Upper Bounding Diameters of State Spaces of Factored Transition Systems

Friedrich Kurz and Mohammad Abdulaziz

March 17, 2025

Abstract

A *completeness threshold* is required to guarantee the completeness of planning as satisfiability, and bounded model checking of safety properties. One valid completeness threshold is the *diameter* of the underlying transition system. The diameter is the maximum element in the set of lengths of all shortest paths between pairs of states. The diameter is not calculated exactly in our setting, where the transition system is succinctly described using a (propositionally) factored representation. Rather, an upper bound on the diameter is calculated compositionally, by bounding the diameters of small abstract subsystems, and then composing those.

We port a HOL4 formalisation of a compositional algorithm for computing a relatively tight upper bound on the system diameter. This compositional algorithm exploits acyclicity in the state space to achieve compositionality, and it was introduced by Abdulaziz et. al [1] (in particular Algorithm 1). The formalisation that we port is described as a part of another paper by Abdulaziz et. al [2], in particular in section 6. As a part of this porting we developed a libray about transition systems, which shall be of use in future related mechanisation efforts.

Contents

theory *FactoredSystemLib*
imports *Main HOL−Library.Finite-Map*
begin

1 Factored Systems Library

This section contains definitions used in the factored system theory (FactoredSystem.thy) and in other theories.

1.1 Semantics of Map Addition

Most importantly, we are redefining the map addition operator (‘++‘) to reflect HOL4 semantics which are left to right (ltr), rather than right-to-left as in Isabelle/HOL.

This means that given a finite map (‘M = M1 ++ M2‘) and a variable ‘v‘ which is in the domain of both ‘M1‘ and ‘M2‘, the lookup ‘M v‘ will yield ‘M1 v‘ in HOL4 but ‘M2 v‘ in Isabelle/HOL. This behavior can be confirmed by looking at the definition of ‘fmap_add‘ (‘++f‘, Finite_Map.thy:460)—which is lifted from ‘map_add‘ (Map.thy:24)

(++) (infixl "++" 100) where *m1 ++ m2 =* (*λx. case m2 x of None* ⇒ *m1 x* | *Some y* ⇒ *Some y*)

to finite sets—and the HOL4 definition of "FUNION‘ (finite_mapScript.sml:770) which recurs on ‘union_lemma‘ (finite_mapScript.sml:756)

!f̂map g. ?union. (FDOM union = FDOM f Union (g ‘ FDOM)) / (!x. FAPPLY union x = if x IN FDOM f then FAPPLY f x else FAPPLY g x)

The ltr semantics are also reflected in [Abdulaziz et al., Definition 2, p.9].

hide-const (**open**) *Map.map-add*
no-notation *Map.map-add* (**infixl** ‹++› *100*)
definition *fmap-add-ltr* :: (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'b*) *fmap* (**infixl** ‹++› *100*) **where**
  $m1 \mathbin{++} m2 \equiv m2 \mathbin{++_f} m1$

1.2 States, Actions and Problems.

Planning problems are typically formalized by considering possible states and the effect of actions upon these states.

In this case we consider a world model in propositional logic: i.e. states are finite maps of variables (with arbitrary type ’a) to boolean values and actions are pairs of states where the first component specifies preconditions and the second component specifies effects (postconditions) of applying the action to a given state. [Abdulaziz et al., Definition 2, p.9]

type-synonym (*'a*) *state* = (*'a*, *bool*) *fmap*
type-synonym (*'a*) *action* = (*'a state* × *'a state*)
type-synonym (*'a*) *problem* = (*'a state* × *'a state*) *set*

For a given action $\pi = (p, e)$ the action domain $\mathcal{D}\ \pi$ is the set of variables ‘v‘ where a value is assigned to ‘v‘ in either ‘p‘ or ‘e‘, i.e. ‘p v‘ or ‘e v‘ are defined. [Abdulaziz et al., Definition 2, p.9]

definition *action-dom* **where**
  action-dom s1 s2 ≡ (*fmdom' s1* ∪ *fmdom' s2*)

— NOTE lemma ‘action_dom_pair‘
action_dom a = FDOM (FST a) Union ((SND a) ‘ FDOM)

was removed because the curried definition of ‘action_dom‘ in the translation makes it redundant.

Now, for a given problem (i.e. action set) δ, the problem domain $\mathcal{D}\ \delta$ is given by the union of the action domains of all actions in δ. [Abdulaziz et al., Definition 3, p.9]

Moreover, the set of valid states $U\ \delta$ is given by the union over all states whose domain is equal to the problem domain and the set of valid action sequences (or, valid plans) is given by the Kleene closure of δ, i.e. $\delta\text{-}star = \{\pi.\ set\ \pi \subseteq \delta\}$. [Abdulaziz et al., Definition 3, p.9]

Ultimately, the effect of executing an action ‘a‘ on a state ‘s‘ is given by calculating the succeding state. In general, the succeding state is either the preceding state—if the action does not apply to the state, i.e. if the preconditions are not met—; or, the union of the effects of the action application and the state. [Abdulaziz et al., Definition 3, p.9]

definition *prob-dom* **where**
  $prob\text{-}dom\ prob \equiv \bigcup\ ((\lambda\ (s1,\ s2).\ action\text{-}dom\ s1\ s2)\ `\ prob)$

definition *valid-states* **where**
  $valid\text{-}states\ prob \equiv \{s.\ fmdom'\ s = prob\text{-}dom\ prob\}$

definition *valid-plans* **where**
  $valid\text{-}plans\ prob \equiv \{as.\ set\ as \subseteq prob\}$

definition *state-succ* **where**
  $state\text{-}succ\ s\ a \equiv (if\ fst\ a \subseteq_f\ s\ then\ (snd\ a\ {+}{+}\ s)\ else\ s)$

end
theory *ListUtils*
  imports *Main HOL−Library.Sublist*
begin

— TODO assure translations * ’sublist’ –> ’subseq’ * list_frag l l’ –> sublist l’ l (switch operands!)

lemma *len-ge-0*:
  fixes *l*
  shows $length\ l \geq 0$
  ⟨*proof*⟩

lemma *len-gt-pref-is-pref*:
  fixes *l l1 l2*
  assumes $(length\ l2 > length\ l1)\ (prefix\ l1\ l)\ (prefix\ l2\ l)$
  shows $(prefix\ l1\ l2)$
  ⟨*proof*⟩

lemma *nempty-list-append-length-add*:
  fixes *l1 l2 l3*

assumes $l2 \neq []$
shows $length\ (l1\ @\ l3) < length\ (l1\ @\ l2\ @l3)$
⟨proof⟩

lemma *append-filter*:
fixes $f1 :: 'a \Rightarrow bool$ **and** $f2\ as1\ as2$ **and** $p :: 'a\ list$
assumes $(as1\ @\ as2 = filter\ f1\ (map\ f2\ p))$
shows $(\exists\, p\text{-}1\ p\text{-}2.$
$(p\text{-}1\ @\ p\text{-}2 = p)$
$\wedge\ (as1 = filter\ f1\ (map\ f2\ p\text{-}1))$
$\wedge\ (as2 = filter\ f1\ (map\ f2\ p\text{-}2))$
)
⟨proof⟩
lemma *append-eq-as-proj-1*:
fixes $f1 :: 'a \Rightarrow bool$ **and** $f2\ as1\ as2\ as3$ **and** $p :: 'a\ list$
assumes $(as1\ @\ as2\ @\ as3 = filter\ f1\ (map\ f2\ \ p))$
shows $(\exists\, p\text{-}1\ p\text{-}2\ p\text{-}3.$
$(p\text{-}1\ @\ p\text{-}2\ @\ p\text{-}3 = p)$
$\wedge\ (as1 = filter\ f1\ (map\ f2\ p\text{-}1))$
$\wedge\ (as2 = filter\ f1\ (map\ f2\ p\text{-}2))$
$\wedge\ (as3 = filter\ f1\ (map\ f2\ p\text{-}3))$
)
⟨proof⟩

lemma *filter-empty-every-not*: $\bigwedge P\ l.\ (filter\ (\lambda x.\ P\ x)\ l = []) = list\text{-}all\ (\lambda x.\ \neg P\ x)\ l$
⟨proof⟩
lemma *MEM-SPLIT*:
fixes $x\ l$
assumes $\neg ListMem\ x\ l$
shows $\forall\, l1\ l2.\ l \neq l1\ @\ [x]\ @\ l2$
⟨proof⟩
lemma *APPEND-EQ-APPEND-MID*:
fixes $l1\ l2\ m1\ m2\ e$
shows
$(l1\ @\ [e]\ @\ l2 = m1\ @\ m2)$
$\longleftrightarrow$
$(\exists\, l.\ (m1 = l1\ @\ [e]\ @\ l) \wedge (l2 = l\ @\ m2)) \vee$
$(\exists\, l.\ (l1 = m1\ @\ l) \wedge (m2 = l\ @\ [e]\ @\ l2))$
⟨proof⟩
lemma *LIST-FRAG-DICHOTOMY*:
fixes $l\ la\ x\ lb$
assumes $sublist\ l\ (la\ @\ [x]\ @\ lb)\ \neg ListMem\ x\ l$
shows $sublist\ l\ la \vee sublist\ l\ lb$
⟨proof⟩

lemma *LIST-FRAG-DICHOTOMY-2*:

fixes *l la x lb P*
assumes *sublist l* (*la* @ [*x*] @ *lb*) ¬*P x list-all P l*
shows *sublist l la* ∨ *sublist l lb*
⟨*proof*⟩

lemma *frag-len-filter-le*:
fixes *P l′ l*
assumes *sublist l′ l*
shows *length* (*filter P l′*) ≤ *length* (*filter P l*)
⟨*proof*⟩

end

theory *FSSublist*
imports *Main HOL−Library.Sublist ListUtils*
begin

This file is a port of the original HOL4 source file sublistScript.sml.

2 Factored System Sublist

2.1 Sublist Characterization

We take a look at the characterization of sublists. As a precursor, we are replacing the original definition of 'sublist' in HOL4 (sublistScript.sml:10) with the semantically equivalent 'subseq' of Isabelle/HOL's to be able to use the associated theorems and automation.

In HOL4 'sublist' is defined as

(sublist [] l1 = T) / (sublist (h::t) [] = F) / (sublist (x::l1) (y::l2) = (x = y) / sublist l1 l2 sublist (x::l1) l2)

[Abdulaziz et al., HOL4 Definition 10, p.19]. Whereas 'subseq' (Sublist.tyh:927) is defined as an abbrevation of 'list_emb' with the predicate (=), i.e.

subseq xs ys ≡ *subseq xs ys*

'list_emb' itself is defined as an inductive predicate. However, an equivalent function definition is provided in 'list_emb_code' (Sublist.thy:784) which is very close to 'sublist' in HOL4.

The correctness of the equivalence claim is shown below by the technical lemma 'sublist_HOL4_equiv_subseq' (where the HOL4 definition of 'sublist' is renamed to 'sublist_HOL4').

fun *sublist-HOL4* **where**
sublist-HOL4 [] *l1* = *True*
| (*sublist-HOL4* (*h* # *t*) [] = *False*)
| (*sublist-HOL4* (*x* # *l1*) (*y* # *l2*) = ((*x* = *y*) ∧ *sublist-HOL4 l1 l2* ∨ *sublist-HOL4* (*x* # *l1*) *l2*))

— NOTE added lemma
lemma *sublist-HOL4-equiv-subseq*:
  fixes *l1 l2*
  shows *sublist-HOL4 l1 l2* ⟷ *subseq l1 l2*
⟨*proof*⟩

Likewise as with ‘sublist‘ and ‘subseq‘, the HOL4 definition of ‘list_frag‘ (list_utilsScript.sml:207) has a an Isabelle/HOL counterpart in ‘sublist‘ (Sublist.thy:1124).

The equivalence claim is proven in the technical lemma ‘list_frag_HOL4_equiv_sublist‘. Note that ‘sublist‘ reverses the argument order of ‘list_frag‘. Other than that, both definitions are syntactically identical.

definition *list-frag-HOL4* **where**
  list-frag-HOL4 l frag ≡ ∃ *pfx sfx*. *pfx* @ *frag* @ *sfx* = *l*

lemma *list-frag-HOL4-equiv-sublist*:
  shows *list-frag-HOL4 l l′* ⟷ *sublist l′ l*
  ⟨*proof*⟩

Given these equivalences, occurences of ‘sublist‘ and ‘list_frag‘ in the original HOL4 source are now always translated directly to ‘subseq‘ and ‘sublist‘ respectively.

The remainer of this subsection is concerned with characterizations of ‘sublist‘/ ‘subseq‘.

lemma *sublist-EQNS*:
  subseq [] *l* = *True*
  subseq (*h* # *t*) [] = *False*
  ⟨*proof*⟩

lemma *sublist-refl*: *subseq l l*
  ⟨*proof*⟩

lemma *sublist-cons*:
  assumes *subseq l1 l2*
  shows *subseq l1* (*h* # *l2*)
  ⟨*proof*⟩

lemma *sublist-NIL*: *subseq l1* [] = (*l1* = [])
  ⟨*proof*⟩

lemma *sublist-trans*:
  fixes *l1 l2*
  assumes *subseq l1 l2 subseq l2 l3*

shows *subseq l1 l3*
⟨*proof*⟩
lemma *sublist-length*:
fixes *l l′*
assumes *subseq l l′*
shows *length l* ≤ *length l′*
⟨*proof*⟩
lemma *sublist-CONS1-E*:
fixes *l1 l2*
assumes *subseq* (*h* # *l1*) *l2*
shows *subseq l1 l2*
⟨*proof*⟩

lemma *sublist-equal-lengths*:
fixes *l1 l2*
assumes *subseq l1 l2* (*length l1* = *length l2*)
shows (*l1* = *l2*)
⟨*proof*⟩
lemma *sublist-antisym*:
assumes *subseq l1 l2 subseq l2 l1*
shows (*l1* = *l2*)
⟨*proof*⟩

lemma *sublist-append-back*:
fixes *l1 l2*
shows *subseq l1* (*l2* @ *l1*)
⟨*proof*⟩
lemma *sublist-snoc*:
fixes *l1 l2*
assumes *subseq l1 l2*
shows *subseq l1* (*l2* @ [*h*])
⟨*proof*⟩

lemma *sublist-append-front*:
fixes *l1 l2*
shows *subseq l1* (*l1* @ *l2*)
⟨*proof*⟩

lemma *append-sublist-1*:
assumes *subseq* (*l1* @ *l2*) *l*
shows *subseq l1 l* ∧ *subseq l2 l*
⟨*proof*⟩
lemma *sublist-prefix*:
shows *subseq* (*h* # *l1*) *l2* ⟹ ∃ *l2a l2b*. *l2* = *l2a* @ [*h*] @ *l2b* ∧ ¬*ListMem h l2a*
⟨*proof*⟩

lemma *sublist-skip*:
fixes *l1 l2 h l1'*
assumes $l1 = (h \# l1')$ $l2 = l2a @ [h] @ l2b$ *subseq l1 l2* $\neg(ListMem\ h\ l2a)$
shows *subseq l1* $(h \# l2b)$
⟨*proof*⟩
lemma *sublist-split-trans*:
fixes *l1 l2 h l1'*
assumes $l1 = (h \# l1')$ $l2 = l2a @ [h] @ l2b$ *subseq l1 l2* $\neg(ListMem\ h\ l2a)$
shows *subseq l1' l2b*
⟨*proof*⟩

lemma *sublist-cons-exists*:
shows
subseq $(h \# l1)$ *l2*
$\longleftrightarrow (\exists\, l2a\ l2b.\ (l2 = l2a @ [h] @ l2b) \wedge \neg ListMem\ h\ l2a \wedge subseq\ l1\ l2b)$

⟨*proof*⟩

lemma *sublist-append-exists*:
fixes *l1 l2*
shows *subseq* $(l1 @ l2)$ $l3 \Longrightarrow \exists\, l3a\ l3b.\ (l3 = l3a @ l3b) \wedge subseq\ l1\ l3a \wedge subseq\ l2\ l3b$
⟨*proof*⟩
lemma *sublist-append-both-I*:
assumes *subseq a b subseq c d*
shows *subseq* $(a @ c)$ $(b @ d)$
⟨*proof*⟩

lemma *sublist-append*:
assumes *subseq l1 l1' subseq l2 l2'*
shows *subseq* $(l1 @ l2)$ $(l1' @ l2')$
⟨*proof*⟩

lemma *sublist-append2*:
assumes *subseq l1 l2*
shows *subseq l1* $(l2 @ l3)$
⟨*proof*⟩

lemma *append-sublist*:
shows *subseq* $(l1 @ l2 @ l3)$ $l \Longrightarrow$ *subseq* $(l1 @ l3)$ *l*
⟨*proof*⟩

lemma *sublist-subset*:
assumes *subseq l1 l2*

shows *set l1* $\subseteq$ *set l2*
⟨*proof*⟩

lemma *sublist-filter*:
fixes *P l*
shows *subseq* (*filter P l*) *l*
⟨*proof*⟩

lemma *sublist-cons-2*:
fixes *l1 l2 h*
shows (*subseq* (*h* # *l1*) (*h* # *l2*) $\longleftrightarrow$ (*subseq l1 l2*))
⟨*proof*⟩

lemma *sublist-every*:
fixes *l1 l2 P*
assumes (*subseq l1 l2* $\wedge$ *list-all P l2*)
shows *list-all P l1*
⟨*proof*⟩

lemma *sublist-SING-MEM*: *subseq* [*h*] *l* $\longleftrightarrow$ *ListMem h l*
⟨*proof*⟩
lemma *sublist-append-exists-2*:
fixes *l1 l2 l3*
assumes *subseq* (*h* # *l1*) *l2*
shows ($\exists$ *l3 l4*. (*l2* = *l3* @ [*h*] @ *l4*) $\wedge$ (*subseq l1 l4*))
⟨*proof*⟩

lemma *sublist-append-4*:
fixes *l l1 l2 h*
assumes (*subseq* (*h* # *l*) (*l1* @ [*h*] @ *l2*)) (*list-all* (λx. $\neg$(*h* = *x*)) *l1*)
shows *subseq l l2*
⟨*proof*⟩

lemma *sublist-append-5*:
fixes *l l1 l2 h*
assumes (*subseq* (*h* # *l*) (*l1* @ *l2*)) (*list-all* (λx. $\neg$(*h* = *x*)) *l1*)
shows *subseq* (*h* # *l*) *l2*
⟨*proof*⟩

lemma *sublist-append-6*:
fixes *l l1 l2 h*
assumes (*subseq* (*h* # *l*) (*l1* @ *l2*)) ($\neg$(*ListMem h l1*))

shows *subseq* $(h \# l)$ *l2*
⟨*proof*⟩

lemma *sublist-MEM*:
fixes *h l1 l2*
shows *subseq* $(h \# l1)$ *l2* $\Longrightarrow$ *ListMem h l2*
⟨*proof*⟩

lemma *sublist-cons-4*:
fixes *l h l'*
shows *subseq l l'* $\Longrightarrow$ *subseq l* $(h \# l')$
⟨*proof*⟩

2.2 Main Theorems

theorem *sublist-imp-len-filter-le*:
fixes *P l l'*
assumes *subseq l' l*
shows *length* (*filter P l'*) $\leq$ *length* (*filter P l*)
⟨*proof*⟩
theorem *list-with-three-types-shorten-type2*:
fixes *P1 P2 P3 k1 f PProbs PProbl s l*
assumes (*PProbs s*) (*PProbl l*)
(∀ *l s*.
(*PProbs s*)
∧ (*PProbl l*)
∧ (*list-all P1 l*)
⟶ (∃ *l'*.
(*f s l'* = *f s l*)
∧ (*length* (*filter P2 l'*) ≤ *k1*)
∧ (*length* (*filter P3 l'*) ≤ *length* (*filter P3 l*))
∧ (*list-all P1 l'*)
∧ (*subseq l' l*)
)
)
(∀ *s l1 l2*. *f* (*f s l1*) *l2* = *f s* (*l1* @ *l2*))
(∀ *s l*. (*PProbs s*) ∧ (*PProbl l*) ⟶ (*PProbs* (*f s l*)))
(∀ *l1 l2*. (*subseq l1 l2*) ∧ (*PProbl l2*) ⟶ (*PProbl l1*))
(∀ *l1 l2*. *PProbl* (*l1* @ *l2*) ⟷ (*PProbl l1* ∧ *PProbl l2*))
shows (∃ *l'*.
(*f s l'* = *f s l*)
∧ (*length* (*filter P3 l'*) ≤ *length* (*filter P3 l*))
∧ (∀ *l''*.
(*sublist l'' l'*) ∧ (*list-all P1 l''*)
⟶ (*length* (*filter P2 l''*) ≤ *k1*)
)
∧ (*subseq l' l*)

)
⟨*proof*⟩

lemma *isPREFIX-sublist*:
fixes *x y*
assumes *prefix x y*
shows *subseq x y*
⟨*proof*⟩

end
theory *HoArithUtils*
imports *Main*
begin

lemma *general-theorem*:
fixes *P f* **and** *l* :: *nat*
assumes $(\forall p.\ P\ p \wedge f\ p > l \longrightarrow (\exists p'.\ P\ p' \wedge f\ p' < f\ p))$
shows $(\forall p.\ P\ p \longrightarrow (\exists p'.\ P\ p' \wedge f\ p' \leq l))$
⟨*proof*⟩

end
theory *FmapUtils*
imports *HOL−Library.Finite-Map FactoredSystemLib*
begin

— TODO A lemma 'fmrestrict_set_twice_eq' 'fmrestrict_set ?vs (fmrestrict_set ?vs ?f) = fmrestrict_set ?vs ?f' to replace the recurring proofs steps using 'by (simp add: fmfilter_alt_defs(4))' would make sense.

— NOTE hide the '++' operator from 'Map' to prevent warnings.
hide-const (**open**) *Map.map-add*
no-notation *Map.map-add* (**infixl** ‹++› *100*)

— TODO more explicit proof.
lemma *IN-FDOM-DRESTRICT-DIFF*:
fixes *vs v f*
assumes $\neg(v \in vs)$ $fmdom'\ f \subseteq fdom$ $v \in fmdom'\ f$
shows $v \in fmdom'\ (fmrestrict\text{-}set\ (fdom - vs)\ f)$
⟨*proof*⟩

lemma *disj-dom-drest-fupdate-eq*:
$disjnt\ (fmdom'\ x)\ vs \Longrightarrow (fmrestrict\text{-}set\ vs\ s = fmrestrict\text{-}set\ vs\ (x\ {+}{+}\ s))$

⟨*proof*⟩
lemma *graph-plan-card-state-set*:
fixes *PROB vs*
assumes *finite vs*

shows *card* (*fmdom′* (*fmrestrict-set vs s*)) ≤ *card vs*
⟨*proof*⟩

lemma *exec-drest-5*:
fixes *x vs*
assumes *fmdom′ x* ⊆ *vs*
shows (*fmrestrict-set vs x* = *x*)
⟨*proof*⟩

lemma *graph-plan-lemma-5*:
fixes *s s′ vs*
assumes (*fmrestrict-set* (*fmdom′ s* − *vs*) *s* = *fmrestrict-set* (*fmdom′ s′* − *vs*) *s′*)
(*fmrestrict-set vs s* = *fmrestrict-set vs s′*)
shows (*s* = *s′*)
⟨*proof*⟩

lemma *drest-smap-drest-smap-drest*:
fixes *x s vs*
shows *fmrestrict-set vs x* $\subseteq_f$ *s* ⟷ *fmrestrict-set vs x* $\subseteq_f$ *fmrestrict-set vs s*
⟨*proof*⟩

lemma *sat-precond-as-proj-1*:
fixes *s s′ vs x*
assumes *fmrestrict-set vs s* = *fmrestrict-set vs s′*
shows *fmrestrict-set vs x* $\subseteq_f$ *s* ⟷ *fmrestrict-set vs x* $\subseteq_f$ *s′*
⟨*proof*⟩

lemma *sat-precond-as-proj-4*:
fixes *fm1 fm2 vs*
assumes *fm2* $\subseteq_f$ *fm1*
shows (*fmrestrict-set vs fm2* $\subseteq_f$ *fm1*)
⟨*proof*⟩

lemma *sublist-as-proj-eq-as-1*:
fixes *x s vs*
assumes (*x* $\subseteq_f$ *fmrestrict-set vs s*)
shows (*x* $\subseteq_f$ *s*)
⟨*proof*⟩

lemma *limited-dom-neq-restricted-neq*:
assumes *fmdom′ f1* ⊆ *vs f1* ++ *f2* ≠ *f2*
shows *fmrestrict-set vs* (*f1* ++ *f2*) ≠ *fmrestrict-set vs f2*
⟨*proof*⟩

lemma *fmlookup-fmrestrict-set-dom*: ⋀*vs s. dom* (*fmlookup* (*fmrestrict-set vs s*)) = *vs* ∩ (*fmdom′ s*)
⟨*proof*⟩

end

theory *FactoredSystem*
imports *Main HOL−Library.Finite-Map HOL−Library.Sublist FSSublist FactoredSystemLib ListUtils HoArithUtils FmapUtils*
begin

3 Factored System

hide-const (**open**) *Map.map-add*
no-notation *Map.map-add* (**infixl** ‹++› *100*)

3.1 Semantics of Plan Execution

This section aims at characterizing the semantics of executing plans—i.e. sequences of actions—on a given initial state.

The semantics of action execution were previously introduced via the notion of succeding state (‘state_succ‘). Plan execution (‘exec_plan‘) extends this notion to sequences of actions by calculating the succeding state from the given state and action pair and then recursively executing the remaining actions on the succeding state. [Abdulaziz et al., HOL4 Definition 3, p.9]

lemma *state-succ-pair*: *state-succ s* (*p*, *e*) = (*if* (*p* $\subseteq_f$ *s*) *then* (*e* ++ *s*) *else s*)
⟨*proof*⟩
fun *exec-plan* **where**
exec-plan s [] = *s*
| *exec-plan s* (*a* # *as*) = *exec-plan* (*state-succ s a*) *as*

lemma *exec-plan-Append*:
fixes *as-a as-b s*
shows *exec-plan s* (*as-a* @ *as-b*) = *exec-plan* (*exec-plan s as-a*) *as-b*
⟨*proof*⟩

Plan execution effectively eliminates cycles: i.e., if a given plan ‘as‘ may be partitioned into plans ‘as1‘, ‘as2‘ and ‘as3‘, s.t. the sequential execution of ‘as1‘ and ‘as2‘ yields the same state, ‘as2‘ may be skipped during plan execution.

lemma *cycle-removal-lemma*:
fixes *as1 as2 as3*
assumes (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*)
shows (*exec-plan s* (*as1* @ *as2* @ *as3*) = *exec-plan s* (*as1* @ *as3*))
⟨*proof*⟩

3.1.1 Characterization of the Set of Possible States

To show the construction principle of the set of possible states—in lemma ‘construction_of_all_possible_states_lemma‘—the following ancillary proves of finite map properties are required.

Most importantly, in lemma ‘fmupd_fmrestrict_subset‘ we show how finite mappings ‘s‘ with domain $\{v\} \cup X$ and ‘s v = (Some x)‘ are constructed from their restrictions to ‘X‘ via update, i.e.

s = fmupd v x (fmrestrict_set X s)

This is used in lemma ‘construction_of_all_possible_states_lemma‘ to show that the set of possible states for variables $\{v\} \cup X$ is constructed inductively from the set of all possible states for variables ‘X‘ via update on point $v \notin X$.

lemma *empty-domain-fmap-set*: $\{s.\ fmdom'\ s = \{\}\} = \{fmempty\}$
⟨*proof*⟩
lemma *possible-states-set-ii-a*:
  fixes *s x v*
  assumes $(v \in fmdom'\ s)$
  shows $(fmdom'\ ((\lambda s.\ fmupd\ v\ x\ s)\ s) = fmdom'\ s)$
  ⟨*proof*⟩
lemma *possible-states-set-ii-b*:
  fixes *s x v*
  assumes $(v \notin fmdom'\ s)$
  shows $(fmdom'\ ((\lambda s.\ fmupd\ v\ x\ s)\ s) = fmdom'\ s \cup \{v\})$
  ⟨*proof*⟩
lemma *fmap-neq*:
  fixes $s :: ('a,\ bool)\ fmap$ **and** $s' :: ('a,\ bool)\ fmap$
  assumes $(fmdom'\ s = fmdom'\ s')$
  shows $((s \neq s') \longleftrightarrow (\exists\, v{\in}(fmdom'\ s).\ fmlookup\ s\ v \neq fmlookup\ s'\ v))$
  ⟨*proof*⟩
lemma *fmdom′-fmsubset-restrict-set*:
  fixes *X1 X2* **and** $s :: ('a,\ bool)\ fmap$
  assumes $X1 \subseteq X2$ $fmdom'\ s = X2$
  shows $fmdom'\ (fmrestrict\text{-}set\ X1\ s) = X1$
  ⟨*proof*⟩
lemma *fmsubset-restrict-set*:
  fixes *X1 X2* **and** $s :: 'a\ state$
  assumes $X1 \subseteq X2$ $s \in \{s.\ fmdom'\ s = X2\}$
  shows $fmrestrict\text{-}set\ X1\ s \in \{s.\ fmdom'\ s = X1\}$
  ⟨*proof*⟩
lemma *fmupd-fmsubset-restrict-set*:
  fixes *X v x* **and** $s :: 'a\ state$
  assumes $s \in \{s.\ fmdom'\ s = insert\ v\ X\}$ $fmlookup\ s\ v = Some\ x$
  shows $s = fmupd\ v\ x\ (fmrestrict\text{-}set\ X\ s)$
⟨*proof*⟩

lemma *construction-of-all-possible-states-lemma*:
  fixes *v X*
  assumes $(v \notin X)$
  shows $(\{s.\ fmdom'\ s = insert\ v\ X\}$
    $= ((\lambda s.\ fmupd\ v\ True\ s)$ ‘ $\{s.\ fmdom'\ s = X\})$
      $\cup\ ((\lambda s.\ fmupd\ v\ False\ s)$ ‘ $\{s.\ fmdom'\ s = X\})$
)

⟨*proof*⟩

Another important property of the state set is cardinality, i.e. the number of distinct states which can be modelled using a given finite variable set.

As lemma ‘card_of_set_of_all_possible_states‘ shows, for a finite variable set ‘X‘, the number of possible states is ‘2 ĉard X‘, i.e. the number of assigning two discrete values to ‘card X‘ slots as known from combinatorics.

Again, some additional properties of finite maps had to be proven. Pivotally, in lemma ‘updates_disjoint‘, it is shown that the image of updating a set of states with domain ‘X‘ on a point $x \notin X$ with either ‘True‘ or ‘False‘ yields two distinct sets of states with domain $\{x\} \cup X$.

lemma *FINITE-states*:
  fixes X :: *'a set*
  shows $\mathit{finite}\ X \Longrightarrow \mathit{finite}\ \{(s :: \mathit{'a\ state}).\ \mathit{fmdom'}\ s = X\}$
⟨*proof*⟩
lemma *bool-update-effect*:
  fixes $s\ X\ x\ v\ b$
  assumes $\mathit{finite}\ X\ s \in \{s :: \mathit{'a\ state}.\ \mathit{fmdom'}\ s = X\}\ x \in X\ x \neq v$
  shows $\mathit{fmlookup}\ ((\lambda s :: \mathit{'a\ state}.\ \mathit{fmupd}\ v\ b\ s)\ s)\ x = \mathit{fmlookup}\ s\ x$
  ⟨*proof*⟩
lemma *bool-update-inj*:
  fixes X :: *'a set* **and** $v\ b$
  assumes $\mathit{finite}\ X\ v \notin X$
  shows $\mathit{inj\text{-}on}\ (\lambda s.\ \mathit{fmupd}\ v\ b\ s)\ \{s :: \mathit{'a\ state}.\ \mathit{fmdom'}\ s = X\}$
⟨*proof*⟩
lemma *card-update*:
  fixes $X\ v\ b$
  assumes $\mathit{finite}\ (X :: \mathit{'a\ set})\ v \notin X$
  shows
    $\mathit{card}\ ((\lambda s.\ \mathit{fmupd}\ v\ b\ s)\ \text{‘}\ \{s :: \mathit{'a\ state}.\ \mathit{fmdom'}\ s = X\})$
    $= \mathit{card}\ \{s :: \mathit{'a\ state}.\ \mathit{fmdom'}\ s = X\}$

⟨*proof*⟩
lemma *updates-disjoint*:
  fixes $X\ x$
  assumes $\mathit{finite}\ X\ x \notin X$
  shows
    $((\lambda s.\ \mathit{fmupd}\ x\ \mathit{True}\ s)\ \text{‘}\ \{s.\ \mathit{fmdom'}\ s = X\})$
    $\cap ((\lambda s.\ \mathit{fmupd}\ x\ \mathit{False}\ s)\ \text{‘}\ \{s.\ \mathit{fmdom'}\ s = X\}) = \{\}$

⟨*proof*⟩

lemma *card-of-set-of-all-possible-states*:
  fixes X :: *'a set*
  assumes *finite X*
  shows $\mathit{card}\ \{(s :: \mathit{'a\ state}).\ \mathit{fmdom'}\ s = X\} = 2\ \hat{}\ (\mathit{card}\ X)$

⟨*proof*⟩

3.1.2 State Lists and State Sets

fun *state-list* **where**
state-list s [] = [*s*]
| *state-list s* (*a* # *as*) = *s* # *state-list* (*state-succ s a*) *as*

lemma *empty-state-list-lemma*:
fixes *as s*
shows ¬([] = *state-list s as*)
⟨*proof*⟩

lemma *state-list-length-non-zero*:
fixes *as s*
shows ¬(*0* = *length* (*state-list s as*))
⟨*proof*⟩

lemma *state-list-length-lemma*:
fixes *as s*
shows *length as* = *length* (*state-list s as*) − *1*
⟨*proof*⟩

lemma *state-list-length-lemma-2*:
fixes *as s*
shows (*length* (*state-list s as*)) = (*length as* + *1*)
⟨*proof*⟩
fun *state-set* **where**
state-set [] = {}
| *state-set* (*s* # *ss*) = *insert* [*s*] (*Cons s* ‘ (*state-set ss*))

lemma *state-set-thm*:
fixes *s1*
shows *s1* ∈ *state-set s2* ⟷ *prefix s1 s2* ∧ *s1* ≠ []
⟨*proof*⟩

lemma *state-set-finite*:
fixes *X*
shows *finite* (*state-set X*)
⟨*proof*⟩

lemma *LENGTH-state-set*:

fixes *X e*
assumes *e* ∈ *state-set X*
shows *length e* ≤ *length X*
⟨*proof*⟩

lemma *lemma-temp*:
fixes *x s as h*
assumes *x* ∈ *state-set* (*state-list s as*)
shows *length* (*h* # *state-list s as*) > *length x*
⟨*proof*⟩

lemma *NIL-NOTIN-stateset*:
fixes *X*
shows [] ∉ *state-set X*
⟨*proof*⟩
lemma *state-set-card-i*:
fixes *X a*
shows [*a*] ∉ (*Cons a* ‘ *state-set X*)
⟨*proof*⟩
lemma *state-set-card-ii*:
fixes *X a*
shows *card* (*Cons a* ‘ *state-set X*) = *card* (*state-set X*)
⟨*proof*⟩
lemma *state-set-card-iii*:
fixes *X a*
shows *card* (*state-set* (*a* # *X*)) = *1* + *card* (*state-set X*)
⟨*proof*⟩

lemma *state-set-card*:
fixes *X*
shows *card* (*state-set X*) = *length X*
⟨*proof*⟩

3.1.3 Properties of Domain Changes During Plan Execution

lemma *FDOM-state-succ*:
assumes *fmdom′* (*snd a*) ⊆ *fmdom′ s*
shows (*fmdom′* (*state-succ s a*) = *fmdom′ s*)
⟨*proof*⟩

lemma *FDOM-state-succ-subset*:
fmdom′ (*state-succ s a*) ⊆ (*fmdom′ s* ∪ *fmdom′* (*snd a*))
⟨*proof*⟩

lemma *FDOM-eff-subset-FDOM-valid-states*:

fixes *p e s*
assumes (*p*, *e*) ∈ *PROB* (*s* ∈ *valid-states PROB*)
shows (*fmdom′ e* ⊆ *fmdom′ s*)
⟨*proof*⟩

lemma *FDOM-eff-subset-FDOM-valid-states-pair*:
fixes *a s*
assumes *a* ∈ *PROB s* ∈ *valid-states PROB*
shows *fmdom′* (*snd a*) ⊆ *fmdom′ s*
⟨*proof*⟩

lemma *FDOM-pre-subset-FDOM-valid-states*:
fixes *p e s*
assumes (*p*, *e*) ∈ *PROB s* ∈ *valid-states PROB*
shows *fmdom′ p* ⊆ *fmdom′ s*
⟨*proof*⟩

lemma *FDOM-pre-subset-FDOM-valid-states-pair*:
fixes *a s*
assumes *a* ∈ *PROB s* ∈ *valid-states PROB*
shows *fmdom′* (*fst a*) ⊆ *fmdom′ s*
⟨*proof*⟩
lemma *action-dom-subset-valid-states-FDOM*:
fixes *p e s*
assumes (*p*, *e*) ∈ *PROB s* ∈ *valid-states PROB*
shows *action-dom p e* ⊆ *fmdom′ s*
⟨*proof*⟩
lemma *FDOM-eff-subset-prob-dom*:
fixes *p e*
assumes (*p*, *e*) ∈ *PROB*
shows *fmdom′ e* ⊆ *prob-dom PROB*
⟨*proof*⟩

lemma *FDOM-eff-subset-prob-dom-pair*:
fixes *a*
assumes *a* ∈ *PROB*
shows *fmdom′* (*snd a*) ⊆ *prob-dom PROB*
⟨*proof*⟩
lemma *FDOM-pre-subset-prob-dom*:
fixes *p e*
assumes (*p*, *e*) ∈ *PROB*
shows *fmdom′ p* ⊆ *prob-dom PROB*
⟨*proof*⟩

lemma *FDOM-pre-subset-prob-dom-pair*:
fixes *a*
assumes *a* ∈ *PROB*
shows *fmdom′* (*fst a*) ⊆ *prob-dom PROB*
⟨*proof*⟩

3.1.4 Properties of Valid Plans

lemma *valid-plan-valid-head*:
assumes (*h* # *as* ∈ *valid-plans PROB*)
shows *h* ∈ *PROB*
⟨*proof*⟩

lemma *valid-plan-valid-tail*:
assumes (*h* # *as* ∈ *valid-plans PROB*)
shows (*as* ∈ *valid-plans PROB*)
⟨*proof*⟩
lemma *valid-plan-pre-subset-prob-dom-pair*:
assumes *as* ∈ *valid-plans PROB*
shows (∀ *a*. *ListMem a as* ⟶ *fmdom′* (*fst a*) ⊆ (*prob-dom PROB*))
⟨*proof*⟩

lemma *valid-append-valid-suff*:
assumes *as1* @ *as2* ∈ (*valid-plans PROB*)
shows *as2* ∈ (*valid-plans PROB*)
⟨*proof*⟩

lemma *valid-append-valid-pref*:
assumes *as1* @ *as2* ∈ (*valid-plans PROB*)
shows *as1* ∈ (*valid-plans PROB*)
⟨*proof*⟩

lemma *valid-pref-suff-valid-append*:
assumes *as1* ∈ (*valid-plans PROB*) *as2* ∈ (*valid-plans PROB*)
shows (*as1* @ *as2*) ∈ (*valid-plans PROB*)
⟨*proof*⟩
lemma *MEM-statelist-FDOM*:
fixes *PROB h as s0*
assumes *s0* ∈ (*valid-states PROB*) *as* ∈ (*valid-plans PROB*) *ListMem h* (*state-list s0 as*)
shows (*fmdom′ h* = *fmdom′ s0*)
⟨*proof*⟩
lemma *MEM-statelist-valid-state*:
fixes *PROB h as s0*
assumes *s0* ∈ *valid-states PROB as* ∈ *valid-plans PROB ListMem h* (*state-list*

s0 as)
  shows (*h* ∈ *valid-states PROB*)
  ⟨*proof*⟩
lemma *lemma-1-i*:
  fixes *s a PROB*
  assumes *s* ∈ *valid-states PROB a* ∈ *PROB*
  shows *state-succ s a* ∈ *valid-states PROB*
  ⟨*proof*⟩
lemma *lemma-1-ii*:
  last ' ((#) *s* ' *state-set* (*state-list* (*state-succ s a*) *as*))
  = *last* ' *state-set* (*state-list* (*state-succ s a*) *as*)
  ⟨*proof*⟩

lemma *lemma-1*:
  fixes *as* :: ((′*a*, ′*b*) *fmap* × (′*a*, ′*b*) *fmap*) *list* **and** *PPROB*
  assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
  shows ((*last* ' (*state-set* (*state-list s as*))) ⊆ *valid-states PROB*)
  ⟨*proof*⟩
lemma *len-in-state-set-le-max-len*:
  fixes *as x PROB*
  assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*) ¬(*as* = [])
    (*x* ∈ *state-set* (*state-list s as*))
  shows (*length x* ≤ (*Suc* (*length as*)))
  ⟨*proof*⟩

lemma *card-state-set-cons*:
  fixes *as s h*
  shows
    (*card* (*state-set* (*state-list s* (*h* # *as*)))
    = *Suc* (*card* (*state-set* (*state-list* (*state-succ s h*) *as*))))

  ⟨*proof*⟩

lemma *card-state-set*:
  fixes *as s*
  shows (*Suc* (*length as*)) = *card* (*state-set* (*state-list s as*))
  ⟨*proof*⟩

lemma *neq-mems-state-set-neq-len*:
  fixes *as x y s*
  assumes *x* ∈ *state-set* (*state-list s as*) (*y* ∈ *state-set* (*state-list s as*)) ¬(*x* = *y*)
  shows ¬(*length x* = *length y*)
⟨*proof*⟩
definition *inj* :: (′*a* ⇒ ′*b*) ⇒ ′*a set* ⇒ ′*b set* ⇒ *bool* **where**
  inj f A B ≡ (∀ *x* ∈ *A*. *f x* ∈ *B*) ∧ *inj-on f A*

— NOTE added lemma; refactored from ‘not_eq_last_diff_paths‘.
lemma *not-eq-last-diff-paths-i*:
fixes *s as PROB*
assumes *s* ∈ *valid-states PROB as* ∈ *valid-plans PROB x* ∈ *state-set* (*state-list s as*)
shows *last x* ∈ *valid-states PROB*
⟨*proof*⟩

lemma *not-eq-last-diff-paths-ii*:
assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
¬(*inj* (*last*) (*state-set* (*state-list s as*)) (*valid-states PROB*))
shows ∃ *l1*. ∃ *l2*.
l1 ∈ *state-set* (*state-list s as*)
∧ *l2* ∈ *state-set* (*state-list s as*)
∧ *last l1* = *last l2*
∧ *l1* ≠ *l2*

⟨*proof*⟩

lemma *not-eq-last-diff-paths*:
fixes *as PROB s*
assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
¬(*inj* (*last*) (*state-set* (*state-list s as*)) (*valid-states PROB*))
shows (∃ *slist-1 slist-2*.
(*slist-1* ∈ *state-set* (*state-list s as*))
∧ (*slist-2* ∈ *state-set* (*state-list s as*))
∧ ((*last slist-1*) = (*last slist-2*))
∧ ¬(*length slist-1* = *length slist-2*))

⟨*proof*⟩

lemma *nempty-sl-in-state-set*:
fixes *sl*
assumes *sl* ≠ []
shows *sl* ∈ *state-set sl*
⟨*proof*⟩

lemma *empty-list-nin-state-set*:
fixes *h slist as*
assumes (*h* # *slist*) ∈ *state-set* (*state-list s as*)
shows (*h* = *s*)
⟨*proof*⟩

lemma *cons-in-state-set-2*:
fixes *s slist h t*

assumes (*slist* ≠ []) ((*s* # *slist*) ∈ *state-set* (*state-list s* (*h* # *t*)))
shows (*slist* ∈ *state-set* (*state-list* (*state-succ s h*) *t*))
⟨*proof*⟩
lemma *valid-action-valid-succ*:
assumes *h* ∈ *PROB s* ∈ *valid-states PROB*
shows (*state-succ s h*) ∈ *valid-states PROB*
⟨*proof*⟩

lemma *in-state-set-imp-eq-exec-prefix*:
fixes *slist as PROB s*
assumes (*as* ≠ []) (*slist* ≠ []) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*slist* ∈ *state-set* (*state-list s as*))
shows
(∃ *as'*. (*prefix as' as*) ∧ (*exec-plan s as'* = *last slist*) ∧ (*length slist* = *Suc* (*length as'*)))
⟨*proof*⟩

lemma *eq-last-state-imp-append-nempty-as*:
fixes *as PROB slist-1 slist-2*
assumes (*as* ≠ []) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*) (*slist-1* ≠ [])
(*slist-2* ≠ []) (*slist-1* ∈ *state-set* (*state-list s as*))
(*slist-2* ∈ *state-set* (*state-list s as*)) ¬(*length slist-1* = *length slist-2*)
(*last slist-1* = *last slist-2*)
shows (∃ *as1 as2 as3*.
(*as1* @ *as2* @ *as3* = *as*)
∧ (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*)
∧ ¬(*as2* = [])
)
⟨*proof*⟩

lemma *FINITE-prob-dom*:
assumes *finite PROB*
shows *finite* (*prob-dom PROB*)
⟨*proof*⟩

lemma *CARD-valid-states*:
assumes *finite* (*PROB* :: *'a problem*)
shows (*card* (*valid-states PROB* :: *'a state set*) = *2* ^ *card* (*prob-dom PROB*))
⟨*proof*⟩
lemma *FINITE-valid-states*:
fixes *PROB* :: *'a problem*
shows *finite PROB* ⟹ *finite* ((*valid-states PROB*) :: *'a state set*)
⟨*proof*⟩
lemma *lemma-2*:

fixes *PROB* :: *'a problem* **and** *as* :: (*'a action*) *list* **and** *s* :: *'a state*
assumes *finite PROB* *s* ∈ (*valid-states PROB*) (*as* ∈ *valid-plans PROB*)
((*length as*) > (*2* ^ (*card* (*fmdom' s*)) − *1*))
shows (∃ *as1 as2 as3*.
(*as1* @ *as2* @ *as3* = *as*)
∧ (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*)
∧ ¬(*as2* = [])
)
⟨*proof*⟩

lemma *lemma-2-prob-dom*:
fixes *PROB* **and** *as* :: (*'a action*) *list* **and** *s* :: *'a state*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*length as* > (*2* ^ (*card* (*prob-dom PROB*))) − *1*)
shows (∃ *as1 as2 as3*.
(*as1* @ *as2* @ *as3* = *as*)
∧ (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*)
∧ ¬(*as2* = [])
)
⟨*proof*⟩
lemma *lemma-3*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*length as* > (*2* ^ (*card* (*prob-dom PROB*)) − *1*))
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*length as'* < *length as*)
∧ (*subseq as' as*)
)
⟨*proof*⟩
lemma *sublist-valid-is-valid*:
fixes *as' as PROB*
assumes (*as* ∈ *valid-plans PROB*) (*subseq as' as*)
shows *as'* ∈ *valid-plans PROB*
⟨*proof*⟩
theorem *main-lemma*:
fixes *PROB* :: *'a problem* **and** *as s*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ (*2* ^ (*card* (*prob-dom PROB*))) − *1*)
)
⟨*proof*⟩

3.2 Reachable States

definition *reachable-s* **where**

reachable-s PROB s ≡ {*exec-plan s as* | *as*. *as* ∈ *valid-plans PROB*}

— NOTE types for 's' and 'PROB' had to be fixed (type mismatch in goal).
lemma *valid-as-valid-exec*:
fixes *as* **and** *s* :: *'a state* **and** *PROB* :: *'a problem*
assumes (*as* ∈ *valid-plans PROB*) (*s* ∈ *valid-states PROB*)
shows (*exec-plan s as* ∈ *valid-states PROB*)
⟨*proof*⟩

lemma *exec-plan-fdom-subset*:
fixes *as s PROB*
assumes (*as* ∈ *valid-plans PROB*)
shows (*fmdom'* (*exec-plan s as*) ⊆ (*fmdom' s* ∪ *prob-dom PROB*))
⟨*proof*⟩
lemma *reachable-s-finite-thm-1-a*:
fixes *s* **and** *PROB* :: *'a problem*
assumes (*s* :: *'a state*) ∈ *valid-states PROB*
shows (∀ *l*∈*reachable-s PROB s*. *l*∈*valid-states PROB*)
⟨*proof*⟩

lemma *reachable-s-finite-thm-1*:
assumes ((*s* :: *'a state*) ∈ *valid-states PROB*)
shows (*reachable-s PROB s* ⊆ *valid-states PROB*)
⟨*proof*⟩
lemma *reachable-s-finite-thm*:
fixes *s* :: *'a state*
assumes *finite* (*PROB* :: *'a problem*) (*s* ∈ *valid-states PROB*)
shows *finite* (*reachable-s PROB s*)
⟨*proof*⟩

lemma *empty-plan-is-valid*: [] ∈ (*valid-plans PROB*)
⟨*proof*⟩

lemma *valid-head-and-tail-valid-plan*:
assumes (*h* ∈ *PROB*) (*as* ∈ *valid-plans PROB*)
shows ((*h* # *as*) ∈ *valid-plans PROB*)
⟨*proof*⟩
lemma *lemma-1-reachability-s-i*:
fixes *PROB s*
assumes *s* ∈ *valid-states PROB*
shows *s* ∈ *reachable-s PROB s*
⟨*proof*⟩
lemma *lemma-1-reachability-s*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as*
assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)

shows ((*last* ‘ *state-set* (*state-list s as*)) ⊆ (*reachable-s PROB s*))
⟨*proof*⟩
lemma *not-eq-last-diff-paths-reachability-s*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as*
assumes *s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*
¬(*inj last* (*state-set* (*state-list s as*)) (*reachable-s PROB s*))
shows (∃ *slist-1 slist-2*.
slist-1 ∈ *state-set* (*state-list s as*)
∧ *slist-2* ∈ *state-set* (*state-list s as*)
∧ (*last slist-1* = *last slist-2*)
∧ ¬(*length slist-1* = *length slist-2*)
)
⟨*proof*⟩
lemma *lemma-2-reachability-s-i*:
fixes *f* :: *'a* ⇒ *'b* **and** *s t*
assumes *finite t card t* < *card s*
shows ¬(*inj f s t*)
⟨*proof*⟩

lemma *lemma-2-reachability-s*:
fixes *PROB* :: *'a problem* **and** *as s*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*length as* > *card* (*reachable-s PROB s*) − *1*)
shows (∃ *as1 as2 as3*.
(*as1* @ *as2* @ *as3* = *as*) ∧ (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*) ∧ ¬(*as2* = []))
⟨*proof*⟩

lemma *lemma-3-reachability-s*:
fixes *as* **and** *PROB* :: *'a problem* **and** *s*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*length as* > (*card* (*reachable-s PROB s*) − *1*))
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*length as'* < *length as*)
∧ (*subseq as' as*)
)
⟨*proof*⟩
lemma *main-lemma-reachability-s*:
fixes *PROB* :: *'a problem* **and** *as* **and** *s* :: *'a state*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*) ∧ *subseq as' as*
∧ (*length as'* ≤ (*card* (*reachable-s PROB s*) − *1*)))
⟨*proof*⟩

lemma *reachable-s-non-empty*: ¬(*reachable-s PROB s* = {})

⟨*proof*⟩

lemma *card-reachable-s-non-zero*:
fixes *s*
assumes *finite* (*PROB* :: *'a problem*) ($s \in$ *valid-states PROB*)
shows ($0 <$ *card* (*reachable-s PROB s*))
⟨*proof*⟩

lemma *exec-fdom-empty-prob*:
fixes *s*
assumes (*prob-dom PROB* = {}) ($s \in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
shows (*exec-plan s as* = *fmempty*)
⟨*proof*⟩
lemma *reachable-s-empty-prob*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state*
assumes (*prob-dom PROB* = {}) ($s \in$ *valid-states PROB*)
shows ((*reachable-s PROB s*) $\subseteq$ {*fmempty*})
⟨*proof*⟩
lemma *sublist-valid-plan--alt*:
assumes (*as1* $\in$ *valid-plans PROB*) (*subseq as2 as1*)
shows (*as2* $\in$ *valid-plans PROB*)
⟨*proof*⟩

lemma *fmsubset-eq*:
assumes $s1 \subseteq_f s2$
shows ($\forall$ *a*. *a* |∈| *fmdom s1* $\longrightarrow$ *fmlookup s1 a* = *fmlookup s2 a*)
⟨*proof*⟩
lemma *submap-imp-state-succ-submap-a*:
assumes $s1 \subseteq_f s2$ $s2 \subseteq_f s3$
shows $s1 \subseteq_f s3$
⟨*proof*⟩
lemma *submap-imp-state-succ-submap-b*:
assumes $s1 \subseteq_f s2$
shows ($s0$ ++ $s1$) $\subseteq_f$ ($s0$ ++ $s2$)
⟨*proof*⟩
lemma *submap-imp-state-succ-submap*:
fixes *a* :: *'a action* **and** *s1 s2*
assumes (*fst a* $\subseteq_f$ *s1*) ($s1 \subseteq_f s2$)
shows (*state-succ s1 a* $\subseteq_f$ *state-succ s2 a*)
⟨*proof*⟩
lemma *pred-dom-subset-succ-submap*:
fixes *a* :: *'a action* **and** *s1 s2* :: *'a state*
assumes (*fmdom'* (*fst a*) $\subseteq$ *fmdom' s1*) ($s1 \subseteq_f s2$)
shows (*state-succ s1 a* $\subseteq_f$ *state-succ s2 a*)
⟨*proof*⟩

lemma *valid-as-submap-init-submap-exec-i*:
  fixes *s a*
  shows *fmdom′ s* $\subseteq$ *fmdom′* (*state-succ s a*)
⟨*proof*⟩
lemma *valid-as-submap-init-submap-exec*:
  fixes *s1 s2* :: *′a state*
  assumes (*s1* $\subseteq_f$ *s2*) ($\forall$ *a*. *ListMem a as* $\longrightarrow$ (*fmdom′* (*fst a*) $\subseteq$ *fmdom′ s1*))
  shows (*exec-plan s1 as* $\subseteq_f$ *exec-plan s2 as*)
  ⟨*proof*⟩

lemma *valid-plan-mems*:
  assumes (*as* $\in$ *valid-plans PROB*) (*ListMem a as*)
  shows *a* $\in$ *PROB*
  ⟨*proof*⟩
lemma *valid-states-nempty*:
  fixes *PROB* :: ((*′a*, *′b*) *fmap* $\times$ (*′a*, *′b*) *fmap*) *set*
  assumes *finite PROB*
  shows $\exists$ *s*. *s* $\in$ (*valid-states PROB*)
  ⟨*proof*⟩

lemma *empty-prob-dom-single-val-state*:
  assumes (*prob-dom PROB* = {})
  shows ($\exists$ *s*. *valid-states PROB* = {*s*})
⟨*proof*⟩

lemma *empty-prob-dom-imp-empty-plan-always-good*:
  fixes *PROB s*
  assumes (*prob-dom PROB* = {}) (*s* $\in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
  shows (*exec-plan s* [] = *exec-plan s as*)
  ⟨*proof*⟩

lemma *empty-prob-dom*:
  fixes *PROB*
  assumes (*prob-dom PROB* = {})
  shows (*PROB* = {(*fmempty*, *fmempty*)} $\vee$ *PROB* = {})
  ⟨*proof*⟩

lemma *empty-prob-dom-finite*:
  fixes *PROB* :: *′a problem*
  assumes *prob-dom PROB* = {}
  shows *finite PROB*
⟨*proof*⟩
lemma *disj-imp-eq-proj-exec*:

fixes a :: $('a, 'b)$ *fmap* $\times$ $('a, 'b)$ *fmap* **and** *vs* s
assumes (*fmdom'* (*snd a*) $\cap$ *vs*) $=$ {}
shows (*fmrestrict-set vs s* $=$ *fmrestrict-set vs* (*state-succ s a*))
⟨*proof*⟩

lemma *no-change-vs-eff-submap*:
fixes *a vs s*
assumes (*fmrestrict-set vs s* $=$ *fmrestrict-set vs* (*state-succ s a*)) (*fst a* $\subseteq_f$ *s*)
shows (*fmrestrict-set vs* (*snd a*) $\subseteq_f$ (*fmrestrict-set vs s*))
⟨*proof*⟩
lemma *sat-precond-as-proj-3*:
fixes s **and** a :: $('a, 'b)$ *fmap* $\times$ $('a, 'b)$ *fmap* **and** *vs*
assumes (*fmdom'* (*fmrestrict-set vs* (*snd a*)) $=$ {})
shows ((*fmrestrict-set vs* (*state-succ s a*)) $=$ (*fmrestrict-set vs s*))
⟨*proof*⟩
lemma *proj-eq-proj-exec-eq*:
fixes s s' *vs* **and** a :: $('a, 'b)$ *fmap* $\times$ $('a, 'b)$ *fmap* **and** a'
assumes ((*fmrestrict-set vs s*) $=$ (*fmrestrict-set vs s'*)) ((*fst a* $\subseteq_f$ *s*) $=$ (*fst a'* $\subseteq_f$ *s'*))
(*fmrestrict-set vs* (*snd a*) $=$ *fmrestrict-set vs* (*snd a'*))
shows (*fmrestrict-set vs* (*state-succ s a*) $=$ *fmrestrict-set vs* (*state-succ s' a'*))
⟨*proof*⟩

lemma *empty-eff-exec-eq*:
fixes *s a*
assumes (*fmdom'* (*snd a*) $=$ {})
shows (*state-succ s a* $=$ *s*)
⟨*proof*⟩

lemma *exec-as-proj-valid-2*:
fixes *a*
assumes $a \in PROB$
shows (*action-dom* (*fst a*) (*snd a*) $\subseteq$ *prob-dom PROB*)
⟨*proof*⟩

lemma *valid-filter-valid-as*:
assumes (*as* $\in$ *valid-plans PROB*)
shows (*filter P as* $\in$ *valid-plans PROB*)
⟨*proof*⟩

lemma *sublist-valid-plan*:
assumes (*subseq as' as*) (*as* $\in$ *valid-plans PROB*)
shows (*as'* $\in$ *valid-plans PROB*)
⟨*proof*⟩

lemma *prob-subset-dom-subset*:
assumes $PROB1 \subseteq PROB2$
shows (*prob-dom PROB1* $\subseteq$ *prob-dom PROB2*)
⟨*proof*⟩

lemma *state-succ-valid-act-disjoint*:
assumes ($a \in PROB$) ($vs \cap$ (*prob-dom PROB*) $= \{\}$)
shows (*fmrestrict-set vs* (*state-succ s a*) = *fmrestrict-set vs s*)
⟨*proof*⟩

lemma *exec-valid-as-disjoint*:
fixes *s*
assumes ($vs \cap$ (*prob-dom PROB*) $= \{\}$) (*as* $\in$ *valid-plans PROB*)
shows (*fmrestrict-set vs* (*exec-plan s as*) = *fmrestrict-set vs s*)
⟨*proof*⟩

definition *state-successors* **where**
state-successors PROB s $\equiv$ ((*state-succ s* ‘ *PROB*) $- \{s\}$)

3.3 State Spaces

definition *stateSpace* **where**
stateSpace ss vs $\equiv$ ($\forall s.\ s \in ss \longrightarrow$ (*fmdom′ s* = *vs*))

lemma *EQ-SS-DOM*:
assumes $\neg$(*ss* = {}) (*stateSpace ss vs1*) (*stateSpace ss vs2*)
shows (*vs1* = *vs2*)
⟨*proof*⟩
lemma *FINITE-SS*:
fixes *ss* :: (*′a*, *bool*) *fmap set*
assumes $\neg$(*ss* = {}) (*stateSpace ss domain*)
shows *finite ss*
⟨*proof*⟩

lemma *disjoint-effects-no-effects*:
fixes *s*
assumes ($\forall a$. *ListMem a as* $\longrightarrow$ (*fmdom′* (*fmrestrict-set vs* (*snd a*)) = {}))
shows (*fmrestrict-set vs* (*exec-plan s as*) = (*fmrestrict-set vs s*))
⟨*proof*⟩

3.4 Needed Asses

definition *action-needed-vars* **where**

action-needed-vars a s ≡ {*v*. (*v* ∈ *fmdom′ s*) ∧ (*v* ∈ *fmdom′* (*fst a*))
∧ (*fmlookup* (*fst a*) *v* = *fmlookup s v*)}

— NOTE name shortened to 'action_needed_asses'.

definition *action-needed-asses* **where**
action-needed-asses a s ≡ *fmrestrict-set* (*action-needed-vars a s*) *s*

— NOTE type for 'a' had to be fixed (type mismatch in goal).

lemma *act-needed-asses-submap-succ-submap*:
fixes *a s1 s2*
assumes (*action-needed-asses a s2* $\subseteq_f$ *action-needed-asses a s1*) (*s1* $\subseteq_f$ *s2*)
shows (*state-succ s1 a* $\subseteq_f$ *state-succ s2 a*)
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-i*:
fixes *a s*
assumes *v* ∈ *fmdom′* (*action-needed-asses a s*)
shows
fmlookup (*action-needed-asses a s*) *v* = *fmlookup s v*
∧ *fmlookup* (*action-needed-asses a s*) *v* = *fmlookup* (*fst a*) *v*
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-ii*:
fixes *f g v*
assumes *v* ∈ *fmdom′ f f* $\subseteq_f$ *g*
shows *fmlookup f v* = *fmlookup g v*
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-iii*:
fixes *f g v*
shows
fmdom′ (*action-needed-asses a s*)
= {*v* ∈ *fmdom′ s*. *v* ∈ *fmdom′* (*fst a*) ∧ *fmlookup* (*fst a*) *v* = *fmlookup s v*}
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-iv*:
fixes *f a v*
assumes *v* ∈ *fmdom′* (*action-needed-asses a s*)
shows
fmlookup (*action-needed-asses a s*) *v* = *fmlookup s v*
∧ *fmlookup* (*action-needed-asses a s*) *v* = *fmlookup* (*fst a*) *v*
∧ *fmlookup* (*fst a*) *v* = *fmlookup s v*
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-v*:
fixes *f g v*
assumes *v* ∈ *fmdom′ f f* $\subseteq_f$ *g*
shows *v* ∈ *fmdom′ g*
⟨*proof*⟩

lemma *as-needed-asses-submap-exec-vi*:
fixes *a s1 s2 v*
assumes *v* ∈ *fmdom′* (*action-needed-asses a s1*)
(*action-needed-asses a s1*) $\subseteq_f$ (*action-needed-asses a s2*)
shows

$(fmlookup\ (action\text{-}needed\text{-}asses\ a\ s1)\ v) = fmlookup\ (fst\ a)\ v$
$\wedge\ (fmlookup\ (action\text{-}needed\text{-}asses\ a\ s2)\ v) = fmlookup\ (fst\ a)\ v\ \wedge$
$fmlookup\ s1\ v = fmlookup\ (fst\ a)\ v \wedge fmlookup\ s2\ v = fmlookup\ (fst\ a)\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-vii*:
fixes $f\ g\ v$
assumes $\forall\, v \in fmdom'\ f.\ fmlookup\ f\ v = fmlookup\ g\ v$
shows $f \subseteq_f g$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-viii*:
fixes $f\ g\ v$
assumes $f \subseteq_f g$
shows $\forall\, v \in fmdom'\ f.\ fmlookup\ f\ v = fmlookup\ g\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-viii'*:
fixes $f\ g\ v$
assumes $f \subseteq_f g$
shows $fmdom'\ f \subseteq fmdom'\ g$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-ix*:
fixes $f\ g$
shows $f \subseteq_f g = (\forall\, v \in fmdom'\ f.\ fmlookup\ f\ v = fmlookup\ g\ v)$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-x*:
fixes $f\ a\ v$
assumes $v \in fmdom'\ (action\text{-}needed\text{-}asses\ a\ f)$
shows $v \in fmdom'\ (fst\ a) \wedge v \in fmdom'\ f \wedge fmlookup\ (fst\ a)\ v = fmlookup\ f\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-xi*:
fixes $v\ a\ f\ g$
assumes $v \in fmdom'\ (action\text{-}needed\text{-}asses\ a\ (f\ {+}{+}\ g))\ v \in fmdom'\ f$
shows
$fmlookup\ (action\text{-}needed\text{-}asses\ a\ (f\ {+}{+}\ g))\ v = fmlookup\ f\ v$
$\wedge\ fmlookup\ (action\text{-}needed\text{-}asses\ a\ (f\ {+}{+}\ g))\ v = fmlookup\ (fst\ a)\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-xii*:
fixes $f\ g\ v$
assumes $v \in fmdom'\ f$
shows $fmlookup\ (f\ {+}{+}\ g)\ v = fmlookup\ f\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec-xii'*:
fixes $f\ g\ v$
assumes $v \notin fmdom'\ f\ v \in fmdom'\ g$
shows $fmlookup\ (f\ {+}{+}\ g)\ v = fmlookup\ g\ v$
⟨*proof*⟩
lemma *as-needed-asses-submap-exec*:
fixes $s1\ s2$
assumes $(s1 \subseteq_f s2)$
$(\forall\, a.\ ListMem\ a\ as \longrightarrow (action\text{-}needed\text{-}asses\ a\ s2 \subseteq_f action\text{-}needed\text{-}asses\ a\ s1))$

shows (*exec-plan s1 as* $\subseteq_f$ *exec-plan s2 as*)
⟨*proof*⟩
definition *system-needed-vars* **where**
system-needed-vars PROB s ≡ (⋃{*action-needed-vars a s* | *a*. *a* ∈ *PROB*})

— NOTE name shortened.
definition *system-needed-asses* **where**
system-needed-asses PROB s ≡ (*fmrestrict-set* (*system-needed-vars PROB s*) *s*)

lemma *action-needed-vars-subset-sys-needed-vars-subset*:
assumes (*a* ∈ *PROB*)
shows (*action-needed-vars a s* ⊆ *system-needed-vars PROB s*)
⟨*proof*⟩

lemma *action-needed-asses-submap-sys-needed-asses*:
assumes (*a* ∈ *PROB*)
shows (*action-needed-asses a s* $\subseteq_f$ *system-needed-asses PROB s*)
⟨*proof*⟩

lemma *system-needed-asses-include-action-needed-asses-1*:
assumes (*a* ∈ *PROB*)
shows (*action-needed-vars a* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) = *action-needed-vars a s*)
⟨*proof*⟩
lemma *system-needed-asses-include-action-needed-asses-i*:
fixes *A B f*
assumes *A* ⊆ *B*
shows *fmrestrict-set A* (*fmrestrict-set B f*) = *fmrestrict-set A f*
⟨*proof*⟩

lemma *system-needed-asses-include-action-needed-asses*:
assumes (*a* ∈ *PROB*)
shows (*action-needed-asses a* (*system-needed-asses PROB s*) = *action-needed-asses a s*)
⟨*proof*⟩

lemma *system-needed-asses-submap*:
system-needed-asses PROB s $\subseteq_f$ *s*
⟨*proof*⟩

lemma *as-works-from-system-needed-asses*:
assumes (*as* ∈ *valid-plans PROB*)
shows (*exec-plan* (*system-needed-asses PROB s*) *as* $\subseteq_f$ *exec-plan s as*)

⟨*proof*⟩

end
theory *ActionSeqProcess*
imports *Main HOL−Library.Sublist FactoredSystemLib FactoredSystem FSSublist*
begin

4 Action Sequence Process

This section defines the preconditions satisfied predicate for action sequences and shows relations between the execution of action sequnences and their projections some. The preconditions satisfied predicate (‘sat_precond_as‘) states that in each recursion step, the given state and the next action are compatible, i.e. the actions preconditions are met by the state. This is used as premise to propositions on projections of action sequences to avoid that an invalid unprojected sequence is suddenly valid after projection. [Abdulaziz et al., p.13]

fun *sat-precond-as* **where**
sat-precond-as s [] = *True*
| *sat-precond-as s* (*a* # *as*) = (*fst a* $\subseteq_f$ *s* ∧ *sat-precond-as* (*state-succ s a*) *as*)

— NOTE added lemma.
lemma *sat-precond-as-pair*:
sat-precond-as s ((*p*, *e*) # *as*) = (*p* $\subseteq_f$ *s* ∧ *sat-precond-as* (*state-succ s* (*p*, *e*)) *as*)
⟨*proof*⟩
fun *rem-effectless-act* **where**
rem-effectless-act [] = []
| *rem-effectless-act* (*a* # *as*) = (*if fmdom′* (*snd a*) ≠ {}
then (*a* # *rem-effectless-act as*)
else rem-effectless-act as
)

— NOTE 'fun' because of multiple defining equations.
fun *no-effectless-act* **where**
no-effectless-act [] = *True*
| *no-effectless-act* (*a* # *as*) = ((*fmdom′* (*snd a*) ≠ {}) ∧ *no-effectless-act as*)

lemma *graph-plan-lemma-4*:
fixes *s s′ as vs P*
assumes (∀ *a*. (*ListMem a as* ∧ *P a*) ⟶ ((*fmdom′* (*snd a*) ∩ *vs*) = {})) *sat-precond-as s as*

sat-precond-as s′ (filter (λa. ¬(P a)) as) (fmrestrict-set vs s = fmrestrict-set vs s′)
shows
(*fmrestrict-set vs (exec-plan s as)*
= *fmrestrict-set vs (exec-plan s′ (filter (λ a. ¬(P a)) as)))*

⟨*proof*⟩
fun *rem-condless-act* **where**
rem-condless-act s pfx-a [] = pfx-a
| *rem-condless-act s pfx-a (a # as) = (if fst a* $\subseteq_f$ *exec-plan s pfx-a*
then rem-condless-act s (pfx-a @ [a]) as
else rem-condless-act s pfx-a as
)

lemma *rem-condless-act-pair*:
rem-condless-act s pfx-a ((p, e) # as) = (if p $\subseteq_f$ *exec-plan s pfx-a*
then rem-condless-act s (pfx-a @ [(p,e)]) as
else rem-condless-act s pfx-a as
)

(*rem-condless-act s pfx-a [] = pfx-a*)
⟨*proof*⟩

lemma *exec-remcondless-cons*:
fixes *s h as pfx*
shows
exec-plan s (rem-condless-act s (h # pfx) as)
= *exec-plan (state-succ s h) (rem-condless-act (state-succ s h) pfx as)*

⟨*proof*⟩

lemma *rem-condless-valid-1*:
fixes *as s*
shows (*exec-plan s as = exec-plan s (rem-condless-act s [] as)*)
⟨*proof*⟩

lemma *rem-condless-act-cons*:
fixes *h′ pfx as s*
shows (*rem-condless-act s (h′ # pfx) as) = (h′ # rem-condless-act (state-succ s h′) pfx as)*
⟨*proof*⟩

lemma *rem-condless-act-cons-prefix*:
fixes *h h′ as as′ s*

assumes *prefix* (h' # as') (*rem-condless-act* s [h] *as*)
shows (
(*prefix* as' (*rem-condless-act* (*state-succ* s h) [] *as*))
$\wedge$ $h' = h$
)
⟨*proof*⟩

lemma *rem-condless-valid-2*:
fixes *as s*
shows *sat-precond-as s* (*rem-condless-act s* [] *as*)
⟨*proof*⟩

lemma *rem-condless-valid-3*:
fixes *as s*
shows *length* (*rem-condless-act s* [] *as*) $\leq$ *length as*
⟨*proof*⟩

lemma *rem-condless-valid-4*:
fixes *as A s*
assumes (*set as* $\subseteq$ *A*)
shows (*set* (*rem-condless-act s* [] *as*) $\subseteq$ *A*)
⟨*proof*⟩

lemma *rem-condless-valid-6*:
fixes *as s P*
shows *length* (*filter P* (*rem-condless-act s* [] *as*)) $\leq$ *length* (*filter P as*)
⟨*proof*⟩

lemma *rem-condless-valid-7*:
fixes *s P as as2*
assumes (*list-all P as* $\wedge$ *list-all P as2*)
shows *list-all P* (*rem-condless-act s as2 as*)
⟨*proof*⟩

lemma *rem-condless-valid-8*:
fixes *s as*
shows *subseq* (*rem-condless-act s* [] *as*) *as*
⟨*proof*⟩

lemma *rem-condless-valid-10*:
fixes *PROB as*
assumes *as* $\in$ (*valid-plans PROB*)

shows (*rem-condless-act s* [] *as* ∈ *valid-plans PROB*)
⟨*proof*⟩

lemma *rem-condless-valid*:
fixes *as A s*
assumes (*exec-plan s as* = *exec-plan s* (*rem-condless-act s* [] *as*))
(*sat-precond-as s* (*rem-condless-act s* [] *as*))
(*length* (*rem-condless-act s* [] *as*) ≤ *length as*)
((*set as* ⊆ *A*) ⟶ (*set* (*rem-condless-act s* [] *as*) ⊆ *A*))
shows (∀ *P*. (*length* (*filter P* (*rem-condless-act s* [] *as*)) ≤ *length* (*filter P as*)))
⟨*proof*⟩
lemma *submap-sat-precond-submap*:
fixes *as* :: ′*a action list*
assumes ($s1 \subseteq_f s2$) (*sat-precond-as s1 as*)
shows (*sat-precond-as s2 as*)
⟨*proof*⟩
lemma *submap-init-submap-exec-i*:
fixes *s1 s2*
assumes ($s1 \subseteq_f s2$) (*sat-precond-as s1* (*a* # *as*))
shows *state-succ s1 a* $\subseteq_f$ *state-succ s2 a*
⟨*proof*⟩

lemma *submap-init-submap-exec*:
fixes *s1 s2*
assumes ($s1 \subseteq_f s2$) (*sat-precond-as s1 as*)
shows (*exec-plan s1 as* $\subseteq_f$ *exec-plan s2 as*)
⟨*proof*⟩
lemma *sat-precond-drest-sat-precond*:
fixes *vs s* **and** *as* :: ′*a action list*
assumes *sat-precond-as* (*fmrestrict-set vs s*) *as*
shows (*sat-precond-as s as*)
⟨*proof*⟩
definition *varset-action* **where**
varset-action a varset ≡ (*fmdom*′ (*snd a*) ⊆ *varset*)
for *a* :: ′*a action*

lemma *varset-action-pair*: (*varset-action* (*p*, *e*) *vs*) = (*fmdom*′ *e* ⊆ *vs*)
⟨*proof*⟩

lemma *eq-effect-eq-vset*:
fixes *x y*
assumes (*snd x* = *snd y*)
shows ((λ*a*. *varset-action a vs*) *x* = (λ*a*. *varset-action a vs*) *y*)
⟨*proof*⟩

lemma *rem-effectless-works-1*:
  fixes *s as*
  shows (*exec-plan s as* = *exec-plan s* (*rem-effectless-act as*))
  ⟨*proof*⟩

lemma *rem-effectless-works-2*:
  fixes *as s*
  assumes (*sat-precond-as s as*)
  shows (*sat-precond-as s* (*rem-effectless-act as*))
  ⟨*proof*⟩

lemma *rem-effectless-works-3*:
  fixes *as*
  shows *length* (*rem-effectless-act as*) ≤ *length as*
  ⟨*proof*⟩

lemma *rem-effectless-works-4*:
  fixes *A as*
  assumes (*set as* ⊆ *A*)
  shows (*set* (*rem-effectless-act as*) ⊆ *A*)
  ⟨*proof*⟩

lemma *rem-effectless-works-4'*:
  fixes *A as*
  assumes (*as* ∈ *valid-plans A*)
  shows (*rem-effectless-act as* ∈ *valid-plans A*)
  ⟨*proof*⟩
lemma *rem-effectless-works-5-i*:
  shows *subseq* (*rem-effectless-act as*) *as*
  ⟨*proof*⟩

lemma *rem-effectless-works-5*:
  fixes *P as*
  shows *length* (*filter P* (*rem-effectless-act as*)) ≤ *length* (*filter P as*)
  ⟨*proof*⟩

lemma *rem-effectless-works-6*:
  fixes *as*
  shows *no-effectless-act* (*rem-effectless-act as*)
  ⟨*proof*⟩

lemma *rem-effectless-works-7*:
  fixes *as*

shows *no-effectless-act as* = *list-all* ($\lambda a.$ *fmdom'* (*snd a*) $\neq$ {}) *as*
⟨*proof*⟩

lemma *rem-effectless-works-8*:
fixes *P as*
assumes (*list-all P as*)
shows *list-all P* (*rem-effectless-act as*)
⟨*proof*⟩
lemma *rem-effectless-works-9*:
fixes *as*
shows *subseq* (*rem-effectless-act as*) *as*
⟨*proof*⟩

lemma *rem-effectless-works-10*:
fixes *as P*
assumes (*no-effectless-act as*)
shows (*no-effectless-act* (*filter P as*))
⟨*proof*⟩

lemma *rem-effectless-works-11*:
fixes *as1 as2*
assumes *subseq as1* (*rem-effectless-act as2*)
shows (*subseq as1 as2*)
⟨*proof*⟩

lemma *rem-effectless-works-12*:
fixes *as1 as2*
shows (*no-effectless-act* (*as1* @ *as2*)) = (*no-effectless-act as1* $\wedge$ *no-effectless-act*(*as2*))
⟨*proof*⟩
lemma *rem-effectless-works-13-i*:
fixes *x l*
assumes *ListMem x l list-all P l*
shows *P x*
⟨*proof*⟩

lemma *rem-effectless-works-13*:
fixes *as1 as2*
assumes (*subseq as1 as2*) (*no-effectless-act as2*)
shows (*no-effectless-act as1*)
⟨*proof*⟩

lemma *rem-effectless-works-14*:
fixes *PROB as*
shows *exec-plan s as* = *exec-plan s* (*rem-effectless-act as*)

⟨*proof*⟩

lemma *rem-effectless-works*:
fixes *s A as*
assumes (*exec-plan s as* = *exec-plan s* (*rem-effectless-act as*))
(*sat-precond-as s as* ⟶ *sat-precond-as s* (*rem-effectless-act as*))
(*length* (*rem-effectless-act as*) ≤ *length as*)
((*set as* ⊆ *A*) ⟶ (*set* (*rem-effectless-act as*) ⊆ *A*))
(*no-effectless-act* (*rem-effectless-act as*))
shows (∀ *P*. *length* (*filter P* (*rem-effectless-act as*)) ≤ *length* (*filter P as*))
⟨*proof*⟩
definition *rem-effectless-act-set* **where**
rem-effectless-act-set A ≡ {*a* ∈ *A*. *fmdom'* (*snd a*) ≠ {}}

lemma *rem-effectless-act-subset-rem-effectless-act-set-thm*:
fixes *as A*
assumes (*set as* ⊆ *A*)
shows (*set* (*rem-effectless-act as*) ⊆ *rem-effectless-act-set A*)
⟨*proof*⟩

lemma *rem-effectless-act-set-no-empty-actions-thm*:
fixes *A*
shows *rem-effectless-act-set A* ⊆ {*a*. *fmdom'* (*snd a*) ≠ {}}
⟨*proof*⟩
lemma *rem-condless-valid-9*:
fixes *s as*
assumes *no-effectless-act as*
shows *no-effectless-act* (*rem-condless-act s* [] *as*)
⟨*proof*⟩

lemma *graph-plan-lemma-17*:
fixes *as-1 as-2 as s*
assumes (*as-1* @ *as-2* = *as*) (*sat-precond-as s as*)
shows ((*sat-precond-as s as-1*) ∧ *sat-precond-as* (*exec-plan s as-1*) *as-2*)
⟨*proof*⟩

lemma *nempty-eff-every-nempty-act*:
fixes *as*
assumes (*no-effectless-act as*) (∀ *x*. ¬(*fmdom'* (*snd* (*f x*)) = {}))
shows (*list-all* (λ*a*. ¬(*f a* = (*fmempty*, *fmempty*))) *as*)
⟨*proof*⟩

lemma *empty-replace-proj-dual7*:

fixes *s as as'*
assumes *sat-precond-as s* (*as* @ *as'*)
shows *sat-precond-as* (*exec-plan s as*) *as'*
⟨*proof*⟩

lemma *not-vset-not-disj-eff-prod-dom-diff*:
fixes *PROB a vs*
assumes ($a \in PROB$) ($\neg$*varset-action a vs*)
shows $\neg$((*fmdom'* (*snd a*) $\cap$ ((*prob-dom PROB*) $-$ *vs*)) $=$ {})
⟨*proof*⟩

lemma *vset-disj-dom-eff-diff*:
fixes *PROB a vs*
assumes (*varset-action a vs*)
shows (((*fmdom'* (*snd a*)) $\cap$ (*prob-dom PROB* $-$ *vs*)) $=$ {})
⟨*proof*⟩

lemma *vset-diff-disj-eff-vs*:
fixes *PROB a vs*
assumes (*varset-action a* (*prob-dom PROB* $-$ *vs*))
shows (((*fmdom'* (*snd a*)) $\cap$ *vs*) $=$ {})
⟨*proof*⟩

lemma *vset-nempty-efff-not-disj-eff-vs*:
fixes *PROB a vs*
assumes (*varset-action a vs*) (*fmdom'* (*snd a*) $\neq$ {})
shows $\neg$((*fmdom'* (*snd a*) $\cap$ *vs*)) $=$ {}
⟨*proof*⟩

lemma *vset-disj-eff-diff*:
fixes *s a vs*
assumes (*varset-action a vs*)
shows ((*fmdom'* (*snd a*) $\cap$ (*s* $-$ *vs*)) $=$ {})
⟨*proof*⟩
lemma *list-all-list-mem*:
fixes *P* **and** *l* :: *'a list*
shows *list-all P l* $\longleftrightarrow$ ($\forall$ *e. ListMem e l* $\longrightarrow$ *P e*)
⟨*proof*⟩

lemma *every-vset-imp-drestrict-exec-eq*:
fixes *PROB vs as s*
assumes (*list-all* (λ*a. varset-action a* ((*prob-dom PROB*) $-$ *vs*)) *as*)
shows (*fmrestrict-set vs s* $=$ *fmrestrict-set vs* (*exec-plan s as*))

⟨*proof*⟩

lemma *no-effectless-act-works*:
fixes *as*
assumes (*no-effectless-act as*)
shows (*filter* (*λa*. ¬(*fmdom′* (*snd a*) = {})) *as* = *as*)
⟨*proof*⟩

lemma *varset-act-diff-un-imp-varset-diff*:
fixes *a vs vs′ vs′′*
assumes (*varset-action a* (*vs′′* − (*vs′* ∪ *vs*)))
shows (*varset-action a* (*vs′′* − *vs*))
⟨*proof*⟩

lemma *vset-diff-union-vset-diff*:
fixes *s vs vs′ a*
assumes (*varset-action a* (*s* − (*vs* ∪ *vs′*)))
shows (*varset-action a* (*s* − *vs′*))
⟨*proof*⟩

lemma *valid-filter-vset-dom-idempot*:
fixes *PROB as*
assumes (*as* ∈ *valid-plans PROB*)
shows (*filter* (*λa*. *varset-action a* (*prob-dom PROB*)) *as* = *as*)
⟨*proof*⟩

lemma *n-replace-proj-le-n-as-1*:
fixes *a vs vs′*
assumes (*vs* ⊆ *vs′*) (*varset-action a vs*)
shows (*varset-action a vs′*)
⟨*proof*⟩

lemma *sat-precond-as-pfx*:
fixes *s*
assumes (*sat-precond-as s* (*as* @ *as′*))
shows (*sat-precond-as s as*)
⟨*proof*⟩

end
theory *RelUtils*
imports *Main HOL.Transitive-Closure*
begin

— NOTE added definition.
definition *reflexive* **where**
reflexive $R \equiv \forall x.\ R\ x\ x$

— NOTE translation of 'TC' in relationScript.sml:69.
— TODO can we replace this with something from 'HOL.Transitive_Closure'?
definition *TC* **where**
$TC\ R\ a\ b \equiv (\forall P.\ (\forall x\ y.\ R\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow P\ a\ b)$

— NOTE adapts transitive closure definitions of Isabelle and HOL4.
lemma *TC-equiv-tranclp*: $TC\ R\ a\ b \longleftrightarrow (R^{++}\ a\ b)$
⟨*proof*⟩

lemma *TC-IMP-NOT-TC-CONJ-1*:
fixes $R\ P$ **and** $x\ y$
assumes $\neg(R^{++}\ x\ y)$
shows $\neg((\lambda x\ y.\ R\ x\ y \wedge P\ x\ y)^{++}\ x\ y)$
⟨*proof*⟩

lemma *TC-IMP-NOT-TC-CONJ*:
fixes $R\ R'\ P\ x\ y$
assumes $\forall x\ y.\ P\ x\ y \longrightarrow R'\ x\ y \longrightarrow R\ x\ y$ $\neg R^{++}\ x\ y$
shows $\neg(\lambda x\ y.\ R'\ x\ y \wedge P\ x\ y)^{++}\ x\ y$
⟨*proof*⟩
lemma *TC-INDUCT*:
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** P
assumes $(\forall x\ y.\ R\ x\ y \longrightarrow P\ x\ y)$ $(\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z)$
shows $\forall u\ v.\ (TC\ R)\ u\ v \longrightarrow P\ u\ v$
⟨*proof*⟩

lemma *REFL-IMP-3-CONJ-1*:
fixes $R\ P\ x\ y$
assumes $((\lambda x\ y.\ R\ x\ y \wedge P\ x\ y)^{++}\ x\ y)$
shows $R^{++}\ x\ y$
⟨*proof*⟩

lemma *REFL-IMP-3-CONJ*:
fixes R'
assumes *reflexive* R'
shows $(\forall P\ x\ y.$
$(R'^{++}\ x\ y) \longrightarrow (\ ((\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ y) \vee (\exists z.\ \neg P\ z \wedge R'^{++}\ x\ z \wedge R'^{++}\ z\ y)))$
⟨*proof*⟩

lemma *REFL-TC-CONJ*:
fixes $R\ R' :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $P\ x\ y$
assumes *reflexive* R' $\forall x\ y.\ P\ x \wedge P\ y \longrightarrow (R'\ x\ y \longrightarrow R\ x\ y)$ $\neg(R^{++}\ x\ y)$

shows $(\neg(R'^{++}\ x\ y) \vee (\exists\, z.\ \neg P\ z \wedge (R')^{++}\ x\ z \wedge (R')^{++}\ z\ y))$
⟨*proof*⟩
lemma *TC-CASES1-NEQ*:
fixes *R x z*
assumes $R^{++}\ x\ z$
shows $R\ x\ z \vee (\exists\, y :: 'a.\ \neg(x = y) \wedge \neg(y = z) \wedge R\ x\ y \wedge R^{++}\ y\ z)$
⟨*proof*⟩
end
theory *Dependency*
imports *Main HOL−Library.Finite-Map FactoredSystem ActionSeqProcess RelUtils*
begin

5 Dependency

State variable dependency analysis may be used to find structure in a factored system and find useful projections, for example on variable sets which are closed under mutual dependency. [Abdulaziz et al., p.13]

In the following the dependency predicate (‘dep‘) is formalized and some dependency related propositions are proven. Dependency between variables ‘v1‘, ‘v2‘ w.r.t to an action set δ is given if one of the following holds: (1) ‘v1‘ and ‘v2‘ are equal (2) an action $(p, e) \in \delta$ exists where $v1 \in \mathcal{D}\ p$ and $v2 \in \mathcal{D}\ e$ (meaning that it is a necessary condition that ‘p v1‘ is given if the action has effect ‘e v2‘). (3) or, an action $(p, e) \in \delta$ exists s.t. $v1\ v2 \in \mathcal{D}\ e$ This notion is extended to sets of variables ‘vs1‘, ‘vs2‘ (‘dep_var_set‘): ‘vs1‘ and ‘vs2‘ are dependent iff ‘vs1‘ and ‘vs2‘ are disjoint and if dependent ‘v1‘, ‘v2‘ exist where $v1 \in vs1$, $v2 \in vs2$. [Abdulaziz et al., Definition 7, p.13][Abdulaziz et al., HOL4 Definition 5, p.14]

5.1 Dependent Variables and Variable Sets

definition *dep* **where**
$dep\ PROB\ v1\ v2 \equiv (\exists\, a.$
$a \in PROB$
$\wedge\ ($
$((v1 \in fmdom'\ (fst\ a)) \wedge (v2 \in fmdom'\ (snd\ a)))$
$\vee\ ((v1 \in fmdom'\ (snd\ a) \wedge v2 \in fmdom'\ (snd\ a)))$
$)$
$)$
$\vee\ (v1 = v2)$

— NOTE name shortened to ’dep_var_set’.
definition *dep-var-set* **where**
$dep\text{-}var\text{-}set\ PROB\ vs1\ vs2 \equiv (disjnt\ vs1\ vs2) \wedge$
$(\exists\ v1\ v2.\ (v1 \in vs1) \wedge (v2 \in vs2) \wedge (dep\ PROB\ v1\ v2)$
$)$

lemma *dep-var-set-self-empty*:
fixes *PROB vs*
assumes *dep-var-set PROB vs vs*
shows ($vs = \{\}$)
⟨*proof*⟩

lemma *DEP-REFL*:
fixes *PROB*
shows *reflexive* ($\lambda v\ v'.\ dep\ PROB\ v\ v'$)
⟨*proof*⟩
lemma *NEQ-DEP-IMP-IN-DOM-i*:
fixes *a v*
assumes $a \in PROB$ $v \in fmdom'$ (*fst a*)
shows $v \in$ *prob-dom PROB*
⟨*proof*⟩
lemma *NEQ-DEP-IMP-IN-DOM-ii*:
fixes *a v*
assumes $a \in PROB$ $v \in fmdom'$ (*snd a*)
shows $v \in$ *prob-dom PROB*
⟨*proof*⟩

lemma *NEQ-DEP-IMP-IN-DOM*:
fixes *PROB* :: $(('a, 'b)\ fmap \times ('a, 'b)\ fmap)\ set$ **and** $v\ v'$
assumes $\neg(v = v')$ (*dep PROB* $v\ v'$)
shows ($v \in$ (*prob-dom PROB*) $\wedge$ $v' \in$ (*prob-dom PROB*))
⟨*proof*⟩

lemma *dep-sos-imp-mem-dep*:
fixes *PROB S vs*
assumes (*dep-var-set PROB* ($\bigcup$ *S*) *vs*)
shows ($\exists\, vs'.\ vs' \in S \wedge$ *dep-var-set PROB* vs' *vs*)
⟨*proof*⟩

lemma *dep-union-imp-or-dep*:
fixes *PROB vs* vs' vs''
assumes (*dep-var-set PROB vs* ($vs' \cup vs''$))
shows (*dep-var-set PROB vs* vs' $\vee$ *dep-var-set PROB vs* vs'')
⟨*proof*⟩
lemma *dep-biunion-imp-or-dep*:
fixes *PROB vs S*
assumes (*dep-var-set PROB vs* ($\bigcup S$))
shows ($\exists\, vs'.\ vs' \in S \wedge$ *dep-var-set PROB vs* vs')
⟨*proof*⟩

5.2 Transitive Closure of Dependent Variables and Variable Sets

definition *dep-tc* **where**
  dep-tc PROB = *TC* (λ*v1′ v2′*. *dep PROB v1′ v2′*)

— NOTE type of ‘PROB‘ had to be fixed for MP on ‘NEQ_DEP_IMP_IN_DOM‘.
lemma *dep-tc-imp-in-dom*:
  fixes *PROB* :: ((′*a*, ′*b*) *fmap* × (′*a*, ′*b*) *fmap*) *set* **and** *v1 v2*
  assumes ¬(*v1* = *v2*) (*dep-tc PROB v1 v2*)
  shows (*v1* ∈ *prob-dom PROB*)
⟨*proof*⟩

lemma *not-dep-disj-imp-not-dep*:
  fixes *PROB vs-1 vs-2 vs-3*
  assumes ((*vs-1* ∩ *vs-2*) = {}) (*vs-3* ⊆ *vs-2*) ¬(*dep-var-set PROB vs-1 vs-2*)
  shows ¬(*dep-var-set PROB vs-1 vs-3*)
  ⟨*proof*⟩

lemma *dep-slist-imp-mem-dep*:
  fixes *PROB vs lvs*
  assumes (*dep-var-set PROB* (⋃ (*set lvs*)) *vs*)
  shows (∃ *vs′*. *ListMem vs′ lvs* ∧ *dep-var-set PROB vs′ vs*)
⟨*proof*⟩

lemma *n-bigunion-le-sum-3*:
  fixes *PROB vs svs*
  assumes (∀ *vs′*. *vs′* ∈ *svs* ⟶ ¬(*dep-var-set PROB vs′ vs*))
  shows ¬(*dep-var-set PROB* (⋃*svs*) *vs*)
⟨*proof*⟩

lemma *disj-not-dep-vset-union-imp-or*:
  fixes *PROB a vs vs′*
  assumes (*a* ∈ *PROB*) (*disjnt vs vs′*)
    (¬(*dep-var-set PROB vs′ vs*) ∨ ¬(*dep-var-set PROB vs vs′*))
    (*varset-action a* (*vs* ∪ *vs′*))
  shows (*varset-action a vs* ∨ *varset-action a vs′*)
  ⟨*proof*⟩

end
theory *Invariants*
  imports *Main FactoredSystem*
begin

definition *fdom* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set$ **where**
fdom f $\equiv \{x.\ \exists\, y.\ f\ x = y\}$

— TODO function domain for total function in Isabelle/HOL?
— TODO why is fm total? Shouldn't it be partial and thus needing the the premise ‘fm x = Some True‘ instead of just ‘fm x‘?
definition *invariant* :: $('a \Rightarrow bool) \Rightarrow bool$ **where**
invariant fm $\equiv (\forall\, x.\ (x \in fdom\ fm \wedge fm\ x) \longrightarrow False) \wedge (\exists\, x.\ x \in fdom\ fm \wedge fm\ x)$

end
theory *SetUtils*
imports *Main*
begin

— TODO use Inf instead of Min where necessary.

— TODO can be replaced by *card-Un-disjoint* ($[\![finite\ A;\ finite\ B;\ A \cap B = \{\}]\!] \Longrightarrow card\ (A \cup B) = card\ A + card\ B$) ?
lemma *card-union'*: $(finite\ s) \wedge (finite\ t) \wedge (disjnt\ s\ t) \Longrightarrow (card\ (s \cup t) = card\ s + card\ t)$
⟨*proof*⟩

lemma *CARD-INJ-IMAGE-2*:
fixes *f s*
assumes $finite\ s\ (\forall\, x\ y.\ ((x \in s) \wedge (y \in s)) \longrightarrow ((f\ x = f\ y) \longleftrightarrow (x = y)))$
shows $(card\ (f\ `\ s) = card\ s)$
⟨*proof*⟩

lemma *scc-main-lemma-x*: $\bigwedge s\ t\ x.\ (x \in s) \wedge \neg(x \in t) \Longrightarrow \neg(s = t)$
⟨*proof*⟩

lemma *neq-funs-neq-images*:
fixes *s*
assumes $\forall\, x.\ x \in s \longrightarrow (\forall\, y.\ y \in s \longrightarrow f1\ x \neq f2\ y)\ \exists\, x.\ x \in s$
shows $f1\ `\ s \neq f2\ `\ s$
⟨*proof*⟩

5.3 Sets of Numbers

lemma *mems-le-finite-i*:
fixes $s :: nat\ set$ **and** $k :: nat$
shows $(\forall\ x.\ x \in s \longrightarrow x \leq k) \Longrightarrow finite\ s$
⟨*proof*⟩
lemma *mems-le-finite*:
fixes $s :: nat\ set$ **and** $k :: nat$
shows $\bigwedge(s :: nat\ set)\ k.\ (\forall\ x.\ x \in s \longrightarrow x \leq k) \Longrightarrow finite\ s$
⟨*proof*⟩
lemma *mem-le-imp-MIN-le*:

fixes s :: *nat set* **and** k :: *nat*
assumes $\exists x. (x \in s) \wedge (x \leq k)$
shows $(\mathit{Inf}\ s \leq k)$
$\langle proof \rangle$
lemma *mem-lt-imp-MIN-lt*:
fixes s :: *nat set* **and** k :: *nat*
assumes $(\exists x. x \in s \wedge x < k)$
shows $(\mathit{Inf}\ s) < k$
$\langle proof \rangle$
lemma *bound-child-parent-neq-mems-state-set-neq-len*:
fixes s **and** k :: *nat*
assumes $(\forall x. x \in s \longrightarrow x < k)$
shows *finite s*
$\langle proof \rangle$

lemma *bound-main-lemma-2*: $\bigwedge(s :: \mathit{nat\ set})\ k.\ (s \neq \{\}) \wedge (\forall x.\ x \in s \longrightarrow x \leq k) \Longrightarrow \mathit{Sup}\ s \leq k$
$\langle proof \rangle$
lemma *bound-child-parent-not-eq-last-diff-paths*: $\bigwedge s\ (k :: \mathit{nat}).$
$(s \neq \{\})$
$\Longrightarrow (\forall x.\ x \in s \longrightarrow x < k)$
$\Longrightarrow \mathit{Sup}\ s < k$

$\langle proof \rangle$

lemma *FINITE-ALL-DISTINCT-LISTS-i*:
fixes P
assumes *finite P*
shows
$\{p.\ \mathit{distinct}\ p \wedge \mathit{set}\ p \subseteq P\}$
$= \{[]\} \cup (\bigcup ((\lambda e.\ \{e \# p0 \mid p0.\ \mathit{distinct}\ p0 \wedge \mathit{set}\ p0 \subseteq (P - \{e\})\})\ `\ P))$
$\langle proof \rangle$

lemma *FINITE-ALL-DISTINCT-LISTS*:
fixes P
assumes *finite P*
shows *finite* $\{p.\ \mathit{distinct}\ p \wedge \mathit{set}\ p \subseteq P\}$
$\langle proof \rangle$

lemma *subset-inter-diff-empty*:
assumes $s \subseteq t$
shows $(s \cap (u - t) = \{\})$
$\langle proof \rangle$

end
theory *TopologicalProps*
imports *Main FactoredSystem ActionSeqProcess SetUtils*
begin

6 Topological Properties

6.1 Basic Definitions and Properties

definition *PLS-charles* **where**
PLS-charles s as PROB ≡ {*length as′* | *as′*.
(*as′* ∈ *valid-plans PROB*) ∧ (*exec-plan s as′* = *exec-plan s as*)}

definition *MPLS-charles* **where**
MPLS-charles PROB ≡ {*Inf* (*PLS-charles* (*fst p*) (*snd p*) *PROB*) | *p*.
((*fst p*) ∈ *valid-states PROB*)
∧ ((*snd p*) ∈ *valid-plans PROB*)
}

— NOTE name shortened to 'problem_plan_bound_charles'.
definition *problem-plan-bound-charles* **where**
problem-plan-bound-charles PROB ≡ *Sup* (*MPLS-charles PROB*)

— NOTE name shortened to 'PLS_state'.
definition *PLS-state-1* **where**
PLS-state-1 s as ≡ *length* ‘ {*as′*. (*exec-plan s as′* = *exec-plan s as*)}

— NOTE name shortened to 'MPLS_stage_1'.
definition *MPLS-stage-1* **where**
MPLS-stage-1 PROB ≡
(λ (*s*, *as*). *Inf* (*PLS-state-1 s as*))
‘ {(*s*, *as*). (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)}

— NOTE name shortened to 'problem_plan_bound_stage_1'.
definition *problem-plan-bound-stage-1* **where**
problem-plan-bound-stage-1 PROB ≡ *Sup* (*MPLS-stage-1 PROB*)
for *PROB* :: *′a problem*

— NOTE name shortened.
definition *PLS* **where**
PLS s as ≡ *length* ‘ {*as′*. (*exec-plan s as′* = *exec-plan s as*) ∧ (*subseq as′ as*)}

— NOTE added lemma.
— NOTE proof finite PLS for use in 'proof in_MPLS_leq_2_pow_n_i'
lemma *finite-PLS*: *finite* (*PLS s as*)
⟨*proof*⟩
definition *MPLS* **where**

MPLS PROB ≡
  (λ (*s*, *as*). *Inf* (*PLS s as*))
  ‘ {(*s*, *as*). (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)}

— NOTE name shortened.
definition *problem-plan-bound* **where**
problem-plan-bound PROB ≡ *Sup* (*MPLS PROB*)

lemma *expanded-problem-plan-bound-thm-1*:
fixes *PROB*
shows
(*problem-plan-bound PROB*) = *Sup* (
  (λ(*s*,*as*). *Inf* (*PLS s as*)) ‘
  {(*s*, *as*). (*s* ∈ (*valid-states PROB*)) ∧ (*as* ∈ *valid-plans PROB*)}
)

⟨*proof*⟩

lemma *expanded-problem-plan-bound-thm*:
fixes *PROB* :: ((′*a*, ′*b*) *fmap* × (′*a*, ′*b*) *fmap*) *set*
shows
problem-plan-bound PROB = *Sup* ({*Inf* (*PLS s as*) | *s as*.
  (*s* ∈ *valid-states PROB*)
  ∧ (*as* ∈ *valid-plans PROB*)
})

⟨*proof*⟩

6.2 Recurrence Diameter

The recurrence diameter—defined as the longest simple path in the digraph modelling the state space—provides a loose upper bound on the system diameter. [Abdulaziz et al., Definition 9, p.15]

fun *valid-path* **where**
valid-path Pi [] = *True*
| *valid-path Pi* [*s*] = (*s* ∈ *valid-states Pi*)
| *valid-path Pi* (*s1* # *s2* # *rest*) = (
  (*s1* ∈ *valid-states Pi*)
  ∧ (∃ *a*. (*a* ∈ *Pi*) ∧ (*exec-plan s1* [*a*] = *s2*))
  ∧ (*valid-path Pi* (*s2* # *rest*))
)

lemma *valid-path-ITP2015*:
(*valid-path Pi* [] ⟷ *True*)
∧ (*valid-path Pi* [*s*] ⟷ (*s* ∈ *valid-states Pi*))

∧ (*valid-path Pi* (*s1* # *s2* # *rest*) ⟷
  (*s1* ∈ *valid-states Pi*)
  ∧ (∃ *a*.
    (*a* ∈ *Pi*)
    ∧ (*exec-plan s1* [*a*] = *s2*)
)
  ∧ (*valid-path Pi* (*s2* # *rest*))
)

⟨*proof*⟩
definition *RD* **where**
RD Pi ≡ (*Sup* {*length p* − *1* | *p*. *valid-path Pi p* ∧ *distinct p*})
for *Pi* :: ′*a problem*

lemma *in-PLS-leq-2-pow-n*:
fixes *PROB* :: ′*a problem* **and** *s* :: ′*a state* **and** *as*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *x*.
  (*x* ∈ *PLS s as*)
  ∧ (*x* ≤ (*2* ^ *card* (*prob-dom PROB*)) − *1*)
)
⟨*proof*⟩

lemma *in-MPLS-leq-2-pow-n*:
fixes *PROB* :: ′*a problem* **and** *x*
assumes *finite PROB* (*x* ∈ *MPLS PROB*)
shows (*x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*)
⟨*proof*⟩

lemma *FINITE-MPLS*:
assumes *finite* (*Pi* :: ′*a problem*)
shows *finite* (*MPLS Pi*)
⟨*proof*⟩
fun *statelist*′ **where**
statelist′ *s* [] = [*s*]
| *statelist*′ *s* (*a* # *as*) = (*s* # *statelist*′ (*state-succ s a*) *as*)

lemma *LENGTH-statelist*′:
fixes *as s*
shows *length* (*statelist*′ *s as*) = (*length as* + *1*)
⟨*proof*⟩

lemma *valid-path-statelist*′:
fixes *as* **and** *s* :: (′*a*, ′*b*) *fmap*

assumes (*as* ∈ *valid-plans Pi*) (*s* ∈ *valid-states Pi*)
shows (*valid-path Pi* (*statelist′ s as*))
⟨*proof*⟩
lemma *statelist′-exec-plan*:
fixes *a s p*
assumes (*statelist′ s as* = *p*)
shows (*exec-plan s as* = *last p*)
⟨*proof*⟩

lemma *statelist′-EQ-NIL*: *statelist′ s as* ≠ []
⟨*proof*⟩
lemma *statelist′-TAKE-i*:
assumes *Suc m* ≤ *length* (*a* # *as*)
shows *m* ≤ *length as*
⟨*proof*⟩

lemma *statelist′-TAKE*:
fixes *as s p*
assumes (*statelist′ s as* = *p*)
shows (∀ *n*. *n* ≤ *length as* ⟶ (*exec-plan s* (*take n as*)) = (*p* ! *n*))
⟨*proof*⟩

lemma *MPLS-nempty*:
fixes *PROB* :: ((′*a*, ′*b*) *fmap* × (′*a*, ′*b*) *fmap*) *set*
assumes *finite PROB*
shows *MPLS PROB* ≠ {}
⟨*proof*⟩

theorem *bound-main-lemma*:
fixes *PROB* :: ′*a problem*
assumes *finite PROB*
shows (*problem-plan-bound PROB* ≤ (*2* ^ (*card* (*prob-dom PROB*))) − *1*)
⟨*proof*⟩
lemma *bound-child-parent-card-state-set-cons*:
fixes *P f*
assumes (∀ (*PROB* :: ′*a problem*) *as* (*s* :: ′*a state*).
(*P PROB*)
∧ (*as* ∈ *valid-plans PROB*)
∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′)
∧ (*subseq as*′ *as*)
∧ (*length as*′ < *f PROB*)
)
)
shows (∀ *PROB s as*.

(*P PROB*)
∧ (*as* ∈ *valid-plans PROB*)
∧ (*s* ∈ (*valid-states PROB*))
⟶ (∃ *x*.
(*x* ∈ *PLS s as*)
∧ (*x* < *f PROB*)
)
)
⟨*proof*⟩
lemma *bound-on-all-plans-bounds-MPLS*:
fixes *P f*
assumes (∀ (*PROB* :: ′*a problem*) *as* (*s* :: ′*a state*).
(*P PROB*)
∧ (*s* ∈ *valid-states PROB*)
∧ (*as* ∈ *valid-plans PROB*)
⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′)
∧ (*subseq as*′ *as*)
∧ (*length as*′ < *f PROB*)
)
)
shows (∀ *PROB x*. *P PROB*
⟶ (*x* ∈ *MPLS*(*PROB*))
⟶ (*x* < *f PROB*)
)
⟨*proof*⟩

lemma *bound-child-parent-card-state-set-cons-finite*:
fixes *P f*
assumes (∀ *PROB as s*.
P PROB ∧ *finite PROB* ∧ *as* ∈ (*valid-plans PROB*) ∧ *s* ∈ (*valid-states PROB*)

⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′)
∧ *subseq as*′ *as*
∧ *length as*′ < *f*(*PROB*)
)
)
shows (∀ *PROB s as*.
P PROB ∧ *finite PROB* ∧ *as* ∈ (*valid-plans PROB*) ∧ (*s* ∈ (*valid-states PROB*))
⟶ (∃ *x*. (*x* ∈ *PLS s as*) ∧ *x* < *f PROB*)
)
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-MPLS-finite*:
fixes *P f*
assumes (∀ *PROB as s*.

$P\ PROB \wedge finite\ PROB \wedge s \in (valid\text{-}states\ PROB) \wedge as \in (valid\text{-}plans\ PROB)$

$\longrightarrow (\exists\, as'.$
$(exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as')$
$\wedge\ subseq\ as'\ as$
$\wedge\ length\ as' < f(PROB)$
)
)
shows $(\forall\, PROB\ x.$
$P\ PROB \wedge finite\ PROB$
$\longrightarrow (x \in MPLS\ PROB)$
$\longrightarrow x < f\ PROB$
)
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-problem-plan-bound*:
fixes $P\ f$
assumes $(\forall\, PROB\ as\ s.$
$(P\ PROB)$
$\wedge\ finite\ PROB$
$\wedge\ (s \in valid\text{-}states\ PROB)$
$\wedge\ (as \in valid\text{-}plans\ PROB)$
$\longrightarrow (\exists\, as'.$
$(exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as')$
$\wedge\ (subseq\ as'\ as)$
$\wedge\ (length\ as' < f\ PROB)$
)
)
shows $(\forall\, PROB.$
$(P\ PROB)$
$\wedge\ finite\ PROB$
$\longrightarrow (problem\text{-}plan\text{-}bound\ PROB < f\ PROB)$
)
⟨*proof*⟩

lemma *bound-child-parent-card-state-set-cons-thesis*:
assumes $finite\ PROB\ (\forall\, as\ s.$
$as \in (valid\text{-}plans\ PROB)$
$\wedge\ s \in (valid\text{-}states\ PROB)$
$\longrightarrow (\exists\, as'.$
$(exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as')$
$\wedge\ subseq\ as'\ as$
$\wedge\ length\ as' < k$
)
$)\ as \in (valid\text{-}plans\ PROB)\ (s \in (valid\text{-}states\ PROB))$
shows $(\exists\, x.\ (x \in PLS\ s\ as) \wedge x < k)$
⟨*proof*⟩

lemma *x-in-MPLS-if*:
  fixes *x PROB*
  assumes $x \in$ *MPLS PROB*
  shows $\exists$ *s as*. $s \in$ *valid-states PROB* $\wedge$ *as* $\in$ *valid-plans PROB* $\wedge$ $x =$ *Inf* (*PLS s as*)
  ⟨*proof*⟩

lemma *bound-on-all-plans-bounds-MPLS-thesis*:
  assumes *finite PROB* ($\forall$ *as s*.
    ($s \in$ *valid-states PROB*)
    $\wedge$ (*as* $\in$ *valid-plans PROB*)
    $\longrightarrow$ ($\exists$ *as'*.
      (*exec-plan s as* = *exec-plan s as'*)
      $\wedge$ (*subseq as' as*)
      $\wedge$ (*length as'* $< k$)
)
) ($x \in$ *MPLS PROB*)
  shows ($x < k$)
⟨*proof*⟩
lemma *bounded-MPLS-contains-supremum*:
  fixes *PROB*
  assumes *finite PROB* ($\exists$ *k*. $\forall$ $x \in$ *MPLS PROB*. $x < k$)
  shows *Sup* (*MPLS PROB*) $\in$ *MPLS PROB*
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-problem-plan-bound-thesis'*:
  assumes *finite PROB* ($\forall$ *as s*.
      $s \in$ (*valid-states PROB*)
      $\wedge$ *as* $\in$ (*valid-plans PROB*)
      $\longrightarrow$ ($\exists$ *as'*.
        (*exec-plan s as* = *exec-plan s as'*)
        $\wedge$ *subseq as' as*
        $\wedge$ *length as'* $< k$
)
)
  shows *problem-plan-bound PROB* $< k$
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-problem-plan-bound-thesis*:
  assumes *finite PROB* ($\forall$ *as s*.
      ($s \in$ *valid-states PROB*)
      $\wedge$ (*as* $\in$ *valid-plans PROB*)
      $\longrightarrow$ ($\exists$ *as'*.
        (*exec-plan s as* = *exec-plan s as'*)
        $\wedge$ (*subseq as' as*)
        $\wedge$ (*length as'* $\leq k$)
)
)

shows (*problem-plan-bound PROB* $\leq$ *k*)
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-problem-plan-bound-*:
fixes *P f PROB*
assumes ($\forall$ *PROB′ as s*.
finite PROB $\wedge$ (*P PROB′*) $\wedge$ (*s* $\in$ *valid-states PROB′*) $\wedge$ (*as* $\in$ *valid-plans PROB′*)
$\longrightarrow$ ($\exists$ *as′*.
(*exec-plan s as* = *exec-plan s as′*)
$\wedge$ (*subseq as′ as*)
$\wedge$ (*length as′* < *f PROB′*)
)
) (*P PROB*) *finite PROB*
shows (*problem-plan-bound PROB* < *f PROB*)
⟨*proof*⟩

lemma *S-VALID-AS-VALID-IMP-MIN-IN-PLS*:
fixes *PROB s as*
assumes (*s* $\in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
shows (*Inf* (*PLS s as*) $\in$ (*MPLS PROB*))
⟨*proof*⟩
lemma *problem-plan-bound-ge-min-pls*:
fixes *PROB* :: *′a problem* **and** *s* :: *′a state* **and** *as k*
assumes *finite PROB* (*s* $\in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
(*problem-plan-bound PROB* $\leq$ *k*)
shows (*Inf* (*PLS s as*) $\leq$ *problem-plan-bound PROB*)
⟨*proof*⟩

lemma *PLS-NEMPTY*:
fixes *s as*
shows *PLS s as* $\neq$ {}
⟨*proof*⟩

lemma *PLS-nempty-and-has-min*:
fixes *s as*
shows ($\exists$ *x*. (*x* $\in$ *PLS s as*) $\wedge$ (*x* = *Inf* (*PLS s as*)))
⟨*proof*⟩

lemma *PLS-works*:
fixes *x s as*
assumes (*x* $\in$ *PLS s as*)
shows($\exists$ *as′*.
(*exec-plan s as* = *exec-plan s as′*)

∧ (*length as′* = *x*)
∧ (*subseq as′ as*)
)
⟨*proof*⟩

lemma *problem-plan-bound-works*:
fixes *PROB* :: *′a problem* **and** *as* **and** *s* :: *′a state*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as′*.
(*exec-plan s as* = *exec-plan s as′*)
∧ (*subseq as′ as*)
∧ (*length as′* ≤ *problem-plan-bound PROB*)
)
⟨*proof*⟩

definition *MPLS-s* **where**
MPLS-s PROB s ≡ (λ (*s*, *as*). *Inf* (*PLS s as*)) ‘ {(*s*, *as*) | *as*. *as* ∈ *valid-plans PROB*}

— NOTE type of ‘PROB‘ had to be fixed (type mismatch in goal).

lemma *bound-main-lemma-s-3*:
fixes *PROB* :: ((*′a*, *′b*) *fmap* × (*′a*, *′b*) *fmap*) *set* **and** *s*
shows *MPLS-s PROB s* ≠ {}
⟨*proof*⟩

definition *problem-plan-bound-s* **where**
problem-plan-bound-s PROB s = *Sup* (*MPLS-s PROB s*)

— NOTE removed typing from assumption due to matching problems in later proofs.

lemma *bound-on-all-plans-bounds-PLS-s*:
fixes *P f*
assumes (∀ *PROB as s*.
finite PROB ∧ (*P PROB*) ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *as′*.
(*exec-plan s as* = *exec-plan s as′*)
∧ (*subseq as′ as*)
∧ (*length as′* < *f PROB s*)
)
)
shows (∀ *PROB s as*.
finite PROB ∧ (*P PROB*) ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *x*.
(*x* ∈ *PLS s as*)
∧ (*x* < *f PROB s*)
)
)

⟨*proof*⟩
lemma *bound-on-all-plans-bounds-MPLS-s-i*:
fixes *PROB s x*
assumes *s* ∈ *valid-states PROB x* ∈ *MPLS-s PROB s*
shows ∃ *as*. *x* = *Inf* (*PLS s as*) ∧ *as* ∈ *valid-plans PROB*
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-MPLS-s*:
fixes *P f*
assumes (∀ *PROB as s*.
finite PROB ∧ (*P PROB*) ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* < *f PROB s*)
)
)
shows (∀ *PROB x s*.
finite PROB ∧ (*P PROB*) ∧ (*s* ∈ *valid-states PROB*) ⟶ (*x* ∈ *MPLS-s PROB s*)
⟶ (*x* < *f PROB s*)
)
⟨*proof*⟩
lemma *Sup-MPLS-s-lt-if*:
fixes *PROB s k*
assumes (∀ *x*∈*MPLS-s PROB s*. *x* < *k*)
shows *Sup* (*MPLS-s PROB s*) < *k*
⟨*proof*⟩
lemma *bound-child-parent-lemma-s-2*:
fixes *PROB* :: *'a problem* **and** *P* :: *'a problem* ⇒ *bool* **and** *s f*
assumes (∀ (*PROB* :: *'a problem*) *as s*.
finite PROB ∧ (*P PROB*) ∧ (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)
⟶ (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* < *f PROB s*)
)
)
shows (
finite PROB ∧ (*P PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ *problem-plan-bound-s PROB s* < *f PROB s*
)
⟨*proof*⟩

theorem *bound-main-lemma-reachability-s*:
fixes *PROB* :: *'a problem* **and** *s*

assumes *finite PROB s* ∈ *valid-states PROB*
shows (*problem-plan-bound-s PROB s* < *card* (*reachable-s PROB s*))
⟨*proof*⟩

lemma *problem-plan-bound-s-LESS-EQ-problem-plan-bound-thm*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state*
assumes *finite PROB* (*s* ∈ *valid-states PROB*)
shows (*problem-plan-bound-s PROB s* < *problem-plan-bound PROB* + *1*)
⟨*proof*⟩

lemma *AS-VALID-MPLS-VALID*:
fixes *PROB as*
assumes (*as* ∈ *valid-plans PROB*)
shows (*Inf* (*PLS s as*) ∈ *MPLS-s PROB s*)
⟨*proof*⟩
lemma *bound-main-lemma-s-1*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *x*
assumes *finite PROB s* ∈ (*valid-states PROB*) *x* ∈ *MPLS-s PROB s*
shows (*x* ≤ (*2* ^ *card* (*prob-dom PROB*)) − *1*)
⟨*proof*⟩

lemma *problem-plan-bound-s-ge-min-pls*:
fixes *PROB* :: *'a problem* **and** *as k s*
assumes *finite PROB s* ∈ (*valid-states PROB*) *as* ∈ (*valid-plans PROB*)
problem-plan-bound-s PROB s ≤ *k*
shows (*Inf* (*PLS s as*) ≤ *problem-plan-bound-s PROB s*)
⟨*proof*⟩

theorem *bound-main-lemma-s*:
fixes *PROB* :: *'a problem* **and** *s*
assumes *finite PROB* (*s* ∈ *valid-states PROB*)
shows (*problem-plan-bound-s PROB s* ≤ *2* ^ (*card* (*prob-dom PROB*)) − *1*)
⟨*proof*⟩

lemma *problem-plan-bound-s-works*:
fixes *PROB* :: *'a problem* **and** *as s*
assumes *finite PROB* (*as* ∈ *valid-plans PROB*) (*s* ∈ *valid-states PROB*)
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *problem-plan-bound-s PROB s*)
)
⟨*proof*⟩
lemma *PLS-def-ITP2015*:

fixes *s as*
shows *PLS s as* = {*length as′* | *as′*. (*exec-plan s as′* = *exec-plan s as*) ∧ (*subseq as′ as*)}
⟨*proof*⟩
lemma *expanded-problem-plan-bound-charles-thm*:
fixes *PROB* :: ′*a problem*
shows
problem-plan-bound-charles PROB
= *Sup* (
{
Inf (*PLS-charles* (*fst p*) (*snd p*) *PROB*)
| *p*. (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)})

⟨*proof*⟩

lemma *bound-main-lemma-charles-3*:
fixes *PROB* :: ′*a problem*
assumes *finite PROB*
shows *MPLS-charles PROB* ≠ {}
⟨*proof*⟩

lemma *in-PLS-charles-leq-2-pow-n*:
fixes *PROB* :: ′*a problem* **and** *s as*
assumes *finite PROB s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*
shows (∃ *x*.
(*x* ∈ *PLS-charles s as PROB*)
∧ (*x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*))

⟨*proof*⟩
lemma *x-in-MPLS-charles-then*:
fixes *PROB s as*
assumes *x* ∈ *MPLS-charles PROB*
shows ∃ *s as*.
s ∈ *valid-states PROB* ∧ *as* ∈ *valid-plans PROB* ∧ *x* = *Inf* (*PLS-charles s as PROB*)

⟨*proof*⟩

lemma *in-MPLS-charles-leq-2-pow-n*:
fixes *PROB* :: ′*a problem* **and** *x*
assumes *finite PROB x* ∈ *MPLS-charles PROB*
shows *x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
⟨*proof*⟩

lemma *bound-main-lemma-charles*:
fixes *PROB* :: ′*a problem*

assumes *finite PROB*
shows *problem-plan-bound-charles PROB* ≤ *2* ^ (*card* (*prob-dom PROB*)) − *1*
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-PLS-charles*:
fixes *P* **and** *f*
assumes ∀ (*PROB* :: ′*a problem*) *as s*.
(*P PROB*) ∧ *finite PROB* ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′) ∧ (*subseq as*′ *as*)∧ (*length as*′ < *f PROB*))

shows (∀ *PROB s as*.
(*P PROB*) ∧ *finite PROB* ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *x*.
(*x* ∈ *PLS-charles s as PROB*)
∧ (*x* < *f PROB*)))

⟨*proof*⟩
lemma *bound-on-all-plans-bounds-MPLS-charles-i*:
assumes ∀ (*PROB* :: ′*a problem*) *s as*.
(*P PROB*) ∧ *finite PROB* ∧ (*as* ∈ *valid-plans PROB*) ∧ (*s* ∈ *valid-states PROB*)
⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′) ∧ (*subseq as*′ *as*) ∧ (*length as*′ < *f PROB*))

shows ∀ (*PROB* :: ′*a problem*) *s as*.
P PROB ∧ *finite PROB* ∧ *as* ∈ *valid-plans PROB* ∧ *s* ∈ *valid-states PROB*
⟶ *Inf* {*n*. *n* ∈ *PLS-charles s as PROB*} < *f PROB*

⟨*proof*⟩

lemma *bound-on-all-plans-bounds-MPLS-charles*:
fixes *P f*
assumes (∀ (*PROB* :: ′*a problem*) *as s*.
(*P PROB*) ∧ *finite PROB* ∧ (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)
⟶ (∃ *as*′.
(*exec-plan s as* = *exec-plan s as*′)
∧ (*subseq as*′ *as*)
∧ (*length as*′ < *f PROB*)
)
)
shows (∀ *PROB x*.
(*P PROB*) ∧ *finite PROB*
⟶ (*x* ∈ *MPLS-charles PROB*)
⟶ (*x* < *f PROB*)

)
⟨*proof*⟩
lemma *bound-on-all-plans-bounds-problem-plan-bound-charles-i*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* $\forall x \in$ *MPLS-charles PROB*. $x < k$
shows *Sup* (*MPLS-charles PROB*) $\in$ *MPLS-charles PROB*
⟨*proof*⟩

lemma *bound-on-all-plans-bounds-problem-plan-bound-charles*:
fixes *P f*
assumes ($\forall$ (*PROB* :: *'a problem*) *as s*.
(*P PROB*) $\wedge$ *finite PROB* $\wedge$ (*s* $\in$ *valid-states PROB*) $\wedge$ (*as* $\in$ *valid-plans PROB*)
$\longrightarrow$ ($\exists$ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
$\wedge$ (*subseq as' as*)
$\wedge$ (*length as'* < *f PROB*)))

shows ($\forall$ *PROB*.
(*P PROB*) $\wedge$ *finite PROB* $\longrightarrow$ (*problem-plan-bound-charles PROB* < *f PROB*))

⟨*proof*⟩

6.3 The Relation between Diameter, Sublist Diameter and Recurrence Diameter Bounds.

The goal of this subsection is to verify the relation between diameter, sublist diameter and recurrence diameter bounds given by HOL4 Theorem 1, i.e.

d $\delta \leq$ l $\delta \wedge$ l $\delta \leq$ rd δ

where d δ, l δ and rd δ denote the diameter, sublist diameter and recurrence diameter bounds. [Abdualaziz et al., p.20]

The relevant lemmas are ‘sublistD_bounds_D‘ and ‘RD_bounds_sublistD‘ which culminate in theorem ‘sublistD_bounds_D_and_RD_bounds_sublistD‘.

lemma *sublistD-bounds-D*:
fixes *PROB* :: *'a problem*
assumes *finite PROB*
shows *problem-plan-bound-charles PROB* $\leq$ *problem-plan-bound PROB*
⟨*proof*⟩
lemma *MAX-SET-ELIM'*:
fixes *P Q*
assumes *finite P* $P \neq \{\}$ ($\forall x$. ($\forall y$. $y \in P \longrightarrow y \leq x$) $\wedge$ $x \in P \longrightarrow R\ x$)
shows *R* (*Max P*)
⟨*proof*⟩
lemma *MIN-SET-ELIM'*:
fixes *P Q*
assumes *finite P* $P \neq \{\}$ $\forall x$. ($\forall y$. $y \in P \longrightarrow x \leq y$) $\wedge$ $x \in P \longrightarrow Q\ x$
shows *Q* (*Min P*)

⟨*proof*⟩
lemma *RD-bounds-sublistD-i-a*:
  fixes *Pi* :: *'a problem*
  assumes *finite Pi*
  shows *finite* {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
⟨*proof*⟩
lemma *RD-bounds-sublistD-i-b*:
  fixes *Pi* :: *'a problem*
  shows {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*} ≠ {}
⟨*proof*⟩
lemma *RD-bounds-sublistD-i-c*:
  fixes *Pi* :: *'a problem* **and** *as* :: ((*'a*, *bool*) *fmap* × (*'a*, *bool*) *fmap*) *list* **and** *x*
    and *s* :: (*'a*, *bool*) *fmap*
  assumes *s* ∈ *valid-states Pi as* ∈ *valid-plans Pi*
    (∀ *y*. *y* ∈ {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*} ⟶ *y* ≤ *x*)
    x ∈ {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
  shows *Min* (*PLS s as*) ≤ *Max* {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
⟨*proof*⟩
lemma *RD-bounds-sublistD-i*:
  fixes *Pi* :: *'a problem* **and** *x*
  assumes *finite Pi* (∀ *y*. *y* ∈ *MPLS Pi* ⟶ *y* ≤ *x*) *x* ∈ *MPLS Pi*
  shows *x* ≤ *Max* {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
⟨*proof*⟩
lemma *RD-bounds-sublistD*:
  fixes *Pi* :: *'a problem*
  assumes *finite Pi*
  shows *problem-plan-bound Pi* ≤ *RD Pi*
⟨*proof*⟩
theorem *sublistD-bounds-D-and-RD-bounds-sublistD*:
  fixes *PROB* :: *'a problem*
  assumes *finite PROB*
  shows
    problem-plan-bound-charles PROB ≤ *problem-plan-bound PROB*
    ∧ *problem-plan-bound PROB* ≤ *RD PROB*

  ⟨*proof*⟩
lemma *empty-problem-bound*:
  fixes *PROB* :: *'a problem*
  assumes (*prob-dom PROB* = {})
  shows (*problem-plan-bound PROB* = *0*)
⟨*proof*⟩

lemma *problem-plan-bound-works'*:
  fixes *PROB* :: *'a problem* **and** *as s*
  assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
  shows (∃ *as'*.
    (*exec-plan s as'* = *exec-plan s as*)
    ∧ (*subseq as' as*)

$\wedge$ (*length as'* $\leq$ *problem-plan-bound PROB*)
$\wedge$ (*sat-precond-as s as'*)
)
⟨*proof*⟩
lemma *problem-plan-bound-UBound*:
assumes ($\forall$ *as s*.
(*s* $\in$ *valid-states PROB*)
$\wedge$ (*as* $\in$ *valid-plans PROB*)
$\longrightarrow$ ($\exists$ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
$\wedge$ *subseq as' as*
$\wedge$ (*length as'* < *f PROB*)
)
) *finite PROB*
shows (*problem-plan-bound PROB* < *f PROB*)
⟨*proof*⟩

6.4 Traversal Diameter

definition *traversed-states* **where**
traversed-states s as $\equiv$ *set* (*state-list s as*)

lemma *finite-traversed-states*: *finite* (*traversed-states s as*)
⟨*proof*⟩

lemma *traversed-states-nempty*: *traversed-states s as* $\neq$ {}
⟨*proof*⟩

lemma *traversed-states-geq-1*:
fixes *s*
shows *1* $\leq$ *card* (*traversed-states s as*)
⟨*proof*⟩

lemma *init-is-traversed*: *s* $\in$ *traversed-states s as*
⟨*proof*⟩
definition *td* **where**
td PROB $\equiv$ *Sup* {
(*card* (*traversed-states* (*fst p*) (*snd p*))) − *1*
| *p*. (*fst p* $\in$ *valid-states PROB*) $\wedge$ (*snd p* $\in$ *valid-plans PROB*)}

lemma *traversed-states-rem-condless-act*: $\bigwedge$*s*.
traversed-states s (*rem-condless-act s* [] *as*) = *traversed-states s as*

⟨*proof*⟩
lemma *td-UBound-i*:
fixes *PROB* :: ((*'a*, *'b*) *fmap* × (*'a*, *'b*) *fmap*) *set*
assumes *finite PROB*
shows
{
(*card* (*traversed-states* (*fst p*) (*snd p*))) − *1*
| *p*. (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)}
≠ {}

⟨*proof*⟩

lemma *td-UBound*:
fixes *PROB* :: ((*'a*, *'b*) *fmap* × (*'a*, *'b*) *fmap*) *set*
assumes *finite PROB* (∀ *s as*.
(*sat-precond-as s as*) ∧ (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)
⟶ (*card* (*traversed-states s as*) ≤ *k*)
)
shows (*td PROB* ≤ *k* − *1*)
⟨*proof*⟩

end
theory *SystemAbstraction*
imports
Main
HOL−Library.Sublist
HOL−Library.Finite-Map
FactoredSystem
FactoredSystemLib
ActionSeqProcess
Dependency
TopologicalProps
FmapUtils
ListUtils

begin

— NOTE hide 'Map.map_add' because of conflicting notation with 'FactoredSystemLib.map_add_ltr'.
hide-const (**open**) *Map.map-add*
no-notation *Map.map-add* (**infixl** ‹++› *100*)

7 System Abstraction

Projection of an object (state, action, sequence of action or factored representation) to a variable set ‘vs‘ restricts the domain of the object or its components—in case of composite objects—to ‘vs‘. [Abdulaziz et al., p.12]

This section presents the relevant definitions (‘action_proj‘, ‘as_proj‘, ‘prob_proj‘ and ‘ss_proj‘) as well as their characterization.

7.1 Projection of Actions, Sequences of Actions and Factored Representations.

definition *action-proj* **where**
action-proj a vs ≡ (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*))

lemma *action-proj-pair*: *action-proj* (*p*, *e*) *vs* = (*fmrestrict-set vs p*, *fmrestrict-set vs e*)
⟨*proof*⟩

definition *prob-proj* **where**
prob-proj PROB vs ≡ (λ*a. action-proj a vs*) ‘ *PROB*

— NOTE using ’fun’ due to multiple defining equations.
— NOTE name shortened.
fun *as-proj* **where**
as-proj [] - = []
| *as-proj* (*a* # *as*) *vs* = (*if fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {}
then action-proj a vs # *as-proj as vs*
else as-proj as vs
)

— TODO the lemma might be superfluous (follows directly from ’as_proj.simps’).
lemma *as-proj-pair*:
as-proj ((*p*, *e*) # *as*) *vs* = (*if* (*fmdom′* (*fmrestrict-set vs e*) ≠ {})
then action-proj (*p*, *e*) *vs* # *as-proj as vs*
else as-proj as vs
)
as-proj [] *vs* = []
⟨*proof*⟩

lemma *proj-state-succ*:
fixes *s a vs*
assumes (*fst a* $\subseteq_f$ *s*)
shows (*state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*) = *fmrestrict-set vs* (*state-succ s a*))
⟨*proof*⟩

lemma *graph-plan-lemma-1*:

fixes *s vs as*
assumes *sat-precond-as s as*
shows (*exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*) = (*fmrestrict-set vs* (*exec-plan s as*)))
⟨*proof*⟩
lemma *proj-action-dom-eq-inter*:
shows
action-dom (*fst* (*action-proj a vs*)) (*snd* (*action-proj a vs*))
= (*action-dom* (*fst a*) (*snd a*) ∩ *vs*)

⟨*proof*⟩

lemma *graph-plan-neq-mems-state-set-neq-len*:
shows *prob-dom* (*prob-proj PROB vs*) = (*prob-dom PROB* ∩ *vs*)
⟨*proof*⟩
lemma *graph-plan-not-eq-last-diff-paths*:
fixes *PROB vs*
assumes (*s* ∈ *valid-states PROB*)
shows ((*fmrestrict-set vs s*) ∈ *valid-states* (*prob-proj PROB vs*))

⟨*proof*⟩

lemma *dom-eff-subset-imp-dom-succ-eq-proj*:
fixes *h s vs*
assumes (*fmdom′* (*snd h*) ⊆ *fmdom′ s*)
shows (*fmdom′* (*state-succ s* (*action-proj h vs*)) = *fmdom′* (*state-succ s h*))
⟨*proof*⟩

lemma *drest-proj-succ-eq-drest-succ*:
fixes *h s vs*
assumes *fst h* $\subseteq_f$ *s* (*fmdom′* (*snd h*) ⊆ *fmdom′ s*)
shows (*fmrestrict-set vs* (*state-succ s* (*action-proj h vs*)) = *fmrestrict-set vs* (*state-succ s h*))
⟨*proof*⟩
lemma *drest-succ-proj-eq-drest-succ*:
fixes *s vs as*
assumes (*fst a* $\subseteq_f$ *s*)
shows (*state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*) = *fmrestrict-set vs* (*state-succ s a*))
⟨*proof*⟩

lemma *exec-drest-cons-proj-eq-succ*:
fixes *as PROB vs a*
assumes *fst a* $\subseteq_f$ *s*
shows (

exec-plan (*fmrestrict-set vs s*) (*action-proj a vs* # *as*)
= *exec-plan* (*fmrestrict-set vs* (*state-succ s a*)) *as*
)
⟨*proof*⟩

lemma *exec-drest*:
fixes *as a vs*
assumes (*fst a* $\subseteq_f$ *s*)
shows (
exec-plan (*fmrestrict-set vs* (*state-succ s a*)) *as*
= *exec-plan* (*fmrestrict-set vs s*) (*action-proj a vs* # *as*)
)
⟨*proof*⟩

lemma *not-empty-eff-in-as-proj*:
fixes *as a vs*
assumes *fmdom'* (*fmrestrict-set vs* (*snd a*)) ≠ {}
shows (*as-proj* (*a* # *as*) *vs* = (*action-proj a vs* # *as-proj as vs*))
⟨*proof*⟩

lemma *empty-eff-not-in-as-proj*:
fixes *as a vs*
assumes (*fmdom'* (*fmrestrict-set vs* (*snd a*)) = {})
shows (*as-proj* (*a* # *as*) *vs* = *as-proj as vs*)
⟨*proof*⟩

lemma *empty-eff-drest-no-eff*:
fixes *s* **and** *a* **and** *vs*
assumes (*fmdom'* (*fmrestrict-set vs* (*snd a*)) = {})
shows (*fmrestrict-set vs* (*state-succ s* (*action-proj a vs*)) = *fmrestrict-set vs s*)
⟨*proof*⟩

lemma *sat-precond-exec-as-proj-eq-proj-exec*:
fixes *as vs s*
assumes (*sat-precond-as s as*)
shows (*exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*) = *fmrestrict-set vs* (*exec-plan s as*))
⟨*proof*⟩

lemma *action-proj-in-prob-proj*:
assumes (*a* ∈ *PROB*)
shows (*action-proj a vs* ∈ *prob-proj PROB vs*)
⟨*proof*⟩

lemma *valid-as-valid-as-proj*:
fixes *PROB vs*
assumes (*as* ∈ *valid-plans PROB*)
shows (*as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*))
⟨*proof*⟩

lemma *finite-imp-finite-prob-proj*:
fixes *PROB*
assumes *finite PROB*
shows (*finite* (*prob-proj PROB vs*))
⟨*proof*⟩

lemma
fixes *PROB vs as* **and** *s* :: *'a state*
assumes *finite PROB s* ∈ *valid-states PROB as* ∈ (*valid-plans PROB*) *finite vs*
length (*as-proj as vs*) > ((*2* :: *nat*) ^ *card vs*) − *1 sat-precond-as s as*
shows (∃ *as1 as2 as3*.
(*as1* @ *as2* @ *as3* = *as-proj as vs*)
∧ (*exec-plan* (*fmrestrict-set vs s*) (*as1* @ *as2*) = *exec-plan* (*fmrestrict-set vs s*) *as1*)
∧ (*as2* ≠ [])
)
⟨*proof*⟩

lemma *as-proj-eq-filter-action-proj*:
fixes *as vs*
shows *as-proj as vs* = *filter* (λ*a*. *fmdom'* (*snd a*) ≠ {}) (*map* (λ*a*. *action-proj a vs*) *as*)
⟨*proof*⟩

lemma *append-eq-as-proj*:
fixes *as1 as2 as3 p vs*
assumes (*as1* @ *as2* @ *as3* = *as-proj p vs*)
shows (∃ *p-1 p-2 p-3*.
(*p-1* @ *p-2* @ *p-3* = *p*)
∧ (*as2* = *as-proj p-2 vs*)
∧ (*as1* = *as-proj p-1 vs*)
)
⟨*proof*⟩

lemma *succ-drest-eq-drest-succ*:
fixes *a s vs*
shows
state-succ (*fmrestrict-set vs s*) (*action-proj a vs*)
= *fmrestrict-set vs* (*state-succ s* (*action-proj a vs*))

⟨*proof*⟩

lemma *proj-exec-proj-eq-exec-proj*:
fixes *s as vs*
shows
fmrestrict-set vs (*exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*))
= *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*)

⟨*proof*⟩

lemma *proj-exec-proj-eq-exec-proj'*:
fixes *s as vs*
shows
fmrestrict-set vs (*exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*))
= *fmrestrict-set vs* (*exec-plan s* (*as-proj as vs*))

⟨*proof*⟩

lemma *graph-plan-lemma-9*:
fixes *s as vs*
shows
fmrestrict-set vs (*exec-plan s* (*as-proj as vs*))
= *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*)

⟨*proof*⟩

lemma *act-dom-proj-eff-subset-act-dom-eff*:
fixes *a vs*
shows *fmdom'* (*snd* (*action-proj a vs*)) ⊆ *fmdom'* (*snd a*)
⟨*proof*⟩

lemma *exec-as-proj-valid*:
fixes *as s PROB vs*
assumes *s* ∈ *valid-states PROB* (*as* ∈ *valid-plans PROB*)
shows (*exec-plan s* (*as-proj as vs*) ∈ *valid-states PROB*)
⟨*proof*⟩

lemma *drest-exec-as-proj-eq-drest-exec*:
fixes *s as vs*
assumes *sat-precond-as s as*
shows (*fmrestrict-set vs* (*exec-plan s* (*as-proj as vs*)) = *fmrestrict-set vs* (*exec-plan s as*))

⟨*proof*⟩

lemma *action-proj-idempot*:
fixes *a vs*
shows *action-proj* (*action-proj a vs*) *vs* = (*action-proj a vs*)
⟨*proof*⟩

lemma *action-proj-idempot'*:
fixes *a vs*
assumes (*action-dom* (*fst a*) (*snd a*) ⊆ *vs*)
shows (*action-proj a vs* = *a*)
⟨*proof*⟩

lemma *action-proj-idempot''*:
fixes *P vs*
assumes *prob-dom P* ⊆ *vs*
shows *prob-proj P vs* = *P*
⟨*proof*⟩

lemma *sat-precond-as-proj*:
fixes *as s s' vs*
assumes (*sat-precond-as s as*) (*fmrestrict-set vs s* = *fmrestrict-set vs s'*)
shows (*sat-precond-as s'* (*as-proj as vs*))
⟨*proof*⟩

lemma *sat-precond-drest-as-proj*:
fixes *as s s' vs*
assumes (*sat-precond-as s as*) (*fmrestrict-set vs s* = *fmrestrict-set vs s'*)
shows (*sat-precond-as* (*fmrestrict-set vs s'*) (*as-proj as vs*))
⟨*proof*⟩

lemma *as-proj-eq-as*:
assumes (*no-effectless-act as*) (*as* ∈ *valid-plans PROB*) (*prob-dom PROB* ⊆ *vs*)
shows (*as-proj as vs* = *as*)
⟨*proof*⟩

lemma *exec-rem-effless-as-proj-eq-exec-as-proj*:
fixes *s*
shows *exec-plan s* (*as-proj* (*rem-effectless-act as*) *vs*) = *exec-plan s* (*as-proj as vs*)
⟨*proof*⟩

lemma *exec-as-proj-eq-exec-as*:
fixes *PROB as vs s*
assumes (*as* ∈ *valid-plans PROB*) (*prob-dom PROB* ⊆ *vs*)
shows (*exec-plan s* (*as-proj as vs*) = *exec-plan s as*)
⟨*proof*⟩

lemma *dom-prob-proj*: *prob-dom* (*prob-proj PROB vs*) ⊆ *vs*
⟨*proof*⟩
lemma *subset-proj-absorb-1-a*:
fixes *f vs1 vs2*
assumes (*vs1* ⊆ *vs2*)
shows *fmrestrict-set vs1* (*fmrestrict-set vs2 f*) = *fmrestrict-set vs1 f*
⟨*proof*⟩

lemma *subset-proj-absorb-1*:
assumes (*vs1* ⊆ *vs2*)
shows (*action-proj* (*action-proj a vs2*) *vs1* = *action-proj a vs1*)
⟨*proof*⟩

lemma *subset-proj-absorb*:
fixes *PROB vs1 vs2*
assumes *vs1* ⊆ *vs2*
shows *prob-proj* (*prob-proj PROB vs2*) *vs1* = *prob-proj PROB vs1*
⟨*proof*⟩

lemma *union-proj-absorb*:
fixes *PROB vs vs′*
shows *prob-proj* (*prob-proj PROB* (*vs* ∪ *vs′*)) *vs* = *prob-proj PROB vs*
⟨*proof*⟩

lemma *NOT-VS-IN-DOM-PROJ-PRE-EFF*:
fixes *ROB vs v a*
assumes ¬(*v* ∈ *vs*) (*a* ∈ *PROB*)
shows (
((*v* ∈ *fmdom′* (*fst a*)) ⟶ (*v* ∈ *fmdom′* (*fst* (*action-proj a* (*prob-dom PROB* − *vs*)))))
∧ ((*v* ∈ *fmdom′* (*snd a*)) ⟶ (*v* ∈ *fmdom′* (*snd* (*action-proj a* (*prob-dom PROB* − *vs*)))))
)
⟨*proof*⟩

lemma *IN-DISJ-DEP-IMP-DEP-DIFF*:
fixes *PROB vs vs′ v v′*

assumes $(v \in vs')$ $(v' \in vs')$ $(disjnt\ vs\ vs')$
shows $(dep\ PROB\ v\ v' \longrightarrow dep\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))\ v\ v')$
⟨*proof*⟩

lemma *PROB-DOM-PROJ-DIFF*:
fixes *P vs*
shows $prob\text{-}dom\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs)) = (prob\text{-}dom\ PROB) - vs$
⟨*proof*⟩

lemma *two-children-parent-mems-le-finite*:
fixes *PROB vs*
assumes $(vs \subseteq prob\text{-}dom\ PROB)$
shows $(prob\text{-}dom\ (prob\text{-}proj\ PROB\ vs) = vs)$
⟨*proof*⟩
lemma *PROJ-DOM-PRE-EFF-SUBSET-DOM*:
fixes *a vs*
shows
$(fmdom'\ (fst\ (action\text{-}proj\ a\ vs)) \subseteq fmdom'\ (fst\ a))$
$\wedge\ (fmdom'\ (snd\ (action\text{-}proj\ a\ vs)) \subseteq fmdom'\ (snd\ a))$

⟨*proof*⟩

lemma *NOT-IN-PRE-EFF-NOT-IN-PRE-EFF-PROJ*:
fixes *a v vs*
shows
$(\neg(v \in fmdom'\ (fst\ a)) \longrightarrow \neg(v \in fmdom'\ (fst\ (action\text{-}proj\ a\ vs))))$
$\wedge\ (\neg(v \in fmdom'\ (snd\ a)) \longrightarrow \neg(v \in fmdom'\ (snd\ (action\text{-}proj\ a\ vs))))$

⟨*proof*⟩

lemma *dep-proj-dep*:
assumes $dep\ (prob\text{-}proj\ PROB\ vs)\ v\ v'$
shows $dep\ PROB\ v\ v'$
⟨*proof*⟩

lemma *NDEP-PROJ-NDEP*:
fixes $PROB\ vs\ vs'\ vs''$
assumes $(\neg dep\text{-}var\text{-}set\ PROB\ vs\ vs')$
shows $(\neg dep\text{-}var\text{-}set\ (prob\text{-}proj\ PROB\ vs'')\ vs\ vs')$
⟨*proof*⟩

lemma *SUBSET-PROJ-DOM-DISJ*:

fixes *PROB vs vs'*
assumes $(vs \subseteq (prob\text{-}dom\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs'))))$
shows *disjnt vs vs'*
⟨*proof*⟩

lemma *NOT-VS-DEP-IMP-DEP-PROJ*:
fixes *PROB vs v v'*
assumes $\neg(v \in vs)$ $\neg(v' \in vs)$ $(dep\ PROB\ v\ v')$
shows $(dep\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))\ v\ v')$
⟨*proof*⟩

lemma *DISJ-PROJ-NDEP-IMP-NDEP*:
fixes *PROB vs vs' vs''*
assumes
$(disjnt\ vs\ vs'')$ $disjnt\ vs\ vs'$
$\neg(dep\text{-}var\text{-}set\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))\ vs'\ vs'')$
shows $\neg(dep\text{-}var\text{-}set\ PROB\ vs'\ vs'')$
⟨*proof*⟩

lemma *PROJ-DOM-IDEMPOT*:
fixes *PROB*
shows $prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB) = PROB$
⟨*proof*⟩

lemma *prob-proj-idempot*:
fixes *vs vs'*
assumes $(vs \subseteq vs')$
shows $(prob\text{-}proj\ PROB\ vs = prob\text{-}proj\ (prob\text{-}proj\ PROB\ vs')\ vs)$
⟨*proof*⟩

lemma *prob-proj-dom-diff-eq-prob-proj-prob-proj-dom-diff*:
fixes *vs vs'*
shows
$prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - (vs \cup vs'))$
$= prob\text{-}proj$
$(prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))$
$(prob\text{-}dom\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs)) - vs')$

⟨*proof*⟩

lemma *PROJ-DEP-IMP-DEP*:
fixes *PROB vs v v'*
assumes $dep\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))\ v\ v'$
shows $dep\ PROB\ v\ v'$
⟨*proof*⟩

lemma *PROJ-NDEP-TC-IMP-NDEP-TC-OR*:
fixes *PROB vs v v'*
assumes $\neg((\lambda v1'\ v2'.\ dep\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB - vs))\ v1'\ v2')^{++}$
$v\ v')$
shows (
$(\neg((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v\ v'))$
$\vee\ (\exists\, v''.$
$v'' \in vs$
$\wedge\ ((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v\ v'')$
$\wedge\ ((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v''\ v')$
)
)
⟨*proof*⟩

lemma *every-action-proj-eq-as-proj*:
fixes *as vs*
shows *list-all* (λ *a*. *action-proj a vs* = *a*) (*as-proj as vs*)
⟨*proof*⟩

lemma *empty-eff-not-in-as-proj-2*:
fixes *a as vs*
assumes *fmdom'* (*snd* (*action-proj a vs*)) = {}
shows (*as-proj as vs* = *as-proj* (*a* # *as*) *vs*)
⟨*proof*⟩

declare[[*smt-timeout=100*]]

lemma *sublist-as-proj-eq-as*:
fixes *as' as vs*
assumes *subseq as'* (*as-proj as vs*)
shows (*as-proj as' vs* = *as'*)
⟨*proof*⟩

lemma *DISJ-EFF-DISJ-PROJ-EFF*:
fixes *a s vs*
assumes *fmdom'* (*snd a*) ∩ *s* = {}
shows (*fmdom'* (*snd* (*action-proj a vs*)) ∩ *s* = {})

⟨*proof*⟩
lemma *state-succ-proj-eq-state-succ*:
fixes *a s vs*
assumes (*varset-action a vs*) ($fst\ a \subseteq_f s$) (*fmdom'* (*snd a*) ⊆ *fmdom' s*)
shows (*state-succ s* (*action-proj a vs*) = *state-succ s a*)
⟨*proof*⟩

lemma *no-effectless-proj*:
fixes *vs as*
shows *no-effectless-act* (*as-proj as vs*)
⟨*proof*⟩
lemma *as-proj-valid-in-prob-proj*:
fixes *PROB vs as*
assumes (*as* ∈ *valid-plans PROB*)
shows (*as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*))
⟨*proof*⟩
lemma *prob-proj-comm*:
fixes *PROB vs vs'*
shows *prob-proj* (*prob-proj PROB vs*) *vs'* = *prob-proj* (*prob-proj PROB vs'*) *vs*
⟨*proof*⟩
lemma *vset-proj-imp-vset*:
fixes *vs vs' a*
assumes (*varset-action a vs'*) (*varset-action* (*action-proj a vs'*) *vs*)
shows (*varset-action a vs*)
⟨*proof*⟩

lemma *vset-imp-vset-act-proj-diff*:
fixes *PROB vs vs' a*
assumes (*varset-action a vs*)
shows (*varset-action* (*action-proj a* (*prob-dom PROB* − *vs'*)) *vs*)
⟨*proof*⟩

lemma *action-proj-disj-diff*:
assumes (*action-dom* (*fst a*) (*snd a*) ⊆ *vs1*) (*vs2* ∩ *vs3* = {})
shows (*action-proj* (*action-proj a* (*vs1* − *vs2*)) *vs3* = *action-proj a vs3*)
⟨*proof*⟩

lemma *disj-proj-proj-eq-proj*:
fixes *PROB vs vs'*
assumes (*vs* ∩ *vs'* = {})
shows *prob-proj* (*prob-proj PROB* (*prob-dom PROB* − *vs'*)) *vs* = *prob-proj PROB vs*
⟨*proof*⟩

lemma *n-replace-proj-le-n-as-2*:
fixes *a vs vs'*
assumes (*vs* ⊆ *vs'*) (*varset-action a vs'*)
shows (*varset-action* (*action-proj a vs'*) *vs* ⟷ *varset-action a vs*)
⟨*proof*⟩
lemma *empty-problem-proj-bound*:

fixes *PROB* :: *'a problem*
shows *problem-plan-bound* (*prob-proj PROB* {}) = *0*
⟨*proof*⟩

lemma *problem-plan-bound-works-proj*:
fixes *PROB* :: *'a problem* **and** *s as vs*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*) (*sat-precond-as s as*)
shows (∃ *as'*.
(*exec-plan* (*fmrestrict-set vs s*) *as'* = *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*))
∧ (*length as'* ≤ *problem-plan-bound* (*prob-proj PROB vs*))
∧ (*subseq as'* (*as-proj as vs*))
∧ (*sat-precond-as s as'*)
∧ (*no-effectless-act as'*)
)
⟨*proof*⟩
lemma *action-proj-inter-i*: *fmrestrict-set V* (*fmrestrict-set W f*) = *fmrestrict-set* (*V* ∩ *W*) *f*
⟨*proof*⟩

lemma *action-proj-inter*: *action-proj* (*action-proj a vs1*) *vs2* = *action-proj a* (*vs1* ∩ *vs2*)
⟨*proof*⟩

lemma *prob-proj-inter*: *prob-proj* (*prob-proj PROB vs1*) *vs2* = *prob-proj PROB* (*vs1* ∩ *vs2*)
⟨*proof*⟩

7.2 Snapshotting

A snapshot is an abstraction concept of the system in which the assignment of a set of variables is fixed and actions whose preconditions or effects violate the fixed assignments are eliminated. [Abdulaziz et al., p.28]

Formally this notion is build on the definition of agreement of states (‘agree‘), which states that variables ‘v‘, ‘v'‘in the shared domain of two states must be assigned to the same value. A snapshot w.r.t to a state ‘s‘ is then defined as the set of actions of a problem where the precondition and the effect agree. [Abdulaziz et al., Definition 16, HOL4 Definition 16, p.28]

definition *agree* **where**
agree s1 s2 ≡ (∀ *v*. (*v* ∈ *fmdom' s1*) ∧ (*v* ∈ *fmdom' s2*) ⟶ (*fmlookup s1 v* = *fmlookup s2 v*))

— NOTE added lemma.
lemma *state-succ-fixpoint-if*:
fixes *a s PROB*

assumes $a \in PROB$ $(s \in valid\text{-}states\ PROB)$ $fst\ a \subseteq_f s$ $agree\ (snd\ a)\ s$
shows $state\text{-}succ\ s\ a = s$
⟨*proof*⟩

lemma *agree-state-succ-idempot*:
assumes $(a \in PROB)$ $(s \in valid\text{-}states\ PROB)$ $(agree\ (snd\ a)\ s)$
shows $(state\text{-}succ\ s\ a = s)$
⟨*proof*⟩
lemma *fmdom′-fmrestrict-set*:
fixes $X\ f$
shows $fmdom'\ (fmrestrict\text{-}set\ X\ f) = X \cap (fmdom'\ f)$
⟨*proof*⟩
lemma *fmdom′-fmrestrict-set-fmadd*:
fixes $X\ f\ g$
shows $fmdom'\ (fmrestrict\text{-}set\ X\ (f ++_f g)) = X \cap (fmdom'\ f \cup fmdom'\ g)$
⟨*proof*⟩
lemma *fmrestrict-agree*:
fixes $X\ x\ f\ g$
assumes $agree\ (fmrestrict\text{-}set\ X\ f)\ (fmrestrict\text{-}set\ X\ g)$ $x \in X \cap fmdom'\ f \cap fmdom'\ g$
shows $fmlookup\ (fmrestrict\text{-}set\ X\ f)\ x = fmlookup\ (fmrestrict\text{-}set\ X\ g)\ x$
⟨*proof*⟩

lemma *agree-restrict-state-succ-idempot*:
assumes $(a \in PROB)$ $(s \in valid\text{-}states\ PROB)$
$(agree\ (fmrestrict\text{-}set\ vs\ (snd\ a))\ (fmrestrict\text{-}set\ vs\ s))$
shows $(fmrestrict\text{-}set\ vs\ (state\text{-}succ\ s\ a) = fmrestrict\text{-}set\ vs\ s)$
⟨*proof*⟩

lemma *agree-exec-idempot*:
assumes $(as \in valid\text{-}plans\ PROB)$ $(s \in valid\text{-}states\ PROB)$
$(\forall a.\ ListMem\ a\ as \longrightarrow agree\ (snd\ a)\ s)$
shows $(exec\text{-}plan\ s\ as = s)$
⟨*proof*⟩

lemma *agree-restrict-exec-idempot*:
fixes $s\ s'$
assumes $(as \in valid\text{-}plans\ PROB)$ $(s' \in valid\text{-}states\ PROB)$ $(s \in valid\text{-}states\ PROB)$
$(\forall a.\ ListMem\ a\ as \longrightarrow agree\ (fmrestrict\text{-}set\ vs\ (snd\ a))\ (fmrestrict\text{-}set\ vs\ s))$
$(fmrestrict\text{-}set\ vs\ s' = fmrestrict\text{-}set\ vs\ s)$
shows $(fmrestrict\text{-}set\ vs\ (exec\text{-}plan\ s'\ as) = fmrestrict\text{-}set\ vs\ s)$
⟨*proof*⟩

lemma *agree-restrict-exec-idempot-pair*:

fixes *s s′*
assumes (*as* ∈ *valid-plans PROB*) (*s′* ∈ *valid-states PROB*) (*s* ∈ *valid-states PROB*)
(∀ *p e*. *ListMem* (*p*, *e*) *as* ⟶ *agree* (*fmrestrict-set vs e*) (*fmrestrict-set vs s*))
(*fmrestrict-set vs s′* = *fmrestrict-set vs s*)
shows (*fmrestrict-set vs* (*exec-plan s′ as*) = *fmrestrict-set vs s*)
⟨*proof*⟩

lemma *agree-comm*: *agree x x′* = *agree x′ x*
⟨*proof*⟩

lemma *restricted-agree-imp-agree*:
assumes (*fmdom′ s2* ⊆ *vs*) (*agree* (*fmrestrict-set vs s1*) *s2*)
shows (*agree s1 s2*)
⟨*proof*⟩

lemma *agree-imp-submap*:
assumes *f1* $\subseteq_f$ *f2*
shows *agree f1 f2*
⟨*proof*⟩

lemma *agree-FUNION*:
assumes (*agree fm fm1*) (*agree fm fm2*)
shows (*agree fm* (*fm1* ++ *fm2*))
⟨*proof*⟩

lemma *agree-fm-list-union*:
fixes *fm*
assumes (∀ *fm′*. *ListMem fm′ fmList* ⟶ *agree fm fm′*)
shows (*agree fm* (*foldr fmap-add-ltr fmList fmempty*))
⟨*proof*⟩

lemma *DRESTRICT-EQ-AGREE*:
assumes (*fmdom′ s2* ⊆ *vs2*) (*fmdom′ s1* ⊆ *vs1*)
shows ((*fmrestrict-set vs2 s1* = *fmrestrict-set vs1 s2*) ⟶ *agree s1 s2*)
⟨*proof*⟩

lemma *SUBMAPS-AGREE*: (*s1* $\subseteq_f$ *s*) ∧ (*s2* $\subseteq_f$ *s*) ⟹ (*agree s1 s2*)
⟨*proof*⟩
definition *snapshot* **where**
snapshot PROB s = {*a* | *a*. *a* ∈ *PROB* ∧ *agree* (*fst a*) *s* ∧ *agree* (*snd a*) *s*}

lemma *snapshot-pair*: *snapshot PROB s* = {(*p*, *e*). (*p*, *e*) ∈ *PROB* ∧ *agree p s* ∧ *agree e s*}
⟨*proof*⟩

lemma *action-agree-valid-in-snapshot*:
assumes (*a* ∈ *PROB*) (*agree* (*fst a*) *s*) (*agree* (*snd a*) *s*)
shows (*a* ∈ *snapshot PROB s*)
⟨*proof*⟩

lemma *as-mem-agree-valid-in-snapshot*:
assumes (∀ *a*. *ListMem a as* ⟶ *agree* (*fst a*) *s* ∧ *agree* (*snd a*) *s*) (*as* ∈ *valid-plans PROB*)
shows (*as* ∈ *valid-plans* (*snapshot PROB s*))
⟨*proof*⟩

lemma *fmrestrict-agree-monotonous*:
fixes *f g X*
assumes *agree f g*
shows *agree* (*fmrestrict-set X f*) (*fmrestrict-set X g*)
⟨*proof*⟩
lemma *SUBMAP-FUNION-DRESTRICT-i*:
fixes *v vsa vsb f g*
assumes *v* ∈ *vsa*
shows
fmlookup (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *f*) *v*
= *fmlookup* (*fmrestrict-set* (*vsa* ∩ *vs*) *f*) *v*

⟨*proof*⟩

lemma *SUBMAP-FUNION-DRESTRICT'*:
assumes (*agree fma fmb*) (*vsa* ⊆ *fmdom' fma*) (*vsb* ⊆ *fmdom' fmb*)
(*fmrestrict-set vsa fm* = *fmrestrict-set* (*vsa* ∩ *vs*) *fma*)
(*fmrestrict-set vsb fm* = *fmrestrict-set* (*vsb* ∩ *vs*) *fmb*)
shows (*fmrestrict-set* (*vsa* ∪ *vsb*) *fm* = *fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) (*fma* ++ *fmb*))
⟨*proof*⟩

lemma *UNION-FUNION-DRESTRICT-SUBMAP*:
assumes (*vs1* ⊆ *fmdom' fma*) (*vs2* ⊆ *fmdom' fmb*) (*agree fma fmb*)
(*fmrestrict-set vs1 fma* $\subseteq_f$ *s*) (*fmrestrict-set vs2 fmb* $\subseteq_f$ *s*)
shows (*fmrestrict-set* (*vs1* ∪ *vs2*) (*fma* ++ *fmb*) $\subseteq_f$ *s*)
⟨*proof*⟩

lemma *agree-DRESTRICT*:
assumes *agree s1 s2*
shows *agree* (*fmrestrict-set vs s1*) (*fmrestrict-set vs s2*)
⟨*proof*⟩

lemma *agree-DRESTRICT-2*:
  assumes (*fmdom′ s1* ⊆ *vs1*) (*fmdom′ s2* ⊆ *vs2*) (*agree s1 s2*)
  shows (*agree* (*fmrestrict-set vs2 s1*) (*fmrestrict-set vs1 s2*))
  ⟨*proof*⟩
lemma *snapshot-eq-filter*:
  shows *snapshot PROB s* = *Set.filter* (λ*a. agree* (*fst a*) *s* ∧ *agree* (*snd a*) *s*) *PROB*
  ⟨*proof*⟩
corollary *snapshot-subset*:
  shows *snapshot PROB s* ⊆ *PROB*
  ⟨*proof*⟩

lemma *FINITE-snapshot*:
  assumes *finite PROB*
  shows *finite* (*snapshot PROB s*)
⟨*proof*⟩
lemma *dom-proj-snapshot*:
  prob-dom (*prob-proj PROB* (*prob-dom* (*snapshot PROB s*))) = *prob-dom* (*snapshot PROB s*)
  ⟨*proof*⟩

lemma *valid-states-snapshot*:
  valid-states (*prob-proj PROB* (*prob-dom* (*snapshot PROB s*))) = *valid-states* (*snapshot PROB s*)
  ⟨*proof*⟩

lemma *valid-proj-neq-succ-restricted-neq-succ*:
  assumes (*x′* ∈ *prob-proj PROB vs*) (*state-succ s x′* ≠ *s*)
  shows (*fmrestrict-set vs* (*state-succ s x′*) ≠ *fmrestrict-set vs s*)
  ⟨*proof*⟩

lemma *proj-successors*:
  ((λ*s. fmrestrict-set vs s*) ‘ (*state-successors* (*prob-proj PROB vs*) *s*))
    ⊆ (*state-successors* (*prob-proj PROB vs*) (*fmrestrict-set vs s*))

⟨*proof*⟩

lemma *state-in-successor-proj-in-state-in-successor*:
  (*s′* ∈ *state-successors* (*prob-proj PROB vs*) *s*)
  ⟹ (*fmrestrict-set vs s′* ∈ *state-successors* (*prob-proj PROB vs*) (*fmrestrict-set vs s*))
  ⟨*proof*⟩

lemma *proj-FDOM-eff-subset-FDOM-valid-states*:
  fixes *p e s*
  assumes ((*p*, *e*) ∈ *prob-proj PROB vs*) (*s* ∈ *valid-states PROB*)
  shows (*fmdom′ e* ⊆ *fmdom′ s*)
  ⟨*proof*⟩

lemma *valid-proj-action-valid-succ*:
  assumes (*h* ∈ *prob-proj PROB vs*) (*s* ∈ *valid-states PROB*)
  shows (*state-succ s h* ∈ *valid-states PROB*)
⟨*proof*⟩

lemma *proj-successors-of-valid-are-valid*:
  assumes (*s* ∈ *valid-states PROB*)
  shows (*state-successors* (*prob-proj PROB vs*) *s* ⊆ (*valid-states PROB*))
  ⟨*proof*⟩

7.3 State Space Projection

definition *ss-proj* **where**
  ss-proj ss vs ≡ (λ*s*. *fmrestrict-set vs s*) ‘ *ss*

— NOTE added lemma.
— TODO refactor into 'Fmap_Utils'.
lemma *fmrestrict-set-inter-img*:
  fixes *A X Y*
  shows *fmrestrict-set* (*X* ∩ *Y*) ‘ *A* = (*fmrestrict-set X* ∘ *fmrestrict-set Y*) ‘ *A*
⟨*proof*⟩

lemma *invariantStateSpace-thm-9*:
  fixes *ss vs1 vs2*
  shows *ss-proj ss* (*vs1* ∩ *vs2*) = *ss-proj* (*ss-proj ss vs2*) *vs1*
⟨*proof*⟩

lemma *FINITE-ss-proj*:
  fixes *ss vs*
  assumes *finite ss*
  shows *finite* (*ss-proj ss vs*)
  ⟨*proof*⟩

lemma *nempty-stateSpace-nempty-ss-proj*:
  assumes (*ss* ≠ {})
  shows (*ss-proj ss vs* ≠ {})
  ⟨*proof*⟩

lemma *invariantStateSpace-thm-5*:
  fixes *ss vs domain*
  assumes (*stateSpace ss domain*)
  shows (*stateSpace* (*ss-proj ss vs*) (*domain* ∩ *vs*))
  ⟨*proof*⟩

lemma *dom-subset-ssproj-eq-ss*:
  fixes *ss domain vs*
  assumes (*stateSpace ss domain*) (*domain* ⊆ *vs*)
  shows (*ss-proj ss vs* = *ss*)
  ⟨*proof*⟩

lemma *neq-vs-neq-ss-proj*:
fixes *vs*
assumes ($ss \neq \{\}$) (*stateSpace ss vs*) ($vs1 \subseteq vs$) ($vs2 \subseteq vs$) ($vs1 \neq vs2$)
shows (*ss-proj ss vs1* $\neq$ *ss-proj ss vs2*)
⟨*proof*⟩

lemma *subset-dom-stateSpace-ss-proj*:
fixes *vs1 vs2*
assumes ($vs1 \subseteq vs2$) (*stateSpace ss vs2*)
shows (*stateSpace* (*ss-proj ss vs1*) *vs1*)
⟨*proof*⟩

lemma *card-proj-leq*:
assumes *finite PROB*
shows *card* (*prob-proj PROB vs*) $\leq$ *card PROB*
⟨*proof*⟩

end
theory *Acyclicity*
imports *Main*
begin

8 Acyclicity

Two of the discussed bounding algorithms ("top-down" and "bottom-up") exploit acyclicity of the system under projection on sets of state variables closed under mutual variable dependency. [Abdulaziz et al., p.11]

This specific notion of acyclicity is formalised using topologically sorted dependency graphs induced by the variable dependency relation. [Abdulaziz et al., p.14]

8.1 Topological Sorting of Dependency Graphs

fun *top-sorted-abs* **where**
top-sorted-abs R [] = *True*
| *top-sorted-abs R* (*h* # *l*) = (*list-all* ($\lambda x.\ \neg R\ x\ h$) *l* $\wedge$ *top-sorted-abs R l*)

lemma *top-sorted-abs-mem*:
assumes (*top-sorted-abs R* (*h* # *l*)) (*ListMem x l*)
shows ($\neg$ *R x h*)
⟨*proof*⟩

lemma *top-sorted-cons*:
assumes *top-sorted-abs R* (*h* # *l*)
shows (*top-sorted-abs R l*)
⟨*proof*⟩

8.2 The Weightiest Path Function (wlp)

The weightiest path function is a generalization of an algorithm which computes the longest path in a DAG starting at a given vertex ‘v‘. Its arguments are the relation ‘R‘ which induces the graph, a weighing function ‘w‘ assigning weights to vertices, an accumulating functions ‘f‘ and ‘g‘ which aggregate vertex weights into a path weight and the weights of different paths respectively, the considered vertex and the graph represented as a topological sorted list. [Abdulaziz et al., p.18]

Typical weight combining functions have the properties defined by ‘geq_arg‘ and ‘increasing‘. [Abdulaziz et al., p.18]

fun *wlp* **where**
  wlp R w g f x [] = *w x*
| *wlp R w g f x* (*h* # *l*) = (*if R x h*
    then g (*f* (*w x*) (*wlp R w g f h l*)) (*wlp R w g f x l*)
    else wlp R w g f x l
)

— NOTE name shortened.
definition *geq-arg* **where**
  geq-arg f ≡ (∀ *x y*. (*x* ≤ *f x y*) ∧ (*y* ≤ *f x y*))

lemma *individual-weight-less-eq-lp*:
  fixes *w* :: *'a* ⇒ *nat*
  assumes *geq-arg g*
  shows (*w x* ≤ *wlp R w g f x l*)
  ⟨*proof*⟩
lemma *lp-geq-lp-from-successor*:
  fixes *vtx1* **and** *f g* :: *nat* ⇒ *nat* ⇒ *nat*
  assumes *geq-arg f geq-arg g* (∀ *vtx*. *ListMem vtx G* ⟶ ¬*R vtx vtx*) *R vtx2 vtx1*
    ListMem vtx1 G top-sorted-abs R G
  shows (*f* (*w vtx2*) (*wlp R w g f vtx1 G*) ≤ (*wlp R w g f vtx2 G*))
  ⟨*proof*⟩

definition *increasing* **where**
  increasing f ≡ (∀ *e b c d*. (*e* ≤ *c*) ∧ (*b* ≤ *d*) ⟶ (*f e b* ≤ *f c d*))

lemma *weight-fun-leq-imp-lp-leq*: ⋀*x*.
  (*increasing f*)
  ⟹ (*increasing g*)
  ⟹ (∀ *y*. *ListMem y l* ⟶ *w1 y* ≤ *w2 y*)
  ⟹ (*w1 x* ≤ *w2 x*)
  ⟹ (*wlp R w1 g f x l* ≤ *wlp R w2 g f x l*)

⟨*proof*⟩
lemma *wlp-congruence-rule*:
fixes *l1 l2 R1 R2 w1 w2 g1 g2 f1 f2 x1 x2*
assumes $(l1 = l2)$ $(\forall y.\ ListMem\ y\ l2 \longrightarrow (R1\ x1\ y = R2\ x2\ y))$
$(\forall y.\ ListMem\ y\ l2 \longrightarrow (R1\ y\ x1 = R2\ y\ x2))$ $(w1\ x1 = w2\ x2)$
$(\forall y1\ y2.\ (y1 = y2) \longrightarrow (f1\ (w1\ x1)\ y1 = f2\ (w2\ x2)\ y2))$
$(\forall y1\ y2\ z1\ z2.\ (y1 = y2) \wedge (z1 = z2) \longrightarrow ((g1\ (f1\ (w1\ x1)\ y1)\ z1) = (g2\ (f2\ (w2\ x2)\ y2)\ z2)))$
$(\forall x\ y.\ ListMem\ x\ l2 \wedge ListMem\ y\ l2 \longrightarrow (R1\ x\ y = R2\ x\ y))$
$(\forall x.\ ListMem\ x\ l2 \longrightarrow (w1\ x = w2\ x))$
$(\forall x\ y\ z.\ ListMem\ x\ l2 \longrightarrow (g1\ (f1\ (w1\ x)\ y)\ z = g2\ (f2\ (w2\ x)\ y)\ z))$
$(\forall x\ y.\ ListMem\ x\ l2 \longrightarrow (f1\ (w1\ x)\ y = f2\ (w1\ x)\ y))$
shows $((wlp\ R1\ w1\ g1\ f1\ x1\ l1) = (wlp\ R2\ w2\ g2\ f2\ x2\ l2))$
⟨*proof*⟩

lemma *wlp-ite-weights*:
fixes *x*
assumes $\forall y.\ ListMem\ y\ l1 \longrightarrow P\ y\ P\ x$
shows $((wlp\ R\ (\lambda y.\ if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l1) = (wlp\ R\ w1\ g\ f\ x\ l1))$
⟨*proof*⟩

lemma *map-wlp-ite-weights*:
$(\forall x.\ ListMem\ x\ l1 \longrightarrow P\ x)$
$\Longrightarrow (\forall x.\ ListMem\ x\ l2 \longrightarrow P\ x)$
$\Longrightarrow$ (
$map\ (\lambda x.\ wlp\ R\ (\lambda y.\ if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l1)\ l2$
$= map\ (\lambda x.\ wlp\ R\ w1\ g\ f\ x\ l1)\ l2$
)

⟨*proof*⟩

lemma *wlp-weight-lamda-exp*: $\bigwedge x.\ wlp\ R\ w\ g\ f\ x\ l = wlp\ R\ (\lambda y.\ w\ y)\ g\ f\ x\ l$
⟨*proof*⟩

lemma *img-wlp-ite-weights*:
$(\forall x.\ ListMem\ x\ l \longrightarrow P\ x)$
$\Longrightarrow (\forall x.\ x \in s \longrightarrow P\ x)$
$\Longrightarrow$ (
$(\lambda x.\ wlp\ R\ (\lambda y.\ if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l)$ ‘ s
$= (\lambda x.\ wlp\ R\ w1\ g\ f\ x\ l)$ ‘ s
)

⟨*proof*⟩

end
theory *AcycSspace*
  imports
    FactoredSystem
    ActionSeqProcess
    SystemAbstraction
    Acyclicity
    FmapUtils
begin

9 Acyclic State Spaces

value (*state-successors* (*prob-proj PROB vs*))
definition *S*
  where *S vs lss PROB s* ≡ *wlp*
    (λ*x y*. *y* ∈ (*state-successors* (*prob-proj PROB vs*) *x*))
    (λ*s*. *problem-plan-bound* (*snapshot PROB s*))
    (*max* :: *nat* ⇒ *nat* ⇒ *nat*) (λ*x y*. *x* + *y* + *1*) *s lss*

— NOTE name shortened.
— NOTE using 'fun' because of multiple defining equations.
fun *vars-change* **where**
  vars-change [] *vs s* = []
| *vars-change* (*a* # *as*) *vs s* = (*if fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s*
    then state-succ s a # *vars-change as vs* (*state-succ s a*)
    else vars-change as vs (*state-succ s a*)
)

lemma *vars-change-cat*:
  fixes *s*
  shows
    vars-change (*as1* @ *as2*) *vs s*
    = (*vars-change as1 vs s* @ *vars-change as2 vs* (*exec-plan s as1*))

  ⟨*proof*⟩

lemma *empty-change-no-change*:
  fixes *s*
  assumes (*vars-change as vs s* = [])
  shows (*fmrestrict-set vs* (*exec-plan s as*) = *fmrestrict-set vs s*)
  ⟨*proof*⟩
lemma *zero-change-imp-all-effects-submap*:
  fixes *s s'*
  assumes (*vars-change as vs s* = []) (*sat-precond-as s as*) (*ListMem b as*)
    (*fmrestrict-set vs s* = *fmrestrict-set vs s'*)

shows (*fmrestrict-set vs* (*snd b*) $\subseteq_f$ *fmrestrict-set vs s'*)
⟨*proof*⟩

lemma *zero-change-imp-all-preconds-submap*:
fixes *s s'*
assumes (*vars-change as vs s* = []) (*sat-precond-as s as*) (*ListMem b as*)
(*fmrestrict-set vs s* = *fmrestrict-set vs s'*)
shows (*fmrestrict-set vs* (*fst b*) $\subseteq_f$ *fmrestrict-set vs s'*)
⟨*proof*⟩

lemma *no-vs-change-valid-in-snapshot*:
assumes (*as* ∈ *valid-plans PROB*) (*sat-precond-as s as*) (*vars-change as vs s* =
[])
shows (*as* ∈ *valid-plans* (*snapshot PROB* (*fmrestrict-set vs s*)))
⟨*proof*⟩
lemma *no-vs-change-obtain-snapshot-bound-1st-step*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* (*vars-change as vs s* = []) (*sat-precond-as s as*)
(*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(
exec-plan (*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) *s*)
as
= *exec-plan* (*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
s) *as'*
)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)
⟨*proof*⟩
lemma *no-vs-change-obtain-snapshot-bound-2nd-step*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* (*vars-change as vs s* = []) (*sat-precond-as s as*)
(*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(
exec-plan (*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) *s*)
as
= *exec-plan* (*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
s) *as'*
)
∧ (*subseq as' as*)
∧ (*sat-precond-as s as'*)
∧ (*length as'* ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)
⟨*proof*⟩

lemma *no-vs-change-obtain-snapshot-bound-3rd-step*:
assumes *finite* (*PROB* :: *'a problem*) (*vars-change as vs s* = []) (*no-effectless-act as*)
(*sat-precond-as s as*) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(
fmrestrict-set (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as*)
= *fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as'*)
)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)
⟨*proof*⟩
lemma *no-vs-change-snapshot-s-vs-is-valid-bound-i*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* (*vars-change as vs s* = []) (*no-effectless-act as*)
(*sat-precond-as s as*) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
fmrestrict-set (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as*) =
fmrestrict-set (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as'*)
subseq as' as length as' ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*))
shows
fmrestrict-set (*fmdom'* (*exec-plan s as*) − *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
(*exec-plan s as*)
= *fmrestrict-set* (*fmdom'* (*exec-plan s as*) − *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
s
∧ *fmrestrict-set* (*fmdom'* (*exec-plan s as'*) − *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
(*exec-plan s as'*)
= *fmrestrict-set* (*fmdom'* (*exec-plan s as'*) − *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
s
⟨*proof*⟩
lemma *no-vs-change-snapshot-s-vs-is-valid-bound*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* (*vars-change as vs s* = []) (*no-effectless-act as*)
(*sat-precond-as s as*) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* <= *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)

⟨*proof*⟩
lemma *snapshot-bound-leq-S*:
  shows
    problem-plan-bound (*snapshot PROB* (*fmrestrict-set vs s*))
    ≤ *S vs lss PROB* (*fmrestrict-set vs s*)

⟨*proof*⟩
lemma *S-geq-S-succ-plus-ell*:
  assumes ($s \in$ *valid-states PROB*)
    (*top-sorted-abs* ($\lambda x\ y.\ y \in$ *state-successors* (*prob-proj PROB vs*) *x*) *lss*)
    ($s' \in$ *state-successors* (*prob-proj PROB vs*) *s*) (*set lss* = *valid-states* (*prob-proj PROB vs*))
  shows (
    problem-plan-bound (*snapshot PROB* (*fmrestrict-set vs s*))
      + *S vs lss PROB* (*fmrestrict-set vs s'*)
      + (*1* :: *nat*)
    ≤ *S vs lss PROB* (*fmrestrict-set vs s*)
)
⟨*proof*⟩

lemma *vars-change-cons*:
  fixes *s s'*
  assumes (*vars-change as vs s* = (*s'* # *ss*))
  shows (∃ *as1 act as2*.
    (*as* = *as1* @ (*act* # *as2*))
    ∧ (*vars-change as1 vs s* = [])
    ∧ (*state-succ* (*exec-plan s as1*) *act* = *s'*)
    ∧ (*vars-change as2 vs* (*state-succ* (*exec-plan s as1*) *act*) = *ss*)
)
  ⟨*proof*⟩

lemma *vars-change-cons-2*:
  fixes *s s'*
  assumes (*vars-change as vs s* = (*s'* # *ss*))
  shows (*fmrestrict-set vs s'* ≠ *fmrestrict-set vs s*)
  ⟨*proof*⟩
lemma *problem-plan-bound-S-bound-1st-step*:
  fixes *PROB* :: *'a problem*
  assumes *finite PROB* (*top-sorted-abs* ($\lambda x\ y.\ y \in$ *state-successors* (*prob-proj PROB vs*) *x*) *lss*)
    (*set lss* = *valid-states* (*prob-proj PROB vs*)) ($s \in$ *valid-states PROB*)
    ($as \in$ *valid-plans PROB*) (*no-effectless-act as*) (*sat-precond-as s as*)
  shows (∃ *as'*.
      (*exec-plan s as'* = *exec-plan s as*)
      ∧ (*subseq as' as*)
      ∧ (*length as'* <= *S vs lss PROB* (*fmrestrict-set vs s*))
)

⟨*proof*⟩
lemma *problem-plan-bound-S-bound-2nd-step*:
assumes *finite* (*PROB* :: *'a problem*)
(*top-sorted-abs* (λx *y*. *y* ∈ *state-successors* (*prob-proj PROB vs*) *x*) *lss*)
(*set lss* = *valid-states* (*prob-proj PROB vs*)) (*s* ∈ *valid-states PROB*)
(*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(*exec-plan s as'* = *exec-plan s as*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *S vs lss PROB* (*fmrestrict-set vs s*))
)
⟨*proof*⟩
lemma *S-in-MPLS-leq-2-pow-n*:
assumes *finite* (*PROB* :: *'a problem*)
(*top-sorted-abs* (λ *x y*. *y* ∈ *state-successors* (*prob-proj PROB vs*) *x*) *lss*)
(*set lss* = *valid-states* (*prob-proj PROB vs*)) (*s* ∈ *valid-states PROB*)
(*as* ∈ *valid-plans PROB*)
shows (∃ *as'*.
(*exec-plan s as'* = *exec-plan s as*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *Sup* {*S vs lss PROB s'* | *s'*. *s'* ∈ *valid-states* (*prob-proj PROB vs*)})
)
⟨*proof*⟩
lemma *problem-plan-bound-S-bound*:
fixes *PROB* :: *'a problem*
assumes *finite PROB* (*top-sorted-abs* (λ*x y*. *y* ∈ *state-successors* (*prob-proj PROB vs*) *x*) *lss*)
(*set lss* = *valid-states* (*prob-proj PROB vs*))
shows
problem-plan-bound PROB
≤ *Sup* {*S vs lss PROB* (*s'* :: *'a state*) | *s'*. *s'* ∈ *valid-states* (*prob-proj PROB vs*)}

⟨*proof*⟩

9.1 State Space Acyclicity

State space acyclicity is again formalized using graphs to model the state space. However the relation inducing the graph is the successor relation on states. [Abdulaziz et al., Definition 15, HOL4 Definition 15, p.27]

With this, the acyclic system compositional bound ‘S‘ can be shown to be an upper bound on the sublist diameter (lemma ‘problem_plan_bound_S_bound_thesis‘). [Abdulaziz et al., p.29]

definition *sspace-DAG* **where**
sspace-DAG PROB lss ≡ (
(*set lss* = *valid-states PROB*)
∧ (*top-sorted-abs* (λ*x y*. *y* ∈ *state-successors PROB x*) *lss*)

)

lemma *problem-plan-bound-S-bound-2nd-step-thesis*:
assumes *finite* (*PROB* :: *'a problem*) (*sspace-DAG* (*prob-proj PROB vs*) *lss*)
(*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (∃ *as'*. (*exec-plan s as'* = *exec-plan s as*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *S vs lss PROB* (*fmrestrict-set vs s*))
)
⟨*proof*⟩

And finally, this is the main lemma about the upper bounding algorithm.

theorem *problem-plan-bound-S-bound-thesis*:
assumes *finite* (*PROB* :: *'a problem*) (*sspace-DAG* (*prob-proj PROB vs*) *lss*)
shows (
problem-plan-bound PROB
≤ *Sup* {*S vs lss PROB s'* | *s'*. *s'* ∈ *valid-states* (*prob-proj PROB vs*)}
)
⟨*proof*⟩

end

References

[1] M. Abdulaziz, C. Gretton, and M. Norrish. A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2017.

[2] M. Abdulaziz, M. Norrish, and C. Gretton. Formally verified algorithms for upper-bounding state space diameters. *Journal of Automated Reasoning*, pages 1–36, 2018.