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* = ($\lambda x.$ *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 ++ *m2* ≡ *m2* $++_f$ *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. δ-*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-dom prob $\equiv \bigcup$ $((\lambda\ (s1,\ s2).\ action\text{-}dom\ s1\ s2)$ ' $prob)$

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

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

definition *state-succ* **where**
  state-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*
  by *simp*

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*)
  using *assms* **proof** (*induction l2 arbitrary*: *l1 l*)
  case *Nil*
  then have ¬(*length* [] > *length l1*)
    by *simp*

```
  then show ?case
    using Nil
    by blast
next
  case (Cons a l2)
  then show ?case proof(induction l1 arbitrary: l)
    case Nil
    then show ?case
      using Nil-prefix
      by blast
  next
    case (Cons b l1)
    then show ?case proof(cases l)
      case Nil
      then have ¬(prefix (a # l2) l)
        by simp
      then show ?thesis using Cons.prems(4)
        by simp
    next
      case (Cons c l)
      then have 1: length l2 > length l1
        using Cons.prems(2)
        by fastforce
      then show ?thesis using Cons proof(cases l)
        case Nil
        then have l1 = [c] l2 = [c]
          using Cons.prems(3, 4) local.Cons 1
          by fastforce+
        then show ?thesis
          using 1
          by auto
      next
        case (Cons d l')
        {
          thm len-ge-0
          have length l1 ≥ 0
            by simp
          then have length l2 > 0
            using 1
            by force
          then have l2 ≠ [] using 1
            by blast
        }
        then have length (a # l1) ≤ length (b # l2)
          using 1 le-eq-less-or-eq
          by simp
        then show ?thesis
          using Cons.prems(3, 4) prefix-length-prefix
          by fastforce
```

```
      qed
    qed
  qed
qed

lemma nempty-list-append-length-add:
  fixes l1 l2 l3
  assumes l2 ≠ []
  shows length (l1 @ l3) < length (l1 @ l2 @l3)
  using assms
  by (induction l2) auto


lemma append-filter:
  fixes f1 :: 'a ⇒ bool and f2 as1 as2 and p :: 'a list
  assumes (as1 @ as2 = filter f1 (map f2 p))
  shows (∃ p-1 p-2.
    (p-1 @ p-2 = p)
    ∧ (as1 = filter f1 (map f2 p-1))
    ∧ (as2 = filter f1 (map f2 p-2))
  )
  using assms
proof (induction p arbitrary: f1 f2 as1 as2)
  case Nil
  from Nil have 1: as1 @ as2 = []
    by force
  then have 2: as1 = [] as2 = []
    by blast+
  let ?p1=[]
  let ?p2=[]
  from 1 2
  have ?p1 @ ?p2 = [] as1 = (filter f1 (map f2 ?p1)) as2 = (filter f1 (map f2
?p2))
    subgoal by blast
    subgoal using 2(1) by simp
    subgoal using 2(2) by simp
    done
  then show ?case
    by fast
next
  case cons: (Cons a p)
  then show ?case
  proof (cases as1)
    case Nil
    from cons.prems Nil
    have 1: as2 = filter f1 (map f2 (a # p))
      by simp
    let ?p1=[]
    let ?p2=a # p
```

have *?p1* @ *?p2* = *a* # *p as1* = *filter f1* (*map f2 ?p1*) *as2* = *filter f1* (*map f2 ?p2*)
subgoal by *simp*
subgoal using *Nil* **by** *simp*
subgoal using *1* **by** *auto*
done
then show *?thesis*
by *blast*
next
case (*Cons a′ p′*)
then show *?thesis*
proof (*cases* ¬*f1* (*f2 a*))
case *True*
hence *filter f1* (*map f2* (*a* # *p*)) = *filter f1* (*map f2 p*)
by *fastforce*
hence *as1* @ *as2* = *filter f1* (*map f2 p*)
using *cons.prems*
by *argo*
then obtain *p1 p2* **where** *a*:
p1 @ *p2* = *p as1* = *filter f1* (*map f2 p1*) *as2* = *filter f1* (*map f2 p2*)
using *cons.IH*
by *meson*
let *?p1*=*a* # *p1*
let *?p2*=*p2*
have *?p1* @ *?p2* = *a* # *p as1* = *filter f1* (*map f2 ?p1*) *as2* = *filter f1* (*map f2 ?p2*)
subgoal using *a*(*1*) **by** *fastforce*
subgoal using *True a*(*2*) **by** *auto*
subgoal using *a*(*3*) **by** *blast*
done
then show *?thesis*
by *blast*
next
case *False*
hence *filter f1* (*map f2* (*a* # *p*)) = *f2 a* # *filter f1* (*map f2 p*)
by *fastforce*
then have *1*: *a′* = *f2 a p′* @ *as2* = *filter f1* (*map f2 p*) *as1* = *a′* # *p′*
using *cons.prems Cons*
by *fastforce+*
then obtain *p1 p2* **where** *2*:
p1 @ *p2* = *p p′* = *filter f1* (*map f2 p1*) *as2* = *filter f1* (*map f2 p2*)
using *cons.IH*
by *meson*
let *?p1*=*a* # *p1*
let *?p2*=*p2*
have *?p1* @ *?p2* = *a* # *p as1* = *filter f1* (*map f2 ?p1*) *as2* = *filter f1* (*map f2 ?p2*)
subgoal using *2*(*1*) **by** *simp*
subgoal using *False 1*(*1*, *3*) *2*(*2*) **by** *force*

```
      subgoal using 2(3) by blast
      done
    then show ?thesis
      by blast
  qed
 qed
qed
```

— NOTE types of 'f1' and 'p' had to be fixed for 'append_eq_as_proj_1'.

```
lemma append-eq-as-proj-1:
 fixes f1 :: 'a ⇒ bool and f2 as1 as2 as3 and p :: 'a list
 assumes (as1 @ as2 @ as3 = filter f1 (map f2 p))
 shows (∃ p-1 p-2 p-3.
  (p-1 @ p-2 @ p-3 = p)
  ∧ (as1 = filter f1 (map f2 p-1))
  ∧ (as2 = filter f1 (map f2 p-2))
  ∧ (as3 = filter f1 (map f2 p-3))
 )
proof −
 from assms
 obtain p-1 p-2 where 1: (p-1 @ p-2 = p) (as1 = filter f1 (map f2 p-1))
  (as2 @ as3 = filter f1 (map f2 p-2))
  using append-filter[of as1 (as2 @ as3)]
  by meson
 moreover from 1
 obtain p-a p-b where (p-a @ p-b = p-2) (as2 = filter f1 (map f2 p-a))
  (as3 = filter f1 (map f2 p-b))
  using append-filter[where p=p-2]
  by meson
 ultimately show ?thesis
  by blast
qed

lemma filter-empty-every-not: ⋀P l. (filter (λx. P x) l = []) = list-all (λx. ¬P x)
l
proof −
 fix P l
 show (filter (λx. P x) l = []) = list-all (λx. ¬P x) l
  apply(induction l)
   apply(auto)
  done
qed
```

— NOTE added lemma (listScript.sml:810).

```
lemma MEM-SPLIT:
 fixes x l
 assumes ¬ListMem x l
 shows ∀ l1 l2. l ≠ l1 @ [x] @ l2
proof −
```

```
{
  assume C: ¬(∀ l1 l2. l ≠ l1 @ [x] @ l2)
  then have ∃ l1 l2. l = l1 @ [x] @ l2
    by blast
  then obtain l1 l2 where 1: l = l1 @ [x] @ l2
    by blast
  from assms
  have 2: (∀ xs. l ≠ x # xs) ∧ (∀ xs. (∀ y. l ≠ y # xs) ∨ ¬ ListMem x xs)
    using ListMem-iff
    by fastforce
  then have False
  proof (cases l1)
    case Nil
    let ?xs=l2
    from 1 Nil have l = [x] @ ?xs
      by blast
    then show ?thesis
      using 2
      by simp
  next
    case (Cons a list)
    {
      let ?y=a
      let ?xs=list @ [x] @ l2
      from 1 Cons have l = ?y # ?xs
        by simp
      moreover have ListMem x ?xs
        by (simp add: ListMem-iff)
      ultimately have ∃ xs. ∃ y. l = y # xs ∧ ListMem x xs
        by blast
      then have ¬(∀ xs. (∀ y. l ≠ y # xs) ∨ ¬ ListMem x xs)
        by presburger
    }
    then show ?thesis
      using 2
      by auto
  qed
}
then show ?thesis
  by blast
qed


— NOTE added lemma (listScript.sml:2784)
lemma APPEND-EQ-APPEND-MID:
  fixes l1 l2 m1 m2 e
  shows
    (l1 @ [e] @ l2 = m1 @ m2)
      ⟷
```

```
      (∃l. (m1 = l1 @ [e] @ l) ∧ (l2 = l @ m2)) ∨
      (∃l. (l1 = m1 @ l) ∧ (m2 = l @ [e] @ l2))
proof (induction l1 arbitrary: m1)
  case Nil
  then show ?case
    by (simp; metis Cons-eq-append-conv)+
next
  case (Cons a l1)
  then show ?case
    by (cases m1; simp; blast)
qed

— NOTE variable ‘P‘ was removed (redundant).
lemma LIST-FRAG-DICHOTOMY:
  fixes l la x lb
  assumes sublist l (la @ [x] @ lb) ¬ListMem x l
  shows sublist l la ∨ sublist l lb
proof –
  {
    from assms(1)
    obtain pfx sfx where 1: pfx @ l @ sfx = la @ [x] @ lb
      unfolding sublist-def
      by force
    from assms(2)
    have 2: ∀l1 l2. l ≠ l1 @ [x] @ l2
      using MEM-SPLIT[OF assms(2)]
      by blast
    from 1
    consider (a) (∃lc. pfx = la @ [x] @ lc ∧ lb = lc @ l @ sfx)
      | (b) (∃lc. la = pfx @ lc ∧ l @ sfx = lc @ [x] @ lb)
      using APPEND-EQ-APPEND-MID[of la x lb pfx l @ sfx]
      by presburger
    then have ∃pfx' sfx. (pfx' @ l @ sfx = la) ∨ (pfx'@ l @ sfx = lb)
    proof (cases)
      case a
        — NOTE ‘lc‘ is ‘l'‘ in original proof.
      then obtain lc where a: pfx = la @ [x] @ lc lb = lc @ l @ sfx
        by blast
      then show ?thesis
        by blast
    next
      case b
      then obtain lc where i: la = pfx @ lc l @ sfx = lc @ [x] @ lb
        by blast
      then show ?thesis
        using 2
        by (metis APPEND-EQ-APPEND-MID)
    qed
  }
```

```
  then show ?thesis
    unfolding sublist-def
    by blast
qed


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 −
  {
    assume ¬P x list-all P l
    then have ¬ListMem x l
    proof (induction l arbitrary: x P)
      case Nil
      then show ?case
        using ListMem-iff
        by force
    next
      case (Cons a l)
      {
        have list-all P l
          using Cons.prems(2)
          by simp
        then have ¬ListMem x l
          using Cons.prems(1) Cons.IH
          by blast
      }
      moreover {
        have P a
          using Cons.prems(2)
          by simp
        then  have a ≠ x
          using Cons.prems(1)
          by meson
      }
      ultimately show ?case
        using Cons.prems(1, 2) ListMem-iff list.pred-set
        by metis
    qed
  }
  then have ¬ListMem x l
    using assms(2, 3)
    by fast
  then show ?thesis
    using assms(1) LIST-FRAG-DICHOTOMY
    by metis
qed
```

```
lemma frag-len-filter-le:
  fixes P l′ l
  assumes sublist l′ l
  shows length (filter P l′) ≤ length (filter P l)
proof −
  obtain ps ss where l = ps @ l′ @ ss
    using assms sublist-def
    by blast
  then have 1:
    length (filter P l) = length (filter P ps) + length (filter P l′) + length (filter P
ss)
    by force
  then have length (filter P ps) ≥ 0 length (filter P ss) ≥ 0
    by blast+
  then show ?thesis
    using 1
    by linarith
qed

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 –
  have *subseq l1 l2* = *list-emb* (λ*x y*. *x* = *y*) *l1 l2*
    by *blast*
  moreover {
    have *sublist-HOL4 l1 l2* ⟷ *list-emb* (λ*x y*. *x* = *y*) *l1 l2*
    proof (*induction rule*: *sublist-HOL4.induct*)
      case (*3 x l1 y l2*)
      then show *sublist-HOL4* (*x* # *l1*) (*y* # *l2*) ⟷ *list-emb* (λ*x y*. *x* = *y*) (*x* # *l1*) (*y* # *l2*)
      proof (*cases x* = *y*)
        case *True*
        then show *?thesis*
          using *3.IH*(*1*, *2*)
          by (*metis sublist-HOL4.simps*(*3*) *subseq-Cons′ subseq-Cons2-iff*)
      next
        case *False*
        then show *?thesis*
          using *3.IH*(*2*)
          by *force*
      qed
    qed *simp+*
  }
  ultimately show *?thesis*
    by *blast*
qed

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*
unfolding *list-frag-HOL4-def sublist-def*
by *blast*

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*
by *auto*

lemma *sublist-refl*: *subseq l l*
by *auto*

lemma *sublist-cons*:
assumes *subseq l1 l2*
shows *subseq l1* (*h* # *l2*)
using *assms*
by *blast*

lemma *sublist-NIL*: *subseq l1* [] = (*l1* = [])
by *fastforce*

lemma *sublist-trans*:
fixes *l1 l2*
assumes *subseq l1 l2 subseq l2 l3*
shows *subseq l1 l3*
using *assms*
by *force*

— NOTE can be solved directly with 'list_emb_length'.
lemma *sublist-length*:
fixes *l l′*
assumes *subseq l l′*
shows *length l* ≤ *length l′*
using *assms list-emb-length*
by *blast*

— NOTE can be solved directly with subseq_Cons'.

lemma *sublist-CONS1-E*:
  fixes *l1 l2*
  assumes *subseq* (*h* # *l1*) *l2*
  shows *subseq l1 l2*
  using *assms subseq-Cons′*
  by *metis*

lemma *sublist-equal-lengths*:
  fixes *l1 l2*
  assumes *subseq l1 l2* (*length l1* = *length l2*)
  shows (*l1* = *l2*)
  using *assms subseq-same-length*
  by *blast*

— NOTE can be solved directly with 'subseq_order.antisym'.
lemma *sublist-antisym*:
  assumes *subseq l1 l2 subseq l2 l1*
  shows (*l1* = *l2*)
  using *assms subseq-order.antisym*
  by *blast*

lemma *sublist-append-back*:
  fixes *l1 l2*
  shows *subseq l1* (*l2* @ *l1*)
  by *blast*

— NOTE can be solved directly with 'subseq_rev_drop_many'.
lemma *sublist-snoc*:
  fixes *l1 l2*
  assumes *subseq l1 l2*
  shows *subseq l1* (*l2* @ [*h*])
  using *assms subseq-rev-drop-many*
  by *blast*

lemma *sublist-append-front*:
  fixes *l1 l2*
  shows *subseq l1* (*l1* @ *l2*)
  by *fast*

lemma *append-sublist-1*:
  assumes *subseq* (*l1* @ *l2*) *l*
  shows *subseq l1 l* ∧ *subseq l2 l*
  using *assms sublist-append-back sublist-append-front sublist-trans*

```
  by blast

— NOTE added lemma (eventually wasn't needed in the remaining proofs).
lemma sublist-prefix:
  shows subseq (h # l1) l2 ⟹ ∃ l2a l2b. l2 = l2a @ [h] @ l2b ∧ ¬ListMem h l2a
proof (induction l2 arbitrary: h l1)
  — NOTE l2 cannot be empty when h # l1 isn't.
  case Nil
  have ¬(subseq (h # l1) [])
    by simp
  then show ?case
    using Nil.prems
    by blast
next
  case (Cons a l2)
  then show ?case proof (cases a = h)
    — NOTE If a = h then a trivial solution exists in l2a = [] and l2b = l2.
    case True
    then show ∃ l2a l2b. (Cons a l2) = l2a @ [h] @ l2b ∧ ¬ListMem h l2a
      using ListMem-iff
      by force
  next
    case False
    have subseq (h # l1) l2
      using Cons.prems False subseq-Cons2-neq
      by force
    then obtain l2a l2b where l2 = l2a @ [h] @ l2b ¬ListMem h l2a
      using Cons.IH Cons.prems
      by meson
    moreover have a # l2 = (a # l2a) @ [h] @ l2b
      using calculation(1)
      by simp
    moreover have ¬(ListMem h (a # l2a))
      using False calculation(2) ListMem.simps
      by fastforce
    ultimately show ?thesis
      by blast
  qed
qed

— NOTE added lemma (eventually wasn't needed in the remaining proofs).
lemma sublist-skip:
  fixes l1 l2 h l1'
  assumes l1 = (h # l1') l2 = l2a @ [h] @ l2b subseq l1 l2 ¬(ListMem h l2a)
  shows subseq l1 (h # l2b)
  using assms
proof (induction l2a arbitrary: l1 l2 h l1')
  case Nil
```

```
  then have l2 = h # l2b
    by fastforce
  then show ?case using Nil.prems(3)
    by blast
next
  case (Cons a l2a)
  have a ≠ h
    using Cons.prems(4) ListMem.simps
    by fast
  then have subseq l1 (l2a @ [h] @ l2b)
    using Cons.prems(1, 2, 3) subseq-Cons2-neq
    by force
  moreover have ¬ListMem h l2a
    using Cons.prems(4) insert
    by metis
  ultimately have subseq l1 (h # l2b)
    using Cons.IH Cons.prems
    by meson
  then show ?case
    by simp
qed
```

— NOTE added lemma (eventually wasn't needed in the remaining proofs).

```
lemma sublist-split-trans:
  fixes l1 l2 h l1'
  assumes l1 = (h # l1') l2 = l2a @ [h] @ l2b subseq l1 l2 ¬(ListMem h l2a)
  shows subseq l1' l2b
proof −
  have subseq (h # l1') (h # l2b)
    using assms sublist-skip
    by metis
  then show ?thesis
    using subseq-Cons2'
    by metis
qed

lemma sublist-cons-exists:
  shows
    subseq (h # l1) l2
    ⟷ (∃ l2a l2b. (l2 = l2a @ [h] @ l2b) ∧ ¬ListMem h l2a ∧ subseq l1 l2b)

proof −
```

— NOTE show both directions of the equivalence in pure proof blocks.

```
  {
    have
      subseq (h # l1) l2 ⟹ (∃ l2a l2b. (l2 = l2a @ [h] @ l2b) ∧ ¬ListMem h l2a
∧ subseq l1 l2b)
    proof (induction l2 arbitrary: h l1)
      case (Cons a l2)
```

```
      show ?case
      proof (cases a = h)
        case True
          — NOTE This case has a trivial solution in '?l2a = []', '?l2b = l2'.
        let ?l2a=[]
        have (a # l2) = ?l2a @ [h] @ l2
          using True
          by auto
        moreover have ¬(ListMem h ?l2a)
          using ListMem-iff
          by force
        moreover have subseq l1 l2
          using Cons.prems True
          by simp
        ultimately show ?thesis
          by blast
      next
        case False
        have 1: subseq (h # l1) l2
          using Cons.prems False subseq-Cons2-neq
          by metis
        then obtain l2a l2b where l2 = l2a @ [h] @ l2b ¬ListMem h l2a
          using Cons.IH Cons.prems
          by meson
        moreover have a # l2 = (a # l2a) @ [h] @ l2b
          using calculation(1)
          by simp
        moreover have ¬(ListMem h (a # l2a))
          using False calculation(2) ListMem.simps
          by fastforce
        ultimately show ?thesis
          using 1 sublist-split-trans
          by metis
      qed
    qed simp
  }
  moreover
  {
    assume ∃ l2a l2b. (l2 = l2a @ [h] @ l2b) ∧ ¬ListMem h l2a ∧ subseq l1 l2b
    then have  subseq (h # l1) l2
      by auto
  }
  ultimately show ?thesis
    by argo
qed


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*
using *list-emb-appendD*
by *fast*

— NOTE can be solved directly with 'list_emb_append_mono'.
lemma *sublist-append-both-I*:
assumes *subseq a b subseq c d*
shows *subseq* (*a* @ *c*) (*b* @ *d*)
using *assms list-emb-append-mono*
by *blast*

lemma *sublist-append*:
assumes *subseq l1 l1′ subseq l2 l2′*
shows *subseq* (*l1* @ *l2*) (*l1′* @ *l2′*)
using *assms sublist-append-both-I*
by *blast*

lemma *sublist-append2*:
assumes *subseq l1 l2*
shows *subseq l1* (*l2* @ *l3*)
using *assms sublist-append*[*of l1 l2* [] *l3*]
by *fast*

lemma *append-sublist*:
shows *subseq* (*l1* @ *l2* @ *l3*) *l* $\Longrightarrow$ *subseq* (*l1* @ *l3*) *l*
proof (*induction l*)
case *Nil*
then show *?case*
using *sublist-NIL*
by *fastforce*
next
case (*Cons a l*)
then show *?case*
proof (*cases l1*)
case *Nil*
then show *?thesis*
using *Cons.prems append-sublist-1*
by *auto*
next
case (*Cons a list*)
then show *?thesis*
using *Cons.prems subseq-append′ subseq-order.dual-order.trans*
by *blast*
qed

qed

lemma *sublist-subset*:
assumes *subseq l1 l2*
shows *set l1* ⊆ *set l2*
using *assms set-nths-subset subseq-conv-nths*
by *metis*

lemma *sublist-filter*:
fixes *P l*
shows *subseq* (*filter P l*) *l*
using *subseq-filter-left*
by *blast*

lemma *sublist-cons-2*:
fixes *l1 l2 h*
shows (*subseq* (*h* # *l1*) (*h* # *l2*) ⟷ (*subseq l1 l2*))
by *fastforce*

lemma *sublist-every*:
fixes *l1 l2 P*
assumes (*subseq l1 l2* ∧ *list-all P l2*)
shows *list-all P l1*
by (*metis* (*full-types*) *Ball-set assms list-emb-set*)

lemma *sublist-SING-MEM*: *subseq* [*h*] *l* ⟷ *ListMem h l*
using *ListMem-iff subseq-singleton-left*
by *metis*

— NOTE renamed due to previous declaration of ‘sublist_append_exists_2.
lemma *sublist-append-exists-2*:
fixes *l1 l2 l3*
assumes *subseq* (*h* # *l1*) *l2*
shows (∃ *l3 l4*. (*l2* = *l3* @ [*h*] @ *l4*) ∧ (*subseq l1 l4*))
using *assms sublist-cons-exists*
by *metis*

lemma *sublist-append-4*:
fixes *l l1 l2 h*
assumes (*subseq* (*h* # *l*) (*l1* @ [*h*] @ *l2*)) (*list-all* (λ*x*. ¬(*h* = *x*)) *l1*)
shows *subseq l l2*
using *assms*

proof (*induction l1*)
qed *auto*

lemma *sublist-append-5*:
  fixes *l l1 l2 h*
  assumes (*subseq* (*h* # *l*) (*l1* @ *l2*)) (*list-all* (λ*x*. ¬(*h* = *x*)) *l1*)
  shows *subseq* (*h* # *l*) *l2*
  using *assms*
proof (*induction l1*)
qed *auto*

lemma *sublist-append-6*:
  fixes *l l1 l2 h*
  assumes (*subseq* (*h* # *l*) (*l1* @ *l2*)) (¬(*ListMem h l1*))
  shows *subseq* (*h* # *l*) *l2*
  using *assms*
proof (*induction l1*)
  case (*Cons a l1*)
  then show *?case*
    by (*simp add*: *ListMem-iff*)
qed *simp*

lemma *sublist-MEM*:
  fixes *h l1 l2*
  shows *subseq* (*h* # *l1*) *l2* ⟹ *ListMem h l2*
proof (*induction l2*)
next
  case (*Cons a l2*)
  then show *?case*
    using *elem insert subseq-Cons2-neq*
    by *metis*
qed *simp*

lemma *sublist-cons-4*:
  fixes *l h l′*
  shows *subseq l l′* ⟹ *subseq l* (*h* # *l′*)
  using *sublist-cons*
  by *blast*

2.2 Main Theorems

theorem *sublist-imp-len-filter-le*:
  fixes *P l l′*
  assumes *subseq l′ l*
  shows *length* (*filter P l′*) ≤ *length* (*filter P l*)

using *assms*
by (*simp add*: *sublist-length*)

— TODO showcase (non-trivial proof translation/ obscurity).
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*)
)
using *assms*
proof (*induction filter* (λ*x*. ¬*P1 x*) *l arbitrary*: *P1 P2 P3 k1 f PProbs PProbl s l*)
case *Nil*
then have *list-all* (λ*x*. *P1 x*) *l*
using *Nil*(*1*) *filter-empty-every-not*[*of* λ*x*. ¬*P1 x l*]
by *presburger*
then obtain *l'* **where** *1*:
(*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
using *Nil.prems*(*1*, *2*, *3*)
by *blast*
moreover {
fix *l''*
assume *sublist l'' l' list-all P1 l''*
then have *subseq l'' l'*

by *blast*

— NOTE original proof uses 'frag_len_filter_le' which however requires the fact 'sublist l' ?l'. Unfortunately, this could not be derived in Isabelle/HOL.

then have $length\ (filter\ P2\ l'') \leq length\ (filter\ P2\ l')$
using *sublist-imp-len-filter-le*
by *blast*
then have $length\ (filter\ P2\ l'') \leq k1$
using *1*
by *linarith*
}
ultimately show *?case*
by *blast*
next
case (*Cons a x*)

— NOTE The proof of the induction step basically consists of construction a list '?l'=l" @ [a] @ l"'' where 'l"' and 'l"'' are lists obtained from certain specifications of the induction hypothesis.

then obtain *l1 l2* **where** *2*:
$l = l1\ @\ a\ \#\ l2\ (\forall u \in set\ l1.\ P1\ u)\ \neg\ P1\ a \wedge x = [x \leftarrow l2\ .\ \neg\ P1\ x]$
using *Cons*(*2*) *filter-eq-Cons-iff*[*of* $\lambda x.\ \neg P1\ x$]
by *metis*
then have *3*: *PProbl l2*
using *Cons.prems*(*2*, *6*) *2*(*1*) *sublist-append-back*
by *blast*

— NOTE Use the induction hypothesis to obtain a specific 'l"''.

{
have $x = filter\ (\lambda x.\ \neg P1\ x)\ l2$
using *2*(*3*)
by *blast*
moreover have *PProbs* (*f* (*f s l1*) [*a*])
using *Cons.prems*(*1*, *2*, *5*, *6*, *7*) *2*(*1*) *elem sublist-SING-MEM*
by *metis*
moreover have $\forall l\ s.\ PProbs\ s \wedge PProbl\ l \wedge list\text{-}all\ P1\ l \longrightarrow (\exists l'.$
$f\ s\ l' = f\ s\ l \wedge length\ (filter\ P2\ l') \leq k1 \wedge length\ (filter\ P3\ l') \leq length\ (filter\ P3\ l)$
$\wedge\ list\text{-}all\ P1\ l' \wedge subseq\ l'\ l)$
using *Cons.prems*(*3*)
by *blast*
moreover have $\forall s\ l1\ l2.\ f\ (f\ s\ l1)\ l2 = f\ s\ (l1\ @\ l2)$
$\forall s\ l.\ PProbs\ s \wedge PProbl\ l \longrightarrow PProbs\ (f\ s\ l)$
$\forall l1\ l2.\ subseq\ l1\ l2 \wedge PProbl\ l2 \longrightarrow PProbl\ l1$
$\forall l1\ l2.\ PProbl\ (l1\ @\ l2) = (PProbl\ l1 \wedge PProbl\ l2)$
using *Cons.prems*(*4*, *5*, *6*, *7*)
by *blast+*
ultimately have $\exists l'.$
$f\ (f\ (f\ s\ l1)\ [a])\ l' = f\ (f\ (f\ s\ l1)\ [a])\ l2 \wedge length\ (filter\ P3\ l') \leq length\ (filter\ P3\ l2)$
$\wedge\ (\forall l''.\ sublist\ l''\ l' \wedge list\text{-}all\ P1\ l'' \longrightarrow length\ (filter\ P2\ l'') \leq k1) \wedge subseq\ l'\ l2$

using *3 Cons*(*1*)[*of P1 l2*, **where** *s*=(*f* (*f s l1*) [*a*])]
by *blast*
}
then obtain l''' **where** *4*:
f (*f* (*f s l1*) [*a*]) l''' = *f* (*f* (*f s l1*) [*a*]) *l2*
length (*filter P3* l''') ≤ *length* (*filter P3 l2*)
(∀ l''. *sublist* l'' l''' ∧ *list-all P1* l'' ⟶ *length* (*filter P2* l'') ≤ *k1*) ∧ *subseq* l''' *l2*
by *blast*
then have *f s* (*l1* @ [*a*] @ l''') = *f s* (*l1* @ [*a*] @ *l2*)
using *Cons.prems*(*4*)
by *auto*
then have *subseq* l''' *l2*
using *4*(*3*)
by *blast*
— NOTE Use the induction hypothesis to obtain a specific 'l"'.
{
have ∀ *l s*.
PProbs s ∧ *PProbl l1* ∧ *list-all P1 l1*
⟶ (∃ l''.
f s l'' = *f s l1* ∧ *length* (*filter P2* l'') ≤ *k1* ∧ *length* (*filter P3* l'') ≤ *length* (*filter P3 l1*)
∧ *list-all P1* l'' ∧ *subseq* l'' *l1*)
using *Cons.prems*(*3*)
by *blast*
then have ∃ l''.
f s l'' = *f s l1* ∧ *length* (*filter P2* l'') ≤ *k1* ∧ *length* (*filter P3* l'') ≤ *length* (*filter P3 l1*)
∧ *list-all P1* l'' ∧ *subseq* l'' *l1*
using *Cons.prems*(*1*, *2*, *7*) *2*(*1*, *2*)
by (*metis Ball-set*)
}

then obtain l'' **where** *5*:
f s l'' = *f s l1 length* (*filter P2* l'') ≤ *k1*
length (*filter P3* l'') ≤ *length* (*filter P3 l1*) *list-all P1* l'' ∧ *subseq* l'' *l1*
by *blast*

Proof the proposition by providing the witness $l' = l''$ @ [*a*] @ l'''.

let *?l'*=(l'' @ [*a*] @ l''')
{
have ∀ *s l1 l2*. *f* (*f s l1*) *l2* = *f s* (*l1* @ *l2*)
by (*simp add*: *Cons.prems*(*4*))

Rewrite and show the goal.

have *f s ?l'* = *f s* (*l1* @ [*a*] @ *l2*) ⟷ *f s* (l'' @ (*a* # l''')) = *f s* (*l1* @ (*a* # *l2*))
by *simp*
also have ... ⟷ *f* (*f* (*f s l1*) [*a*]) l''' = *f* (*f* (*f s l1*) [*a*]) *l2*

```
    by (metis Cons.prems(4) ‹f s l'' = f s l1› calculation)
  finally have f s ?l' = f s (l1 @ [a] @ l2)
    using 4(1)
    by blast
}
moreover
{
  have
    length (filter P3 ?l') ≤ length (filter P3 (l1 @ [a] @ l2))
    ⟷
      (length (filter P3 l'') + 1 + length (filter P3 l''')
      ≤ length (filter P3 l1) + 1 + length (filter P3 l2))
    by force
  then have
    length (filter P3 ?l') ≤ length (filter P3 (l1 @ [a] @ l2))
    ⟷
      length (filter P3 l'') + length (filter P3 l''')
      ≤ length (filter P3 l1) + length (filter P3 l2)
    by linarith
  then have length (filter P3 ?l') ≤ length (filter P3 (l1 @ [a] @ l2))
    using 4(2) ‹length (filter P3 l'') ≤ length (filter P3 l1)›
      add-mono-thms-linordered-semiring(1)
    by blast
}
moreover
{
  fix l''''
  assume P: sublist l'''' ?l' list-all P1 l''''
  have list-all P1 l1
    using 2(2) Ball-set
    by blast
  consider (i) sublist l'''' l'' | (ii) sublist l'''' l'''
    using P(1, 2) 2(3)  LIST-FRAG-DICHOTOMY-2
    by metis
  then have length (filter P2 l'''') ≤ k1
  proof (cases)
    case i
    then have length (filter P2 l'''') ≤ length (filter P2 l'')
      using frag-len-filter-le
      by blast
    then show ?thesis
      using 5(2) order-trans
      by blast
  next
    case ii
    then show ?thesis
      using 4(3) P(2)
      by blast
  qed
```

}
— NOTE the following two steps seem to be necessary to convince Isabelle that the split $l = l1\ @\ a\ \#\ l2$ matches the split '(l1 @ [a] @ l2' and the previous proof steps therefore is prove the goal.
moreover {
have *subseq ?l′* (*l1* @ [*a*] @ *l2*)
by (*simp add*: *FSSublist.sublist-append* ‹*list-all P1 l″* ∧ *subseq l″ l1*› ‹*subseq l‴ l2*›)
}
moreover have $l = l1\ @\ [a]\ @\ l2$
using *2*
by *force*
ultimately show *?case*
by *blast*
qed

lemma *isPREFIX-sublist*:
fixes *x y*
assumes *prefix x y*
shows *subseq x y*
using *assms prefix-order.dual-order.antisym*
by *blast*

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−
have $\forall p.\ (n = f\ p) \wedge P\ p \longrightarrow (\exists p'.\ P\ p' \wedge f\ p' \leq l)$ **for** *n*
apply(*rule Nat.nat-less-induct*[**where** $?P = \%n.\ \forall p.\ (n = f\ p) \wedge P\ p \longrightarrow (\exists p'.\ P\ p' \wedge f\ p' \leq l)$])
by (*metis assms not-less*)
then show *?thesis* **by** *auto*
qed

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-set* (*fdom* − *vs*) *f*)
  using *assms*
  by (*metis DiffI Int-def Int-iff Set.filter-def fmdom′-filter fmfilter-alt-defs*(*4*) *inf.order-iff*)

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

proof −
  fix *vs s x*
  assume *P*: *disjnt* (*fmdom′ x*) *vs*
  moreover have *1*: $\forall x''$. ($x'' \in vs$) $\longrightarrow$ (*fmlookup* (*x* ++ *s*) x'' = *fmlookup s* x'')
    by (*metis calculation disjnt-iff fmap-add-ltr-def fmdom′-notD fmdom-notI fmlookup-add*)
  moreover
  {
    fix x''
    have *fmlookup* (*fmrestrict-set vs s*) x'' = *fmlookup* (*fmrestrict-set vs* (*x* ++ *s*)) x''
      apply(*cases* $x'' \notin$ *fmdom′ x*)
       apply(*cases* $x'' \notin vs$)
        apply(*auto simp add*: *1*)
      done
  }
  ultimately show (*fmrestrict-set vs s* = *fmrestrict-set vs* (*x* ++ *s*))
    using *fmap-ext* **by** *blast*
qed

— TODO refactor into 'FmapUtils.thy'.
lemma *graph-plan-card-state-set*:
  fixes *PROB vs*
  assumes *finite vs*
  shows *card* (*fmdom′* (*fmrestrict-set vs s*)) $\leq$ *card vs*
proof −
  let *?vs′* = *fmdom′* (*fmrestrict-set vs s*)
  have *?vs′* $\subseteq$ *vs*
    using *fmdom′-restrict-set*
    by *metis*
  moreover have *card ?vs′* $\leq$ *card vs*

using *assms calculation card-mono*
by *blast*
ultimately show *?thesis* **by** *blast*
qed

lemma *exec-drest-5*:
fixes *x vs*
assumes $fmdom'\ x \subseteq vs$
shows $(fmrestrict\text{-}set\ vs\ x = x)$
proof –
— TODO refactor and make into ISAR proof.
{
fix *v*
have $fmlookup\ (fmrestrict\text{-}set\ vs\ x)\ v = fmlookup\ x\ v$
apply$(cases\ v \in fmdom'\ x)$
subgoal using *assms* **by** *auto*
subgoal by $(simp\ add: fmdom'\text{-}notD)$
done
then have $fmlookup\ (fmrestrict\text{-}set\ vs\ x)\ v = fmlookup\ x\ v$
by *fast*
}
moreover have $fmlookup\ (fmrestrict\text{-}set\ vs\ x) = fmlookup\ x$
using *calculation fmap-ext*
by *auto*
ultimately show *?thesis*
using *fmlookup-inject*
by *blast*
qed

lemma *graph-plan-lemma-5*:
fixes $s\ s'\ vs$
assumes $(fmrestrict\text{-}set\ (fmdom'\ s - vs)\ s = fmrestrict\text{-}set\ (fmdom'\ s' - vs)\ s')$
$(fmrestrict\text{-}set\ vs\ s = fmrestrict\text{-}set\ vs\ s')$
shows $(s = s')$
proof –
have $\forall x.\ fmlookup\ s\ x = fmlookup\ s'\ x$
using $assms(1,\ 2)\ fmdom'\text{-}notD\ fminusI\ fmlookup\text{-}restrict\text{-}set\ Diff\text{-}iff$
by *metis*
then show *?thesis* **using** *fmap-ext*
by *blast*
qed

lemma *drest-smap-drest-smap-drest*:
fixes *x s vs*
shows $fmrestrict\text{-}set\ vs\ x \subseteq_f s \longleftrightarrow fmrestrict\text{-}set\ vs\ x \subseteq_f fmrestrict\text{-}set\ vs\ s$
proof –
— TODO this could be refactored into standalone lemma since it's very common in proofs.
have $1: fmlookup\ (fmrestrict\text{-}set\ vs\ s) \subseteq_m fmlookup\ s$

by (*metis fmdom′.rep-eq fmdom′-notI fmlookup-restrict-set map-le-def*)
moreover
{
assume *P1*: *fmrestrict-set vs x* $\subseteq_f$ *s*
moreover have *2*: *fmlookup* (*fmrestrict-set vs x*) $\subseteq_m$ *fmlookup s*
using *P1 fmsubset.rep-eq* **by** *blast*
{
fix *v*
assume $v \in$ *fmdom′* (*fmrestrict-set vs x*)
then have *fmlookup* (*fmrestrict-set vs x*) *v* = *fmlookup* (*fmrestrict-set vs s*) *v*
by (*metis* (*full-types*) *2 domIff fmdom′-notI fmlookup-restrict-set map-le-def*)
}
ultimately have *fmrestrict-set vs x* $\subseteq_f$ *fmrestrict-set vs s*
unfolding *fmsubset.rep-eq*
by (*simp add*: *map-le-def*)
}
moreover
{
assume *P2*: *fmrestrict-set vs x* $\subseteq_f$ *fmrestrict-set vs s*
moreover have *fmrestrict-set vs s* $\subseteq_f$ *s*
using *1 fmsubset.rep-eq*
by *blast*
ultimately have *fmrestrict-set vs x* $\subseteq_f$ *s*
using *fmsubset.rep-eq map-le-trans*
by *blast*
}
ultimately show *?thesis* **by** *blast*
qed

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* $\longleftrightarrow$ *fmrestrict-set vs x* $\subseteq_f$ *s′*
using *assms drest-smap-drest-smap-drest*
by *metis*

lemma *sat-precond-as-proj-4*:
fixes *fm1 fm2 vs*
assumes *fm2* $\subseteq_f$ *fm1*
shows (*fmrestrict-set vs fm2* $\subseteq_f$ *fm1*)
using *assms fmpred-restrict-set fmsubset-alt-def*
by *metis*

lemma *sublist-as-proj-eq-as-1*:
fixes *x s vs*
assumes ($x \subseteq_f$ *fmrestrict-set vs s*)
shows ($x \subseteq_f s$)
using *assms*
by (*meson fmsubset.rep-eq fmsubset-alt-def fmsubset-pred drest-smap-drest-smap-drest*

map-le-refl)

lemma *limited-dom-neq-restricted-neq*:
  assumes *fmdom′ f1* ⊆ *vs f1* ++ *f2* ≠ *f2*
  shows *fmrestrict-set vs* (*f1* ++ *f2*) ≠ *fmrestrict-set vs f2*
proof −
  {
    assume *C*: *fmrestrict-set vs* (*f1* ++ *f2*) = *fmrestrict-set vs f2*
    then have ∀ *x* ∈ *fmdom′* (*fmrestrict-set vs* (*f1* ++ *f2*)).
      fmlookup (*fmrestrict-set vs* (*f1* ++ *f2*)) *x*
      = *fmlookup* (*fmrestrict-set vs f2*) *x*
      by *simp*
    obtain *v* **where** *a*: *v* ∈ *fmdom′ f1 fmlookup* (*f1* ++ *f2*) *v* ≠ *fmlookup f2 v*
      using *assms*(*2*)
      by (*metis fmap-add-ltr-def fmap-ext fmdom′-notD fmdom-notI fmlookup-add*)
    then have *b*: *v* ∈ *vs*
      using *assms*(*1*)
      by *blast*
    moreover {
      have *fmdom′* (*fmrestrict-set vs* (*f1* ++ *f2*)) = *vs* ∩ *fmdom′* (*f1* ++ *f2*)
        by (*simp add*: *fmdom′-alt-def fmfilter-alt-defs*(*4*))
      then have *v* ∈ *fmdom′* (*fmrestrict-set vs* (*f1* ++ *f2*))
        using *C a b*
        by *fastforce*
    }
    then have *False*
      by (*metis C a*(*2*) *calculation fmlookup-restrict-set*)
  }
  then show *?thesis*
    by *auto*
qed

lemma *fmlookup-fmrestrict-set-dom*: ⋀*vs s*. *dom* (*fmlookup* (*fmrestrict-set vs s*))
= *vs* ∩ (*fmdom′ s*)
by (*auto simp add*: *fmdom′-restrict-set-precise*)

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)
by (*simp add*: *state-succ-def*)

— NOTE shortened to ’exec_plan’
— NOTE using ’fun’ because of multiple definining equations.
— NOTE first argument was curried.
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*
by (*induction as-a arbitrary*: *s as-b*) *auto*

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*))
using *assms exec-plan-Append*
by *metis*

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 −
  let ?A = {s. fmdom' s = {}}
  let ?B = {fmempty}
  fix s
  show ?thesis proof(rule ccontr)
    assume C: ?A ≠ ?B
    then show False proof −
      {
        assume C1: ?A ⊂ ?B
        have ?A = {} using C1 by force
        then have False using fmdom'-empty by blast
      }
      moreover
      {
        assume C2: ¬(?A ⊂ ?B)
        then have fmdom' fmempty = {}
          by auto
        moreover have fmempty ∈ ?A
          by auto
        moreover have ?A ≠ {}
          using calculation(2) by blast
        moreover have ∀ a∈?A.a∉?B
          by (metis (mono-tags, lifting)
                C Collect-cong calculation(1) fmrestrict-set-dom fmrestrict-set-null
singleton-conv)
        moreover have fmempty ∈ ?B by auto
        moreover have ∃ a∈?A.a∈?B
          by simp
        moreover have ¬(∀ a∈?A.a∉?B)
          by simp
        ultimately have False
          by blast
      }
      ultimately show False
        by fastforce
    qed
  qed
qed
```

— NOTE added lemma.

```
lemma possible-states-set-ii-a:
  fixes s x v
  assumes (v ∈ fmdom' s)
  shows (fmdom' ((λs. fmupd v x s) s) = fmdom' s)
  using assms insert-absorb
```

by *auto*

— NOTE added lemma.
lemma *possible-states-set-ii-b*:
fixes *s x v*
assumes ($v \notin fmdom'\ s$)
shows ($fmdom'\ ((\lambda s.\ fmupd\ v\ x\ s)\ s) = fmdom'\ s \cup \{v\}$)
by *auto*

— NOTE added lemma.
lemma *fmap-neq*:
fixes s :: ($'a$, *bool*) *fmap* **and** s' :: ($'a$, *bool*) *fmap*
assumes ($fmdom'\ s = fmdom'\ s'$)
shows ($(s \neq s') \longleftrightarrow (\exists\, v \in (fmdom'\ s).\ fmlookup\ s\ v \neq fmlookup\ s'\ v)$)
using *assms fmap-ext fmdom'-notD*
by *metis*

— NOTE added lemma.
lemma *fmdom'-fmsubset-restrict-set*:
fixes *X1 X2* **and** s :: ($'a$, *bool*) *fmap*
assumes $X1 \subseteq X2$ $fmdom'\ s = X2$
shows $fmdom'\ (fmrestrict\text{-}set\ X1\ s) = X1$
using *assms*
by (*metis* (*no-types*, *lifting*)
antisym-conv fmdom'-notD fmdom'-notI fmlookup-restrict-set rev-subsetD subsetI)

— NOTE added lemma.
lemma *fmsubset-restrict-set*:
fixes *X1 X2* **and** s :: *'a state*
assumes $X1 \subseteq X2$ $s \in \{s.\ fmdom'\ s = X2\}$
shows $fmrestrict\text{-}set\ X1\ s \in \{s.\ fmdom'\ s = X1\}$
using *assms fmdom'-fmsubset-restrict-set*
by *blast*

— NOTE added lemma.
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 –
— Show that domains of 's' and 'fmupd v x (fmrestrict_set X s)' are identical.
have *1*: $fmdom'\ s = insert\ v\ X$
using *assms*(*1*)
by *simp*
{
have $X \subseteq insert\ v\ X$
by *auto*

```
    then have fmdom' (fmrestrict-set X s) = X
      using 1 fmdom'-fmsubset-restrict-set
      by metis
    then have fmdom' (fmupd v x (fmrestrict-set X s)) = insert v X
      using assms(1) fmdom'-fmupd
      by auto
  }
  note 2 = this
  moreover
  {
    fix w
      — Show case for undefined variables (where lookup yields 'None').
    {
      assume w ∉ insert v X
      then have w ∉ fmdom' s w ∉ fmdom' (fmupd v x (fmrestrict-set X s))
        using 1 2
        by argo+
      then have fmlookup s w = fmlookup (fmupd v x (fmrestrict-set X s)) w
        using fmdom'-notD
        by metis
    }
      — Show case for defined variables (where lookup yields 'Some y').
    moreover {
      assume w ∈ insert v X
      then have w ∈ fmdom' s w ∈ fmdom' (fmupd v x (fmrestrict-set X s))
        using 1 2
        by argo+
      then have fmlookup s w = fmlookup (fmupd v x (fmrestrict-set X s)) w
        by (cases w = v)
          (auto simp add: assms calculation)
    }
    ultimately have fmlookup s w = fmlookup (fmupd v x (fmrestrict-set X s)) w
      by blast
  }
  then show ?thesis
    using fmap-ext
    by blast
qed

lemma construction-of-all-possible-states-lemma:
  fixes v X
  assumes (v ∉ X)
  shows ({s. fmdom' s = insert v X}
    = ((λs. fmupd v True s) ` {s. fmdom' s = X})
      ∪ ((λs. fmupd v False s) ` {s. fmdom' s = X})
  )
proof −
  fix v X
  let ?A = {s :: 'a state. fmdom' s = insert v X}
```

let $?B$ =
$((\lambda s.\ fmupd\ v\ True\ s)$ ‘ $\{s :: 'a\ state.\ fmdom'\ s = X\})$
$\cup\ ((\lambda s.\ fmupd\ v\ False\ s)$ ‘ $\{s :: 'a\ state.\ fmdom'\ s = X\})$

Show the goal by mutual inclusion. The inclusion *fmupd v True* ‘ $\{s.\ fmdom'\ s = X\} \cup$ *fmupd v False* ‘ $\{s.\ fmdom'\ s = X\} \subseteq \{s.\ fmdom'\ s = insert\ v\ X\}$ is trivial and can be solved by automation. For the complimentary proof $\{s.\ fmdom'\ s = insert\ v\ X\} \subseteq$ *fmupd v True* ‘ $\{s.\ fmdom'\ s = X\} \cup$ *fmupd v False* ‘ $\{s.\ fmdom'\ s = X\}$ however we need to do more work. In our case we choose a proof by contradiction and show that an $s \in \{s.\ fmdom'\ s = insert\ v\ X\}$ which is not also in '?B' cannot exist.

{
have $?A \subseteq ?B$ **proof**(*rule ccontr*)
assume C: $\neg(?A \subseteq ?B)$
moreover have $\exists s{\in}?A.\ s{\notin}?B$
using *C*
by *auto*
moreover obtain *s* **where** *obtain-s*: $s{\in}?A \wedge s{\notin}?B$
using *calculation*
by *auto*
moreover have $s{\notin}?B$
using *obtain-s*
by *auto*
moreover have $fmdom'\ s = X \cup \{v\}$
using *obtain-s*
by *auto*
moreover have $\forall s'{\in}?B.\ fmdom'\ s' = X \cup \{v\}$
by *auto*
moreover have
$(s \notin ((\lambda s.\ fmupd\ v\ True\ s)$ ‘ $\{s.\ fmdom'\ s = X\}))$
$(s \notin ((\lambda s.\ fmupd\ v\ False\ s)$ ‘ $\{s.\ fmdom'\ s = X\}))$
using *obtain-s*
by *blast+*

Show that every state $s \in \{s.\ fmdom'\ s = insert\ v\ X\}$ has been constructed from another state with domain 'X'.

moreover
{
fix $s :: 'a\ state$
assume *1*: $s \in \{s :: 'a\ state.\ fmdom'\ s = insert\ v\ X\}$
then have $fmrestrict\text{-}set\ X\ s \in \{s :: 'a\ state.\ fmdom'\ s = X\}$
using *subset-insertI fmsubset-restrict-set*
by *metis*
moreover
{
assume $fmlookup\ s\ v = Some\ True$
then have $s = fmupd\ v\ True\ (fmrestrict\text{-}set\ X\ s)$
using *1 fmupd-fmsubset-restrict-set*

```
          by metis
        }
        moreover {
          assume fmlookup s v = Some False
          then have s = fmupd v False (fmrestrict-set X s)
            using 1 fmupd-fmsubset-restrict-set
            by fastforce
        }
        moreover have fmlookup s v ≠ None
          using 1 fmdom′-notI
          by fastforce
        ultimately have
          (s ∈ ((λs. fmupd v True s) ` {s. fmdom′ s = X}))
          ∨ (s ∈ ((λs. fmupd v False s) ` {s. fmdom′ s = X}))

          by force
      }
      ultimately show False
        by meson
    qed
  }
  moreover have ?B ⊆ ?A
    by force
  ultimately show ?A = ?B by blast
qed
```

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 finite X ⟹ finite {(s :: ′a state). fmdom′ s = X}
proof (induction rule: finite.induct)
  case emptyI
  then have {s. fmdom′ s = {}} = {fmempty}
    by (simp add: empty-domain-fmap-set)
  then show ?case
    by (simp add: ‹{s. fmdom′ s = {}} = {fmempty}›)
next
  case (insertI A a)
  assume P1: finite A
```

```
    and P2: finite {s. fmdom' s = A}
  then show ?case
  proof (cases a ∈ A)
    case True
    then show ?thesis
      using insertI.IH insert-Diff
      by fastforce
  next
    case False
    then show ?thesis
    proof −
      have finite (
          ((λs. fmupd a True s) ‘ {s. fmdom' s = A})
            ∪ ((λs. fmupd a False s) ‘ {s. fmdom' s = A}))
        using False construction-of-all-possible-states-lemma insertI.IH
        by blast
      then show ?thesis
        using False construction-of-all-possible-states-lemma
        by fastforce
    qed
  qed
qed

— NOTE added lemma.
lemma bool-update-effect:
  fixes s X x v b
  assumes finite X s ∈ {s :: 'a state. fmdom' s = X} x ∈ X x ≠ v
  shows fmlookup ((λs :: 'a state. fmupd v b s) s) x = fmlookup s x
  using assms fmupd-lookup
  by auto

— NOTE added lemma.
lemma bool-update-inj:
  fixes X :: 'a set and v b
  assumes finite X v ∉ X
  shows inj-on (λs. fmupd v b s) {s :: 'a state. fmdom' s = X}
proof −
  let ?f = λs :: 'a state. fmupd v b s
  {
    fix s1 s2 :: 'a state
    assume s1 ∈ {s :: 'a state. fmdom' s = X} s2 ∈ {s :: 'a state. fmdom' s = X}
      ?f s1 = ?f s2
    moreover
    {
      have
        ∀x∈X. x ≠ v ⟶ fmlookup (?f s1) x = fmlookup s1 x
        ∀x∈X. x ≠ v ⟶ fmlookup (?f s2) x = fmlookup s2 x
        by simp+
      then have
```

$\forall x \in X.\ x \neq v \longrightarrow$ *fmlookup s1 x* = *fmlookup s2 x*
using *calculation(3)*
by *auto*
}
moreover have *fmlookup s1 v* = *fmlookup s2 v*
using *calculation* ‹$v \notin X$›
by *force*
ultimately have *s1* = *s2*
using *fmap-neq*
by *fastforce*
}
then show *inj-on* ($\lambda s.$ *fmupd v b s*) {*s* :: *'a state. fmdom' s* = *X*}
using *inj-onI*
by *blast*
qed

— NOTE added lemma.
lemma *card-update*:
fixes *X v b*
assumes *finite* (*X* :: *'a set*) $v \notin X$
shows
card (($\lambda s.$ *fmupd v b s*) ‘ {*s* :: *'a state. fmdom' s* = *X*})
= *card* {*s* :: *'a state. fmdom' s* = *X*}

proof −
have *inj-on* ($\lambda s.$ *fmupd v b s*) {*s* :: *'a state. fmdom' s* = *X*}
using *assms bool-update-inj*
by *fast*
then show
card (($\lambda s.$ *fmupd v b s*) ‘ {*s* :: *'a state. fmdom' s* = *X*}) = *card* {*s* :: *'a state. fmdom' s* = *X*}
using *card-image* **by** *blast*
qed

— NOTE added lemma.
lemma *updates-disjoint*:
fixes *X x*
assumes *finite X* $x \notin X$
shows
(($\lambda s.$ *fmupd x True s*) ‘ {*s. fmdom' s* = *X*})
∩ (($\lambda s.$ *fmupd x False s*) ‘ {*s. fmdom' s* = *X*}) = {}

proof −
let *?A* = (($\lambda s.$ *fmupd x True s*) ‘ {*s. fmdom' s* = *X*})
let *?B* = (($\lambda s.$ *fmupd x False s*) ‘ {*s. fmdom' s* = *X*})
{
assume *C*: $\neg(\forall a \in ?A.\ \forall b \in ?B.\ a \neq b)$
then have
$\forall a \in ?A.\ \forall b \in ?B.$ *fmlookup a x* $\neq$ *fmlookup b x*

by *simp*
then have $\forall\, a{\in}?A.\ \forall\, b{\in}?B.\ a \neq b$
by *blast*
then have *False*
using *C*
by *blast*
}
then show $?A \cap ?B = \{\}$
using *disjoint-iff-not-equal*
by *blast*
qed

lemma *card-of-set-of-all-possible-states*:
fixes X :: *'a set*
assumes *finite X*
shows $card\ \{(s :: 'a\ state).\ fmdom'\ s = X\} = 2\ \hat{}\ (card\ X)$
using *assms*
proof (*induction X*)
case *empty*
then have *1*: $\{s :: 'a\ state.\ fmdom'\ s = \{\}\} = \{fmempty\}$
using *empty-domain-fmap-set*
by *simp*
then have $card\ \{fmempty\} = 1$
using *is-singleton-altdef*
by *blast*
then have $2\hat{}(card\ \{\}) = 1$
by *auto*
then show *?case*
using *1*
by *auto*
next
case (*insert x F*)
then show *?case*
— TODO refactor and simplify proof further.
proof (*cases* $x \in F$)
case *True*
then show *?thesis*
using *insert.hyps*(*2*)
by *blast*
next
case *False*
then have
$\{s :: 'a\ state.\ fmdom'\ s = insert\ x\ F\}$
$= (\lambda s.\ fmupd\ x\ True\ s)\ `\ \{s.\ fmdom'\ s = F\} \cup (\lambda s.\ fmupd\ x\ False\ s)\ `\ \{s.\ fmdom'\ s = F\}$

using *False construction-of-all-possible-states-lemma*
by *metis*

```
  then have 2:
      card ({s :: 'a state. fmdom' s = insert x F})
      = card ((λs. fmupd x True s) ` {s. fmdom' s = F} ∪ (λs. fmupd x False s)
` {s. fmdom' s = F})

    by argo
  then have 3: 2^(card (insert x F)) = 2 * 2^(card F)
    using False insert.hyps(1)
    by simp
  then have
    card ((λs. fmupd x True s) ` {s. fmdom' s = F}) = 2^(card F)
    card ((λs. fmupd x False s) ` {s. fmdom' s = F}) = 2^(card F)
    using False card-update insert.IH insert.hyps(1)
    by metis+
  moreover have
        ((λs. fmupd x True s) ` {s. fmdom' s = F})
        ∩ ((λs. fmupd x False s) ` {s. fmdom' s = F})
      = {}

    using False insert.hyps(1) updates-disjoint
    by metis
  moreover have card (
        ((λs. fmupd x True s) ` {s. fmdom' s = F})
        ∪ ((λs. fmupd x False s) ` {s. fmdom' s = F})
      )
      = card (((λs. fmupd x True s) ` {s. fmdom' s = F}))
        + card ((λs. fmupd x False s) ` {s. fmdom' s = F})

    using calculation card-Un-disjoint card.infinite
      power-eq-0-iff rel-simps(76)
    by metis
  then have card (
        ((λs. fmupd x True s) ` {s. fmdom' s = F})
        ∪ ((λs. fmupd x False s) ` {s. fmdom' s = F})
      )
      = 2 * (2^(card F))
    using calculation(1, 2)
    by presburger
  then have card (
        ((λs. fmupd x True s) ` {s. fmdom' s = F})
        ∪ ((λs. fmupd x False s) ` {s. fmdom' s = F})
      )
      = 2^(card (insert x F))
    using insert.IH 3
    by metis
  then show ?thesis
    using 2
    by argo
qed
```

qed

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 (*induction as*)
qed *auto*

lemma *state-list-length-non-zero*:
fixes *as s*
shows ¬(*0* = *length* (*state-list s as*))
proof (*induction as*)
qed *auto*

lemma *state-list-length-lemma*:
fixes *as s*
shows *length as* = *length* (*state-list s as*) − *1*
proof (*induction as arbitrary*: *s*)
next case (*Cons a as*)
have *length* (*state-list s* (*Cons a as*)) − *1* = *length* (*state-list* (*state-succ s a*) *as*)
by *auto*
— TODO unwrap metis proof.
then show *length* (*Cons a as*) = *length* (*state-list s* (*Cons a as*)) − *1*
by (*metis Cons.IH Suc-diff-1 empty-state-list-lemma length-Cons length-greater-0-conv*)
qed *simp*

lemma *state-list-length-lemma-2*:
fixes *as s*
shows (*length* (*state-list s as*)) = (*length as* + *1*)
proof (*induction as arbitrary*: *s*)
qed *auto*

— NOTE using fun because of multiple defining equations.
— NOTE name shortened to 'state_def'
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 −
  — NOTE Show equivalence by proving both directions. Left-to-right is trivial. Right-to-Left primarily involves exploiting the prefix premise, induction hypothesis and 'state_set' definition.
  have *s1* ∈ *state-set s2* ⟹ *prefix s1 s2* ∧ *s1* ≠ []
    by (*induction s2 arbitrary*: *s1*) *auto*
  moreover {
    assume *P*: *prefix s1 s2 s1* ≠ []
    then have *s1* ∈ *state-set s2*
    proof (*induction s2 arbitrary*: *s1*)
      case (*Cons a s2*)
      obtain *s1'* **where** *1*: *s1* = *a* # *s1' prefix s1' s2*
        using *Cons.prems*(*1*, *2*) *prefix-Cons*
        by *metis*
      then show *?case* **proof** (*cases s1'* = [])
        case *True*
        then show *?thesis*
          using *1*
          by *force*
      next
        case *False*
        then have *s1'* ∈ *state-set s2*
          using *1 False Cons.IH*
          by *blast*
        then show *?thesis*
          using *1*
          by *fastforce*
      qed
    qed *simp*
  }
  ultimately show *s1* ∈ *state-set s2* ⟷ *prefix s1 s2* ∧ *s1* ≠ []
    by *blast*
qed

lemma *state-set-finite*:
  fixes *X*
  shows *finite* (*state-set X*)
  by (*induction X*) *auto*

lemma *LENGTH-state-set*:
  fixes *X e*
  assumes *e* ∈ *state-set X*

shows *length e* ≤ *length X*
using *assms*
by (*induction X arbitrary*: *e*) *auto*

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*
using *assms LENGTH-state-set le-imp-less-Suc*
by *force*

lemma *NIL-NOTIN-stateset*:
fixes *X*
shows [] ∉ *state-set X*
by (*induction X*) *auto*

— NOTE added lemma.
lemma *state-set-card-i*:
fixes *X a*
shows [*a*] ∉ (*Cons a* ' *state-set X*)
by (*induction X*) *auto*

— NOTE added lemma.
lemma *state-set-card-ii*:
fixes *X a*
shows *card* (*Cons a* ' *state-set X*) = *card* (*state-set X*)
proof −
have *inj-on* (*Cons a*) (*state-set X*)
by *simp*
then show *?thesis*
using *card-image*
by *blast*
qed

— NOTE added lemma.
lemma *state-set-card-iii*:
fixes *X a*
shows *card* (*state-set* (*a* # *X*)) = *1* + *card* (*state-set X*)
proof −
have *card* (*state-set* (*a* # *X*)) = *card* (*insert* [*a*] (*Cons a* ' *state-set X*))
by *auto*
— TODO unwrap this metis step.
also have ... = *1* + *card* (*Cons a* ' *state-set X*)
using *state-set-card-i*
by (*metis Suc-eq-plus1-left card-insert-disjoint finite-imageI state-set-finite*)
also have... = *1* + *card* (*state-set X*)

using *state-set-card-ii*
by *metis*
finally show *card* (*state-set* (*a* # *X*)) = *1* + *card* (*state-set* *X*)
by *blast*
qed

lemma *state-set-card*:
fixes *X*
shows *card* (*state-set* *X*) = *length* *X*
proof (*induction* *X*)
case (*Cons* *a* *X*)
then have *card* (*state-set* (*a* # *X*)) = *1* + *card* (*state-set* *X*)
using *state-set-card-iii*
by *fast*
then show *?case*
using *Cons*
by *fastforce*
qed *auto*

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*)
unfolding *state-succ-def fmap-add-ltr-def*
using *assms*
by *force*

lemma *FDOM-state-succ-subset*:
fmdom′ (*state-succ* *s* *a*) ⊆ (*fmdom′* *s* ∪ *fmdom′* (*snd* *a*))
unfolding *state-succ-def fmap-add-ltr-def*
by *simp*

— NOTE definition ‘qispl_then‘ removed (was not being used).

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 −
{
have *fmdom′* *e* ⊆ *action-dom* *p* *e*
unfolding *action-dom-def*
by *blast*
also have . . . ⊆ *prob-dom* *PROB*
unfolding *action-dom-def prob-dom-def*

```
      using assms(1)
      by blast
    finally have fmdom' e ⊆ fmdom' s
      using assms
      by (auto simp: valid-states-def)
  }
  then show fmdom' e ⊆ fmdom' s
    by simp
qed


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 −
  {
    have fmdom' (snd a) ⊆ (λ(s1, s2). action-dom s1 s2) a
      unfolding action-dom-def
      using case-prod-beta
      by fastforce
    also have ... ⊆ prob-dom PROB
      using assms(1) prob-dom-def Sup-upper
      by fast
    finally have fmdom' (snd a) ⊆ fmdom' s
      using assms(2) valid-states-def
      by fast
  }
  then show ?thesis
    by simp
qed


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 −
  {
    have fmdom' p ⊆ (λ(s1, s2). action-dom s1 s2) (p, e)
      using action-dom-def
      by fast
    also have ... ⊆ prob-dom PROB
      using assms(1)
      by (simp add: Sup-upper pair-imageI prob-dom-def)
    finally have fmdom' p ⊆ fmdom' s
      using assms(2) valid-states-def
      by fast
  }
```

```
  then show ?thesis
    by simp
qed


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 –
  {
    have fmdom' (fst a) ⊆ (λ(s1, s2). action-dom s1 s2) a
      using action-dom-def
      by force
    also have ... ⊆ prob-dom PROB
      using assms(1)
      by (simp add: Sup-upper pair-imageI prob-dom-def)
    finally have fmdom' (fst a) ⊆ fmdom' s
      using assms(2) valid-states-def
      by fast
  }
  then show ?thesis
    by simp
qed


— TODO unwrap the simp 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
  using assms
  by (simp add: Sup-upper pair-imageI prob-dom-def valid-states-def)


— TODO unwrap the metis proof.
lemma FDOM-eff-subset-prob-dom:
  fixes p e
  assumes (p, e) ∈ PROB
  shows fmdom' e ⊆ prob-dom PROB
  using assms
  by (metis Sup-upper Un-subset-iff action-dom-def pair-imageI prob-dom-def)


lemma FDOM-eff-subset-prob-dom-pair:
  fixes a
  assumes a ∈ PROB
  shows fmdom' (snd a) ⊆ prob-dom PROB
  using assms(1) FDOM-eff-subset-prob-dom surjective-pairing
```

by *metis*

— TODO unwrap metis proof.
lemma *FDOM-pre-subset-prob-dom*:
fixes *p e*
assumes $(p,\ e) \in PROB$
shows *fmdom′ p* ⊆ *prob-dom PROB*
using *assms*
by (*metis* (*no-types*) *Sup-upper Un-subset-iff action-dom-def pair-imageI prob-dom-def*)

lemma *FDOM-pre-subset-prob-dom-pair*:
fixes *a*
assumes *a* ∈ *PROB*
shows *fmdom′* (*fst a*) ⊆ *prob-dom PROB*
using *assms FDOM-pre-subset-prob-dom surjective-pairing*
by *metis*

3.1.4 Properties of Valid Plans

lemma *valid-plan-valid-head*:
assumes (*h* # *as* ∈ *valid-plans PROB*)
shows *h* ∈ *PROB*
using *assms valid-plans-def*
by *force*

lemma *valid-plan-valid-tail*:
assumes (*h* # *as* ∈ *valid-plans PROB*)
shows (*as* ∈ *valid-plans PROB*)
using *assms*
by (*simp add*: *valid-plans-def*)

— TODO unwrap simp 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*))
unfolding *valid-plans-def*
using *assms*
by (*simp add*: *FDOM-pre-subset-prob-dom-pair ListMem-iff rev-subsetD valid-plans-def*)

lemma *valid-append-valid-suff*:
assumes *as1* @ *as2* ∈ (*valid-plans PROB*)
shows *as2* ∈ (*valid-plans PROB*)
using *assms*
by (*simp add*: *valid-plans-def*)

lemma *valid-append-valid-pref*:
  assumes *as1* @ *as2* ∈ (*valid-plans PROB*)
  shows *as1* ∈ (*valid-plans PROB*)
  using *assms*
  by (*simp add*: *valid-plans-def*)

lemma *valid-pref-suff-valid-append*:
  assumes *as1* ∈ (*valid-plans PROB*) *as2* ∈ (*valid-plans PROB*)
  shows (*as1* @ *as2*) ∈ (*valid-plans PROB*)
  using *assms*
  by (*simp add*: *valid-plans-def*)

— NOTE showcase (case split seems necessary for MP of IH but the original proof does not need it).
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*)
  using *assms*
proof (*induction as arbitrary*: *PROB h s0*)
  case *Nil*
  have *h* = *s0*
    using *Nil.prems*(*3*) *ListMem-iff*
    by *force*
  then show *?case*
    by *simp*
next
  case (*Cons a as*)
  then show *?case*
    — NOTE This case split seems necessary to be able to infer
'ListMem h (state_list (state_succ s0 a) as)'
which is required in order to apply MP to the induction hypothesis.
  proof (*cases h* = *s0*)
    case *False*
      — TODO proof steps could be refactored into auxillary lemmas.
    {
      have *a* ∈ *PROB*
        using *Cons.prems*(*2*) *valid-plan-valid-head*
        by *fast*
      then have *fmdom′* (*snd a*) ⊆ *fmdom′ s0*
        using *Cons.prems*(*1*) *FDOM-eff-subset-FDOM-valid-states-pair*
        by *blast*
      then have *fmdom′* (*state-succ s0 a*) = *fmdom′ s0*
        using *FDOM-state-succ*[*of* - *s0*] *Cons.prems*(*1*) *valid-states-def*

```
        by presburger
      }
      note 1 = this
      {
        have fmdom' s0 = prob-dom PROB
          using Cons.prems(1) valid-states-def
          by fast
        then have state-succ s0 a ∈ valid-states PROB
          unfolding valid-states-def
          using 1
          by force
      }
      note 2 = this
      {
        have ListMem h (state-list (state-succ s0 a) as)
          using Cons.prems(3) False
          by (simp add: ListMem-iff)
      }
      note 3 = this
      {
        have as ∈ valid-plans PROB
          using Cons.prems(2) valid-plan-valid-tail
          by fast
        then have fmdom' h = fmdom' (state-succ s0 a)
          using 1 2 3 Cons.IH[of state-succ s0 a]
          by blast
      }
      then show ?thesis
        using 1
        by argo
    qed simp
qed
```

— TODO unwrap metis 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)
  using assms
  by (metis MEM-statelist-FDOM mem-Collect-eq valid-states-def)
```

— TODO refactor (characterization lemma for 'state_succ').
— TODO unwrap metis proof.
— NOTE added lemma.

```
lemma lemma-1-i:
  fixes s a PROB
```

assumes *s* ∈ *valid-states PROB a* ∈ *PROB*
shows *state-succ s a* ∈ *valid-states PROB*
using *assms*
by (*metis FDOM-eff-subset-FDOM-valid-states-pair FDOM-state-succ mem-Collect-eq valid-states-def*)

— TODO unwrap smt proof.
— NOTE added lemma.
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*)
by (*smt NIL-NOTIN-stateset image-cong image-image last-ConsR*)

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*)
using *assms*
proof (*induction as arbitrary*: *s PROB*)
— NOTE Base case simplifies to {*s*} ⊆ *valid-states PROB* which itself follows directly from 1st assumption.
case (*Cons a as*)

Split the 'insert' term produced by *state-set* (*state-list s* (*a* # *as*)) and proof inclusion in 'valid_states PROB' for both parts.

{
— NOTE Inclusion of the first subset follows from the induction premise by simplification. The inclusion of the second subset is shown by applying the induction hypothesis to 'state_succ s a' and some elementary set simplifications.
have *last* [*s*] ∈ *valid-states PROB*
using *Cons.prems*(*1*)
by *simp*
moreover {
{
have *a* ∈ *PROB*
using *Cons.prems*(*2*) *valid-plan-valid-head*
by *fast*
then have *state-succ s a* ∈ *valid-states PROB*
using *Cons.prems*(*1*) *lemma-1-i*
by *blast*
}
moreover have *as* ∈ *valid-plans PROB*
using *Cons.prems*(*2*) *valid-plan-valid-tail*
by *fast*
then have (*last* ' *state-set* (*state-list* (*state-succ s a*) *as*)) ⊆ *valid-states PROB*
using *calculation Cons.IH*[*of state-succ s a*]
by *presburger*
then have (*last* ' ((#) *s* ' *state-set* (*state-list* (*state-succ s a*) *as*))) ⊆ *valid-states PROB*

```
        using lemma-1-ii
        by metis
    }
    ultimately have
     (last ‘ insert [s] ((#) s ‘ state-set (state-list (state-succ s a) as))) ⊆ valid-states
PROB
      by simp
  }
  then show ?case
    by fastforce
qed auto
```

— TODO unwrap metis 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)))
  using assms
 by (metis LENGTH-state-set Suc-eq-plus1-left add.commute state-list-length-lemma-2)


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))))

  by (metis length-Cons state-list.simps(2) state-set-card)


lemma card-state-set:
  fixes as s
  shows (Suc (length as)) = card (state-set (state-list s as))
  by (simp add: state-list-length-lemma-2 state-set-card)


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 −
  have x ≠ [] prefix x (state-list s as)
    using assms(1) state-set-thm
    by blast+
  moreover have y ≠ [] prefix y (state-list s as)
    using assms(2) state-set-thm
    by blast+
```

ultimately show *?thesis*
using *assms(3) append-eq-append-conv prefixE*
by *metis*
qed

— NOTE added definition (imported from pred_setScript.sml:1562).
definition *inj* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$ **where**
inj f A B $\equiv (\forall x \in A.\ f\ x \in B) \land$ *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 $s \in$ *valid-states PROB as* $\in$ *valid-plans PROB x* $\in$ *state-set (state-list s as)*
shows *last x* $\in$ *valid-states PROB*
proof −
have *last x* $\in$ *last ‘ (state-set (state-list s as))*
using *assms(3)*
by *simp*
then show *?thesis*
using *assms(1, 2) lemma-1*
by *blast*
qed

lemma *not-eq-last-diff-paths-ii*:
assumes ($s \in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
$\neg$(*inj (last) (state-set (state-list s as)) (valid-states PROB)*)
shows $\exists$ *l1*. $\exists$ *l2*.
l1 $\in$ *state-set (state-list s as)*
$\land$ *l2* $\in$ *state-set (state-list s as)*
$\land$ *last l1 = last l2*
$\land$ *l1* $\neq$ *l2*

proof −
let *?S=state-set (state-list s as)*
have *1*: $\neg(\forall x \in ?S.$ *last x* $\in$ *valid-states PROB*) = *False*
using *assms(1, 2) not-eq-last-diff-paths-i*
by *blast*
{
have
($\neg$(*inj (last) ?S (valid-states PROB)*)) = ($\neg$(($\forall x \in ?S.\ \forall y \in ?S.$ *last x = last y* $\longrightarrow x = y$)))
unfolding *inj-def inj-on-def*
using *1*
by *blast*
then have
($\neg$(*inj (last) ?S (valid-states PROB)*))

= (∃ *x*. ∃ *y*. *x*∈*?S* ∧ *y*∈*?S* ∧ *last x* = *last y* ∧ *x* ≠ *y*)

using *assms*(*3*)
by *blast*
}
then show *?thesis*
using *assms*(*3*) **by** *blast*
qed

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 −
obtain *l1 l2* **where**
l1 ∈ *state-set* (*state-list s as*)
∧ *l2* ∈ *state-set* (*state-list s as*)
∧ *last l1* = *last l2*
∧ *l1* ≠ *l2*

using *assms*(*1*, *2*, *3*) *not-eq-last-diff-paths-ii*
by *blast*
then show *?thesis*
using *neq-mems-state-set-neq-len*
by *blast*
qed

— NOTE this lemma was removed due to being redundant and being shadowed later on:
lemma empty_list_nin_state_set

lemma *nempty-sl-in-state-set*:
fixes *sl*
assumes *sl* ≠ []
shows *sl* ∈ *state-set sl*
using *assms state-set-thm*
by *auto*

lemma *empty-list-nin-state-set*:
fixes *h slist as*

assumes (*h* # *slist*) ∈ *state-set* (*state-list s as*)
shows (*h* = *s*)
using *assms*
by (*induction as*) *auto*

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*))
using *assms*
by (*induction slist*) *auto*

— TODO move up and replace 'FactoredSystem.lemma_1_i'?
lemma *valid-action-valid-succ*:
assumes *h* ∈ *PROB s* ∈ *valid-states PROB*
shows (*state-succ s h*) ∈ *valid-states PROB*
using *assms lemma-1-i*
by *blast*

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'*)))
using *assms*
proof (*induction slist arbitrary*: *as s PROB*)
case *cons-1*: (*Cons a slist*)
have *1*: *s* # *slist* ∈ *state-set* (*state-list s as*)
using *cons-1.prems*(*5*) *empty-list-nin-state-set*
by *auto*
then show *?case*
using *cons-1*
proof (*cases as*)
case *cons-2*: (*Cons a' R_{as}*)
then have *a*: *state-succ s a'* ∈ *valid-states PROB*
using *cons-1.prems*(*3*, *4*) *valid-action-valid-succ valid-plan-valid-head*
by *metis*
then have *b*: R_{as} ∈ *valid-plans PROB*
using *cons-1.prems*(*4*) *cons-2 valid-plan-valid-tail*
by *fast*
then show *?thesis*
proof (*cases slist*)
case *Nil*
then show *?thesis*

using *cons-1.prems*(*5*) *empty-list-nin-state-set*
by *auto*
next
case *cons-3*: (*Cons a″ R_{slist}*)
then have *i*: *a″* # R_{slist} ∈ *state-set* (*state-list* (*state-succ s a′*) R_{as})
using *1 cons-2 cons-in-state-set-2*
by *blast*
then show *?thesis*
proof (*cases* R_{as})
case *Nil*
then show *?thesis*
using *i cons-2 cons-3*
by *auto*
next
case (*Cons a‴* R_{as}′)
then obtain *as′* **where**
prefix as′ (*a‴* # R_{as}′) *exec-plan* (*state-succ s a′*) *as′* = *last slist*
length slist = *Suc* (*length as′*)
using *cons-1.IH*[*of a‴* # R_{as}′ *state-succ s a′ PROB*]
using *i a b cons-3*
by *blast*
then show *?thesis*
using *Cons-prefix-Cons cons-2 cons-3 exec-plan.simps*(*2*) *last.simps*
length-Cons
list.distinct(*1*) *local.Cons*
by *metis*
qed
qed
qed *auto*
qed *auto*

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 −
obtain *as-1* **where** *1*: (*prefix as-1 as*) (*exec-plan s as-1* = *last slist-1*)
length slist-1 = *Suc* (*length as-1*)
using *assms*(*1*, *2*, *3*, *4*, *6*) *in-state-set-imp-eq-exec-prefix*
by *blast*

obtain *as-2* **where** *2*: (*prefix as-2 as*) (*exec-plan s as-2* = *last slist-2*)
  (*length slist-2*) = *Suc* (*length as-2*)
  using *assms*(*1*, *2*, *3*, *5*, *7*) *in-state-set-imp-eq-exec-prefix*
  by *blast*
then have *length as-1* ≠ *length as-2*
  using *assms*(*8*) *1*(*3*) *2*(*3*)
  by *fastforce*
then consider (*i*) *length as-1* < *length as-2* | (*ii*) *length as-1* > *length as-2*
  by *force*
then show *?thesis*
proof (*cases*)
  case *i*
  then have *prefix as-1 as-2*
    using *1*(*1*) *2*(*1*) *len-gt-pref-is-pref*
    by *blast*
  then obtain *a* **where** *a1*: *as-2* = *as-1* @ *a*
    using *prefixE*
    by *blast*
  then obtain *b* **where** *b1*: *as* = *as-2* @ *b*
    using *prefixE* *2*(*1*)
    by *blast*
  let *?as1*=*as-1*
  let *?as2*=*a*
  let *?as3*=*b*
  have *as* = *?as1* @ *?as2* @ *?as3*
    using *a1 b1*
    by *simp*
  moreover have *exec-plan s* (*?as1* @ *?as2*) = *exec-plan s ?as1*
    using *1*(*2*) *2*(*2*) *a1 assms*(*9*)
    by *auto*
  moreover have *?as2* ≠ []
    using *i a1*
    by *simp*
  ultimately show *?thesis*
    by *blast*
next
  case *ii*
  then have *prefix as-2 as-1*
    using *1*(*1*) *2*(*1*) *len-gt-pref-is-pref*
    by *blast*
  then obtain *a* **where** *a2*: *as-1* = *as-2* @ *a*
    using *prefixE*
    by *blast*
  then obtain *b* **where** *b2*: *as* = *as-1* @ *b*
    using *prefixE* *1*(*1*)
    by *blast*
  let *?as1*=*as-2*
  let *?as2*=*a*
  let *?as3*=*b*

```
    have as = ?as1 @ ?as2 @ ?as3
      using a2 b2
      by simp
    moreover have exec-plan s (?as1 @ ?as2) = exec-plan s ?as1
      using 1(2) 2(2) a2 assms(9)
      by auto
    moreover have ?as2 ≠ []
      using ii a2
      by simp
    ultimately show ?thesis
      by blast
  qed
qed


lemma FINITE-prob-dom:
  assumes finite PROB
  shows  finite (prob-dom PROB)
proof −
  {
    fix x
    assume P2: x ∈ PROB
    then have 1: (λ(s1, s2). action-dom s1 s2) x = fmdom' (fst x) ∪ fmdom' (snd x)
      by (simp add: action-dom-def case-prod-beta')
    then have 2: finite (fset (fmdom (fst x))) finite (fset (fmdom (snd x)))
      by auto
    then have 3: fset (fmdom (fst x)) = fmdom' (fst x) fset (fmdom (snd x)) = fmdom' (snd x)
      by (auto simp add: fmdom'-alt-def)
    then have finite (fmdom' (fst x))
      using 2 by auto
    then have finite (fmdom' (snd x))
      using 2 3 by auto
    then have finite ((λ(s1, s2). action-dom s1 s2) x)
      using 1 2 3
      by simp
  }
  then show finite (prob-dom PROB)
    unfolding prob-dom-def
    using assms
    by blast
qed


lemma CARD-valid-states:
  assumes finite (PROB :: 'a problem)
  shows (card (valid-states PROB :: 'a state set) = 2 ^ card (prob-dom PROB))
proof −
```

have *1*: *finite* (*prob-dom PROB*)
  using *assms FINITE-prob-dom*
  by *blast*
have(*card* (*valid-states PROB* :: *'a state set*)) = *card* {*s* :: *'a state. fmdom' s* = *prob-dom PROB*}
  unfolding *valid-states-def*
  by *simp*
also have ... = *2* ^ (*card* (*prob-dom PROB*))
  using *1 card-of-set-of-all-possible-states*
  by *blast*
finally show *?thesis*
  by *blast*
qed

— NOTE type of 'valid_states PROB' has to be asserted to match 'FINITE_states' in the proof.
lemma *FINITE-valid-states*:
fixes *PROB* :: *'a problem*
shows *finite PROB* ⟹ *finite* ((*valid-states PROB*) :: *'a state set*)
proof (*induction PROB rule*: *finite.induct*)
case *emptyI*
then have *valid-states* {} = {*fmempty*}
  unfolding *valid-states-def prob-dom-def*
  using *empty-domain-fmap-set*
  by *force*
then show *?case*
  by(*subst* ‹*valid-states* {} = {*fmempty*}›) *auto*
next
case (*insertI A a*)
{
  then have *finite* (*insert a A*)
    by *blast*
  then have *finite* (*prob-dom* (*insert a A*))
    using *FINITE-prob-dom*
    by *blast*
  then have *finite* {*s* :: *'a state. fmdom' s* = *prob-dom* (*insert a A*)}
    using *FINITE-states*
    by *blast*
}
then show *?case*
  unfolding *valid-states-def*
  by *simp*
qed

— NOTE type of 'PROB' had to be fixed for use of 'FINITE_valid_states'.
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 −
have *Suc* (*length as*) > *2*^(*card* (*fmdom′ s*))
using *assms*(*4*)
by *linarith*
then have *1*: *card* (*state-set* (*state-list s as*)) > *2*^*card* (*fmdom′ s*)
using *card-state-set*[*symmetric*]
by *metis*
{
— NOTE type of 'valid_states PROB' had to be asserted to match 'FINITE_valid_states'.
have *2*: *finite* (*prob-dom PROB*) *finite* ((*valid-states PROB*) :: ′*a state set*)
using *assms*(*1*) *FINITE-prob-dom FINITE-valid-states*
by *blast*+
have *3*: *fmdom′ s* = *prob-dom PROB*
using *assms*(*2*) *valid-states-def*
by *fast*
then have *card* ((*valid-states PROB*) :: ′*a state set*) = *2*^*card* (*fmdom′ s*)
using *assms*(*1*) *CARD-valid-states*
by *auto*
then have *4*: *card* (*state-set* (*state-list* (*s* :: ′*a state*) *as*)) > *card* ((*valid-states PROB*) :: ′*a state set*)
unfolding *valid-states-def*
using *1 2*(*1*) *3 card-of-set-of-all-possible-states*[*of prob-dom PROB*]
by *argo*
— TODO refactor into lemma.
{
let *?S*=*state-set* (*state-list* (*s* :: ′*a state*) *as*)
let *?T*=*valid-states PROB* :: ′*a state set*
assume *C2*: *inj-on last ?S*
— TODO unwrap the metis step or refactor into lemma.
have *a*: *?T* ⊆ *last* ‘ *?S*
using *C2*
by (*metis 2*(*2*) *4 assms*(*2*) *assms*(*3*) *card-image card-mono lemma-1 not-less*)
have *finite* (*state-set* (*state-list s as*))
using *state-set-finite*
by *auto*
then have *card* (*last* ‘ *?S*) = *card ?S*
using *C2 inj-on-iff-eq-card*
by *blast*
also have ... > *card ?T*
using *4*
by *blast*

then have $\exists x. x \in (last \text{ ` } ?S) \wedge x \notin ?T$
using *C2 a assms*(*2*) *assms*(*3*) *calculation lemma-1*
by *fastforce*
}
note *5* = *this*
moreover
{
assume *C*: *inj last* (*state-set* (*state-list* (*s* :: *'a state*) *as*)) (*valid-states PROB*)
then have *inj-on last* (*state-set* (*state-list* (*s* :: *'a state*) *as*))
using *C inj-def*
by *blast*
then obtain *x* **where** $x \in last$ ` (*state-set* (*state-list s as*)) $\wedge$ $x \notin$ *valid-states PROB*
using *5*
by *presburger*
then have ¬(∀ *x*∈*state-set* (*state-list s as*). *last x* ∈ *valid-states PROB*)
by *blast*
then have ¬*inj last* (*state-set* (*state-list* (*s* :: *'a state*) *as*)) (*valid-states PROB*)
using *inj-def*
by *metis*
then have *False*
using *C*
by *simp*
}
ultimately have ¬*inj last* (*state-set* (*state-list* (*s* :: *'a state*) *as*)) (*valid-states PROB*)
unfolding *inj-def*
by *blast*
}
then obtain *slist-1 slist-2* **where** *6*:
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*
using *assms*(*2*, *3*) *not-eq-last-diff-paths*
by *blast*
then show *?thesis*
proof (*cases as*)
case *Nil*

4th assumption is violated in the 'Nil' case.

then have ¬(*2* ^ *card* (*fmdom' s*) − *1* < *length as*)
using *Nil*
by *simp*
then show *?thesis*
using *assms*(*4*)
by *blast*
next

case (*Cons a list*)
then have *as* ≠ []
by *simp*
moreover have *slist-1* ≠ [] *slist-2* ≠ []
using *6(1, 2) NIL-NOTIN-stateset*
by *blast+*
ultimately show *?thesis*
using *assms(2, 3) 6(1, 2, 3, 4) eq-last-state-imp-append-nempty-as*
by *fastforce*
qed
qed

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 −
have *prob-dom PROB* = *fmdom' s*
using *assms(2) valid-states-def*
by *fast*
then have *2* ^ *card* (*fmdom' s*) − *1* < *length as*
using *assms(4)*
by *argo*
then show *?thesis*
using *assms(1, 2, 3) lemma-2*
by *blast*
qed

— NOTE type for ‘s‘ had to be fixed (type mismatch in obtain statement).
— NOTE type for ‘as1‘, ‘as2‘ and ‘as3‘ had to be fixed (due type mismatch on ‘as1‘ in ‘cycle_removal_lemma‘)
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 −
have *prob-dom PROB* = *fmdom' s*

using *assms*(*2*) *valid-states-def*
by *fast*
then have *2* ^ *card* (*fmdom′ s*) − *1* < *length as*
using *assms*(*4*)
by *argo*
then obtain *as1 as2 as3* :: *′a action list* **where** *1*:
as1 @ *as2* @ *as3* = *as exec-plan s* (*as1* @ *as2*) = *exec-plan s as1 as2* ≠ []
using *assms*(*1*, *2*, *3*) *lemma-2*
by *metis*
have *2*: *exec-plan s* (*as1* @ *as3*) = *exec-plan s* (*as1* @ *as2* @ *as3*)
using *1 cycle-removal-lemma*
by *fastforce*
let *?as′* = *as1* @ *as3*
have *exec-plan s as* = *exec-plan s ?as′*
using *1 2*
by *auto*
moreover have *length ?as′* < *length as*
using *1 nempty-list-append-length-add*
by *blast*
moreover have *subseq ?as′ as*
using *1 subseq-append′*
by *blast*
ultimately show (∃ *as′*.
(*exec-plan s as* = *exec-plan s as′*) ∧ (*length as′* < *length as*) ∧ (*subseq as′ as*))
by *blast*
qed

— TODO unwrap meson step.
lemma *sublist-valid-is-valid*:
fixes *as′ as PROB*
assumes (*as* ∈ *valid-plans PROB*) (*subseq as′ as*)
shows *as′* ∈ *valid-plans PROB*
using *assms*
by (*simp add*: *valid-plans-def*) (*meson dual-order.trans fset-of-list-subset sub-list-subset*)

— NOTE type of 's' had to be fixed (type mismatch in goal).
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 (*cases length as* ≤ (*2* ^ (*card* (*prob-dom PROB*))) − *1*)
case *True*

then have *exec-plan s as = exec-plan s as*
by *simp*
then have *subseq as as*
by *auto*
then have *length as* ≤ (*2*^(*card* (*prob-dom PROB*)) − *1*)
using *True*
by *auto*
then show *?thesis*
by *blast*
next
case *False*
then have *length as* > (*2* ^(*card* (*prob-dom PROB*))) − *1*
using *False*
by *auto*
then obtain *as′* **where** *1*:
exec-plan s as = exec-plan s as′ length as′ < length as subseq as′ as
using *assms lemma-3*
by *blast*
{
fix *p*
assume *exec-plan s as = exec-plan s p subseq p as*
2 ^ *card* (*prob-dom PROB*) − *1* < *length p*
then have (∃ *p′*. (*exec-plan s as = exec-plan s p′* ∧ *subseq p′ as*) ∧ *length p′* < *length p*)
using *assms*(*1*, *2*, *3*) *lemma-3 sublist-valid-is-valid*
by *fastforce*
}
then have ∀ *p*. *exec-plan s as = exec-plan s p* ∧ *subseq p as* ⟶
(∃ *p′*. (*exec-plan s as = exec-plan s p′* ∧ *subseq p′ as*)
∧ *length p′* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*)
using *general-theorem*[**where**
P = λ(*as′′* :: *′a action list*). (*exec-plan s as = exec-plan s as′′*) ∧ *subseq as′′ as*
and *l* = (*2* ^(*card* (*prob-dom* (*PROB* ::*′a problem*)))) − *1* **and** *f* = *length*]
by *blast*
then obtain *p′* **where**
exec-plan s as = exec-plan s p′ subseq p′ as length p′ ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
by *blast*
then show *?thesis*
using *sublist-refl*
by *blast*
qed

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*)
  using *assms*
proof (*induction as arbitrary*: *s PROB*)
  case (*Cons a as*)
  then have *a* ∈ *PROB*
    using *valid-plan-valid-head*
    by *metis*
  then have *state-succ s a* ∈ *valid-states PROB*
    using *Cons.prems*(*2*) *valid-action-valid-succ*
    by *blast*
  moreover have *as* ∈ *valid-plans PROB*
    using *Cons.prems*(*1*) *valid-plan-valid-tail*
    by *fast*
  ultimately show *?case*
    using *Cons.IH*
    by *force*
qed *simp*

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*))
  using *assms*
proof (*induction as arbitrary*: *s PROB*)
  case (*Cons a as*)
  have *as* ∈ *valid-plans PROB*
    using *Cons.prems valid-plan-valid-tail*
    by *fast*
  then have *fmdom'* (*exec-plan* (*state-succ s a*) *as*) ⊆ *fmdom'* (*state-succ s a*) ∪
prob-dom PROB
    using *Cons.IH*[*of - state-succ s a*]
    by *simp*
      — TODO unwrap metis proofs.
  moreover have *fmdom' s* ∪ *fmdom'* (*snd a*) ∪ *prob-dom PROB* = *fmdom' s* ∪
prob-dom PROB
    by (*metis*
      Cons.prems FDOM-eff-subset-prob-dom-pair sup-absorb2 sup-assoc valid-plan-valid-head)
  ultimately show *?case*
    by (*metis* (*no-types*, *lifting*)
      FDOM-state-succ-subset exec-plan.simps(*2*) *order-refl subset-trans sup.mono*)
qed *simp*

— NOTE added lemma.
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 −
have *1*: ∀ *l*∈*reachable-s PROB s*. ∃ *as*. *l* = *exec-plan s as* ∧ *as* ∈ *valid-plans PROB*
using *reachable-s-def*
by *fastforce*
{
fix *l*
assume *P1*: *l* ∈ *reachable-s PROB s*
— NOTE type for 's' and 'as' had to be fixed due to type mismatch in obtain statement.
then obtain *as* :: *'a action list* **where** *a*: *l* = *exec-plan s as* ∧ *as* ∈ *valid-plans PROB*
using *1*
by *blast*
then have *exec-plan s as* ∈ *valid-states PROB*
using *assms a valid-as-valid-exec*
by *blast*
then have *l* ∈ *valid-states PROB*
using *a*
by *simp*
}
then show ∀ *l* ∈ *reachable-s PROB s*. *l* ∈ *valid-states PROB*
by *blast*
qed

lemma *reachable-s-finite-thm-1*:
assumes ((*s* :: *'a state*) ∈ *valid-states PROB*)
shows (*reachable-s PROB s* ⊆ *valid-states PROB*)
using *assms reachable-s-finite-thm-1-a*
by *blast*

— NOTE second declaration skipped (this is declared twice in the source; see above)
— NOTE type for 's' had to be fixed (type mismatch in goal).
lemma *reachable-s-finite-thm*:
fixes *s* :: *'a state*
assumes *finite* (*PROB* :: *'a problem*) (*s* ∈ *valid-states PROB*)
shows *finite* (*reachable-s PROB s*)
using *assms*
by (*meson FINITE-valid-states reachable-s-finite-thm-1 rev-finite-subset*)

lemma *empty-plan-is-valid*: [] ∈ (*valid-plans PROB*)
by (*simp add*: *valid-plans-def*)

lemma *valid-head-and-tail-valid-plan*:
  assumes ($h \in PROB$) ($as \in$ *valid-plans* $PROB$)
  shows ((h # as) $\in$ *valid-plans* $PROB$)
  using *assms*
  by (*auto simp*: *valid-plans-def*)

— TODO refactor
— NOTE added lemma
lemma *lemma-1-reachability-s-i*:
  fixes *PROB s*
  assumes $s \in$ *valid-states PROB*
  shows $s \in$ *reachable-s PROB s*
proof −
  have [] $\in$ *valid-plans PROB*
    using *empty-plan-is-valid*
    by *blast*
  then show *?thesis*
    unfolding *reachable-s-def*
    by *force*
qed

— NOTE types for 'PROB' and 's' had to be fixed (type mismatch in goal).
lemma *lemma-1-reachability-s*:
  fixes *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as*
  assumes ($s \in$ *valid-states PROB*) ($as \in$ *valid-plans PROB*)
  shows ((*last* ‘ *state-set* (*state-list s as*)) $\subseteq$ (*reachable-s PROB s*))
  using *assms*
proof(*induction as arbitrary*: *PROB s*)
  case *Nil*
  then have (*last* ‘ *state-set* (*state-list s* [])) = {*s*}
    by *force*
  then show *?case*
    unfolding *reachable-s-def*
    using *empty-plan-is-valid*
    by *force*
next
  case *cons*: (*Cons a as*)
  let *?S*=*last* ‘ *state-set* (*state-list s* (*a* # *as*))
  {
    let *?as*=[]
    have *last* [*s*] = *exec-plan s ?as*
      by *simp*
    moreover have *?as* $\in$ *valid-plans PROB*
      using *empty-plan-is-valid*
      by *auto*
    ultimately have $\exists$ *as*. (*last* [*s*] = *exec-plan s as*) $\wedge$ *as* $\in$ *valid-plans PROB*

```
    by blast
  }
  note 1 = this
  {
    fix x
    assume P: x ∈ ?S
    then consider
      (a) x = last [s]
      | (b) x ∈ last ` ((#) s ` state-set (state-list (state-succ s a) as))
      by auto
    then have x ∈ reachable-s PROB s
    proof (cases)
      case a
      then have x = s
        by simp
      then show ?thesis
        using cons.prems(1) P lemma-1-reachability-s-i
        by blast
    next
      case b
      then obtain x'' where i:
        x'' ∈ state-set (state-list (state-succ s a) as)
        x = last (s # x'')
        by blast
      then show ?thesis
      proof (cases x'')
        case Nil
        then have x = s
          using i
          by fastforce
        then show ?thesis
          using cons.prems(1) lemma-1-reachability-s-i
          by blast
      next
        case (Cons a' list)
        then obtain x' where a:
          last (a' # list) = last x' x' ∈ state-set (state-list (state-succ s a) as)
          using i(1)
          by blast
        {
          have state-succ s a ∈ valid-states PROB
            using cons.prems(1, 2) valid-action-valid-succ valid-plan-valid-head
            by metis
          moreover have as ∈ valid-plans PROB
            using cons.prems(2) valid-plan-valid-tail
            by fast
          ultimately have
               last ` state-set (state-list (state-succ s a) as) ⊆ reachable-s PROB
(state-succ s a)
```

using *cons.IH*[*of state-succ s a*]
by *auto*
then have ∃ *as′*.
last (*a′* # *list*) = *exec-plan* (*state-succ s a*) *as′* ∧ (*as′* ∈ (*valid-plans PROB*))
unfolding *state-set.simps state-list.simps reachable-s-def*
using *i*(*1*) *Cons*
by *blast*
}
then obtain *as′* **where** *b*:
last (*a′* # *list*) = *exec-plan* (*state-succ s a*) *as′* (*as′* ∈ (*valid-plans PROB*))
by *blast*
then have *x* = *exec-plan* (*state-succ s a*) *as′*
using *i*(*2*) *Cons a*(*1*)
by *auto*
then show *?thesis* **unfolding** *reachable-s-def*
using *cons.prems*(*2*) *b*(*2*)
by (*metis* (*mono-tags*, *lifting*) *exec-plan.simps*(*2*) *mem-Collect-eq valid-head-and-tail-valid-plan valid-plan-valid-head*)
qed
qed
}
then show *?case*
by *blast*
qed

— NOTE types for 'PROB' and 's' had to be fixed for use of 'lemma_1_reachability_s'.
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 −
{
fix *x*
assume *P1*: *x* ∈ *state-set* (*state-list s as*)
have *a*: *last* ‘ *state-set* (*state-list s as*) ⊆ *reachable-s PROB s*
using *assms*(*1*, *2*) *lemma-1-reachability-s*
by *fast*
then have ∀ *as PROB*. *s* ∈ (*valid-states PROB*) ∧ *as* ∈ (*valid-plans PROB*) ⟶ (*last* ‘ (*state-set* (*state-list s as*)) ⊆ *reachable-s PROB s*)
using *lemma-1-reachability-s*

by *fast*
then have *last x ∈ valid-states PROB*
using *assms(1, 2) P1 lemma-1*
by *fast*
then have *last x ∈ reachable-s PROB s*
using *P1 a*
by *fast*
}
note *1 = this*

Show the goal by disproving the contradiction.

```
{
  assume C: (∀ 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))
  moreover {
    fix slist-1 slist-2
    assume C1: slist-1 ∈ state-set (state-list s as) slist-2 ∈ state-set (state-list s
as)
      (last slist-1 = last slist-2)
    moreover have i: (length slist-1 = length slist-2)
      using C1 C
      by blast
    moreover have slist-1 = slist-2
      using C1(1, 2) i neq-mems-state-set-neq-len
      by auto
    ultimately have inj-on last (state-set (state-list s as))
      unfolding inj-on-def
      using C neq-mems-state-set-neq-len
      by blast
    then have False
      using 1 inj-def assms(3)
      by blast
  }
  ultimately have False
    by (metis empty-state-list-lemma nempty-sl-in-state-set)
}
then show ?thesis
  by blast
qed
```

— NOTE added lemma (translation of ‘PHP‘ in pred_setScript.sml:3155).

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 –
{

```
    assume C: inj f s t
    then have 1: (∀ x∈s. f x ∈ t) inj-on f s
      unfolding inj-def
      by blast+
    moreover {
      have f ‘ s ⊆ t
        using 1
        by fast
      then have card (f ‘ s) ≤ card t
        using assms(1) card-mono
        by auto
    }
    moreover have card (f ‘ s) = card s
      using 1 card-image
      by fast
    ultimately have False
      using assms(2)
      by linarith
  }
  then show ?thesis
    by blast
qed

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 −
  {
    have Suc (length as) > card (reachable-s PROB s)
      using assms(4)
      by fastforce
    then have card (state-set (state-list s as)) > card (reachable-s PROB s)
      using card-state-set
      by metis
  }
  note 1 = this
  {
    have finite (reachable-s PROB s)
      using assms(1, 2) reachable-s-finite-thm
      by blast
    then have ¬(inj last (state-set (state-list s as)) (reachable-s PROB s))
      using assms(4) 1 lemma-2-reachability-s-i
      by blast
  }
  note 2 = this
```

obtain *slist-1 slist-2* **where** *3*:
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*
using *assms*(*2*, *3*) *2 not-eq-last-diff-paths-reachability-s*
by *blast*
then show *?thesis* **using** *assms*
proof(*cases as*)
case (*Cons a list*)
then show *?thesis*
using *assms*(*2*, *3*) *3 eq-last-state-imp-append-nempty-as state-set-thm list.distinct*(*1*)
by *metis*
qed *force*
qed

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 −
obtain *as1 as2 as3* :: *'a action list* **where** *1*:
(*as1* @ *as2* @ *as3* = *as*) (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*) ~(*as2*=[])
using *assms lemma-2-reachability-s*
by *metis*
then have (*exec-plan s* (*as1* @ *as2*) = *exec-plan s as1*)
using *1*
by *blast*
then have *2*: *exec-plan s* (*as1* @ *as3*) = *exec-plan s* (*as1* @ *as2* @ *as3*)
using *1 cycle-removal-lemma*
by *fastforce*
let *?as'* = *as1* @ *as3*
have *3*: *exec-plan s as* = *exec-plan s ?as'*
using *1 2*
by *argo*
then have *as2* ≠ []
using *1*
by *blast*
then have *4*: *length ?as'* < *length as*
using *nempty-list-append-length-add 1*
by *blast*
then have *subseq ?as' as*
using *1 subseq-append'*
by *blast*
then show *?thesis*

using *3 4*
by *blast*
qed

— NOTE type for 'as' had to be fixed (type mismatch in goal).
lemma *main-lemma-reachability-s*:
fixes *PROB* :: *'a problem* **and** *as* **and** *s* :: *'a state*
assumes *finite PROB* ($s \in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
shows ($\exists$ *as'*.
(*exec-plan s as* = *exec-plan s as'*) $\wedge$ *subseq as' as*
$\wedge$ (*length as'* $\leq$ (*card* (*reachable-s PROB s*) $-$ *1*)))
proof (*cases length as* $\leq$ *card* (*reachable-s PROB s*) $-$ *1*)
case *False*
let *?as'* = *as*
have *length as* > *card* (*reachable-s PROB s*) $-$ *1*
using *False*
by *simp*
{
fix *p*
assume *P*: *exec-plan s as* = *exec-plan s p subseq p as*
card (*reachable-s PROB s*) $-$ *1* < *length p*
moreover have *p* $\in$ *valid-plans PROB*
using *assms*(*3*) *P*(*2*) *sublist-valid-is-valid*
by *blast*
ultimately obtain *as'* **where** *1*:
exec-plan s p = *exec-plan s as' length as'* < *length p subseq as' p*
using *assms lemma-3-reachability-s*
by *blast*
then have *exec-plan s as* = *exec-plan s as'*
using *P*
by *presburger*
moreover have *subseq as' as*
using *P 1 sublist-trans*
by *blast*
ultimately have ($\exists$ *p'*. (*exec-plan s as* = *exec-plan s p'* $\wedge$ *subseq p' as*) $\wedge$ *length p'* < *length p*)
using *1*
by *blast*
}
then have $\forall$ *p*.
exec-plan s as = *exec-plan s p* $\wedge$ *subseq p as*
$\longrightarrow$ ($\exists$ *p'*.
(*exec-plan s as* = *exec-plan s p'* $\wedge$ *subseq p' as*)
$\wedge$ *length p'* $\leq$ *card* (*reachable-s PROB s*) $-$ *1*)
using *general-theorem*[*of* λ*as''*. (*exec-plan s as* = *exec-plan s as''*) $\wedge$ *subseq as'' as*
(*card* (*reachable-s* (*PROB* :: *'a problem*) (*s* :: *'a state*)) $-$ *1*) *length*]
by *blast*

then show *?thesis*
by *blast*
qed *blast*

lemma *reachable-s-non-empty*: ¬(*reachable-s PROB s* = {})
using *empty-plan-is-valid reachable-s-def*
by *blast*

lemma *card-reachable-s-non-zero*:
fixes *s*
assumes *finite* (*PROB* :: *'a problem*) (*s* ∈ *valid-states PROB*)
shows (*0* < *card* (*reachable-s PROB s*))
using *assms*
by (*simp add*: *card-gt-0-iff reachable-s-finite-thm reachable-s-non-empty*)

lemma *exec-fdom-empty-prob*:
fixes *s*
assumes (*prob-dom PROB* = {}) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (*exec-plan s as* = *fmempty*)
proof −
have *fmdom' s* = {}
using *assms*(*1*, *2*)
by (*simp add*: *valid-states-def*)
then show *exec-plan s as* = *fmempty*
using *assms*(*1*, *3*)
by (*metis*
exec-plan-fdom-subset fmrestrict-set-dom fmrestrict-set-null subset-empty
sup-bot.left-neutral)
qed

— NOTE types for 'PROB' and 's' had to be fixed (type mismatch in goal).
lemma *reachable-s-empty-prob*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state*
assumes (*prob-dom PROB* = {}) (*s* ∈ *valid-states PROB*)
shows ((*reachable-s PROB s*) ⊆ {*fmempty*})
proof −
{
fix *x*
assume *P1*: *x* ∈ *reachable-s PROB s*
then obtain *as* :: *'a action list* **where** *a*:
as ∈ *valid-plans PROB x* = *exec-plan s as*
using *reachable-s-def*
by *blast*
then have *as* ∈ *valid-plans PROB x* = *exec-plan s as*

```
      using a
      by auto
    then have x = fmempty using assms(1, 2) exec-fdom-empty-prob
      by blast
  }
  then show ((reachable-s PROB s) ⊆ {fmempty})
    by blast
qed
```

— NOTE this is semantically equivalent to ‘sublist_valid_is_valid‘.
— NOTE Renamed to ’sublist_valid_plan_alt’ because another lemma by the same name is declared later.

```
lemma sublist-valid-plan--alt:
  assumes (as1 ∈ valid-plans PROB) (subseq as2 as1)
  shows (as2 ∈ valid-plans PROB)
  using assms
  by (auto simp add: sublist-valid-is-valid)
```

```
lemma fmsubset-eq:
  assumes s1 ⊆f s2
  shows (∀ a. a |∈| fmdom s1 ⟶ fmlookup s1 a = fmlookup s2 a)
  using assms
  by (metis (mono-tags, lifting) domIff fmdom-notI fmsubset.rep-eq map-le-def)
```

— NOTE added lemma.
— TODO refactor/move into ’FmapUtils.thy’.

```
lemma submap-imp-state-succ-submap-a:
  assumes s1 ⊆f s2 s2 ⊆f s3
  shows s1 ⊆f s3
  using assms fmsubset.rep-eq map-le-trans
  by blast
```

— NOTE added lemma.
— TODO refactor into FmapUtils?

```
lemma submap-imp-state-succ-submap-b:
  assumes s1 ⊆f s2
  shows (s0 ++ s1) ⊆f (s0 ++ s2)
proof –
  {
    assume C: ¬((s0 ++ s1) ⊆f (s0 ++ s2))
    then have 1: (s0 ++ s1) = (s1 ++f s0)
      using fmap-add-ltr-def
      by blast
    then have 2: (s0 ++ s2) = (s2 ++f s0)
      using fmap-add-ltr-def
```

```
      by auto
    then obtain a where 3:
      a |∈| fmdom (s1 ++f s0) ∧ fmlookup (s1 ++f s0) ≠ fmlookup (s2 ++f s0)
      using C 1 2 fmsubset.rep-eq domIff fmdom-notD map-le-def
      by (metis (no-types, lifting))
    then have False
      using assms(1) C proof (cases a |∈| fmdom s1)
      case True
      moreover have fmlookup s1 a = fmlookup s2 a
        by (meson assms(1) calculation fmsubset-eq)
      moreover have fmlookup (s0 ++f s1) a = fmlookup s1 a
        by (simp add: True)
      moreover have a |∈| fmdom s2
        using True calculation(2) fmdom-notD by fastforce
      moreover have fmlookup (s0 ++f s2) a = fmlookup s2 a
        by (simp add: calculation(4))
      moreover have fmlookup (s0 ++f s1) a = fmlookup (s0 ++f s2) a
        using calculation(2, 3, 5)
        by auto
      ultimately show ?thesis
        by (smt 1 2 C assms domIff fmlookup-add fmsubset.rep-eq map-le-def)
    next
      case False
      moreover have fmlookup (s0 ++f s1) a = fmlookup s0 a
        by (auto simp add: False)
      ultimately show ?thesis proof (cases a |∈| fmdom s0)
        case True
        have a |∉| fmdom (s1 ++f s0)
            by (smt 1 2 C UnE assms dom-map-add fmadd.rep-eq fmsubset.rep-eq
map-add-def
              map-add-dom-app-simps(1) map-le-def)
        then show ?thesis
          using 3 by blast
      next
        case False
        then have a |∉| fmdom (s1 ++f s0)
          using ‹fmlookup (s0 ++f s1) a = fmlookup s0 a›
          by force
        then show ?thesis
          using 3
          by blast
      qed
    qed
  }
  then show ?thesis
    by blast
qed
```

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

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 –
  have *1*: *state-succ s1 a* = (*snd a* ++ *s1*)
    using *assms*(*1*)
    by (*simp add*: *state-succ-def*)
  then have *fst a* $\subseteq_f$ *s2*
    using *assms*(*1*, *2*) *submap-imp-state-succ-submap-a*
    by *auto*
  then have *2*: *state-succ s2 a* = (*snd a* ++ *s2*)
    using *1 state-succ-def*
    by *metis*
  then have *snd a* ++ *s1* $\subseteq_f$ *snd a* ++ *s2*
    using *assms*(*2*) *submap-imp-state-succ-submap-b*
    by *fast*
  then show *?thesis*
    using *1 2*
    by *argo*
qed

— NOTE types for ‘a‘, ‘s1‘ and ‘s2‘ had to be fixed (type mismatch in goal).
lemma *pred-dom-subset-succ-submap*:
  fixes *a* :: *'a action* **and** *s1 s2* :: *'a state*
  assumes (*fmdom'* (*fst a*) ⊆ *fmdom' s1*) (*s1* $\subseteq_f$ *s2*)
  shows (*state-succ s1 a* $\subseteq_f$ *state-succ s2 a*)
  using *assms*
  unfolding *state-succ-def*
proof (*auto*)
  assume *P1*: *fmdom'* (*fst a*) ⊆ *fmdom' s1 s1* $\subseteq_f$ *s2 fst a* $\subseteq_f$ *s1 fst a* $\subseteq_f$ *s2*
  then show *snd a* ++ *s1* $\subseteq_f$ *snd a* ++ *s2*
    using *submap-imp-state-succ-submap-b*
    by *fast*
next
  assume *P2*: *fmdom'* (*fst a*) ⊆ *fmdom' s1 s1* $\subseteq_f$ *s2 fst a* $\subseteq_f$ *s1* ¬ *fst a* $\subseteq_f$ *s2*
  then show *snd a* ++ *s1* $\subseteq_f$ *s2*
    using *submap-imp-state-succ-submap-a*
    by *blast*
next
  assume *P3*: *fmdom'* (*fst a*) ⊆ *fmdom' s1 s1* $\subseteq_f$ *s2* ¬ *fst a* $\subseteq_f$ *s1 fst a* $\subseteq_f$ *s2*
  {
    have *a*: *fmlookup s1* $\subseteq_m$ *fmlookup s2*
      using *P3*(*2*) *fmsubset.rep-eq*
      by *blast*
    {
      have ¬(*fmlookup* (*fst a*) $\subseteq_m$ *fmlookup s1*)
        using *P3*(*3*) *fmsubset.rep-eq*

by *blast*
then have $\exists\, v \in$ *dom* (*fmlookup* (*fst a*)). *fmlookup* (*fst a*) $v \neq$ *fmlookup s1 v*
using *map-le-def*
by *fast*
}
then obtain *v* **where** *b*: $v \in$ *dom* (*fmlookup* (*fst a*)) *fmlookup* (*fst a*) $v \neq$ *fmlookup s1 v*
by *blast*
then have *fmlookup* (*fst a*) $v \neq$ *fmlookup s2 v*
using *assms*(*1*) *a* *contra-subsetD fmdom'.rep-eq map-le-def*
by *metis*
then have $\neg$(*fst a* $\subseteq_f$ *s2*)
using *b fmsubset.rep-eq map-le-def*
by *metis*
}
then show *s1* $\subseteq_f$ *snd a* ++ *s2*
using *P3*(*4*)
by *simp*
qed

— NOTE added lemma.
— TODO refactor.
lemma *valid-as-submap-init-submap-exec-i*:
fixes *s a*
shows *fmdom' s* $\subseteq$ *fmdom'* (*state-succ s a*)
proof (*cases fst a* $\subseteq_f$ *s*)
case *True*
then have *state-succ s a* = *s* $++_f$ (*snd a*)
unfolding *state-succ-def*
using *fmap-add-ltr-def*
by *auto*
then have *fmdom'* (*state-succ s a*) = *fmdom' s* $\cup$ *fmdom'* (*snd a*)
using *fmdom'-add*
by *simp*
then show *?thesis*
by *simp*
next
case *False*
then show *?thesis*
unfolding *state-succ-def*
by *simp*
qed

— NOTE types for ‘s1‘ and ‘s2‘ had to be fixed in order to apply ‘pred_dom_subset_succ_submap‘.
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*)
using *assms*
proof (*induction as arbitrary*: *s1 s2*)
case (*Cons a as*)
{
have *ListMem a* (*a* # *as*)
using *elem*
by *fast*
then have *fmdom′* (*fst a*) ⊆ *fmdom′ s1*
using *Cons.prems*(*2*)
by *blast*
then have *state-succ s1 a* $\subseteq_f$ *state-succ s2 a*
using *Cons.prems*(*1*) *pred-dom-subset-succ-submap*
by *fast*
}
note *1* = *this*
{
fix *b*
assume *ListMem b as*
then have *ListMem b* (*a* # *as*)
using *insert*
by *fast*
then have *a*: *fmdom′* (*fst b*) ⊆ *fmdom′ s1*
using *Cons.prems*(*2*)
by *blast*
then have *fmdom′ s1* ⊆ *fmdom′* (*state-succ s1 a*)
using *valid-as-submap-init-submap-exec-i*
by *metis*
then have *fmdom′* (*fst b*) ⊆ *fmdom′* (*state-succ s1 a*)
using *a*
by *simp*
}
then show *?case*
using *1 Cons.IH*[*of* (*state-succ s1 a*) (*state-succ s2 a*)]
by *fastforce*
qed *auto*

lemma *valid-plan-mems*:
assumes (*as* ∈ *valid-plans PROB*) (*ListMem a as*)
shows *a* ∈ *PROB*
using *assms ListMem-iff in-set-conv-decomp valid-append-valid-suff valid-plan-valid-head*
by (*metis*)

— NOTE typing moved into 'fixes' due to type mismatches when using lemma.
— NOTE showcase (this can't be used due to type problems when the type is specified within proposition.
lemma *valid-states-nempty*:

fixes *PROB* :: (('*a*, '*b*) *fmap* × ('*a*, '*b*) *fmap*) *set*
assumes *finite PROB*
shows ∃ *s*. *s* ∈ (*valid-states PROB*)
unfolding *valid-states-def*
using *fmchoice'*[*OF FINITE-prob-dom*[*OF assms*], where *Q* = λ- -. *True*]
by *auto*

lemma *empty-prob-dom-single-val-state*:
assumes (*prob-dom PROB* = {})
shows (∃ *s*. *valid-states PROB* = {*s*})
proof −
{
assume *C*: ¬(∃ *s*. *valid-states PROB* = {*s*})
then have *valid-states PROB* = {*s*. *fmdom' s* = {}}
using *assms*
by (*simp add*: *valid-states-def*)
then have ∃ *s*. *valid-states PROB* = {*s*}
using *empty-domain-fmap-set*
by *blast*
then have *False*
using *C*
by *blast*
}
then show *?thesis*
by *blast*
qed

lemma *empty-prob-dom-imp-empty-plan-always-good*:
fixes *PROB s*
assumes (*prob-dom PROB* = {}) (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (*exec-plan s* [] = *exec-plan s as*)
using *assms empty-plan-is-valid exec-fdom-empty-prob*
by *fastforce*

lemma *empty-prob-dom*:
fixes *PROB*
assumes (*prob-dom PROB* = {})
shows (*PROB* = {(*fmempty*, *fmempty*)} ∨ *PROB* = {})
using *assms*
proof (*cases PROB* = {})
case *False*
have ⋃((λ(*s1*, *s2*). *fmdom' s1* ∪ *fmdom' s2*) ‘ *PROB*) = {}
using *assms*
by (*simp add*: *prob-dom-def action-dom-def*)
then have *1*:∀ *a*∈*PROB*. (λ(*s1*, *s2*). *fmdom' s1* ∪ *fmdom' s2*) *a* = {}

```
    using Union-empty-conv
    by auto
  {
    fix a
    assume P1: a∈PROB
    then have (λ(s1, s2). fmdom' s1 ∪ fmdom' s2) a = {}
      using 1
      by simp
    then have a: fmdom' (fst a) = {} fmdom' (snd a) = {}
      by auto+
    then have b: fst a = fmempty
      using fmrestrict-set-dom fmrestrict-set-null
      by metis
    then have snd a = fmempty
      using a(2) fmrestrict-set-dom fmrestrict-set-null
      by metis
    then have a = (fmempty, fmempty)
      using b surjective-pairing
      by metis
  }
  then have PROB = {(fmempty, fmempty)}
    using False
    by blast
  then show ?thesis
    by blast
qed simp


lemma empty-prob-dom-finite:
  fixes PROB :: 'a problem
  assumes prob-dom PROB = {}
  shows finite PROB
proof −
  consider (i) PROB = {(fmempty, fmempty)} | (ii) PROB = {}
    using assms empty-prob-dom
    by auto
  then show ?thesis by (cases) auto
qed
```

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

```
lemma disj-imp-eq-proj-exec:
  fixes a :: ('a, 'b) fmap × ('a, 'b) fmap and vs s
  assumes (fmdom' (snd a) ∩ vs) = {}
  shows (fmrestrict-set vs s = fmrestrict-set vs (state-succ s a))
proof −
  have disjnt (fmdom' (snd a)) vs
    using assms disjnt-def
    by fast
```

```
  then show ?thesis
    using disj-dom-drest-fupdate-eq state-succ-pair surjective-pairing
    by metis
qed


lemma no-change-vs-eff-submap:
  fixes a vs s
  assumes (fmrestrict-set vs s = fmrestrict-set vs (state-succ s a)) (fst a ⊆f s)
  shows (fmrestrict-set vs (snd a) ⊆f (fmrestrict-set vs s))
proof –
  {
    fix x
    assume P3: x ∈ dom (fmlookup (fmrestrict-set vs (snd a)))
    then have (fmlookup (fmrestrict-set vs (snd a))) x = (fmlookup (fmrestrict-set
vs s)) x
    proof (cases fmlookup (fmrestrict-set vs (snd a)) x)
      case None
      then show ?thesis using P3 by blast
    next
      case (Some y)
      then have fmrestrict-set vs s = fmrestrict-set vs (s ++f snd a)
        using assms
        by (simp add: state-succ-def fmap-add-ltr-def)
     then have fmlookup (fmrestrict-set vs s) = fmlookup (fmrestrict-set vs (s ++f
snd a))
        by auto
      then have 1:
            fmlookup (fmrestrict-set vs s) x
            = (if x ∈ vs then fmlookup (s ++f snd a) x else None)

        using fmlookup-restrict-set
        by metis
      then show ?thesis
      proof (cases x ∈ vs)
        case True
        then have fmlookup (fmrestrict-set vs s) x = fmlookup (s ++f snd a) x
          using True 1
          by auto
        then show ?thesis
          using Some fmadd.rep-eq fmlookup-restrict-set map-add-Some-iff
          by (metis (mono-tags, lifting))
      next
        case False
        then have 1: fmlookup (fmrestrict-set vs s) x = None
          using False 1
          by auto
        then show ?thesis
          using 1 False
```

by *auto*
qed
qed
}
then have (*fmlookup* (*fmrestrict-set vs* (*snd a*)) $\subseteq_m$ *fmlookup* (*fmrestrict-set vs s*))
using *map-le-def*
by *blast*
then show *?thesis*
using *fmsubset.rep-eq*
by *blast*
qed

— NOTE type of 'a' had to be fixed.
lemma *sat-precond-as-proj-3*:
fixes *s* **and** *a* :: ('*a*, '*b*) *fmap* × ('*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 −
have *fmdom'* (*fmrestrict-set vs* (*fmrestrict-set vs* (*snd a*))) = {}
using *assms fmrestrict-set-dom fmrestrict-set-empty fmrestrict-set-null*
by *metis*
{
fix *x*
assume *C*: *x* ∈ *fmdom'* (*snd a*) ∧ *x* ∈ *vs*
then have *a*: *x* ∈ *fmdom'* (*snd a*) *x* ∈ *vs*
using *C*
by *blast+*
then have *fmlookup* (*snd a*) *x* ≠ *None*
using *fmdom'-notI*
by *metis*
then have *fmlookup* (*fmrestrict-set vs* (*snd a*)) *x* ≠ *None*
using *a(2)*
by *force*
then have *x* ∈ *fmdom'* (*fmrestrict-set vs* (*snd a*))
using *fmdom'-notD*
by *metis*
then have *fmdom'* (*fmrestrict-set vs* (*snd a*)) ≠ {}
by *blast*
then have *False*
using *assms*
by *blast*
}
then have ∀ *x*. ¬(*x* ∈ *fmdom'* (*snd a*) ∧ *x* ∈ *vs*)
by *blast*
then have *1*: *fmdom'* (*snd a*) ∩ *vs* = {}
by *blast*
have *disjnt* (*fmdom'* (*snd a*)) *vs*

using *1 disjnt-def*
by *blast*
then show *?thesis*
using *1 disj-imp-eq-proj-exec*
by *metis*
qed

— NOTE type for ‘a‘ had to be fixed (type mismatch in goal).
— TODO showcase (quick win with simp).
lemma *proj-eq-proj-exec-eq*:
fixes *s s′ vs* **and** *a* :: (*′a*, *′b*) *fmap* × (*′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′*))
using *assms*
by (*simp add*: *fmap-add-ltr-def state-succ-def*)

lemma *empty-eff-exec-eq*:
fixes *s a*
assumes (*fmdom′* (*snd a*) = {})
shows (*state-succ s a* = *s*)
using *assms*
unfolding *state-succ-def fmap-add-ltr-def*
by (*metis fmadd-empty*(*2*) *fmrestrict-set-dom fmrestrict-set-null*)

lemma *exec-as-proj-valid-2*:
fixes *a*
assumes *a* ∈ *PROB*
shows (*action-dom* (*fst a*) (*snd a*) ⊆ *prob-dom PROB*)
using *assms*
by (*simp add*: *FDOM-eff-subset-prob-dom-pair FDOM-pre-subset-prob-dom-pair action-dom-def*)

lemma *valid-filter-valid-as*:
assumes (*as* ∈ *valid-plans PROB*)
shows (*filter P as* ∈ *valid-plans PROB*)
using *assms*
by(*auto simp*: *valid-plans-def*)

lemma *sublist-valid-plan*:
assumes (*subseq as′ as*) (*as* ∈ *valid-plans PROB*)
shows (*as′* ∈ *valid-plans PROB*)
using *assms*

by (*auto simp*: *valid-plans-def*) (*meson fset-mp fset-of-list-elem sublist-subset subsetCE*)

lemma *prob-subset-dom-subset*:
assumes *PROB1* ⊆ *PROB2*
shows (*prob-dom PROB1* ⊆ *prob-dom PROB2*)
using *assms*
by (*auto simp add*: *prob-dom-def*)

lemma *state-succ-valid-act-disjoint*:
assumes (*a* ∈ *PROB*) (*vs* ∩ (*prob-dom PROB*) = {})
shows (*fmrestrict-set vs* (*state-succ s a*) = *fmrestrict-set vs s*)
using *assms*
by (*smt*
FDOM-eff-subset-prob-dom-pair disj-imp-eq-proj-exec inf.absorb1
inf-bot-right inf-commute inf-left-commute
)

lemma *exec-valid-as-disjoint*:
fixes *s*
assumes (*vs* ∩ (*prob-dom PROB*) = {}) (*as* ∈ *valid-plans PROB*)
shows (*fmrestrict-set vs* (*exec-plan s as*) = *fmrestrict-set vs s*)
using *assms*
proof (*induction as arbitrary*: *s vs PROB*)
case (*Cons a as*)
then show *?case*
by (*metis exec-plan.simps*(*2*) *state-succ-valid-act-disjoint valid-plan-valid-head valid-plan-valid-tail*)
qed *simp*

definition *state-successors* **where**
state-successors PROB s ≡ ((*state-succ s* ‘ *PROB*) − {*s*})

3.3 State Spaces

definition *stateSpace* **where**
stateSpace ss vs ≡ (∀ *s*. *s* ∈ *ss* ⟶ (*fmdom′ s* = *vs*))

lemma *EQ-SS-DOM*:
assumes ¬(*ss* = {}) (*stateSpace ss vs1*) (*stateSpace ss vs2*)
shows (*vs1* = *vs2*)
using *assms*
by (*auto simp*: *stateSpace-def*)

— NOTE Name 'dom' changed to 'domain' because of name clash with 'Map.dom'.

lemma *FINITE-SS*:
  fixes *ss* :: (*'a*, *bool*) *fmap set*
  assumes ¬(*ss* = {}) (*stateSpace ss domain*)
  shows *finite ss*
proof −
  have *1*: *stateSpace ss domain* = (∀ *s*. *s* ∈ *ss* ⟶ (*fmdom' s* = *domain*))
    by (*simp add*: *stateSpace-def*)
  {
    fix *s*
    assume *P1*: *s* ∈ *ss*
    have *fmdom' s* = *domain*
      using *assms 1 P1*
      by *blast*
    then have *s* ∈ {*s*. *fmdom' s* = *domain*}
      by *auto*
  }
  then have *2*: *ss* ⊆ {*s*. *fmdom' s* = *domain*}
    by *blast*
      — TODO add lemma (finite (fmdom' s))
  then have *finite domain*
    using *1 assms*
    by *fastforce*
  then have *finite* {*s* :: *'a state*. *fmdom' s* = *domain*}
    using *FINITE-states*
    by *blast*
  then show *?thesis*
    using *2 finite-subset*
    by *auto*
qed

lemma *disjoint-effects-no-effects*:
  fixes *s*
  assumes (∀ *a*. *ListMem a as* ⟶ (*fmdom'* (*fmrestrict-set vs* (*snd a*)) = {}))
  shows (*fmrestrict-set vs* (*exec-plan s as*) = (*fmrestrict-set vs s*))
  using *assms*
proof (*induction as arbitrary*: *s vs*)
  case (*Cons a as*)
  then have *ListMem a* (*a* # *as*)
    using *elem*
    by *fast*
  then have *fmdom'* (*fmrestrict-set vs* (*snd a*)) = {}
    using *Cons.prems*(*1*)
    by *blast*
  then have *fmrestrict-set vs* (*state-succ s a*) = *fmrestrict-set vs s*
    using *sat-precond-as-proj-3*
    by *blast*

then show *?case*
by (*simp add*: *Cons.IH Cons.prems insert*)
qed *auto*

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*)
using *assms*
unfolding *state-succ-def*
proof (*auto*)
assume *P1*: *action-needed-asses a s2* $\subseteq_f$ *action-needed-asses a s1 s1* $\subseteq_f$ *s2 fst a* $\subseteq_f$ *s1*
fst a $\subseteq_f$ *s2*
then show *snd a* ++ *s1* $\subseteq_f$ *snd a* ++ *s2*
using *submap-imp-state-succ-submap-b*
by *blast*
next
assume *P2*: *action-needed-asses a s2* $\subseteq_f$ *action-needed-asses a s1 s1* $\subseteq_f$ *s2 fst a* $\subseteq_f$ *s1*
¬ *fst a* $\subseteq_f$ *s2*
then show *snd a* ++ *s1* $\subseteq_f$ *s2*
using *submap-imp-state-succ-submap-a*
by *blast*
next
assume *P3*: *action-needed-asses a s2* $\subseteq_f$ *action-needed-asses a s1 s1* $\subseteq_f$ *s2* ¬ *fst a* $\subseteq_f$ *s1*
fst a $\subseteq_f$ *s2*
let *?vs1*={*v* ∈ *fmdom′ s1*. *v* ∈ *fmdom′* (*fst a*) ∧ *fmlookup* (*fst a*) *v* = *fmlookup s1 v*}
let *?vs2*={*v* ∈ *fmdom′ s2*. *v* ∈ *fmdom′* (*fst a*) ∧ *fmlookup* (*fst a*) *v* = *fmlookup s2 v*}
let *?f*=*fmrestrict-set ?vs1 s1*
let *?g*=*fmrestrict-set ?vs2 s2*
have *1*: *fmdom′ ?f* = *?vs1 fmdom′ ?g* = *?vs2*
unfolding *action-needed-asses-def action-needed-vars-def fmdom′-restrict-set-precise*
by *blast*+
have *2*: *fmlookup ?g* $\subseteq_m$ *fmlookup ?f*

```
    using P3(1)
    unfolding action-needed-asses-def action-needed-vars-def
    using fmsubset.rep-eq
    by blast
  {
    {
      fix v
      assume P3-1: v ∈ fmdom′ ?g
      then have v ∈ fmdom′ s2 v ∈ fmdom′ (fst a) fmlookup (fst a) v = fmlookup
s2 v
        using 1
        by simp+
      then have fmlookup (fst a) v = fmlookup ?g v
        by simp
      then have fmlookup (fst a) v = fmlookup ?f v
        using 2
        by (metis (mono-tags, lifting) P3-1 domIff fmdom′-notI map-le-def)
    }
    then have i: fmlookup (fst a) ⊆m fmlookup ?f
      using P3(4) 1(2)
      by (smt domIff fmdom′-notD fmsubset.rep-eq map-le-def mem-Collect-eq)
    {
      fix v
      assume P3-2: v ∈ dom (fmlookup (fst a))
      then have fmlookup (fst a) v = fmlookup ?f v
        using i
        by (meson domIff fmdom′-notI map-le-def)
      then have v ∈ ?vs1
        using P3-2 1(1)
        by (metis (no-types, lifting) domIff fmdom′-notD)
      then have fmlookup (fst a) v = fmlookup s1 v
        by blast
    }
    then have fst a ⊆f s1
      by (simp add: map-le-def fmsubset.rep-eq)
  }
  then show s1 ⊆f snd a ++ s2
    using P3(3)
    by simp
qed
```

— NOTE added lemma.
— TODO refactor.

```
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*
using *assms*
unfolding *action-needed-asses-def action-needed-vars-def*
using *fmdom′-notI fmlookup-restrict-set*
by (*smt mem-Collect-eq*)

— NOTE added lemma.
— TODO refactor.
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*
using *assms*
by (*meson fmdom′-notI fmdom-notD fmsubset-eq*)

— NOTE added lemma.
— TODO refactor.
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*}
unfolding *action-needed-asses-def action-needed-vars-def*
by (*simp add*: *Set.filter-def fmfilter-alt-defs*(*4*))

— NOTE added lemma.
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*
using *assms*
proof −
have *1*: *v* ∈ {*v* ∈ *fmdom′ s*. *v* ∈ *fmdom′* (*fst a*) ∧ *fmlookup* (*fst a*) *v* = *fmlookup s v*}
using *assms as-needed-asses-submap-exec-iii*
by *metis*
then have *2*: *fmlookup* (*action-needed-asses a s*) *v* = *fmlookup s v*
unfolding *action-needed-asses-def action-needed-vars-def*
by *force*
moreover have *3*: *fmlookup* (*action-needed-asses a s*) *v* = *fmlookup* (*fst a*) *v*
using *1 2*
by *simp*
moreover have *fmlookup* (*fst a*) *v* = *fmlookup s v*
using *2 3*
by *argo*
ultimately show *?thesis*

  by *blast*
qed

— NOTE added lemma.
— TODO refactor (into Fmap_Utils.thy).
lemma *as-needed-asses-submap-exec-v*:
 fixes *f g v*
 assumes $v \in fmdom'\ f$ $f \subseteq_f g$
 shows $v \in fmdom'\ g$
proof −
 obtain *b* **where** *1*: *fmlookup f v = b* $b \neq None$
  using *assms*(*1*)
  by (*meson fmdom'-notI*)
 then have *fmlookup g v = b*
  using *as-needed-asses-submap-exec-ii*[*OF assms*]
  by *argo*
 then show *?thesis*
  using *1 fmdom'-notD*
  by *fastforce*
qed

— NOTE added lemma.
— TODO refactor.
lemma *as-needed-asses-submap-exec-vi*:
 fixes *a s1 s2 v*
 assumes $v \in fmdom'$ (*action-needed-asses a s1*)
  (*action-needed-asses a s1*) $\subseteq_f$ (*action-needed-asses a s2*)
 shows
  (*fmlookup* (*action-needed-asses a s1*) *v*) = *fmlookup* (*fst a*) *v*
  ∧ (*fmlookup* (*action-needed-asses a s2*) *v*) = *fmlookup* (*fst a*) *v* ∧
  fmlookup s1 v = *fmlookup* (*fst a*) *v* ∧ *fmlookup s2 v* = *fmlookup* (*fst a*) *v*
 using *assms*
proof −
 have *1*:
  fmlookup (*action-needed-asses a s1*) *v* = *fmlookup s1 v*
  fmlookup (*action-needed-asses a s1*) *v* = *fmlookup* (*fst a*) *v*
  fmlookup (*fst a*) *v* = *fmlookup s1 v*
  using *as-needed-asses-submap-exec-iv*[*OF assms*(*1*)]
  by *blast+*
 moreover {
  have *fmlookup* (*action-needed-asses a s1*) *v* = *fmlookup* (*action-needed-asses a s2*) *v*
   using *as-needed-asses-submap-exec-ii*[*OF assms*]
   by *simp*
  then have *fmlookup* (*action-needed-asses a s2*) *v* = *fmlookup* (*fst a*) *v*
   using *1*(*2*)
   by *argo*
 }
 note *2* = *this*

moreover {
  have *v* ∈ *fmdom′* (*action-needed-asses a s2*)
    using *as-needed-asses-submap-exec-v*[*OF assms*]
    by *simp*
  then have *fmlookup s2 v* = *fmlookup* (*action-needed-asses a s2*) *v*
    using *as-needed-asses-submap-exec-i*
    by *metis*
  also have . . . = *fmlookup* (*fst a*) *v*
    using *2*
    by *simp*
  finally have *fmlookup s2 v* = *fmlookup* (*fst a*) *v*
    by *simp*
}
ultimately show *?thesis*
  by *argo*
qed

— TODO refactor.
— NOTE added lemma.
lemma *as-needed-asses-submap-exec-vii*:
  fixes *f g v*
  assumes ∀ *v* ∈ *fmdom′ f*. *fmlookup f v* = *fmlookup g v*
  shows $f \subseteq_f g$
proof −
  {
    fix *v*
    assume *a*: *v* ∈ *fmdom′ f*
    then have *v* ∈ *dom* (*fmlookup f*)
      by *simp*
    moreover have *fmlookup f v* = *fmlookup g v*
      using *assms a*
      by *blast*
    ultimately have *v* ∈ *dom* (*fmlookup f*) ⟶ *fmlookup f v* = *fmlookup g v*
      by *blast*
  }
  then have *fmlookup f* $\subseteq_m$ *fmlookup g*
    by (*simp add*: *map-le-def*)
  then show *?thesis*
    by (*simp add*: *fmsubset.rep-eq*)
qed

— TODO refactor.
— NOTE added lemma.
lemma *as-needed-asses-submap-exec-viii*:
  fixes *f g v*
  assumes $f \subseteq_f g$
  shows ∀ *v* ∈ *fmdom′ f*. *fmlookup f v* = *fmlookup g v*
proof −
  have *1*: *fmlookup f* $\subseteq_m$ *fmlookup g*

using *assms*
by (*simp add*: *fmsubset.rep-eq*)
{
fix *v*
assume $v \in$ *fmdom′ f*
then have $v \in$ *dom* (*fmlookup f*)
by *simp*
then have *fmlookup f v* = *fmlookup g v*
using *1 map-le-def*
by *metis*
}
then show *?thesis*
by *blast*
qed

— NOTE added lemma.
lemma *as-needed-asses-submap-exec-viii′*:
fixes *f g v*
assumes $f \subseteq_f g$
shows *fmdom′ f* $\subseteq$ *fmdom′ g*
using *assms as-needed-asses-submap-exec-v subsetI*
by *metis*

— NOTE added lemma.
— TODO refactor.
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*)
using *as-needed-asses-submap-exec-vii as-needed-asses-submap-exec-viii*
by *metis*

— NOTE added lemma.
lemma *as-needed-asses-submap-exec-x*:
fixes *f a v*
assumes $v \in$ *fmdom′* (*action-needed-asses a f*)
shows $v \in$ *fmdom′* (*fst a*) $\wedge$ $v \in$ *fmdom′ f* $\wedge$ *fmlookup* (*fst a*) *v* = *fmlookup f v*
using *assms*
unfolding *action-needed-asses-def action-needed-vars-def*
using *as-needed-asses-submap-exec-i assms*
by (*metis fmdom′-notD fmdom′-notI*)

— NOTE added lemma.
— TODO refactor.
lemma *as-needed-asses-submap-exec-xi*:
fixes *v a f g*
assumes $v \in$ *fmdom′* (*action-needed-asses a* (*f* ++ *g*)) $v \in$ *fmdom′ f*
shows
fmlookup (*action-needed-asses a* (*f* ++ *g*)) *v* = *fmlookup f v*
$\wedge$ *fmlookup* (*action-needed-asses a* (*f* ++ *g*)) *v* = *fmlookup* (*fst a*) *v*

proof −
  have *1*: $v \in \{v \in fmdom'\ (f ++ g).\ v \in fmdom'\ (fst\ a) \wedge fmlookup\ (fst\ a)\ v = fmlookup\ (f ++ g)\ v\}$
    using *as-needed-asses-submap-exec-x*[*OF assms*(*1*)]
    by *blast*
  {
    have *v* |∈| *fmdom f*
      using *assms*(*2*)
      by (*meson fmdom′-notI fmdom-notD*)
    then have *fmlookup* (*f* ++ *g*) *v* = *fmlookup f v*
      unfolding *fmap-add-ltr-def fmlookup-add*
      by *simp*
  }
  note *2* = *this*
  {

    have *fmlookup* (*action-needed-asses a* (*f* ++ *g*)) *v* = *fmlookup* (*f* ++ *g*) *v*
      unfolding *action-needed-asses-def action-needed-vars-def*
      using *1*
      by *force*
    then have *fmlookup* (*action-needed-asses a* (*f* ++ *g*)) *v* = *fmlookup f v*
      using *2*
      by *simp*
  }
  note *3* = *this*
  moreover {
    have *fmlookup* (*fst a*) *v* = *fmlookup* (*f* ++ *g*) *v*
      using *1*
      by *simp*
    also have ... = *fmlookup f v*
      using *2*
      by *simp*
    also have ... = *fmlookup* (*action-needed-asses a* (*f* ++ *g*)) *v*
      using *3*
      by *simp*
    finally have *fmlookup* (*action-needed-asses a* (*f* ++ *g*)) *v* = *fmlookup* (*fst a*) *v*
      by *simp*
  }
  ultimately show *?thesis*
    by *blast*
qed

— NOTE added lemma.
— TODO refactor (into Fmap_Utils.thy).
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 −

```
  have v |∈| fmdom f
    using assms(1) fmdom'-notI fmdom-notD
    by metis
  then show ?thesis
    unfolding fmap-add-ltr-def
    using fmlookup-add
    by force
qed

— NOTE added lemma.
lemma as-needed-asses-submap-exec-xii':
  fixes f g v
  assumes v ∉ fmdom' f v ∈ fmdom' g
  shows fmlookup (f ++ g) v = fmlookup g v
proof −
  have ¬(v |∈| fmdom f)
    using assms(1) fmdom'-notI fmdom-notD
    by fastforce
  moreover have v |∈| fmdom g
    using assms(2) fmdom'-notI fmdom-notD
    by metis
  ultimately show ?thesis
    unfolding fmap-add-ltr-def
    using fmlookup-add
    by simp
qed


— NOTE showcase.
lemma as-needed-asses-submap-exec:
  fixes s1 s2
  assumes (s1 ⊆f s2)
    (∀ a. ListMem a as ⟶ (action-needed-asses a s2 ⊆f action-needed-asses a s1))
  shows (exec-plan s1 as ⊆f exec-plan s2 as)
  using assms
proof (induction as arbitrary: s1 s2)
  case (Cons a as)
    — Proof the premises of the induction hypothesis for 'state_succ s1 a' and
'state_succ s2 a'.
  {
    then have action-needed-asses a s2 ⊆f action-needed-asses a s1
      using Cons.prems(2) elem
      by metis
    then have state-succ s1 a ⊆f state-succ s2 a
      using Cons.prems(1) act-needed-asses-submap-succ-submap
      by blast
  }
  note 1 = this
  moreover {
```

```
fix a'
assume P: ListMem a' as
  — Show the goal by rule 'as_needed_asses_submap_exec_ix'.
let ?f=action-needed-asses a' (state-succ s2 a)
let ?g=action-needed-asses a' (state-succ s1 a)
{
  fix v
  assume P-1: v ∈ fmdom' ?f
  then have fmlookup ?f v = fmlookup ?g v
    unfolding state-succ-def
```

Split cases on the if-then branches introduced by the definition of 'state_succ'.

```
  proof (auto)
     assume P-1-1: v ∈ fmdom' (action-needed-asses a' (snd a ++ s2)) fst a
⊆f s2
      fst a ⊆f s1
    have i: action-needed-asses a' s2 ⊆f action-needed-asses a' s1
      using Cons.prems(2) P insert
      by fast
    then show
        fmlookup (action-needed-asses a' (snd a ++ s2)) v
        = fmlookup (action-needed-asses a' (snd a ++ s1)) v
    proof (cases v ∈ fmdom' ?g)
      case true: True
      then have A:
        v ∈ fmdom' (fst a') ∧ v ∈ fmdom' (snd a ++ s1)
            ∧ fmlookup (fst a') v = fmlookup (snd a ++ s1) v
        using as-needed-asses-submap-exec-x[OF true]
        unfolding state-succ-def
        using P-1-1(3)
        by simp
      then have B:
        v ∈ fmdom' (fst a') ∧ v ∈ fmdom' (snd a ++ s2)
            ∧ fmlookup (fst a') v = fmlookup (snd a ++ s2) v
        using as-needed-asses-submap-exec-x[OF P-1]
        unfolding state-succ-def
        using P-1-1(2)
        by simp
      then show ?thesis
      proof (cases v ∈ fmdom' (snd a))
        case True
        then have I:
          fmlookup (snd a ++ s2) v = fmlookup (snd a) v
          fmlookup (snd a ++ s1) v = fmlookup (snd a) v
          using as-needed-asses-submap-exec-xii
          by fast+
        moreover {
          have fmlookup ?f v = fmlookup (snd a ++ s2) v
            using as-needed-asses-submap-exec-iv[OF P-1]
```

unfolding *state-succ-def*
using *P-1-1*(*2*)
by *presburger*
then have *fmlookup ?f v* = *fmlookup* (*snd a*) *v*
using *I*(*1*)
by *argo*
}
moreover {
have *fmlookup ?g v* = *fmlookup* (*snd a* ++ *s1*) *v*
using *as-needed-asses-submap-exec-iv*[*OF true*]
unfolding *state-succ-def*
using *P-1-1*(*3*)
by *presburger*
then have *fmlookup ?g v* = *fmlookup* (*snd a*) *v*
using *I*(*2*)
by *argo*
}
ultimately show *?thesis*
unfolding *state-succ-def*
using *P-1-1*(*2*, *3*)
by *presburger*
next
case *False*
then have *I*: *v* ∈ *fmdom′ s1 v* ∈ *fmdom′ s2*
using *A B*
unfolding *fmap-add-ltr-def fmdom′-add*
by *blast+*
{
have *fmlookup ?g v* = *fmlookup* (*snd a* ++ *s1*) *v*
using *as-needed-asses-submap-exec-iv*[*OF true*]
unfolding *state-succ-def*
using *P-1-1*(*3*)
by *presburger*
then have *fmlookup ?g v* = *fmlookup s1 v*
using *as-needed-asses-submap-exec-xii′*[*OF False I*(*1*)]
by *simp*
moreover {
have *fmlookup* (*snd a* ++ *s1*) *v* = *fmlookup s1 v*
using *as-needed-asses-submap-exec-xii′*[*OF False I*(*1*)]
by *simp*
moreover from ‹*fmlookup* (*snd a* ++ *s1*) *v* = *fmlookup s1 v*›
have *fmlookup* (*fst a′*) *v* = *fmlookup s1 v*
using *A*(*1*)
by *argo*
ultimately have *fmlookup* (*action-needed-asses a′ s1*) *v* = *fmlookup s1 v*
using *A*(*1*) *I*(*1*)
unfolding *action-needed-asses-def action-needed-vars-def fmlookup-restrict-set*

```
        by simp
      }
      ultimately have fmlookup ?g v = fmlookup (action-needed-asses a' s1)
v
        by argo
    }
    note II = this
    {
      have fmlookup ?f v = fmlookup (snd a ++ s2) v
        using as-needed-asses-submap-exec-iv[OF P-1]
        unfolding state-succ-def
        using P-1-1(2)
        by presburger
      moreover from ‹fmlookup ?f v = fmlookup (snd a ++ s2) v›
      have α: fmlookup ?f v = fmlookup s2 v
        using as-needed-asses-submap-exec-xii'[OF False I(2)]
        by argo
      ultimately have fmlookup (snd a ++ s2) v = fmlookup s2 v
        by argo
      moreover {
        from ‹fmlookup (snd a ++ s2) v = fmlookup s2 v›
        have fmlookup (fst a') v = fmlookup s2 v
          using B(1)
          by argo
        then have fmlookup (action-needed-asses a' s2) v = fmlookup s2 v
          using B(1) I(2)
          unfolding action-needed-asses-def action-needed-vars-def
            fmlookup-restrict-set
          by simp
      }
      ultimately have fmlookup ?f v = fmlookup (action-needed-asses a' s2)
v
        using α
        by argo
    }
    note III = this
    {
      have v ∈ fmdom' (action-needed-asses a' s2)
      proof −
        have fmlookup (fst a') v = fmlookup s1 v
          by (simp add: A False I(1) as-needed-asses-submap-exec-xii')
        then show ?thesis
          by (simp add: A Cons.prems(1) I(1, 2)
              as-needed-asses-submap-exec-ii as-needed-asses-submap-exec-iii)
      qed
      then have
              fmlookup (action-needed-asses a' s2) v
              = fmlookup (action-needed-asses a' s1) v
        using i as-needed-asses-submap-exec-ix[of action-needed-asses a' s2
```

action-needed-asses a′ s1]
by blast
}
note IV = this
{
have fmlookup ?f v = fmlookup (action-needed-asses a′ s2) v
using III
by simp
also have … = fmlookup (action-needed-asses a′ s1) v
using IV
by simp
finally have … = fmlookup ?g v
using II
by simp
}
then show ?thesis
unfolding action-needed-asses-def action-needed-vars-def state-succ-def
using P-1-1 A B
by simp
qed
next
case false: False
have A:
v ∈ fmdom′ (fst a′) ∧ v ∈ fmdom′ (snd a ++ s2)
∧ fmlookup (fst a′) v = fmlookup (snd a ++ s2) v
using as-needed-asses-submap-exec-x[OF P-1]
unfolding state-succ-def
using P-1-1(2)
by simp
from false have B:
¬(v ∈ fmdom′ (snd a ++ s1)) ∨ ¬(fmlookup (fst a′) v = fmlookup (snd a ++ s1) v)
by (simp add: A P-1-1(3) as-needed-asses-submap-exec-iii state-succ-def)
then show ?thesis
proof (cases v ∈ fmdom′ (snd a))
case True
then have I: v ∈ fmdom′ (snd a ++ s1)
unfolding fmap-add-ltr-def fmdom′-add
by simp
{
from True have
fmlookup (snd a ++ s2) v = fmlookup (snd a) v
fmlookup (snd a ++ s1) v = fmlookup (snd a) v
using as-needed-asses-submap-exec-xii
by fast+
then have fmlookup (snd a ++ s1) v = fmlookup (snd a ++ s2) v
by auto
also have … = fmlookup (fst a′) v
using A

```
        by simp
      finally have fmlookup (snd a ++ s1) v = fmlookup (fst a') v
        by simp
    }
    then show ?thesis using B I
      by presburger
  next
    case False
    then have I: v ∈ fmdom' s2
      using A unfolding fmap-add-ltr-def fmdom'-add
      by blast
    {
      from P-1 have fmlookup ?f v ≠ None
        by (meson fmdom'-notI)
      moreover from false
      have fmlookup ?g v = None
        by (simp add: fmdom'-notD)
      ultimately have fmlookup ?f v ≠ fmlookup ?g v
        by simp
    }
    moreover
    {
      {
        from P-1-1(2) have state-succ s2 a = snd a ++ s2
          unfolding state-succ-def
          by simp
        moreover from ‹state-succ s2 a = snd a ++ s2› have
          fmlookup (state-succ s2 a) v = fmlookup s2 v
          using as-needed-asses-submap-exec-xii'[OF False I]
          by simp
         ultimately have fmlookup ?f v = fmlookup (action-needed-asses a'
s2) v
          unfolding action-needed-asses-def action-needed-vars-def
          by (simp add: A I)
      }
      note I = this
      moreover {
        from P-1-1(3) have state-succ s1 a = snd a ++ s1
          unfolding state-succ-def
          by simp
        moreover from ‹state-succ s1 a = snd a ++ s1› False
        have fmlookup (state-succ s1 a) v = fmlookup s1 v
          unfolding fmap-add-ltr-def
          using fmlookup-add
          by (simp add: fmdom'-alt-def)
         ultimately have fmlookup ?g v = fmlookup (action-needed-asses a'
s1) v
          unfolding action-needed-asses-def action-needed-vars-def
          using FDOM-state-succ-subset
```

```
          by auto
        }
        moreover {
          have v ∈ fmdom' (action-needed-asses a' s2)
          proof −
            have v ∈ fmdom' s2 ∪ fmdom' (snd a)
                    by (metis (no-types) A FDOM-state-succ-subset P-1-1(2)
state-succ-def subsetCE)
            then show ?thesis
          by (simp add: A False as-needed-asses-submap-exec-iii as-needed-asses-submap-exec-xii')
          qed
          then have
               fmlookup (action-needed-asses a' s2) v
               = fmlookup (action-needed-asses a' s1) v
            using i as-needed-asses-submap-exec-ix[of action-needed-asses a' s2
               action-needed-asses a' s1]
            by blast
        }
        ultimately have fmlookup ?f v = fmlookup ?g v
          by simp
      }
      ultimately show ?thesis
        by simp
    qed
  qed
next
  assume P2: v ∈ fmdom' (action-needed-asses a' (snd a ++ s2)) fst a ⊆f
s2
    ¬ fst a ⊆f s1
  then show
     fmlookup (action-needed-asses a' (snd a ++ s2)) v
     = fmlookup (action-needed-asses a' s1) v
  proof −
    obtain aa :: ('a, 'b) fmap ⇒ ('a, 'b) fmap ⇒ 'a where
      ∀ x0 x1. (∃ v2. v2 ∈ fmdom' x1
         ∧ fmlookup x1 v2 ≠ fmlookup x0 v2) = (aa x0 x1 ∈ fmdom' x1
         ∧ fmlookup x1 (aa x0 x1) ≠ fmlookup x0 (aa x0 x1))
      by moura
    then have f1: ∀ f fa. aa fa f ∈ fmdom' f
       ∧ fmlookup f (aa fa f) ≠ fmlookup fa (aa fa f) ∨ f ⊆f fa
      by (meson as-needed-asses-submap-exec-vii)
    then have f2: aa s1 (fst a) ∈ fmdom' (fst a)
       ∧ fmlookup (fst a) (aa s1 (fst a)) ≠ fmlookup s1 (aa s1 (fst a))
      using P2(3) by blast
    then have aa s1 (fst a) ∈ fmdom' s2
      by (metis (full-types) P2(2) as-needed-asses-submap-exec-v)
    then have aa s1 (fst a) ∈ fmdom' (action-needed-asses a s2)
      using f2 by (simp add: P2(2) as-needed-asses-submap-exec-iii
         as-needed-asses-submap-exec-viii)
```

then show *?thesis*
using *f1* **by** (*metis* (*no-types*) *Cons.prems*(*2*) *P2*(*3*) *as-needed-asses-submap-exec-vi elem*)
qed
next
assume *P3*: $v \in$ *fmdom′* (*action-needed-asses a′ s2*) $\neg$ *fst a* $\subseteq_f$ *s2 fst a* $\subseteq_f$ *s1*
then show
fmlookup (*action-needed-asses a′ s2*) *v*
= *fmlookup* (*action-needed-asses a′* (*snd a* ++ *s1*)) *v*
using *Cons.prems*(*1*) *submap-imp-state-succ-submap-a*
by *blast*
next
assume *P4*: $v \in$ *fmdom′* (*action-needed-asses a′ s2*) $\neg$ *fst a* $\subseteq_f$ *s2* $\neg$ *fst a* $\subseteq_f$ *s1*
then show
fmlookup (*action-needed-asses a′ s2*) *v*
= *fmlookup* (*action-needed-asses a′ s1*) *v*
by (*simp add*: *Cons.prems*(*2*) *P as-needed-asses-submap-exec-ii insert*)
qed
}
then have *a*: *?f* $\subseteq_f$ *?g*
using *as-needed-asses-submap-exec-ix*
by *blast*
}
note *2* = *this*
then show *?case*
unfolding *exec-plan.simps*
using *Cons.IH*[*of state-succ s1 a state-succ s2 a*, *OF 1*]
by *blast*
qed *simp*

— NOTE name shortened.
definition *system-needed-vars* **where**
system-needed-vars PROB s ≡ ($\bigcup${*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*)
using *assms*
by (*auto simp*: *system-needed-vars-def*) (*metis surjective-pairing*)

lemma *action-needed-asses-submap-sys-needed-asses*:
assumes ($a \in PROB$)
shows (*action-needed-asses a s* $\subseteq_f$ *system-needed-asses PROB s*)
proof −
have *action-needed-asses a s* = *fmrestrict-set* (*action-needed-vars a s*) *s*
unfolding *action-needed-asses-def*
by *simp*
then have *system-needed-asses PROB s* = (*fmrestrict-set* (*system-needed-vars PROB s*) *s*)
unfolding *system-needed-asses-def*
by *simp*
then have *1*: *action-needed-vars a s* ⊆ *system-needed-vars PROB s*
unfolding *action-needed-vars-subset-sys-needed-vars-subset*
using *assms action-needed-vars-subset-sys-needed-vars-subset*
by *fast*
{
fix *x*
assume *P1*: *x* ∈ *dom* (*fmlookup* (*fmrestrict-set* (*action-needed-vars a s*) *s*))
then have *a*: *fmlookup* (*fmrestrict-set* (*action-needed-vars a s*) *s*) *x* = *fmlookup s x*
by (*auto simp*: *fmdom'-restrict-set-precise*)
then have *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) *x* = *fmlookup s x*
using *1 contra-subsetD*
by *fastforce*
then have
fmlookup (*fmrestrict-set* (*action-needed-vars a s*) *s*) *x*
= *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) *x*

using *a*
by *argo*
}
then have
fmlookup (*fmrestrict-set* (*action-needed-vars a s*) *s*)
$\subseteq_m$ *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*)

using *map-le-def*
by *blast*
then show (*action-needed-asses a s* $\subseteq_f$ *system-needed-asses PROB s*)
by (*simp add*: *fmsubset.rep-eq action-needed-asses-def system-needed-asses-def*)
qed

lemma *system-needed-asses-include-action-needed-asses-1*:
assumes ($a \in PROB$)
shows (*action-needed-vars a* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) = *action-needed-vars a s*)
proof −
let *?A*={*v* ∈ *fmdom'* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*).

$v \in$ *fmdom'* (*fst a*)
$\wedge$ *fmlookup* (*fst a*) $v =$ *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) *v*}
let *?B*={$v \in$ *fmdom' s*. $v \in$ *fmdom'* (*fst a*) $\wedge$ *fmlookup* (*fst a*) $v =$ *fmlookup s v*}
{
fix *v*
assume $v \in$ *?A*
then have *i*: $v \in$ *fmdom'* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) $v \in$ *fmdom'* (*fst a*)
fmlookup (*fst a*) $v =$ *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) *v*
by *blast+*
then have $v \in$ *fmdom' s*
by (*simp add*: *fmdom'-restrict-set-precise*)
moreover have *fmlookup* (*fst a*) $v =$ *fmlookup s v*
using *i*(*2*, *3*) *fmdom'-notI*
by *force*
ultimately have $v \in$ *?B*
using *i*
by *blast*
}
then have *1*: *?A* $\subseteq$ *?B*
by *blast*
{
fix *v*
assume *P*: $v \in$ *?B*
then have *ii*: $v \in$ *fmdom' s* $v \in$ *fmdom'* (*fst a*) *fmlookup* (*fst a*) $v =$ *fmlookup s v*
by *blast+*
moreover {
have $\exists s'$. $v \in s' \wedge (\exists a$. ($s' =$ *action-needed-vars a s*) $\wedge a \in$ *PROB*)
unfolding *action-needed-vars-def*
using *assms P action-needed-vars-def*
by *metis*
then obtain *s'* **where** α: $v \in s'$ ($\exists a$. ($s' =$ *action-needed-vars a s*) $\wedge a \in$ *PROB*)
by *blast*
moreover obtain *a'* **where** $s' =$ *action-needed-vars a' s a'* $\in$ *PROB*
using α
by *blast*
ultimately have $v \in$ *fmdom'* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*)
unfolding *fmdom'-restrict-set-precise*
using *action-needed-vars-subset-sys-needed-vars-subset ii*(*1*) **by** *blast*
}
note *iii* = *this*
moreover have *fmlookup* (*fst a*) $v =$ *fmlookup* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) *v*
using *ii*(*3*) *iii fmdom'-notI*

by *force*
ultimately have $v \in$ *?A*
by *blast*
}
then have *?B* $\subseteq$ *?A*
by *blast*
then show *?thesis*
unfolding *action-needed-vars-def*
using *1*
by *blast*
qed

— NOTE added lemma.
— TODO refactor (proven elsewhere?).
lemma *system-needed-asses-include-action-needed-asses-i*:
fixes *A B f*
assumes $A \subseteq B$
shows *fmrestrict-set A* (*fmrestrict-set B f*) = *fmrestrict-set A f*
proof −
{
let *?f′=fmrestrict-set A f*
let *?f′′=fmrestrict-set A* (*fmrestrict-set B f*)
assume *C*: *?f′′* $\neq$ *?f′*
then obtain *v* **where** *1*: *fmlookup ?f′′ v* $\neq$ *fmlookup ?f′ v*
by (*meson fmap-ext*)
then have *False*
proof (*cases* $v \in A$)
case *True*
have *fmlookup ?f′′ v* = *fmlookup* (*fmrestrict-set B f*) *v*
using *True fmlookup-restrict-set*
by *simp*
moreover have *fmlookup* (*fmrestrict-set B f*) *v* = *fmlookup ?f′ v*
using *True assms*(*1*)
by *auto*
ultimately show *?thesis*
using *1*
by *argo*
next
case *False*
then have *fmlookup ?f′ v* = *None fmlookup ?f′′ v* = *None*
using *fmlookup-restrict-set*
by *auto+*
then show *?thesis*
using *1*
by *argo*
qed
}
then show *?thesis*
by *blast*

qed

lemma *system-needed-asses-include-action-needed-asses*:
assumes ($a \in PROB$)
shows (*action-needed-asses a* (*system-needed-asses PROB s*) = *action-needed-asses a s*)
proof –
{
have *action-needed-vars a s* ⊆ *system-needed-vars PROB s*
using *action-needed-vars-subset-sys-needed-vars-subset*[*OF assms*]
by *simp*
then have
fmrestrict-set (*action-needed-vars a s*) (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) =
fmrestrict-set (*action-needed-vars a s*) *s*
using *system-needed-asses-include-action-needed-asses-i*
by *fast*
}
moreover
{
have
action-needed-vars a (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) = *action-needed-vars a s*
using *system-needed-asses-include-action-needed-asses-1*[*OF assms*]
by *simp*
then have *fmrestrict-set* (*action-needed-vars a* (*fmrestrict-set* (*system-needed-vars PROB s*) *s*))
(*fmrestrict-set* (*system-needed-vars PROB s*) *s*) =
fmrestrict-set (*action-needed-vars a s*) *s*
⟷ *fmrestrict-set* (*action-needed-vars a s*) (*fmrestrict-set* (*system-needed-vars PROB s*) *s*) =
fmrestrict-set (*action-needed-vars a s*) *s*
by *simp*
}
ultimately show *?thesis*
unfolding *action-needed-asses-def system-needed-asses-def*
by *simp*
qed

lemma *system-needed-asses-submap*:
system-needed-asses PROB s $\subseteq_f$ *s*
proof –
{
fix *x*
assume *P*: *x*∈ *dom* (*fmlookup* (*system-needed-asses PROB s*))
then have *system-needed-asses PROB s* = (*fmrestrict-set* (*system-needed-vars PROB s*) *s*)

```
      by (simp add: system-needed-asses-def)
    then have fmlookup (system-needed-asses PROB s) x = fmlookup s x
      using P
      by (auto simp: fmdom'-restrict-set-precise)
  }
  then have fmlookup (system-needed-asses PROB s) ⊆m fmlookup s
    using map-le-def
    by blast
  then show ?thesis
    using fmsubset.rep-eq
    by fast
qed


lemma as-works-from-system-needed-asses:
  assumes (as ∈ valid-plans PROB)
  shows (exec-plan (system-needed-asses PROB s) as ⊆f exec-plan s as)
  using assms
  by (metis
      action-needed-asses-def
      as-needed-asses-submap-exec
      fmsubset-restrict-set-mono system-needed-asses-def
      system-needed-asses-include-action-needed-asses
      system-needed-asses-include-action-needed-asses-1
      system-needed-asses-submap
      valid-plan-mems
      )


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*)
by *simp*

— NOTE 'fun' because of multiple defining equations.
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* (λ*a*. ¬(*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*)))

using *assms*
unfolding *exec-plan.simps*
proof(*induction as arbitrary*: *s s' vs P*)
case (*Cons a as*)
then have *1*: *fst a* $\subseteq_f$ *s sat-precond-as* (*state-succ s a*) *as*
by *auto*
then have *2*: ∀ *a'*. *ListMem a' as* ∧ *P a'* ⟶ *fmdom'* (*snd a'*) ∩ *vs* = {}
by (*simp add*: *Cons.prems*(*1*) *insert*)
then show *?case*
proof (*cases P a*)
case *True*
{
then have *filter* (λ*a*. ¬(*P a*)) (*a* # *as*) = *filter* (λ*a*. ¬(*P a*)) *as*
by *simp*

```
      then have sat-precond-as s′ (filter (λa. ¬(P a)) as)
        using Cons.prems(3) True
        by argo
    }
    note a = this
    {
      then have ListMem a (a # as)
        using elem
        by fast
      then have (fmdom′ (snd a) ∩ vs) = {}
        using Cons.prems(1) True
        by blast
      then have fmrestrict-set vs (state-succ s a) = fmrestrict-set vs s
        using disj-imp-eq-proj-exec[symmetric]
        by fast
    }
    then show ?thesis
      unfolding exec-plan.simps
       using Cons.prems(4) 1(2) 2 True a Cons.IH[where s=state-succ s a and
s′=s′]
      by fastforce
  next
    case False
    {
      have filter (λa. ¬(P a)) (a # as) = a # filter (λa. ¬(P a)) as
        using False
        by auto
      then have fst a ⊆f s′ sat-precond-as (state-succ s′ a) (filter (λa. ¬(P a)) as)
        using Cons.prems(3) False
        by force+
    }
    note b = this
    then have fmrestrict-set vs (state-succ s a) = fmrestrict-set vs (state-succ s′ a)
      using proj-eq-proj-exec-eq
      using Cons.prems(4) 1(1)
      by blast
    then show ?thesis
      unfolding exec-plan.simps
      using 1(2) 2 False b Cons.IH[where s=state-succ s a and s′=state-succ s′
a]
      by force
  qed
qed simp
```

— NOTE curried instead of triples.
— NOTE 'fun' because of multiple defining equations.

```
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*)
by *simp+*

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*)

by (*induction as arbitrary*: *s h pfx*) *auto*

lemma *rem-condless-valid-1*:
fixes *as s*
shows (*exec-plan s as* = *exec-plan s* (*rem-condless-act s* [] *as*))
by (*induction as arbitrary*: *s*)
(*auto simp add*: *exec-remcondless-cons FDOM-state-succ state-succ-def*)

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*)
by (*induction as arbitrary*: *h′ pfx s*) *auto*

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*))
∧ *h′* = *h*
)
using *assms*
proof (*induction as arbitrary*: *h h′ as′ s*)
case *Nil*

```
  then have rem-condless-act s [h] [] = [h]
    by simp
  then have 1: as′ = []
    using Nil.prems
    by simp
  then have rem-condless-act (state-succ s h) [] [] = []
    by simp
  then have 2: prefix as′ (rem-condless-act (state-succ s h) [] [])
    using 1
    by simp
  then have h = h′
    using Nil.prems
    by force
  then show ?case
    using 2
    by blast
next
  case (Cons a as)
  {
    have rem-condless-act s [h] (a # as) = h # rem-condless-act (state-succ s h)
[] (a # as)
      using rem-condless-act-cons
      by fast
    then have h = h′
      using Cons.prems
      by simp
  }
  moreover {
    obtain l where (h′ # as′) @ l = (h # rem-condless-act (state-succ s h) [] (a
# as))
      using Cons.prems rem-condless-act-cons prefixE
      by metis
    then have prefix (as′ @ l) (rem-condless-act (state-succ s h) [] (a # as))
      by simp
    then have prefix as′ (rem-condless-act (state-succ s h) [] (a # as))
      using append-prefixD
      by blast
  }
  ultimately show ?case
    by fastforce
qed


lemma rem-condless-valid-2:
  fixes as s
  shows sat-precond-as s (rem-condless-act s [] as)
  by (induction as arbitrary: s) (auto simp: rem-condless-act-cons)
```

lemma *rem-condless-valid-3*:
  fixes *as s*
  shows *length* (*rem-condless-act s* [] *as*) ≤ *length as*
  by (*induction as arbitrary*: *s*)
    (*auto simp*: *rem-condless-act-cons le-SucI*)

lemma *rem-condless-valid-4*:
  fixes *as A s*
  assumes (*set as* ⊆ *A*)
  shows (*set* (*rem-condless-act s* [] *as*) ⊆ *A*)
  using *assms*
  by (*induction as arbitrary*: *A s*) (*auto simp*: *rem-condless-act-cons*)

lemma *rem-condless-valid-6*:
  fixes *as s P*
  shows *length* (*filter P* (*rem-condless-act s* [] *as*)) ≤ *length* (*filter P as*)
proof (*induction as arbitrary*: *P s*)
  case (*Cons a as*)
  then show *?case*
    by (*simp add*: *rem-condless-act-cons le-SucI*)
qed *simp*

lemma *rem-condless-valid-7*:
  fixes *s P as as2*
  assumes (*list-all P as* ∧ *list-all P as2*)
  shows *list-all P* (*rem-condless-act s as2 as*)
  using *assms*
  by (*induction as arbitrary*: *P s as2*) *auto*

lemma *rem-condless-valid-8*:
  fixes *s as*
  shows *subseq* (*rem-condless-act s* [] *as*) *as*
  by (*induction as arbitrary*: *s*) (*auto simp*: *sublist-cons-4 rem-condless-act-cons*)

lemma *rem-condless-valid-10*:
  fixes *PROB as*
  assumes *as* ∈ (*valid-plans PROB*)
  shows (*rem-condless-act s* [] *as* ∈ *valid-plans PROB*)
  using *assms valid-plans-def rem-condless-valid-1 rem-condless-valid-4*
  by *blast*

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*)))
using *rem-condless-valid-1 rem-condless-valid-2 rem-condless-valid-3 rem-condless-valid-6 rem-condless-valid-4*
by *fast*

— NOTE type of 'as' had to be fixed for lemma submap_imp_state_succ_submap.
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*)
using *assms*
proof (*induction as arbitrary*: *s1 s2*)
case (*Cons a as*)
{
have *fst a* $\subseteq_f$ *s1*
using *Cons.prems*(*2*)
by *simp*
then have *fst a* $\subseteq_f$ *s2*
using *Cons.prems*(*1*) *submap-imp-state-succ-submap-a*
by *blast*
}
note *1* = *this*
{
have *2*: *fst a* $\subseteq_f$ *s1 sat-precond-as* (*state-succ s1 a*) *as*
using *Cons.prems*(*2*)
by *simp+*
then have *state-succ s1 a* $\subseteq_f$ *state-succ s2 a*
using *Cons.prems*(*1*) *submap-imp-state-succ-submap*
by *blast*
then have *3*: *sat-precond-as* (*state-succ s2 a*) *as*
using *2*(*2*) *Cons.IH*
by *blast*
}
then show *?case*
using *1*
by *auto*
qed *auto*

— NOTE added lemma.
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*

```
  using assms
proof (cases fst a ⊆f s1)
  case true: True
  then show ?thesis
  proof (cases fst a ⊆f s2)
    case True
    then show ?thesis
      unfolding state-succ-def
      using assms submap-imp-state-succ-submap-b state-succ-def true
      by auto
  next
    case False
    then show ?thesis
      using assms submap-imp-state-succ-submap-a true
      by blast
  qed
next
  case false: False
  then show ?thesis
  proof (cases fst a ⊆f s2)
    case True
    then show ?thesis
      using assms false
      by auto
  next
    case False
    then show ?thesis
      unfolding state-succ-def
      using false assms
      by simp
  qed
qed

lemma submap-init-submap-exec:
  fixes s1 s2
  assumes (s1 ⊆f s2) (sat-precond-as s1 as)
  shows (exec-plan s1 as ⊆f exec-plan s2 as)
  using assms
proof (induction as arbitrary: s1 s2)
  case (Cons a as)
  have state-succ s1 a ⊆f state-succ s2 a
    using Cons.prems submap-init-submap-exec-i
    by blast
  moreover have sat-precond-as (state-succ s1 a) as
    using Cons.prems(2)
    by simp
  ultimately have exec-plan (state-succ s1 a) as ⊆f exec-plan (state-succ s2 a)
as
    using Cons.IH
```

by *blast*
then show *?case*
by *simp*
qed *simp*

— NOTE type of ‘as‘ had to be fixed for ‘submap_sat_precond_submap‘.
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 –
have *fmrestrict-set vs s* $\subseteq_f$ *s*
by *simp*
then show (*sat-precond-as s as*)
using *assms submap-sat-precond-submap*
by *blast*
qed

— NOTE name shortened to ’varset_action’.
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*)
unfolding *varset-action-def*
by *auto*

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*)
unfolding *varset-action-def*
using *assms*
by *presburger*

lemma *rem-effectless-works-1*:
fixes *s as*
shows (*exec-plan s as* = *exec-plan s* (*rem-effectless-act as*))
by (*induction as arbitrary*: *s*) (*auto simp*: *empty-eff-exec-eq*)

lemma *rem-effectless-works-2*:
fixes *as s*
assumes (*sat-precond-as s as*)

shows (*sat-precond-as s* (*rem-effectless-act as*))
using *assms*
by (*induction as arbitrary*: *s*) (*auto simp*: *empty-eff-exec-eq*)

lemma *rem-effectless-works-3*:
fixes *as*
shows *length* (*rem-effectless-act as*) $\leq$ *length as*
by (*induction as*) *auto*

lemma *rem-effectless-works-4*:
fixes *A as*
assumes (*set as* $\subseteq$ *A*)
shows (*set* (*rem-effectless-act as*) $\subseteq$ *A*)
using *assms*
by (*induction as arbitrary*: *A*) *auto*

lemma *rem-effectless-works-4'*:
fixes *A as*
assumes (*as* $\in$ *valid-plans A*)
shows (*rem-effectless-act as* $\in$ *valid-plans A*)
using *assms*
by (*induction as arbitrary*: *A*) (*auto simp*: *valid-plans-def*)

— NOTE added lemma.
lemma *rem-effectless-works-5-i*:
shows *subseq* (*rem-effectless-act as*) *as*
by (*induction as*) *auto*

lemma *rem-effectless-works-5*:
fixes *P as*
shows *length* (*filter P* (*rem-effectless-act as*)) $\leq$ *length* (*filter P as*)
using *rem-effectless-works-5-i sublist-imp-len-filter-le*
by *blast*

lemma *rem-effectless-works-6*:
fixes *as*
shows *no-effectless-act* (*rem-effectless-act as*)
by (*induction as*) *auto*

lemma *rem-effectless-works-7*:
fixes *as*
shows *no-effectless-act as* = *list-all* (λa. *fmdom'* (*snd a*) $\neq$ {}) *as*
by (*induction as*) *auto*

lemma *rem-effectless-works-8*:
fixes *P as*
assumes (*list-all P as*)
shows *list-all P* (*rem-effectless-act as*)
using *assms*
by (*induction as arbitrary*: *P*) *auto*

— TODO move and replace 'rem_effectless_works_5_i'.
lemma *rem-effectless-works-9*:
fixes *as*
shows *subseq* (*rem-effectless-act as*) *as*
by (*induction as*) *auto*

lemma *rem-effectless-works-10*:
fixes *as P*
assumes (*no-effectless-act as*)
shows (*no-effectless-act* (*filter P as*))
using *assms*
by (*auto simp*: *rem-effectless-works-7*) (*metis Ball-set filter-set member-filter*)

lemma *rem-effectless-works-11*:
fixes *as1 as2*
assumes *subseq as1* (*rem-effectless-act as2*)
shows (*subseq as1 as2*)
using *assms rem-effectless-works-9 sublist-trans*
by *blast*

lemma *rem-effectless-works-12*:
fixes *as1 as2*
shows (*no-effectless-act* (*as1* @ *as2*)) = (*no-effectless-act as1* ∧ *no-effectless-act*(*as2*))
by (*induction as1*) *auto*

— TODO refactor into 'List_Utils.thy'.
lemma *rem-effectless-works-13-i*:
fixes *x l*
assumes *ListMem x l list-all P l*
shows *P x*
using *assms* **proof** (*induction l*)
case (*insert x xs y*)
have *1*: *P y*
using *insert.prems list.pred-inject*
by *simp*

```
  then have 2: list-all P l
    using assms(2) list.pred-inject
    by force
  then show ?case
    using 1
  proof (cases y = x)
    case False
    then show ?thesis
      using insert 2
      by fastforce
  qed simp
qed simp

lemma rem-effectless-works-13:
  fixes as1 as2
  assumes (subseq as1 as2) (no-effectless-act as2)
  shows (no-effectless-act as1)
  using assms
proof (induction as1 arbitrary: as2)
  case (Cons a as1)
  {
    have subseq as1 as2
      using Cons.prems(1) sublist-CONS1-E
      by metis
    then have no-effectless-act as1
      using Cons.prems(2) Cons.IH
      by blast
  }
  moreover
  {
    have list-all (λa. fmdom' (snd a) ≠ {}) as2
      using Cons.prems(2) rem-effectless-works-7
      by blast
    moreover have ListMem a as2
      using Cons.prems(1) sublist-MEM
      by fast
    ultimately have fmdom' (snd a) ≠ {}
      using rem-effectless-works-13-i
      by fastforce
  }
  ultimately show ?case
    by simp
qed simp


lemma rem-effectless-works-14:
  fixes PROB as
  shows exec-plan s as = exec-plan s (rem-effectless-act as)
  using rem-effectless-works-1
```

by *blast*

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*))
using *assms rem-effectless-works-5*
by *blast*

— NOTE name shortened.
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*)
unfolding *rem-effectless-act-set-def*
using *assms*
by (*induction as*) *auto*

lemma *rem-effectless-act-set-no-empty-actions-thm*:
fixes *A*
shows *rem-effectless-act-set A* ⊆ {*a*. *fmdom′* (*snd a*) ≠ {}}
unfolding *rem-effectless-act-set-def*
by *blast*

— NOTE proof required additional lemmas 'rem_effectless_works_7' and 'rem_condless_valid_7'.
lemma *rem-condless-valid-9*:
fixes *s as*
assumes *no-effectless-act as*
shows *no-effectless-act* (*rem-condless-act s* [] *as*)
using *assms*
proof (*induction as arbitrary*: *s*)
case (*Cons a as*)
then show *?case*
using *Cons*
proof (*cases* *fst a* $\subseteq_f$ *exec-plan s* [])
case *True*

then have *rem-condless-act s* [] (*a* # *as*) = *a* # *rem-condless-act* (*state-succ s a*) [] *as*
using *rem-condless-act-cons*
by *fastforce*
moreover
{
have *fmdom′* (*snd a*) ≠ {} *no-effectless-act as*
using *Cons.prems*
by *simp+*
then have *no-effectless-act* (*rem-condless-act* (*state-succ s a*) [] *as*)
using *Cons.IH*
by *blast*
}
moreover have *no-effectless-act* [*a*]
using *Cons.prems*
by *simp*
ultimately show *?thesis*
using *rem-effectless-works-12*
by *force*
qed *simp*
qed *simp*

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*)
using *assms*
proof (*induction as arbitrary*: *as-1 as-2 s*)
case (*Cons a as*)
then show *?case* **proof**(*cases as-1*)
case *Nil*
then show *?thesis*
using *Cons.prems*(*1*, *2*)
by *auto*
next
case (*Cons a list*)
then show *?thesis*
using *Cons.prems*(*1*, *2*) *Cons.IH hd-append2 list.distinct*(*1*) *list.sel*(*1*, *3*) *tl-append2*
by *auto*
qed
qed *auto*

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*)

```
  using assms
proof (induction as arbitrary: f)
  case (Cons a as)
  then show ?case using fmdom'-empty snd-conv
    by (metis (mono-tags, lifting) Ball-set)
qed simp
```

```
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'
  using assms
  by (induction as arbitrary: as' s) auto
```

```
lemma not-vset-not-disj-eff-prod-dom-diff:
  fixes PROB a vs
  assumes (a ∈ PROB) (¬varset-action a vs)
  shows ¬((fmdom' (snd a) ∩ ((prob-dom PROB) − vs)) = {})
proof −
  have 1: fmdom' (snd a) ≠ {}
    using assms(2) varset-action-def
    by blast
  {
    have fmdom' (snd a) ⊆ prob-dom PROB
      using assms(1) FDOM-eff-subset-prob-dom-pair
      by metis
    then have
      fmdom' (snd a) ∩ (prob-dom PROB − vs)
      = (fmdom' (snd a)) − (fmdom' (snd a) ∩ vs)
      using Diff-Int-distrib
      by blast
  }
  note 2 = this
  then show ?thesis
    using 1 2
  proof (cases fmdom' (snd a) ∩ vs = {})
    case False
    {
      have ¬(fmdom' (snd a) ⊆ vs)
        using assms(2) varset-action-def
        by fast
      then have (fmdom' (snd a) ∩ vs ≠ fmdom' (snd a))
        by auto
      then have (fmdom' (snd a) ∩ vs) ⊂ fmdom' (snd a)
        by blast
    }
    then show ?thesis using 2
```

by *auto*
qed *force*
qed

lemma *vset-disj-dom-eff-diff*:
fixes *PROB a vs*
assumes (*varset-action a vs*)
shows (((*fmdom'* (*snd a*)) ∩ (*prob-dom PROB* − *vs*)) = {})
using *assms*
unfolding *varset-action-def*
by *auto*

lemma *vset-diff-disj-eff-vs*:
fixes *PROB a vs*
assumes (*varset-action a* (*prob-dom PROB* − *vs*))
shows (((*fmdom'* (*snd a*)) ∩ *vs*) = {})
using *assms*
unfolding *varset-action-def*
by *blast*

lemma *vset-nempty-efff-not-disj-eff-vs*:
fixes *PROB a vs*
assumes (*varset-action a vs*) (*fmdom'* (*snd a*) ≠ {})
shows ¬((*fmdom'* (*snd a*) ∩ *vs*)) = {}
using *assms*
unfolding *varset-action-def*
by *auto*

lemma *vset-disj-eff-diff*:
fixes *s a vs*
assumes (*varset-action a vs*)
shows ((*fmdom'* (*snd a*) ∩ (*s* − *vs*)) = {})
proof −
have *1*: *fmdom'* (*snd a*) ⊆ *vs*
using *assms*
by (*simp add*: *varset-action-def*)
moreover {
have *fmdom'* (*snd a*) ∩ (*s* − *vs*) = (*fmdom'* (*snd a*) ∩ *s*) − (*fmdom'* (*snd a*) ∩ *vs*)
using *Diff-Int-distrib*
by *fast*
also have ... = (*fmdom'* (*snd a*) ∩ *s*) − (*fmdom'* (*snd a*))
using *1*
by *auto*
finally have *fmdom'* (*snd a*) ∩ (*s* − *vs*) = {}

```
    by simp
  }
  ultimately show ?thesis
    by blast
qed


— NOTE added lemma.
lemma list-all-list-mem:
  fixes P and l :: 'a list
  shows list-all P l ⟷ (∀ e. ListMem e l ⟶ P e)
proof –
  {
    assume P1: list-all P l
    {
      fix e
      assume P11: ListMem e l
      then have P e
        using P1 P11
      proof (induction l arbitrary: P)
        case (insert x xs y)
        then show ?case proof (cases y = x)
          case False
          then have list-all P xs ListMem x xs
            using insert.prems(1) insert.hyps
            by fastforce+
          then show ?thesis
            using insert.IH
            by blast
        qed simp
      qed simp
    }
  }
  moreover
  {
    assume P2: (∀ e. ListMem e l ⟶ P e)
    then have list-all P l
    proof(induction l arbitrary: P)
      case (Cons a l)
      {
        have ∀ e. ListMem e l ⟶ P e
          using Cons.prems insert
          by fast
        then have list-all P l
          using Cons.IH
          by blast
      }
      moreover have P a
        using Cons.prems elem
```

by *fast*
ultimately show *?case*
by *simp*
qed *simp*
}
ultimately show *?thesis*
by *blast*
qed

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 $-$
have *1*: $\forall$ *e. ListMem e as* $\longrightarrow$ *varset-action e* ((*prob-dom PROB*) $-$ *vs*)
using *assms list-all-list-mem*
by *metis*
{
fix *a*
assume *ListMem a as*
then have *varset-action a* (*prob-dom PROB* $-$ *vs*)
using *1*
by *blast*
then have *disjnt* (*fmdom′* (*snd a*)) *vs*
unfolding *disjnt-def*
using *vset-diff-disj-eff-vs*
by *blast*
}
then have *list-all* (λ*a. disjnt* (*fmdom′* (*snd a*)) *vs*) *as*
using *list-all-list-mem*
by *blast*
then have *list-all* (λ*a. disjnt* (*fmdom′* (*snd a*)) *vs*) (*rem-condless-act s* [] *as*)
by (*simp add*: *rem-condless-valid-7*)
then have *exec-plan s as* = *exec-plan s* (*rem-condless-act s* [] *as*)
using *rem-condless-valid-1*
by *blast*
then have*sat-precond-as s* (*rem-condless-act s* [] *as*)
using *rem-condless-valid-2*
by *blast*
then have *sat-precond-as s* [*a*←*as* . ¬ *varset-action a* (*prob-dom PROB* $-$ *vs*)]
by (*simp add*: *1 ListMem-iff*)
then have *fmrestrict-set vs s* = *fmrestrict-set vs s* **by** *simp*
then have
fmrestrict-set vs (*exec-plan s as*) =
fmrestrict-set vs (*exec-plan s* [*a*←*as* . ¬ *varset-action a* (*prob-dom PROB* $-$ *vs*)])

using *1 graph-plan-lemma-4* [**where**

s = *s* **and** *s′* = *s* **and** *as* = *rem-condless-act s* [] *as* **and** *vs* = *vs* **and**
P = *λa. varset-action a* (*prob-dom PROB* − *vs*)
] *filter-empty-every-not vset-diff-disj-eff-vs 1disjoint-effects-no-effects*
exec-plan.simps(*1*) *fmdom′-restrict-set-precise list-all-list-mem*
by *smt*
then have *list-all* (*λa. varset-action a* (*prob-dom PROB* − *vs*)) (*rem-condless-act s* [] *as*)
using *assms*(*1*) *rem-condless-valid-7 list.pred-inject*(*1*)
by *blast*
then have *filter* (*λa.* ¬(*varset-action a* (*prob-dom PROB* − *vs*))) (*rem-condless-act s* [] *as*) = []
using *filter-empty-every-not*
by *fastforce*
then have
sat-precond-as s (*filter* (*λa.* ¬(*varset-action a* (*prob-dom PROB* − *vs*)))
(*rem-condless-act s* []*as*))

by *fastforce*
then show *?thesis*
using *1 vset-diff-disj-eff-vs disjoint-effects-no-effects fmdom′-restrict-set-precise*
by *metis*
qed

lemma *no-effectless-act-works*:
fixes *as*
assumes (*no-effectless-act as*)
shows (*filter* (*λa.* ¬(*fmdom′* (*snd a*) = {})) *as* = *as*)
using *assms*
by (*simp add*: *Ball-set rem-effectless-works-7*)

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*))
using *assms*
unfolding *varset-action-def*
by *blast*

lemma *vset-diff-union-vset-diff*:
fixes *s vs vs′ a*
assumes (*varset-action a* (*s* − (*vs* ∪ *vs′*)))
shows (*varset-action a* (*s* − *vs′*))
using *assms*
unfolding *varset-action-def*
by *blast*

```
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)
  using assms
proof (induction as)
  case (Cons a as)
  {
    have as ∈ valid-plans PROB
      using Cons.prems valid-plan-valid-tail
      by fast
    then have (filter (λa. varset-action a (prob-dom PROB)) as = as)
      using Cons.IH
      by blast
  }
  moreover {
    have a ∈ PROB
      using Cons.prems valid-plan-valid-head
      by fast
    then have varset-action a (prob-dom PROB)
      unfolding varset-action-def
      using FDOM-eff-subset-prob-dom-pair
      by metis
  }
  ultimately show ?case
    by simp
qed fastforce


lemma n-replace-proj-le-n-as-1:
  fixes a vs vs′
  assumes (vs ⊆ vs′) (varset-action a vs)
  shows (varset-action a vs′)
  using assms
  unfolding varset-action-def
  by simp


lemma sat-precond-as-pfx:
  fixes s
  assumes (sat-precond-as s (as @ as′))
  shows (sat-precond-as s as)
  using assms
proof (induction as arbitrary: s as′)
  case (Cons a as)
  have fst a ⊆f s
    using Cons.prems
    by fastforce
```

moreover have *sat-precond-as* (*state-succ s a*) (*as* @ *as'*)
using *Cons.prems*
by *simp*
ultimately show *?case*
using *Cons.IH sat-precond-as.simps*(*2*)
by *blast*
qed *simp*

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**
$TC\ R\ a\ b \equiv (\forall P.\ (\forall x\ y.\ R\ x\ y \longrightarrow P\ x\ y) \land (\forall x\ y\ z.\ P\ x\ y \land P\ y\ z \longrightarrow P\ x\ z) \longrightarrow P\ a\ b)$

— NOTE adapts transitive closure definitions of Isabelle and HOL4.
lemma *TC-equiv-tranclp*: $TC\ R\ a\ b \longleftrightarrow (R^{++}\ a\ b)$
proof –
{
have $TC\ R\ a\ b \Longrightarrow (R^{++}\ a\ b)$
unfolding *TC-def*
using *tranclp.r-into-trancl tranclp-trans*
by *metis*
}
moreover
{
have $(R^{++}\ a\ b) \Longrightarrow TC\ R\ a\ b$ **proof**(*induction rule*: *tranclp.induct*)
case (*r-into-trancl a b*)
then show *?case* **by**(*subst TC-def*; *auto*)
next
case (*trancl-into-trancl a b c*)
then show *?case* **unfolding** *TC-def* **by** *blast*
qed
}
ultimately show *?thesis*
by *fast*
qed

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 −
from *assms*(*1*) **have** *1*: $\neg TC\ R\ x\ y$
using *TC-equiv-tranclp*
by *fast*
{
assume *P*: $\neg TC\ R\ x\ y$
then obtain *P* **where** *a*: $(\forall x\ y.\ R\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow \neg P\ x\ y$
unfolding *TC-def*
by *blast*
{
assume *P-1*: $(\forall x\ y.\ R\ x\ y \longrightarrow P\ x\ y)\ (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z)$
then have $(\forall x\ y.\ R\ x\ y \wedge P\ x\ y \longrightarrow P\ x\ y)\ (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z)$
by *blast+*
moreover from *a* **and** *P-1* **have** $\neg P\ x\ y$
by *blast*
then have $\exists P.\ (\forall x\ y.\ R\ x\ y \wedge P\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow \neg P\ x\ y$
by *blast*
}
then have $\exists P.$
$(\forall x\ y.\ R\ x\ y \wedge P\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow \neg P\ x\ y$
by *blast*
}
note *2* = *this*
{
from *1 2* **have** $\exists P.$
$(\forall x\ y.\ R\ x\ y \wedge P\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z.\ P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow \neg P\ x\ y$
by *blast*
then have $\neg TC\ (\lambda x\ y.\ R\ x\ y \wedge P\ x\ y)\ x\ y$
unfolding *TC-def*
by (*metis assms tranclp.r-into-trancl tranclp-trans*)
then have $\neg(\lambda x\ y.\ R\ x\ y \wedge P\ x\ y)^{++}\ x\ y$
using *TC-equiv-tranclp*
by *fast*
}
then show *?thesis*
by *blast*
qed

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 –
  from *assms*(*2*)
  have *1*: ¬(λ*x y*. *R x y* ∧ *P x y*)$^{++}$ *x y*
    using *TC-IMP-NOT-TC-CONJ-1*[**where** *P*=λ*x y*. *P x y*]
    by *blast*
  {
    {
      from *1* **have** ¬*TC* (λ*x y*. *R x y* ∧ *P x y*) *x y*
        using *TC-equiv-tranclp*
        by *fast*
      then have ∃ *Pa*.
      (∀ *x y*. *R x y* ∧ *P x y* ⟶ *Pa x y*) ∧ (∀ *x y z*. *Pa x y* ∧ *Pa y z* ⟶ *Pa x z*)
      ⟶ ¬*Pa x y*
        unfolding *TC-def*
        by *blast*
    }
    then obtain *Pa* **where** *a*:
       (∀ *x y*. *R x y* ∧ *P x y* ⟶ *Pa x y*) ∧ (∀ *x y z*. *Pa x y* ∧ *Pa y z* ⟶ *Pa x z*)
⟶ ¬*Pa x y*
      by *blast*
    then have ¬(∀ *Pa*. (∀ *x y*. *R′ x y* ∧ *P x y* ⟶ *Pa x y*) ∧ (∀ *x y z*. *Pa x y* ∧ *Pa y z* ⟶ *Pa x z*) ⟶ *Pa x y*)
      by (*metis assms*(*1*) *assms*(*2*) *tranclp.r-into-trancl tranclp-trans*)
    then have ¬*TC* (λ*x y*. *R′ x y* ∧ *P x y*) *x y*
      unfolding *TC-def*
      by *blast*
  }
  then show *?thesis*
    using *TC-equiv-tranclp*
    by *fast*
qed

— NOTE added lemma (relationScript.sml:314)
lemma *TC-INDUCT*:
  fixes *R* :: *′a* ⇒ *′a* ⇒ *bool* **and** *P*
  assumes (∀ *x y*. *R x y* ⟶ *P x y*) (∀ *x y z*. *P x y* ∧ *P y z* ⟶ *P x z*)
  shows ∀ *u v*. (*TC R*) *u v* ⟶ *P u v*
  using *assms*
  unfolding *TC-def*
  by *metis*

lemma *REFL-IMP-3-CONJ-1*:
  fixes *R P x y*
  assumes ((λ*x y*. *R x y* ∧ *P x y*)$^{++}$ *x y*)
  shows R^{++} *x y*
  using *assms*
proof –
  show *?thesis*
    using *assms TC-IMP-NOT-TC-CONJ-1*

by *fast*
qed

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 −
{
fix P
{
have $\forall x\ y.\ R'\ x\ y \longrightarrow (\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ y \vee (\exists z.\ \neg\ P\ z \wedge R'^{++}\ x\ z \wedge R'^{++}\ z\ y)$
proof (*auto*)
fix $x\ y$
assume P: $R'\ x\ y\ \forall z.\ R'^{++}\ x\ z \longrightarrow P\ z \vee \neg\ R'^{++}\ z\ y$
then show $(\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ y$
proof −
have a: $\bigwedge a.\ \neg\ R'\ x\ a \vee \neg\ R'\ a\ y \vee P\ a$
using $P(2)$
by *blast*
have *reflexive* R'
by (*meson assms*)
then show *?thesis*
using $a\ P(1)$
by (*simp add*: *reflexive-def tranclp.r-into-trancl*)
qed
qed
}
moreover {
have $\forall x\ y\ z.\ ((\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ y \vee (\exists z.\ \neg\ P\ z \wedge R'^{++}\ x\ z \wedge R'^{++}\ z\ y)) \wedge$
$((\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ y\ z \vee (\exists za.\ \neg\ P\ za \wedge R'^{++}\ y\ za \wedge R'^{++}\ za\ z)) \longrightarrow$
$(\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ z \vee (\exists za.\ \neg\ P\ za \wedge R'^{++}\ x\ za \wedge R'^{++}\ za\ z)$
proof (*auto*)
fix $x\ y\ z\ za$
assume P: $\forall za.\ R'^{++}\ x\ za \longrightarrow P\ za \vee \neg\ R'^{++}\ za\ z\ (\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ y$
$\neg\ P\ za\ R'^{++}\ y\ za\ R'^{++}\ za\ z$
then show $(\lambda x\ y.\ R'\ x\ y \wedge P\ x \wedge P\ y)^{++}\ x\ z$
using P
by (*meson P rtranclp-tranclp-tranclp TC-IMP-NOT-TC-CONJ-1 tranclp-into-rtranclp*)
next
fix $x\ y\ z\ za$

assume P: $\forall\, za.\ R'^{++}\ x\ za \longrightarrow P\ za \lor \neg\ R'^{++}\ za\ z\ \neg\ P\ za\ R'^{++}\ x\ za\ R'^{++}$ $za\ y$
$(\lambda x\ y.\ R'\ x\ y \land P\ x \land P\ y)^{++}\ y\ z$
then show $(\lambda x\ y.\ R'\ x\ y \land P\ x \land P\ y)^{++}\ x\ z$
by (*meson P TC-IMP-NOT-TC-CONJ-1 tranclp-trans*)
qed
}
ultimately have $\forall\, u\ v.$
TC $R'\ u\ v$
$\longrightarrow (\lambda x\ y.\ R'\ x\ y \land P\ x \land P\ y)^{++}\ u\ v \lor (\exists\, z.\ \neg\ P\ z \land R'^{++}\ u\ z \land R'^{++}\ z\ v)$
using *TC-INDUCT*[**where** $R{=}R'$ **and**
$P{=}\lambda x\ y.\ (\ ((\lambda x\ y.\ R'\ x\ y \land P\ x \land P\ y)^{++}\ x\ y) \lor (\exists\, z.\ \neg P\ z \land R'^{++}\ x\ z \land$ $R'^{++}\ z\ y))$]
by *fast*
}
then show *?thesis*
by (*simp add*: *TC-equiv-tranclp*)
qed

lemma *REFL-TC-CONJ*:
fixes $R\ R'$ :: $'a \Rightarrow\ 'a \Rightarrow$ *bool* **and** $P\ x\ y$
assumes *reflexive* R' $\forall\, x\ y.\ P\ x \land P\ y \longrightarrow (R'\ x\ y \longrightarrow R\ x\ y)\ \neg(R^{++}\ x\ y)$
shows $(\neg(R'^{++}\ x\ y) \lor (\exists\, z.\ \neg P\ z \land (R')^{++}\ x\ z \land (R')^{++}\ z\ y))$
using *assms*
proof (*cases* $\neg R'^{++}\ x\ y$)
next
case *False*
then show *?thesis* **using** *assms*
TC-IMP-NOT-TC-CONJ[**where** $P{=}\lambda x\ y.\ P\ x \land P\ y$]
REFL-IMP-3-CONJ[*of* R']
by *blast*
qed *blast*

— NOTE This is not a trivial translation: 'TC_INDUCT' in relationScript.sml:314 differs significantly from 'trancl_induct' and 'trancl_trans_induct' in Transitive_Closure:375, 391
lemma *TC-CASES1-NEQ*:
fixes $R\ x\ z$
assumes $R^{++}\ x\ z$
shows $R\ x\ z \lor (\exists\, y :: 'a.\ \neg(x = y) \land \neg(y = z) \land R\ x\ y \land R^{++}\ y\ z)$
proof −
{
fix $u\ v$
have $\forall\, x\ y.\ R\ x\ y \longrightarrow R\ x\ y \lor (\exists\, ya.\ x \neq ya \land ya \neq y \land R\ x\ ya \land R^{++}\ ya\ y)$
by *meson*
moreover have $\forall\, x\ y\ z.$
$(R\ x\ y \lor (\exists\, ya.\ x \neq ya \land ya \neq y \land R\ x\ ya \land R^{++}\ ya\ y))$
$\land\ (R\ y\ z \lor (\exists\, ya.\ y \neq ya \land ya \neq z \land R\ y\ ya \land R^{++}\ ya\ z))$
$\longrightarrow R\ x\ z \lor (\exists\, y.\ x \neq y \land y \neq z \land R\ x\ y \land R^{++}\ y\ z)$

by (*metis tranclp.r-into-trancl tranclp-trans*)
ultimately have *TC R u v* $\longrightarrow$ *R u v* $\vee$ ($\exists$ *y*. *u* $\neq$ *y* $\wedge$ *y* $\neq$ *v* $\wedge$ *R u y* $\wedge$ R^{++} *y v*)
using *TC-INDUCT*[**where** *P*=λ*x z*. *R x z* $\vee$ ($\exists$ *y* :: *'a*. $\neg$(*x* = *y*) $\wedge$ $\neg$(*y* = *z*) $\wedge$ *R x y* $\wedge$ R^{++} *y z*)]
by *blast*
}
then show *?thesis*
using *assms TC-equiv-tranclp*
by (*simp add*: *TC-equiv-tranclp*)
qed
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-var-set PROB vs1 vs2 ≡ (*disjnt vs1 vs2*) ∧
(∃ *v1 v2*. (*v1* ∈ *vs1*) ∧ (*v2* ∈ *vs2*) ∧ (*dep PROB v1 v2*)
)

lemma *dep-var-set-self-empty*:
fixes *PROB vs*
assumes *dep-var-set PROB vs vs*
shows (*vs* = {})
using *assms*
unfolding *dep-var-set-def*
proof −
obtain *v1 v2* **where**
v1 ∈ *vs v2* ∈ *vs disjnt vs vs dep PROB v1 v2*
using *assms*
unfolding *dep-var-set-def*
by *blast*
then show *?thesis*
by *force*
qed

lemma *DEP-REFL*:
fixes *PROB*
shows *reflexive* (λ*v v′*. *dep PROB v v′*)
unfolding *dep-def reflexive-def*
by *presburger*

— NOTE added lemma.
lemma *NEQ-DEP-IMP-IN-DOM-i*:
fixes *a v*
assumes *a* ∈ *PROB v* ∈ *fmdom′* (*fst a*)
shows *v* ∈ *prob-dom PROB*
proof −
have *v* ∈ *fmdom′* (*fst a*)
using *assms*(*2*)
by *simp*
moreover have *fmdom′* (*fst a*) ⊆ *prob-dom PROB*
using *assms*(*1*)
unfolding *prob-dom-def action-dom-def*
using *case-prod-beta′*
by *auto*
ultimately show *?thesis*
by *blast*
qed

— NOTE added lemma.
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 −
have $v \in fmdom'$ (*snd a*)
using *assms*(*2*)
by *simp*
moreover have *fmdom'* (*snd a*) ⊆ *prob-dom PROB*
using *assms*(*1*)
unfolding *prob-dom-def action-dom-def*
using *case-prod-beta'*
by *auto*
ultimately show *?thesis*
by *blast*
qed

lemma *NEQ-DEP-IMP-IN-DOM*:
fixes *PROB* :: (('*a*, '*b*) *fmap* × ('*a*, '*b*) *fmap*) *set* **and** *v v'*
assumes ¬($v = v'$) (*dep PROB v v'*)
shows ($v \in$ (*prob-dom PROB*) ∧ $v' \in$ (*prob-dom PROB*))
using *assms*
unfolding *dep-def*
using *FDOM-pre-subset-prob-dom-pair FDOM-eff-subset-prob-dom-pair*
proof −
obtain *a* **where** *1*:
$a \in PROB$
($v \in fmdom'$ (*fst a*) ∧ $v' \in fmdom'$ (*snd a*) ∨ $v \in fmdom'$ (*snd a*) ∧ $v' \in fmdom'$ (*snd a*))
using *assms*
unfolding *dep-def*
by *blast*
then consider
(*i*) $v \in fmdom'$ (*fst a*) ∧ $v' \in fmdom'$ (*snd a*)
| (*ii*) $v \in fmdom'$ (*snd a*) ∧ $v' \in fmdom'$ (*snd a*)
by *blast*
then show *?thesis*
proof (*cases*)
case *i*
then have $v \in fmdom'$ (*fst a*) $v' \in fmdom'$ (*snd a*)
by *simp+*
then have $v \in$ *prob-dom PROB* $v' \in$ *prob-dom PROB*
using *1 NEQ-DEP-IMP-IN-DOM-i NEQ-DEP-IMP-IN-DOM-ii*
by *metis+*
then show *?thesis*
by *simp*
next
case *ii*
then have $v \in fmdom'$ (*snd a*) $v' \in fmdom'$ (*snd a*)
by *simp+*

then have $v \in$ *prob-dom PROB* $v' \in$ *prob-dom PROB*
using *1 NEQ-DEP-IMP-IN-DOM-ii*
by *metis+*
then show *?thesis*
by *simp*
qed
qed

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 −
obtain *v1 v2* **where** *obtain-v1-v2*: *v1* $\in \bigcup$ *S v2* $\in$ *vs disjnt* ($\bigcup$ *S*) *vs dep PROB v1 v2*
using *assms dep-var-set-def*[*of PROB* $\bigcup$ *S vs*]
by *blast*
moreover
{
fix *vs'*
assume *vs'* $\in$ *S*
moreover have *vs'* $\subseteq$ ($\bigcup$ *S*)
using *calculation Union-upper*
by *blast*
ultimately have *disjnt vs' vs*
using *obtain-v1-v2*(*3*) *disjnt-subset1*
by *blast*
}
ultimately show *?thesis*
unfolding *dep-var-set-def*
by *blast*
qed

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 −
obtain *v1 v2* **where**
obtain-v1-v2: *v1* $\in$ *vs v2* $\in$ *vs'* $\cup$ *vs'' disjnt vs* (*vs'* $\cup$ *vs''*) *dep PROB v1 v2*
using *assms dep-var-set-def*[*of PROB vs* (*vs'* $\cup$ *vs''*)]
by *blast*
— NOTE The proofs for the cases introduced here yield the goal's left and right side respectively.
consider (*i*) *v2* $\in$ *vs'* | (*ii*) *v2* $\in$ *vs''*
using *obtain-v1-v2*(*2*)
by *blast*

then show *?thesis*
proof (*cases*)
case *i*
have $vs' \subseteq vs' \cup vs''$
by *auto*
moreover have *disjnt* $(vs' \cup vs'')$ *vs*
using *obtain-v1-v2*(*3*) *disjnt-sym*
by *blast*
ultimately have *disjnt vs vs'*
using *disjnt-subset1 disjnt-sym*
by *blast*
then have *dep-var-set PROB vs vs'*
unfolding *dep-var-set-def*
using *obtain-v1-v2*(*1*, *4*) *i*
by *blast*
then show *?thesis*
by *simp*
next
case *ii*
then have $vs'' \subseteq vs' \cup vs''$
by *simp*
moreover have *disjnt* $(vs' \cup vs'')$ *vs*
using *obtain-v1-v2*(*3*) *disjnt-sym*
by *fast*
ultimately have *disjnt vs vs''*
using *disjnt-subset1 disjnt-sym*
by *metis*
then have *dep-var-set PROB vs vs''*
unfolding *dep-var-set-def*
using *obtain-v1-v2*(*1*, *4*) *ii*
by *blast*
then show *?thesis*
by *simp*
qed
qed

— NOTE This is symmetrical to ‘dep_sos_imp_mem_dep‘ w.r.t to ‘vs‘ and $\bigcup S$.
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 −
obtain *v1 v2* **where** *obtain-v1-v2*: $v1 \in vs$ $v2 \in (\bigcup S)$ *disjnt vs* $(\bigcup S)$ *dep PROB v1 v2*
using *assms dep-var-set-def*[*of PROB vs* $\bigcup$ *S*]
by *blast*
moreover
{

fix *vs′*
assume *vs′* $\in$ *S*
then have *vs′* $\subseteq$ ($\bigcup$ *S*)
using *calculation Union-upper*
by *blast*
moreover have *disjnt* ($\bigcup$ *S*) *vs*
using *obtain-v1-v2*(*3*) *disjnt-sym*
by *blast*
ultimately have *disjnt vs vs′*
using *obtain-v1-v2*(*3*) *disjnt-subset1 disjnt-sym*
by *metis*
}
ultimately show *?thesis*
unfolding *dep-var-set-def*
by *blast*
qed

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* $\times$ (*′a*, *′b*) *fmap*) *set* **and** *v1 v2*
assumes $\neg$(*v1* = *v2*) (*dep-tc PROB v1 v2*)
shows (*v1* $\in$ *prob-dom PROB*)
proof −
have *TC* (*dep PROB*) *v1 v2*
using *assms*(*2*)
unfolding *dep-tc-def*
by *simp*
then have *dep PROB v1 v2* $\vee$ ($\exists$ *y*. *v1* $\neq$ *y* $\wedge$ *y* $\neq$ *v2* $\wedge$ *dep PROB v1 y* $\wedge$ *TC* (*dep PROB*) *y v2*)
using *TC-CASES1-NEQ*[**where** *R* = (λ*v1′ v2′*. *dep PROB v1′ v2′*) **and** *x* = *v1* **and** *z* = *v2*]
by (*simp add*: *TC-equiv-tranclp*)
— NOTE Split on the disjunction yielded by the previous step.
then consider
(*i*) *dep PROB v1 v2*
| (*ii*) ($\exists$ *y*. *v1* $\neq$ *y* $\wedge$ *y* $\neq$ *v2* $\wedge$ *dep PROB v1 y* $\wedge$ *TC* (*dep PROB*) *y v2*)
by *fast*
then show *?thesis*
proof (*cases*)
case *i*
{
consider

```
      (II) (∃ a.
          a ∈ PROB ∧
          (
            v1 ∈ fmdom' (fst a) ∧ v2 ∈ fmdom' (snd a)
            ∨ v1 ∈ fmdom' (snd a) ∧ v2 ∈ fmdom' (snd a)))
      | (III) v1 = v2
      using i
      unfolding dep-def
      by blast
    then have ?thesis
    proof (cases)
      case II
      then obtain a where 1:
        a ∈ PROB (v1 ∈ fmdom' (fst a) ∧ v2 ∈ fmdom' (snd a)
            ∨ v1 ∈ fmdom' (snd a) ∧ v2 ∈ fmdom' (snd a))
        by blast
      then have v1 ∈ fmdom' (fst a) ∪ fmdom' (snd a)
        by blast
      then have 2: v1 ∈ action-dom (fst a) (snd a)
        unfolding action-dom-def
        by blast
      then have action-dom (fst a) (snd a) ⊆ prob-dom PROB
        using 1(1) exec-as-proj-valid-2
        by fast
      then have v1 ∈ prob-dom PROB
        using 1 2
        by fast
      then show ?thesis
        by simp
    next
      case III
      then show ?thesis
        using assms(1)
        by simp
    qed
  }
  then show ?thesis
    by simp
 next
  case ii
  then obtain y where v1 ≠ y y ≠ v2 dep PROB v1 y TC (dep PROB) y v2
    using ii
    by blast
  then show ?thesis
    using NEQ-DEP-IMP-IN-DOM
    by metis
 qed
qed
```

lemma *not-dep-disj-imp-not-dep*:
  fixes *PROB vs-1 vs-2 vs-3*
  assumes $((vs\text{-}1 \cap vs\text{-}2) = \{\})$ $(vs\text{-}3 \subseteq vs\text{-}2)$ $\neg(dep\text{-}var\text{-}set\ PROB\ vs\text{-}1\ vs\text{-}2)$
  shows $\neg(dep\text{-}var\text{-}set\ PROB\ vs\text{-}1\ vs\text{-}3)$
  using *assms subset-eq*
  unfolding *dep-var-set-def disjnt-def*
  by *blast*

lemma *dep-slist-imp-mem-dep*:
  fixes *PROB vs lvs*
  assumes $(dep\text{-}var\text{-}set\ PROB\ (\bigcup\ (set\ lvs))\ vs)$
  shows $(\exists\, vs'.\ ListMem\ vs'\ lvs \wedge dep\text{-}var\text{-}set\ PROB\ vs'\ vs)$
proof –
  obtain *v1 v2* **where**
    obtain-v1-v2: $v1 \in \bigcup(set\ lvs)$ $v2 \in vs$ $disjnt\ (\bigcup(set\ lvs))\ vs$ $dep\ PROB\ v1\ v2$
    using *assms dep-var-set-def*[*of PROB* $\bigcup$ (*set lvs*) *vs*]
    by *blast*
  then obtain *vs'* **where** *obtain-vs'*: $vs' \in set\ lvs$ $v1 \in vs'$
    by *blast*
  then have *ListMem vs' lvs*
    using *ListMem-iff*
    by *fast*
  moreover {
    have *disjnt vs' vs*
      using *obtain-v1-v2*(*3*) *obtain-vs'*(*1*) **by** *auto*
    then have *dep-var-set PROB vs' vs*
      unfolding *dep-var-set-def*
      using *obtain-v1-v2*(*1*, *2*, *4*) *obtain-vs'*(*2*)
      by *blast*
  }
  ultimately show *?thesis*
    by *blast*
qed

lemma *n-bigunion-le-sum-3*:
  fixes *PROB vs svs*
  assumes $(\forall\ vs'.\ vs' \in svs \longrightarrow \neg(dep\text{-}var\text{-}set\ PROB\ vs'\ vs))$
  shows $\neg(dep\text{-}var\text{-}set\ PROB\ (\bigcup svs)\ vs)$
proof –
  {
    assume $(dep\text{-}var\text{-}set\ PROB\ (\bigcup svs)\ vs)$
    then obtain *v1 v2* **where** *obtain-vs*: $v1 \in \bigcup svs$ $v2 \in vs$ $disjnt\ (\bigcup svs)\ vs$ *dep*
PROB v1 v2
      unfolding *dep-var-set-def*
      by *blast*
    then obtain *vs'* **where** *obtain-vs'*: $v1 \in vs'$ $vs' \in svs$

by *blast*
then have *a*: *disjnt vs′ vs*
using *obtain-vs*(*3*) *obtain-vs′*(*2*) *disjnt-subset1*
by *blast*
then have ∀ *v1 v2*. ¬(*v1* ∈ *vs′*) ∨ ¬(*v2* ∈ *vs*) ∨ ¬*disjnt vs′ vs* ∨ ¬*dep PROB v1 v2*
using *assms obtain-vs′*(*2*) *dep-var-set-def*
by *fast*
then have *False*
using *a obtain-vs′*(*1*) *obtain-vs*(*2*, *4*)
by *blast*
}
then show *?thesis*
by *blast*
qed

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′*)
using *assms*
unfolding *varset-action-def dep-var-set-def dep-def*
proof −
assume *a1*: *fmdom′* (*snd a*) ⊆ *vs* ∪ *vs′*
assume *disjnt vs vs′*
assume ¬ (*disjnt vs′ vs* ∧
(∃ *v1 v2*. *v1* ∈ *vs′* ∧ *v2* ∈ *vs* ∧ ((∃ *a*. *a* ∈ *PROB* ∧ (*v1* ∈ *fmdom′* (*fst a*) ∧ *v2* ∈ *fmdom′* (*snd a*) ∨ *v1* ∈ *fmdom′* (*snd a*) ∧ *v2* ∈ *fmdom′* (*snd a*))) ∨ *v1* = *v2*))) ∨
¬ (*disjnt vs vs′* ∧
(∃ *v1 v2*. *v1* ∈ *vs* ∧ *v2* ∈ *vs′* ∧ ((∃ *a*. *a* ∈ *PROB* ∧ (*v1* ∈ *fmdom′* (*fst a*) ∧ *v2* ∈ *fmdom′* (*snd a*) ∨ *v1* ∈ *fmdom′* (*snd a*) ∧ *v2* ∈ *fmdom′* (*snd a*))) ∨ *v1* = *v2*)))
then have *f2*: ⋀*aa ab*. *aa* ∉ *vs* ∨ *ab* ∉ *vs′* ∨ *aa* ∉ *fmdom′* (*snd a*) ∨ *ab* ∉ *fmdom′* (*snd a*)
using ‹*a* ∈ *PROB*› ‹*disjnt vs vs′*› *disjnt-sym* **by** *blast*
obtain *aa* :: ′*a set* ⇒ ′*a set* ⇒ ′*a* **where**
f3: ⋀*A Aa a Ab Ac*. (*A* ⊆ *Aa* ∨ *aa A Aa* ∈ *A*) ∧ (*aa A Aa* ∉ *Aa* ∨ *A* ⊆ *Aa*)
∧ ((*a*::′*a*) ∉ *Ab* ∨ ¬ *Ab* ⊆ *Ac* ∨ *a* ∈ *Ac*)
by (*atomize-elim*, (*subst choice-iff*[*symmetric*])+, *blast*)
then have ⋀*A*. *fmdom′* (*snd a*) ⊆ *A* ∨ *aa* (*fmdom′* (*snd a*)) *A* ∈ *vs* ∨ *aa* (*fmdom′* (*snd a*)) *A* ∈ *vs′*
using *a1* **by** (*meson Un-iff*)
then show *fmdom′* (*snd a*) ⊆ *vs* ∨ *fmdom′* (*snd a*) ⊆ *vs′*
using *f3 f2* **by** *meson*
qed

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'*: $(finite\ s) \wedge (finite\ t) \wedge (disjnt\ s\ t) \Longrightarrow (card\ (s \cup t) = card\ s + card\ t)$
  by (*simp add*: *card-Un-disjoint disjnt-def*)

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 −
  {
    fix *x y*
    assume $x \in s\ y \in s$
    then have $f\ x = f\ y \longrightarrow x = y$
      using *assms*(*2*)
      by *blast*
  }
  then have *inj-on f s*
    by (*simp add*: *inj-onI*)
  then show *?thesis*
    using *assms*(*1*) *inj-on-iff-eq-card*
    by *blast*
qed

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

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
  using *assms*
  by *blast*

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 −
  assume *P*: $(\forall\ x.\ x \in s \longrightarrow x \leq k)$
  let *?f* = *id* :: *nat* ⇒ *nat*
  let $?S = \{i.\ i \leq k\}$
  have $s \subseteq ?S$ **using** *P* **by** *blast*
  moreover have *?f* ‘ *?S* = *?S* **by** *auto*
  moreover have *finite ?S* **using** *nat-seg-image-imp-finite* **by** *auto*
  moreover have *finite s* **using** *calculation finite-subset* **by** *auto*
  ultimately show *?thesis* **by** *auto*
qed
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*
  using *mems-le-finite-i* **by** *auto*

— NOTE translated ‘s‘ to ‘nat set‘ (more generality wasn’t required.).
lemma *mem-le-imp-MIN-le*:
  fixes *s* :: *nat set* **and** *k* :: *nat*
  assumes $\exists x.\ (x \in s) \wedge (x \leq k)$
  shows (*Inf* $s \leq k$)
proof −
  from *assms* **obtain** *x* **where** *1*: $x \in s\ x \leq k$
    by *blast*
  {
    assume *C*: *Inf* $s > k$
    then have *Inf* $s > x$ **using** *1(2)*
      by *fastforce*
    then have *False*
      using *1(1) cInf-lower leD*
      by *fast*
  }
  then show *?thesis*
    by *fastforce*
qed

— NOTE nat –> bool is the type of a HOL4 set and was translated to 'nat set'.
— NOTE We cannot use 'Min' instead of 'Inf' because there is no indication that 'n. s n' will be finite. Without that *Min* $\{n.\ s\ n\} \in \{n.\ s\ n\}$ is not necessarily true.

lemma *mem-lt-imp-MIN-lt*:
  fixes s :: *nat set* **and** k :: *nat*
  assumes $(\exists x.\ x \in s \wedge x < k)$
  shows $(Inf\ s) < k$
proof –
  obtain x **where** *1*: $x \in s$ $x < k$
    using *assms*
    by *blast*
  then have *2*: $s \neq \{\}$
    by *blast*
  then have $Inf\ s \in s$
    using *Inf-nat-def LeastI*
    by *force*
  moreover have $\forall x{\in}s.\ Inf\ s \leq x$
    by (*simp add*: *cInf-lower*)
  ultimately show $(Inf\ s) < k$
    using *assms leD*
    by *force*
qed

— NOTE type for 'k' had to be fixed (type unordered error; also not true for e.g. real sets).

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*
  using *assms bounded-nat-set-is-finite*
  by *blast*

lemma *bound-main-lemma-2*: $\bigwedge(s :: nat\ set)\ k.\ (s \neq \{\}) \wedge (\forall x.\ x \in s \longrightarrow x \leq k) \Longrightarrow Sup\ s \leq k$
proof –
  fix s :: *nat set* **and** k
  {
    assume *P1*: $s \neq \{\}$
    assume *P2*: $(\forall x.\ x \in s \longrightarrow x \leq k)$
    have *finite s* **using** *P2 mems-le-finite* **by** *auto*
    moreover have $Max\ s \in s$ **using** *P1 calculation Max-in* **by** *auto*
    moreover have $Max\ s \leq k$ **using** *P2 calculation* **by** *auto*
  }
  then show $(s \neq \{\}) \wedge (\forall x.\ x \in s \longrightarrow x \leq k) \Longrightarrow Sup\ s \leq k$
    by (*simp add*: *Sup-nat-def*)
qed

— NOTE type of 'k' fixed to nat to be able to use 'bound_child_parent_neq_mems_state_set_neq_len'.

lemma *bound-child-parent-not-eq-last-diff-paths*: $\bigwedge s\ (k :: nat)$.
  $(s \neq \{\})$
  $\Longrightarrow (\forall x.\ x \in s \longrightarrow x < k)$
  $\Longrightarrow Sup\ s < k$

  by (*simp add*: *Sup-nat-def bound-child-parent-neq-mems-state-set-neq-len*)

lemma *FINITE-ALL-DISTINCT-LISTS-i*:
  fixes *P*
  assumes *finite P*
  shows
    $\{p.\ distinct\ p \wedge set\ p \subseteq P\}$
    $= \{[]\} \cup (\bigcup\ ((\lambda e.\ \{e\ \#\ p0\ |\ p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P - \{e\})\})\ `\ P))$
proof −
  let $?A=\{p.\ distinct\ p \wedge set\ p \subseteq P\ \}$
  let $?B=\{[]\} \cup (\bigcup\ ((\lambda e.\ \{e\ \#\ p0\ |\ p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P - \{e\})\})\ `\ P))$
  {
    {
      fix *a*
      assume *P*: $a \in ?A$
      then have $a \in ?B$
      proof (*cases a*)

The empty list is distinct and its corresponding set is the empty set which is a trivial subset of '?B'. The 'Nil' case can therefore be derived by automation.

        case (*Cons h list*)
        {
          let $?b'=h$
          {
            from *P* **have** $set\ a \subseteq P$
              by *simp*
            then have $set\ list \subseteq (P - \{h\})$
              using *P dual-order.trans local.Cons*
              by *auto*
          }
          moreover from *P Cons*
          have *distinct list*
            by *force*
           ultimately have $a \in ((\lambda e.\ \{e\ \#\ p0\ |\ p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P -$
$\{e\})\})\ ?b')$
            using *Cons*
            by *blast*
          moreover {
            from *P Cons* **have** $?b' \in set\ a$
              by *simp*
            moreover from *P* **have** $set\ a \subseteq P$
              by *simp*
            ultimately have $?b' \in P$

```
          by auto
        }
        ultimately have
         ∃b' ∈ P. a ∈ ((λe. {e # p0 | p0. distinct p0 ∧ set p0 ⊆ (P − {e})}) b')
          by meson
      }
      then obtain b' where
        b' ∈ P a ∈ ((λe. {e # p0 | p0. distinct p0 ∧ set p0 ⊆ (P − {e})}) b')
        by blast
      then show ?thesis
        by blast
    qed blast
  }
  then have ?A ⊆ ?B
    by auto
}
moreover {
  {
    fix b
    assume P: b ∈ ?B
    have b ∈ ?A
```

The empty list is in '?B' by construction. The 'Nil' case can therefore be derived straightforwardly.

```
    proof (cases b)
      case (Cons a list)
      from P Cons obtain b' where a:
        b' ∈ P b ∈ {b' # p0 | p0. distinct p0 ∧ set p0 ⊆ (P − {b'})}
        by fast
      then obtain p0 where b: b = b' # p0 distinct p0 set p0 ⊆ (P − {b'})
        by blast
      then have distinct (b' # p0)
        by (simp add: subset-Diff-insert)
      moreover have set (b' # p0) ⊆ P
        using a(1) b(3)
        by auto
      ultimately show ?thesis
        using b(1)
        by fast
    qed simp
  }
  then have ?B ⊆ ?A
    by blast
}
ultimately show ?thesis
  using set-eq-subset
  by blast
qed
```

```
lemma FINITE-ALL-DISTINCT-LISTS:
  fixes P
  assumes finite P
  shows finite {p. distinct p ∧ set p ⊆ P}
  using assms
proof (induction card P arbitrary: P)
  case 0
  then have P = {}
    by force
  then show ?case
    using 0
    by simp
next
  case (Suc x)
  {
```

Proof the finiteness of the union by proving both sets of the union are finite. The singleton set '[]' is trivially finite.

```
    {
      {
        fix e
        assume P: e ∈ P
        have
          {e # p0 | p0. distinct p0 ∧ set p0 ⊆ P − {e}}
          = (λp. e # p) ' { p. distinct p ∧ set p ⊆ P − {e}}
          by blast
        moreover {
          let ?P′=P − {e}
          from Suc.prems
          have finite ?P′
            by blast
```

The finiteness can now be shown using the induction hypothesis. However 'e' might already be contained in '?P', so we have to split cases first.

```
          have finite ((λp. e # p) ' {p. distinct p ∧ set p ⊆ ?P′})
          proof (cases e ∈ P)
            case True
            then have x = card ?P′ using Suc.prems Suc(2)
              by fastforce
            moreover from Suc.prems
            have finite ?P′
              by blast
            ultimately show ?thesis
              using Suc(1)
              by blast
          next
            case False
            then have ?P′ = P
              by simp
```

then have *finite* $\{p.\ distinct\ p \wedge set\ p \subseteq ?P'\}$
using *False P* **by** *linarith*
then show *?thesis*
using *finite-imageI*
by *blast*
qed
}
ultimately have *finite* $\{e \# p0 \mid p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P - \{e\})\}$
by *argo*
}
then have *finite* $(\bigcup ((\lambda e.\ \{e \# p0 \mid p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P - \{e\})\})$
$`\ P))$
using *Suc.prems*
by *blast*
}
then have
finite $(\{[]\} \cup (\bigcup ((\lambda e.\ \{e \# p0 \mid p0.\ distinct\ p0 \wedge set\ p0 \subseteq (P - \{e\})\})\ `$
$P)))$
using *finite-Un*
by *blast*
}
then show *?case*
using *FINITE-ALL-DISTINCT-LISTS-i*[*OF Suc.prems*]
by *force*
qed

lemma *subset-inter-diff-empty*:
assumes $s \subseteq t$
shows $(s \cap (u - t) = \{\})$
using *assms*
by *auto*

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 $\equiv$ $\{$*length as'* $\mid$ *as'*.
$($*as'* $\in$ *valid-plans PROB*$) \wedge ($*exec-plan s as'* $=$ *exec-plan s as*$)\}$

definition *MPLS-charles* **where**
MPLS-charles PROB $\equiv$ $\{$*Inf* $($*PLS-charles* $($*fst p*$)$ $($*snd p*$)$ *PROB*$)$ $\mid$ *p*.
$(($*fst p*$) \in$ *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 −
let *?S* = {*as'*. (*exec-plan s as'* = *exec-plan s as*) ∧ (*subseq as' as*)}
let *?S1* = *length* ' {*as'*. (*exec-plan s as'* = *exec-plan s as*) }
let *?S2* = *length* ' {*as'*. (*subseq as' as*)}
let *?n* = *length as* + *1*
have *finite ?S2*
using *bounded-nat-set-is-finite*[**where** *n* = *?n* **and** *N* = *?S2*]
by *fastforce*
moreover have *length* ' *?S* ⊆ (*?S1* ∩ *?S2*)
by *blast*
ultimately have *finite* (*length* ' *?S*)
using *infinite-super*
by *auto*

then show *?thesis*
unfolding *PLS-def*
by *blast*
qed

— NOTE name shortened.
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*)}
)

unfolding *problem-plan-bound-def MPLS-def*
by *blast*

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 −
{
have (
{*Inf* (*PLS s as*) | *s as*. (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)}
) = ((λ(*s*, *as*). *Inf* (*PLS s as*)) ‘ {(*s*, *as*).
(*s* ∈ *valid-states PROB*)
∧ (*as* ∈ *valid-plans PROB*)
})

by *fast*
also have . . . =

```
    (λ(s, as). Inf (PLS s as)) `
    ({s. fmdom' s = prob-dom PROB} × {as. set as ⊆ PROB})

    unfolding valid-states-def valid-plans-def
    by simp
  finally have
     Sup ({Inf (PLS s as) | s as. (s ∈ valid-states PROB) ∧ (as ∈ valid-plans
PROB)})
     = Sup (
       (λ(s, as). Inf (PLS s as)) `
       ({s. fmdom' s = prob-dom PROB} × {as. set as ⊆ PROB})
     )

    by argo
 }
 moreover have
   problem-plan-bound PROB
   =
     Sup ((λ(s, as). Inf (PLS s as)) `
     ({s. fmdom' s = prob-dom PROB} × {as. set as ⊆ PROB}))

   unfolding problem-plan-bound-def MPLS-def valid-states-def valid-plans-def
   by fastforce
 ultimately show
   problem-plan-bound PROB
   = Sup ({Inf (PLS s as) | s as.
     (s ∈ valid-states PROB)
     ∧ (as ∈ valid-plans PROB)
   })

   by argo
qed
```

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*))
)

using *valid-states-def*
by *simp*

— NOTE name shortened.
— NOTE second declaration skipped (declared twice in source).
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 −
obtain *as'* **where** *1*:
exec-plan s as = *exec-plan s as' subseq as' as length as'* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms main-lemma*
by *blast*
let *?x*=*length as'*
have *?x* ∈ *PLS s as*
unfolding *PLS-def*
using *1*
by *simp*
moreover have *?x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *1*(*3*)
by *blast*
ultimately show (∃ *x*.
(*x* ∈ *PLS s as*)
∧ (*x* ≤ (*2* ^ *card* (*prob-dom PROB*)) − *1*)
)
unfolding *PLS-def*
by *blast*

qed

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 −
let *?mpls* = *MPLS PROB*
— NOTE obtain p = (s, as) where 'x = Inf (PLS s as)' from premise.
have *?mpls* =
(λ (*s*, *as*). *Inf* (*PLS s as*)) '
{(*s*, *as*). (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)}

using *MPLS-def*
by *blast*
then obtain *s* :: (*'a*, *bool*) *fmap* **and** *as* :: ((*'a*, *bool*) *fmap* × (*'a*, *bool*) *fmap*) *list*
where *obtain-s-as*: *x* ∈
((λ (*s*, *as*). *Inf* (*PLS s as*)) '
{(*s*, *as*). (*s* ∈ *valid-states PROB*) ∧ (*as* ∈ *valid-plans PROB*)})

using *assms*(*2*)
by *blast*
then have
x ∈ {*Inf* (*PLS* (*fst p*) (*snd p*)) | *p*. (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)}
using *assms*(*1*) *obtain-s-as*
by *auto*
then have
∃ *p*. *x* = *Inf* (*PLS* (*fst p*) (*snd p*)) ∧ (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)
by *blast*
then obtain *p* :: (*'a*, *bool*) *fmap* × ((*'a*, *bool*) *fmap* × (*'a*, *bool*) *fmap*) *list* **where** *obtain-p*:
x = *Inf* (*PLS* (*fst p*) (*snd p*)) (*fst p* ∈ *valid-states PROB*) (*snd p* ∈ *valid-plans PROB*)
by *blast*
then have *fst p* ∈ *valid-states PROB snd p* ∈ *valid-plans PROB*
using *obtain-p*
by *blast+*
then obtain *x'* :: *nat* **where** *obtain-x'*:
x' ∈ *PLS* (*fst p*) (*snd p*) ∧ *x'* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms*(*1*) *in-PLS-leq-2-pow-n*[**where** *s* = *fst p* **and** *as* = *snd p*]
by *blast*
then have *1*: *x'* ≤ *2* ^ *card* (*prob-dom PROB*) − *1* *x'* ∈ *PLS* (*fst p*) (*snd p*)
x = *Inf* (*PLS* (*fst p*) (*snd p*)) *finite* (*PLS* (*fst p*) (*snd p*))
using *obtain-x' obtain-p finite-PLS*
by *blast+*

```
  moreover have x ≤ x′
    using 1(2, 4) obtain-p(1) cInf-le-finite
    by blast
  ultimately show (x ≤ 2 ^ card (prob-dom PROB) − 1)
    by linarith
qed


lemma FINITE-MPLS:
  assumes finite (Pi :: ′a problem)
  shows finite (MPLS Pi)
proof −
  have ∀ x ∈ MPLS Pi. x ≤ 2 ^ card (prob-dom Pi) − 1
    using assms in-MPLS-leq-2-pow-n
    by blast
  then show finite (MPLS Pi)
    using mems-le-finite[of MPLS Pi 2 ^ card (prob-dom Pi) − 1]
    by blast
qed


— NOTE 'fun' because of multiple defining equations.
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)
  by (induction as arbitrary: s) auto


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))
  using assms
proof (induction as arbitrary: s Pi)
  case cons: (Cons a as)
  then have 1: a ∈ Pi as ∈ valid-plans Pi
    using valid-plan-valid-head valid-plan-valid-tail
    by metis+
  then show ?case
  proof (cases as)
    case Nil
    {
      have state-succ s a ∈ valid-states Pi
        using 1 cons.prems(2) valid-action-valid-succ
```

```
        by blast
      then have valid-path Pi [state-succ s a]
        using 1 cons.prems(2) cons.IH
        by force
      moreover have (∃ aa. aa ∈ Pi ∧ exec-plan s [aa] = state-succ s a)
        using 1(1)
        by fastforce
      ultimately have valid-path Pi (statelist' s [a])
        using cons.prems(2)
        by simp
    }
    then show ?thesis
      using Nil
      by blast
  next
    case (Cons b list)
    {
      have s ∈ valid-states Pi
        using cons.prems(2)
        by simp
          — TODO this step is inefficient ( 5s).
      then have
        valid-path Pi (state-succ s a # statelist' (state-succ (state-succ s a) b) list)
        using 1 cons.IH cons.prems(2) Cons lemma-1-i
        by fastforce
      moreover have
        (∃ aa b. (aa, b) ∈ Pi ∧ state-succ s (aa, b) = state-succ s a)
        using 1(1) surjective-pairing
        by metis
      ultimately have valid-path Pi (statelist' s (a # b # list))
        using cons.prems(2)
        by auto
    }
    then show ?thesis
      using Cons
      by blast
  qed
qed simp


— TODO explicit proof.
lemma statelist'-exec-plan:
  fixes a s p
  assumes (statelist' s as = p)
  shows (exec-plan s as = last p)
  using assms
  apply(induction as arbitrary: s p)
   apply(auto)
  apply(cases as)
```

by
(*metis LENGTH-statelist′ One-nat-def add-Suc-right list.size*(*3*) *nat.simps*(*3*))
(*metis* (*no-types*) *LENGTH-statelist′ One-nat-def add-Suc-right list.size*(*3*) *nat.simps*(*3*))

lemma *statelist′-EQ-NIL*: *statelist′ s as* ≠ []
by (*cases as*) *auto*

— NOTE added lemma.
lemma *statelist′-TAKE-i*:
assumes *Suc m* ≤ *length* (*a* # *as*)
shows *m* ≤ *length as*
using *assms*
by (*induction as arbitrary*: *a m*) *auto*

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*))
using *assms*
proof (*induction as arbitrary*: *s p*)
case *Nil*
{
fix *n*
assume *P1*: *n* ≤ *length* []
then have *exec-plan s* (*take n* []) = *s*
by *simp*
moreover have *p* ! *0* = *s*
using *Nil.prems*
by *force*
ultimately have *exec-plan s* (*take n* []) = *p* ! *n*
using *P1*
by *simp*
}
then show *?case* **by** *blast*
next
case (*Cons a as*)
{
fix *n*
assume *P2*: *n* ≤ *length* (*a* # *as*)
then have *exec-plan s* (*take n* (*a* # *as*)) = *p* ! *n*
using *Cons.prems*
proof (*cases n* = *0*)
case *False*
then obtain *m* **where** *a*: *n* = *Suc m*
using *not0-implies-Suc*
by *presburger*

moreover have *b*: *statelist′ s* (*a* # *as*) ! *n* = *statelist′* (*state-succ s a*) *as* ! *m*
using *a nth-Cons-Suc*
by *simp*
moreover have *c*: *exec-plan s* (*take n* (*a* # *as*)) = *exec-plan* (*state-succ s a*) (*take m as*)
using *a*
by *force*
moreover have *m* ≤ *length as*
using *a P2 statelist′-TAKE-i*
by *simp*
moreover have
exec-plan (*state-succ s a*) (*take m as*) = *statelist′* (*state-succ s a*) *as* ! *m*
using *calculation*(*2*, *3*, *4*) *Cons.IH*
by *blast*
ultimately show *?thesis*
using *Cons.prems*
by *argo*
qed *fastforce*
}
then show *?case* **by** *blast*
qed

lemma *MPLS-nempty*:
fixes *PROB* :: ((′*a*, ′*b*) *fmap* × (′*a*, ′*b*) *fmap*) *set*
assumes *finite PROB*
shows *MPLS PROB* ≠ {}
proof −
let *?S*={(*s*, *as*). *s* ∈ *valid-states PROB* ∧ *as* ∈ *valid-plans PROB*}
— NOTE type of 's' had to be fixed for 'valid_states_nempty'.
obtain *s* :: (′*a*, ′*b*) *fmap* **where** *s* ∈ *valid-states PROB*
using *assms valid-states-nempty*
by *blast*
moreover have [] ∈ *valid-plans PROB*
using *empty-plan-is-valid*
by *auto*
ultimately have (*s*, []) ∈ *?S*
by *blast*
then show *?thesis*
unfolding *MPLS-def*
by *blast*
qed

theorem *bound-main-lemma*:
fixes *PROB* :: ′*a problem*
assumes *finite PROB*
shows (*problem-plan-bound PROB* ≤ (*2* ^ (*card* (*prob-dom PROB*))) − *1*)
proof −

have *MPLS PROB* ≠ {}
  using *assms MPLS-nempty*
  by *auto*
moreover have (∀ *x*. *x* ∈ *MPLS PROB* ⟶ *x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*)
  using *assms in-MPLS-leq-2-pow-n*
  by *blast*
ultimately show *?thesis*
  unfolding *problem-plan-bound-def*
  using *cSup-least*
  by *blast*
qed

— NOTE types in premise had to be fixed to be able to match 'valid_as_valid_exec'.
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 −
  {
    fix *PROB* :: *'a problem* **and** *as* **and** *s* :: *'a state*
    assume *P1*: (*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*)
)
    have (*exec-plan s as* ∈ *valid-states PROB*)
      using *assms P1 valid-as-valid-exec*

```
      by blast
    then have (P PROB)
      ∧ (as ∈ valid-plans PROB)
      ∧ (s ∈ (valid-states PROB))
      ⟶ (∃ x.
        (x ∈ PLS s as)
        ∧ (x < f PROB)
      )

      unfolding PLS-def
      using P1
      by force
  }
  then show (∀ PROB s as.
    (P PROB)
    ∧ (as ∈ valid-plans PROB)
    ∧ (s ∈ (valid-states PROB))
    ⟶ (∃ x.
      (x ∈ PLS s as)
      ∧ (x < f PROB)
    )
  )
    using assms
    by simp
qed
```

— NOTE types of premise had to be fixed to be able to use lemma 'bound_child_parent_card_state_set_cons'.

```
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 −
  {
    fix PROB :: 'a problem and as and s :: 'a state
    assume (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*)
)
then have (∃ *x*. *x* ∈ *PLS s as* ∧ *x* < *f PROB*)
using *assms*(*1*) *bound-child-parent-card-state-set-cons*[**where** *P* = *P* **and** *f* = *f*]
by *presburger*
}
note *1* = *this*
{
fix *PROB x*
assume *P1*: *P PROB x* ∈ *MPLS PROB*
— TODO refactor 'x_in_MPLS_if' and use here.
then obtain *s as* **where** *a*:
x = *Inf* (*PLS s as*) *s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*
unfolding *MPLS-def*
by *auto*
moreover have (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* < *f PROB*)
)
using *P1*(*1*) *assms calculation*(*2*, *3*)
by *blast*
ultimately obtain *x'* **where** *x'* ∈ *PLS s as x'* < *f PROB*
using *P1 1*
by *blast*
then have *x* < *f PROB*
using *a*(*1*) *mem-lt-imp-MIN-lt*
by *fastforce*
}
then show *?thesis*
by *blast*
qed

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 ($\forall$ *PROB s as*.
P PROB $\wedge$ *finite PROB* $\wedge$ *as* $\in$ (*valid-plans PROB*) $\wedge$ (*s* $\in$ (*valid-states PROB*))
$\longrightarrow$ ($\exists$ *x*. (*x* $\in$ *PLS s as*) $\wedge$ *x* $<$ *f PROB*)
)
proof −
{
fix *PROB s as*
assume *P PROB finite PROB as* $\in$ (*valid-plans PROB*) *s* $\in$ (*valid-states PROB*)
($\exists$ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
$\wedge$ *subseq as' as*
$\wedge$ *length as'* $<$ *f PROB*
)

then obtain *as'* **where**
(*exec-plan s as* = *exec-plan s as'*) *subseq as' as length as'* $<$ *f PROB*
by *blast*
moreover have *length as'* $\in$ *PLS s as*
unfolding *PLS-def*
using *calculation*
by *fastforce*
ultimately have ($\exists$ *x*. (*x* $\in$ *PLS s as*) $\wedge$ *x* $<$ *f PROB*)
by *blast*
}
then show ($\forall$ *PROB s as*.
P PROB
$\wedge$ *finite PROB*
$\wedge$ *as* $\in$ (*valid-plans PROB*)
$\wedge$ (*s* $\in$ (*valid-states PROB*))
$\longrightarrow$ ($\exists$ *x*. (*x* $\in$ *PLS s as*) $\wedge$ *x* $<$ *f PROB*)
)
using *assms*
by *auto*
qed

lemma *bound-on-all-plans-bounds-MPLS-finite*:
fixes *P f*
assumes ($\forall$ *PROB 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 (∀ *PROB x*.
P PROB ∧ *finite PROB*
⟶ (*x* ∈ *MPLS PROB*)
⟶ *x* < *f PROB*
)
proof −
{
fix *PROB x*
assume *P1*: *P PROB finite PROB x* ∈ *MPLS PROB*
— TODO refactor 'x_in_MPLS_if' and use here.
then obtain *s as* **where** *a*:
x = *Inf* (*PLS s as*) *s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*
unfolding *MPLS-def*
by *auto*
moreover have (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* < *f PROB*)
)
using *P1*(*1*, *2*) *assms calculation*(*2*, *3*)
by *blast*
moreover obtain *x'* **where** *x'* ∈ *PLS s as x'* < *f PROB*
using *PLS-def calculation*(*4*)
by *fastforce*
then have *x* < *f PROB*
using *a*(*1*) *mem-lt-imp-MIN-lt*
by *fastforce*
}
then show *?thesis*
using *assms*
by *blast*
qed

lemma *bound-on-all-plans-bounds-problem-plan-bound*:
fixes *P f*
assumes (∀ *PROB 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*.

```
    (P PROB)
    ∧ finite PROB
    ⟶ (problem-plan-bound PROB < f PROB)
  )
proof −
  have 1: ∀ PROB x.
    P PROB
    ∧ finite PROB
    ⟶ x ∈ MPLS PROB
    ⟶ x < f PROB

    using assms bound-on-all-plans-bounds-MPLS-finite
    by blast
  {
    fix PROB x
    assume P PROB ∧ finite PROB
      ⟶ x ∈ MPLS PROB
      ⟶ x < f PROB

    then have ∀ PROB.
      P PROB ∧ finite PROB
      ⟶ problem-plan-bound PROB < f PROB

      unfolding problem-plan-bound-def
      using 1 bound-child-parent-not-eq-last-diff-paths 1 MPLS-nempty
      by metis
    then have ∀ PROB.
      P PROB ∧ finite PROB
      ⟶ problem-plan-bound PROB < f PROB

      using MPLS-nempty
      by blast

  }
  then show (∀ PROB.
    (P PROB)
    ∧ finite PROB
    ⟶ (problem-plan-bound PROB < f PROB)
  )
    using 1
    by blast
qed


lemma bound-child-parent-card-state-set-cons-thesis:
  assumes finite PROB (∀ as s.
    as ∈ (valid-plans PROB)
    ∧ s ∈ (valid-states PROB)
    ⟶ (∃ 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 $(\exists x.\ (x \in PLS\ s\ as) \wedge x < k)$
unfolding *PLS-def*
using *assms*
by *fastforce*

— NOTE added lemma.
— TODO refactor/move up.
lemma *x-in-MPLS-if*:
fixes *x PROB*
assumes $x \in MPLS\ PROB$
shows $\exists s\ as.\ s \in valid\text{-}states\ PROB \wedge as \in valid\text{-}plans\ PROB \wedge x = Inf\ (PLS\ s\ as)$
using *assms*
unfolding *MPLS-def*
by *fast*

lemma *bound-on-all-plans-bounds-MPLS-thesis*:
assumes *finite PROB* $(\forall as\ s.$
$(s \in valid\text{-}states\ PROB)$
$\wedge\ (as \in valid\text{-}plans\ PROB)$
$\longrightarrow (\exists as'.$
$(exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as')$
$\wedge\ (subseq\ as'\ as)$
$\wedge\ (length\ as' < k)$
)
) $(x \in MPLS\ PROB)$
shows $(x < k)$
proof −
obtain *s as* **where** *1*: $s \in valid\text{-}states\ PROB\ as \in valid\text{-}plans\ PROB\ x = Inf\ (PLS\ s\ as)$
using *assms(3) x-in-MPLS-if*
by *blast*
then obtain $x' :: nat$ **where** $x' \in PLS\ s\ as\ x' < k$
using *assms(1, 2) bound-child-parent-card-state-set-cons-thesis*
by *blast*
then have $Inf\ (PLS\ s\ as) < k$
using *mem-lt-imp-MIN-lt*
by *blast*
then show $x < k$
using *1*
by *simp*
qed

— NOTE added lemma.

```
lemma bounded-MPLS-contains-supremum:
  fixes PROB
  assumes finite PROB (∃ k. ∀ x ∈ MPLS PROB. x < k)
  shows Sup (MPLS PROB) ∈ MPLS PROB
proof −
  obtain k where ∀ x ∈ MPLS PROB. x < k
    using assms(2)
    by blast
  moreover have finite (MPLS PROB)
    using assms(2) finite-nat-set-iff-bounded
    by presburger
  moreover have MPLS PROB ≠ {}
    using assms(1) MPLS-nempty
    by auto
  ultimately show Sup (MPLS PROB) ∈ MPLS PROB
    unfolding Sup-nat-def
    by simp
qed

lemma bound-on-all-plans-bounds-problem-plan-bound-thesis':
  assumes finite PROB (∀ as s.
      s ∈ (valid-states PROB)
      ∧ as ∈ (valid-plans PROB)
      ⟶ (∃ as'.
        (exec-plan s as = exec-plan s as')
        ∧ subseq as' as
        ∧ length as' < k
      )
    )
  shows problem-plan-bound PROB < k
proof −
  have 1: ∀ x ∈ MPLS PROB. x < k
    using assms(1, 2) bound-on-all-plans-bounds-MPLS-thesis
    by blast
  then have Sup (MPLS PROB) ∈ MPLS PROB
    using assms(1) bounded-MPLS-contains-supremum
    by auto
  then have Sup (MPLS PROB) < k
    using 1
    by blast
  then show ?thesis
    unfolding problem-plan-bound-def
    by simp
qed


lemma bound-on-all-plans-bounds-problem-plan-bound-thesis:
```

assumes *finite PROB* (∀ *as s*.
(*s* ∈ *valid-states PROB*)
∧ (*as* ∈ *valid-plans PROB*)
⟶ (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)
∧ (*subseq as' as*)
∧ (*length as'* ≤ *k*)
)
)
shows (*problem-plan-bound PROB* ≤ *k*)
proof −
have *1*: ∀ *x*∈*MPLS PROB*. *x* < *k* + *1*
using *assms*(*1*, *2*) *bound-on-all-plans-bounds-MPLS-thesis*[**where** *k* = *k* + *1*]
Suc-eq-plus1
less-Suc-eq-le
by *metis*
then have *Sup* (*MPLS PROB*) ∈ *MPLS PROB*
using *assms*(*1*) *bounded-MPLS-contains-supremum*
by *fast*
then show (*problem-plan-bound PROB* ≤ *k*)
unfolding *problem-plan-bound-def*
using *1*
by *fastforce*
qed

lemma *bound-on-all-plans-bounds-problem-plan-bound-*:
fixes *P f PROB*
assumes (∀ *PROB' 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'*)
)
) (*P PROB*) *finite PROB*
shows (*problem-plan-bound PROB* < *f PROB*)
unfolding *problem-plan-bound-def MPLS-def*
using *assms bound-on-all-plans-bounds-problem-plan-bound-thesis' expanded-problem-plan-bound-thm-1*
by *metis*

lemma *S-VALID-AS-VALID-IMP-MIN-IN-PLS*:
fixes *PROB s as*
assumes (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
shows (*Inf* (*PLS s as*) ∈ (*MPLS PROB*))
unfolding *MPLS-def*
using *assms*

by *fast*

— NOTE type of ‘s‘ had to be fixed (type mismatch in goal).
— NOTE premises rewritten to implications for proof set up.
lemma *problem-plan-bound-ge-min-pls*:
fixes *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as k*
assumes *finite PROB* (*s* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)
(*problem-plan-bound PROB* ≤ *k*)
shows (*Inf* (*PLS s as*) ≤ *problem-plan-bound PROB*)
proof −
have *Inf* (*PLS s as*) ∈ *MPLS PROB*
using *assms*(*2*, *3*) *S-VALID-AS-VALID-IMP-MIN-IN-PLS*
by *blast*
moreover have *finite* (*MPLS PROB*)
using *assms*(*1*) *FINITE-MPLS*
by *blast*
ultimately have *Inf* (*PLS s as*) ≤ *Sup* (*MPLS PROB*)
using *le-cSup-finite*
by *blast*
then show *?thesis*
unfolding *problem-plan-bound-def*
by *simp*
qed

lemma *PLS-NEMPTY*:
fixes *s as*
shows *PLS s as* ≠ {}
unfolding *PLS-def*
by *blast*

lemma *PLS-nempty-and-has-min*:
fixes *s as*
shows (∃ *x*. (*x* ∈ *PLS s as*) ∧ (*x* = *Inf* (*PLS s as*)))
proof −
have *PLS s as* ≠ {}
using *PLS-NEMPTY*
by *blast*
then have *Inf* (*PLS s as*) ∈ *PLS s as*
unfolding *Inf-nat-def*
using *LeastI-ex Max-in finite-PLS*
by *metis*
then show *?thesis*
by *blast*
qed

```
lemma PLS-works:
  fixes x s as
  assumes (x ∈ PLS s as)
  shows(∃ as'.
      (exec-plan s as = exec-plan s as')
      ∧ (length as' = x)
      ∧ (subseq as' as)
    )
  using assms
  unfolding PLS-def
  by (smt imageE mem-Collect-eq)
```

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

```
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 –
  have problem-plan-bound PROB ≤ 2 ^ card (prob-dom PROB) − 1
    using assms(1) bound-main-lemma
    by blast
  then have 1: Inf (PLS s as) ≤ problem-plan-bound PROB
    using
      assms(1, 2, 3)
      problem-plan-bound-ge-min-pls
    by blast
  then have ∃ x. x ∈ PLS s as ∧ x = Inf (PLS s as)
    using PLS-nempty-and-has-min
    by blast
  then have Inf (PLS s as) ∈ (PLS s as)
    by blast
  then obtain as' where 2:
    exec-plan s as = exec-plan s as' length as' = Inf (PLS s as) subseq as' as
    using PLS-works
    by blast
  then have length as' ≤ problem-plan-bound PROB
    using 1
    by argo
  then show (∃ as'.
    (exec-plan s as = exec-plan s as')
    ∧ (subseq as' as)
    ∧ (length as' ≤ problem-plan-bound PROB)
  )
    using 2(1) 2(3)
```

by *blast*
qed

— NOTE name shortened.
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 −
— TODO (*s*, []) ∈ {} could be refactored (this is used in 'MPLS_nempty' too).
have [] ∈ *valid-plans PROB*
using *empty-plan-is-valid*
by *blast*
then have (*s*, []) ∈ {(*s*, *as*). *as* ∈ *valid-plans PROB*}
by *simp*
then show *MPLS-s PROB s* ≠ {}
unfolding *MPLS-s-def*
by *blast*
qed

— NOTE name shortened.
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*)
)
)

using *assms*
unfolding *PLS-def*
by *fastforce*

— NOTE added lemma.
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 −
let *?S*={(*s*, *as*) | *as*. *as* ∈ *valid-plans PROB*}
obtain *x′* **where** *1*:
x′ ∈ *?S*
x = (λ (*s*, *as*). *Inf* (*PLS s as*)) *x′*
using *assms*
unfolding *MPLS-s-def*
by *blast*
let *?as*=*snd x′*
let *?s*=*fst x′*
have *?as* ∈ *valid-plans PROB*
using *1*(*1*)
by *auto*
moreover have *?s* = *s*
using *1*(*1*)
by *fastforce*
moreover have *x* = *Inf* (*PLS ?s ?as*)
using *1*(*2*)
by (*simp add*: *case-prod-unfold*)
ultimately show *?thesis*
by *blast*
qed

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*)
)
using *assms*
unfolding *MPLS-def*

proof −
have *1*: ∀ *PROB s as.*
finite PROB ∧ *P PROB* ∧ *as* ∈ *valid-plans PROB* ∧ *s* ∈ *valid-states PROB* ⟶
(∃ *x. x* ∈ *PLS s as* ∧ *x* < *f PROB s*)
using *bound-on-all-plans-bounds-PLS-s*[*OF assms*] **.**
{
fix *PROB x* **and** *s* :: (*'a*, *'b*) *fmap*
assume *P1*: *finite PROB* (*P PROB*) (*s* ∈ *valid-states PROB*)
{
assume (*x* ∈ *MPLS-s PROB s*)
then obtain *as* **where** *i*: *x* = *Inf* (*PLS s as*) *as* ∈ *valid-plans PROB*
using *P1 bound-on-all-plans-bounds-MPLS-s-i*
by *blast*
then obtain *x'* **where** *x'* ∈ *PLS s as x'* < *f PROB s*
using *P1 i 1*
by *blast*
then have *x* < *f PROB s*
using *mem-lt-imp-MIN-lt i*(*1*)
by *blast*
}
then have (*x* ∈ *MPLS-s PROB s*) ⟶ (*x* < *f PROB s*)
by *blast*
}
then show *?thesis*
by *blast*
qed

— NOTE added lemma.
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 −
have *MPLS-s PROB s* ≠ {}
using *bound-main-lemma-s-3*
by *fast*
then have *Sup* (*MPLS-s PROB s*) ∈ *MPLS-s PROB s*
using *assms Sup-nat-def bounded-nat-set-is-finite*
by *force*

then show *Sup* (*MPLS-s PROB s*) < *k*
using *assms*
by *blast*
qed

— NOTE type of ‘P‘ had to be fixed (type mismatch in goal).
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 −
— NOTE manual instantiation is required (automation fails otherwise).
have ∀ (*PROB* :: *'a problem*) *x s*.
finite PROB ∧ *P PROB* ∧ *s* ∈ *valid-states PROB*
⟶ *x* ∈ *MPLS-s PROB s*
⟶ *x* < *f PROB s*

using *assms bound-on-all-plans-bounds-MPLS-s*[*of P f*]
by *simp*
then show
finite PROB ∧ (*P PROB*) ∧ (*s* ∈ *valid-states PROB*) ⟶ (*problem-plan-bound-s PROB s* < *f PROB s*)
unfolding *problem-plan-bound-s-def*
using *Sup-MPLS-s-lt-if problem-plan-bound-s-def*
by *metis*
qed

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 −
— NOTE derive premise for MP of 'bound_child_parent_lemma_s_2'.
— NOTE type of ‘s‘ had to be fixed (warning in assumption declaration).
{
fix *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as*
assume *P1*: *finite PROB s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*

```
    then obtain as' where a: exec-plan s as = exec-plan s as' subseq as' as
      length as' ≤ card (reachable-s PROB s) − 1
      using P1 main-lemma-reachability-s
      by blast
    then have length as' < card (reachable-s PROB s)
      using P1(1, 2) card-reachable-s-non-zero
      by fastforce
    then have (∃ as'.
     exec-plan s as = exec-plan s as' ∧ subseq as' as ∧ length as' < card (reachable-s
PROB s))

      using a
      by blast
  }
  then have
    finite PROB ∧ True ∧ s ∈ valid-states PROB
    ⟶ problem-plan-bound-s PROB s < card (reachable-s PROB s)

   using bound-child-parent-lemma-s-2[where PROB = PROB and P = λ-. True
and s = s
       and f = λPROB s. card (reachable-s PROB s)]
    by blast
  then show ?thesis
    using assms(1, 2)
    by blast
qed


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 −
  {
    fix PROB :: 'a problem and s :: 'a state and as
    assume finite PROB s ∈ valid-states PROB as ∈ valid-plans PROB
    then obtain as' where a: exec-plan s as = exec-plan s as' subseq as' as
      length as' ≤ problem-plan-bound PROB
      using problem-plan-bound-works
      by blast
    then have length as' < problem-plan-bound PROB + 1
      by linarith
    then have ∃ as'.
        exec-plan s as = exec-plan s as' ∧ subseq as' as ∧ length as' ≤ prob-
lem-plan-bound PROB + 1

      using a
      by fastforce
  }
```

— TODO unsure why a proof is needed at all here.
then have ∀ (*PROB* :: *'a problem*) *as s*.
finite PROB ∧ *True* ∧ *s* ∈ *valid-states PROB* ∧ *as* ∈ *valid-plans PROB*
⟶ (∃ *as'*.
exec-plan s as = *exec-plan s as'* ∧ *subseq as' as* ∧ *length as'* < *problem-plan-bound PROB* + *1*)

by (*metis Suc-eq-plus1 problem-plan-bound-works le-imp-less-Suc*)
then show (*problem-plan-bound-s PROB s* < *problem-plan-bound PROB* + *1*)
using *assms bound-child-parent-lemma-s-2*[**where** *PROB* = *PROB* **and** *s* = *s* **and** *P* = λ-. *True*
and *f* = λ*PROB s*. *problem-plan-bound PROB* + *1*]
by *fast*
qed

— NOTE lemma 'bound_main_lemma_s_1' skipped (this is being equivalently redeclared later).

lemma *AS-VALID-MPLS-VALID*:
fixes *PROB as*
assumes (*as* ∈ *valid-plans PROB*)
shows (*Inf* (*PLS s as*) ∈ *MPLS-s PROB s*)
using *assms*
unfolding *MPLS-s-def*
by *fast*

— NOTE moved up because it's used in the following lemma.
— NOTE type of 's' had to be fixed for 'in_PLS_leq_2_pow_n'.
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 −
obtain *as* :: ((*'a*, *bool*) *fmap* × (*'a*, *bool*) *fmap*) *list* **where** *as* ∈ *valid-plans PROB*
using *empty-plan-is-valid*
by *blast*
then obtain *x* **where** *1*: *x* ∈ *PLS s as x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms in-PLS-leq-2-pow-n*
by *blast*
then have *Inf* (*PLS s as*) ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *mem-le-imp-MIN-le*[**where** *s* = *PLS s as* **and** *k* = *2* ^ *card* (*prob-dom PROB*) − *1*]
by *blast*
then have *x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms*(*3*) *1*

by *blast*
— TODO unsure why a proof is needed here (typing problem?).
then show *?thesis*
using *assms*(*1*, *2*, *3*) *S-VALID-AS-VALID-IMP-MIN-IN-PLS bound-on-all-plans-bounds-MPLS-s-i*

in-MPLS-leq-2-pow-n
by *metis*
qed

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 −
have ∀ *x*∈*MPLS-s PROB s*. *x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms*(*1*, *2*) *bound-main-lemma-s-1* **by** *blast*
then have *1*: *finite* (*MPLS-s PROB s*)
using *mems-le-finite*[**where** *s* = *MPLS-s PROB s* **and** *k* = *2* ^ *card* (*prob-dom PROB*) − *1*]
by *blast*
then have *MPLS-s PROB s* ≠ {}
using *bound-main-lemma-s-3*
by *fast*
then have *Inf* (*PLS s as*) ∈ *MPLS-s PROB s*
using *assms AS-VALID-MPLS-VALID*
by *blast*
then show (*Inf* (*PLS s as*) ≤ *problem-plan-bound-s PROB s*)
unfolding *problem-plan-bound-s-def*
using *1 le-cSup-finite*
by *blast*
qed

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 −
have *1*: ∀ *x*∈*MPLS-s PROB s*. *x* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms bound-main-lemma-s-1*
by *metis*
then have *MPLS-s PROB s* ≠ {}
using *bound-main-lemma-s-3*
by *fast*
then have *Sup* (*MPLS-s PROB s*) ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *1 bound-main-lemma-2*[**where** *s* = *MPLS-s PROB s* **and** *k* = *2* ^ *card* (*prob-dom PROB*) − *1*]

by *blast*
then show *problem-plan-bound-s PROB s* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
unfolding *problem-plan-bound-s-def*
by *blast*
qed

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 −
have *problem-plan-bound-s PROB s* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
using *assms*(*1*, *3*) *bound-main-lemma-s*
by *blast*
then have *1*: *Inf* (*PLS s as*) ≤ *problem-plan-bound-s PROB s*
using *assms problem-plan-bound-s-ge-min-pls*[*of PROB s as 2* ^ *card* (*prob-dom PROB*) − *1*]
by *blast*
then obtain *x* **where** *obtain-x*: *x* ∈ *PLS s as* ∧ *x* = *Inf* (*PLS s as*)
using *PLS-nempty-and-has-min*
by *blast*
then have ∃ *as'*. *exec-plan s as* = *exec-plan s as'* ∧ *length as'* = *Inf* (*PLS s as*) ∧ *subseq as' as*
using *PLS-works*[**where** *s* = *s* **and** *as* = *as* **and** *x* = *Inf* (*PLS s as*)]
obtain-x
by *fastforce*
then show (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*) ∧ (*subseq as' as*)
∧ (*length as'* ≤ *problem-plan-bound-s PROB s*)
)
using *1*
by *metis*
qed

— NOTE skipped second declaration (declared twice in source).
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*)}
using *PLS-def*
by *blast*

— NOTE Set comprehension had to be rewritten to image (there is no pattern matching in the part left of the pipe symbol).

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*)})

  unfolding *problem-plan-bound-charles-def MPLS-charles-def*
  by *blast*

lemma *bound-main-lemma-charles-3*:
  fixes *PROB* :: *'a problem*
  assumes *finite PROB*
  shows *MPLS-charles PROB* ≠ {}
proof −
  have *1*: [] ∈ *valid-plans PROB*
    using *empty-plan-is-valid*
    by *auto*
  then obtain *s* :: *'a state* **where** *obtain-s*: *s* ∈ *valid-states PROB*
    using *assms valid-states-nempty*
    by *auto*
  then have *Inf* (*PLS-charles s* [] *PROB*) ∈ *MPLS-charles PROB*
    unfolding *MPLS-charles-def*
    using *1*
    by *auto*
  then show *MPLS-charles PROB* ≠ {}
    by *blast*
qed

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 −
  obtain *as'* **where** *1*:
    exec-plan s as = *exec-plan s as' subseq as' as length as'* ≤ *2* ^ *card* (*prob-dom PROB*) − *1*
    using *assms main-lemma*
    by *blast*
  then have *as'* ∈ *valid-plans PROB*

using *assms*(*3*) *sublist-valid-plan*
by *blast*
then have *length as'* $\in$ *PLS-charles s as PROB*
unfolding *PLS-charles-def*
using *1*
by *auto*
then show *?thesis*
using *1*(*3*)
by *fast*
qed

— NOTE added lemma.
— NOTE this lemma retrieves ‘s‘, ‘as‘ for a given $x \in$ *MPLS-charles PROB* and characterizes it as the minimum of 'PLS_charles s as PROB'.
lemma *x-in-MPLS-charles-then*:
fixes *PROB s as*
assumes $x \in$ *MPLS-charles PROB*
shows $\exists$ *s as*.
s $\in$ *valid-states PROB* $\wedge$ *as* $\in$ *valid-plans PROB* $\wedge$ *x* = *Inf* (*PLS-charles s as PROB*)

proof −
have $\exists$ *p* $\in$ {*p*. (*fst p*) $\in$ *valid-states PROB* $\wedge$ (*snd p*) $\in$ *valid-plans PROB*}. *x* = *Inf* (*PLS-charles* (*fst p*) (*snd p*) *PROB*)
using *MPLS-charles-def assms*
by *fast*
then obtain *p* **where** *1*:
p $\in$ {*p*. (*fst p*) $\in$ *valid-states PROB* $\wedge$ (*snd p*) $\in$ *valid-plans PROB*}
x = *Inf* (*PLS-charles* (*fst p*) (*snd p*) *PROB*)
by *blast*
then have *fst p* $\in$ *valid-states PROB snd p* $\in$ *valid-plans PROB*
by *blast+*
then show *?thesis*
using *1*
by *fast*
qed

lemma *in-MPLS-charles-leq-2-pow-n*:
fixes *PROB* :: *'a problem* **and** *x*
assumes *finite PROB x* $\in$ *MPLS-charles PROB*
shows $x \leq 2$ ^ *card* (*prob-dom PROB*) − *1*
proof −
obtain *s as* **where** *1*:
s $\in$ *valid-states PROB as* $\in$ *valid-plans PROB x* = *Inf* (*PLS-charles s as PROB*)
using *assms*(*2*) *x-in-MPLS-charles-then*
by *blast*
then obtain *x'* **where** *2*: *x'* $\in$ *PLS-charles s as PROBx'* ≤ 2 ^ *card* (*prob-dom PROB*) − *1*

using *assms*(*1*) *in-PLS-charles-leq-2-pow-n*
by *blast*
then have $x \leq x'$
using *1*(*3*) *mem-le-imp-MIN-le*
by *blast*
then show *?thesis*
using *1 2*
by *linarith*
qed

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 −
have *1*: ∀ *x*∈*MPLS-charles PROB*. *x* ≤ *2* ^(*card* (*prob-dom PROB*)) − *1*
using *assms in-MPLS-charles-leq-2-pow-n*
by *blast*
then have *MPLS-charles PROB* ≠ {}
using *assms bound-main-lemma-charles-3*
by *blast*
then have *Sup* (*MPLS-charles PROB*) ≤ *2* ^(*card* (*prob-dom PROB*)) − *1*
using *1 bound-main-lemma-2*
by *meson*
then show *?thesis*
using *problem-plan-bound-charles-def*
by *metis*
qed

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 −
{
— NOTE type for 's' had to be fixed (type mismatch in first proof step.

fix *PROB* :: *'a problem* **and** *as* **and** *s* :: *'a state*
assume *P*:
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*)
)
then obtain *as'* **where** *1*:
(*exec-plan s as* = *exec-plan s as'*) (*subseq as' as*) (*length as'* < *f PROB*)
using *P*(*5*)
by *blast*
then have *2*: *as'* ∈ *valid-plans PROB*
using *P*(*3*) *sublist-valid-plan*
by *blast*
let *?x* = *length as'*
have *?x* ∈ *PLS-charles s as PROB*
unfolding *PLS-charles-def*
using *1 2*
by *auto*
then have ∃ *x*. *x* ∈ *PLS-charles s as PROB* ∧ *x* < *f PROB*
using *1 2*
by *blast*
}
then show *?thesis*
using *assms*
by *auto*
qed

— NOTE added lemma (refactored from ‘bound_on_all_plans_bounds_MPLS_charles‘).
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 −
{
fix *PROB* :: *'a problem* **and** *s as*
have *P PROB* ∧ *finite PROB* ∧ *as* ∈ *valid-plans PROB* ∧ *s* ∈ *valid-states PROB*
⟶ (∃ *x*. *x* ∈ *PLS-charles s as PROB* ∧ *x* < *f PROB*)

```
      using assms bound-on-all-plans-bounds-PLS-charles[of P f]
      by blast
    then have
      P PROB ∧ finite PROB ∧ as ∈ valid-plans PROB ∧ s ∈ valid-states PROB
      ⟶ Inf {n. n ∈ PLS-charles s as PROB} < f PROB

      using mem-lt-imp-MIN-lt CollectI
      by metis
  }
  then show ?thesis
    by blast
qed

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 −
  have 1: ∀ (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

    using assms bound-on-all-plans-bounds-MPLS-charles-i
    by blast
  moreover
  {
    fix PROB :: 'a problem and x
    assume P1: (P PROB) finite PROB x ∈ MPLS-charles PROB
    then obtain s as where a:
       as ∈ valid-plans PROB s ∈ valid-states PROB x = Inf (PLS-charles s as
PROB)
      using x-in-MPLS-charles-then
      by blast
    then have Inf {n. n ∈ PLS-charles s as PROB} < f PROB
      using 1 P1
      by blast
    then have x < f PROB
```

using *a*
by *simp*
}
ultimately show *?thesis*
by *blast*
qed

— NOTE added lemma (refactored from 'bound_on_all_plans_bounds_problem_plan_bound_charles').
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 −
have *1*: *MPLS-charles PROB* $\neq$ {}
using *assms*(*1*) *bound-main-lemma-charles-3*
by *auto*
then have *finite* (*MPLS-charles PROB*)
using *assms*(*2*) *finite-nat-set-iff-bounded*
by *blast*
then show *?thesis*
unfolding *Sup-nat-def*
using *1*
by *simp*
qed

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 −
have *1*: $\forall$ *PROB x*. *P PROB* $\wedge$ *finite PROB* $\longrightarrow$ *x* $\in$ *MPLS-charles PROB* $\longrightarrow$ *x* < *f PROB*
using *assms bound-on-all-plans-bounds-MPLS-charles*
by *blast*
moreover
{
fix *PROB*
assume *P*: *P PROB finite PROB*

```
  moreover have 2: ∀x. x ∈ MPLS-charles PROB ⟶ x < f PROB
    using 1 P
    by blast
  moreover
  {
    fix x
    assume P1: x ∈ MPLS-charles PROB
    moreover have x < f PROB
      using P(1, 2) P1 1
      by presburger
    moreover have MPLS-charles PROB ≠ {}
      using P1
      by blast
    moreover have Sup (MPLS-charles PROB) < f PROB
    using calculation(3) 2 bound-child-parent-not-eq-last-diff-paths[of MPLS-charles
PROB f PROB]
      by blast
    ultimately have (problem-plan-bound-charles PROB < f PROB)
      unfolding problem-plan-bound-charles-def
      by blast
  }
  moreover have Sup (MPLS-charles PROB) ∈ MPLS-charles PROB
    using P(2) 2 bound-on-all-plans-bounds-problem-plan-bound-charles-i
    by blast
  ultimately have problem-plan-bound-charles PROB < f PROB
    unfolding problem-plan-bound-charles-def
    by blast
}
ultimately show ?thesis
  by blast
qed
```

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 ≤ problem-plan-bound PROB
proof –
```

— NOTE obtain the premise needed for MP of 'bound_on_all_plans_bounds_problem_plan_bound_charles'.

{
fix *PROB* :: *'a problem* **and** *s* :: *'a state* **and** *as*
assume *P*: *finite PROB s* ∈ *valid-states PROB as* ∈ *valid-plans PROB*
then have ∃ *as'*.
exec-plan s as = *exec-plan s as'* ∧ *subseq as' as* ∧ *length as'* ≤ *problem-plan-bound PROB*

using *problem-plan-bound-works*
by *blast*
then have ∃ *as'*.
exec-plan s as = *exec-plan s as'* ∧ *subseq as' as* ∧ *length as'* < *problem-plan-bound PROB* + *1*

by *force*
}
then have *problem-plan-bound-charles PROB* < *problem-plan-bound PROB* + *1*
using *assms bound-on-all-plans-bounds-problem-plan-bound-charles*[**where** *f* = λ*PROB*. *problem-plan-bound PROB* + *1*
and *P* = λ-. *True*]
by *blast*
then show *?thesis*
by *simp*
qed

— NOTE added lemma (this was adapted from pred_setScript.sml:4887 with exlusion of the premise for the empty set since 'Max ' is undefined in Isabelle/HOL.)
lemma *MAX-SET-ELIM'*:
fixes *P Q*
assumes *finite P P* ≠ {} (∀ *x*. (∀ *y*. *y* ∈ *P* ⟶ *y* ≤ *x*) ∧ *x* ∈ *P* ⟶ *R x*)
shows *R* (*Max P*)
using *assms*
by *force*

— NOTE added lemma.
— NOTE adapted from pred_setScript.sml:4895 (premise 'finite P' was added).
lemma *MIN-SET-ELIM'*:
fixes *P Q*
assumes *finite P P* ≠ {} ∀ *x*. (∀ *y*. *y* ∈ *P* ⟶ *x* ≤ *y*) ∧ *x* ∈ *P* ⟶ *Q x*
shows *Q* (*Min P*)
proof −
let *?x=Min P*
have *Min P* ∈ *P*
using *Min-in*[*OF assms*(*1*) *assms*(*2*)]
by *simp*
moreover **{**
fix *y*

```
    assume P: y ∈ P
    then have ?x ≤ y
      using Min.coboundedI[OF assms(1)]
      by blast
    then have Q ?x using P assms
      by auto
  }
  ultimately show ?thesis
    by blast
qed

— NOTE added lemma (refactored from ‘RD_bounds_sublistD‘).
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 −
  {
    let ?ss={length p − 1 |p. valid-path Pi p ∧ distinct p}
    let ?ss'={p. valid-path Pi p ∧ distinct p}
    have 1: ?ss = (λx. length x − 1) ‘ ?ss'
      by blast
    {
    — NOTE type of ‘valid_states Pi‘ had to be asserted to match ‘FINITE_valid_states‘.
      let ?S={p. distinct p ∧ set p ⊆ (valid-states Pi :: 'a state set)}
      {
        from assms have finite (valid-states Pi :: 'a state set)
          using FINITE-valid-states[of Pi]
          by simp
        then have finite ?S
          using FINITE-ALL-DISTINCT-LISTS
          by blast
      }
      moreover {
        {
          fix x
          assume x ∈ ?ss'
          then have x ∈ ?S
          proof (induction x)
            case (Cons a x)
            then have a: valid-path Pi (a # x) distinct (a # x)
              by blast+
            moreover {
              fix x'
              assume P: x' ∈ set (a # x)
              then have x' ∈ valid-states Pi
              proof (cases x)
                case Nil
                from a(1) Nil
```

```
            have a ∈ valid-states Pi
              by simp
            moreover from P Nil
            have x′ = a
              by force
            ultimately show ?thesis
              by simp
          next
            case (Cons a′ list)
            {
              {
                from Cons.prems have valid-path Pi (a # x)
                  by simp
                then have a ∈ valid-states Pi valid-path Pi (a′ # list)
                  using Cons
                  by fastforce+
              }
              note a = this
              moreover {
                from Cons.prems have distinct (a # x)
                  by blast
                then have distinct (a′ # list)
                  using Cons
                  by simp
              }
              ultimately
              have (a′ # list) ∈ ?ss′
                by blast
              then have (a′ # list) ∈ ?S
                using Cons Cons.IH
                by argo
            }
            then show ?thesis
              using P a(1) local.Cons set-ConsD
              by fastforce
          qed
        }
        ultimately show ?case
          by blast
      qed simp
    }
    then have ?ss′ ⊆ ?S
      by blast
  }
  ultimately have finite ?ss′
    using rev-finite-subset
    by auto
}
note 2 = this
```

from *1 2* **have** *finite ?ss*
  using *finite-imageI*
  by *auto*
}
then show *?thesis*
  by *blast*
qed

— NOTE added lemma (refactored from ‘RD_bounds_sublistD‘).
lemma *RD-bounds-sublistD-i-b*:
  fixes *Pi* :: *'a problem*
  shows {*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*} ≠ {}
proof −
  let *?Q*={*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
  let *?Q'*={*p. valid-path Pi p* ∧ *distinct p*}
  {
    have *valid-path Pi* []
      by *simp*
    moreover have *distinct* []
      by *simp*
    ultimately have [] ∈ *?Q'*
      by *simp*
  }
  note *1* = *this*
  have *?Q* = (λ*p. length p* − *1*) ‘ *?Q'*
    by *blast*
  then have *length* [] − *1* ∈ *?Q*
    using *1*
    by (*metis* (*mono-tags*, *lifting*) *image-iff list.size*(*3*))
  then show *?thesis*
    by *blast*
qed

— NOTE added lemma (refactored from ‘RD_bounds_sublistD‘).
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 −
  let *?P*=(*PLS s as*)
  let *?Q*={*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
  from *assms*(*4*) **obtain** *p* **where** *1*:
    x = *length p* − *1 valid-path Pi p distinct p*
    by *blast*
  {
    fix *p'*

assume *valid-path Pi p′ distinct p′*
then obtain *y* **where** $y \in ?Q$ $y = length\ p' - 1$
by *blast*
— NOTE we cannot infer $length\ p' - 1 \leq length\ p - 1$ since 'length p' = 0' might be true.
then have *a*: $length\ p' - 1 \leq length\ p - 1$
using *assms*(*3*) *1*(*1*)
by *meson*
}
note *2* = *this*
{
from *finite-PLS PLS-NEMPTY*
have *finite* (*PLS s as*) $PLS\ s\ as \neq \{\}$
by *blast+*
moreover {
fix *n*
assume *P*: $(\forall y.\ y \in PLS\ s\ as \longrightarrow n \leq y)$ $n \in PLS\ s\ as$
from *P*(*2*) **obtain** *as′* **where** *i*:
n = *length as′ exec-plan s as′* = *exec-plan s as subseq as′ as*
unfolding *PLS-def*
by *blast*
let *?p′*=*statelist′ s as′*
{
have *length as′* = *length ?p′* − *1*
by (*simp add*: *LENGTH-statelist′*)
— MARKER (topologicalPropsScript.sml:195)
have *1* + (*length p* − *1*) = *length p* − *1* + *1*
by *presburger*
— MARKER (topologicalPropsScript.sml:200)
{
from *assms*(*2*) *i*(*3*) *sublist-valid-plan*
have $as' \in$ *valid-plans Pi*
by *blast*
then have *valid-path Pi ?p′*
using *assms*(*1*) *valid-path-statelist′*
by *auto*
}
moreover {
{
assume *C*: ¬*distinct ?p′*
— NOTE renamed variable 'drop' to 'drop'' to avoid shadowing of the function by the same name in Isabelle/HOL.
then obtain *rs pfx drop′ tail* **where** *C-1*: *?p′* = *pfx* @ [*rs*] @ *drop′* @ [*rs*] @ *tail*
using *not-distinct-decomp*[*OF C*]
by *fast*
let *?pfxn*=*length pfx*
have *C-2*: *?p′* ! *?pfxn* = *rs*
by (*simp add*: *C-1*)

from *LENGTH-statelist′*
have *C-3*: *length as′ + 1 = length ?p′*
by *metis*
then have *?pfxn ≤ length as′*
using *C-1*
by *fastforce*
then have *C-4*: *exec-plan s* (*take ?pfxn as′*) = *rs*
using *C-2 statelist′-TAKE*
by *blast*
let *?prsd = length* (*pfx* @ [*rs*] @ *drop′*)
let *?ap1 = take ?pfxn as′*
— MARKER (topologicalPropsScript.sml:215)
from *C-1*
have *C-5*: *?p′ ! ?prsd = rs*
by (*metis append-Cons length-append nth-append-length nth-append-length-plus*)
from *C-1 C-3*
have *C-6*: *?prsd ≤ length as′*
by *simp*
then have *C-7*: *exec-plan s* (*take ?prsd as′*) = *rs*
using *C-5 statelist′-TAKE*
by *auto*
let *?ap2=take ?prsd as′*
let *?asfx=drop ?prsd as′*
have *C-8*: *as′ = ?ap2* @ *?asfx*
by *force*
then have *exec-plan s as′ = exec-plan* (*exec-plan s ?ap2*) *?asfx*
using *exec-plan-Append*
by *metis*
then have *C-9*: *exec-plan s as′ = exec-plan s* (*?ap1* @ *?asfx*)
using *C-4 C-7 exec-plan-Append*
by *metis*
from *C-6*
have *C-10*: (*length ?ap1 = ?pfxn*) ∧ (*length ?ap2 = ?prsd*)
by *fastforce*
then have *C-11*: *length* (*?ap1* @ *?asfx*) < *length* (*?ap2* @ *?asfx*)
by *auto*
{
from *C-10*
have *?pfxn + length ?asfx = length* (*?ap1* @ *?asfx*)
by *simp*
from *C-9 i*(*2*)
have *C-12*: *exec-plan s* (*?ap1* @ *?asfx*) = *exec-plan s as*
by *argo*
{
{
{
have *prefix ?ap1 ?ap2*
by (*metis* (*no-types*) *length-append prefix-def take-add*)
then have *subseq ?ap1 ?ap2*

```
              using isPREFIX-sublist
              by blast
            }
            moreover have sublist ?asfx ?asfx
              using sublist-refl
              by blast
            ultimately have subseq (?ap1 @ ?asfx) as'
              using C-8 subseq-append
              by metis
          }
          moreover from i(3)
          have subseq as' as
            by simp
          ultimately have subseq (?ap1 @ ?asfx) as
            using sublist-trans
            by blast
        }
        then have length (?ap1 @ ?asfx) ∈ PLS s as
          unfolding PLS-def
          using C-12
          by blast
      }
      then have False
        using P(1) i(1) C-10
        by auto
    }
    hence distinct ?p'
      by auto
  }
  ultimately have length ?p' − 1 ≤ length p − 1
    using 2
    by blast
}
note ii = this
{
  from i(1) have n + 1 = length ?p'
    using LENGTH-statelist'[symmetric]
    by blast
  also have ... ≤ 1 + (length p − 1)
    using ii
    by linarith
  finally have n ≤ length p − 1
    by fastforce
}
then have n ≤ length p − 1
  by blast
}
ultimately have Min ?P ≤ length p − 1
  using MIN-SET-ELIM'[where P=?P and Q=λx. x ≤ length p − 1]
```

by *blast*
}
note *3 = this*
{
have *length p − 1 ≤ Max {length p − 1 |p. valid-path Pi p ∧ distinct p}*
using *assms(3, 4) 1(1)*
by *(smt Max.coboundedI bdd-aboveI bdd-above-nat)*
moreover
have *Min (PLS s as) ≤ length p − 1*
using *3*
by *blast*
ultimately
have *Min (PLS s as) ≤ Max {length p − 1 |p. valid-path Pi p ∧ distinct p}*
by *linarith*
}
then show *?thesis*
by *blast*
qed

— NOTE added lemma (refactored from 'RD_bounds_sublistD').
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 −
{
let *?P=MPLS Pi*
let *?Q={length p − 1 |p. valid-path Pi p ∧ distinct p}*
from *assms(3)*
obtain *s as* **where** *1*:
s ∈ valid-states Pi as ∈ valid-plans Pi x = Inf (PLS s as)
unfolding *MPLS-def*
by *fast*
have *x ≤ Max ?Q* **proof** −

Show that 'x' is not only the infimum but also the minimum of 'PLS s as'.

{
have *finite (PLS s as)*
using *finite-PLS*
by *auto*
moreover
have *PLS s as ≠ {}*
using *PLS-NEMPTY*
by *auto*
ultimately
have *a: Inf (PLS s as) = Min (PLS s as)*
using *cInf-eq-Min[of PLS s as]*
by *blast*

from *1*(*3*) *a* **have** $x = Min\ (PLS\ s\ as)$
by *blast*
}
note $a = this$
{
let *?limit=Min* (*PLS s as*)
from *assms*(*1*)
have *a*: *finite ?Q*
using *RD-bounds-sublistD-i-a*
by *blast*
have *b*: $?Q \neq \{\}$
using *RD-bounds-sublistD-i-b*
by *fast*
from *1*(*1*, *2*)
have *c*: $\forall x.\ (\forall y.\ y \in ?Q \longrightarrow y \leq x) \wedge x \in ?Q \longrightarrow ?limit \leq Max\ ?Q$
using *RD-bounds-sublistD-i-c*
by *blast*
have $?limit \leq Max\ ?Q$
using *MAX-SET-ELIM'*[**where** *P=?Q* **and** $R{=}\lambda x.\ ?limit \leq Max\ ?Q$, *OF a b c*]
by *blast*
}
note $b = this$
from *a b* **show** $x \leq Max\ ?Q$
by *blast*
qed
}
then show *?thesis*
using *assms*
unfolding *MPLS-def*
by *blast*
qed

— NOTE type of ‘Pi‘ had to be fixed for use of ‘FINITE_valid_states‘.
lemma *RD-bounds-sublistD*:
fixes *Pi* :: *'a problem*
assumes *finite Pi*
shows *problem-plan-bound Pi* $\leq$ *RD Pi*
proof −
let *?P=MPLS Pi*
let *?Q*={*length p* − *1* |*p. valid-path Pi p* ∧ *distinct p*}
{
from *assms*
have *1*: *finite ?P*
using *FINITE-MPLS*
by *blast*
from *assms*
have *2*: $?P \neq \{\}$
using *MPLS-nempty*

by *blast*
from *assms*
have *3*: $\forall x.\ (\forall y.\ y \in ?P \longrightarrow y \leq x) \land x \in ?P \longrightarrow x \leq Max\ ?Q$
using *RD-bounds-sublistD-i*
by *blast*
have $Max\ ?P \leq Max\ ?Q$
using *MAX-SET-ELIM'*[*OF 1 2 3*]
by *blast*
}
then show *?thesis*
unfolding *problem-plan-bound-def RD-def Sup-nat-def*
using *RD-bounds-sublistD-i-b* **by** *auto*
qed

— NOTE type for ‘PROB‘ had to be fixed in order to be able to match ‘sublistD_bounds_D‘.
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*

using *assms sublistD-bounds-D RD-bounds-sublistD*
by *auto*

— NOTE type of ‘PROB‘ had to be fixed for MP of lemmas.
lemma *empty-problem-bound*:
fixes *PROB* :: *'a problem*
assumes (*prob-dom PROB* = {})
shows (*problem-plan-bound PROB* = *0*)
proof −
{
fix *PROB'* **and** *as* :: ((*'a*, *'b*) *fmap* × (*'a*, *'b*) *fmap*) *list* **and** *s* :: (*'a*, *'b*) *fmap*
assume
finite PROB prob-dom PROB' = {} *s* ∈ *valid-states PROB' as* ∈ *valid-plans PROB'*
then have *exec-plan s* [] = *exec-plan s as*
using *empty-prob-dom-imp-empty-plan-always-good*
by *blast*
then have (∃ *as'*. *exec-plan s as* = *exec-plan s as'* ∧ *subseq as' as* ∧ *length as'* < *1*)
by *force*
}
then show *?thesis*
using *bound-on-all-plans-bounds-problem-plan-bound-*[**where** *P*=λ*P*. *prob-dom P* = {} **and** *f*=λ*P*. *1*, *of PROB*]

using *assms empty-prob-dom-finite*
by *blast*
qed

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*)
∧ (*length as'* ≤ *problem-plan-bound PROB*)
∧ (*sat-precond-as s as'*)
)
proof −
obtain *as'* **where** *1*:
exec-plan s as = *exec-plan s as' subseq as' as length as'* ≤ *problem-plan-bound PROB*
using *assms problem-plan-bound-works*
by *blast*
— NOTE this step seems to be handled implicitely in original proof.
moreover have *rem-condless-act s* [] *as'* ∈ *valid-plans PROB*
using *assms*(*3*) *1*(*2*) *rem-condless-valid-10 sublist-valid-plan*
by *blast*
moreover have *subseq* (*rem-condless-act s* [] *as'*) *as'*
using *rem-condless-valid-8*
by *blast*
moreover have *length* (*rem-condless-act s* [] *as'*) ≤ *length as'*
using *rem-condless-valid-3*
by *blast*
moreover have *sat-precond-as s* (*rem-condless-act s* [] *as'*)
using *rem-condless-valid-2*
by *blast*
moreover have *exec-plan s as'* = *exec-plan s* (*rem-condless-act s* [] *as'*)
using *rem-condless-valid-1*
by *blast*
ultimately show *?thesis*
by *fastforce*
qed

— TODO remove? Can be solved directly with 'TopologicalProps.bound_on_all_plans_bounds_problem_plan_bound_thesis'.
lemma *problem-plan-bound-UBound*:
assumes (∀ *as s*.
(*s* ∈ *valid-states PROB*)
∧ (*as* ∈ *valid-plans PROB*)
⟶ (∃ *as'*.
(*exec-plan s as* = *exec-plan s as'*)

∧ *subseq as′ as*
∧ (*length as′* < *f PROB*)
)
) *finite PROB*
shows (*problem-plan-bound PROB* < *f PROB*)
proof −
let *?P* = *λPr. PROB* = *Pr*
have *?P PROB* **by** *simp*
then show *?thesis*
using *assms bound-on-all-plans-bounds-problem-plan-bound-*[**where** *P* = *?P*]
by *force*
qed

6.4 Traversal Diameter

definition *traversed-states* **where**
traversed-states s as ≡ *set* (*state-list s as*)

lemma *finite-traversed-states*: *finite* (*traversed-states s as*)
unfolding *traversed-states-def*
by *simp*

lemma *traversed-states-nempty*: *traversed-states s as* ≠ {}
unfolding *traversed-states-def*
by (*induction as*) *auto*

lemma *traversed-states-geq-1*:
fixes *s*
shows *1* ≤ *card* (*traversed-states s as*)
proof −
have *card* (*traversed-states s as*) ≠ *0*
using *traversed-states-nempty finite-traversed-states card-0-eq*
by *blast*
then show *1* ≤ *card* (*traversed-states s as*)
by *linarith*
qed

lemma *init-is-traversed*: *s* ∈ *traversed-states s as*
unfolding *traversed-states-def*
by (*induction as*) *auto*

— NOTE name shortened.
definition *td* **where**
td PROB ≡ *Sup* {

(*card* (*traversed-states* (*fst p*) (*snd p*))) − *1*
| *p*. (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)}

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

apply(*induction as*)
apply(*auto simp add*: *traversed-states-def rem-condless-act-cons*)
subgoal by (*simp add*: *state-succ-pair*)
subgoal using *init-is-traversed traversed-states-def* **by** *blast*
subgoal by (*simp add*: *state-succ-pair*)
done

— NOTE added lemma.
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 −
let *?S*={*p*. (*fst p* ∈ *valid-states PROB*) ∧ (*snd p* ∈ *valid-plans PROB*)}
obtain *s* :: ′*a state* **where** *s* ∈ *valid-states PROB*
using *assms valid-states-nempty*
by *blast*
moreover have [] ∈ *valid-plans PROB*
using *empty-plan-is-valid*
by *auto*
ultimately have *?S* ≠ {}
using *assms valid-states-nempty*
by *auto*
then show *?thesis*
by *blast*
qed

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 −
let *?S*={

```
    (card (traversed-states (fst p) (snd p))) − 1
    | p. (fst p ∈ valid-states PROB) ∧ (snd p ∈ valid-plans PROB)}

  {
    fix x
    assume x ∈ ?S
    then obtain p where 1:
      x = card (traversed-states (fst p) (snd p)) − 1 fst p ∈ valid-states PROB
      snd p ∈ valid-plans PROB
      by blast
    let ?s=fst p
    let ?as=snd p
    {
      let ?as'=(rem-condless-act ?s [] ?as)
      have 2: traversed-states ?s ?as = traversed-states ?s ?as'
        using traversed-states-rem-condless-act
        by blast
      moreover have sat-precond-as ?s ?as'
        using rem-condless-valid-2
        by blast
      moreover have ?as' ∈ valid-plans PROB
        using 1(3) rem-condless-valid-10
        by blast
      ultimately have card (traversed-states ?s ?as') ≤ k
        using assms(2) 1(2)
        by blast
      then have card (traversed-states ?s ?as) ≤ k
        using 2
        by argo
    }
    then have x ≤ k − 1
      using 1
      by linarith
  }
  moreover have ?S ≠ {}
    using assms td-UBound-i
    by fast
  ultimately show ?thesis
    unfolding td-def
    using td-UBound-i bound-main-lemma-2[of ?S k − 1]
    by presburger
qed


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*)
unfolding *action-proj-def*
by *simp*

definition *prob-proj* **where**
prob-proj PROB vs ≡ (*λa. 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* = []
by (*simp*)+

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 −
have
fmrestrict-set vs (*if fst a* $\subseteq_f$ *s then snd a* ++ *s else s*)
= *fmrestrict-set vs* (*snd a* ++ *s*)

using *assms*
by *simp*
moreover
{
assume *fst* (*action-proj a vs*) $\subseteq_f$ *fmrestrict-set vs s*
then have
(*state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*)
= *fmrestrict-set vs* (*snd a* ++ *s*))

unfolding *state-succ-def action-proj-def fmap-add-ltr-def*
by *force*
}
moreover {
assume ¬(*fst* (*action-proj a vs*) $\subseteq_f$ *fmrestrict-set vs s*)
then have
(*state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*)
= *fmrestrict-set vs* (*snd a* ++ *s*))

unfolding *state-succ-def action-proj-def*
using *assms fmsubset-restrict-set-mono*
by *auto*
}
ultimately show *?thesis*
unfolding *state-succ-def*
by *argo*

qed

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*)))
using *assms*
proof (*induction as arbitrary*: *s vs*)
case (*Cons a as*)
then show *?case*
proof (*cases fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {})
case *True*
then have
state-succ (*fmrestrict-set vs s*) (*action-proj a vs*) = *fmrestrict-set vs* (*state-succ s a*)
using *Cons.prems proj-state-succ*
by *fastforce*
then show *?thesis*
unfolding *exec-plan.simps sat-precond-as.simps as-proj.simps*
using *Cons.IH Cons.prems True*
by *simp*
next
case *False*
then have (*fmdom′* (*snd a*) ∩ *vs* = {})
using *False fmdom′-restrict-set-precise*[*of vs snd a*]
by *argo*
then have *fmrestrict-set vs s* = *fmrestrict-set vs* (*state-succ s a*)
using *disj-imp-eq-proj-exec*
by *blast*
then show *?thesis*
unfolding *exec-plan.simps sat-precond-as.simps as-proj.simps*
using *Cons.IH Cons.prems False*
by *simp*
qed
qed *simp*

— TODO the proofs are inefficient (detailed proofs?).
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*)

unfolding *action-dom-def action-proj-def*
by (*auto simp*: *fmdom′-restrict-set-precise*)

lemma *graph-plan-neq-mems-state-set-neq-len*:
  shows *prob-dom* (*prob-proj PROB vs*) = (*prob-dom PROB* ∩ *vs*)
proof –
  have
      prob-dom (*prob-proj PROB vs*)
      = (
        ⋃(*s1*, *s2*)∈(λ*a*. (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*)))
        ‘ *PROB*. *action-dom s1 s2*
)

    unfolding *prob-dom-def prob-proj-def action-proj-def*
    by *blast*
  moreover
  {
    have
    (*prob-dom PROB* ∩ *vs*)
    = (⋃ *a*∈*PROB*. *action-dom* (*fst a*) (*snd a*) ∩ *vs*)

      unfolding *prob-dom-def prob-proj-def*
      using *SUP-cong*
      by *auto*
    also have . . . = (⋃ *a*∈*PROB*. *action-dom* (*fst* (*action-proj a vs*)) (*snd* (*action-proj a vs*)))
      using *proj-action-dom-eq-inter*[*symmetric*]
      by *fast*
    finally have
      (*prob-dom PROB* ∩ *vs*)
      = (⋃ *a*∈*PROB*. *fmdom′* (*fmrestrict-set vs* (*fst a*)) ∪ *fmdom′* (*fmrestrict-set vs* (*snd a*)))

      unfolding *action-dom-def action-proj-def*
      by *simp*
  }
  ultimately show *?thesis*
    by (*metis* (*mono-tags*, *lifting*) *SUP-cong UN-simps*(*10*) *action-dom-def case-prod-beta′ prod.sel*(*1*)
        snd-conv)
qed

— TODO more detailed 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*))

  unfolding *valid-states-def*
  using *graph-plan-neq-mems-state-set-neq-len*
  by (*metis* (*mono-tags*, *lifting*)

assms fmdom'.rep-eq fmlookup-fmrestrict-set-dom inf-commute mem-Collect-eq valid-states-def)

lemma *dom-eff-subset-imp-dom-succ-eq-proj*:
fixes *h s vs*
assumes (*fmdom'* (*snd h*) $\subseteq$ *fmdom' s*)
shows (*fmdom'* (*state-succ s* (*action-proj h vs*)) = *fmdom'* (*state-succ s h*))
proof (*cases fst* (*fmrestrict-set vs* (*fst h*), *fmrestrict-set vs* (*snd h*)) $\subseteq_f$ *s*)
case *true*: *True*
then show *?thesis*
proof (*cases fst h* $\subseteq_f$ *s*)
case *True*
then show *?thesis*
unfolding *state-succ-def action-proj-def*
using *true True*
by *simp* (*smt assms fmap-add-ltr-def fmdom'.rep-eq fmdom'-add fmlookup-fmrestrict-set-dom inf.absorb-iff2 inf.left-commute sup.absorb-iff1*)
next
case *False*
then show *?thesis*
unfolding *state-succ-def action-proj-def*
using *true False*
by *simp* (*metis* (*no-types*) *assms dual-order.trans fmap-add-ltr-def fmdom'.rep-eq fmdom'-add fmlookup-fmrestrict-set-dom inf-le2 sup.absorb-iff1*)
qed
next
case *False*
then have *fmdom' s* = *fmdom'* (*if fst h* $\subseteq_f$ *s then snd h* ++ *s else s*)
using *sat-precond-as-proj-4*
by *auto*
then show *?thesis*
unfolding *state-succ-def action-proj-def*
using *False*
by *presburger*
qed

lemma *drest-proj-succ-eq-drest-succ*:
fixes *h s vs*
assumes *fst h* $\subseteq_f$ *s* (*fmdom'* (*snd h*) $\subseteq$ *fmdom' s*)
shows (*fmrestrict-set vs* (*state-succ s* (*action-proj h vs*)) = *fmrestrict-set vs* (*state-succ s h*))
proof –
{
have *1*: *fmrestrict-set vs* (*fst h*) $\subseteq_f$ *s*
using *assms*(*1*) *submap-imp-state-succ-submap-a*
by (*simp add*: *sat-precond-as-proj-4*)

then have
fmrestrict-set vs (*state-succ s* (*action-proj h vs*))
= *fmrestrict-set vs* (*fmrestrict-set vs* (*snd h*) ++ *s*)

unfolding *state-succ-def action-proj-def*
by *simp*
also have ... = *fmrestrict-set vs s* $++_f$ *fmrestrict-set vs* (*fmrestrict-set vs* (*snd h*))
unfolding *fmap-add-ltr-def*
by *simp*
— TODO refactor the step 'fmrestrict_set ?X (fmrestrict_set ?X ?f) = fmrestrict_set ?X ?f' into own lemma in 'FmapUtils.thy'.
also have ... = *fmrestrict-set vs s* $++_f$ *fmrestrict-set vs* (*snd h*)
using *fmfilter-alt-defs*(*4*) *fmfilter-cong fmlookup-filter fmrestrict-set-dom option.simps*(*3*)
by *metis*
finally have
fmrestrict-set vs (*state-succ s* (*action-proj h vs*))
= *fmrestrict-set vs* (*snd h* ++ *s*)

unfolding *fmap-add-ltr-def*
by *simp*
}
moreover have *fmrestrict-set vs* (*state-succ s h*) = *fmrestrict-set vs* ((*snd h*) ++ *s*)
unfolding *state-succ-def*
using *assms*(*1*)
by *simp*
ultimately show *?thesis*
by *simp*
qed

— TODO remove? This is equivalent to 'proj_state_succ'.
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*))
using *assms proj-state-succ*
by *blast*

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 –
 have *exec-plan* (*state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*)) *as* =
 exec-plan (*fmrestrict-set vs* (*state-succ s a*)) *as*
  using *assms drest-succ-proj-eq-drest-succ*
  by *metis*
 then show *?thesis*
  unfolding *prob-proj-def*
  by *simp*
qed

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*)
)
 using *assms proj-state-succ*
 by *fastforce*

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*))
 unfolding *action-proj-def as-proj.simps*
 using *assms*
 by *argo*

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*)
 unfolding *action-proj-def*
 using *assms*
 by *simp*

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 –
 have *fmdom'* (*snd* (*action-proj a vs*)) = {}
  unfolding *action-proj-def*
  using *assms*
  by *simp*

then have *state-succ s* (*action-proj a vs*) = *s*
using *empty-eff-exec-eq*
by *fast*
then show *?thesis*
by *simp*
qed

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*))
using *assms*
proof (*induction as*)
case (*Cons a as*)
then show *?case*
using *Cons.prems graph-plan-lemma-1*
by *blast*
qed *auto*

lemma *action-proj-in-prob-proj*:
assumes (*a* ∈ *PROB*)
shows (*action-proj a vs* ∈ *prob-proj PROB vs*)
unfolding *action-proj-def prob-proj-def*
using *assms*
by *simp*

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*))
using *assms*
proof (*induction as arbitrary*: *PROB vs*)
case (*Cons a as*)
then show *?case*
using *assms Cons*
proof(*cases fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {})
case *True*
then have *1*: *as-proj* (*a* # *as*) *vs* = *action-proj a vs* # *as-proj as vs*
using *True*
by *simp*
then have *as* ∈ *valid-plans PROB*
using *Cons.prems valid-plan-valid-tail*
by *fast*
then have *as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*)
using *Cons.IH 1*

by *simp*
then have *action-proj a vs # as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*)
using *Cons.prems action-proj-in-prob-proj valid-head-and-tail-valid-plan valid-plan-valid-head*
by *metis*
then show *?thesis*
using *1*
by *argo*
next
case *False*
then have *as-proj* (*a* # *as*) *vs* = *as-proj as vs*
using *False*
by *auto*
then have *as-proj* (*a* # *as*) *vs* ∈ *valid-plans* (*prob-proj PROB vs*)
using *assms Cons valid-plan-valid-tail*
by *metis*
then show *?thesis*
using *assms Cons.IH*(*1*)
by *blast*
qed
qed (*simp add*: *valid-plans-def*)

lemma *finite-imp-finite-prob-proj*:
fixes *PROB*
assumes *finite PROB*
shows (*finite* (*prob-proj PROB vs*))
unfolding *prob-proj-def*
using *assms*
by *simp*

— NOTE Base 2 in 5th assumption had to be explicitely fixed to 'nat' type to be able to use the linearity lemma for powers of natural numbers.
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 −
{
have *card* (*fmdom'* (*fmrestrict-set vs s*)) ≤ *card vs*
using *assms*(*4*) *graph-plan-card-state-set*
by *fast*
then have (*2* :: *nat*) ^(*card* (*fmdom'* (*fmrestrict-set vs s*))) − *1* ≤ *2* ^(*card*

vs) − *1*
      using *power-increasing diff-le-mono*
      by *force*
    also have ... < *length* (*as-proj as vs*)
      using *assms*(*5*)
      by *blast*
    finally have *2* ^ *card* (*fmdom'* (*fmrestrict-set vs s*)) − *1* < *length* (*as-proj as vs*)
      by *blast*
  }
  note *1* = *this*
  moreover have *fmrestrict-set vs s* ∈ *valid-states* (*prob-proj PROB vs*)
    using *assms*(*2*) *graph-plan-not-eq-last-diff-paths*
    by *blast*
  moreover have *as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*)
    using *assms*(*3*) *valid-as-valid-as-proj*
    by *blast*
  moreover have *finite* (*prob-proj PROB vs*)
    using *assms*(*1*) *finite-imp-finite-prob-proj*
    by *blast*
  ultimately show *?thesis*
    using *lemma-2*[**where** *PROB=prob-proj PROB vs* **and** *as=as-proj as vs* **and** *s=fmrestrict-set vs s*]
    by *blast*
qed

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*)
  by (*induction as*) (*auto simp add*: *action-proj-def*)

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*)
)
  using *assms append-eq-as-proj-1 as-proj-eq-filter-action-proj*
  by (*metis* (*no-types*, *lifting*))

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 −
let *?lhs* = *state-succ* (*fmrestrict-set vs s*) (*action-proj a vs*)
let *?rhs* = *fmrestrict-set vs* (*state-succ s* (*action-proj a vs*))
— NOTE Show lhs and rhs equality by splitting on the cases introduced by the if-then branching of 'state_succ'.
{
assume *P1*: *fst* (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*)) $\subseteq_f$ *fmrestrict-set vs s*
then have *a*: *fst* (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*)) $\subseteq_f$ *s*
using *drest-smap-drest-smap-drest*
by *auto*
then have *?lhs* = *fmrestrict-set vs* (*snd a*) ++ *fmrestrict-set vs s*
unfolding *state-succ-def action-proj-def*
using *P1*
by *simp*
moreover **{**
have *rhs*: *?rhs* = *fmrestrict-set vs* (*fmrestrict-set vs* (*snd a*) ++ *s*)
unfolding *state-succ-def action-proj-def*
using *a*
by *auto*
also have . . . = (*fmrestrict-set vs* (*fmrestrict-set vs* (*snd a*)) ++ *fmrestrict-set vs s*)
unfolding *fmap-add-ltr-def*
by *simp*
finally have *?rhs* = (*fmrestrict-set vs* (*snd a*) ++ *fmrestrict-set vs s*)
unfolding *fmfilter-alt-defs*(*4*)
by *fastforce*
}
ultimately have *?lhs* = *?rhs*
by *argo*
}
moreover **{**
assume *P2*: ¬(*fst* (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*)) $\subseteq_f$ *fmrestrict-set vs s*)
then have *a*: ¬(*fst* (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*)) $\subseteq_f$ *s*)
using *drest-smap-drest-smap-drest*
by *auto*
then have *?lhs* = *fmrestrict-set vs s*
unfolding *state-succ-def action-proj-def*
using *P2*
by *argo*
moreover have *?rhs* = *fmrestrict-set vs s*
unfolding *state-succ-def action-proj-def*
using *a*
by *presburger*
ultimately have *?lhs* = *?rhs*

by *simp*
}
ultimately show *?lhs* = *?rhs*
by *blast*
qed

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 (*induction as arbitrary*: *s vs*)
case (*Cons a as*)
then show *?case*
by (*simp add*: *succ-drest-eq-drest-succ*)
qed (*simp add*: *fmfilter-alt-defs*(*4*))

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 (*induction as arbitrary*: *s vs*)
case (*Cons a as*)
then show *?case*
by (*simp add*: *succ-drest-eq-drest-succ*)
qed (*simp add*: *fmfilter-alt-defs*(*4*))

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*)

by (*metis proj-exec-proj-eq-exec-proj' proj-exec-proj-eq-exec-proj*)

lemma *act-dom-proj-eff-subset-act-dom-eff*:
fixes *a vs*
shows *fmdom'* (*snd* (*action-proj a vs*)) ⊆ *fmdom'* (*snd a*)
proof −
have *snd* (*action-proj a vs*) = *fmrestrict-set vs* (*snd a*)
unfolding *action-proj-def*
by *simp*

then have *fmlookup* (*fmrestrict-set vs* (*snd a*)) $\subseteq_m$ *fmlookup* (*snd a*)
by (*simp add*: *map-le-def fmdom′-restrict-set-precise*)
then have *dom* (*fmlookup* (*fmrestrict-set vs* (*snd a*))) ⊆ *dom* (*fmlookup* (*snd a*))
using *map-le-implies-dom-le*
by *blast*
then have *fmdom′* (*fmrestrict-set vs* (*snd a*)) ⊆ *fmdom′* (*snd a*)
using *fmdom′.rep-eq*
by *metis*
then show *?thesis*
unfolding *action-proj-def*
by *simp*
qed

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*)
using *assms*
proof (*induction as arbitrary*: *s PROB vs*)
case (*Cons a as*)
then have *1*: *as* ∈ *valid-plans PROB*
using *Cons.prems*(*2*) *valid-plan-valid-tail*
by *fast*
then have *2*: *exec-plan s* (*as-proj as vs*) ∈ *valid-states PROB*
using *Cons.prems*(*1*) *Cons.IH*(*1*)
by *blast*
— NOTE split on the if-then branch introduced by 'as_proj'.
moreover {
assume *P*: *fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {}
then have
exec-plan s (*as-proj* (*a* # *as*) *vs*)
= *exec-plan* (*state-succ s* (*action-proj a vs*)) (*as-proj as vs*)

by *simp*
— NOTE split on the if-then branch introduced by 'state_succ'
moreover
{
assume *fst* (*action-proj a vs*) $\subseteq_f$ *s*
then have *3*:
exec-plan (*state-succ s* (*action-proj a vs*)) (*as-proj as vs*)
= *exec-plan* (*snd* (*action-proj a vs*) ++ *s*) (*as-proj as vs*)

unfolding *state-succ-def*
using *calculation*
by *simp*
{
— TODO Unsure why this proof step is necessary at all, but it should be refactored into a dedicated lemma *s* ∈ *valid-states PROB* ⟹ *fmdom′ s* = *prob-dom*

PROB.
{
have *s* ∈ *valid-states PROB*
using *Cons.prems*
by *simp*
then have *s* ∈ {*s*′. *fmdom*′ *s*′ = *prob-dom PROB*}
unfolding *valid-states-def*
by *simp*
then obtain *s*′ **where** *s*′ = *s fmdom*′ *s*′ = *prob-dom PROB*
by *auto*
then have *fmdom*′ *s* = *prob-dom PROB*
by *simp*
}
— TODO Refactor this step ('also ...' for subset chain; replace fact 'fmdom' s = prob_dom PROB' in last step with MP step from lemma refactored above.
moreover {
have (*snd* (*action-proj a vs*) ++ *s*) = (*s* $++_f$ *fmrestrict-set vs* (*snd a*))
unfolding *action-proj-def fmap-add-ltr-def*
by *simp*
then have *a*: *a* ∈ *PROB*
using *Cons.prems*(*2*) *valid-plan-valid-head*
by *fast*
then have *action-dom* (*fst a*) (*snd a*) ⊆ *prob-dom PROB*
using *exec-as-proj-valid-2*
by *blast*
then have *fmdom*′ (*snd a*) ⊆ *action-dom* (*fst a*) (*snd a*)
unfolding *action-dom-def*
by *simp*
then have *fmdom*′ (*fmrestrict-set vs* (*snd a*)) ⊆ *fmdom*′ (*snd a*)
using *action-proj-def act-dom-proj-eff-subset-act-dom-eff snd-conv*
by *metis*
then have *fmdom*′ (*fmrestrict-set vs* (*snd a*)) ⊆ *prob-dom PROB*
using *FDOM-eff-subset-prob-dom-pair a*
by *blast*
then have *fmdom*′ (*s* $++_f$ *fmrestrict-set vs* (*snd a*)) = *fmdom*′ *s*
by (*simp add*: *calculation sup.absorb-iff1*)
}
ultimately have (*snd* (*action-proj a vs*) ++ *s*) ∈ *valid-states PROB*
unfolding *action-proj-def fmap-add-ltr-def valid-states-def*
by *simp*
}
then have *exec-plan s* (*as-proj* (*a* # *as*) *vs*) ∈ *valid-states PROB*
using *1 3 calculation*(*1*) *Cons.IH*[**where** *s* = *snd* (*action-proj a vs*) ++ *s*]
by *presburger*
}
moreover {
assume ¬(*fst* (*action-proj a vs*) $\subseteq_f$ *s*)
then have
exec-plan (*state-succ s* (*action-proj a vs*)) (*as-proj as vs*)

= *exec-plan s* (*as-proj as vs*)

unfolding *state-succ-def*
by *simp*
then have *exec-plan s* (*as-proj* (*a* # *as*) *vs*) ∈ *valid-states PROB*
using *2*
by *force*
}
ultimately have *exec-plan s* (*as-proj* (*a* # *as*) *vs*) ∈ *valid-states PROB*
by *blast*
}
moreover
{
assume *fmdom′* (*fmrestrict-set vs* (*snd a*)) = {}
then have
exec-plan s (*as-proj* (*a* # *as*) *vs*) =
exec-plan s (*as-proj as vs*)

by *simp*
then have *exec-plan s* (*as-proj* (*a* # *as*) *vs*) ∈ *valid-states PROB*
using *2*
by *argo*
}
ultimately show *?case*
by *blast*
qed *simp*

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 −
have *1*:
(*fmrestrict-set vs* (*exec-plan s* (*as-proj as vs*))
= *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*))

using *graph-plan-lemma-9* **by** *auto*
then obtain *s′* **where** *2*: *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*) = *fmrestrict-set vs s′*
using *1*
by *metis*
then have *fmrestrict-set vs s′* = *fmrestrict-set vs* (*exec-plan s as*)
using *assms sat-precond-exec-as-proj-eq-proj-exec*
by *metis*
then show
fmrestrict-set vs (*exec-plan s* (*as-proj as vs*)) = *fmrestrict-set vs* (*exec-plan s as*)

using *1 2*
by *argo*
qed

lemma *action-proj-idempot*:
fixes *a vs*
shows *action-proj* (*action-proj a vs*) *vs* = (*action-proj a vs*)
unfolding *action-proj-def*
by (*simp add*: *fmfilter-alt-defs*(*4*))

lemma *action-proj-idempot′*:
fixes *a vs*
assumes (*action-dom* (*fst a*) (*snd a*) ⊆ *vs*)
shows (*action-proj a vs* = *a*)
using *assms*
proof −
have *1*: *action-proj a vs* = (*fmrestrict-set vs* (*fst a*), *fmrestrict-set vs* (*snd a*))
by (*simp add*: *action-proj-def*)
then have *2*: (*fmdom′* (*fst a*) ∪ *fmdom′* (*snd a*)) ⊆ *vs*
unfolding *action-dom-def*
using *assms*
by (*auto simp add*: *action-dom-def*)
— NOTE Show that both components of 'a' remain unchanged.
{
then have *fmdom′* (*fst a*) ⊆ *vs*
by *blast*
then have *fmrestrict-set vs* (*fst a*) = (*fst a*)
using *exec-drest-5*
by *auto*
}
moreover {
have *fmdom′* (*snd a*) ⊆ *vs*
using *2*
by *auto*
then have *fmrestrict-set vs* (*snd a*) = (*snd a*)
using *exec-drest-5*
by *blast*
}
ultimately show *?thesis*
using *1*
by *simp*
qed

lemma *action-proj-idempot′′*:
fixes *P vs*
assumes *prob-dom P* ⊆ *vs*

shows *prob-proj P vs = P*
using *assms*
proof –
— TODO refactor.
{
fix *a*
assume *a* ∈ *P*
then have *action-dom* (*fst a*) (*snd a*) ⊆ *vs*
using *assms exec-as-proj-valid-2*
by *fast*
then have *action-proj a vs = a*
using *action-proj-idempot′*
by *fast*
}
then have *prob-proj P vs = P*
unfolding *prob-proj-def*
by *force*
then show *?thesis*
unfolding *prob-proj-def*
by *simp*
qed

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*))
using *assms*
proof (*induction as arbitrary*: *s s′ vs*)
case (*Cons a as*)
then have *1*:
fst a $\subseteq_f$ *s sat-precond-as* (*state-succ s a*) *as*
using *Cons.prems*(*1*)
by *simp+*
then have *2*: *fmrestrict-set vs* (*fst a*) $\subseteq_f$ *s*
using *assms*(*1*) *sat-precond-as-proj-4*
by *blast*
moreover
{
assume *fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {}
then have
sat-precond-as s′ (*as-proj* (*a # as*) *vs*)
= (
fst (*action-proj a vs*) $\subseteq_f$ *s′*
∧ *sat-precond-as* (*state-succ s′* (*action-proj a vs*)) (*as-proj as vs*)
)

using *calculation*
by *simp*

moreover
{
have *fst* (*action-proj a vs*) $\subseteq_f$ s' = (*fmrestrict-set vs* (*fst a*) $\subseteq_f$ s')
unfolding *action-proj-def*
by *simp*
moreover have (*fmrestrict-set vs* (*fst a*) $\subseteq_f$ *s*) = (*fmrestrict-set vs* (*fst a*) $\subseteq_f$ s')
using *Cons.prems*(*2*) *sat-precond-as-proj-1*
by *blast*
ultimately have *fst* (*action-proj a vs*) $\subseteq_f$ s'
using *2*
by *blast*
}
— TODO detailled proof for this sledgehammered step.
moreover have *sat-precond-as* (*state-succ* s' (*action-proj a vs*)) (*as-proj as vs*)
using *1 Cons.IH Cons.prems*(*2*) *drest-succ-proj-eq-drest-succ succ-drest-eq-drest-succ*
by *metis*
ultimately have (*sat-precond-as* s' (*as-proj* (*a* # *as*) *vs*))
by *blast*
}
moreover
{
assume *P1*: ¬(*fmdom'* (*fmrestrict-set vs* (*snd a*)) ≠ {})
then have *sat-precond-as* s' (*as-proj* (*a* # *as*) *vs*)
proof (*cases as-proj* (*a* # *as*) *vs*)
case *Cons2*: (*Cons* a' *list*)
— TODO unfold the sledgehammered metis steps.
then have *a*:
sat-precond-as s' (*as-proj* (*a* # *as*) *vs*)
= (*fst* a' $\subseteq_f$ s') ∧ *sat-precond-as* (*state-succ* s' a') *list*

using *P1 Cons.IH Cons.prems*(*1*, *2*) *Cons2*
by (*metis sat-precond-as-proj-3 empty-eff-not-in-as-proj sat-precond-as.simps*(*2*))
then have *b*: *fst* a' $\subseteq_f$ s'
unfolding *sat-precond-as.simps*(*2*)
using *P1 Cons.IH Cons.prems*(*1*, *2*) *sat-precond-as-proj-3 empty-eff-not-in-as-proj*
by (*metis sat-precond-as.simps*(*2*))
then have *sat-precond-as* (*state-succ* s' a') *list*
using *a*
by *blast*
then show *?thesis*
using *a b*
by *blast*
qed *fastforce*
}
ultimately show *?case*
by *blast*
qed *simp*

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*))
using *assms*
proof (*induction as arbitrary*: *s s′ vs*)
case (*Cons a as*)
then have *1*: *fst a* $\subseteq_f$ *s sat-precond-as* (*state-succ s a*) *as*
using *Cons.prems*
by *auto+*
then have *fmrestrict-set vs* (*fst a*) $\subseteq_f$ *fmrestrict-set vs s*
using *fmsubset-restrict-set-mono*
by *blast*
then have *fst* (*action-proj a vs*) $\subseteq_f$ *fmrestrict-set vs s′*
unfolding *action-proj-def*
using *Cons.prems*(*2*) *sat-precond-as-proj-1*
by *simp*
then have *fmrestrict-set vs* (*snd a*) = *fmrestrict-set vs* (*snd* (*action-proj a vs*))
unfolding *action-proj-def*
by (*simp add*: *fmfilter-alt-defs*(*4*))
then have *fst* (*action-proj a vs*) $\subseteq_f$ *s*
unfolding *action-proj-def*
using *1*(*1*) *fst-conv sat-precond-as-proj-4*
by *auto*
— TODO unfold these sledgehammered steps.
then have
fmrestrict-set vs (*state-succ s a*)
= *fmrestrict-set vs* (*state-succ* (*fmrestrict-set vs s′*) (*action-proj a vs*))

using *1*(*1*) *Cons.prems*(*2*)
by (*metis fmfilter-alt-defs*(*4*) *fmfilter-true fmlookup-restrict-set drest-succ-proj-eq-drest-succ option.simps*(*3*))
then show *?case*
using *Cons.prems*(*1*, *2*)
by (*metis fmfilter-alt-defs*(*4*) *fmfilter-true fmlookup-restrict-set sat-precond-as-proj option.simps*(*3*))
qed *simp*

lemma *as-proj-eq-as*:
assumes (*no-effectless-act as*) (*as* ∈ *valid-plans PROB*) (*prob-dom PROB* ⊆ *vs*)
shows (*as-proj as vs* = *as*)
using *assms*
proof (*induction as arbitrary*: *PROB vs*)
case (*Cons a as*)
— NOTE We only need to look at the first branch of 'as_proj'.
— TODO step should be refactored and proven explicitely because it's so pivotal.
then have *fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {}

unfolding *fmdom′-restrict-set-precise*
by (*metis*
FDOM-eff-subset-prob-dom-pair dual-order.trans inf.orderE
no-effectless-act.simps(*2*) *valid-plan-valid-head*)
— NOTE Proof 'action_proj a vs = a' for the first branch of 'as_proj'.
moreover {
assume *fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {}
— NOTE show 'action_proj a vs = a'.
moreover {
have *as-proj* (*a* # *as*) *vs* = *action-proj a vs* # *as-proj as vs*
using *calculation*
by *force*
then have *a* ∈ *PROB*
using *Cons.prems*(*2*) *valid-plan-valid-head*
by *fast*
then have *action-dom* (*fst a*) (*snd a*) ⊆ *prob-dom PROB*
using *exec-as-proj-valid-2*
by *fast*
then have *action-dom* (*fst a*) (*snd a*) ⊆ *vs*
using *Cons.prems*(*3*)
by *fast*
then have *action-proj a vs* = *a*
using *action-proj-idempot′*
by *fast*
}
— NOTE show that 'as_proj as vs = as'.
moreover {
have *1*: *no-effectless-act as*
using *Cons.prems*(*1*)
by *simp*
then have *as* ∈ *valid-plans PROB*
using *Cons.prems*(*2*) *valid-plan-valid-tail*
by *fast*
then have *as-proj as vs* = *as*
using *Cons.prems*(*3*) *Cons.IH 1*
by *blast*
}
ultimately have *as-proj* (*a* # *as*) *vs* = *a* # *as*
by *simp*
}
ultimately show *?case*
by *fast*
qed *simp*

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 (*induction as arbitrary*: *s vs*)
case (*Cons a as*)
— Split cases on the branching introduced by 'remove_effectless_act' and 'as_proj'.
then show *?case*
proof (*cases fmdom′* (*snd a*) ≠ {})
case *true1*: *True*
then show *?thesis*
proof (*cases fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {})
case *False*
then show *?thesis* **by** (*simp add*: *Cons true1*)
qed (*simp add*: *Cons true1*)
next
case *False*
then show *?thesis*
proof (*cases fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {})
case *true2*: *True*
then have *1*: *fmdom′* (*snd a*) ∩ *vs* = {}
using *False Int-empty-left*
by *force*
— NOTE This step shows that the case for *fmdom′* (*fmrestrict-set vs* (*snd a*)) ≠ {} is impossible.
— TODO could be refactored into a (simp) lemma ('as_proj_eq_as' also uses this?).
then have *fmdom′* (*fmrestrict-set vs* (*snd a*)) = {}
by (*simp add*: *fmdom′-restrict-set-precise*)
then show *?thesis*
using *true2*
by *blast*
qed (*simp add*: *Cons*)
qed
qed *simp*

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*)
using *assms as-proj-eq-as exec-rem-effless-as-proj-eq-exec-as-proj rem-effectless-works-1 rem-effectless-works-6*
rem-effectless-works-9 sublist-valid-plan
by *metis*

lemma *dom-prob-proj*: *prob-dom* (*prob-proj PROB vs*) ⊆ *vs*
using *graph-plan-neq-mems-state-set-neq-len*
by *fast*

— NOTE added lemma.
— TODO refactor into ‘FmapUtils.thy‘.
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*
  using *assms*
proof −
  {
    fix *v*
    have *fmlookup* (*fmrestrict-set vs1* (*fmrestrict-set vs2 f*)) *v* = *fmlookup* (*fmrestrict-set vs1 f*) *v*
      using *assms*
    proof (*cases v* ∈ *vs1*)
      case *False*
      then show *?thesis*
      proof (*cases v* ∈ *vs2*)
        case *False*
        then have *v* ∉ *vs1*
          using *False assms*
          by *blast*
        then have
          fmlookup (*fmrestrict-set vs1* (*fmrestrict-set vs2 f*)) *v* = *None*
          fmlookup (*fmrestrict-set vs1 f*) *v* = *None*
          by *simp*+
        then show *?thesis*
          by *argo*
      qed *simp*
    qed *auto*
  }
  then show *?thesis*
    using *fmap-ext*
    by *blast*
qed

lemma *subset-proj-absorb-1*:
  assumes (*vs1* ⊆ *vs2*)
  shows (*action-proj* (*action-proj a vs2*) *vs1* = *action-proj a vs1*)
  using *assms*
proof −
  have
    fmrestrict-set vs1 (*fmrestrict-set vs2* (*fst a*)) = *fmrestrict-set vs1* (*fst a*)
    fmrestrict-set vs1 (*fmrestrict-set vs2* (*snd a*)) = *fmrestrict-set vs1* (*snd a*)
    using *assms subset-proj-absorb-1-a*
    by *blast*+
  then show *?thesis*
    unfolding *action-proj-def*
    by *simp*
qed

```
lemma subset-proj-absorb:
  fixes PROB vs1 vs2
  assumes vs1 ⊆ vs2
  shows prob-proj (prob-proj PROB vs2) vs1 = prob-proj PROB vs1
proof –
  {
    have
      prob-proj (prob-proj PROB vs2) vs1
      = ((λa. action-proj a vs1) ∘ (λa. action-proj a vs2)) ` PROB

      unfolding prob-proj-def
      by fastforce
    also have ... = (λa. action-proj (action-proj a vs2) vs1) ` PROB
      by fastforce
    also have ... = (λa. action-proj a vs1) ` PROB
      using assms subset-proj-absorb-1
      by metis
    also have ... = prob-proj PROB vs1
      unfolding prob-proj-def
      by simp
    finally have prob-proj (prob-proj PROB vs2) vs1 = prob-proj PROB vs1
      by simp
  }
  then show ?thesis
    by simp
qed


lemma union-proj-absorb:
  fixes PROB vs vs'
  shows prob-proj (prob-proj PROB (vs ∪ vs')) vs = prob-proj PROB vs
  by (simp add: subset-proj-absorb)


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)))))
  )
  unfolding action-proj-def
  using assms
  by (simp add: IN-FDOM-DRESTRICT-DIFF FDOM-pre-subset-prob-dom-pair
      FDOM-eff-subset-prob-dom-pair)
```

```
lemma IN-DISJ-DEP-IMP-DEP-DIFF:
  fixes PROB vs vs' v v'
  assumes (v ∈ vs') (v' ∈ vs') (disjnt vs vs')
  shows (dep PROB v v' ⟶ dep (prob-proj PROB (prob-dom PROB − vs)) v v')
  using assms
proof (cases v = v')
  case False
  {
    assume P: dep PROB v v'
    then obtain a where a:
        (v ∈ fmdom' (fst a) ∧ v' ∈ fmdom' (snd a) ∨ v ∈ fmdom' (snd a) ∧ v' ∈
fmdom' (snd a))
      a ∈ PROB
      unfolding dep-def
      using False
      by blast
    {
      have v ∉ vs
        using assms(1, 3)
        unfolding disjnt-def
        by blast
      then have (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))))
        using a NOT-VS-IN-DOM-PROJ-PRE-EFF
        by metis+
    }
    note b = this
    then consider (i) v ∈ fmdom' (fst a) ∧ v' ∈ fmdom' (snd a)
      | (ii) v ∈ fmdom' (snd a) ∧ v' ∈ fmdom' (snd a)
      using a
      by blast
    then have dep (prob-proj PROB (prob-dom PROB − vs)) v v'
    proof (cases)
      case i
      then show ?thesis
        using assms(2, 3) a(2) b(1)
      by (meson dep-def disjnt-iff action-proj-in-prob-proj NOT-VS-IN-DOM-PROJ-PRE-EFF)
    next
      case ii
      then show ?thesis
        using assms(2, 3) a(2) b(2)
      by (meson dep-def disjnt-iff action-proj-in-prob-proj NOT-VS-IN-DOM-PROJ-PRE-EFF)
    qed
  }
  then show ?thesis
```

by *blast*
qed (*auto simp*: *dep-def prob-proj-def disjnt-def*)

lemma *PROB-DOM-PROJ-DIFF*:
fixes *P vs*
shows *prob-dom* (*prob-proj PROB* (*prob-dom PROB* − *vs*)) = (*prob-dom PROB*) − *vs*
using *graph-plan-neq-mems-state-set-neq-len*
by *fastforce*

lemma *two-children-parent-mems-le-finite*:
fixes *PROB vs*
assumes (*vs* ⊆ *prob-dom PROB*)
shows (*prob-dom* (*prob-proj PROB vs*) = *vs*)
using *assms graph-plan-neq-mems-state-set-neq-len*
by *fast*

— TODO showcase (non-trivial proof).
— TODO find explicit proof.
lemma *PROJ-DOM-PRE-EFF-SUBSET-DOM*:
fixes *a vs*
shows
(*fmdom′* (*fst* (*action-proj a vs*)) ⊆ *fmdom′* (*fst a*))
∧ (*fmdom′* (*snd* (*action-proj a vs*)) ⊆ *fmdom′* (*snd a*))

unfolding *action-proj-def*
by (*auto simp*: *fmdom′-restrict-set-precise*)

lemma *NOT-IN-PRE-EFF-NOT-IN-PRE-EFF-PROJ*:
fixes *a v vs*
shows
(¬(*v* ∈ *fmdom′* (*fst a*)) ⟶ ¬(*v* ∈ *fmdom′* (*fst* (*action-proj a vs*))))
∧ (¬(*v* ∈ *fmdom′* (*snd a*)) ⟶ ¬(*v* ∈ *fmdom′* (*snd* (*action-proj a vs*))))

using *PROJ-DOM-PRE-EFF-SUBSET-DOM rev-subsetD*
by *metis*

lemma *dep-proj-dep*:
assumes *dep* (*prob-proj PROB vs*) *v v′*
shows *dep PROB v v′*
using *assms*
unfolding *dep-def prob-proj-def action-proj-def image-def*
apply (*auto simp*: *fmdom′-restrict-set-precise*)
by *auto*

lemma *NDEP-PROJ-NDEP*:
fixes *PROB vs vs′ vs″*
assumes (¬*dep-var-set PROB vs vs′*)
shows (¬*dep-var-set* (*prob-proj PROB vs″*) *vs vs′*)
using *assms dep-proj-dep*
unfolding *dep-var-set-def*
by *metis*

lemma *SUBSET-PROJ-DOM-DISJ*:
fixes *PROB vs vs′*
assumes (*vs* ⊆ (*prob-dom* (*prob-proj PROB* (*prob-dom PROB* − *vs′*))))
shows *disjnt vs vs′*
using *assms*
by (*auto simp add*: *PROB-DOM-PROJ-DIFF subset-iff disjnt-iff*)

— TODO showcase (lemma which is solved effortlessly by automation).
lemma *NOT-VS-DEP-IMP-DEP-PROJ*:
fixes *PROB vs v v′*
assumes ¬(*v* ∈ *vs*) ¬(*v′* ∈ *vs*) (*dep PROB v v′*)
shows (*dep* (*prob-proj PROB* (*prob-dom PROB* − *vs*)) *v v′*)
using *assms*
by (*metis Diff-disjoint Diff-iff disjnt-def insertCI IN-DISJ-DEP-IMP-DEP-DIFF*)

lemma *DISJ-PROJ-NDEP-IMP-NDEP*:
fixes *PROB vs vs′ vs″*
assumes
(*disjnt vs vs″*) *disjnt vs vs′*
¬(*dep-var-set* (*prob-proj PROB* (*prob-dom PROB* − *vs*)) *vs′ vs″*)
shows ¬(*dep-var-set PROB vs′ vs″*)
proof −
{
assume *C*: *dep-var-set PROB vs′ vs″*
then obtain *v1 v2* **where** *v1* ∈ *vs′ v2* ∈ *vs″ disjnt vs′ vs″ dep PROB v1 v2*
unfolding *dep-var-set-def*
by *blast*
then have ∃ *v1 v2*.
v1 ∈ *vs′* ∧ *v2* ∈ *vs″* ∧ *disjnt vs′ vs″* ∧ *dep* (*prob-proj PROB* (*prob-dom PROB*
− *vs*)) *v1 v2*

using *assms*(*1*, *2*) *IntI disjnt-def empty-iff NOT-VS-DEP-IMP-DEP-PROJ*
by *metis*
then have *False*
using *assms*
unfolding *dep-var-set-def*

by *blast*
}
then show *?thesis*
using *assms*
unfolding *dep-var-set-def*
by *argo*
qed

lemma *PROJ-DOM-IDEMPOT*:
fixes *PROB*
shows *prob-proj PROB* (*prob-dom PROB*) = *PROB*
using *action-proj-idempot''*
by *blast*

lemma *prob-proj-idempot*:
fixes *vs vs'*
assumes (*vs* ⊆ *vs'*)
shows (*prob-proj PROB vs* = *prob-proj* (*prob-proj PROB vs'*) *vs*)
using *assms subset-proj-absorb*
by *blast*

lemma *prob-proj-dom-diff-eq-prob-proj-prob-proj-dom-diff*:
fixes *vs vs'*
shows
prob-proj PROB (*prob-dom PROB* − (*vs* ∪ *vs'*))
= *prob-proj*
(*prob-proj PROB* (*prob-dom PROB* − *vs*))
(*prob-dom* (*prob-proj PROB* (*prob-dom PROB* − *vs*)) − *vs'*)

using *PROB-DOM-PROJ-DIFF subset-proj-absorb*
by (*metis Compl-Diff-eq Diff-subset compl-eq-compl-iff sup-assoc*)

lemma *PROJ-DEP-IMP-DEP*:
fixes *PROB vs v v'*
assumes *dep* (*prob-proj PROB* (*prob-dom PROB* − *vs*)) *v v'*
shows *dep PROB v v'*
using *assms*
unfolding *dep-def prob-proj-def*
proof (*cases v* = *v'*)
case *False*
then show (∃ *a*.
a ∈ *PROB*
∧ (*v* ∈ *fmdom'* (*fst a*) ∧ *v'* ∈ *fmdom'* (*snd a*) ∨ *v* ∈ *fmdom'* (*snd a*) ∧ *v'* ∈ *fmdom'* (*snd a*)))
∨ *v* = *v'*

using *assms*
unfolding *dep-def prob-proj-def*
by (*smt image-iff NOT-IN-PRE-EFF-NOT-IN-PRE-EFF-PROJ*)
qed *blast*

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 (
$(\neg((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v\ v'))$
$\vee\ (\exists\, v''.$
$v'' \in vs$
$\wedge\ ((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v\ v'')$
$\wedge\ ((\lambda v1'\ v2'.\ dep\ PROB\ v1'\ v2')^{++}\ v''\ v')$
)
)
using *assms NOT-VS-DEP-IMP-DEP-PROJ DEP-REFL REFL-TC-CONJ*[*of*
$\lambda v\ v'.\ dep\ PROB\ v\ \ v'\ \lambda v.\ \neg(v \in vs)\ \lambda v\ v'.\ dep\ (prob\text{-}proj\ PROB\ (prob\text{-}dom\ PROB- \ vs))\ v\ v'$
v v′]
by *fastforce*

lemma *every-action-proj-eq-as-proj*:
fixes *as vs*
shows *list-all* (λ *a. action-proj a vs* = *a*) (*as-proj as vs*)
by (*induction as*) (*auto simp add*: *action-proj-idempot*)

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*)
using *assms*
by (*auto simp add*: *action-proj-def*)

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′*)
using *assms*
proof (*induction as arbitrary*: *as′ vs*)
case *Nil*
moreover have *as′* = []
using *Nil.prems sublist-NIL*

```
    by force
  then show ?case
    by simp
next
  case cons: (Cons a as)
  then show ?case
  proof (cases as′)
    case (Cons aa list)
    then show ?thesis
    proof (cases fmdom′ (fmrestrict-set vs (snd aa)) ≠ {})
      case True
      then have as-proj as′ vs = action-proj aa vs # as-proj list vs
        using Cons True
        by auto
      then show ?thesis
      by (metis as-proj.simps(2) cons.IH cons.prems action-proj-idempot local.Cons
            subseq-Cons2-iff)
    next
      case False
      then have as-proj as′ vs = as-proj list vs
        using Cons False
        by simp
      then show ?thesis using cons False
        unfolding Cons
          by (smt action-proj-def action-proj-idempot as-proj.simps(2) prod.inject
subseq-Cons2-neq)
    qed
  qed simp
qed


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

proof −
  have 1: snd (action-proj a vs) = fmrestrict-set vs (snd a)
    unfolding action-proj-def
    by simp
  then have fmdom′ (fmrestrict-set vs (snd a)) ⊆ fmdom′ (snd a)
    using act-dom-proj-eff-subset-act-dom-eff
    by metis
  then show ?thesis
    using assms 1
    by auto
qed
```

— NOTE showcase (the step using ‘graph_plan_lemma_5‘— labelled by ’[1]’— is non-trivial proof due to missing premises and the last six proof steps are redundant).
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 −
  have *1*: *fmdom′* (*snd a*) ∩ (*fmdom′ s* − *vs*) = {}
    using *assms*(*1*) *vset-disj-eff-diff*
    by *blast*
  then have *2*:
    fmrestrict-set (*fmdom′ s* − *vs*) *s* = *fmrestrict-set* (*fmdom′ s* − *vs*) (*state-succ s a*)
    using *disj-imp-eq-proj-exec*[**where** *vs* = *fmdom′ s* − *vs*]
    by *blast*
  then have *fmdom′* (*snd* (*action-proj a vs*)) ∩ (*fmdom′ s* − *vs*) = {}
    using *1 DISJ-EFF-DISJ-PROJ-EFF*[**where** *s* = (*fmdom′ s* − *vs*)]
    by *blast*
  then have
    fmrestrict-set (*fmdom′ s* − *vs*) *s*
    = *fmrestrict-set* (*fmdom′ s* − *vs*) (*state-succ s* (*action-proj a vs*))

    using *disj-imp-eq-proj-exec*[**where** *a* = (*action-proj a vs*) **and** *vs* = *fmdom′ s* − *vs*]
    by *blast*
  then have *fmdom′* (*snd* (*action-proj a vs*)) ∩ (*fmdom′ s* − *vs*) = {}
    using *1 DISJ-EFF-DISJ-PROJ-EFF*[**where** *s* = (*fmdom′ s* − *vs*)]
    by *blast*
  then have
    fmrestrict-set (*fmdom′ s* − *vs*) *s* =
    fmrestrict-set (*fmdom′ s* − *vs*) (*state-succ s* (*action-proj a vs*))

    using *disj-imp-eq-proj-exec*[*of action-proj a vs fmdom′ s* − *vs*]
    by *fast*
      — [1]
      — TODO unwrap this step.
  then show *?thesis*
    using *2 FDOM-state-succ graph-plan-lemma-5*[**where** *s* = *state-succ s* (*action-proj a vs*)
      and *s′* = *state-succ s a* **and** *vs* = *vs*] *assms*(*2*, *3*) *dom-eff-subset-imp-dom-succ-eq-proj drest-proj-succ-eq-drest-succ*
    by *metis*
qed

— NOTE duplicate declaration of lemma ‘state_succ_proj_eq_state_succ‘ removed.

lemma *no-effectless-proj*:
  fixes *vs as*
  shows *no-effectless-act* (*as-proj as vs*)
  by (*induction as arbitrary*: *vs*) (*auto simp add*: *action-proj-def*)

— NOTE duplicate (this is identical to 'valid_as_valid_as_proj').
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*))
  using *assms valid-as-valid-as-proj*
  by *blast*

— TODO Unwrap the smt proof.
lemma *prob-proj-comm*:
  fixes *PROB vs vs′*
  shows *prob-proj* (*prob-proj PROB vs*) *vs′* = *prob-proj* (*prob-proj PROB vs′*) *vs*
  by (*smt graph-plan-neq-mems-state-set-neq-len inf-commute inf-le2 PROJ-DOM-IDEMPOT prob-proj-idempot*)

— TODO Unwrap the metis 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*)
  unfolding *varset-action-def action-proj-def*
  using *assms*
  by (*metis action-proj-def exec-drest-5 snd-conv varset-action-def*)

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 −
  have *1*: (*fmdom′* (*snd a*) ⊆ *vs*)
    using *assms varset-action-def*
    by *metis*
  moreover
  {
    — TODO refactor and put into 'Fmap_Utils'.
    have
      fmdom′ (*snd* (
        fmrestrict-set (*prob-dom PROB* − *vs′*) (*fst a*)
        , *fmrestrict-set* (*prob-dom PROB* − *vs′*) (*snd a*)
      ))

= (*fmdom′* (*snd a*) ∩ (*prob-dom PROB* − *vs′*))

by (*simp add*: *Int-def Set.filter-def fmfilter-alt-defs*(*4*))
also have . . . ⊆ *fmdom′* (*snd a*)
by *simp*
finally have *fmdom′* (*snd* (
fmrestrict-set (*prob-dom PROB* − *vs′*) (*fst a*)
, *fmrestrict-set* (*prob-dom PROB* − *vs′*) (*snd a*)
))
⊆ *vs*

using *1* **by** *simp*
}
ultimately show *?thesis*
unfolding *varset-action-def dep-var-set-def dep-def action-proj-def*
by *blast*
qed

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 −
have ∀ *f fa fb p*.
action-proj (*action-proj* (*action-proj p f*) *fb*) *fa* = *action-proj* (*action-proj p f*) *fb*
∨ ¬ *action-dom* (*fst p*::(′*a*, ′*b*) *fmap*) (*snd p*::(-, ′*c*) *fmap*) ∩ (*f* ∩ *fb*) ⊆ *fa*

by (*metis* (*no-types*) *action-proj-idempot′ proj-action-dom-eq-inter inf-assoc*)
then have ∀ *f fa p*.
action-proj (*action-proj* (*p*::(′*a*, ′*b*) *fmap* × (-, ′*c*) *fmap*) *f*) *fa*
= *action-proj p* (*f* ∩ *fa*)

by (*metis* (*no-types*) *inf.cobounded2 inf-commute subset-proj-absorb-1*)
then show *?thesis*
using *assms*
by (*metis Diff-Int-distrib2 Diff-empty action-proj-idempot′*)
qed

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 −
{
fix *a*
assume *P*: *a* ∈ *PROB*

moreover have *action-dom* (*fst a*) (*snd a*) ⊆ *prob-dom PROB*
using *P exec-as-proj-valid-2*
by *blast*
ultimately have *action-proj* (*action-proj a* (*prob-dom PROB* − *vs′*)) *vs* = *action-proj a vs*
using *assms action-proj-disj-diff*[*of a prob-dom PROB vs′ vs*]
by *blast*
}
then show *?thesis*
unfolding *prob-proj-def*
by (*smt image-cong image-image*)
qed

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*)
unfolding *varset-action-def action-proj-def*
using *assms*
by (*simp add*: *exec-drest-5 varset-action-def*)

— NOTE type of 'PROB' had to be fixed for use of 'empty_problem_bound'.
lemma *empty-problem-proj-bound*:
fixes *PROB* :: *′a problem*
shows *problem-plan-bound* (*prob-proj PROB* {}) = *0*
proof −
— TODO refactor?
{
have *prob-proj* {} {} = {}
unfolding *prob-proj-def action-proj-def*
using *image-empty*
by *simp*
moreover {
assume *P*: *PROB* ≠ {}
have ∀ *a*. (*fmrestrict-set* {} (*fst a*), *fmrestrict-set* {} (*snd a*)) = (*fmempty*, *fmempty*)
using *fmrestrict-set-null*
by *simp*
then have *prob-proj PROB* {} = {(*fmempty*, *fmempty*)}
unfolding *prob-proj-def action-proj-def*
using *P*
by *auto*
}
ultimately consider
(*i*) *prob-proj PROB* {} = {}
| (*ii*) *prob-proj PROB* {} = {(*fmempty*, *fmempty*)}
by (*cases PROB* = {}) *force+*

then have *prob-dom* (*prob-proj PROB* {}) = {}
unfolding *prob-dom-def action-dom-def* **using** *fmdom′-empty*
by (*cases*) *force+*
}
then show *?thesis*
using *empty-problem-bound*[**where** *PROB=prob-proj PROB* {}]
by *blast*
qed

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 −
{
have *exec-plan* (*fmrestrict-set vs s*) (*as-proj as vs*) = *fmrestrict-set vs* (*exec-plan s as*)
using *assms*(*4*) *sat-precond-exec-as-proj-eq-proj-exec*
by *blast*
moreover have *fmrestrict-set vs s* ∈ *valid-states* (*prob-proj PROB vs*)
using *assms*(*2*) *graph-plan-not-eq-last-diff-paths*
by *auto*
moreover have *as-proj as vs* ∈ *valid-plans* (*prob-proj PROB vs*)
using *assms*(*3*) *valid-as-valid-as-proj*
by *blast*
moreover have *finite* (*prob-proj PROB vs*)
unfolding *prob-proj-def*
using *assms*(*1*)
by *simp*
ultimately have ∃ *as*′.
exec-plan (*fmrestrict-set vs s*) (*as-proj as vs*) = *exec-plan* (*fmrestrict-set vs s*) *as*′
∧ *subseq as*′ (*as-proj as vs*) ∧ *length as*′ ≤ *problem-plan-bound* (*prob-proj PROB vs*)

using *problem-plan-bound-works*[*of prob-proj PROB vs fmrestrict-set vs s as-proj as vs*]
by *blast*
}
then obtain *as*′ **where**

exec-plan (*fmrestrict-set vs s*) (*as-proj as vs*) = *exec-plan* (*fmrestrict-set vs s*) *as′*
subseq as′ (*as-proj as vs*) ∧ *length as′* ≤ *problem-plan-bound* (*prob-proj PROB vs*)
by *fast*
moreover {
have
exec-plan (*fmrestrict-set vs s*) *as*
= *exec-plan* (*fmrestrict-set vs s*) (*rem-condless-act* (*fmrestrict-set vs s*) [] *as*)

using *rem-condless-valid-1*[*of fmrestrict-set vs s as*]
by *blast*
then have *subseq* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*) *as′*
using *rem-condless-valid-8* [*of fmrestrict-set vs s as′*]
by *blast*
}
moreover have *length* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*) ≤ *length as′*
using *rem-condless-valid-3*[*of fmrestrict-set vs s*]
by *fast*
moreover have *4*:
sat-precond-as (*fmrestrict-set vs s*) (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*)
using *rem-condless-valid-2*[*of fmrestrict-set vs s as′*]
by *blast*
moreover have
exec-plan (*fmrestrict-set vs s*) (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*)
= *exec-plan* (*fmrestrict-set vs s*)
(*rem-effectless-act* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*))

using *rem-effectless-works-1*[*of fmrestrict-set vs s*
rem-condless-act (*fmrestrict-set vs s*) [] *as′*]
by *blast*
moreover {
have
subseq (*rem-effectless-act* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as*))
(*rem-condless-act* (*fmrestrict-set vs s*) [] *as*)

using *rem-effectless-works-9*[*of*
(*rem-condless-act* (*fmrestrict-set vs s*) [] (*as* :: *′a action list*))]
by *blast*
then have
length (*rem-effectless-act* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*))
≤ *length* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*)

using *rem-effectless-works-3*[*of*
(*rem-condless-act* (*fmrestrict-set vs s*) [] (*as′* :: *′a action list*))]
by *simp*
then have
sat-precond-as (*fmrestrict-set vs s*)
(*rem-effectless-act* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as′*))

using *4 rem-effectless-works-2*[*of fmrestrict-set vs s*
(*rem-condless-act* (*fmrestrict-set vs s*) [] *as*′)]
by *blast*
then have
no-effectless-act (*rem-effectless-act* (*rem-condless-act* (*fmrestrict-set vs s*) [] *as*′))
using *rem-effectless-works-6*[*of* (*rem-condless-act* (*fmrestrict-set vs s*) [] (*as*′ ::′*a action list*))]
by *simp*
}
ultimately show *?thesis*
using *rem-effectless-works-13 rem-condless-valid-1 order-trans*
no-effectless-proj sat-precond-drest-sat-precond subseq-order.order-trans
by (*metis* (*no-types*, *lifting*))
qed

— NOTE added lemma.
— TODO refactor into ‘Fmap_Utils‘.
lemma *action-proj-inter-i*: *fmrestrict-set V* (*fmrestrict-set W f*) = *fmrestrict-set* (*V* ∩ *W*) *f*
unfolding *fmfilter-alt-defs*(*4*)
by *simp*

lemma *action-proj-inter*: *action-proj* (*action-proj a vs1*) *vs2* = *action-proj a* (*vs1* ∩ *vs2*)
proof −
have
fmrestrict-set vs2 (*fmrestrict-set vs1* (*fst a*)) = *fmrestrict-set* (*vs1* ∩ *vs2*) (*fst a*)
fmrestrict-set vs2 (*fmrestrict-set vs1* (*snd a*)) = *fmrestrict-set* (*vs1* ∩ *vs2*) (*snd a*)
using *inf-commute action-proj-inter-i*
by *metis*+
then show *?thesis*
unfolding *action-proj-def*
by *simp*
qed

lemma *prob-proj-inter*: *prob-proj* (*prob-proj PROB vs1*) *vs2* = *prob-proj PROB* (*vs1* ∩ *vs2*)
unfolding *prob-proj-def*
using *set-eq-iff image-iff action-proj-inter*
supply[[*smt-timeout=100*]]
by (*smt image-cong image-image*)

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 $\equiv$ ($\forall$ *v*. (*v* $\in$ *fmdom′ s1*) $\wedge$ (*v* $\in$ *fmdom′ s2*) $\longrightarrow$ (*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-states PROB*) *fst a* $\subseteq_f$ *s agree* (*snd a*) *s*
shows *state-succ s a* = *s*
proof −
{
have *fmdom′* (*snd a*) $\subseteq$ *fmdom′ s*
using *assms*(*1*, *2*) *FDOM-eff-subset-FDOM-valid-states-pair*
by *blast*
moreover have $\forall$ *x*. *x* $\in$ *fmdom′* (*snd a*) $\longrightarrow$ *fmlookup* (*snd a*) *x* = *fmlookup s x*
using *assms*(*4*) *calculation*(*1*) *agree-def subsetCE*
by *metis*
moreover have *s* $++_f$ *snd a* = *s*
using *calculation*(*2*)
by (*metis fmap-ext fmdom′-notD fmdom-notI fmlookup-add*)
}
then show *?thesis*
using *fmap-add-ltr-def state-succ-def*
by *metis*
qed

lemma *agree-state-succ-idempot*:
assumes (*a* $\in$ *PROB*) (*s* $\in$ *valid-states PROB*) (*agree* (*snd a*) *s*)
shows (*state-succ s a* = *s*)
proof (*cases fst a* $\subseteq_f$ *s*)
case *True*
then show *?thesis*
using *assms state-succ-fixpoint-if*
by *blast*
next

case *False*
then show *?thesis*
unfolding *state-succ-def fmap-add-ltr-def*
by *simp*
qed

— NOTE added lemma.
— TODO refactor into 'Fmap_Utils'.
lemma *fmdom′-fmrestrict-set*:
fixes *X f*
shows $fmdom'$ $(fmrestrict\text{-}set\ X\ f) = X \cap (fmdom'\ f)$
unfolding *fmdom′-alt-def fmfilter-alt-defs*(*4*)
by *auto*

— NOTE added lemma.
— TODO refactor into 'Fmap_Utils'.
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 −
have $fmrestrict\text{-}set\ X\ (f\ {+}{+}_f\ g) = fmrestrict\text{-}set\ X\ f\ {+}{+}_f\ fmrestrict\text{-}set\ X\ g$
using *fmrestrict-set-add-distrib*
by *fast*
then show *?thesis*
using *fmdom′-fmrestrict-set fmdom′-add*
by *metis*
qed

— NOTE added lemma.
— TODO refactor into 'Fmap_Utils'.
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 −
{
fix *v*
assume $v \in X \cap fmdom'\ f \cap fmdom'\ g$
then have $v \in fmdom'\ (fmrestrict\text{-}set\ X\ f) \wedge v \in fmdom'\ (fmrestrict\text{-}set\ X\ g)$
using *fmdom′-fmrestrict-set*
by *force*
then have $fmlookup\ (fmrestrict\text{-}set\ X\ f)\ v = fmlookup\ (fmrestrict\text{-}set\ X\ g)\ v$
using *assms*(*1*)
unfolding *agree-def*
by *blast*
}
then show *?thesis*

using *assms*
by *blast*
qed

lemma *agree-restrict-state-succ-idempot*:
assumes ($a \in PROB$) ($s \in$ *valid-states* $PROB$)
(*agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*))
shows (*fmrestrict-set vs* (*state-succ s a*) = *fmrestrict-set vs s*)
proof (*cases fst a* $\subseteq_f$ *s*)
case *True*
then have *state-succ s a* = s $++_f$ *snd a*
unfolding *state-succ-def fmap-add-ltr-def*
by *simp*
{
fix *v*
have *fmlookup* (*fmrestrict-set vs* (s $++_f$ *snd a*)) *v* = *fmlookup* (*fmrestrict-set vs s*) *v*
proof (*cases* $v \in$ *fmdom′* (*snd a*))
case *True*
then have *1*: *fmdom′* (*fmrestrict-set vs* (s $++_f$ *snd a*)) = *vs* $\cap$ (*fmdom′ s* $\cup$ *fmdom′* (*snd a*))
unfolding *fmap-add-ltr-def*
using *fmdom′-fmrestrict-set-fmadd*
by *metis*
then have *2*: *fmdom′* (*fmrestrict-set vs* (*snd a*)) = *vs* $\cap$ *fmdom′* (*snd a*)
using *fmdom′-fmrestrict-set*
by *metis*
then show *?thesis*
using *1 2*
proof (*cases* $v \in vs$)
case *true*: *True*
then show *?thesis*
proof (*cases* $v \in$ (*fmdom′ s* $\cap$ *fmdom′* (*snd a*)))
case *True*
then have $v \in vs \cap$ *fmdom′ s* $\cap$ *fmdom′* (*snd a*)
using *true*
by *blast*
then have *fmlookup* (*fmrestrict-set vs* (*snd a*)) *v* = *fmlookup* (*fmrestrict-set vs s*) *v*
using *assms*(*3*) *fmrestrict-agree*
by *fast*
then show *?thesis*
by *fastforce*
next
case *False*
then have *fmdom′* (*snd a*) $\subseteq$ *fmdom′ s*
using *assms*(*1*, *2*) *FDOM-eff-subset-FDOM-valid-states-pair*
by *metis*
then have $v \notin$ *fmdom′* (*snd a*)

```
          using true False
          by blast
        then show ?thesis
          by fastforce
      qed
    qed auto
  qed fastforce
 }
 then show ?thesis
  unfolding state-succ-def fmap-add-ltr-def
  using fmap-ext
  by metis
next
 case False
 then show ?thesis
  unfolding state-succ-def
  by simp
qed
```

```
lemma agree-exec-idempot:
 assumes (as ∈ valid-plans PROB) (s ∈ valid-states PROB)
  (∀ a. ListMem a as ⟶ agree (snd a) s)
 shows (exec-plan s as = s)
 using assms
proof (induction as arbitrary: PROB s)
 case (Cons a as)
 then have 1: a ∈ PROB
  using Cons.prems(1) valid-plan-valid-head
  by fast
 then have 2: as ∈ valid-plans PROB
  using Cons.prems(1) valid-plan-valid-tail
  by fast
 then have 3: ∀ a. ListMem a as ⟶ agree (snd a) s
  using Cons.prems(3) ListMem.simps
  by metis
 then have ListMem a (a # as)
  using elem
  by fast
 then have agree (snd a) s
  using Cons.prems(3)
  by blast
 then have 4: state-succ s a = s
  using Cons.prems(1, 2) 1 agree-state-succ-idempot
  by blast
 then have exec-plan s as = s
  using Cons.IH Cons.prems(2) 2 3
  by blast
 then show ?case
```

    using *4*
    by *simp*
qed *simp*

lemma *agree-restrict-exec-idempot*:
  fixes *s s′*
  assumes (*as* ∈ *valid-plans PROB*) (*s′* ∈ *valid-states PROB*) (*s* ∈ *valid-states PROB*)
    (∀ *a*. *ListMem a as* ⟶ *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*))
    (*fmrestrict-set vs s′* = *fmrestrict-set vs s*)
  shows (*fmrestrict-set vs* (*exec-plan s′ as*) = *fmrestrict-set vs s*)
  using *assms*
proof (*induction as arbitrary*: *PROB s s′ vs*)
  case (*Cons a as*)
  have *1*: *as* ∈ *valid-plans PROB*
    using *Cons.prems*(*1*) *valid-plan-valid-tail*
    by *fast*
  then have *2*: ∀ *a*. *ListMem a as* ⟶ *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*)
    using *Cons.prems*(*4*) *ListMem.simps*
    by *metis*
  then have *3*: *a* ∈ *PROB*
    using *Cons.prems*(*1*) *valid-plan-valid-head*
    by *metis*
  moreover
  {
    have *ListMem a* (*a* # *as*)
      using *elem*
      by *fast*
    then have *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*)
      using *Cons.prems*(*4*) *calculation*(*1*)
      by *blast*
    then have *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s′*)
      using *Cons.prems*(*5*)
      by *simp*
  }
  ultimately show *?case*
    using *assms*
  proof (*cases fst a* $\subseteq_f$ *s′*)
    case *True*
    {
      have *a*: *s′* ∈ *valid-states PROB*
        using *Cons.prems*(*2*)
        by *simp*
      moreover have *state-succ s′ a* ∈ *valid-states PROB*
        using *3 a lemma-1-i*
        by *blast*
      moreover have

∀ *a*. *ListMem a as* ⟶ *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*)
using *2*
by *blast*
moreover {
have *ListMem a* (*a* # *as*)
using *elem*
by *fast*
then have *agree* (*fmrestrict-set vs* (*snd a*)) (*fmrestrict-set vs s*)
using *Cons.prems*(*4*) *calculation*(*1*)
by *blast*
then have *fmrestrict-set vs* (*state-succ s′ a*) = *fmrestrict-set vs s*
using *Cons.prems*(*5*) *3 a agree-restrict-state-succ-idempot*
by *metis*
}
ultimately have *fmrestrict-set vs* (*exec-plan* (*state-succ s′ a*) *as*) = *fmrestrict-set vs s*
using *assms*(*3*) *1 Cons.IH*[**where** *s′*=*state-succ s′ a*]
by *auto*
}
then show *?thesis*
by *simp*
next case *False*
moreover have *exec-plan s′* (*a* # *as*) = *exec-plan s′ as*
using *False*
by (*simp add*: *state-succ-def*)
ultimately show *?thesis*
using *Cons.IH Cons.prems*(*2*, *3*, *5*) *1 2*
by *presburger*
qed
qed *simp*

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*)
using *assms agree-restrict-exec-idempot*
by *fastforce*

lemma *agree-comm*: *agree x x′* = *agree x′ x*
unfolding *agree-def*
by *fastforce*

lemma *restricted-agree-imp-agree*:

assumes (*fmdom′ s2* ⊆ *vs*) (*agree* (*fmrestrict-set vs s1*) *s2*)
shows (*agree s1 s2*)
using *assms contra-subsetD fmlookup-restrict-set Int-iff fmdom′-fmrestrict-set*
unfolding *agree-def*
by *metis*

lemma *agree-imp-submap*:
assumes *f1* $\subseteq_f$ *f2*
shows *agree f1 f2*
using *assms*
unfolding *agree-def*
by (*simp add*: *as-needed-asses-submap-exec-ii*)

lemma *agree-FUNION*:
assumes (*agree fm fm1*) (*agree fm fm2*)
shows (*agree fm* (*fm1* ++ *fm2*))
unfolding *agree-def fmap-add-ltr-def*
using *assms*
by (*metis agree-def fmlookup-add fmlookup-dom′-iff*)

lemma *agree-fm-list-union*:
fixes *fm*
assumes (∀ *fm′*. *ListMem fm′ fmList* ⟶ *agree fm fm′*)
shows (*agree fm* (*foldr fmap-add-ltr fmList fmempty*))
using *assms* **proof** (*induction fmList arbitrary*: *fm*)
case *Nil*
then have *foldr fmap-add-ltr* [] *fmempty* = *fmempty*
  using *Nil*
  by *simp*
then show *?case*
  unfolding *agree-def*
  by *auto*
next
case (*Cons a fmList*)
then have ∀ *fm′*. *ListMem fm′ fmList* ⟶ *agree fm fm′*
  using *Cons.prems insert*
  by *fast*
then have *1*: *agree fm* (*foldr fmap-add-ltr fmList fmempty*)
  using *Cons.IH*
  by *blast*
then have *agree fm a*
  using *Cons.prems elem*
  by *fast*
then have *agree fm* (*a* ++ *foldr fmap-add-ltr fmList fmempty*)
  using *1 agree-FUNION*
  by *blast*

then show *?case*
by *simp*
qed

lemma *DRESTRICT-EQ-AGREE*:
assumes (*fmdom′ s2* ⊆ *vs2*) (*fmdom′ s1* ⊆ *vs1*)
shows ((*fmrestrict-set vs2 s1* = *fmrestrict-set vs1 s2*) ⟶ *agree s1 s2*)
using *assms fmdom′-restrict-set restricted-agree-imp-agree*
by (*metis agree-def*)

lemma *SUBMAPS-AGREE*: (*s1* $\subseteq_f$ *s*) ∧ (*s2* $\subseteq_f$ *s*) ⟹ (*agree s1 s2*)
unfolding *agree-def*
by (*metis as-needed-asses-submap-exec-ii*)

— NOTE name shortened.
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*}
unfolding *snapshot-def*
by *fastforce*

lemma *action-agree-valid-in-snapshot*:
assumes (*a* ∈ *PROB*) (*agree* (*fst a*) *s*) (*agree* (*snd a*) *s*)
shows (*a* ∈ *snapshot PROB s*)
unfolding *snapshot-def*
using *assms*
by *blast*

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*))
using *assms*
proof (*induction as*)
case *Nil*
then show *?case*
using *empty-plan-is-valid*
by *blast*
next
case (*Cons a as*)
{
have ∀ *a*. *ListMem a as* ⟶ *agree* (*fst a*) *s* ∧ *agree* (*snd a*) *s*
using *Cons.prems*(*1*) *insert*

```
      by fast
    moreover have (as ∈ valid-plans PROB)
      using Cons.prems(2) valid-plan-valid-tail
      by fast
    ultimately have set as ⊆ snapshot PROB s
      using Cons.IH valid-plans-def
      by fast
  }
  note 1 = this
  {
    have a: a ∈ PROB
      using Cons.prems(2) valid-plan-valid-head
      by metis
    then have ListMem a (a # as)
      using elem
      by fast
    then have agree (fst a) s ∧ agree (snd a) s
      using Cons.prems(1)
      by blast
    then have a ∈ snapshot PROB s
      using a snapshot-def
      by auto
  }
  then have set (a # as) ⊆ snapshot PROB s
    using 1 set-simps(2)
    by simp
  then show ?case using valid-plans-def
    by blast
qed

lemma fmrestrict-agree-monotonous:
  fixes f g X
  assumes agree f g
  shows agree (fmrestrict-set X f) (fmrestrict-set X g)
proof –
  let ?F=fmdom' (fmrestrict-set X f)
  let ?G=fmdom' (fmrestrict-set X g)
  have 1: ?F = X ∩ fmdom' f ?G = X ∩ fmdom' g
    using fmdom'-fmrestrict-set
    by metis+
  {
    fix v
    assume v ∈ ?F v ∈ ?G
    then have v ∈ fmdom' f v ∈ fmdom' g
      using 1
      by blast+
    then have fmlookup f v = fmlookup g v
      using assms
      unfolding agree-def
```

by *blast*
then have *fmlookup* (*fmrestrict-set X f*) *v* = *fmlookup* (*fmrestrict-set X g*) *v*
unfolding *fmlookup-restrict-set*
by *argo*
}
then show *?thesis*
using *assms*
unfolding *agree-def*
by *blast*
qed

— TODO remove if not used.
lemma *SUBMAP-FUNION-DRESTRICT-i*:
fixes *v vsa vsb f g*
assumes $v \in vsa$
shows
fmlookup (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *f*) *v*
= *fmlookup* (*fmrestrict-set* (*vsa* ∩ *vs*) *f*) *v*

unfolding *fmlookup-restrict-set*
using *assms*
by *auto*

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 −
let *?f=fmrestrict-set* (*vsa* ∪ *vsb*) *fm*
let *?g=fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) (*fma* ++ *fmb*)
have *1*: *?g* = *fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fmb* $++_f$ *fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fma*
unfolding *fmap-add-ltr-def fmrestrict-set-add-distrib*
by *simp*
have *2*: *agree* (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fma*) (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fmb*)
using *assms*(*1*) *fmrestrict-agree-monotonous*
by *blast*
have *3*:
fmdom′ (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fma*) = ((*vsa* ∪ *vsb*) ∩ *vs*) ∩ *fmdom′ fma*
fmdom′ (*fmrestrict-set* ((*vsa* ∪ *vsb*) ∩ *vs*) *fmb*) = ((*vsa* ∪ *vsb*) ∩ *vs*) ∩ *fmdom′ fmb*
using *fmdom′-fmrestrict-set*
by *metis+*
{

```
    fix v
    have fmlookup ?f v = fmlookup ?g v
    proof (cases v ∈ ((vsa ∪ vsb) ∩ vs))
      case True
        — TODO unwrap smt proof.
      then show ?thesis
        using assms(1, 2, 3, 4, 5) 1
         by (smt (verit) IntD1 SUBMAP-FUNION-DRESTRICT-i UnE agree-def
domIff fmdom'.rep-eq fmdom'-alt-def
            fmdom'-fmrestrict-set fmlookup-add fmlookup-restrict-set inf-sup-distrib2
            subset-iff sup-commute)
    next
      case False
      then show ?thesis
      proof -
        have v ∉ vsa ∪ vsb ∨ v ∉ vs
          using False
          by blast
        then have fmlookup (fmrestrict-set (vsa ∪ vsb) fm) v = None
          using assms(4, 5)
          by (metis Int-iff Un-iff fmlookup-restrict-set)
        then show ?thesis
          using False
          by auto
      qed
    qed
  }
  then show ?thesis
    using 1 fmap-ext
    by blast
qed

lemma UNION-FUNION-DRESTRICT-SUBMAP:
  assumes (vs1 ⊆ fmdom' fma) (vs2 ⊆ fmdom' fmb) (agree fma fmb)
    (fmrestrict-set vs1 fma ⊆f s) (fmrestrict-set vs2 fmb ⊆f s)
  shows (fmrestrict-set (vs1 ∪ vs2) (fma ++ fmb) ⊆f s)
proof -
  {
    let ?f=fmrestrict-set (vs1 ∪ vs2) (fma ++ fmb)
    fix v
    assume P: v ∈ fmdom' ?f
    {
      have v ∈ (vs1 ∪ vs2) ∩ (fmdom' fma ∪ fmdom' fmb)
        using P
        unfolding fmap-add-ltr-def fmdom'-fmrestrict-set fmdom'-add
        by force
      then have v ∈ vs1 ∪ vs2 v ∈ fmdom' fma ∪ fmdom' fmb
        by fast+
    }
```

```
    note 1 = this
    then have 2: fmlookup ?f v = fmlookup (fmb ++_f fma) v
      unfolding fmlookup-restrict-set fmap-add-ltr-def
      by argo
    then consider
      (i) v ∈ vs1
      | (ii) v ∈ vs2
      | (iii) ¬v∈ vs1 ∧ ¬v∈vs2
      by blast
    then have fmlookup ?f v = fmlookup s v
    proof (cases)
      case i
      then have v ∈ fmdom' fma
        using assms(1)
        by blast
      then have fmlookup ?f v = fmlookup fma v
        unfolding 2 fmlookup-add
        by (simp add: fmdom'-alt-def)
      also have ... = fmlookup (fmrestrict-set vs1 fma) v
        unfolding fmlookup-restrict-set
        using i
        by simp
      finally show ?thesis
        using assms(4)
      by (metis (mono-tags, lifting) P domIff fmdom'-notI fmsubset.rep-eq map-le-def)
    next
      — TODO unwrap smt proof.
      case ii
      then show ?thesis
        using assms(2, 3, 5) 2 P
        by (smt SUBMAP-FUNION-DRESTRICT-i agree-def
            fmdom'.rep-eq fmdom'-fmrestrict-set fmdom'-notD fmdom'-notI fm-
lookup-add
          fmrestrict-set-dom fmsubset.rep-eq inf.orderE map-le-def subset-Un-eq)
    next
      case iii
      then show ?thesis
        using 1
        by blast
    qed
  }
  then show ?thesis
    by (simp add: as-needed-asses-submap-exec-vii)
qed
```

— TODO unwrap sledgehammered metis proof.

```
lemma agree-DRESTRICT:
  assumes agree s1 s2
```

shows *agree* (*fmrestrict-set vs s1*) (*fmrestrict-set vs s2*)
using *assms* **by** (*fact fmrestrict-agree-monotonous*)

lemma *agree-DRESTRICT-2*:
assumes (*fmdom′ s1* ⊆ *vs1*) (*fmdom′ s2* ⊆ *vs2*) (*agree s1 s2*)
shows (*agree* (*fmrestrict-set vs2 s1*) (*fmrestrict-set vs1 s2*))
using *assms*
unfolding *agree-def fmdom′-restrict-set-precise*
by *auto*

— NOTE added lemma.
lemma *snapshot-eq-filter*:
shows *snapshot PROB s* = *Set.filter* (λ*a*. *agree* (*fst a*) *s* ∧ *agree* (*snd a*) *s*) *PROB*
unfolding *snapshot-def Set.filter-def*
by *presburger*

— NOTE moved up.
corollary *snapshot-subset*:
shows *snapshot PROB s* ⊆ *PROB*
unfolding *snapshot-def*
using *snapshot-eq-filter*
by *blast*

lemma *FINITE-snapshot*:
assumes *finite PROB*
shows *finite* (*snapshot PROB s*)
proof −
have *snapshot PROB s* ⊆ *PROB*
using *snapshot-subset*
by *blast*
then show *?thesis*
using *assms finite-subset*[*of snapshot PROB s PROB*]
by *blast*
qed

— NOTE moved up (declared above the previous lemma). lemma snapshot_subset

— TODO unwrap metis proof.
lemma *dom-proj-snapshot*:
prob-dom (*prob-proj PROB* (*prob-dom* (*snapshot PROB s*))) = *prob-dom* (*snapshot PROB s*)
by (*metis snapshot-subset two-children-parent-mems-le-finite prob-subset-dom-subset*)

lemma *valid-states-snapshot*:
valid-states (*prob-proj PROB* (*prob-dom* (*snapshot PROB s*))) = *valid-states* (*snapshot PROB s*)
by (*metis dom-proj-snapshot valid-states-def*)

lemma *valid-proj-neq-succ-restricted-neq-succ*:

assumes ($x' \in$ *prob-proj PROB vs*) (*state-succ s* $x' \neq s$)
shows (*fmrestrict-set vs* (*state-succ s* x') $\neq$ *fmrestrict-set vs s*)
unfolding *state-succ-def*
using *FDOM-eff-subset-prob-dom-pair dom-prob-proj limited-dom-neq-restricted-neq*
using *assms*(*1*, *2*)
by (*smt dual-order.trans state-succ-def*)

lemma *proj-successors*:
(($\lambda s.$ *fmrestrict-set vs s*) ‘ (*state-successors* (*prob-proj PROB vs*) *s*))
$\subseteq$ (*state-successors* (*prob-proj PROB vs*) (*fmrestrict-set vs s*))

proof −
let *?A*=(($\lambda s.$ *fmrestrict-set vs s*) ‘ (*state-successors* (*prob-proj PROB vs*) *s*))
let *?B*=(*state-successors* (*prob-proj PROB vs*) (*fmrestrict-set vs s*))
{
fix *x*
assume *P*: $x \in$ *?A*
then obtain x' x'' **where** *a*:
$x'' \in$ *prob-proj PROB vs* x' = *state-succ s* x'' $x' \neq s$ x = *fmrestrict-set vs* x'
unfolding *state-successors-def subset-iff*
by *blast*
moreover {
have ($\exists x''$.
$x'' \in$ *prob-proj PROB vs* $\wedge$ x = *state-succ* (*fmrestrict-set vs s*) x''
$\wedge$ $x \neq$ *fmrestrict-set vs s*)
proof (*cases fst* $x'' \subseteq_f$ *s*)
case *true*: *True*
then show *?thesis*
proof (*cases fst* $x'' \subseteq_f$ *fmrestrict-set vs s*)
case *True*
{
have *fmdom'* (*snd* x'') $\subseteq$ *vs*
using *a*(*1*) *FDOM-eff-subset-prob-dom-pair dom-prob-proj dual-order.trans*
by *metis*
then have *fmrestrict-set vs* (*snd* x'') = *snd* x''
using *exec-drest-5*
by *fast*
}
note *i* = *this*
{
have x = *fmrestrict-set vs* (*snd* x'' ++ *s*)
using *a*(*2*, *4*) *true*
unfolding *state-succ-def*
by *simp*
then have x = *fmrestrict-set vs* (*snd* x'') ++ *fmrestrict-set vs s*
unfolding *fmap-add-ltr-def*
using *fmrestrict-set-add-distrib*
by *simp*
then have x = *snd* x'' ++ *fmrestrict-set vs s*

using *i*
by *simp*
then have $x = \textit{state-succ}\ (\textit{fmrestrict-set vs s})\ x''$
unfolding *state-succ-def*
using *True*
by *argo*
}
moreover have $x \neq \textit{fmrestrict-set vs s}$
using *a valid-proj-neq-succ-restricted-neq-succ*
by *fast*
ultimately show *?thesis*
using *a(1)*
by *blast*
next
case *False*
then show *?thesis*
proof −
have $x'' \in (\lambda p.\ \textit{action-proj p vs})$ ' *PROB*
using *calculation(1) prob-proj-def*
by *auto*
then have $\textit{action-proj}\ x''\ vs = x''$
using *action-proj-idempot*
by *blast*
then show *?thesis*
by (*metis* (*no-types*) *False action-proj-pair fmsubset-restrict-set-mono*
fstI
surjective-pairing true)
qed
qed
next
case *False*
then show *?thesis*
proof (*cases* $\textit{fst}\ x'' \subseteq_f \textit{fmrestrict-set vs s}$)
case *True*
then have $\textit{fmdom}'\ (\textit{snd}\ x'') \subseteq vs$
using *FDOM-eff-subset-prob-dom-pair dom-prob-proj*
using *a(1) dual-order.trans*
by *metis*
then have $\textit{fmrestrict-set vs}\ (\textit{snd}\ x'') = \textit{snd}\ x''$
using *exec-drest-5*
by *fast*
then show *?thesis*
unfolding *state-succ-def fmap-add-ltr-def*
using *False True sublist-as-proj-eq-as-1*
by *fast*
next
case *False*
then have $\textit{fmdom}'\ (\textit{fst}\ x'') \subseteq vs$
using *FDOM-pre-subset-prob-dom-pair dom-prob-proj*

using *a*(*1*) *dual-order.trans*
by *metis*
then have *fmrestrict-set vs* (*fst x''*) = *fst x''*
by (*simp add*: *exec-drest-5*)
then show *?thesis*
unfolding *state-succ-def fmap-add-ltr-def*
using *a False fmsubset-restrict-set-mono*
by (*metis state-succ-def*)
qed
qed
}
then obtain *x''* **where** *x''* ∈ *prob-proj PROB vs x* = *state-succ* (*fmrestrict-set vs s*) *x''*
x ≠ *fmrestrict-set vs s*
by *blast*
then have *x* ∈ *?B* **unfolding** *state-successors-def*
by *blast*
}
then show *?thesis*
by *blast*
qed

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*))
using *proj-successors*
by *force*

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*)
using *assms*
proof −
{
obtain *p' e'* **where** (*p'*, *e'*) ∈ *PROB* (*p*, *e*) = *action-proj* (*p'*, *e'*) *vs*
using *assms*(*1*)
unfolding *prob-proj-def*
by *fast*
then have *fmdom' e* ⊆ *prob-dom* (*prob-proj PROB vs*)
using *assms FDOM-eff-subset-prob-dom*
by *blast*
also have . . . = *prob-dom PROB* ∩ *vs*
using *graph-plan-neq-mems-state-set-neq-len*
by *fast*
finally have *fmdom' e* ⊆ *prob-dom PROB*
by *simp*
}

moreover have *fmdom′ s = prob-dom PROB*
using *assms(2)*
unfolding *valid-states-def*
by *simp*
ultimately show *?thesis*
by *simp*
qed

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 −
have *fmdom′* (*snd h*) ⊆ *fmdom′ s*
using *assms proj-FDOM-eff-subset-FDOM-valid-states surjective-pairing*
by *metis*
moreover have *fmdom′* (*state-succ s h*) = *fmdom′ s*
using *calculation(1) FDOM-state-succ*
by *metis*
ultimately show *?thesis*
using *assms(2) valid-states-def*
by *blast*
qed

lemma *proj-successors-of-valid-are-valid*:
assumes (*s* ∈ *valid-states PROB*)
shows (*state-successors* (*prob-proj PROB vs*) *s* ⊆ (*valid-states PROB*))
unfolding *state-successors-def*
using *assms valid-proj-action-valid-succ*
by *blast*

7.3 State Space Projection

definition *ss-proj* **where**
ss-proj ss vs ≡ (*λs. 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 −
— NOTE Proof by mutual inclusion.
let *?lhs* = *fmrestrict-set* (*X* ∩ *Y*) ' *A*
let *?rhs* = (*fmrestrict-set X* ∘ *fmrestrict-set Y*) ' *A*
{
fix *a*
assume *a* ∈ *A*
have (*fmrestrict-set X* ∘ *fmrestrict-set Y*) *a* = *fmrestrict-set X* (*fmrestrict-set Y a*)

```
      by auto
    also have ... = fmrestrict-set (X ∩ Y) a
      using action-proj-inter-i
      by fast
    finally have (fmrestrict-set X ∘ fmrestrict-set Y) a = fmrestrict-set (X ∩ Y)
a
      by auto
  }
  note 1 = this
  {
    fix a
    assume P: a ∈ A
    then have fmrestrict-set (X ∩ Y) a ∈ ?lhs
      by simp
    moreover have (fmrestrict-set X ∘ fmrestrict-set Y) a ∈ ?rhs
      using P
      by blast
    ultimately have
     fmrestrict-set (X ∩ Y) a ∈ ?rhs (fmrestrict-set X ∘ fmrestrict-set Y) a ∈ ?lhs
      using P 1
      by metis+
  }
  then show ?thesis
    by blast
qed

lemma invariantStateSpace-thm-9:
  fixes ss vs1 vs2
  shows ss-proj ss (vs1 ∩ vs2) = ss-proj (ss-proj ss vs2) vs1
proof −
  {
    have
      ss-proj ss (vs1 ∩ vs2)
      = fmrestrict-set (vs1 ∩ vs2) ‘ ss

      unfolding ss-proj-def
      by simp
    also have ... = (fmrestrict-set vs1 ∘ fmrestrict-set vs2) ‘ ss
      using fmrestrict-set-inter-img
      by metis
    finally have ss-proj ss (vs1 ∩ vs2) = ss-proj (ss-proj ss vs2) vs1
      unfolding ss-proj-def
      by force
  }
  then show ?thesis
    by simp
qed

lemma FINITE-ss-proj:
```

fixes *ss vs*
assumes *finite ss*
shows *finite* (*ss-proj ss vs*)
unfolding *ss-proj-def*
using *assms*
by *simp*

lemma *nempty-stateSpace-nempty-ss-proj*:
assumes ($ss \neq \{\}$)
shows ($ss\text{-}proj\ ss\ vs \neq \{\}$)
unfolding *ss-proj-def*
using *assms*
by *simp*

lemma *invariantStateSpace-thm-5*:
fixes *ss vs domain*
assumes (*stateSpace ss domain*)
shows (*stateSpace* (*ss-proj ss vs*) ($domain \cap vs$))
using *assms*
unfolding *stateSpace-def ss-proj-def*
by (*metis* (*no-types*, *lifting*) *fmdom′-fmrestrict-set imageE inf-commute*)

lemma *dom-subset-ssproj-eq-ss*:
fixes *ss domain vs*
assumes (*stateSpace ss domain*) ($domain \subseteq vs$)
shows ($ss\text{-}proj\ ss\ vs = ss$)
unfolding *ss-proj-def stateSpace-def*
using *assms exec-drest-5*
by (*metis* (*mono-tags*, *lifting*) *image-cong image-ident stateSpace-def*)

— TODO refactor duplicate proof steps in case split.
lemma *neq-vs-neq-ss-proj*:
fixes *vs*
assumes ($ss \neq \{\}$) (*stateSpace ss vs*) ($vs1 \subseteq vs$) ($vs2 \subseteq vs$) ($vs1 \neq vs2$)
shows ($ss\text{-}proj\ ss\ vs1 \neq ss\text{-}proj\ ss\ vs2$)
proof −
{
have *1*: $\exists f.\ f \in ss$
using *assms*(*1*)
by *blast*
then obtain *x* **where** ($x \in vs1 \wedge x \notin vs2$) $\vee$ ($x \in vs2 \wedge x \notin vs1$)
using *assms*(*5*)
by *blast*
then consider (*i*) $x \in vs1 \wedge x \notin vs2$ | (*ii*) $x \in vs2 \wedge x \notin vs1$
by *blast*
then have *fmrestrict-set vs1* ‘ *ss* ≠ *fmrestrict-set vs2* ‘ *ss* **proof** (*cases*)
case *i*
{
fix *s′ t′*

assume $s' \in$ *fmrestrict-set vs1* ‘ *ss* $t' \in$ *fmrestrict-set vs2* ‘ *ss*
then obtain s t **where** a:
$s \in ss$ $s' =$ *fmrestrict-set vs1* s $t \in ss$ $t' =$ *fmrestrict-set vs2* t
by *blast*
then have *fmdom'* $s = vs$
using *assms*(*2*)
by (*simp add*: *stateSpace-def*)
then have b: *fmdom'* $s' = vs1$
using *assms*(*3*) *a fmdom'-fmrestrict-set inf.order-iff*
by *metis*
then have *fmdom'* $t = vs$
using *assms*(*2*) *a*(*3*)
by (*simp add*: *stateSpace-def*)
then have *fmdom'* $t' = vs2$
using *assms*(*4*) *a*(*4*) *fmdom'-fmrestrict-set inf.order-iff*
by *metis*
then have *fmlookup* s' $x \neq$ *None* *fmlookup* t' $x =$ *None*
using *i b domIff fmdom'-alt-def fmdom.rep-eq*
by *metis+*
then have $s' \neq t'$
by *blast*
}
then show *?thesis*
using *1 neq-funs-neq-images*
by *blast*
next
case *ii*
{
fix s' t'
assume $s' \in$ *fmrestrict-set vs1* ‘ *ss* $t' \in$ *fmrestrict-set vs2* ‘ *ss*
then obtain s t **where** c:
$s \in ss$ $s' =$ *fmrestrict-set vs1* s $t \in ss$ $t' =$ *fmrestrict-set vs2* t
by *blast*
then have *fmdom'* $s = vs$
using *assms*(*2*)
by (*simp add*: *stateSpace-def*)
then have d: *fmdom'* $s' = vs1$
using *assms*(*3*) *c*(*2*) *fmdom'-fmrestrict-set inf.order-iff*
by *metis*
then have *fmdom'* $t = vs$
using *assms*(*2*) *c*(*3*)
by (*simp add*: *stateSpace-def*)
then have *fmdom'* $t' = vs2$
using *assms*(*4*) *c*(*4*) *fmdom'-fmrestrict-set inf.order-iff*
by *metis*
then have *fmlookup* s' $x =$ *None* *fmlookup* t' $x \neq$ *None*
using *ii d domIff fmdom'-alt-def fmdom.rep-eq*
by *metis+*
then have $s' \neq t'$

```
        by blast
      }
      then show ?thesis
        using 1 neq-funs-neq-images
        by blast
    qed
  }
  then show ?thesis
    unfolding ss-proj-def
    by blast
qed

lemma subset-dom-stateSpace-ss-proj:
  fixes vs1 vs2
  assumes (vs1 ⊆ vs2) (stateSpace ss vs2)
  shows (stateSpace (ss-proj ss vs1) vs1)
  using assms
  by (metis inf.absorb-iff2 invariantStateSpace-thm-5)

lemma card-proj-leq:
  assumes finite PROB
  shows card (prob-proj PROB vs) ≤ card PROB
  unfolding prob-proj-def
  using assms card-image-le
  by blast

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 (λx. ¬R x h) l ∧ top-sorted-abs R l)


lemma top-sorted-abs-mem:
```

assumes (*top-sorted-abs R* (*h* # *l*)) (*ListMem x l*)
shows (¬ *R x h*)
using *assms*
by (*auto simp add*: *ListMem-iff list.pred-set*)

lemma *top-sorted-cons*:
assumes *top-sorted-abs R* (*h* # *l*)
shows (*top-sorted-abs R l*)
using *assms*
by *simp*

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*)
using *assms*
unfolding *geq-arg-def*
proof (*induction l arbitrary*: *R w g f x*)
case (*Cons a l*)
then show *?case*
using *Cons.IH Cons.prems*
proof (*cases R x a*)
case *True*

```
    then show ?thesis
      using Cons le-trans wlp.simps(2)
      by smt
  next
    case False
    then show ?thesis
      using Cons
      by simp
  qed
qed simp
```

— NOTE Types of 'f' and 'g' had to be fixed to be able to use transitivity rule of the less-equal relation.

```
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))
  using assms
  unfolding geq-arg-def
proof (induction G arbitrary: vtx1 f g R vtx2)
  case Nil
  then show ?case
    using ListMem-iff
    by fastforce
next
  case (Cons a G)
  show ?case
  proof (auto)
    assume P1: R vtx1 a R vtx2 a
    then show
        f (w vtx2) (g (f (w vtx1) (wlp R w g f a G)) (wlp R w g f vtx1 G))
        ≤ g (f (w vtx2) (wlp R w g f a G)) (wlp R w g f vtx2 G)
      using Cons.prems(3, 5, 6)
      by (metis ListMem-iff set-ConsD top-sorted-abs-mem)
  next
    assume P2: R vtx1 a ¬ R vtx2 a
    then show
        f (w vtx2) (g (f (w vtx1) (wlp R w g f a G)) (wlp R w g f vtx1 G))
        ≤ wlp R w g f vtx2 G
      using Cons.prems(4, 5, 6)
      by (metis ListMem-iff set-ConsD top-sorted-abs-mem)
  next
    assume P3: ¬ R vtx1 a R vtx2 a
    then show
        f (w vtx2) (wlp R w g f vtx1 G)
        ≤ g (f (w vtx2) (wlp R w g f a G)) (wlp R w g f vtx2 G)
    proof -
```

have *f1*: ∀ *n na*. *n* ≤ *g n na* ∧ *na* ≤ *g n na*
using *Cons.prems*(*2*) **by** *blast*
have *f2*: *vtx1* = *a* ∨ *vtx1* ∈ *set G*
by (*meson Cons.prems*(*5*) *ListMem-iff set-ConsD*)
obtain *aa* :: (′*a* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a* **where**
∀ *x2*. (∃ *v5*. *ListMem v5 G* ∧ *x2 v5 v5*) = (*ListMem* (*aa x2*) *G* ∧ *x2* (*aa x2*) (*aa x2*))
by *moura*
then have
ListMem (*aa R*) *G* ∧ *R* (*aa R*) (*aa R*)
∨ ¬ *ListMem vtx1 G* ∨ *f* (*w vtx2*) (*wlp R w g f vtx1 G*) ≤ *wlp R w g f vtx2 G*
using *f1* **by** (*metis* (*no-types*) *Cons.IH Cons.prems*(*1*, *4*, *6*) *top-sorted-cons*)
then show *?thesis*
using *f2 f1* **by** (*meson Cons.prems*(*3*) *ListMem-iff insert le-trans*)
qed
next
assume *P4*: ¬ *R vtx1 a* ¬ *R vtx2 a*
then show *f* (*w vtx2*) (*wlp R w g f vtx1 G*) ≤ *wlp R w g f vtx2 G*
proof −
have *f1*: *top-sorted-abs R G*
using *Cons.prems*(*6*) **by** *fastforce*
have *ListMem vtx1 G*
by (*metis Cons.prems*(*4*) *Cons.prems*(*5*) *ListMem-iff P4*(*2*) *set-ConsD*)
then show *?thesis*
using *f1* **by** (*simp add*: *Cons.IH Cons.prems*(*1*, *2*, *3*, *4*) *insert*)
qed
qed
qed

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*)

unfolding *increasing-def*
by (*induction l*) (*auto simp add*: *elem insert*)

— NOTE generalizing ‘f2‘, ‘x1‘, ‘x2‘ seems to break the prover.
lemma *wlp-congruence-rule*:
fixes *l1 l2 R1 R2 w1 w2 g1 g2 f1 f2 x1 x2*

```
  assumes (l1 = l2) (∀ y. ListMem y l2 ⟶ (R1 x1 y = R2 x2 y))
    (∀ y. ListMem y l2 ⟶ (R1 y x1 = R2 y x2)) (w1 x1 = w2 x2)
    (∀ y1 y2. (y1 = y2) ⟶ (f1 (w1 x1) y1 = f2 (w2 x2) y2))
    (∀ y1 y2 z1 z2. (y1 = y2) ∧ (z1 = z2) ⟶ ((g1 (f1 (w1 x1) y1) z1) = (g2 (f2
(w2 x2) y2) z2)))
    (∀ x y. ListMem x l2 ∧ ListMem y l2 ⟶ (R1 x y = R2 x y))
    (∀ x. ListMem x l2 ⟶ (w1 x = w2 x))
    (∀ x y z. ListMem x l2 ⟶ (g1 (f1 (w1 x) y) z = g2 (f2 (w2 x) y) z))
    (∀ x y. ListMem x l2 ⟶ (f1 (w1 x) y = f2 (w1 x) y))
  shows ((wlp R1 w1 g1 f1 x1 l1) = (wlp R2 w2 g2 f2 x2 l2))
  using assms
proof (induction l2 arbitrary: l1 x1 x2)
  case (Cons a l2)
  then have (wlp R1 w1 g1 f1 x1 l2) = (wlp R2 w2 g2 f2 x2 l2)
    using Cons
    by (simp add: insert)
  moreover have (wlp R1 w1 g1 f1 a l2) = (wlp R2 w2 g2 f2 a l2)
    using Cons
    by (simp add: elem insert)
  ultimately show ?case
    by (simp add: Cons.prems(1,2, 6) elem)
qed auto


lemma wlp-ite-weights:
  fixes x
  assumes ∀ y. ListMem y l1 ⟶ P y P x
  shows ((wlp R (λy. if P y then w1 y else w2 y) g f x l1) = (wlp R w1 g f x l1))
  using assms

proof (induction l1 arbitrary: R P w1 w2 f g)
  case (Cons a l1)
  let ?w1=(λy. if P y then w1 y else w2 y)
  let ?w2=w1
  {
    have ∀ y. ListMem y l1 ⟶ P y
      using Cons.prems(1) insert
      by fast
    then have ((wlp R (λy. if P y then w1 y else w2 y) g f x l1) = (wlp R w1 g f
x l1))
      using Cons.prems(2) Cons.IH
      by blast
  }
  note 1 = this
  {
    have (if P x then w1 x else w2 x) = w1 x
      ∀ y1 y2. y1 = y2 ⟶ f (if P x then w1 x else w2 x) y1 = f (w1 x) y2
      ∀ y1 y2 z1 z2.
        y1 = y2 ∧ z1 = z2
```

⟶ g (f (*if* P x *then* *w1* x *else* *w2* x) *y1*) *z1* = g (f (*w1* x) *y2*) *z2*
∀ x. *ListMem* x (a # *l1*) ⟶ (*if* P x *then* *w1* x *else* *w2* x) = *w1* x
∀ x y z.
ListMem x (a # *l1*)
⟶ g (f (*if* P x *then* *w1* x *else* *w2* x) y) z = g (f (*w1* x) y) z
∀ x y.
ListMem x (a # *l1*) ⟶ f (*if* P x *then* *w1* x *else* *w2* x) y = f (*if* P x *then* *w1* x *else* *w2* x) y
using *Cons.prems*(*1*, *2*)
by *simp+*
then have *wlp* R (λy. *if* P y *then* *w1* y *else* *w2* y) g f x (a # *l1*) = *wlp* R *w1* g f x (a # *l1*)
using *Cons* *wlp-congruence-rule*[*of* a # *l1* a # *l1* R x R x *?w1* *?w2* f f g g]
by *blast*
}
then show *?case*
by *blast*
qed *auto*

lemma *map-wlp-ite-weights*:
(∀ x. *ListMem* x *l1* ⟶ P x)
⟹ (∀ x. *ListMem* x *l2* ⟶ P x)
⟹ (
map (λx. *wlp* R (λy. *if* P y *then* *w1* y *else* *w2* y) g f x *l1*) *l2*
= *map* (λx. *wlp* R *w1* g f x *l1*) *l2*
)

apply(*induction* *l2*)
apply(*auto*)
subgoal by (*simp add*: *elem* *wlp-congruence-rule*)
subgoal by (*simp add*: *insert*)
done

lemma *wlp-weight-lamda-exp*: ⋀x. *wlp* R w g f x l = *wlp* R (λy. w y) g f x l
proof −
fix x
show *wlp* R w g f x l = *wlp* R (λy. w y) g f x l
by(*induction* l) *auto*
qed

lemma *img-wlp-ite-weights*:
(∀ x. *ListMem* x l ⟶ P x)
⟹ (∀ x. x ∈ s ⟶ P x)
⟹ (
(λx. *wlp* R (λy. *if* P y *then* *w1* y *else* *w2* y) g f x l) ‘ s
= (λx. *wlp* R *w1* g f x l) ‘ s

)

proof –
assume *P1*: ∀ *x*. *ListMem x l* ⟶ *P x*
assume *P2*: ∀ *x*. *x* ∈ *s* ⟶ *P x*
show (
(λ*x*. *wlp R* (λ*y*. *if P y then w1 y else w2 y*) *g f x l*) ' *s*
= (λ*x*. *wlp R w1 g f x l*) ' *s*
)
by (*auto simp add*: *P1 P2 image-iff wlp-ite-weights*)
qed

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*))

by (*induction as1 arbitrary*: *s as2 vs*) *auto*

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*)
using *assms*
proof (*induction as arbitrary*: *s vs*)
case (*Cons a as*)
then show *?case*
proof (*cases fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s*)
case *True*
— NOTE This case violates the induction premise *vars-change* (*a* # *as*) *vs s* = [] since the empty list is impossible.
then have *state-succ s a* # *vars-change as vs* (*state-succ s a*) = []
using *Cons.prems True*
by *simp*
then show *fmrestrict-set vs* (*exec-plan s* (*a* # *as*)) = *fmrestrict-set vs s*
by *blast*
next
case *False*
then have *vars-change as vs* (*state-succ s a*) = []
using *Cons.prems False*
by *force*
then have
fmrestrict-set vs (*exec-plan* (*state-succ s a*) *as*) = *fmrestrict-set vs* (*state-succ s a*)
using *Cons.IH*[*of vs* (*state-succ s a*)]
by *blast*
then show *fmrestrict-set vs* (*exec-plan s* (*a* # *as*)) = *fmrestrict-set vs s*
using *False*
by *simp*
qed
qed *auto*

— NOTE renamed variable 'a' to 'b' to not conflict with naming for list head in induction step.
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′*)
using *assms*
proof (*induction as arbitrary*: *s s′ vs b*)
case (*Cons a as*)
— NOTE Having either *fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s*

or ¬ *ListMem b as* leads to simpler propositions so we split here.
  then show (*fmrestrict-set vs* (*snd b*) $\subseteq_f$ *fmrestrict-set vs s'*)
    using *Cons.prems*(*1*)
  proof (*cases fmrestrict-set vs* (*state-succ s a*) = *fmrestrict-set vs s* ∧ *ListMem b as*)
    case *True*
    let *?s=state-succ s a*
    have *vars-change as vs ?s* = []
      using *True Cons.prems*(*1*)
      by *auto*
    moreover have *sat-precond-as ?s as*
      using *Cons.prems*(*2*) *sat-precond-as.simps*(*2*)
      by *blast*
    ultimately show *?thesis*
      using *True Cons.prems*(*4*) *Cons.IH*
      by *auto*
  next
    case *False*
    then consider
      (*i*) *fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s*
      | (*ii*) ¬*ListMem b as*
      by *blast*
    then show *?thesis*
      using *Cons.prems*(*1*)
    proof (*cases*)
      case *ii*
      then have *a* = *b*
        using *Cons.prems*(*3*) *ListMem-iff set-ConsD*
        by *metis*
        — NOTE Mysteriously sledgehammer finds a proof here while the premises of ‘no_change_vs_eff_submap‘ cannot be proven individually.
      then show *?thesis*
        using *Cons.prems*(*1*, *2*, *4*) *no-change-vs-eff-submap*
        by (*metis list.distinct*(*1*) *sat-precond-as.simps*(*2*) *vars-change.simps*(*2*))
    qed *simp*
  qed
qed (*simp add*: *ListMem-iff*)

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'*)
  using *assms*
proof (*induction as arbitrary*: *vs s s'*)
  case (*Cons a as*)
    — NOTE Having either *fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s* or ¬ *ListMem b as* leads to simpler propositions so we split here.

```
  then show (fmrestrict-set vs (fst b) ⊆f fmrestrict-set vs s')
    using Cons.prems(1)
  proof (cases fmrestrict-set vs (state-succ s a) = fmrestrict-set vs s ∧ ListMem b
as)
    case True
    let ?s=state-succ s a
    have vars-change as vs ?s = []
      using True Cons.prems(1)
      by auto
    moreover have sat-precond-as ?s as
      using Cons.prems(2) sat-precond-as.simps(2)
      by blast
    ultimately show ?thesis
      using True Cons.prems(4) Cons.IH
      by auto
  next
    case False
    then consider
      (i) fmrestrict-set vs (state-succ s a) ≠ fmrestrict-set vs s
      | (ii) ¬ListMem b as
      by blast
    then show ?thesis
      using Cons.prems(1)
    proof (cases)
      case ii
      then have a = b
        using Cons.prems(3) ListMem-iff set-ConsD
        by metis
      then show ?thesis
        using Cons.prems(2, 4) fmsubset-restrict-set-mono
        by (metis sat-precond-as.simps(2))
    qed simp
  qed
qed (simp add: ListMem-iff)


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 −
  {
    fix a
    assume P: ListMem a as
    then have agree (fst a) (fmrestrict-set vs s)
      by (metis agree-imp-submap assms(2) assms(3) fmdom'-restrict-set
          restricted-agree-imp-agree zero-change-imp-all-preconds-submap)
    moreover have agree (snd a) (fmrestrict-set vs s)
     by (metis (no-types) P agree-imp-submap assms(2) assms(3) fmdom'-restrict-set
```

restricted-agree-imp-agree zero-change-imp-all-effects-submap)
ultimately have *agree* (*fst a*) (*fmrestrict-set vs s*) *agree* (*snd a*) (*fmrestrict-set vs s*)
by *simp+*
}
then show *?thesis*
using *assms*(*1*) *as-mem-agree-valid-in-snapshot*
by *blast*
qed

— NOTE type of 'PROB' had to be fixed for 'problem_plan_bound_works'.
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 −
let *?s*=(*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) *s*)
let *?PROB*=(*snapshot PROB* (*fmrestrict-set vs s*))
{
have *finite* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*1*) *FINITE-snapshot*
by *blast*
}
moreover {
have
fmrestrict-set (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) *s*
∈ *valid-states* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*4*) *graph-plan-not-eq-last-diff-paths valid-states-snapshot*
by *blast*
}
moreover {
have *as* ∈ *valid-plans* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*2*, *3*, *5*) *no-vs-change-valid-in-snapshot*
by *blast*
}
ultimately show *?thesis*
using *problem-plan-bound-works*[*of ?PROB ?s as*]
by *blast*

qed

— NOTE type of 'PROB' had to be fixed for 'no_vs_change_obtain_snapshot_bound_1st_step'.
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 −
obtain *as''* **where** *1*:

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))
using *assms no-vs-change-obtain-snapshot-bound-1st-step*
by *blast*
let *?s'*=(*fmrestrict-set* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) *s*)
let *?as'*=*rem-condless-act ?s'* [] *as''*
have *exec-plan ?s' as* = *exec-plan ?s' as''*
using *1*(*1*) *rem-condless-valid-1*
by *blast*
moreover have *subseq ?as' as*
using *1*(*2*) *rem-condless-valid-8 sublist-trans*
by *blast*
moreover have *sat-precond-as s ?as'*
using *sat-precond-drest-sat-precond rem-condless-valid-2*
by *fast*
moreover have (*length ?as'* ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set*
vs s)))
using *1 rem-condless-valid-3 le-trans*
by *blast*
ultimately show *?thesis*
using *1 rem-condless-valid-1*
by *auto*
qed

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*) ($s \in$ *valid-states PROB*) (*as* $\in$ *valid-plans PROB*)
shows ($\exists$ *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'*)
)
$\wedge$ (*subseq as' as*)
$\wedge$ (*length as'* $\leq$ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)
proof –
obtain *as'* :: (('*a*, *bool*) *fmap* $\times$ ('*a*, *bool*) *fmap*) *list* **where**
(
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' $\leq$ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*1*, *2*, *4*, *5*, *6*) *no-vs-change-obtain-snapshot-bound-2nd-step*
by *blast*
moreover have
exec-plan (*fmrestrict-set vs s*) (*as-proj as vs*) = *fmrestrict-set vs* (*exec-plan s as*)
using *assms*(*4*) *sat-precond-exec-as-proj-eq-proj-exec*
by *blast*
moreover have *as-proj as* (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) = *as*
using *assms*(*2*, *3*, *4*, *6*) *as-proj-eq-as no-vs-change-valid-in-snapshot*
by *blast*
ultimately show *?thesis*
using *sublist-as-proj-eq-as proj-exec-proj-eq-exec-proj'*
by *metis*
qed

— NOTE added lemma.
— TODO remove unused assumptions.
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*) ($s \in$ *valid-states PROB*) (*as* $\in$ *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' $\leq$ *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
$\wedge$ *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 $-$
let *?vs*=(*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
let *?vs'*=(*fmdom'* (*exec-plan s as*) $-$ *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
let *?vs''*=(*fmdom'* (*exec-plan s as'*) $-$ *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))
let *?s*=(*exec-plan s as*)
let *?s'*=(*exec-plan s as'*)
have *1*: *as* $\in$ *valid-plans* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*2*, *4*, *6*) *no-vs-change-valid-in-snapshot*
by *blast*
{
{
fix *a*
assume *ListMem a as*
then have *fmdom'* (*snd a*) $\subseteq$ *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))
using *1 FDOM-eff-subset-prob-dom-pair valid-plan-mems*
by *metis*
then have *fmdom'* (*fmrestrict-set* (*fmdom'* (*exec-plan s as*)
$-$ *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*snd a*))
= {}
using *subset-inter-diff-empty*[*of fmdom'* (*snd a*)
prob-dom (*snapshot PROB* (*fmrestrict-set vs s*))] *fmdom'-restrict-set-precise*
by *metis*
}
then have
fmrestrict-set ?vs' (*exec-plan s as*) = *fmrestrict-set ?vs' s*
using *disjoint-effects-no-effects*[*of as ?vs' s*]
by *blast*
}
moreover {

{
fix *a*
assume *P*: *ListMem a as′*
moreover have *α*: *as′* ∈ *valid-plans* (*snapshot PROB* (*fmrestrict-set vs s*))
using *assms*(*8*) *1 sublist-valid-plan*
by *blast*
moreover have *a* ∈ *PROB*
using *P α snapshot-subset subsetCE valid-plan-mems*
by *fast*
ultimately have *fmdom′* (*snd a*) ⊆ *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))
using *FDOM-eff-subset-prob-dom-pair valid-plan-mems*
by *metis*
then have *fmdom′* (*fmrestrict-set* (*fmdom′* (*exec-plan s as′*)
− *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*snd a*))
= {}
using *subset-inter-diff-empty*[*of fmdom′* (*snd a*)
prob-dom (*snapshot PROB* (*fmrestrict-set vs s*))] *fmdom′-restrict-set-precise*
by *metis*
}
then have
fmrestrict-set ?vs″ (*exec-plan s as′*) = *fmrestrict-set ?vs″ s*
using *disjoint-effects-no-effects*[*of as′ ?vs″ s*]
by *blast*
}
ultimately show *?thesis*
by *blast*
qed

— NOTE type for ‘PROB‘ had to be fixed.
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 −
obtain *as′* **where** *1*:
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*))
using *assms no-vs-change-obtain-snapshot-bound-3rd-step*
by *blast*

{

have *a*: *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
using *assms 1 no-vs-change-snapshot-s-vs-is-valid-bound-i*
by *blast+*
moreover have *as′* ∈ *valid-plans* (*snapshot PROB* (*fmrestrict-set vs s*))
using *1*(*2*) *assms*(*2*) *assms*(*4*) *assms*(*6*) *no-vs-change-valid-in-snapshot sublist-valid-plan*
by *blast*
moreover have (*exec-plan s as*) ∈ *valid-states PROB*
using *assms*(*5*, *6*) *valid-as-valid-exec*
by *blast*
moreover have (*exec-plan s as′*) ∈ *valid-states PROB*
using *assms*(*5*, *6*) *1 valid-as-valid-exec sublist-valid-plan*
by *blast*
ultimately have *exec-plan s as* = *exec-plan s as′*
using *assms*
unfolding *valid-states-def*
using *graph-plan-lemma-5*[**where** *vs*=*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)), *OF* - *1*(*1*)]
by *force*
}
then show *?thesis*
using *1*
by *blast*
qed

— TODO showcase (problems with stronger typing: Isabelle requires strict typing for ‘max‘; whereas in HOL4 this is not required, possible because ’MAX’ is natural number specific.

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

proof −
have *geq-arg* (*max* :: *nat* ⇒ *nat* ⇒ *nat*)

unfolding *geq-arg-def*
using *max.cobounded1*
by *simp*
then show *?thesis*
unfolding *S-def*
using *individual-weight-less-eq-lp*[**where**
g=max :: nat ⇒ nat ⇒ nat
and *x=(fmrestrict-set vs s)* **and** *R=(λx y. y ∈ state-successors (prob-proj PROB vs) x)*
and *w=(λs. problem-plan-bound (snapshot PROB s))* **and** *f=(λx y. x + y + 1)* **and** *l=lss*]
by *blast*
qed

— NOTE first argument of 'top_sorted_abs' had to be wrapped into lambda.
— NOTE the type of '1' had to be restricted to 'nat' to ensure the proofs for 'geq_arg' work.

lemma *S-geq-S-succ-plus-ell*:
assumes *(s ∈ valid-states PROB)*
(top-sorted-abs (λx y. y ∈ state-successors (prob-proj PROB vs) x) lss)
(s′ ∈ 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 −
let *?f=λx y. x + y + (1 :: nat)*
let *?R=(λx y. y ∈ state-successors (prob-proj PROB vs) x)*
let *?w=(λs. problem-plan-bound (snapshot PROB s))*
let *?g=max :: nat ⇒ nat ⇒ nat*
let *?vtx1=(fmrestrict-set vs s′)*
let *?G=lss*
let *?vtx2=(fmrestrict-set vs s)*
have *geq-arg ?f*
unfolding *geq-arg-def*
by *simp*
moreover have *geq-arg ?g*
unfolding *geq-arg-def*
by *simp*
moreover have *∀ x. ListMem x lss ⟶ ¬?R x x*
unfolding *state-successors-def*
by *blast*
moreover have *?R ?vtx2 ?vtx1*
unfolding *state-successors-def*
using *assms(3) state-in-successor-proj-in-state-in-successor state-successors-def*

by *blast*
moreover have
ListMem ?vtx1 ?G
using *assms*(*1*, *3*, *4*)
by (*metis ListMem-iff contra-subsetD graph-plan-not-eq-last-diff-paths proj-successors-of-valid-are-valid*)
moreover have *top-sorted-abs ?R ?G*
using *assms*(*2*)
by *simp*
ultimately show *?thesis*
unfolding *S-def*
using *lp-geq-lp-from-successor*[*of ?f ?g ?G ?R ?vtx2 ?vtx1 ?w*]
by *blast*
qed

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*)
)
using *assms*
proof (*induction as arbitrary*: $s\ s'$ *vs ss*)
case (*Cons a as*)
then show *?case*
proof (*cases fmrestrict-set vs* (*state-succ s a*) ≠ *fmrestrict-set vs s*)
case *True*
then have *state-succ s a* = s' *vars-change as vs* (*state-succ s a*) = *ss*
using *Cons.prems*
by *simp+*
then show *?thesis*
by *fastforce*
next
case *False*
then have *vars-change as vs* (*state-succ s a*) = s' # *ss*
using *Cons.prems*
by *simp*
then obtain *as1 act as2* **where**
as = *as1* @ *act* # *as2* *vars-change as1 vs* (*state-succ s a*) = []
state-succ (*exec-plan* (*state-succ s a*) *as1*) *act* = s'
vars-change as2 vs (*state-succ* (*exec-plan* (*state-succ s a*) *as1*) *act*) = *ss*
using *Cons.IH*
by *blast*
then show *?thesis*
by (*metis False append-Cons exec-plan.simps*(*2*) *vars-change.simps*(*2*))
qed

qed *simp*

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*)
using *assms*
apply(*induction as arbitrary*: *s s′ vs ss*)
apply(*auto*)
by (*metis list.inject*)

— NOTE first argument of 'top_sorted_abs had to be wrapped into lambda.
lemma *problem-plan-bound-S-bound-1st-step*:
fixes *PROB* :: *′a problem*
assumes *finite PROB* (*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*) (*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*))
)
using *assms*
proof (*induction vars-change as vs s arbitrary*: *PROB as vs s lss*)
case *Nil*
then obtain *as′* **where**
exec-plan s as = *exec-plan s as′ subseq as′ as*
length as′ ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*))
using *Nil*(*1*) *Nil.prems*(*1*,*4*,*5*,*6*,*7*) *no-vs-change-snapshot-s-vs-is-valid-bound*
by *metis*
moreover have
problem-plan-bound (*snapshot PROB* (*fmrestrict-set vs s*))
≤ *S vs lss PROB* (*fmrestrict-set vs s*)

using *snapshot-bound-leq-S le-trans*
by *fast*
ultimately show *?case*
using *le-trans*
by *fastforce*
next
case (*Cons s′ ss*)
then obtain *as1 act as2* **where** *1*:
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*
using *vars-change-cons*

```
  by smt
```

Obtain conclusion of induction hypothesis for 'as2' and '(state_succ (exec_plan s as1) act)'.

```
 {
  {
   have as1 ∈ valid-plans PROB
    using Cons.prems(5) 1(1) valid-append-valid-pref
    by blast
   moreover have act ∈ PROB
    using Cons.prems(5) 1 valid-append-valid-suff valid-plan-valid-head
    by fast
   ultimately have state-succ (exec-plan s as1) act ∈ valid-states PROB
    using Cons.prems(4) valid-as-valid-exec lemma-1-i
    by blast
  }
  moreover have as2 ∈ valid-plans PROB
   using Cons.prems(5) 1(1) valid-append-valid-suff valid-plan-valid-tail
   by fast
  moreover have no-effectless-act as2
   using Cons.prems(6) 1(1) rem-effectless-works-13 sublist-append-back
   by blast
  moreover have sat-precond-as (state-succ (exec-plan s as1) act) as2
   using Cons.prems(7) 1(1) graph-plan-lemma-17 sat-precond-as.simps(2)
   by blast
  ultimately have ∃ as'.
      exec-plan (state-succ (exec-plan s as1) act) as'
      = exec-plan (state-succ (exec-plan s as1) act) as2
     ∧ subseq as' as2
     ∧ length as' ≤ S vs lss PROB (fmrestrict-set vs (state-succ (exec-plan s as1)
act))
   using Cons.prems(1, 2, 3) 1(4)
    Cons(1)[where as=as2 and s=(state-succ (exec-plan s as1) act)]
   by blast
 }
 note a=this
 {
  have no-effectless-act as1
   using Cons.prems(6) 1(1) rem-effectless-works-12
   by blast
  moreover have sat-precond-as s as1
   using Cons.prems(7) 1(1) sat-precond-as-pfx
   by blast
  moreover have as1 ∈ valid-plans PROB
   using Cons.prems(5) 1(1) valid-append-valid-pref
   by blast
  ultimately have ∃ as'. exec-plan s as1 = exec-plan s as' ∧
   subseq as' as1 ∧ length as' ≤ problem-plan-bound (snapshot PROB (fmrestrict-set
vs s))
```

using *no-vs-change-snapshot-s-vs-is-valid-bound*[*of - as1*]
using *Cons.prems*(*1*, *4*) *1*(*2*)
by *blast*
}
then obtain *as''* **where** *b*:
exec-plan s as1 = *exec-plan s as''* *subseq as'' as1*
length as'' ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*))
by *blast*
{
obtain *as'* **where** *i*:
exec-plan (*state-succ* (*exec-plan s as1*) *act*) *as'*
= *exec-plan* (*state-succ* (*exec-plan s as1*) *act*) *as2*
subseq as' as2
length as' ≤ *S vs lss PROB* (*fmrestrict-set vs* (*state-succ* (*exec-plan s as1*) *act*))
using *a*
by *blast*
let *?as'=as''* @ *act* # *as'*
have *exec-plan s ?as'* = *exec-plan s as*
using *1*(*1*) *b*(*1*) *i*(*1*) *exec-plan-Append exec-plan.simps*(*2*)
by *metis*
moreover have *subseq ?as' as*
using *1*(*1*) *b*(*2*) *i*(*2*) *subseq-append-iff*
by *blast*
moreover
{
{
— NOTE this is proved earlier in the original proof script. Moved here to improve transparency.
have *sat-precond-as* (*exec-plan s as1*) (*act* # *as2*)
using *empty-replace-proj-dual7*
using *1*(*1*) *Cons.prems*(*7*)
by *blast*
then have *fst act* $\subseteq_f$ (*exec-plan s as1*)
by *simp*
}
note *A* = *this*
{
have
fmrestrict-set vs (*state-succ* (*exec-plan s as1*) *act*)
= (*state-succ* (*fmrestrict-set vs* (*exec-plan s as''*)) (*action-proj act vs*))
using *b*(*1*) *A* *drest-succ-proj-eq-drest-succ*[**where** *s*=*exec-plan s as1*, *symmetric*]
by *simp*
also have ... = (*state-succ* (*fmrestrict-set vs s*) (*action-proj act vs*))
using *1*(*2*) *b*(*1*) *empty-change-no-change*
by *fastforce*
finally have ... = *fmrestrict-set vs* (*state-succ s* (*action-proj act vs*))
using *succ-drest-eq-drest-succ*

by *blast*
}
note $B = this$
have C: *fmrestrict-set vs* (*exec-plan s as''*) = *fmrestrict-set vs s*
using *1*(*2*) *b*(*1*) *empty-change-no-change*
by *fastforce*
{
have *act* ∈ *PROB*
using *Cons.prems*(*5*) *1 valid-append-valid-suff valid-plan-valid-head*
by *fast*
then have ℵ: *action-proj act vs* ∈ *prob-proj PROB vs*
using *action-proj-in-prob-proj*
by *blast*
then have (*state-succ s* (*action-proj act vs*)) ∈ (*state-successors* (*prob-proj PROB vs*) *s*)
proof (*cases fst* (*action-proj act vs*) $\subseteq_f$ *s*)
case *True*
then show *?thesis*
unfolding *state-successors-def*
using *Cons.hyps*(*2*) *1*(*3*) *b*(*1*) *A B C* ℵ *DiffI imageI singletonD vars-change-cons-2*
drest-succ-proj-eq-drest-succ
by *metis*
next
case *False*
then show *?thesis*
unfolding *state-successors-def*
using *Cons.hyps*(*2*) *1*(*3*) *b*(*1*) *A B C* ℵ *DiffI imageI singletonD*
drest-succ-proj-eq-drest-succ vars-change-cons-2
by *metis*
qed
}
then have *D*:
problem-plan-bound (*snapshot PROB* (*fmrestrict-set vs s*))
+ *S vs lss PROB* (*fmrestrict-set vs* (*state-succ s* (*action-proj act vs*)))
+ *1*
≤ *S vs lss PROB* (*fmrestrict-set vs s*)
using *Cons.prems*(*2*, *3*, *4*) *S-geq-S-succ-plus-ell*[**where** *s'*=*state-succ s* (*action-proj act vs*)]
by *blast*
{
have
length ?as' ≤ *problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*))
+ *1* + *S vs lss PROB* (*fmrestrict-set vs* (*state-succ* (*exec-plan s as1*) *act*))
using *b i*
by *fastforce*
then have *length ?as'* ≤ *S vs lss PROB* (*fmrestrict-set vs s*)
using *b*(*1*) *A B C D drest-succ-proj-eq-drest-succ*

by (*smt Suc-eq-plus1 add-Suc dual-order.trans*)
}
}
ultimately have *?case*
by *blast*
}
then show *?case*
by *blast*
qed

— NOTE first argument of 'top_sorted_abs' had to be wrapped into lambda.
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 −
— NOTE Proof premises and obtain conclusion of 'problem_plan_bound_S_bound_1st_step'.
{
have *a*: *rem-condless-act s* [] (*rem-effectless-act as*) ∈ *valid-plans PROB*
using *assms*(*5*) *rem-effectless-works-4' rem-condless-valid-10*
by *blast*
then have *b*: *no-effectless-act* (*rem-condless-act s* [] (*rem-effectless-act as*))
using *assms rem-effectless-works-6 rem-condless-valid-9*
by *fast*
then have *sat-precond-as s* (*rem-condless-act s* [] (*rem-effectless-act as*))
using *assms rem-condless-valid-2*
by *blast*
then have ∃ *as'*.
exec-plan s as' = *exec-plan s* (*rem-condless-act s* [] (*rem-effectless-act as*))
∧ *subseq as'* (*rem-condless-act s* [] (*rem-effectless-act as*))
∧ *length as'* ≤ *S vs lss PROB* (*fmrestrict-set vs s*)

using *assms a b problem-plan-bound-S-bound-1st-step*
by *blast*
}
then obtain *as'* **where** *1*:
exec-plan s as' = *exec-plan s* (*rem-condless-act s* [] (*rem-effectless-act as*))
subseq as' (*rem-condless-act s* [] (*rem-effectless-act as*))
length as' ≤ *S vs lss PROB* (*fmrestrict-set vs s*)
by *blast*
then have *2*: *exec-plan s as'* = *exec-plan s as*
using *rem-condless-valid-1 rem-effectless-works-14*

by *metis*
then have *subseq as′ as*
using *1(2) rem-condless-valid-8 rem-effectless-works-9 sublist-trans*
by *metis*
then show *?thesis*
using *1(3) 2*
by *blast*
qed

— NOTE first argument of ‘top_sorted_abs‘ had to be wrapped into lambda.
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 −
obtain *as*′ **where**
exec-plan s as′ = *exec-plan s as subseq as*′ *as*
length as′ ≤ *S vs lss PROB* (*fmrestrict-set vs s*)
using *assms problem-plan-bound-S-bound-2nd-step*
by *blast*
moreover {
— NOTE Derive sufficient conditions for inferring that ‘S vs lss PROB‘ is smaller or equal to the supremum of the set {*S vs lss PROB s*′ |*s*′. *s*′ ∈ *valid-states* (*prob-proj PROB vs*)}: i.e. being contained and that the supremum is contained as well.
let *?S*={*S vs lss PROB s*′ | *s*′. *s*′ ∈ *valid-states* (*prob-proj PROB vs*)}
{
have *fmrestrict-set vs s* ∈ *valid-states* (*prob-proj PROB vs*)
using *assms*(*4*) *graph-plan-not-eq-last-diff-paths*
by *blast*
then have *S vs lss PROB* (*fmrestrict-set vs s*) ∈ *?S*
using *calculation*(*1*)
by *blast*
}
moreover
{
have *finite* (*prob-proj PROB vs*)
by (*simp add*: *assms*(*1*) *prob-proj-def*)
then have *finite ?S*
using *Setcompr-eq-image assms*(*3*)
by (*metis List.finite-set finite-imageI*)

```
    }
    ultimately have S vs lss PROB (fmrestrict-set vs s) ≤ Sup ?S
      using le-cSup-finite by blast
  }
  ultimately show ?thesis
    using le-trans
    by blast
qed
```

— NOTE first argument of 'top_sorted_abs' had to be wrapped into lambda.

```
lemma problem-plan-bound-S-bound:
  fixes PROB :: 'a problem
  assumes finite PROB (top-sorted-abs (λx 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 −
  let ?f=λPROB.
    Sup {S vs lss PROB (s' :: 'a state) | s'. s' ∈ valid-states (prob-proj PROB vs)} + 1
  {
    fix as and s :: 'a state
    assume s ∈ valid-states PROB as ∈ valid-plans PROB
    then obtain as' where a:
      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)}
      using assms S-in-MPLS-leq-2-pow-n
      by blast
    then have length as' < ?f PROB
      by linarith
    moreover have exec-plan s as = exec-plan s as'
      using a(1)
      by simp
    ultimately have
      ∃ as'. exec-plan s as = exec-plan s as' ∧ subseq as' as ∧ length as' < ?f PROB
      using a(2)
      by blast
  }
  then show ?thesis
    using assms(1) problem-plan-bound-UBound[where f=?f]
    by fastforce
qed
```

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 ≡ (
($set\ lss = valid\text{-}states\ PROB$)
$\wedge$ ($top\text{-}sorted\text{-}abs\ (\lambda x\ y.\ y \in state\text{-}successors\ PROB\ x)\ lss$)
)

lemma *problem-plan-bound-S-bound-2nd-step-thesis*:
assumes *finite* ($PROB :: 'a\ problem$) ($sspace\text{-}DAG\ (prob\text{-}proj\ PROB\ vs)\ lss$)
($s \in valid\text{-}states\ PROB$) ($as \in valid\text{-}plans\ PROB$)
shows ($\exists\, as'.$ $(exec\text{-}plan\ s\ as' = exec\text{-}plan\ s\ as)$
$\wedge$ $(subseq\ as'\ as)$
$\wedge$ $(length\ as' \leq S\ vs\ lss\ PROB\ (fmrestrict\text{-}set\ vs\ s))$
)
using *assms problem-plan-bound-S-bound-2nd-step sspace-DAG-def*
by *fast*

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

theorem *problem-plan-bound-S-bound-thesis*:
assumes *finite* ($PROB :: 'a\ problem$) ($sspace\text{-}DAG\ (prob\text{-}proj\ PROB\ vs)\ lss$)
shows (
problem-plan-bound PROB
$\leq Sup\ \{S\ vs\ lss\ PROB\ s' \mid s'.\ s' \in valid\text{-}states\ (prob\text{-}proj\ PROB\ vs)\}$
)
using *assms problem-plan-bound-S-bound sspace-DAG-def*
by *fast*

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.