookup ac q of None ⇒ True | Some v ⇒ accept q = v
  valid_s init step st' accept rho' i' i' (Suc j) s' i' ≤ Suc j Suc j ≤ length rho'
  using valid_w' i'_le_Suc_j
   unfolding valid_window_def Let_def i'_def ti'_def si'_def s'_def e'_def tj'_def sj'_def ac_def
st'_def w_tsj_w'
  by auto
note lookup_e' = valid_after(3,4,5,6)
obtain β ac' where ac'_def: ex_key e' (λt'. memR t' t I)
  (w_accept args) ac = (β, ac')
  by fastforce
have β_def: β = (∃ q∈mmap_keys e'. memR (the (mmap_lookup e' q)) t I ∧ accept q)
  ⋀q. case Mapping.lookup ac' q of None ⇒ True | Some v ⇒ accept q = v
  using ex_key_sound[OF valid_after(5) valid_after(3) ac'_def]
```

```
    by auto
  have i'_set: ⋀l. l < w_i w'' ⟹ memL (ts_at rho' l) (ts_at rho' j) I
    using loop length_rho' i'_le_Suc_j
    unfolding matchP_loop_inv_def
    by (auto simp: ts_at_def rho'_def nth_append i'_def)
  have b_alt: (∃ q ∈ mmap_keys e'. memR (the (mmap_lookup e' q)) t I ∧ accept q) ⟷ sat (MatchP
I r) j
  proof (rule iffI)
    assume ∃ q ∈ mmap_keys e'. memR (the (mmap_lookup e' q)) t I ∧ accept q
    then obtain q where q_def: q ∈ mmap_keys e'
      memR (the (mmap_lookup e' q)) t I accept q
      by auto
    then obtain ts' where ts_def: mmap_lookup e' q = Some ts'
      by (auto dest: Mapping_keys_dest)
    have sup_leadsto init step rho' i' (Suc j) q = Some ts'
      using lookup_e' ts_def q_def valid_after(4,7,8)
      by (auto simp: rho'_def sup_leadsto_app_cong)
    then obtain l where l_def: l < i' steps step rho' init (l,  Suc j) = q
      ts_at rho' l = ts'
      using sup_leadsto_SomeE[OF i'_le_Suc_j]
      unfolding i'_def
      by fastforce
    have l_le_j: l ≤ j and l_le_Suc_j: l ≤ Suc j
      using l_def(1) i'_le_Suc_j
      by (auto simp: i'_def)
    have tau_l: l < j ⟹ fst (rho ! l) = τ σ l
      using run_t_sound'[OF reaches_on_tj, of l] length_rho'
      by (auto simp: rho'_def nth_append)
    have tau_l_left: memL ts' t I
      unfolding l_def(3)[symmetric] tj_eq(2)
      using i'_set l_def(1)
      by (auto simp: i'_def)
    have (l, Suc j) ∈ match r
      using accept_match[OF reaches_on_sj l_le_Suc_j _ wf] q_def(3) length_rho' init_def l_def(2)
        rho'_def
      by auto
    then show sat (MatchP I r) j
      using l_le_j q_def(2) ts_at_tau[OF reaches_on_tj] tau_l_left
      by (auto simp: mem_def tj_eq rho'_def ts_def l_def(3)[symmetric] tau_l tj_def ts_at_def
          nth_append length_rho' intro: exI[of _ l] split: if_splits)
  next
    assume sat (MatchP I r) j
    then obtain l where l_def: l ≤ j l ≤ Suc j mem (τ σ l) (τ σ j) I (l, Suc j) ∈ match r
      by auto
    show (∃ q∈mmap_keys e'. memR (the (mmap_lookup e' q)) t I ∧ accept q)
    proof −
      have l_lt_j: l < Suc j
        using l_def(1) by auto
      then have ts_at_l_j: ts_at rho' l ≤ ts_at rho' j
        using ts_at_mono[OF reaches_on' _ j_len_rho']
        by (auto simp: rho'_def length_rho')
      have ts_j_l: memL (ts_at rho' l) (ts_at rho' j) I
        using l_def(3) ts_at_tau[OF reaches_on_tj] l_lt_j length_rho' tj_eq
        unfolding rho'_def mem_def
        by auto
      have i' = Suc j ∨ ¬memL (ts_at rho' i') (ts_at rho' j) I
      proof (rule Meson.disj_comm, rule disjCI)
        assume i' ≠ Suc j
```

```
      then have i'_j: i' < Suc j
        using valid_after
        by auto
      obtain t' b' where tbi_cur_def: w_read_t args ti' = Some t'
        t' = ts_at rho' i' b' = bs_at rho' i'
        using reach_window_run_ti[OF valid_after(1) i'_j]
          reach_window_run_si[OF valid_after(1) i'_j] run_t_read
        by auto
      show ¬memL (ts_at rho' i') (ts_at rho' j) I
        using loop tbi_cur_def(1) i'_j length_rho'
        unfolding matchP_loop_inv_def matchP_loop_cond_def tj_eq(2) ti'_def[symmetric]
        by (auto simp: i'_def tbi_cur_def)
    qed
    then have l_lt_i': l < i'
    proof (rule disjE)
      assume assm: ¬memL (ts_at rho' i') (ts_at rho' j) I
      show l < i'
      proof (rule ccontr)
        assume ¬l < i'
        then have ts_at_i'_l: ts_at rho' i' ≤ ts_at rho' l
          using ts_at_mono[OF reaches_on'] l_lt_j length_rho'
          by (auto simp: rho'_def length_rho')
        show False
          using assm memL_mono[OF ts_j_l ts_at_i'_l]
          by auto
      qed
    qed (auto simp add: l_lt_j)
    define q where q_def: q = steps step rho' init (l, Suc j)
    then obtain l' where l'_def: sup_leadsto init step rho' i' (Suc j) q = Some (ts_at rho' l')
      l ≤ l' l' < i'
      using sup_leadsto_SomeI[OF l_lt_i'] by fastforce
    have ts_j_l': memR (ts_at rho' l') (ts_at rho' j) I
    proof −
      have ts_at_l_l': ts_at rho' l ≤ ts_at rho' l'
        using ts_at_mono[OF reaches_on' l'_def(2)] l'_def(3) valid_after(4,7,8)
        by (auto simp add: rho'_def length_rho' dual_order.order_iff_strict)
      show ?thesis
        using l_def(3) memR_mono[OF _ ts_at_l_l']
          ts_at_tau[OF reaches_on_tj] l'_def(2,3) valid_after(4,7,8)
        by (auto simp: mem_def rho'_def length_rho')
    qed
    have lookup_e'_q: mmap_lookup e' q = Some (ts_at rho' l')
      using lookup_e' l'_def(1) valid_after(4,7,8)
      by (auto simp: rho'_def sup_leadsto_app_cong)
    show ?thesis
      using accept_match[OF reaches_on_sj l_def(2) _ wf] l_def(4) ts_j_l' lookup_e'_q tj_eq(2)
      by (auto simp: bs_at_def nth_append init_def length_rho'(1) q_def intro!: bexI[of _ q] Mapping_keys_intro)
  qed
qed
have read_tj_Some: ⋀t' l. w_read_t args tj = Some t' ⟹ l < i' ⟹ memL (ts_at rho' l) t' I
proof −
  fix t' l
  assume lassms: (w_read_t args) tj = Some t' l < i'
  obtain tj'''' where reaches_on_tj'''':
    reaches_on (w_run_t args) t0 (map fst (rho' @ [(t', undefined)])) tj''''
    using reaches_on_app[OF reaches_on_tj] read_t_run[OF lassms(1)]
    by auto
```

```
    have t ≤ t'
      using ts_at_mono[OF reaches_on_tj'''', of length rho length rho']
      by (auto simp: ts_at_def nth_append rho'_def)
    then show memL (ts_at rho' l) t' I
      using memL_mono' lassms(2) loop
      unfolding matchP_loop_inv_def
      by (fastforce simp: i'_def)
  qed
  define w''' where w''' = w''(|w_ac := ac'|)
  have rw t0 sub rho' (w_i w''', w_ti w''', w_si w''', w_j w''', w_tj w''', w_sj w''')
    using valid_after(1)
   by (auto simp del: reach_window.simps simp: w'''_def i'_def ti'_def si'_def tj'_def sj'_def w_tsj_w')
  moreover have valid_window args t0 sub rho' w'''
    using valid_w'
    by (auto simp: w'''_def valid_window_def Let_def β_def(2))
  ultimately have valid_matchP I t0 sub rho' (Suc j) w'''
    using i'_set read_tj_Some
    by (auto simp: valid_window_matchP_def w'''_def w_tsj_w' i'_def split: option.splits)
  moreover have eval_mP I w = Some ((t, sat (MatchP I r) j), w''')
     by (simp add: read_t_def Let_def loop_def[symmetric] ac'_def[unfolded e'_def ac_def] w'''_def
w'_def trans[OF β_def(1) b_alt])
  ultimately show ?thesis
    by (auto simp: tj_eq rho'_def)
qed

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 −
  define st where st = w_st w
  define ti where ti = w_ti w
  define si where si = w_si w
  define j where j = w_j w
  define tj where tj = w_tj w
  define sj where sj = w_sj w
  define s where s = w_s w
  define e where e = w_e w
  have valid_before: rw 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. mmap_lookup e q = sup_leadsto init step rho i j q
    valid_s init step st accept rho i i j s
    i = w_i w i ≤ j length rho = j
    using valid_before'[unfolded valid_window_matchF_def] ti_def
      si_def j_def tj_def sj_def s_def e_def
    by (auto simp: valid_window_def Let_def init_def step_def st_def accept_def)
  obtain ti''' where tbi_def: w_run_t args ti = Some (ti''', t)
    using eval read_t_run
    by (fastforce simp: Let_def ti_def si_def split: option.splits if_splits)
  have t_tau: t = τ σ i
    using run_t_sound[OF _ tbi_def] valid_before(1)
    by auto
  note t_def = run_t_read[OF tbi_def(1)]
```

obtain w' **where** *loop_def*: *while_break* (*matchF_cond I t*) (*adv_end args*) w = *Some* w'
using *eval*
by (*auto simp*: *ti_def*[*symmetric*] *t_def split*: *option.splits*)
have *adv_start_last*:
adv_start args w' = w''
using *eval loop_def*[*symmetric*] *run_t_read*[*OF tbi_def*(*1*)]
by (*auto simp*: *ti_def split*: *option.splits prod.splits if_splits*)
have *inv_before*: *matchF_inv'* *t0 sub rho i ti si j tj sj w*
using *valid_before*(*1*) *valid_before'*
unfolding *matchF_loop_inv'_def valid_before*(*6*) *valid_window_matchF_def*
by (*auto simp add*: *ti_def si_def j_def tj_def sj_def simp del*: *reach_window.simps*
dest: *reach_window_shift_all intro*!: *exI*[*of* _ []])
have *i_j*: $i \leq j$ *length rho* = *j*
using *valid_before* **by** *auto*
define j'' **where** j'' = *w_j* w''
define tj'' **where** tj'' = *w_tj* w''
define sj'' **where** sj'' = *w_sj* w''
have *loop*: *matchF_inv'* *t0 sub rho i ti si j tj sj* $w' \wedge \neg$ *matchF_cond I t* w'
proof (*rule while_break_sound*[*of matchF_inv' t0 sub rho i ti si j tj sj matchF_cond I t adv_end args*
$\lambda w'$. *matchF_inv' t0 sub rho i ti si j tj sj* $w' \wedge \neg$ *matchF_cond I t* w' *w*, *unfolded loop_def*, *simplified*])
fix *w_cur w_cur'* :: (*bool iarray*, *nat set*, '*d*, '*t*, '*e*) *window*
assume *assms*: *matchF_inv' t0 sub rho i ti si j tj sj w_cur matchF_cond I t w_cur adv_end args*
w_cur = *Some w_cur'*
define *j_cur* **where** *j_cur* = *w_j w_cur*
define *tj_cur* **where** *tj_cur* = *w_tj w_cur*
define *sj_cur* **where** *sj_cur* = *w_sj w_cur*
obtain *rho'* **where** *rho'_def*: *valid_window args t0 sub* (*rho* @ *rho'*) *w_cur*
rw t0 sub (*rho* @ *rho'*) (*j*, *tj*, *sj*, *w_j w_cur*, *w_tj w_cur*, *w_sj w_cur*)
using *assms*(*1*)[*unfolded matchF_loop_inv'_def valid_window_matchF_def*]
by *auto*
obtain *tj' x sj' y* **where** *append*: *w_run_t args tj_cur* = *Some* (*tj'*, *x*)
w_run_sub args sj_cur = *Some* (*sj'*, *y*)
using *assms*(*3*)
unfolding *tj_cur_def sj_cur_def*
by (*auto simp*: *adv_end_def Let_def split*: *option.splits*)
note *append'* = *append*[*unfolded tj_cur_def sj_cur_def*]
define *rho''* **where** *rho''* = *rho* @ *rho'*
have *reach*: *reaches_on* (*w_run_t args*) *t0* (*map fst* (*rho''* @ [(*x*, *undefined*)])) *tj'*
using *reaches_on_app*[*OF reach_window_run_tj*[*OF rho'_def*(*2*)] *append'*(*1*)]
by (*auto simp*: *rho''_def*)
have *mono*: $\bigwedge t'$. $t' \in$ *set* (*map fst rho''*) $\Longrightarrow t' \leq x$
using *ts_at_mono*[*OF reach*, *of* _ *length rho''*] *nat_less_le*
by (*fastforce simp*: *ts_at_def nth_append in_set_conv_nth split*: *list.splits*)
show *matchF_inv' t0 sub rho i ti si j tj sj w_cur'*
using *assms*(*1*,*3*) *reach_window_app*[*OF rho'_def*(*2*) *append*[*unfolded tj_cur_def sj_cur_def*]]
valid_adv_end[*OF rho'_def*(*1*) *append' mono*] *adv_end_bounds*[*OF append'*]
unfolding *matchF_loop_inv'_def matchF_loop_cond_def rho''_def*
by *auto*
next
obtain *l* **where** *l_def*: $\neg\tau\ \sigma\ l \leq t +$ *right I*
unfolding *t_tau*
using *ex_lt_τ*[*OF bounded*]
by *auto*
{
fix *w1 w2*
assume *lassms*: *matchF_inv' t0 sub rho i ti si j tj sj w1 matchF_cond I t w1*
Some w2 = *adv_end args w1*
define *j_cur* **where** *j_cur* = *w_j w1*

define *tj_cur* **where** *tj_cur* = *w_tj w1*
define *sj_cur* **where** *sj_cur* = *w_sj w1*
obtain *rho'* **where** *rho'_def*: *valid_window args t0 sub* (*rho* @ *rho'*) *w1*
rw t0 sub (*rho* @ *rho'*) (*j*, *tj*, *sj*, *w_j w1*, *w_tj w1*, *w_sj w1*)
using *lassms*(*1*)[*unfolded matchF_loop_inv'_def valid_window_matchF_def*]
by *auto*
obtain *tj' x sj' y* **where** *append*: *w_run_t args tj_cur* = *Some* (*tj'*, *x*)
w_run_sub args sj_cur = *Some* (*sj'*, *y*)
using *lassms*(*3*)
unfolding *tj_cur_def sj_cur_def*
by (*auto simp*: *adv_end_def Let_def split*: *option.splits*)
note *append'* = *append*[*unfolded tj_cur_def sj_cur_def*]
define *rho''* **where** *rho''* = *rho* @ *rho'*
have *reach*: *reaches_on* (*w_run_t args*) *t0* (*map fst* (*rho''* @ [(*x*, *undefined*)])) *tj'*
using *reaches_on_app*[*OF reach_window_run_tj*[*OF rho'_def*(*2*)] *append'*(*1*)]
by (*auto simp*: *rho''_def*)
have *mono*: ⋀*t'*. *t'* ∈ *set* (*map fst rho''*) ⟹ *t'* ≤ *x*
using *ts_at_mono*[*OF reach*, *of* _ *length rho''*] *nat_less_le*
by (*fastforce simp*: *ts_at_def nth_append in_set_conv_nth split*: *list.splits*)
have *t_cur_tau*: *x* = *τ σ j_cur*
using *ts_at_tau*[*OF reach*, *of length rho''*] *rho'_def*(*2*)
by (*auto simp*: *ts_at_def j_cur_def rho''_def*)
have *j_cur* < *l*
using *lassms*(*2*)[*unfolded matchF_loop_cond_def*] *l_def memR_mono'*[*OF* _ *τ_mono*[*of l j_cur σ*]]
unfolding *run_t_read*[*OF append*(*1*), *unfolded t_cur_tau tj_cur_def*]
by (*fastforce dest*: *memR_dest*)
moreover have *w_j w2* = *Suc j_cur*
using *adv_end_bounds*[*OF append'*]
unfolding *lassms*(*3*)[*symmetric*] *j_cur_def*
by *auto*
ultimately have *l* − *w_j w2* < *l* − *w_j w1*
unfolding *j_cur_def*
by *auto*
}
then have {(*ta*, *s*). *matchF_inv' t0 sub rho i ti si j tj sj s* ∧ *matchF_cond I t s* ∧
Some ta = *adv_end args s*} ⊆ *measure* (λ*w*. *l* − *w_j w*)
by *auto*
then show *wf* {(*ta*, *s*). *matchF_inv' t0 sub rho i ti si j tj sj s* ∧ *matchF_cond I t s* ∧
Some ta = *adv_end args s*}
using *wf_measure wf_subset*
by *auto*
qed (*auto simp*: *inv_before*)
define *i'* **where** *i'* = *w_i w'*
define *ti'* **where** *ti'* = *w_ti w'*
define *si'* **where** *si'* = *w_si w'*
define *j'* **where** *j'* = *w_j w'*
define *tj'* **where** *tj'* = *w_tj w'*
define *sj'* **where** *sj'* = *w_sj w'*
obtain *rho'* **where** *rho'_def*: *valid_window args t0 sub* (*rho* @ *rho'*) *w'*
rw t0 sub (*rho* @ *rho'*) (*j*, *tj*, *sj*, *j'*, *tj'*, *sj'*)
i = *i' j* ≤ *j'*
using *loop*
unfolding *matchF_loop_inv'_def i'_def j'_def tj'_def sj'_def*
by *auto*
obtain *tje tm* **where** *tm_def*: *w_read_t args tj'* = *Some tm w_run_t args tj'* = *Some* (*tje*, *tm*)
using *eval read_t_run loop_def t_def ti_def*
by (*auto simp*: *t_def Let_def tj'_def split*: *option.splits if_splits*)

```
  have drop_j_rho: drop j (map fst (rho @ rho')) = map fst rho'
    using i_j
    by auto
  have reaches_on (w_run_t args) ti (drop i (map fst rho)) tj
    using valid_before(1)
    by auto
  then have reaches_on (w_run_t args) ti
    (drop i (map fst rho) @ (drop j (map fst (rho @ rho')))) tj'
    using rho'_def reaches_on_trans
    by fastforce
  then have reaches_on (w_run_t args) ti (drop i (map fst (rho @ rho'))) tj'
    unfolding drop_j_rho
    by (auto simp: i_j)
  then have reach_tm: reaches_on (w_run_t args) ti (drop i (map fst (rho @ rho')) @ [tm]) tje
    using reaches_on_app tm_def(2)
    by fastforce
  have run_tsi': w_run_t args ti' ≠ None
    using tbi_def loop
    by (auto simp: matchF_loop_inv'_def ti'_def si'_def)
  have memR_t_tm: ¬ memR t tm I
    using loop tm_def
    by (auto simp: tj'_def matchF_loop_cond_def)
  have i_le_rho: i ≤ length rho
    using valid_before
    by auto
  define rho'' where rho'' = rho @ rho'
  have t_tfin: t ∈ tfin
    using τ_fin
    by (auto simp: t_tau)
  have i'_lt_j': i' < j'
   using rho'_def(1,2,3)[folded rho''_def] i_j reach_tm[folded rho''_def] memR_t_tm tbi_def memR_tfin_refl[OF
t_tfin]
    by (cases i' = j') (auto dest!: reaches_on_NilD elim!: reaches_on.cases[of _ _ [tm]])
  have adv_last_bounds: j'' = j' tj'' = tj' sj'' = sj'
    using valid_adv_start_bounds[OF rho'_def(1) i'_lt_j'[unfolded i'_def j'_def]]
    unfolding adv_start_last j'_def j''_def tj'_def tj''_def sj'_def sj''_def
    by auto
  show ?thesis
    using eval rho'_def run_tsi' i_j(2) adv_last_bounds tj''_def tj_def sj''_def sj_def
      loop_def t_def ti_def tj'_def tm_def memR_t_tm
    by (auto simp: drop_map run_t_read[OF tbi_def(1)] Let_def
        split: option.splits prod.splits if_splits intro!: exI[of _ rho'])
qed

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 −
  define st where st = w_st w
  define ti where ti = w_ti w
  define si where si = w_si w
  define j where j = w_j w
  define tj where tj = w_tj w
```

define *sj* **where** *sj = w_sj w*
define *s* **where** *s = w_s w*
define *e* **where** *e = w_e w*
have *valid_before*: *rw 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. mmap_lookup e q = sup_leadsto init step rho i j q
valid_s init step st accept rho i i j s
i = w_i w i ≤ j length rho = j
using *valid_before'*[*unfolded valid_window_matchF_def*] *ti_def*
si_def j_def tj_def sj_def s_def e_def
by (*auto simp*: *valid_window_def Let_def init_def step_def st_def accept_def*)
define *rho''* **where** *rho'' = rho @ rho'*
have *ij_le*: *i ≤ j j = length rho*
using *valid_before*
by *auto*
have *reach_tj*: *reaches_on (w_run_t args) t0 (take j (map fst rho'')) tj*
using *valid_before*(*1*) *ij_le*
by (*auto simp*: *take_map rho''_def simp del*: *reach_window.simps dest*!: *reach_window_run_tj*)
have *reach_ti*: *reaches_on (w_run_t args) t0 (take i (map fst rho'')) ti*
using *valid_before*(*1*) *ij_le*
by (*auto simp*: *take_map rho''_def*)
have *reach_si*: *reaches_on (w_run_sub args) sub (take i (map snd rho'')) si*
using *valid_before*(*1*) *ij_le*
by (*auto simp*: *take_map rho''_def*)
have *reach_sj*: *reaches_on (w_run_sub args) sub (take j (map snd rho'')) sj*
using *valid_before*(*1*) *ij_le*
by (*auto simp*: *take_map rho''_def simp del*: *reach_window.simps dest*!: *reach_window_run_sj*)
have *reach_tj'''*: *reaches_on (w_run_t args) t0 (map fst rho'') tj'''*
using *reaches_on_trans*[*OF reach_tj before_end*(*1*)[*folded tj_def*]] *ij_le*(*2*)
by (*auto simp del*: *map_append simp*: *rho''_def take_map drop_map map_append*[*symmetric*])
have *rho''_mono*: *⋀i j. i ≤ j ⟹ j < length rho'' ⟹ ts_at rho'' i ≤ ts_at rho'' j*
using *ts_at_mono*[*OF reach_tj'''*] **.**
obtain *tm'* **where** *reach_tm*: *reaches_on (w_run_t args) t0*
(map fst (rho'' @ [(tm, undefined)])) tm'
using *reaches_on_app*[*OF reach_tj'''*] *read_t_run*[*OF before_end*(*4*)]
by *auto*
have *tj'''_eq*: *⋀tj_cur. reaches_on (w_run_t args) t0 (map fst rho'') tj_cur ⟹*
tj_cur = tj'''
using *reaches_on_inj*[*OF reach_tj'''*]
by *blast*
have *reach_sj'''*: *reaches_on (w_run_sub args) sub (map snd rho'') sj'''*
using *reaches_on_trans*[*OF reach_sj before_end*(*2*)[*folded sj_def*]] *ij_le*(*2*)
by (*auto simp del*: *map_append simp*: *rho''_def take_map drop_map map_append*[*symmetric*])
have *sj'''_eq*: *⋀sj_cur. reaches_on (w_run_sub args) sub (map snd rho'') sj_cur ⟹*
sj_cur = sj'''
using *reaches_on_inj*[*OF reach_sj'''*]
by *blast*
have *reach_window_i*: *rw t0 sub rho'' (i, ti, si, length rho'', tj''', sj''')*
using *reach_windowI*[*OF reach_ti reach_si reach_tj''' reach_sj''' _ refl*] *ij_le*
by (*auto simp*: *rho''_def*)
have *reach_window_j*: *rw t0 sub rho'' (j, tj, sj, length rho'', tj''', sj''')*
using *reach_windowI*[*OF reach_tj reach_sj reach_tj''' reach_sj''' _ refl*] *ij_le*
by (*auto simp*: *rho''_def*)
have *t_def*: *t = τ σ i*
using *valid_before*(*6*) *read_t_run*[*OF before_end*(*3*)] *reaches_on_app*[*OF reach_ti*]
ts_at_tau[**where** *?rho=take i rho'' @ [(t, undefined)*]]
by (*fastforce simp*: *ti_def rho''_def valid_before*(*5*,*7*) *take_map ts_at_def nth_append*)
have *t_tfin*: *t ∈ tfin*

```
    using τ_fin
    by (auto simp: t_def)
  have i_lt_rho'': i < length rho''
    using ij_le before_end(3,4,5) reach_window_i memR_tfin_refl[OF t_tfin]
    by (cases i = length rho'') (auto simp: rho''_def ti_def dest!: reaches_on_NilD)
  obtain ti''' si''' b where tbi_def: w_run_t args ti = Some (ti''', t)
    w_run_sub args si = Some (si''', b) t = ts_at rho'' i b = bs_at rho'' i
    using reach_window_run_ti[OF reach_window_i i_lt_rho'']
      reach_window_run_si[OF reach_window_i i_lt_rho'']
      read_t_run[OF before_end(3), folded ti_def]
    by auto
  note before_end' = before_end(5)
  have read_ti: w_read_t args ti = Some t
    using run_t_read[OF tbi_def(1)] .
  have inv_before: matchF_inv I t0 sub rho'' i ti si tj''' sj''' w
    using valid_before' before_end(1,2,3) reach_window_j ij_le ti_def si_def j_def tj_def sj_def
    unfolding matchF_loop_inv_def valid_window_matchF_def
    by (auto simp: rho''_def ts_at_def nth_append)
  have i_j: i ≤ j
    using valid_before by auto
  have loop: pred_option' (λw'. matchF_inv I t0 sub rho'' i ti si tj''' sj''' w' ∧ ¬ matchF_cond I t w')
(while_break (matchF_cond I t) (adv_end args) w)
  proof (rule while_break_complete[of matchF_inv I t0 sub rho'' i ti si tj''' sj''', OF _ _ _ inv_before])
    fix w_cur :: (bool iarray, nat set, 'd, 't, 'e) window
    assume assms: matchF_inv I t0 sub rho'' i ti si tj''' sj''' w_cur matchF_cond I t w_cur
    define j_cur where j_cur = w_j w_cur
    define tj_cur where tj_cur = w_tj w_cur
    define sj_cur where sj_cur = w_sj w_cur
    define s_cur where s_cur = w_s w_cur
    define e_cur where e_cur = w_e w_cur
    have loop: valid_window args t0 sub (take (w_j w_cur) rho'') w_cur
      rw t0 sub rho'' (j_cur, tj_cur, sj_cur, length rho'', tj''', sj''')
      ⋀l. l∈{w_i w_cur..<w_j w_cur} ⟹ memR (ts_at rho'' i) (ts_at rho'' l) I
      using j_cur_def tj_cur_def sj_cur_def s_cur_def e_cur_def
        assms(1)[unfolded matchF_loop_inv_def]
      by auto
    have j_cur_lt_rho'': j_cur < length rho''
      using assms tj'''_eq before_end(4,5)
      unfolding matchF_loop_inv_def matchF_loop_cond_def
      by (cases j_cur = length rho'') (auto simp: j_cur_def split: option.splits)
    obtain tj_cur' sj_cur' x b_cur where tbi_cur_def: w_run_t args tj_cur = Some (tj_cur', x)
      w_run_sub args sj_cur = Some (sj_cur', b_cur)
      x = ts_at rho'' j_cur b_cur = bs_at rho'' j_cur
      using reach_window_run_ti[OF loop(2) j_cur_lt_rho'']
        reach_window_run_si[OF loop(2) j_cur_lt_rho'']
      by auto
    note reach_window_j'_cur = reach_window_shift[OF loop(2) j_cur_lt_rho'' tbi_cur_def(1,2)]
    note tbi_cur_def' = tbi_cur_def(1,2)[unfolded tj_cur_def sj_cur_def]
    have mono: ⋀t'. t' ∈ set (map fst (take (w_j w_cur) rho'')) ⟹ t' ≤ x
      using rho''_mono[of _ j_cur] j_cur_lt_rho'' nat_less_le
      by (fastforce simp: tbi_cur_def(3) j_cur_def ts_at_def nth_append in_set_conv_nth
          split: list.splits)
    have take_unfold: take (w_j w_cur) rho'' @ [(x, b_cur)] = (take (Suc (w_j w_cur)) rho'')
      using j_cur_lt_rho''
      unfolding tbi_cur_def(3,4)
      by (auto simp: ts_at_def bs_at_def j_cur_def take_Suc_conv_app_nth)
    obtain w_cur' where w_cur'_def: adv_end args w_cur = Some w_cur'
    by (fastforce simp: adv_end_def Let_def tj_cur_def[symmetric] sj_cur_def[symmetric] tbi_cur_def(1,2)
```

```
split: prod.splits)
  have ⋀l. l ∈{w_i w_cur'..<w_j w_cur'} ⟹
    memR (ts_at rho'' i) (ts_at rho'' l) I
    using loop(3) assms(2) w_cur'_def
    unfolding adv_end_bounds[OF tbi_cur_def' w_cur'_def] matchF_loop_cond_def
      run_t_read[OF tbi_cur_def(1)[unfolded tj_cur_def]] tbi_cur_def(3) tbi_def(3)
    by (auto simp: j_cur_def elim: less_SucE)
  then show pred_option' (matchF_inv I t0 sub rho'' i ti si tj''' sj''') (adv_end args w_cur)
    using assms(1) reach_window_j'_cur valid_adv_end[OF loop(1) tbi_cur_def' mono]
      w_cur'_def adv_end_bounds[OF tbi_cur_def' w_cur'_def]
    unfolding matchF_loop_inv_def j_cur_def take_unfold
    by (auto simp: pred_option'_def)
next
  {
    fix w1 w2
    assume lassms: matchF_inv I t0 sub rho'' i ti si tj''' sj''' w1 matchF_cond I t w1
      Some w2 = adv_end args w1
    define j_cur where j_cur = w_j w1
    define tj_cur where tj_cur = w_tj w1
    define sj_cur where sj_cur = w_sj w1
    define s_cur where s_cur = w_s w1
    define e_cur where e_cur = w_e w1
    have loop: valid_window args t0 sub (take (w_j w1) rho'') w1
      rw t0 sub rho'' (j_cur, tj_cur, sj_cur, length rho'', tj''', sj''')
      ⋀l. l∈{w_i w1..<w_j w1} ⟹ memR (ts_at rho'' i) (ts_at rho'' l) I
      using j_cur_def tj_cur_def sj_cur_def s_cur_def e_cur_def
        lassms(1)[unfolded matchF_loop_inv_def]
      by auto
    have j_cur_lt_rho'': j_cur < length rho''
      using lassms tj'''_eq ij_le before_end(4,5)
      unfolding matchF_loop_inv_def matchF_loop_cond_def
      by (cases j_cur = length rho'') (auto simp: j_cur_def split: option.splits)
    obtain tj_cur' sj_cur' x b_cur where tbi_cur_def: w_run_t args tj_cur = Some (tj_cur', x)
      w_run_sub args sj_cur = Some (sj_cur', b_cur)
      x = ts_at rho'' j_cur b_cur = bs_at rho'' j_cur
      using reach_window_run_ti[OF loop(2) j_cur_lt_rho'']
        reach_window_run_si[OF loop(2) j_cur_lt_rho'']
      by auto
    note tbi_cur_def' = tbi_cur_def(1,2)[unfolded tj_cur_def sj_cur_def]
    have length rho'' − w_j w2 < length rho'' − w_j w1
      using j_cur_lt_rho'' adv_end_bounds[OF tbi_cur_def', folded lassms(3)]
      unfolding j_cur_def
      by auto
  }
  then have {(ta, s). matchF_inv I t0 sub rho'' i ti si tj''' sj''' s ∧ matchF_cond I t s ∧
    Some ta = adv_end args s} ⊆ measure (λw. length rho'' − w_j w)
    by auto
  then show wf {(ta, s). matchF_inv I t0 sub rho'' i ti si tj''' sj''' s ∧ matchF_cond I t s ∧
    Some ta = adv_end args s}
    using wf_measure wf_subset
    by auto
qed (auto simp add: inv_before)
obtain w' where w'_def: while_break (matchF_cond I t) (adv_end args) w = Some w'
  using loop
  by (auto simp: pred_option'_def split: option.splits)
define w'' where adv_start_last: w'' = adv_start args w'
define st' where st' = w_st w'
define i' where i' = w_i w'
```

define ti' **where** $ti' = w\_ti\ w'$
define si' **where** $si' = w\_si\ w'$
define j' **where** $j' = w\_j\ w'$
define tj' **where** $tj' = w\_tj\ w'$
define sj' **where** $sj' = w\_sj\ w'$
define s' **where** $s' = w\_s\ w'$
define e' **where** $e' = w\_e\ w'$
have *valid_after*: *valid_window args t0 sub* (*take* (*w_j w'*) *rho''*) *w'*
rw t0 sub rho'' (*j'*, *tj'*, *sj'*, *length rho''*, *tj'''*, *sj'''*)
$\bigwedge l.\ l \in \{i..<j'\} \Longrightarrow memR\ (ts\_at\ rho''\ i)\ (ts\_at\ rho''\ l)\ I$
$i' = i\ ti' = ti\ si' = si$
using *loop*
unfolding *matchF_loop_inv_def w'_def i'_def ti'_def si'_def j'_def tj'_def sj'_def*
by (*auto simp*: *pred_option'_def*)
define i'' **where** $i'' = w\_i\ w''$
define j'' **where** $j'' = w\_j\ w''$
define tj'' **where** $tj'' = w\_tj\ w''$
define sj'' **where** $sj'' = w\_sj\ w''$
have j'_le_rho'': $j' \leq length\ rho''$
using *loop*
unfolding *matchF_loop_inv_def valid_window_matchF_def w'_def j'_def*
by (*auto simp*: *pred_option'_def*)
obtain *te* **where** *tbj'_def*: *w_read_t args tj' = Some te*
te = *ts_at* (*rho''* @ [(*tm*, *undefined*)]) *j'*
proof (*cases j' < length rho''*)
case *True*
show *?thesis*
using *reach_window_run_ti*[*OF valid_after*(*2*) *True*] *that True*
by (*auto simp*: *ts_at_def nth_append dest*!: *run_t_read*)
next
case *False*
then have $tj' = tj'''\ j' = length\ rho''$
using *valid_after*(*2*) *j'_le_rho'' tj'''_eq*
by *auto*
then show *?thesis*
using *that before_end*(*4*)
by (*auto simp*: *ts_at_def nth_append*)
qed
have *not_ets_te*: ¬*memR* (*ts_at rho''* *i*) *te I*
using *loop*
unfolding *w'_def*
by (*auto simp*: *pred_option'_def matchF_loop_cond_def tj'_def*[*symmetric*] *tbj'_def*(*1*) *tbi_def*(*3*) *split*: *option.splits*)
have *i'_set*: $\bigwedge l.\ l \in \{i..<j'\} \Longrightarrow memR\ (ts\_at\ rho''\ i)\ (ts\_at\ rho''\ l)\ I$
¬*memR* (*ts_at rho''* *i*) (*ts_at* (*rho''* @ [(*tm*, *undefined*)]) *j'*) *I*
using *loop tbj'_def not_ets_te valid_after atLeastLessThan_iff*
unfolding *matchF_loop_inv_def matchF_loop_cond_def tbi_def*(*3*)
by (*auto simp*: *tbi_def tj'_def split*: *option.splits*)
have i_le_j': $i \leq j'$
using *valid_after*(*1*)
unfolding *valid_after*(*4*)[*symmetric*]
by (*auto simp*: *valid_window_def Let_def i'_def j'_def*)
have i_lt_j': $i < j'$
using *i_le_j' i'_set*(*2*) *i_lt_rho''*
using *memR_tfin_refl*[*OF τ_fin*] *ts_at_tau*[*OF reach_tj''', of j'*]
by (*cases i = j'*) (*auto simp*: *ts_at_def nth_append*)
then have i'_lt_j': $i' < j'$
unfolding *valid_after*

by *auto*
have *adv_last_bounds*: $i'' = Suc\ i'\ w\_ti\ w'' = ti'''\ w\_si\ w'' = si'''\ j'' = j'$
$tj'' = tj'\ sj'' = sj'$
using *valid_adv_start_bounds*[*OF valid_after*(*1*) *i'_lt_j'*[*unfolded i'_def j'_def*]]
valid_adv_start_bounds'[*OF valid_after*(*1*) *tbi_def*(*1*,*2*)[*folded valid_after*(*5*,*6*),
unfolded ti'_def si'_def]]
unfolding *adv_start_last i'_def i''_def j'_def j''_def tj'_def tj''_def sj'_def sj''_def*
by *auto*
have *i''_i*: $i'' = i + 1$
using *valid_after adv_last_bounds* **by** *auto*
have *i_le_j'*: $i \leq j'$
using *valid_after i'_lt_j'*
by *auto*
then have *i_le_rho*: $i \leq length\ rho''$
using *valid_after*(*2*)
by *auto*
have *valid_s init step st' accept* (*take j' rho''*) *i i j' s'*
using *valid_after*(*1*,*4*) *i'_def*
by (*auto simp*: *valid_window_def Let_def init_def step_def st'_def accept_def j'_def s'_def*)
note *valid_s'* = *this*[*unfolded valid_s_def*]
have *q0_in_keys*: $\{0\} \in mmap\_keys\ s'$
using *valid_s' init_def steps_refl* **by** *auto*
then obtain *q' tstp* **where** *lookup_s'*: *mmap_lookup s'* $\{0\}$ = *Some* (*q'*, *tstp*)
by (*auto dest*: *Mapping_keys_dest*)
have *lookup_sup_acc*: *snd* (*the* (*mmap_lookup s'* $\{0\}$)) =
sup_acc step accept (*take j' rho''*) $\{0\}$ *i j'*
using *conjunct2*[*OF valid_s'*] *lookup_s'*
by *auto* (*smt case_prodD option.simps*(*5*))
have *b_alt*: (*case snd* (*the* (*mmap_lookup s'* $\{0\}$)) *of None* ⇒ *False*
| *Some tstp* ⇒ *memL t* (*fst tstp*) *I*) ⟷ *sat* (*MatchF I r*) *i*
proof (*rule iffI*)
assume *assm*: *case snd* (*the* (*mmap_lookup s'* $\{0\}$)) *of None* ⇒ *False*
| *Some tstp* ⇒ *memL t* (*fst tstp*) *I*
then obtain *ts tp* **where** *tstp_def*:
sup_acc step accept (*take j' rho''*) $\{0\}$ *i j'* = *Some* (*ts*, *tp*)
memL (*ts_at rho'' i*) *ts I*
unfolding *lookup_sup_acc*
by (*auto simp*: *tbi_def split*: *option.splits*)
then have *sup_acc_rho''*: *sup_acc step accept rho''* $\{0\}$ *i j'* = *Some* (*ts*, *tp*)
using *sup_acc_concat_cong*[*of j' take j' rho'' step accept drop j' rho''*] *j'_le_rho''*
by *auto*
have *tp_props*: *tp* ∈ $\{i..<j'\}$ *acc step accept rho''* $\{0\}$ (*i*, *Suc tp*)
using *sup_acc_SomeE*[*OF sup_acc_rho''*] **by** *auto*
have *ts_ts_at*: *ts* = *ts_at rho'' tp*
using *sup_acc_Some_ts*[*OF sup_acc_rho''*] **.**
have *i_le_tp*: *i* ≤ *Suc tp*
using *tp_props* **by** *auto*
have *memR* (*ts_at rho'' i*) (*ts_at rho'' tp*) *I*
using *i'_set*(*1*)[*OF tp_props*(*1*)] **.**
then have *mem* (*ts_at rho'' i*) (*ts_at rho'' tp*) *I*
using *tstp_def*(*2*) **unfolding** *ts_ts_at mem_def* **by** *auto*
then show *sat* (*MatchF I r*) *i*
using *i_le_tp acc_match*[*OF reach_sj''' i_le_tp _ wf*] *tp_props*(*2*) *ts_at_tau*[*OF reach_tj'''*]
tp_props(*1*) *j'_le_rho''*
by *auto*
next
assume *sat* (*MatchF I r*) *i*
then obtain *l* **where** *l_def*: *l* ≥ *i mem* (*τ σ i*) (*τ σ l*) *I* (*i*, *Suc l*) ∈ *match r*

```
    by auto
  have l_lt_rho: l < length rho''
  proof (rule ccontr)
    assume contr: ¬l < length rho''
    have tm = ts_at (rho'' @ [(tm, undefined)]) (length rho'')
      using i_le_rho
      by (auto simp add: ts_at_def rho''_def)
    moreover have ... ≤ τ σ l
      using τ_mono ts_at_tau[OF reach_tm] i_le_rho contr
      by (metis One_nat_def Suc_eq_plus1 length_append lessI list.size(3)
          list.size(4) not_le_imp_less)
    moreover have memR (τ σ i) (τ σ l) I
      using l_def(2)
      unfolding mem_def
      by auto
    ultimately have memR (τ σ i) tm I
      using memR_mono'
      by auto
    then show False
      using before_end' ts_at_tau[OF reach_tj''' i_lt_rho''] tbi_def(3)
      by (auto simp: rho''_def)
  qed
  have l_lt_j': l < j'
  proof (rule ccontr)
    assume contr: ¬l < j'
    then have ts_at_j'_l: ts_at rho'' j' ≤ ts_at rho'' l
      using ts_at_mono[OF reach_tj'''] l_lt_rho
      by (auto simp add: order.not_eq_order_implies_strict)
    have ts_at_l_iu: memR (ts_at rho'' i) (ts_at rho'' l) I
      using l_def(2) ts_at_tau[OF reach_tj''' l_lt_rho] ts_at_tau[OF reach_tj''' i_lt_rho'']
      unfolding mem_def
      by auto
    show False
      using i'_set(2) ts_at_j'_l ts_at_l_iu contr l_lt_rho memR_mono'
      by (auto simp: ts_at_def nth_append split: if_splits)
  qed
  have i_le_Suc_l: i ≤ Suc l
    using l_def(1)
    by auto
  obtain tp where tstp_def: sup_acc step accept rho'' {0} i j' = Some (ts_at rho'' tp, tp)
    l ≤ tp tp < j'
    using l_def(1,3) l_lt_j' l_lt_rho
  by (meson accept_match[OF reach_sj''' i_le_Suc_l _ wf, unfolded steps_is_run] sup_acc_SomeI[unfolded
acc_is_accept, of step accept] acc_is_accept atLeastLessThan_iff less_eq_Suc_le)
  have memL (ts_at rho'' i) (ts_at rho'' l) I
    using l_def(2)
    unfolding ts_at_tau[OF reach_tj''' i_lt_rho'', symmetric]
      ts_at_tau[OF reach_tj''' l_lt_rho, symmetric] mem_def
    by auto
  then have memL (ts_at rho'' i) (ts_at rho'' tp) I
    using ts_at_mono[OF reach_tj''' tstp_def(2)] tstp_def(3) j'_le_rho'' memL_mono'
    by auto
  then show case snd (the (mmap_lookup s' {0})) of None ⇒ False
    | Some tstp ⇒ memL t (fst tstp) I
    using lookup_sup_acc tstp_def j'_le_rho''
      sup_acc_concat_cong[of j' take j' rho'' step accept drop j' rho'']
    by (auto simp: tbi_def split: option.splits)
qed
```

```
  have valid_matchF I t0 sub (take j'' rho'') i'' (adv_start args w')
  proof −
    have ∀ l ∈ {i'..<j'}. memR (ts_at rho'' i') (ts_at rho'' l) I
      using loop i'_def j'_def valid_after
      unfolding matchF_loop_inv_def
      by auto
    then have ∀ l ∈ {i''..<j''}. memR (ts_at rho'' i'') ( ts_at rho'' l) I
      unfolding i''_i valid_after adv_last_bounds
      apply safe
      subgoal for l
        apply (drule ballE[of _ _ l])
        using ts_at_mono[OF reach_tj''', of i Suc i] j'_le_rho'' memR_mono
          apply auto
        done
      done
    moreover have rw t0 sub (take j'' rho'') (i'', ti''', si''', j'', tj'', sj'')
    proof −
      have rw: rw t0 sub (take j' rho'') (i', ti', si', j', tj', sj')
        using valid_after(1)
        by (auto simp: valid_window_def Let_def i'_def ti'_def si'_def j'_def tj'_def sj'_def)
      show ?thesis
        using reach_window_shift[OF rw i'_lt_j'
            tbi_def(1,2)[unfolded valid_after(5,6)[symmetric]]] adv_last_bounds
        by auto
    qed
    moreover have valid_window args t0 sub (take j' rho'') w''
      using valid_adv_start[OF valid_after(1) i'_lt_j'[unfolded i'_def j'_def]]
      unfolding adv_start_last j'_def .
    ultimately show valid_matchF I t0 sub (take j'' rho'') i'' (adv_start args w')
      using j'_le_rho''
    unfolding valid_window_matchF_def adv_last_bounds adv_start_last[symmetric] i''_def[symmetric]
      j'_def j''_def[symmetric] tj'_def tj''_def[symmetric] sj'_def sj''_def[symmetric]
      by (auto simp: ts_at_def)
  qed
  moreover have eval_mF I w = Some ((τ σ i, sat (MatchF I r) i), w'')
    unfolding j''_def adv_start_last[symmetric] adv_last_bounds valid_after rho''_def
      eval_matchF.simps run_t_read[OF tbi_def(1)[unfolded ti_def]]
    using tbj'_def[unfolded tj'_def] not_ets_te[folded tbi_def(3)]
      b_alt[unfolded s'_def] t_def adv_start_last w'_def
    by (auto simp only: Let_def split: option.splits if_splits)
  ultimately show ?thesis
    unfolding j''_def adv_start_last[symmetric] adv_last_bounds valid_after rho''_def
    by auto
qed

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 = τ σ i ∧ b = sat (MatchF I r) i ∧ (∃ rho'. valid_matchF I t0 sub rho' (Suc i) w'')
proof −
  obtain rho' t tm where rho'_def: 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
    using valid_eval_matchF_Some[OF assms(1−3)]
```

```
    by auto
  show ?thesis
    using valid_eval_matchF_complete[OF assms(1) rho'_def wf]
    unfolding eval
    by blast
qed

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 × '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 × ('a, 't, 'h) vydra_aux

fun msize_vydra :: nat ⇒ ('a, 't :: timestamp, 'h) vydra_aux ⇒ 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 ⇒ 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
  by (cases x) (auto split: option.splits)

lemma read_t_run: read_t x = Some t ⟹ ∃ x'. run_t x = Some (x', t)
  by (cases x) (auto split: option.splits)

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 (induction e vs e'' arbitrary: t' X' e''' rule: reaches_on_rev_induct)
  case (2 s s' v vs s'')
  obtain v_t v_X where v_def: v = (v_t, v_X)
    by (cases v) auto
  have run_t_s'': run_t (Some (s'', v_t)) = Some (Some (e''', t'), v_t)
    by (auto simp: 2(5))
  show ?case
    using reaches_on_app[OF 2(3)[OF 2(4) 2(2)[unfolded v_def]] run_t_s'']
    by (auto simp: v_def)
qed (auto intro: reaches_on.intros)

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 −
  have t0_not_None: t0 ≠ None
    apply (rule reaches_on.cases[OF assms(1)])
    using assms(2)
    by (auto simp: t0_def split: option.splits prod.splits)
  then show ?thesis
    using reach_event_t[OF assms(1) _ assms(2)]
    by (auto simp: t0_def split: option.splits)
qed

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 (cases vs rule: rev_cases)
  case (snoc ys y)
```

```
  show ?thesis
    using assms
    apply (cases y)
    apply (auto simp: snoc dest!: reaches_on_split_last)
    apply (meson reaches_on_app[OF reach_event_t0_t] read_t.simps(2) read_t_run)
    done
qed (auto intro: reaches_on.intros)

definition run_subs run = (λvs. 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
  by (auto simp: run_subs_def Let_def iarray_of_list_def split: option.splits if_splits)

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
  apply (cases run (vs ! j))
   apply (auto simp: Option.is_none_def run_subs_def Let_def iarray_of_list_def
      split: option.splits if_splits)
  by (metis image_eqI nth_mem)

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 (induction r)
  case (Lookahead phi)
  then show ?case
    by (auto split: if_splits)
next
  case (Symbol phi)
  then show ?case
    by (auto split: if_splits)
next
  case (Plus r1 r2)
  then show ?case
    by (auto simp: collect_subfmlas_set[of r2 collect_subfmlas r1 []])
next
  case (Times r1 r2)
  then show ?case
```

```
    by (auto simp: collect_subfmlas_set[of r2 collect_subfmlas r1 []])
next
  case (Star r)
  then show ?case
    by fastforce
qed

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
  by pat_completeness auto
termination
  by (relation (λp. size (fst p)) <*mlex*> (λp. next_vydra (snd p)) <*mlex*> (λp. msize_vydra (fst p)
(snd p)) <*mlex*> {}) (auto simp: mlex_prod_def)

lemma wf_since: wf {(t, s). while_since_cond I tt s ∧ Some t = while_since_body (run n) s}
proof −
  let ?X = {(t, s). while_since_cond I tt s ∧ Some t = while_since_body (run n) s}
  have sub: ?X ⊆ measure (λ(vpsi, e, cpsi, cppsi, tppsi). cpsi)
    by (auto simp: while_since_cond_def while_since_body_def Let_def split: option.splits)
  then show ?thesis
    using wf_subset[OF wf_measure]
    by auto
qed

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 ⇒ ('t × ('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)
  using run_hd_sound
  by (auto dest: reaches_on_n)

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 (induction t ts t' arbitrary: t'' x rule: reaches_on_rev_induct)
  case (1 s)
  show ?case
    using 1 reaches_on_run_hd[OF reaches_on.intros(1)]
    by (auto simp: t0_def split: option.splits intro!: reaches_on.intros)
next
  case (2 s s' v vs s'')
  obtain rho e tt where rho_def: s' = Some (e, tt) reaches_on run_hd init_hd rho e
    length rho = Suc (length vs)
    using 2(3)[OF 2(4,2)]
    by auto
  then show ?case
    using 2(2,5) reaches_on_app[OF rho_def(2)] reaches_on_run_hd[OF rho_def(2)]
    by (fastforce split: option.splits)
qed

lemma ru_t_tau: reaches_on ru_t l_t0 ts t' ⟹ ru_t t' = Some (t'', x) ⟹ x = τ σ (length ts)
  using ru_t_event
  by fastforce

lemma ru_t_Some_tau:
```

assumes *reaches_on ru_t l_t0 ts* (*Some* (*e*, *t*))
shows $t = \tau\ \sigma$ (*length ts*)
proof −
obtain *z* **where** *z_def*: *ru_t* (*Some* (*e*, *t*)) = *Some* (*z*, *t*)
by (*cases run_hd e*) *auto*
show *?thesis*
by (*rule ru_t_tau*[*OF assms z_def*])
qed

lemma *ru_t_tau_in*:
assumes *reaches_on ru_t l_t0 ts t j* < *length ts*
shows *ts* ! *j* = $\tau\ \sigma$ *j*
proof −
obtain t' **where** t'*_def*: *reaches_on ru_t l_t0* (*take j ts*) t' *reaches_on ru_t t'* (*drop j ts*) *t*
using *reaches_on_split'*[*OF assms*(*1*), **where** *?i*=*j*] *assms*(*2*)
by *auto*
have *drop*: *drop j ts* = *ts* ! *j* # *tl* (*drop j ts*)
using *assms*(*2*)
by (*cases drop j ts*) (*auto simp add*: *nth_via_drop*)
obtain t'' **where** t''*_def*: *ru_t t'* = *Some* (t'', *ts* ! *j*)
using t'*_def*(*2*) *assms*(*2*) *drop*
by (*auto elim*: *reaches_on.cases*)
show *?thesis*
using *ru_t_event*[*OF t'_def*(*1*) *refl t''_def*] *assms*(*2*)
by *auto*
qed

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 (*induction P n rule*: *last_before.induct*)
case (*2 P n*)
then show *?case*
by (*cases m* = *n*) (*auto split*: *if_splits*)
qed (*auto split*: *if_splits*)

lemma *last_before_Some*: *last_before P n* = *Some m* ⟹ *m* < *n* ∧ *P m* ∧ (∀ *k* ∈ {*m*<..<*n*}. ¬*P k*)
apply (*induction P n rule*: *last_before.induct*)
apply (*auto split*: *if_splits*)
apply (*metis greaterThanLessThan_iff less_antisym*)
done

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* ($\tau\ \sigma$ *j*, *sat phi j*)))

| *wf_vydra phi i n v* $\Longrightarrow$ *reaches_on run_hd init_hd es sub'* $\Longrightarrow$ *length es = i* $\Longrightarrow$
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* $\Longrightarrow$ *wf_vydra psi j n vpsi* $\Longrightarrow$ *j ≤ i* $\Longrightarrow$
reaches_on ru_t l_t0 es sub' $\Longrightarrow$ *length es = j* $\Longrightarrow$ *(⋀t. t ∈ set es* $\Longrightarrow$ *memL t (τ σ i) I)* $\Longrightarrow$
cphi = i − (case last_before (λk. ¬sat phi k) i of None ⇒ 0 | Some k ⇒ Suc k) $\Longrightarrow$ *cpsi = i − j* $\Longrightarrow$
cppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (i − k)) $\Longrightarrow$
tppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (τ σ k)) $\Longrightarrow$
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* $\Longrightarrow$ *wf_vydra psi j n vpsi* $\Longrightarrow$ *i ≤ j* $\Longrightarrow$
reaches_on ru_t l_t0 es back $\Longrightarrow$ *length es = i* $\Longrightarrow$
reaches_on ru_t l_t0 es' front $\Longrightarrow$ *length es' = j* $\Longrightarrow$ *(⋀t. t ∈ set es'* $\Longrightarrow$ *memR (τ σ i) t I)* $\Longrightarrow$
c = j − i $\Longrightarrow$ *z = (case j of 0 ⇒ None | Suc k ⇒ Some (τ σ k, sat phi k, sat psi k))* $\Longrightarrow$
(⋀k. k ∈ {i..<j − 1} $\Longrightarrow$ *sat phi k ∧ (memL (τ σ i) (τ σ k) I ⟶ ¬sat psi k))* $\Longrightarrow$
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* $\Longrightarrow$
n ≥ msize_regex r $\Longrightarrow$ *qf = state_cnt r* $\Longrightarrow$
transs = iarray_of_list (build_nfa_impl r (0, qf, [])) $\Longrightarrow$
args = init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
(ru_t, read_t) (run_subs (ru n)) $\Longrightarrow$
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* $\Longrightarrow$
n ≥ msize_regex r $\Longrightarrow$ *qf = state_cnt r* $\Longrightarrow$
transs = iarray_of_list (build_nfa_impl r (0, qf, [])) $\Longrightarrow$
args = init_args ({0}, NFA.delta' transs qf, NFA.accept' transs qf)
(ru_t, read_t) (run_subs (ru n)) $\Longrightarrow$
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 [])*
using *reaches_ons run_subs_lD*
by (*induction map (su n) (collect_subfmlas r []) rho vs rule*: *reaches_on_rev_induct*) *fastforce+*

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*
using *reaches_ons subfmla*
proof (*induction map (su n) (collect_subfmlas r []) rho vs rule*: *reaches_on_rev_induct*)
case *1*
then show *?case*
by (*auto intro*: *reaches_on.intros*)
next
case (*2 s' v vs' s''*)
note *len_s'_vs = reach_run_subs_len[OF 2(1)]*
obtain *rho'* **where** *reach_s'_vs*: *reaches_on (ru n) (su n phi) rho' (s' ! j)*
length rho' = length vs'
using *2(2)[OF 2(4,5)]*
by *auto*
note *run_subslD = run_subs_lD[OF 2(3)]*
note *run_subsvD = run_subs_vD[OF 2(3) 2(4)[unfolded len_s'_vs[symmetric]]]*
obtain *vj' tj bj* **where** *vj'_def*: *ru n (s' ! j) = Some (vj', tj, bj)*
s'' ! j = vj' IArray.sub v j = bj
using *run_subsvD* **by** *auto*
obtain *rho''* **where** *rho''_def*: *reaches_on (ru n) (su n phi) rho'' (s'' ! j)*
length rho'' = Suc (length vs')
using *reaches_on_app[OF reach_s'_vs(1) vj'_def(1)] vj'_def(2) reach_s'_vs(2)*

```
    by auto
  then show ?case
    using conjunct1[OF run_subslD, unfolded len_s'_vs[symmetric]]
    by auto
qed

lemma IArray_nth_equalityI: IArray.length xs = length ys ⟹
  (⋀i. i < IArray.length xs ⟹ IArray.sub xs i = ys ! i) ⟹ xs = IArray ys
  by (induction xs arbitrary: ys) (auto intro: nth_equalityI)

lemma bs_sat:
  assumes IH: ⋀phi i v v' b. phi ∈ set (collect_subfmlas r []) ⟹ wf_vydra phi i n v ⟹ ru n v = Some
(v', b) ⟹ snd b = sat phi i
    and reaches_ons: ⋀j. j < length (collect_subfmlas r []) ⟹ 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 (λphi. sat phi i) (collect_subfmlas r []))
proof −
  have ⋀j. j < length (collect_subfmlas r []) ⟹
    IArray.sub bs j = sat (collect_subfmlas r [] ! j) i
  proof −
    fix j
    assume lassm: j < length (collect_subfmlas r [])
    define phi where phi = collect_subfmlas r [] ! j
    have phi_in_set: phi ∈ set (collect_subfmlas r [])
      using lassm
      by (auto simp: phi_def)
    have wf: wf_vydra phi i n (vs ! j)
      using reaches_ons lassm phi_def
      by metis
    show IArray.sub bs j = sat (collect_subfmlas r [] ! j) i
      using IH(1)[OF phi_in_set wf] run_subs_vD[OF run_subs(1) lassm[folded run_subs(2)]]
      unfolding phi_def[symmetric]
      by auto
  qed
  moreover have length (IArray.list_of bs) = length vs
    using run_subs(1)
    by (auto simp: run_subs_def Let_def iarray_of_list_def split: if_splits)
  ultimately show ?thesis
    using run_subs(2)
    by (auto simp: iarray_of_list_def intro!: IArray_nth_equalityI)
qed

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 ≤ n (⋀b n. P n (Bool b)) (⋀a n. P n (Atom a))
    (⋀n phi. msize_fmla phi ≤ n ⟹ P n phi ⟹ P (Suc n) (Neg phi))
    (⋀n f phi psi. msize_fmla (Bin f phi psi) ≤ Suc n ⟹ P n phi ⟹ P n psi ⟹
      P (Suc n) (Bin f phi psi))
    (⋀n I phi. msize_fmla phi ≤ n ⟹ P n phi ⟹ P (Suc n) (Prev I phi))
    (⋀n I phi. msize_fmla phi ≤ n ⟹ P n phi ⟹ P (Suc n) (Next I phi))
    (⋀n I phi psi. msize_fmla phi ≤ n ⟹ msize_fmla psi ≤ n ⟹ P n phi ⟹ P n psi ⟹ P (Suc n)
(Since phi I psi))
    (⋀n I phi psi. msize_fmla phi ≤ n ⟹ msize_fmla psi ≤ n ⟹ P n phi ⟹ P n psi ⟹ P (Suc n)
(Until phi I psi))
    (⋀n I r. msize_fmla (MatchP I r) ≤ Suc n ⟹ (⋀x. msize_fmla x ≤ n ⟹ P n x) ⟹
      P (Suc n) (MatchP I r))
```

```
    (⋀n I r. msize_fmla (MatchF I r) ≤ Suc n ⟹ (⋀x. msize_fmla x ≤ n ⟹ P n x) ⟹
      P (Suc n) (MatchF I r))
  shows P n phi
  using assms(1)
proof (induction n arbitrary: phi rule: nat_less_induct)
  case (1 n)
  show ?case
  proof (cases n)
    case 0
    show ?thesis
      using 1 assms(2−)
      by (cases phi) (auto simp: 0)
  next
    case (Suc m)
    show ?thesis
      using 1 assms(2−)
      by (cases phi) (auto simp: Suc)
  qed
qed

lemma wf_vydra_sub: msize_fmla φ ≤ n ⟹ wf_vydra φ 0 n (su n φ)
proof (induction n φ rule: run_induct)
  case (Prev n I phi)
  then show ?case
    using wf_vydra.intros(7)[where ?i=0, OF _ reaches_on.intros(1)]
    by auto
next
  case (Next n I phi)
  then show ?case
    using wf_vydra.intros(8)[where ?i=0, OF _ reaches_on.intros(1)]
    by auto
next
  case (MatchP n I r)
  let ?qf = state_cnt r
  let ?transs = iarray_of_list (build_nfa_impl r (0, ?qf, []))
  let ?args = init_args ({0}, NFA.delta' ?transs ?qf, NFA.accept' ?transs ?qf) (ru_t, read_t) (run_subs
(ru n))
  show ?case
    using MatchP valid_init_window[of ?args l_t0 map (su n) (collect_subfmlas r []), simplified]
    by (auto simp: Let_def valid_window_matchP_def split: option.splits intro: reaches_on.intros
        intro!: wf_vydra.intros(11)[where ?xs=[], OF _ _ refl refl refl])
next
  case (MatchF n I r)
  let ?qf = state_cnt r
  let ?transs = iarray_of_list (build_nfa_impl r (0, ?qf, []))
  let ?args = init_args ({0}, NFA.delta' ?transs ?qf, NFA.accept' ?transs ?qf) (ru_t, read_t) (run_subs
(ru n))
  show ?case
    using MatchF valid_init_window[of ?args l_t0 map (su n) (collect_subfmlas r []), simplified]
    by (auto simp: Let_def valid_window_matchF_def split: option.splits intro: reaches_on.intros
        intro!: wf_vydra.intros(12)[where ?xs=[], OF _ _ refl refl refl])
qed (auto simp: Let_def intro: wf_vydra.intros reaches_on.intros)

lemma ru_t_Some: ∃ 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 ≤ i for es e
proof −
  obtain fs' ft where ft_def: reaches_on ru_t l_t0 (map fst (fs' :: ('t × 'a set) list)) (Some (f, ft))
```

```
    map fst fs = map fst fs' @ [ft] length fs' = i
    using reaches_Suc_i
    by (cases fs rule: rev_cases) (auto dest!: reaches_on_split_last reach_event_t0_t)
  show ?thesis
  proof (cases length es = i)
    case True
    have e_def: e = Some (f, ft)
      using reaches_on_inj[OF aux(1) ft_def(1)]
      by (auto simp: True ft_def(3))
    then show ?thesis
      by (cases run_hd f) (auto simp: e_def)
  next
    case False
    obtain s' s'' where split: reaches_on ru_t l_t0 (take (length es) (map fst fs')) s'
      ru_t s' = Some (s'', map fst fs' ! (length es))
      using reaches_on_split[OF ft_def(1), where ?i=length es] False aux(2)
      by (auto simp: ft_def(3))
    show ?thesis
      using reaches_on_inj[OF aux(1) split(1)] aux(2)
      by (auto simp: ft_def(3) split(2))
  qed
qed

lemma vydra_sound_aux:
  assumes msize_fmla φ ≤ n wf_vydra φ i n v ru n v = Some (v', t, b) bounded_future_fmla φ wf_fmla
φ
  shows wf_vydra φ (Suc i) n v' ∧ (∃ es e. reaches_on run_hd init_hd es e ∧ length es = Suc i) ∧ t =
τ σ i ∧ b = sat φ i
  using assms
proof (induction n φ arbitrary: i v v' t b rule: run_induct)
  case (Bool β n)
  then show ?case
    using reaches_on_run_hd reaches_on_app wf_vydra.intros(3)[OF reaches_on_app refl]
    by (fastforce elim!: wf_vydra.cases[of _ _ _ v] split: option.splits)
next
  case (Atom a n)
  then show ?case
    using reaches_on_run_hd reaches_on_app wf_vydra.intros(4)[OF reaches_on_app refl]
    by (fastforce elim!: wf_vydra.cases[of _ _ _ v] split: option.splits)
next
  case (Neg n x)
  have IH: wf_vydra x i n v ⟹ ru n v = Some (v', t, b) ⟹ wf_vydra x (Suc i) n v' ∧ (∃ es e.
reaches_on run_hd init_hd es e ∧ length es = Suc i) ∧ t = τ σ i ∧ b = sat x i for v v' t b
    using Neg(2,5,6)
    by auto
  show ?case
    apply (rule wf_vydra.cases[OF Neg(3)])
    using Neg(4) IH wf_vydra.intros(5)
    by (fastforce split: option.splits)+
next
  case (Bin n f x1 x2)
  have IH1: wf_vydra x1 i n v ⟹ ru n v = Some (v', t, b) ⟹ wf_vydra x1 (Suc i) n v' ∧ (∃ es e.
reaches_on run_hd init_hd es e ∧ length es = Suc i) ∧ t = τ σ i ∧ b = sat x1 i for v v' t b
    using Bin(2,6,7)
    by auto
  have IH2: wf_vydra x2 i n v ⟹ ru n v = Some (v', t, b) ⟹ wf_vydra x2 (Suc i) n v' ∧ t = τ σ i
∧ b = sat x2 i for v v' t b
    using Bin(3,6,7)
```

```
    by auto
  show ?case
    apply (rule wf_vydra.cases[OF Bin(4)])
    using Bin(5) IH1 IH2 wf_vydra.intros(6)
    by (fastforce split: option.splits)+
next
  case (Prev n I phi)
  show ?case
  proof (cases i)
    case 0
    then show ?thesis
     using Prev run_hd_sound[OF reaches.intros(1)] wf_vydra.intros(7)[OF _ reaches_on.intros(2)[OF
_ reaches_on.intros(1)], where ?i=Suc 0, simplified]
      by (fastforce split: nat.splits option.splits dest!: reaches_on_NilD elim!: wf_vydra.cases[of _ _ _ v]
intro: wf_vydra.intros(1) reaches_on.intros(2)[OF _ reaches_on.intros(1)])
  next
    case (Suc j)
    obtain vphi es sub where v_def: v = VYDRA_Prev I vphi sub (Some (τ σ j, sat phi j))
      wf_vydra phi i n vphi reaches_on run_hd init_hd es sub length es = i
      using Prev(3,4)
      by (auto simp: Suc elim!: wf_vydra.cases[of _ _ _ v])
    obtain sub' X where run_sub: run_hd sub = Some (sub', (t, X))
      using Prev(4)
      by (auto simp: v_def(1) Let_def split: option.splits)
    note reaches_sub' = reaches_on_app[OF v_def(3) run_sub]
    have t_def: t = τ σ (Suc j)
      using reaches_on_run_hd[OF v_def(3) run_sub]
      by (auto simp: Suc v_def(2,4))
    show ?thesis
    proof (cases v' = VYDRA_None)
      case v'_def: True
      show ?thesis
        using Prev(4) v_def(2) reaches_sub'
      by (auto simp: Suc Let_def v_def(1,4) v'_def run_sub t_def split: option.splits intro: wf_vydra.intros(1))
    next
      case False
      obtain vphi' where ru_vphi: ru n vphi = Some (vphi', (τ σ i, sat phi i))
        using Prev(2)[OF v_def(2)] Prev(4,5,6) False
        by (auto simp: v_def(1) Let_def split: option.splits)
      have wf': wf_vydra phi (Suc (Suc j)) n vphi'
        using Prev(2)[OF v_def(2) ru_vphi] Prev(5,6)
        by (auto simp: Suc)
      show ?thesis
        using Prev(4) wf_vydra.intros(7)[OF wf' reaches_sub'] reaches_sub'
        by (auto simp: Let_def Suc t_def v_def(1,4) run_sub ru_vphi)
    qed
  qed
next
  case (Next n I phi)
  obtain w sub to es where v_def: v = VYDRA_Next I w sub to wf_vydra phi (length es) n w
    reaches_on run_hd init_hd es sub length es = (case to of None ⇒ 0 | _ ⇒ Suc i)
    case to of None ⇒ i = 0 | Some told ⇒ told = τ σ i
    using Next(3,4)
    by (auto elim!: wf_vydra.cases[of _ _ _ v] split: option.splits nat.splits)
  obtain sub' tnew X where run_sub: run_hd sub = Some (sub', (tnew, X))
    using Next(4)
    by (auto simp: v_def(1) split: option.splits)
  have tnew_def: tnew = τ σ (length es)
```

```
    using reaches_on_run_hd[OF v_def(3) run_sub]
    by auto
  have aux: ?case if aux_assms: wf_vydra phi (Suc i) n w
    ru (Suc n) (VYDRA_Next I w sub (Some t0)) = Some (v', t, b)
    reaches_on run_hd init_hd es sub length es = Suc i t0 = τ σ i for w sub t0 es
    using aux_assms(1,2,5) wf_vydra.intros(2)[OF aux_assms(1)]
      Next(2)[where ?i=Suc i and ?v=w] Next(5,6) reaches_on_run_hd[OF aux_assms(3)]
      wf_vydra.intros(8)[OF _ reaches_on_app[OF aux_assms(3)], where ?phi=phi and ?i=Suc (Suc
i) and ?n=n] aux_assms(3)
    by (auto simp: run_sub aux_assms(4,5) split: option.splits if_splits)
  show ?case
  proof (cases to)
    case None
    obtain w' z where w_def: ru (Suc n) v = ru (Suc n) (VYDRA_Next I w' sub' (Some tnew))
      ru n w = Some (w', z)
      using Next(4)
      by (cases ru n w) (auto simp: v_def(1) run_sub None split: option.splits)
    have wf: wf_vydra phi (Suc i) n w'
      using v_def w_def(2) Next(2,5,6)
      by (cases z) (auto simp: None intro: wf_vydra.intros(1))
    show ?thesis
      using aux[OF wf Next(4)[unfolded w_def(1)] reaches_on_app[OF v_def(3) run_sub]] v_def(4,5)
tnew_def
      by (auto simp: None)
  next
    case (Some z)
    show ?thesis
      using aux[OF _ _ v_def(3), where ?w=w] v_def(2,4,5) Next(4)
      by (auto simp: v_def(1) Some simp del: run.simps)
  qed
next
  case (Since n I phi psi)
  obtain vphi vpsi e cphi cpsi cppsi tppsi j es where v_def:
    v = VYDRA_Since I vphi vpsi e cphi cpsi cppsi tppsi
    wf_vydra phi i n vphi wf_vydra psi j n vpsi j ≤ i
    reaches_on ru_t l_t0 es e 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))
    using Since(5)
    by (auto elim: wf_vydra.cases)
  obtain vphi' b1 where run_vphi: ru n vphi = Some (vphi', t, b1)
    using Since(6)
    by (auto simp: v_def(1) Let_def split: option.splits)
  obtain fs f where wf_vphi': wf_vydra phi (Suc i) n vphi'
    and reaches_Suc_i: reaches_on run_hd init_hd fs f length fs = Suc i
    and t_def: t = τ σ i and b1_def: b1 = sat phi i
    using Since(3)[OF v_def(2) run_vphi] Since(7,8)
    by auto
  note ru_t_Some = ru_t_Some[OF reaches_Suc_i]
  define loop_inv where loop_inv = (λ(vpsi, e, cpsi :: nat, cppsi, tppsi).
    let j = Suc i − cpsi in cpsi ≤ Suc i ∧
    wf_vydra psi j n vpsi ∧ (∃ es. reaches_on ru_t l_t0 es e ∧ length es = j ∧ (∀ t ∈ set es. memL t (τ
σ i) I)) ∧
    cppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (Suc i − k)) ∧
    tppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (τ σ k)))
  define loop_init where loop_init = (vpsi, e, Suc cpsi, map_option Suc cppsi, tppsi)
 obtain vpsi' e' cpsi' cppsi' tppsi' where loop_def: while_break (while_since_cond I t) (while_since_body
```

```
run_hd (ru n)) loop_init =
    Some (vpsi', e', cpsi', cppsi', tppsi')
    using Since(6)
    by (auto simp: v_def(1) run_vphi loop_init_def Let_def split: option.splits)
  have j_def: j = i − cpsi
    using v_def(4,9)
    by auto
  have cpsi ≤ i
    using v_def(9)
    by auto
  then have loop_inv_init: loop_inv loop_init
    using v_def(3,5,6,7,10,11) last_before_Some
    by (fastforce simp: loop_inv_def loop_init_def Let_def j_def split: option.splits)
  have wf_loop: wf {(s', s). loop_inv s ∧ while_since_cond I t s ∧ Some s' = while_since_body run_hd
(ru n) s}
    by (auto intro: wf_subset[OF wf_since])
  have step_loop: loop_inv s' if loop_assms: loop_inv s while_since_cond I t s while_since_body run_hd
(ru n) s = Some s' for s s'
  proof −
    obtain vpsi e cpsi cppsi tppsi where s_def: s = (vpsi, e, cpsi, cppsi, tppsi)
      by (cases s) auto
    define j where j = Suc i − cpsi
    obtain es where loop_before: cpsi ≤ Suc i wf_vydra psi j n vpsi
      reaches_on ru_t l_t0 es e length es = j ⋀t. t ∈ set es ⟹ memL t (τ σ i) I
      cppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (Suc i − k))
      tppsi = (case last_before (sat psi) j of None ⇒ None | Some k ⇒ Some (τ σ k))
      using loop_assms(1)
      by (auto simp: s_def j_def loop_inv_def Let_def)
    obtain tt h where tt_def: read_t e = Some tt memL tt t I e = Some (h, tt)
      using ru_t_Some[OF loop_before(3)] loop_before(4) loop_assms(2)
      by (cases e) (fastforce simp: while_since_cond_def s_def j_def split: option.splits)+
    obtain e' where e'_def: reaches_on ru_t l_t0 (es @ [tt]) e' ru_t e = Some (e', tt)
      using reaches_on_app[OF loop_before(3)] tt_def(1)
      by (cases run_hd h) (auto simp: tt_def(3))
    obtain vpsi' t' b' where run_vpsi: ru n vpsi = Some (vpsi', (t', b'))
      using loop_assms(3)
      by (auto simp: while_since_body_def s_def Let_def split: option.splits)
    have wf_psi': wf_vydra psi (Suc j) n vpsi' and t'_def: t' = τ σ j and b'_def: b' = sat psi j
      using Since(4)[OF loop_before(2) run_vpsi] Since(7,8)
      by auto
    define j' where j'_def: j' = Suc i − (cpsi − Suc 0)
    have cpsi_pos: cpsi > 0
      using loop_assms(2)
      by (auto simp: while_since_cond_def s_def)
    have j'_j: j' = Suc j
      using loop_before(1) cpsi_pos
      by (auto simp: j'_def j_def)
    define cppsi' where cppsi' = (if b' then Some cpsi else cppsi)
    define tppsi' where tppsi' = (if b' then Some t' else tppsi)
    have cppsi': cppsi' = (case last_before (sat psi) j' of None ⇒ None | Some k ⇒ Some (Suc i − k))
      using cpsi_pos loop_before(1)
      by (auto simp: cppsi'_def b'_def j'_j loop_before(6) j_def)
    have tppsi': tppsi' = (case last_before (sat psi) j' of None ⇒ None | Some k ⇒ Some (τ σ k))
      by (auto simp: tppsi'_def t'_def b'_def j'_j loop_before(7) split: option.splits)
    have s'_def: s' = (vpsi', fst (the (ru_t e)), cpsi − Suc 0, cppsi', tppsi')
      using loop_assms(3)
      by (auto simp: while_since_body_def s_def run_vpsi cppsi'_def tppsi'_def)
    show ?thesis
```

using *loop_before(1,4,5) tt_def(2) wf_psi'[folded j'_j] cppsi' tppsi' e'_def(1)*
by (*fastforce simp*: *loop_inv_def s'_def j'_def[symmetric] e'_def(2) j'_j t_def*)
qed
have *loop*: *loop_inv* (*vpsi'*, *e'*, *cpsi'*, *cppsi'*, *tppsi'*) ¬*while_since_cond I t* (*vpsi'*, *e'*, *cpsi'*, *cppsi'*, *tppsi'*)
using *while_break_sound*[**where** *?P=loop_inv* **and** *?Q=λs. loop_inv s ∧ ¬while_since_cond I t s*, *OF step_loop _ wf_loop loop_inv_init*]
by (*auto simp*: *loop_def*)
define *cphi'* **where** *cphi'* = (*if b1 then Suc cphi else 0*)
have *v'_def*: *v'* = *VYDRA_Since I vphi' vpsi' e' cphi' cpsi' cppsi' tppsi'*
and *b_def*: *b* = (*case cppsi' of None ⇒ False | Some k ⇒ k − 1 ≤ cphi' ∧ memR* (*the tppsi'*) *t I*)
using *Since(6)*
by (*auto simp*: *v_def(1) run_vphi loop_init_def*[*symmetric*] *loop_def cphi'_def Let_def split*: *option.splits*)
have *read_t_e'*: *cpsi'* > *0* ⟹ *read_t e'* = *None* ⟹ *False*
using *loop(1) ru_t_Some*[**where** *?e=e'*] *run_t_read*
by (*fastforce simp*: *loop_inv_def Let_def*)
define *j'* **where** *j'* = *Suc i* − *cpsi'*
have *wf_vpsi'*: *wf_vydra psi j' n vpsi'* **and** *cpsi'_le_Suc_i*: *cpsi'* ≤ *Suc i*
and *cppsi'_def*: *cppsi'* = (*case last_before* (*sat psi*) *j' of None ⇒ None | Some k ⇒ Some* (*Suc i* − *k*))
and *tppsi'_def*: *tppsi'* = (*case last_before* (*sat psi*) *j' of None ⇒ None | Some k ⇒ Some* (*τ σ k*))
using *loop(1)*
by (*auto simp*: *loop_inv_def j'_def*[*symmetric*])
obtain *es'* **where** *es'_def*: *reaches_on ru_t l_t0 es' e' length es'* = *j'* ⋀*t*. *t* ∈ *set es'* ⟹ *memL t* (*τ σ i*) *I*
using *loop(1)*
by (*auto simp*: *loop_inv_def j'_def*[*symmetric*])
have *wf_v'*: *wf_vydra* (*Since phi I psi*) (*Suc i*) (*Suc n*) *v'*
and *cphi'_sat*: *cphi'* = *Suc i* − (*case last_before* (*λk. ¬sat phi k*) (*Suc i*) *of None ⇒ 0 | Some k ⇒ Suc k*)
using *cpsi'_le_Suc_i last_before_Some es'_def(3) memL_mono'*[*OF _ τ_mono*[*of i Suc i σ*]]
by (*force simp*: *v'_def cppsi'_def tppsi'_def j'_def cphi'_def b1_def v_def(8) split*: *option.splits intro*!: *wf_vydra.intros(9)*[*OF wf_vphi' wf_vpsi' _ es'_def(1−2)*])+
have *j'* = *Suc i* ∨ ¬*memL* (*τ σ j'*) (*τ σ i*) *I*
using *loop(2) j'_def read_t_e' ru_t_tau*[*OF es'_def(1)*] *read_t_run*[**where** *?run_hd=run_hd*]
by (*fastforce simp*: *while_since_cond_def es'_def(2) t_def split*: *option.splits*)
then have *tau_k_j'*: *k* ≤ *i* ⟹ *memL* (*τ σ k*) (*τ σ i*) *I* ⟷ *k* < *j'* **for** *k*
using *ru_t_tau_in*[*OF es'_def(1)*] *es'_def(3) τ_mono*[*of j' k σ*] *memL_mono*
by (*fastforce simp*: *es'_def(2) in_set_conv_nth*)
have *b_sat*: *b* = *sat* (*Since phi I psi*) *i*
proof (*rule iffI*)
assume *b*: *b*
obtain *m* **where** *m_def*: *last_before* (*sat psi*) *j'* = *Some m i* − *m* ≤ *cphi' memR* (*τ σ m*) (*τ σ i*) *I*
using *b*
by (*auto simp*: *b_def t_def cppsi'_def tppsi'_def split*: *option.splits*)
note *aux* = *last_before_Some*[*OF m_def(1)*]
have *mem*: *mem* (*τ σ m*) (*τ σ i*) *I*
using *m_def(3) tau_k_j' aux*
by (*auto simp*: *mem_def j'_def*)
have *sat_phi*: *sat phi x* **if** *m* < *x x* ≤ *i* **for** *x*
using *m_def(2) that le_neq_implies_less*
by (*fastforce simp*: *cphi'_sat dest*: *last_before_None last_before_Some split*: *option.splits if_splits*)
show *sat* (*Since phi I psi*) *i*
using *aux mem sat_phi*
by (*auto simp*: *j'_def intro*!: *exI*[*of _ m*])
next
assume *sat*: *sat* (*Since phi I psi*) *i*
then obtain *k* **where** *k_def*: *k* ≤ *i mem* (*τ σ k*) (*τ σ i*) *I sat psi k* ⋀*k'*. *k* < *k'* ∧ *k'* ≤ *i* ⟹ *sat phi*

```
k'
      by auto
    have k_j': k < j'
      using tau_k_j'[OF k_def(1)] k_def(2)
      by (auto simp: mem_def)
    obtain m where m_def: last_before (sat psi) j' = Some m
      using last_before_None[where ?P=sat psi and ?n=j' and ?m=k] k_def(3) k_j'
      by (cases last_before (sat psi) j') auto
    have cppsi'_Some: cppsi' = Some (Suc i - m)
      by (auto simp: cppsi'_def m_def)
    have tppsi'_Some: tppsi' = Some (τ σ m)
      by (auto simp: tppsi'_def m_def)
    have m_k: k ≤ m
      using last_before_Some[OF m_def] k_def(3) k_j'
      by auto
    have tau_i_m: memR (τ σ m) (τ σ i) I
      using τ_mono[OF m_k, where ?s=σ] memR_mono k_def(2)
      by (auto simp: mem_def)
    have i - m ≤ cphi'
      using k_def(1) k_def(4) m_k
      apply (cases k = i)
       apply (auto simp: cphi'_sat b1_def dest!: last_before_Some split: option.splits)
      apply (metis diff_le_mono2 le_neq_implies_less le_trans less_imp_le_nat nat_le_linear)
      done
    then show b
      using tau_i_m
      by (auto simp: b_def t_def cppsi'_Some tppsi'_Some)
  qed
  show ?case
    using wf_v' reaches_Suc_i
    by (auto simp: t_def b_sat)
next
  case (Until n I phi psi)
  obtain back vphi vpsi front c z es es' j where v_def:
    v = VYDRA_Until I back vphi vpsi front c z
    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)
    using Until(5)
    by (auto elim: wf_vydra.cases)
  define loop_init where loop_init = (vphi, vpsi, front, c, z)
  obtain back' vphi' vpsi' epsi' c' zo' zt zb1 zb2 where run_back: ru_t back = Some (back', t)
    and loop_def: while_break (while_until_cond I t) (while_until_body run_hd (ru n)) loop_init =
Some (vphi', vpsi', epsi', c', zo')
    and v'_def: v' = VYDRA_Until I back' vphi' vpsi' epsi' (c' - 1) zo'
    and c'_pos: ¬c' = 0
    and zo'_Some: zo' = Some (zt, (zb1, zb2))
    and b_def: b = (zb2 ∧ memL t zt I)
    using Until(6)
    apply (auto simp: v_def(1) Let_def loop_init_def[symmetric] split: option.splits nat.splits if_splits)
    done
  define j' where j' = i + c'
  have j_eq: j = i + c
    using v_def(4)
    by (auto simp: v_def(10))
  have t_def: t = τ σ i
```

using *ru_t_tau*[*OF v_def*(*5*) *run_back*]
by (*auto simp*: *v_def*(*6*))
define *loop_inv* **where** *loop_inv* = (λ(*vphi*, *vpsi*, *epsi*, *c*, *zo*).
let j = *i* + *c in*
wf_vydra phi j n vphi ∧ *wf_vydra psi j n vpsi* ∧
(∃ *gs*. *reaches_on ru_t l_t0 gs epsi* ∧ *length gs* = *j* ∧ (∀ *t*. *t* ∈ *set gs* ⟶ *memR* (*τ σ i*) *t I*)) ∧
zo = (*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*)))
have *loop_inv_init*: *loop_inv loop_init*
using *v_def*(*2*,*3*,*7*,*9*,*12*)
by (*auto simp*: *loop_inv_def loop_init_def j_eq*[*symmetric*] *v_def*(*8*,*11*))
have *loop_step*: *loop_inv s'* **if** *loop_assms*: *loop_inv s while_until_cond I t s while_until_body run_hd* (*ru n*) *s* = *Some s'* **for** *s s'*
proof −
obtain *vphi_cur vpsi_cur epsi_cur c_cur zo_cur* **where** *s_def*: *s* = (*vphi_cur*, *vpsi_cur*, *epsi_cur*, *c_cur*, *zo_cur*)
by (*cases s*) *auto*
define *j_cur* **where** *j_cur* = *i* + *c_cur*
obtain *epsi'_cur t'_cur vphi'_cur tphi_cur bphi_cur vpsi'_cur tpsi_cur bpsi_cur* **where**
run_epsi: *ru_t epsi_cur* = *Some* (*epsi'_cur*, *t'_cur*)
and *run_vphi*: *ru n vphi_cur* = *Some* (*vphi'_cur*, (*tphi_cur*, *bphi_cur*))
and *run_vpsi*: *ru n vpsi_cur* = *Some* (*vpsi'_cur*, (*tpsi_cur*, *bpsi_cur*))
using *loop_assms*(*2*,*3*)
apply (*auto simp*: *while_until_cond_def while_until_body_def s_def split*: *option.splits dest*: *read_t_run*[**where** *?run_hd*=*run_hd*])
done
have *wf*: *wf_vydra phi j_cur n vphi_cur wf_vydra psi j_cur n vpsi_cur*
and *zo_cur_def*: *zo_cur* = (*case j_cur of* 0 ⇒ *None* | *Suc k* ⇒ *Some* (*τ σ k*, *sat phi k*, *sat psi k*))
using *loop_assms*(*1*)
by (*auto simp*: *loop_inv_def s_def j_cur_def*[*symmetric*])
have *wf'*: *wf_vydra phi* (*Suc j_cur*) *n vphi'_cur tphi_cur* = *τ σ j_cur bphi_cur* = *sat phi j_cur*
wf_vydra psi (*Suc j_cur*) *n vpsi'_cur tpsi_cur* = *τ σ j_cur bpsi_cur* = *sat psi j_cur*
using *Until*(*3*)[*OF wf*(*1*) *run_vphi*] *Until*(*4*)[*OF wf*(*2*) *run_vpsi*] *Until*(*7*,*8*)
apply (*auto*)
done
have *s'_def*: *s'* = (*vphi'_cur*, *vpsi'_cur*, *epsi'_cur*, *Suc c_cur*, *Some* (*t'_cur*, (*bphi_cur*, *bpsi_cur*)))
using *loop_assms*(*3*)
by (*auto simp*: *while_until_body_def s_def run_epsi run_vphi run_vpsi*)
obtain *gs_cur* **where** *gs_cur_def*: *reaches_on ru_t l_t0 gs_cur epsi_cur*
length gs_cur = *j_cur* ⋀*t*. *t* ∈ *set gs_cur* ⟹ *memR* (*τ σ i*) *t I*
using *loop_assms*(*1*)
by (*auto simp*: *loop_inv_def s_def j_cur_def*[*symmetric*])
have *t'_cur_def*: *t'_cur* = *τ σ j_cur*
using *ru_t_tau*[*OF gs_cur_def*(*1*) *run_epsi*]
by (*auto simp*: *gs_cur_def*(*2*))
have *t'_cur_right_I*: *memR t t'_cur I*
using *loop_assms*(*2*) *run_t_read*[*OF run_epsi*]
by (*auto simp*: *while_until_cond_def s_def*)
have *c_cur_zero*: *c_cur* = *0* ⟹ *j_cur* = *i*
by (*auto simp*: *j_cur_def*)
have *k* ∈ {*i*..<*Suc j_cur* − *1*} ⟹ *sat phi k* ∧ (*memL* (*τ σ i*) (*τ σ k*) *I* ⟶ ¬*sat psi k*) **for** *k*
using *loop_assms*(*1*,*2*)
by (*cases k* = *j_cur* − *Suc 0*) (*auto simp*: *while_until_cond_def loop_inv_def j_cur_def*[*symmetric*] *zo_cur_def s_def until_ready_def t_def split*: *nat.splits dest*: *c_cur_zero*)
then show *?thesis*
using *wf' t'_cur_right_I*
using *reaches_on_app*[*OF gs_cur_def*(*1*) *run_epsi*] *gs_cur_def*(*2*,*3*)
by (*auto simp*: *loop_inv_def s'_def j_cur_def*[*symmetric*] *t_def t'_cur_def intro*!: *exI*[*of* _ *gs_cur*

```
@ [t'_cur]])
  qed
  have wf_loop: wf {(s', s). loop_inv s ∧ while_until_cond I t s ∧ Some s' = while_until_body run_hd
(ru n) s}
  proof −
    obtain m where m_def: ¬τ σ m ≤ τ σ i + right I
      using ex_lt_τ[where ?x=right I and ?s=σ] Until(7)
      by auto
    define X where X = {(s', s). loop_inv s ∧ while_until_cond I t s ∧ Some s' = while_until_body
run_hd (ru n) s}
    have memR t (τ σ (i + c)) I ⟹ i + c < m for c
      using m_def order_trans[OF τ_mono[where ?i=m and ?j=i + c and ?s=σ]]
      by (fastforce simp: t_def dest!: memR_dest)
    then have X ⊆ measure (λ(vphi, vpsi, epsi, c, zo). m − c)
      by (fastforce simp: X_def while_until_cond_def while_until_body_def loop_inv_def Let_def split:
option.splits
          dest!: read_t_run[where ?run_hd=run_hd] dest: ru_t_tau)
    then show ?thesis
      using wf_subset
      by (auto simp: X_def[symmetric])
  qed
  have loop: loop_inv (vphi', vpsi', epsi', c', zo') ¬while_until_cond I t (vphi', vpsi', epsi', c', zo')
    using while_break_sound[where ?Q=λs. loop_inv s ∧ ¬while_until_cond I t s, OF _ _ wf_loop
loop_inv_init] loop_step
    by (auto simp: loop_def)
  have tau_right_I: k < j' ⟹ memR (τ σ i) (τ σ k) I for k
    using loop(1) ru_t_tau_in
    by (auto simp: loop_inv_def j'_def[symmetric] in_set_conv_nth)
  have read_t_epsi': read_t epsi' = Some et ⟹ et = τ σ j' for et
    using loop(1) ru_t_tau
    by (fastforce simp: loop_inv_def Let_def j'_def dest!: read_t_run[where ?run_hd=run_hd])
  have end_cond: until_ready I t c' zo' ∨ ¬(memR (τ σ i) (τ σ j') I)
    unfolding t_def[symmetric]
    using Until(6) c'_pos loop(2)[unfolded while_until_cond_def]
    by (auto simp: until_ready_def v_def(1) run_back loop_init_def[symmetric] loop_def zo'_Some
split: if_splits option.splits nat.splits dest: read_t_epsi')
  have Suc_i_le_j': Suc i ≤ j' and c'_j': c' − Suc 0 = j' − Suc i
    using end_cond c'_pos
    by (auto simp: until_ready_def j'_def split: nat.splits)
  have zo'_def: zo' = (case j' of 0 ⇒ None | Suc k ⇒ Some (τ σ k, sat phi k, sat psi k))
    and sat_phi: k ∈ {i..<j' − 1} ⟹ sat phi k
    and not_sat_psi: k ∈ {i..<j' − 1} ⟹ memL (τ σ i) (τ σ k) I ⟹ ¬sat psi k for k
    using loop(1)
    by (auto simp: loop_inv_def j'_def[symmetric])
  have b_sat: b = sat (Until phi I psi) i
  proof (rule iffI)
    assume b: b
    have mem: mem (τ σ i) (τ σ (j' − Suc 0)) I
      using b zo'_def tau_right_I[where ?k=j' − 1]
      by (auto simp: mem_def b_def t_def zo'_Some split: nat.splits)
    have sat_psi: sat psi (j' − 1)
      using b zo'_def
      by (auto simp: b_def zo'_Some split: nat.splits)
    show sat (Until phi I psi) i
      using Suc_i_le_j' mem sat_psi sat_phi
      by (auto intro!: exI[of _ j' − 1])
  next
    assume sat (Until phi I psi) i
```

```
    then obtain k where k_def: i ≤ k mem (τ σ i) (τ σ k) I sat psi k ∀ m ∈ {i..<k}. sat phi m
      by auto
    define X where X = {m ∈ {i..k}. memL (τ σ i) (τ σ m) I ∧ sat psi m}
    have fin_X: finite X and X_nonempty: X ≠ {} and k_X: k ∈ X
      using k_def
      by (auto simp: X_def mem_def)
    define km where km = Min X
    note aux = Min_in[OF fin_X X_nonempty, folded km_def]
    have km_def: i ≤ km km ≤ k memL (τ σ i) (τ σ km) I sat psi km ∀ m ∈ {i..<km}. sat phi m
      ∀ m ∈ {i..<km}. memL (τ σ i) (τ σ m) I ⟶ ¬sat psi m
      using aux Min_le[OF fin_X, folded km_def] k_def(4)
      by (fastforce simp: X_def)+
    have j'_le_km: j' − 1 ≤ km
      using not_sat_psi[OF _ km_def(3)] km_def(1,4)
      by fastforce
    show b
    proof (cases j' − 1 < km)
      case True
      have until_ready I t c' zo'
       using end_cond True km_def(2) k_def(2) memR_mono'[OF _ τ_mono[where ?i=j' and ?j=k
and ?s=σ]]
        by (auto simp: mem_def)
      then show ?thesis
        using c'_pos zo'_def km_def(5) Suc_i_le_j' True
        by (auto simp: until_ready_def zo'_Some b_def split: nat.splits)
    next
      case False
      have km_j': km = j' − 1
        using j'_le_km False
        by auto
      show ?thesis
        using c'_pos zo'_def km_def(3,4)
        by (auto simp: zo'_Some b_def km_j' t_def split: nat.splits)
    qed
  qed
  obtain gs where gs_def: reaches_on ru_t l_t0 gs epsi' length gs = j'
    ⋀t. t ∈ set gs ⟹ memR (τ σ i) t I
    using loop(1)
    by (auto simp: loop_inv_def j'_def[symmetric])
  note aux = τ_mono[where ?s=σ and ?i=i and ?j=Suc i]
  have wf_v': wf_vydra (Until phi I psi) (Suc i) (Suc n) v'
    unfolding v'_def
    apply (rule wf_vydra.intros(10)[where ?j=j' and ?es'=gs])
    using loop(1) reaches_on_app[OF v_def(5) run_back] Suc_i_le_j' c'_j' memL_mono[OF _ aux]
memR_mono[OF _ aux] gs_def
    by (auto simp: loop_inv_def j'_def[symmetric] v_def(6,8))
  show ?case
    using wf_v' ru_t_event[OF v_def(5) refl run_back]
    by (auto simp: b_sat v_def(6))
next
  case (MatchP n I r)
  have IH: x ∈ set (collect_subfmlas r []) ⟹ wf_vydra x j n v ⟹ ru n v = Some (v', t, b) ⟹ wf_vydra
x (Suc j) n v' ∧ t = τ σ j ∧ b = sat x j for x j v v' t b
    using MatchP(2,1,5,6) le_trans[OF collect_subfmlas_msize]
    using bf_collect_subfmlas[where ?r=r and ?phis=[]]
    by (fastforce simp: collect_subfmlas_atms[where ?phis=[], simplified, symmetric])
  have reaches_on (ru n) (su n phi) vs v ⟹ wf_vydra phi (length vs) n v if phi: phi ∈ set (collect_subfmlas
r []) for phi vs v
```

```
    apply (induction vs arbitrary: v rule: rev_induct)
    using MatchP(1) wf_vydra_sub collect_subfmlas_msize[OF phi]
     apply (auto elim!: reaches_on.cases)[1]
    using IH[OF phi]
    apply (fastforce dest!: reaches_on_split_last)
    done
  then have wf: reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) bs s ⟹ j < length
(collect_subfmlas r []) ⟹
    wf_vydra (collect_subfmlas r [] ! j) (length bs) n (s ! j) for bs s j
    using reach_run_subs_run
    by (fastforce simp: in_set_conv_nth)
  let ?qf = state_cnt r
  let ?transs = iarray_of_list (build_nfa_impl r (0, ?qf, []))
  define args where args = init_args ({0}, NFA.delta' ?transs ?qf, NFA.accept' ?transs ?qf) (ru_t,
read_t) (run_subs (ru n))
  interpret MDL_window σ r l_t0 map (su n) (collect_subfmlas r []) args
   using bs_sat[where ?r=r and ?n=n, OF _ wf _ reach_run_subs_len[where ?n=n]] IH run_t_read[of
run_hd]
      read_t_run[of _ _ run_hd] ru_t_tau MatchP(5) collect_subfmlas_atms[where ?phis=[]]
    unfolding args_def iarray_of_list_def
    by unfold_locales auto
  obtain w xs where w_def: v = VYDRA_MatchP I ?transs ?qf w
    valid_window_matchP args I l_t0 (map (su n) (collect_subfmlas r [])) xs i w
    using MatchP(3,4)
    by (auto simp: args_def elim!: wf_vydra.cases[of _ _ _ v])
  obtain tj' t' sj' bs where somes: w_run_t args (w_tj w) = Some (tj', t')
   w_run_sub args (w_sj w) = Some (sj', bs)
   using MatchP(4)
   by (auto simp: w_def(1) adv_end_def args_def Let_def split: option.splits prod.splits)
  obtain w' where w'_def: eval_mP I w = Some ((τ σ i, sat (MatchP I r) i), w')
    t' = τ σ i valid_matchP I l_t0 (map (su n) (collect_subfmlas r [])) (xs @ [(t', bs)]) (Suc i) w'
    using valid_eval_matchP[OF w_def(2) somes] MatchP(6)
    by auto
  have v'_def: v' = VYDRA_MatchP I ?transs ?qf w' (t, b) = (τ σ i, sat (MatchP I r) i)
    using MatchP(4)
   by (auto simp: w_def(1) args_def[symmetric] w'_def(1) simp del: eval_matchP.simps init_args.simps)
  have len_xs: length xs = i
    using w'_def(3)
    by (auto simp: valid_window_matchP_def)
  have ∃ es e. reaches_on run_hd init_hd es e ∧ length es = Suc i
    using ru_t_event valid_window_matchP_reach_tj[OF w_def(2)] somes(1) len_xs
    by (fastforce simp: args_def)
  then show ?case
    using MatchP(1) v'_def(2)  w'_def(3)
    by (auto simp: v'_def(1) args_def[symmetric] simp del: init_args.simps intro!: wf_vydra.intros(11))
next
  case (MatchF n I r)
  have IH: x ∈ set (collect_subfmlas r []) ⟹ wf_vydra x j n v ⟹ ru n v = Some (v', t, b) ⟹ wf_vydra
x (Suc j) n v' ∧ t = τ σ j ∧ b = sat x j for x j v v' t b
    using MatchF(2,1,5,6) le_trans[OF collect_subfmlas_msize]
    using bf_collect_subfmlas[where ?r=r and ?phis=[]]
    by (fastforce simp: collect_subfmlas_atms[where ?phis=[], simplified, symmetric])
  have reaches_on (ru n) (su n phi) vs v ⟹ wf_vydra phi (length vs) n v if phi: phi ∈ set (collect_subfmlas
r []) for phi vs v
    apply (induction vs arbitrary: v rule: rev_induct)
    using MatchF(1) wf_vydra_sub collect_subfmlas_msize[OF phi]
     apply (auto elim!: reaches_on.cases)[1]
    using IH[OF phi]
```

```
    apply (fastforce dest!: reaches_on_split_last)
    done
  then have wf: reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) bs s ⟹ j < length
(collect_subfmlas r []) ⟹
    wf_vydra (collect_subfmlas r [] ! j) (length bs) n (s ! j) for bs s j
    using reach_run_subs_run
    by (fastforce simp: in_set_conv_nth)
  let ?qf = state_cnt r
  let ?transs = iarray_of_list (build_nfa_impl r (0, ?qf, []))
  define args where args = init_args ({0}, NFA.delta' ?transs ?qf, NFA.accept' ?transs ?qf) (ru_t,
read_t) (run_subs (ru n))
  interpret MDL_window σ r l_t0 map (su n) (collect_subfmlas r []) args
   using bs_sat[where ?r=r and ?n=n, OF _ wf_ reach_run_subs_len[where ?n=n]] IH run_t_read[of
run_hd]
      read_t_run[of _ _ run_hd] ru_t_tau MatchF(5) collect_subfmlas_atms[where ?phis=[]]
    unfolding args_def iarray_of_list_def
    by unfold_locales auto
  obtain w xs where w_def: v = VYDRA_MatchF I ?transs ?qf w
    valid_window_matchF args I l_t0 (map (su n) (collect_subfmlas r [])) xs i w
    using MatchF(3,4)
    by (auto simp: args_def elim!: wf_vydra.cases[of _ _ _ v])
  obtain w' rho' where w'_def: eval_mF I w = Some ((t, b), w') valid_matchF I l_t0 (map (su n)
(collect_subfmlas r [])) rho' (Suc i) w' t = τ σ i b = sat (MatchF I r) i
    using valid_eval_matchF_sound[OF w_def(2)] MatchF(4,5,6)
   by (fastforce simp: w_def(1) args_def[symmetric] simp del: eval_matchF.simps init_args.simps split:
option.splits)
  have v'_def: v' = VYDRA_MatchF I ?transs ?qf w'
    using MatchF(4)
   by (auto simp: w_def(1) args_def[symmetric] w'_def(1) simp del: eval_matchF.simps init_args.simps)
  obtain w_ti' ti where ru_w_ti: ru_t (w_ti w) = Some (w_ti', ti)
    using MatchF(4) read_t_run
    by (auto simp: w_def(1) args_def split: option.splits)
  have ∃ es e. reaches_on run_hd init_hd es e ∧ length es = Suc i
    using w_def(2) ru_t_event[OF _ refl ru_w_ti, where ?ts=take (w_i w) (map fst xs)]
    by (auto simp: valid_window_matchF_def args_def)
  then show ?case
    using MatchF(1) w'_def(2)
   by (auto simp: v'_def(1) args_def[symmetric] w'_def(3,4) simp del: init_args.simps intro!: wf_vydra.intros(12))
qed

lemma reaches_ons_run_lD: reaches_on (run_subs (ru n)) vs ws vs' ⟹
  length vs = length vs'
  by (induction vs ws vs' rule: reaches_on_rev_induct)
     (auto simp: run_subs_def Let_def split: option.splits if_splits)

lemma reaches_ons_run_vD: reaches_on (run_subs (ru n)) vs ws vs' ⟹
  i < length vs ⟹ (∃ ys. reaches_on (ru n) (vs ! i) ys (vs' ! i) ∧ length ys = length ws)
proof (induction vs ws vs' rule: reaches_on_rev_induct)
  case (2 s s' bs bss s'')
  obtain ys where ys_def: reaches_on (ru n) (s ! i) ys (s' ! i)
    length s = length s' length ys = length bss
    using reaches_ons_run_lD[OF 2(1)] 2(3)[OF 2(4)]
    by auto
  obtain tb where tb_def: ru n (s' ! i) = Some (s'' ! i, tb)
    using run_subs_vD[OF 2(2) 2(4)[unfolded ys_def(2)]]
    by auto
  show ?case
    using reaches_on_app[OF ys_def(1) tb_def(1)] ys_def(2,3) tb_def
```

```
    by auto
qed (auto intro: reaches_on.intros)

lemma reaches_ons_runI:
  assumes ⋀phi. phi ∈ set (collect_subfmlas r []) ⟹ ∃ws v. reaches_on (ru n) (su n phi) ws v ∧ length
ws = i
  shows ∃ws v. reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) ws v ∧ length ws = i
  using assms
proof (induction i)
  case 0
  show ?case
    by (fastforce intro: reaches_on.intros)
next
  case (Suc i)
  have IH': ⋀phi. phi ∈ set (collect_subfmlas r []) ⟹ ∃ws v. reaches_on (ru n) (su n phi) ws v ∧
length ws = i ∧ ru n v ≠ None
  proof −
    fix phi
    assume lassm: phi ∈ set (collect_subfmlas r [])
    obtain ws v where ws_def: reaches_on (ru n) (su n phi) ws v
      length ws = Suc i
      using Suc(2)[OF lassm]
      by auto
    obtain y ys where ws_snoc: ws = ys @ [y]
      using ws_def(2)
      by (cases ws rule: rev_cases) auto
    show ∃ws v. reaches_on (ru n) (su n phi) ws v ∧ length ws = i ∧ ru n v ≠ None
      using reaches_on_split_last[OF ws_def(1)[unfolded ws_snoc]] ws_def(2) ws_snoc
      by fastforce
  qed
  obtain ws v where ws_def: reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) ws v
length ws = i
    using Suc(1) IH'
    by blast
  have x ∈ set v ⟹ Option.is_none (ru n x) ⟹ False for x
    using ws_def IH'
   by (auto simp: in_set_conv_nth) (metis is_none_code(2) reach_run_subs_len reach_run_subs_run
reaches_on_inj)
  then obtain v' tb where v'_def: run_subs (ru n) v = Some (v', tb)
    by (fastforce simp: run_subs_def Let_def)
  show ?case
    using reaches_on_app[OF ws_def(1) v'_def] ws_def(2)
    by fastforce
qed

lemma reaches_on_takeWhile: reaches_on r s vs s' ⟹ r s' = Some (s'', v) ⟹ ¬f v ⟹
  vs' = takeWhile f vs ⟹
  ∃t' t'' v'. reaches_on r s vs' t' ∧ r t' = Some (t'', v') ∧ ¬f v' ∧
  reaches_on r t' (drop (length vs') vs) s'
  by (induction s vs s' arbitrary: vs' rule: reaches_on.induct) (auto intro: reaches_on.intros)

lemma reaches_on_suffix:
  assumes reaches_on r s vs s' reaches_on r s vs' s'' length vs' ≤ length vs
  shows ∃vs''. reaches_on r s'' vs'' s' ∧ vs = vs' @ vs''
  using reaches_on_split'[where ?i=length vs', OF assms(1,3)] assms(3) reaches_on_inj[OF assms(2)]
  by (metis add_diff_cancel_right' append_take_drop_id diff_diff_cancel length_append length_drop)

lemma vydra_wf_reaches_on:
```

assumes $\bigwedge j\ v.\ j < i \Longrightarrow wf\_vydra\ \varphi\ j\ n\ v \Longrightarrow ru\ n\ v = None \Longrightarrow False\ bounded\_future\_fmla\ \varphi\ wf\_fmla\ \varphi\ msize\_fmla\ \varphi \leq n$
shows $\exists vs\ v.\ reaches\_on\ (ru\ n)\ (su\ n\ \varphi)\ vs\ v \wedge wf\_vydra\ \varphi\ i\ n\ v \wedge length\ vs = i$
using *assms(1)*
proof (*induction i*)
case (*Suc i*)
obtain *vs v* **where** *IH*: *reaches_on* (*ru n*) (*su n* φ) *vs v wf_vydra* φ *i n v length vs* = *i*
using *Suc(1) Suc(2)*[*OF less_SucI*]
by *auto*
show *?case*
using *reaches_on_app*[*OF IH(1)*] *Suc(2)*[*OF _ IH(2)*] *vydra_sound_aux*[*OF assms(4) IH(2) _ assms(2,3)*] *IH(3)*
by *fastforce*
qed (*auto intro*: *reaches_on.intros wf_vydra_sub*[*OF assms(4)*])

lemma *reaches_on_Some*:
assumes *reaches_on r s vs s' reaches_on r s vs' s'' length vs* < *length vs'*
shows $\exists s'''\ x.\ r\ s' = Some\ (s''', x)$
using *reaches_on_split*[*OF assms(2,3)*] *reaches_on_inj*[*OF assms(1)*] *assms(3)*
by *auto*

lemma *reaches_on_progress*:
assumes *reaches_on run_hd init_hd vs e*
shows *progress phi* (*map fst vs*) ≤ *length vs*
using *progress_le_ts*[*of map fst vs phi*] *reaches_on_run_hd* τ*_fin*
by (*fastforce dest*!: *reaches_on_setD*[*OF assms*] *reaches_on_split_last*)

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* $\varphi \leq n$
shows *False*
using *msize run*
proof (*induction n* φ *arbitrary*: *i v rule*: *run_induct*)
case (*Bool b n*)
have *False* **if** *v_def*: *v* = *VYDRA_Bool b g* **for** *g*
proof −
obtain *es* **where** *g_def*: *reaches_on run_hd init_hd es g length es* = *i*
using *Bool(1)*
by (*auto simp*: *v_def elim*: *wf_vydra.cases*)
show *False*
using *Bool(2) reaches_on_Some*[*OF g_def(1) prefix*] *Bool(3)*
by (*auto simp*: *v_def g_def(2) split*: *option.splits*)
qed
then show *False*
using *Bool(1)*
by (*auto elim*!: *wf_vydra.cases*[*of _ _ _ v*])
next
case (*Atom a n*)
have *False* **if** *v_def*: *v* = *VYDRA_Atom a g* **for** *g*
proof −
obtain *es* **where** *g_def*: *reaches_on run_hd init_hd es g length es* = *i*
using *Atom(1)*
by (*auto simp*: *v_def elim*: *wf_vydra.cases*)
show *False*
using *Atom(2) reaches_on_Some*[*OF g_def(1) prefix*] *Atom(3)*
by (*auto simp*: *v_def g_def(2) split*: *option.splits*)

```
  qed
  then show False
    using Atom(1)
    by (auto elim!: wf_vydra.cases[of _ _ _ v])
next
  case (Neg n phi)
  show ?case
    apply (rule wf_vydra.cases[OF Neg(3)])
    using Neg
    by (fastforce split: option.splits)+
next
  case (Bin n f phi psi)
  show ?case
    apply (rule wf_vydra.cases[OF Bin(4)])
    using Bin
    by (fastforce split: option.splits)+
next
  case (Prev n I phi)
  show ?case
  proof (cases i)
    case 0
    obtain vphi where v_def: v = VYDRA_Prev I vphi init_hd None
      using Prev(3)
      by (auto simp: 0 dest: reaches_on_NilD elim!: wf_vydra.cases[of Prev I phi])
    show ?thesis
      using Prev(4,5) prefix
      by (auto simp: 0 v_def elim: reaches_on.cases split: option.splits)
  next
    case (Suc j)
    show ?thesis
    proof (cases v = VYDRA_None)
      case v_def: True
      obtain w where w_def: wf_vydra phi j n w ru n w = None
        using Prev(3)
        by (auto simp: Suc v_def elim!: wf_vydra.cases[of Prev I phi])
      show ?thesis
        using Prev(2)[OF w_def(1,2)] Prev(5,6,7)
        by (auto simp: Suc)
    next
      case False
      obtain vphi tphi bphi es g where v_def: v = VYDRA_Prev I vphi g (Some (tphi, bphi))
        wf_vydra phi (Suc j) n vphi reaches_on run_hd init_hd es g length es = Suc j
        using Prev(3) False
        by (auto simp: Suc elim!: wf_vydra.cases[of Prev I phi])
      show ?thesis
        using Prev(4,5) reaches_on_Some[OF v_def(3) prefix]
        by (auto simp: Let_def Suc v_def(1,4) split: option.splits)
    qed
  qed
next
  case (Next n I phi)
  show ?case
  proof (cases v = VYDRA_None)
    case True
    obtain w where w_def: wf_vydra phi i n w ru n w = None
      using Next(3)
      by (auto simp: True elim: wf_vydra.cases)
    show ?thesis
```

```
    using Next(2)[OF w_def] Next(5,6,7)
    by (auto split: nat.splits)
  next
    case False
    obtain w sub to es where v_def: v = VYDRA_Next I w sub to wf_vydra phi (length es) n w
      reaches_on run_hd init_hd es sub length es = (case to of None ⇒ 0 | _ ⇒ Suc i)
      case to of None ⇒ i = 0 | Some told ⇒ told = τ σ i
      using Next(3) False
      by (auto elim!: wf_vydra.cases[of _ _ _ v] split: option.splits nat.splits)
    show ?thesis
    proof (cases to)
      case None
      obtain w' tw' bw' where w'_def: ru n w = Some (w', (tw', bw'))
        using Next(2)[OF v_def(2)] Next(5,6,7)
        by (fastforce simp: v_def(4) None split: nat.splits)
      have wf: wf_vydra phi (Suc (length es)) n w'
        using v_def(4,5) vydra_sound_aux[OF Next(1) v_def(2) w'_def] Next(6,7)
        by (auto simp: None)
      show ?thesis
        using Next(2)[OF wf] Next(4,5,6,7) reaches_on_Some[OF v_def(3) prefix]
          reaches_on_Some[OF reaches_on_app[OF v_def(3)] prefix] reaches_on_progress[OF prefix,
where ?phi=phi]
        by (cases vs) (fastforce simp: v_def(1,4) w'_def None split: option.splits nat.splits if_splits)+
    next
      case (Some z)
      show ?thesis
      using Next(2)[OF v_def(2)] Next(4,5,6,7) reaches_on_Some[OF v_def(3) prefix] reaches_on_progress[OF
prefix, where ?phi=phi]
        by (auto simp: v_def(1,4) Some split: option.splits nat.splits)
    qed
  qed
next
  case (Since n I phi psi)
  obtain vphi vpsi g cphi cpsi cppsi tppsi j gs where v_def:
    v = VYDRA_Since I vphi vpsi g cphi cpsi cppsi tppsi
    wf_vydra phi i n vphi wf_vydra psi j n vpsi j ≤ i
    reaches_on ru_t l_t0 gs g length gs = j cpsi = i − j
    using Since(5)
    by (auto elim: wf_vydra.cases)
  obtain vphi' t1 b1 where run_vphi: ru n vphi = Some (vphi', t1, b1)
    using Since(3)[OF v_def(2)] Since(7,8,9)
    by fastforce
  obtain cs c where wf_vphi': wf_vydra phi (Suc i) n vphi'
    and reaches_Suc_i: reaches_on run_hd init_hd cs c length cs = Suc i
    and t1_def: t1 = τ σ i
    using vydra_sound_aux[OF Since(1) v_def(2) run_vphi] Since(8,9)
    by auto
  note ru_t_Some = ru_t_Some[OF reaches_Suc_i]
  define loop_inv where loop_inv = (λ(vpsi, e, cpsi :: nat, cppsi :: nat option, tppsi :: 't option).
    let j = Suc i − cpsi in cpsi ≤ Suc i ∧ wf_vydra psi j n vpsi ∧ (∃ es. reaches_on ru_t l_t0 es e ∧
length es = j))
  define loop_init where loop_init = (vpsi, g, Suc cpsi, map_option Suc cppsi, tppsi)
  have j_def: j = i − cpsi and cpsi_i: cpsi ≤ i
    using v_def(4,7)
    by auto
  then have loop_inv_init: loop_inv loop_init
    using v_def(3,5,6,7) last_before_Some
    by (fastforce simp: loop_inv_def loop_init_def Let_def j_def split: option.splits)
```

```
  have wf_loop: wf {(s', s). loop_inv s ∧ while_since_cond I t1 s ∧ Some s' = while_since_body run_hd
(ru n) s}
    by (auto intro: wf_subset[OF wf_since])
  have step_loop: pred_option' loop_inv (while_since_body run_hd (ru n) s)
    if loop_assms: loop_inv s while_since_cond I t1 s for s
  proof −
    obtain vpsi d cpsi cppsi tppsi where s_def: s = (vpsi, d, cpsi, cppsi, tppsi)
      by (cases s) auto
    have cpsi_pos: cpsi > 0
      using loop_assms(2)
      by (auto simp: while_since_cond_def s_def)
    define j where j = Suc i − cpsi
    have j_i: j ≤ i
      using cpsi_pos
      by (auto simp: j_def)
    obtain ds where loop_before: cpsi ≤ Suc i wf_vydra psi j n vpsi reaches_on ru_t l_t0 ds d length
ds = j
      using loop_assms(1)
      by (auto simp: s_def j_def loop_inv_def Let_def)
    obtain h tt where tt_def: read_t d = Some tt d = Some (h, tt)
      using ru_t_Some[OF loop_before(3)] loop_before(4) loop_assms(2)
      by (cases d) (fastforce simp: while_since_cond_def s_def j_def split: option.splits)+
    obtain d' where d'_def: reaches_on ru_t l_t0 (ds @ [tt]) d' ru_t d = Some (d', tt)
      using reaches_on_app[OF loop_before(3)] tt_def(1)
      by (cases run_hd h) (auto simp: tt_def(2))
    obtain vpsi' tpsi' bpsi' where run_vpsi: ru n vpsi = Some (vpsi', (tpsi', bpsi'))
      using Since(4) j_i Since(7,8,9) loop_before(2)
      by fastforce
    have wf_psi': wf_vydra psi (Suc j) n vpsi' and t'_def: tpsi' = τ σ j and b'_def: bpsi' = sat psi j
      using vydra_sound_aux[OF Since(2) loop_before(2) run_vpsi] Since(8,9)
      by auto
    define j' where j'_def: j' = Suc i − (cpsi − Suc 0)
    have j'_j: j' = Suc j
      using loop_before(1) cpsi_pos
      by (auto simp: j'_def j_def)
    show ?thesis
      using wf_psi'(1) loop_before(1,4) d'_def(1)
    by (fastforce simp: while_since_body_def s_def run_vpsi pred_option'_def loop_inv_def j'_def[symmetric]
j'_j d'_def(2) t1_def)
  qed
  show ?case
    using while_break_complete[OF step_loop _ wf_loop loop_inv_init, where ?Q=λ_. True] Since(6)
    by (auto simp: pred_option'_def v_def(1) run_vphi Let_def loop_init_def split: option.splits)
next
  case (Until n I phi psi)
  obtain back vphi vpsi front c z es es' j where v_def:
    v = VYDRA_Until I back vphi vpsi front c z
    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)
    using Until(5)
    by (auto simp: elim: wf_vydra.cases)
  have ru_t_Some: reaches_on ru_t l_t0 gs g ⟹ length gs < length vs ⟹ ∃ g' gt. ru_t g = Some (g',
gt) for gs g
    using reaches_on_Some reaches_on_run_hd_t[OF prefix]
    by fastforce
```

```
  have vs_tau: map fst vs ! k = τ σ k if k_vs: k < length vs for k
    using reaches_on_split[OF prefix k_vs] run_hd_sound k_vs
    apply (cases vs ! k)
    apply (auto)
    apply (metis fst_conv length_map nth_map prefix reaches_on_run_hd_t ru_t_tau_in)
    done
  define m where m = min (length (map fst vs) − 1) (min (progress phi (map fst vs)) (progress psi
(map fst vs)))
  have m_vs: m < length vs
    using Until(7)
    by (cases vs) (auto simp: m_def split: if_splits)
  define A where A = {j. 0 ≤ j ∧ j ≤ m ∧ memR (map fst vs ! j) (map fst vs ! m) I}
  have m_A: m ∈ A
    using memR_tfin_refl[OF τ_fin] vs_tau[OF m_vs]
    by (fastforce simp: A_def)
  then have A_nonempty: A ≠ {}
    by auto
  have A_finite: finite A
    by (auto simp: A_def)
  have p: progress (Until phi I psi) (map fst vs) = Min A
    using Until(7)
    unfolding progress.simps m_def[symmetric] Let_def A_def[symmetric]
    by auto
  have i_A: i ∉ A
    using Until(7) A_finite A_nonempty
    by (auto simp del: progress.simps simp: p)
  have i_m: i < m
    using Until(7) m_A A_finite A_nonempty
    by (auto simp del: progress.simps simp: p)
  have memR_i_m: ¬memR (map fst vs ! i) (map fst vs ! m) I
    using i_A i_m
    by (auto simp: A_def)
  have i_vs: i < length vs
    using i_m m_vs
    by auto
  have j_m: j ≤ m
    using ru_t_tau_in[OF v_def(7), of m] v_def(9)[of τ σ m] memR_i_m
    unfolding vs_tau[OF i_vs] vs_tau[OF m_vs]
    by (force simp: in_set_conv_nth v_def(8))
  have j_vs: j < length vs
    using j_m m_vs
    by auto
  obtain back' t where run_back: ru_t back = Some (back', t) and t_def: t = τ σ i
    using ru_t_Some[OF v_def(5)] v_def(4) j_vs ru_t_tau[OF v_def(5)]
    by (fastforce simp: v_def(6))
  define loop_inv where loop_inv = (λ(vphi, vpsi, front, c, z :: ('t × bool × bool) option).
    let j = i + c in wf_vydra phi j n vphi ∧ wf_vydra psi j n vpsi ∧ j < length vs ∧
    (∃ es'. reaches_on ru_t l_t0 es' front ∧ length es' = j) ∧
    (z = None ⟶ j = 0))
  define loop_init where loop_init = (vphi, vpsi, front, c, z)
  have j_eq: j = i + c
    using v_def(4)
    by (auto simp: v_def(10))
  have j = 0 ⟹ c = 0
    by (auto simp: j_eq)
  then have loop_inv_init: loop_inv loop_init
    using v_def(2,3,4,7,8,9,11) j_vs
    by (auto simp: loop_inv_def loop_init_def j_eq[symmetric] split: nat.splits)
```

have *loop_step*: *pred_option′ loop_inv* (*while_until_body run_hd* (*ru n*) *s*) **if** *loop_assms*: *loop_inv s while_until_cond I t s* **for** *s*
proof −
obtain *vphi_cur vpsi_cur epsi_cur c_cur zo_cur* **where** *s_def*: *s* = (*vphi_cur*, *vpsi_cur*, *epsi_cur*, *c_cur*, *zo_cur*)
by (*cases s*) *auto*
define *j_cur* **where** *j_cur* = *i* + *c_cur*
obtain *gs* **where** *wf*: *wf_vydra phi j_cur n vphi_cur wf_vydra psi j_cur n vpsi_cur*
and *gs_def*: *reaches_on ru_t l_t0 gs epsi_cur length gs* = *j_cur*
and *j_cur_vs*: *j_cur* < *length vs*
using *loop_assms*(*1*)
by (*auto simp*: *loop_inv_def s_def j_cur_def*[*symmetric*])
obtain *epsi′_cur t′_cur* **where** *run_epsi*: *ru_t epsi_cur* = *Some* (*epsi′_cur*, *t′_cur*)
and *t′_cur_def*: *t′_cur* = *τ σ* (*length gs*)
using *ru_t_Some*[*OF gs_def*(*1*)] *ru_t_event*[*OF gs_def*(*1*) *refl*] *j_cur_vs*
by (*auto simp*: *gs_def*(*2*))
have *j_m*: *j_cur* < *m*
using *loop_assms*(*2*) *memR_i_m memR_mono′*[*OF* _ *τ_mono*, *of* _ _ _ _ *m*]
unfolding *vs_tau*[*OF i_vs*] *vs_tau*[*OF m_vs*]
by (*fastforce simp*: *gs_def*(*2*) *while_until_cond_def s_def run_t_read*[*OF run_epsi*] *t_def t′_cur_def*)
have *j_cur_prog_phi*: *j_cur* < *progress phi* (*map fst vs*)
using *j_m*
by (*auto simp*: *m_def*)
have *j_cur_prog_psi*: *j_cur* < *progress psi* (*map fst vs*)
using *j_m*
by (*auto simp*: *m_def*)
obtain *vphi′_cur tphi_cur bphi_cur* **where** *run_vphi*: *ru n vphi_cur* = *Some* (*vphi′_cur*, (*tphi_cur*, *bphi_cur*))
using *Until*(*3*)[*OF wf*(*1*) _ *j_cur_prog_phi*] *Until*(*8*,*9*)
by *fastforce*
obtain *vpsi′_cur tpsi_cur bpsi_cur* **where** *run_vpsi*: *ru n vpsi_cur* = *Some* (*vpsi′_cur*, (*tpsi_cur*, *bpsi_cur*))
using *Until*(*4*)[*OF wf*(*2*) _ *j_cur_prog_psi*] *Until*(*8*,*9*)
by *fastforce*
have *wf′*: *wf_vydra phi* (*Suc j_cur*) *n vphi′_cur wf_vydra psi* (*Suc j_cur*) *n vpsi′_cur*
using *vydra_sound_aux*[*OF Until*(*1*) *wf*(*1*) *run_vphi*] *vydra_sound_aux*[*OF Until*(*2*) *wf*(*2*) *run_vpsi*] *Until*(*8*,*9*)
by *auto*
show *?thesis*
using *wf′ reaches_on_app*[*OF gs_def*(*1*) *run_epsi*] *gs_def*(*2*) *j_m m_vs*
by (*auto simp*: *pred_option′_def while_until_body_def s_def run_epsi run_vphi run_vpsi loop_inv_def j_cur_def*[*symmetric*])
qed
have *wf_loop*: *wf* {(*s′*, *s*). *loop_inv s* ∧ *while_until_cond I t s* ∧ *Some s′* = *while_until_body run_hd* (*ru n*) *s*}
proof −
obtain *m* **where** *m_def*: ¬*τ σ m* ≤ *τ σ i* + *right I*
using *ex_lt_τ*[**where** *?x*=*right I* **and** *?s*=*σ*] *Until*(*8*)
by *auto*
define *X* **where** *X* = {(*s′*, *s*). *loop_inv s* ∧ *while_until_cond I t s* ∧ *Some s′* = *while_until_body run_hd* (*ru n*) *s*}
have *memR t* (*τ σ* (*i* + *c*)) *I* ⟹ *i* + *c* < *m* **for** *c*
using *m_def order_trans*[*OF τ_mono*[**where** *?i*=*m* **and** *?j*=*i* + *c* **and** *?s*=*σ*]]
by (*fastforce simp*: *t_def dest*!: *memR_dest*)
then have *X* ⊆ *measure* (λ(*vphi*, *vpsi*, *epsi*, *c*, *zo*). *m* − *c*)
by (*fastforce simp*: *X_def while_until_cond_def while_until_body_def loop_inv_def Let_def split*: *option.splits*
dest!: *read_t_run*[**where** *?run_hd*=*run_hd*] *dest*: *ru_t_tau*)

```
    then show ?thesis
      using wf_subset
      by (auto simp: X_def[symmetric])
  qed
  obtain vphi' vpsi' front' c' z' where loop:
    while_break (while_until_cond I t) (while_until_body run_hd (ru n)) loop_init = Some (vphi', vpsi',
front', c', z')
    loop_inv (vphi', vpsi', front', c', z') ¬while_until_cond I t (vphi', vpsi', front', c', z')
    using while_break_complete[where ?P=loop_inv, OF loop_step _ wf_loop loop_inv_init]
     by (cases while_break (while_until_cond I t) (while_until_body run_hd (ru n)) loop_init) (force
simp: pred_option'_def)+
  define j' where j' = i + c'
  obtain tf where read_front': read_t front' = Some tf
    using loop(2)
   by (auto simp: loop_inv_def j'_def[symmetric] dest!: ru_t_Some run_t_read[where ?run_hd=run_hd])
  have tf_fin: tf ∈ tfin
   using loop(2) ru_t_Some[where ?g=front'] ru_t_tau[where ?t'=front'] read_t_run[OF read_front']
τ_fin[where ?σ=σ]
    by (fastforce simp: loop_inv_def j'_def[symmetric])
  have c'_pos: c' = 0 ⟹ False
    using loop(2,3) ru_t_tau ru_t_tau[OF reaches_on.intros(1)] read_t_run[OF read_front']
      memR_tfin_refl[OF tf_fin]
     by (fastforce simp: loop_inv_def while_until_cond_def until_ready_def read_front' t_def dest!:
reaches_on_NilD)
  have z'_Some: z' = None ⟹ False
    using loop(2) c'_pos
    by (auto simp: loop_inv_def j'_def[symmetric])
  show ?case
    using Until(6) c'_pos z'_Some
    by (auto simp: v_def(1) run_back loop_init_def[symmetric] loop(1) read_front' split: if_splits op-
tion.splits)
next
  case (MatchP n I r)
  have msize_sub: ⋀x. x ∈ set (collect_subfmlas r []) ⟹ msize_fmla x ≤ n
    using le_trans[OF collect_subfmlas_msize] MatchP(1)
    by auto
  have sound: x ∈ set (collect_subfmlas r []) ⟹ wf_vydra x j n v ⟹ ru n v = Some (v', t, b) ⟹
wf_vydra x (Suc j) n v' ∧ t = τ σ j ∧ b = sat x j for x j v v' t b
    using MatchP vydra_sound_aux[OF msize_sub] le_trans[OF collect_subfmlas_msize]
    using bf_collect_subfmlas[where ?r=r and ?phis=[]]
    by (fastforce simp: collect_subfmlas_atms[where ?phis=[], simplified, symmetric])
 have reaches_on (ru n) (su n phi) vs v ⟹ wf_vydra phi (length vs) n v if phi: phi ∈ set (collect_subfmlas
r []) for phi vs v
    apply (induction vs arbitrary: v rule: rev_induct)
    using MatchP(1) wf_vydra_sub collect_subfmlas_msize[OF phi]
     apply (auto elim!: reaches_on.cases)[1]
    using sound[OF phi]
    apply (fastforce dest!: reaches_on_split_last)
    done
  then have wf: reaches_on (run_subs (ru n)) (map (su n) (collect_subfmlas r [])) bs s ⟹ j < length
(collect_subfmlas r []) ⟹
    wf_vydra (collect_subfmlas r [] ! j) (length bs) n (s ! j) for bs s j
    using reach_run_subs_run
    by (fastforce simp: in_set_conv_nth)
  let ?qf = state_cnt r
  let ?transs = iarray_of_list (build_nfa_impl r (0, ?qf, []))
  define args where args = init_args ({0}, NFA.delta' ?transs ?qf, NFA.accept' ?transs ?qf) (ru_t,
read_t) (run_subs (ru n))
```

interpret *MDL_window* σ *r l_t0 map* (*su n*) (*collect_subfmlas r* []) *args*
using *bs_sat*[**where** *?r=r* **and** *?n=n*, *OF _ wf _ reach_run_subs_len*[**where** *?n=n*]] *sound run_t_read*[*of run_hd*]
read_t_run[*of _ _ run_hd*] *ru_t_tau MatchP*(*5*) *collect_subfmlas_atms*[**where** *?phis*=[]]
unfolding *args_def iarray_of_list_def*
by *unfold_locales auto*
obtain *w xs* **where** *w_def*: *v* = *VYDRA_MatchP I ?transs ?qf w*
valid_window_matchP args I l_t0 (*map* (*su n*) (*collect_subfmlas r* [])) *xs i w*
using *MatchP*(*3*)
by (*auto simp*: *args_def elim*!: *wf_vydra.cases*)
note *args*′ = *args_def*[*unfolded init_args.simps*, *symmetric*]
have *run_args*: *w_run_t args* = *ru_t w_run_sub args* = *run_subs* (*ru n*)
by (*auto simp*: *args_def*)
have *len_xs*: *length xs* = *i*
using *w_def*(*2*)
by (*auto simp*: *valid_window_matchP_def*)
obtain *ej tj* **where** *w_tj*: *ru_t* (*w_tj w*) = *Some* (*ej*, *tj*)
using *reaches_on_Some*[*OF valid_window_matchP_reach_tj*[*OF w_def*(*2*)]] *reaches_on_run_hd_t*[*OF prefix*]
MatchP(*5*) *reaches_on_progress*[*OF prefix*, **where** *?phi=MatchP I r*]
by (*fastforce simp*: *run_args len_xs simp del*: *progress.simps*)
have *run_subs* (*ru n*) (*w_sj w*) = *None*
using *valid_eval_matchP*[*OF w_def*(*2*), *unfolded run_args*] *w_tj MatchP*(*4*,*7*)
by (*cases run_subs* (*ru n*) (*w_sj w*)) (*auto simp*: *w_def*(*1*) *args*′ *simp del*: *eval_matchP.simps split*: *option.splits*)
then obtain *j* **where** *j_def*: *j* < *length* (*w_sj w*) *ru n* (*w_sj w* ! *j*) = *None*
by (*auto simp*: *run_subs_def Let_def in_set_conv_nth Option.is_none_def split*: *if_splits*)
have *len_w_sj*: *length* (*w_sj w*) = *length* (*collect_subfmlas r* [])
using *valid_window_matchP_reach_sj*[*OF w_def*(*2*)] *reach_run_subs_len*
by (*auto simp*: *run_args*)
define *phi* **where** *phi* = *collect_subfmlas r* [] ! *j*
have *phi_in_set*: *phi* ∈ *set* (*collect_subfmlas r* [])
using *j_def*(*1*)
by (*auto simp*: *phi_def in_set_conv_nth len_w_sj*)
have *wf*: *wf_vydra phi i n* (*w_sj w* ! *j*)
using *valid_window_matchP_reach_sj*[*OF w_def*(*2*)] *wf*[*folded len_w_sj*, *OF _ j_def*(*1*)] *len_xs*
by (*fastforce simp*: *run_args phi_def*)
have *i* < *progress phi* (*map fst vs*)
using *MatchP*(*5*) *phi_in_set atms_nonempty*[*of r*] *atms_finite*[*of r*] *collect_subfmlas_atms*[*of r* []]
by *auto*
then show *?case*
using *MatchP*(*2*)[*OF msize_sub*[*OF phi_in_set*] *wf j_def*(*2*)] *MatchP*(*6*,*7*) *phi_in_set*
bf_collect_subfmlas[**where** *?r=r* **and** *?phis*=[]]
by (*auto simp*: *collect_subfmlas_atms*)
next
case (*MatchF n I r*)
have *subfmla*: *msize_fmla x* ≤ *n bounded_future_fmla x wf_fmla x* **if** *x* ∈ *set* (*collect_subfmlas r* []) **for** *x*
using *that MatchF*(*1*,*6*,*7*) *le_trans*[*OF collect_subfmlas_msize*] *bf_collect_subfmlas*[**where** *?r=r* **and** *?phis*=[] **and** *?phi=x*]
collect_subfmlas_atms[**where** *?phis*=[] **and** *?r=r*]
by *auto*
have *sound*: *x* ∈ *set* (*collect_subfmlas r* []) ⟹ *wf_vydra x j n v* ⟹ *ru n v* = *Some* (*v*′, *t*, *b*) ⟹ *wf_vydra x* (*Suc j*) *n v*′ ∧ *t* = *τ* *σ j* ∧ *b* = *sat x j* **for** *x j v v*′ *t b*
using *MatchF vydra_sound_aux subfmla*
by *fastforce*
have *reaches_on* (*ru n*) (*su n phi*) *vs v* ⟹ *wf_vydra phi* (*length vs*) *n v* **if** *phi*: *phi* ∈ *set* (*collect_subfmlas r* []) **for** *phi vs v*

apply (*induction vs arbitrary*: *v rule*: *rev_induct*)
using *MatchF*(*1*) *wf_vydra_sub subfmla*(*1*)[*OF phi*] *sound*[*OF phi*]
apply (*auto elim*!: *reaches_on.cases*)[*1*]
using *sound*[*OF phi*]
apply (*fastforce dest*!: *reaches_on_split_last*)
done
then have *wf*: *reaches_on* (*run_subs* (*ru n*)) (*map* (*su n*) (*collect_subfmlas r* [])) *bs s* ⟹ *j* < *length* (*collect_subfmlas r* []) ⟹
wf_vydra (*collect_subfmlas r* [] ! *j*) (*length bs*) *n* (*s* ! *j*) **for** *bs s j*
using *reach_run_subs_run*
by (*fastforce simp*: *in_set_conv_nth*)
let *?qf* = *state_cnt r*
let *?transs* = *iarray_of_list* (*build_nfa_impl r* (*0*, *?qf*, []))
define *args* **where** *args* = *init_args* ({*0*}, *NFA.delta*′ *?transs ?qf*, *NFA.accept*′ *?transs ?qf*) (*ru_t*, *read_t*) (*run_subs* (*ru n*))
interpret *MDL_window* σ *r l_t0 map* (*su n*) (*collect_subfmlas r* []) *args*
using *bs_sat*[**where** *?r*=*r* **and** *?n*=*n*, *OF _ wf _ reach_run_subs_len*[**where** *?n*=*n*]] *sound run_t_read*[*of run_hd*]
read_t_run[*of _ _ run_hd*] *ru_t_tau MatchF*(*5*) *subfmla*
unfolding *args_def iarray_of_list_def*
by *unfold_locales auto*
obtain *w xs* **where** *w_def*: *v* = *VYDRA_MatchF I ?transs ?qf w*
valid_window_matchF args I l_t0 (*map* (*su n*) (*collect_subfmlas r* [])) *xs i w*
using *MatchF*(*3*)
by (*auto simp*: *args_def elim*!: *wf_vydra.cases*)
note *args*′ = *args_def*[*unfolded init_args.simps*, *symmetric*]
have *run_args*: *w_run_t args* = *ru_t w_read_t args* = *read_t w_run_sub args* = *run_subs* (*ru n*)
by (*auto simp*: *args_def*)
have *vs_tau*: *map fst vs* ! *k* = τ σ *k* **if** *k_vs*: *k* < *length vs* **for** *k*
using *reaches_on_split*[*OF prefix k_vs*] *run_hd_sound k_vs*
apply (*cases vs* ! *k*)
apply (*auto*)
apply (*metis fst_conv length_map nth_map prefix reaches_on_run_hd_t ru_t_tau_in*)
done
define *m* **where** *m* = *min* (*length* (*map fst vs*) − *1*) (*Min* ((λ*f*. *progress f* (*map fst vs*)) ‘ *atms r*))
have *m_vs*: *m* < *length vs*
using *MatchF*(*5*)
by (*cases vs*) (*auto simp*: *m_def split*: *if_splits*)
define *A* **where** *A* = {*j*. *0* ≤ *j* ∧ *j* ≤ *m* ∧ *memR* (*map fst vs* ! *j*) (*map fst vs* ! *m*) *I*}
have *m_A*: *m* ∈ *A*
using *memR_tfin_refl*[*OF* τ*_fin*] *vs_tau*[*OF m_vs*]
by (*fastforce simp*: *A_def*)
then have *A_nonempty*: *A* ≠ {}
by *auto*
have *A_finite*: *finite A*
by (*auto simp*: *A_def*)
have *p*: *progress* (*MatchF I r*) (*map fst vs*) = *Min A*
using *MatchF*(*5*)
unfolding *progress.simps m_def*[*symmetric*] *Let_def A_def*[*symmetric*]
by *auto*
have *i_A*: *i* ∉ *A*
using *MatchF*(*5*) *A_finite A_nonempty*
by (*auto simp del*: *progress.simps simp*: *p*)
have *i_m*: *i* < *m*
using *MatchF*(*5*) *m_A A_finite A_nonempty*
by (*auto simp del*: *progress.simps simp*: *p*)
have *memR_i_m*: ¬*memR* (*map fst vs* ! *i*) (*map fst vs* ! *m*) *I*
using *i_A i_m*

by (*auto simp*: *A_def*)
have *i_vs*: $i < length\ vs$
using *i_m m_vs*
by *auto*
obtain *ti* **where** *read_ti*: $w\_read\_t\ args\ (w\_ti\ w) = Some\ ti$
using *w_def*(2) *reaches_on_Some*[**where** *?r*=*ru_t* **and** *?s*=*l_t0* **and** *?s'*=*w_ti w*]
reaches_on_run_hd_t[*OF prefix*] *i_vs run_t_read*[**where** *?t*=*w_ti w*]
by (*fastforce simp*: *valid_window_matchF_def run_args*)
have *ti_def*: $ti = \tau\ \sigma\ i$
using *w_def*(2) *ru_t_tau read_t_run*[*OF read_ti*]
by (*fastforce simp*: *valid_window_matchF_def run_args*)
note *reach_tj* = *valid_window_matchF_reach_tj*[*OF w_def*(2), *unfolded run_args*]
note *reach_sj* = *valid_window_matchF_reach_sj*[*OF w_def*(2), *unfolded run_args*]
have *len_xs*: $length\ xs = w\_j\ w$ **and** *memR_xs*: $\bigwedge l.\ l \in \{w\_i\ w..<w\_j\ w\} \Longrightarrow memR\ (ts\_at\ xs\ i)\ (ts\_at\ xs\ l)\ I$
and *i_def*: $w\_i\ w = i$
using *w_def*(2)
unfolding *valid_window_matchF_def*
by (*auto simp*: *valid_window_matchF_def run_args*)
have *j_m*: $w\_j\ w \leq m$
using *ru_t_tau_in*[*OF reach_tj*, *of i*] *ru_t_tau_in*[*OF reach_tj*, *of m*] *memR_i_m i_m memR_xs*[*of m*]
unfolding *vs_tau*[*OF i_vs*] *vs_tau*[*OF m_vs*]
by (*force simp*: *in_set_conv_nth len_xs ts_at_def i_def*)
obtain *tm tm'* **where** *tm_def*: *reaches_on ru_t l_t0* (*take m* (*map fst vs*)) *tm*
$ru\_t\ tm = Some\ (tm', fst\ (vs\ !\ m))$
using *reaches_on_split*[**where** *?i*=*m*] *reaches_on_run_hd_t*[*OF prefix*] *m_vs*
by *fastforce*
have *reach_tj_m*: *reaches_on* (*w_run_t args*) (*w_tj w*) (*drop* (*w_j w*) (*take m* (*map fst vs*))) *tm*
using *reaches_on_split'*[*OF tm_def*(1), **where** *?i*=*w_j w*] *reaches_on_inj*[*OF reach_tj*] *m_vs j_m*
by (*auto simp*: *len_xs run_args*)
have *vs_m*: $fst\ (vs\ !\ m) = \tau\ \sigma\ m$
using *vs_tau*[*OF m_vs*] *m_vs*
by *auto*
have *memR_ti_m*: $\neg memR\ ti\ (\tau\ \sigma\ m)\ I$
using *memR_i_m*
unfolding *vs_tau*[*OF i_vs*] *vs_tau*[*OF m_vs*]
by (*auto simp*: *ti_def*)
have *m_prog*: $m \leq progress\ phi\ (map\ fst\ vs)$ **if** $phi \in set\ (collect\_subfmlas\ r\ [])$ **for** *phi*
using *collect_subfmlas_atms*[**where** *?r*=*r* **and** *?phis*=[]] *that atms_finite*[*of r*]
by (*auto simp*: *m_def min.coboundedI2*)
obtain *ws s* **where** *ws_def*: *reaches_on* (*run_subs* (*ru n*)) (*map* (*su n*) (*collect_subfmlas r* [])) *ws s* $length\ ws = m$
using *reaches_ons_runI*[**where** *?r*=*r* **and** *?n*=*n* **and** *?i*=*m*]
vydra_wf_reaches_on[**where** *?i*=*m* **and** *?n*=*n*] *subfmla*
MatchF(2) *m_prog*
by *fastforce*
have *reach_sj_m*: *reaches_on* (*run_subs* (*ru n*)) (*w_sj w*) (*drop* (*w_j w*) *ws*) *s*
using *reaches_on_split'*[*OF ws_def*(1), **where** *?i*=*w_j w*] *reaches_on_inj*[*OF reach_sj*] *i_m j_m*
by (*auto simp*: *ws_def*(2) *len_xs*)
define *rho* **where** *rho* = *zip* (*drop* (*w_j w*) (*take m* (*map fst vs*))) (*drop* (*w_j w*) *ws*)
have *map_fst_rho*: *map fst rho* = *drop* (*w_j w*) (*take m* (*map fst vs*))
and *map_snd_rho*: *map snd rho* = *drop* (*w_j w*) *ws*
using *ws_def*(2) *j_m m_vs*
by (*auto simp*: *rho_def*)
show *False*
using *valid_eval_matchF_complete*[*OF w_def*(2) *reach_tj_m*[*folded map_fst_rho*] *reach_sj_m*[*folded map_snd_rho run_args*] *read_ti run_t_read*[*OF tm_def*(2)[*folded run_args*], *unfolded vs_m*] *memR_ti_m*]

MatchF(4,7)
  by (*auto simp*: *w_def*(1) *args_def simp del*: *eval_matchF.simps*)
qed

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* $\varphi \leq n$
  shows *wf_vydra* φ *i n v*
  using *assms*(1)
proof (*induction i arbitrary*: *v*)
  case 0
  then show *?case*
    using *wf_vydra_sub*[*OF assms*(4)]
    by (*auto elim*: *reaches.cases*)
next
  case (*Suc i*)
  show *?case*
    using *reaches_Suc_split_last*[*OF Suc*(2)] *Suc*(1) *vydra_sound_aux*[*OF assms*(4) _ _ *assms*(2,3)]
    by *fastforce*
qed

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*)
  using *vydra_sound_aux*[*OF order.refl vydra_wf*[*OF assms*(1)[*unfolded ru'_def su'_def*] *assms*(3,4) *order.refl*] *assms*(2)[*unfolded ru'_def*] *assms*(3,4)]
  by *auto*

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 −
  have *aux*: *False* **if** *aux_assms*: *j* $\leq$ *n wf_vydra* φ *j* (*msize_fmla* φ) *v ru* (*msize_fmla* φ) *v* = *None* **for** *j v*
    using *vydra_complete_aux*[*OF prefix aux_assms*(2,3) _ *prog*(2−)] *aux_assms*(1) *prog*(1)
    by *auto*
  obtain *ws v* **where** *ws_def*: *reaches_on* (*ru'* φ) (*su'* φ) *ws v wf_vydra* φ *n* (*msize_fmla* φ) *v length ws* = *n*
    using *vydra_wf_reaches_on*[*OF* _ *prog*(2,3)] *aux*[*OF less_imp_le_nat*]
    unfolding *ru'_def su'_def*
    by *blast*
  have *ru_Some*: *ru'* φ *v* $\neq$ *None*
    using *aux*[*OF order.refl ws_def*(2)]
    by (*fastforce simp*: *ru'_def*)
  obtain *v' t b* **where** *tb_def*: *ru'* φ *v* = *Some* (*v'*, (*t*, *b*))
    using *ru_Some*
    by *auto*
  show *?thesis*
    using *reaches_on_n*[*OF ws_def*(1)] *tb_def vydra_sound'*[*OF reaches_on_n*[*OF ws_def*(1)] *tb_def prog*(2,3)]
    by (*auto simp*: *ws_def*(3))
qed

lemma *map_option_apfst_idle*: *map_option* (*apfst snd*) (*map_option* (*apfst* (*Pair n*)) *x*) = *x*
  by (*cases x*) *auto*

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 −
have *fst_v*: *fst v* = *msize_fmla* φ
by (*rule reaches_invar*[*OF assms*(*1*)]) (*auto simp*: *init_vydra_def run_vydra_def Let_def*)
have *reaches* (*ru'* φ) (*su'* φ) *n* (*snd v*)
using *reaches_cong*[*OF assms*(*1*), **where** *?P*=λ(*m*, *w*). *m* = *msize_fmla* φ **and** *?g*=*snd*]
by (*auto simp*: *init_vydra_def run_vydra_def ru'_def su'_def map_option_apfst_idle Let_def simp del*: *sub.simps*)
then show *?thesis*
using *vydra_sound'*[*OF _ _ assms*(*3*,*4*)] *assms*(*2*) *fst_v*
by (*auto simp*: *run_vydra_def ru'_def split*: *prod.splits*)
qed

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* (*run_vydra run_hd*) (*init_vydra init_hd run_hd* φ) *n v* $\wedge$ *run_vydra run_hd v* = *Some* (*v'*, (τ σ *n*, *sat* φ *n*))
proof −
obtain *v v'* **where** *wits*: *reaches* (*ru'* φ) (*su'* φ) *n v ru'* φ *v* = *Some* (*v'*, τ σ *n*, *sat* φ *n*)
using *vydra_complete'*[*OF assms*]
by *auto*
have *reach*: *reaches* (*run_vydra run_hd*) (*init_vydra init_hd run_hd* φ) *n* (*msize_fmla* φ, *v*)
using *reaches_cong*[*OF wits*(*1*), **where** *?P*=λ*x*. *True* **and** *?f'*=*run_vydra run_hd* **and** *?g*=*Pair* (*msize_fmla* φ)]
by (*auto simp*: *init_vydra_def run_vydra_def ru'_def su'_def Let_def simp del*: *sub.simps*)
show *?thesis*
apply (*rule exI*[*of _* (*msize_fmla* φ, *v*)])
apply (*rule exI*[*of _* (*msize_fmla* φ, *v'*)])
apply (*auto simp*: *run_vydra_def wits*(*2*)[*unfolded ru'_def*] *intro*: *reach*)
done
qed

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 −
obtain *vs* **where** *vs_def*: *reaches_on* (λ*i*. *if P i then Some* (*Suc i*, (τ σ *i*, Γ σ *i*)) *else None*) *s vs s' length vs* = *n*
using *reaches_on*[*OF assms*(*1*)]
by *auto*
have *s'* = *length vs*
using *vs_def*(*1*) *assms*(*2*)
by (*induction s vs s' rule*: *reaches_on_rev_induct*) (*auto split*: *if_splits*)
then show *?thesis*
using *vs_def*(*2*)
by *auto*
qed

interpretation *default_vydra*: *VYDRA_MDL* σ *0* λi. *Some* (*Suc i*, (τ σ *i*, Γ σ *i*))
using *reach_elem*[**where** *?P*=λ_. *True*]
by *unfold_locales auto*

end

lemma *reaches_inj*: *reaches r s i t* $\Longrightarrow$ *reaches r s i t'* $\Longrightarrow$ $t = t'$
using *reaches_on_inj reaches_on*
by *metis*

lemma *progress_sound*:
assumes
$\bigwedge n$. $n <$ *length ts* $\Longrightarrow$ *ts* ! $n = \tau$ σ n
$\bigwedge n$. $n <$ *length ts* $\Longrightarrow$ τ σ $n = \tau$ σ' n
$\bigwedge n$. $n <$ *length ts* $\Longrightarrow$ Γ σ $n = \Gamma$ σ' n
$n <$ *progress phi ts*
bounded_future_fmla phi
wf_fmla phi
shows *MDL.sat* σ *phi n* $\longleftrightarrow$ *MDL.sat* σ' *phi n*
proof −
define *run_hd* **where** *run_hd* = (λi. *if i < length ts then Some* (*Suc i*, (τ σ *i*, Γ σ *i*)) *else None*)
interpret *vydra*: *VYDRA_MDL* σ *0 run_hd*
using *MDL.reach_elem*[**where** *?P*=λi. *i < length ts*]
by *unfold_locales* (*auto simp*: *run_hd_def split*: *if_splits*)
define *run_hd'* **where** *run_hd'* = (λi. *if i < length ts then Some* (*Suc i*, (τ σ' *i*, Γ σ' *i*)) *else None*)
interpret *vydra'*: *VYDRA_MDL* σ' *0 run_hd'*
using *MDL.reach_elem*[**where** *?P*=λi. *i < length ts*]
by *unfold_locales* (*auto simp*: *run_hd'_def split*: *if_splits*)
have *run_hd_hd'*: *run_hd* = *run_hd'*
using *assms*(*1*−*3*)
by (*auto simp*: *run_hd_def run_hd'_def*)
have *reaches_run_hd*: $n \leq$ *length ts* $\Longrightarrow$ *reaches_on run_hd 0* (*map* (λi. (τ σ *i*, Γ σ *i*)) [*0*..<*n*]) *n*
for *n*
by (*induction n*) (*auto simp*: *run_hd_def intro*: *reaches_on.intros*(*1*) *intro*!: *reaches_on_app*)
have *ts_map*: *ts* = *map fst* (*map* (λi. (τ σ *i*, Γ σ *i*)) [*0*..<*length ts*])
by (*subst map_nth*[*symmetric*]) (*auto simp*: *assms*(*1*))
obtain *v v'* **where** *vv_def*: *reaches* (*run_vydra run_hd*) (*init_vydra 0 run_hd phi*) *n v run_vydra run_hd v* = *Some* (*v'*, τ σ *n*, *vydra.sat phi n*)
using *vydra.vydra_complete*[*OF reaches_run_hd*[*OF order.refl*] _ *assms*(*5*,*6*)] *assms*(*4*)
unfolding *ts_map*[*symmetric*]
by *blast*
have *reaches_run_hd'*: $n \leq$ *length ts* $\Longrightarrow$ *reaches_on run_hd' 0* (*map* (λi. (τ σ' *i*, Γ σ' *i*)) [*0*..<*n*]) *n* **for** *n*
by (*induction n*) (*auto simp*: *run_hd'_def intro*: *reaches_on.intros*(*1*) *intro*!: *reaches_on_app*)
have *ts'_map*: *ts* = *map fst* (*map* (λi. (τ σ' *i*, Γ σ' *i*)) [*0*..<*length ts*])
by (*subst map_nth*[*symmetric*]) (*auto simp*: *assms*(*1*,*2*))
obtain *w w'* **where** *ww_def*: *reaches* (*run_vydra run_hd'*) (*init_vydra 0 run_hd' phi*) *n w run_vydra run_hd' w* = *Some* (*w'*, τ σ' *n*, *vydra'.sat phi n*)
using *vydra'.vydra_complete*[*OF reaches_run_hd'*[*OF order.refl*] _ *assms*(*5*,*6*)] *assms*(*4*)
unfolding *ts'_map*[*symmetric*]
by *blast*
note *v_w* = *reaches_inj*[*OF vv_def*(*1*) *ww_def*(*1*)[*folded run_hd_hd'*]]
show *?thesis*
using *vv_def*(*2*) *ww_def*(*2*)
by (*auto simp*: *run_hd_hd' v_w*)
qed

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 *I* ('a, 't) *formula* | *Next* 't *I* ('a, 't) *formula* |
Since ('a, 't) *formula* 't *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*)
by (*induction phi rule*: *mdl2mdl_embed.induct*(*1*)[**where** *?Q*=λ*r*. *wf_regex* (*Times* (*embed r*) (*Symbol* (*Bool True*))) ∧ (∀ *phi* ∈ *atms* (*embed r*). *wf_fmla phi*)]) *auto*

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*
by (*induction r*) *auto*

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*
by (*induction phi rule*: *mdl2mdl_embed.induct*(*1*)[**where** *?Q*=λ*r*. *match* (*embed r*) = *rvmatch r*]) (*auto split*: *nat.splits*)

lemma *mdlstar2mdl*:
fixes *phi* :: ('*a*, '*t* :: *timestamp*) *Preliminaries.formula*
shows *wf_fmla* (*mdl2mdl phi*) ⋀*i*. *sat* (*mdl2mdl phi*) *i* ⟷ *rvsat phi i*
apply (*rule mdl2mdl_wf*)
apply (*rule mdl2mdl_equivalent*)
done

lemma *rvmatch_embed'*:
assumes ⋀*phi i*. *phi* ∈ *atms r* ⟹ *rvsat* (*mdl2mdl' phi*) *i* ⟷ *sat phi i*
shows *rvmatch* (*embed' mdl2mdl' r*) = *match r*
using *assms*
by (*induction r*) *auto*

lemma *mdl2mdlstar*:
fixes *phi* :: ('*a*, '*t* :: *timestamp*) *formula*
assumes *wf_fmla phi*

shows $\bigwedge i$. *rvsat* (*mdl2mdl′ phi*) *i* $\longleftrightarrow$ *sat phi i*
using *assms*
apply (*induction phi rule*: *mdl2mdl′.induct*)
apply (*auto split*: *nat.splits*)[*8*]
subgoal for *I r i*
by *auto* (*metis atms_rderive match_rderive rvmatch_embed′ wf_fmla.simps*(*1*))+
subgoal for *I r i*
by *auto* (*metis atms_rderive match_rderive rvmatch_embed′ wf_fmla.simps*(*1*))+
done

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 by *intro_classes*
(*auto simp*: *ccompare_enat_def split*: *if_splits intro*!: *comparator.intro*)

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*
by (*cases xs*; *cases ys*) *auto*

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
  apply standard
  apply unfold_locales
  using comparator.sym[OF ID_ccompare'] comparator.weak_eq[OF ID_ccompare']
    comparator.comp_trans[OF ID_ccompare'] iarray_list_of_inj
    apply (auto simp: ccompare_iarray_def split: option.splits)
   apply blast+
  done

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 (λx. x ≠ ∞)
  by (auto simp: tfin_enat_def)

lemma tfin_ereal_code[code]: (tfin :: ereal set) = Collect_set (λx. x ≠ −∞ ∧ x ≠ ∞)
```

by (*auto simp*: *tfin_ereal_def*)

lemma *Ball_atms*[*code_unfold*]: *Ball* (*atms r*) *P* = *list_all P* (*collect_subfmlas r* [])
using *collect_subfmlas_atms*[**where** *?phis*=[]]
by (*auto simp*: *list_all_def*)

lemma *MIN_fold*: (*MIN x*∈*set* (*z* # *zs*). *f x*) = *fold min* (*map f zs*) (*f z*)
by (*metis Min.set_eq_fold list.set_map list.simps*(*9*))

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*))
using *collect_subfmlas_atms*[**where** *?r*=*r* **and** *?phis*=[]] *atms_nonempty*[*of r*]
by (*auto split*: *list.splits*) (*smt* (*z3*) *MIN_fold*[**where** *?f*=λ*f*. *progress f ts*] *list.simps*(*15*))

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*}))
by (*auto simp*: *progress_matchP_code*[*unfolded progress.simps*])

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* × '*b*) *set* **where**
tfin_prod = *tfin* × *UNIV*

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

fun *sup_prod* :: '*a* × '*b* ⇒ '*a* × '*b* ⇒ '*a* × '*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* × '*b* ⇒ '*a* × '*b* ⇒ *bool* **where**
less_eq_prod (*a*, *b*) (*c*, *d*) ⟷ *a* < *c* ∨ (*a* = *c* ∧ *b* ≤ *d*)

definition *less_prod* :: '*a* × '*b* ⇒ '*a* × '*b* ⇒ *bool* **where**
less_prod x y ⟷ *x* ≤ *y* ∧ *x* ≠ *y*

instance
apply *standard*
apply (*auto simp*: *zero_prod_def less_prod_def*)[*2*]
subgoal for *x y z*
using *order.strict_trans*
by (*cases x*; *cases y*; *cases z*) *auto*

```
  subgoal for x y
    by (cases x; cases y) auto
  subgoal for x y
    by (cases x; cases y) (auto simp add: sup.commute sup.strict_order_iff)
  subgoal for x y
    apply (cases x; cases y)
    apply (auto simp add: sup.commute sup.strict_order_iff)
    apply (metis sup.absorb_iff2 sup.order_iff timestamp_total)
    done
  subgoal for y x z
    by (cases x; cases y; cases z) auto
  subgoal for i j
    using ι_mono less_le
    apply (auto simp: ι_prod_def less_prod_def)
    by (simp add: ι_mono)
  subgoal for i
    by (auto simp: ι_prod_def tfin_prod_def intro: ι_tfin)
  subgoal for x i
    apply (cases x)
    apply (auto simp: ι_prod_def tfin_prod_def)
    apply (metis ι_progressing dual_order.refl order_less_le)
    done
      apply (auto simp: tfin_prod_def tfin_closed)[1]
      apply (auto simp: zero_prod_def tfin_prod_def intro: zero_tfin)[1]
  subgoal for c d
    by (cases c; cases d) (auto simp: tfin_prod_def intro: tfin_closed)
  subgoal for c d a
    by (cases c; cases d; cases a) (auto simp: add_mono add_mono_strict)
  subgoal for a c
    apply (cases a; cases c)
    apply (auto simp: tfin_prod_def zero_prod_def)
     apply (metis less_eq_prod.simps add.right_neutral add_mono_strict less_prod_def order_le_less
order_less_le prod.inject)
    done
  subgoal for c d a
    apply (cases c; cases d; cases a)
    apply (auto simp add: add_mono_strict less_prod_def order.strict_implies_order)
      apply (metis add_mono_strict sup.strict_order_iff)
     apply (metis add_mono_strict sup.strict_order_iff)
    by (metis add_mono_strict dual_order.order_iff_strict less_le_not_le)
  subgoal for a b
    apply (cases a; cases b)
    apply (auto)
      apply (metis antisym_conv1 timestamp_total)
     apply (metis antisym_conv1 timestamp_total)
    apply (metis antisym_conv1 timestamp_total)
    apply (metis timestamp_total)
    done
  subgoal for a b
    apply (cases a; cases b)
    apply (auto simp: zero_prod_def tfin_prod_def order_less_le timestamp_tfin_le_not_tfin)
    done
  done

end

end
theory Timestamp_Prod
```

```
  imports Timestamp
begin

instantiation prod :: (timestamp, timestamp) timestamp
begin

definition tfin_prod :: ('a × 'b) set where
  tfin_prod = tfin × tfin

definition ι_prod :: nat ⇒ 'a × 'b where
  ι_prod n = (ι n, ι n)

fun sup_prod :: 'a × 'b ⇒ 'a × 'b ⇒ 'a × 'b where
  sup_prod (a, b) (c, d) = (sup a c, sup b d)

fun less_eq_prod :: 'a × 'b ⇒ 'a × 'b ⇒ bool where
  less_eq_prod (a, b) (c, d) ⟷ a ≤ c ∧ b ≤ d

definition less_prod :: 'a × 'b ⇒ 'a × 'b ⇒ bool where
  less_prod x y ⟷ x ≤ y ∧ x ≠ y

instance
  apply standard
            apply (auto simp: add.commute zero_prod_def less_prod_def)[7]
  subgoal for i j
    using ι_mono ι_mono less_le
    by (fastforce simp: ι_prod_def less_prod_def)
  subgoal for i
    by (auto simp: ι_prod_def tfin_prod_def intro: ι_tfin)
  subgoal for x i
    apply (cases x)
    using ι_progressing
    by (auto simp: tfin_prod_def ι_prod_def)
  apply (auto simp: zero_prod_def tfin_prod_def intro: zero_tfin)[1]
  subgoal for c d
    by (cases c; cases d) (auto simp: tfin_prod_def intro: tfin_closed)
  subgoal for c d a
    by (cases c; cases d; cases a) (auto simp add: add_mono)
  subgoal for a c
    apply (cases a; cases c)
    apply (auto simp: tfin_prod_def zero_prod_def)
    apply (metis add.right_neutral add_pos less_eq_prod.simps less_prod_def order_less_le prod.inject
timestamp_class.add_mono)
    done
  done

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.