

Upper Bounding Diameters of State Spaces of Factored Transition Systems

Friedrich Kurz and Mohammad Abdulaziz

May 26, 2024

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 library about transition systems, which shall be of use in future related mechanisation efforts.

Contents

| | | |
|----------|--|-----------|
| 1 | Factored Systems Library | 2 |
| 1.1 | Semantics of Map Addition | 3 |
| 1.2 | States, Actions and Problems. | 3 |
| 2 | Factored System Sublist | 12 |
| 2.1 | Sublist Characterization | 12 |
| 2.2 | Main Theorems | 21 |
| 3 | Factored System | 30 |
| 3.1 | Semantics of Plan Execution | 31 |
| 3.1.1 | Characterization of the Set of Possible States | 31 |

| | | |
|----------|--|------------|
| 3.1.2 | State Lists and State Sets | 41 |
| 3.1.3 | Properties of Domain Changes During Plan Execution | 44 |
| 3.1.4 | Properties of Valid Plans | 47 |
| 3.2 | Reachable States | 63 |
| 3.3 | State Spaces | 84 |
| 3.4 | Needed Asses | 86 |
| 4 | Action Sequence Process | 105 |
| 5 | Dependency | 130 |
| 5.1 | Dependent Variables and Variable Sets | 130 |
| 5.2 | Transitive Closure of Dependent Variables and Variable Sets | 135 |
| 5.3 | Sets of Numbers | 140 |
| 6 | Topological Properties | 145 |
| 6.1 | Basic Definitions and Properties | 145 |
| 6.2 | Recurrence Diameter | 148 |
| 6.3 | The Relation between Diameter, Sublist Diameter and Re- currence Diameter Bounds. | 180 |
| 6.4 | Traversal Diameter | 192 |
| 7 | System Abstraction | 195 |
| 7.1 | Projection of Actions, Sequences of Actions and Factored Representations. | 195 |
| 7.2 | Snapshotting | 231 |
| 7.3 | State Space Projection | 247 |
| 8 | Acyclicity | 251 |
| 8.1 | Topological Sorting of Dependency Graphs | 251 |
| 8.2 | The Weightiest Path Function (wlp) | 252 |
| 9 | Acyclic State Spaces | 257 |
| 9.1 | State Space Acyclicity | 276 |

```

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

```

1 Factored Systems Library

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

1.1 Semantics of Map Addition

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

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

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

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

`!fmap 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. $\delta\text{-star} = \{\pi. \text{set } \pi \subseteq \delta\}$. [Abdulaziz et al., Definition 3, p.9]

Ultimately, the effect of executing an action ‘a’ on a state ‘s’ is given by calculating the succeeding state. In general, the succeeding 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

$\text{prob-dom prob} \equiv \bigcup ((\lambda (s1, s2). \text{action-dom } s1 \ s2) \text{ ‘prob’})$

definition valid-states where

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

definition valid-plans where

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

definition state-succ where

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

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 $\text{length } l \geq 0$

by *simp*

lemma len-gt-pref-is-pref:

fixes $l \ l1 \ l2$

assumes $(\text{length } l2 > \text{length } l1) (\text{prefix } l1 \ l) (\text{prefix } l2 \ l)$

shows $(\text{prefix } l1 \ l2)$

using *assms* **proof** (*induction l2 arbitrary: l1 l*)

case *Nil*

then have $\neg(\text{length } [] > \text{length } l1)$

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  $\neg(\text{prefix } (a \# l2) l)$ 
    by simp
  then show ?thesis using Cons.prem1(4)
    by simp
next
case (Cons c l)
then have 1:  $\text{length } l2 > \text{length } l1$ 
  using Cons.prem1(2)
  by fastforce
then show ?thesis using Cons proof(cases l)
  case Nil
  then have  $l1 = [c] \ l2 = [c]$ 
    using Cons.prem1(3, 4) local.Cons 1
    by fastforce+
  then show ?thesis
    using 1
    by auto
next
case (Cons d l')
{
  thm len-ge-0
  have  $\text{length } l1 \geq 0$ 
    by simp
  then have  $\text{length } l2 > 0$ 
    using 1
    by force
  then have  $l2 \neq []$  using 1
    by blast
}
then have  $\text{length } (a \# l1) \leq \text{length } (b \# l2)$ 
  using 1 le-eq-less-or-eq
  by simp
then show ?thesis
  using Cons.prem1(3, 4) prefix-length-prefix
  by fastforce

```

qed
 qed
 qed
 qed

lemma *nempty-list-append-length-add*:
fixes $l1\ l2\ l3$
assumes $l2 \neq []$
shows $\text{length } (l1 @ l3) < \text{length } (l1 @ l2 @ l3)$
using *assms*
by (*induction l2*) *auto*

lemma *append-filter*:
fixes $f1 :: 'a \Rightarrow \text{bool}$ **and** $f2\ as1\ as2$ **and** $p :: 'a\ \text{list}$
assumes $(as1 @ as2 = \text{filter } f1 (\text{map } f2\ p))$
shows $(\exists p-1\ p-2.$
 $(p-1 @ p-2 = p)$
 $\wedge (as1 = \text{filter } f1 (\text{map } f2\ p-1))$
 $\wedge (as2 = \text{filter } f1 (\text{map } f2\ p-2))$
 $)$
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 $?p1 @ ?p2 = []\ as1 = (\text{filter } f1 (\text{map } f2\ ?p1))\ as2 = (\text{filter } f1 (\text{map } f2\ ?p2))$
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.prem1 Nil*
have $1: as2 = \text{filter } f1 (\text{map } f2 (a \# p))$
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.prem
    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.prem 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 $f1 :: 'a \Rightarrow bool$ **and** $f2\ as1\ as2\ as3$ **and** $p :: 'a\ list$

assumes $(as1\ @\ as2\ @\ as3 = filter\ f1\ (map\ f2\ p))$

shows $(\exists\ p-1\ p-2\ p-3.$

$(p-1\ @\ p-2\ @\ p-3 = p)$

$\wedge (as1 = filter\ f1\ (map\ f2\ p-1))$

$\wedge (as2 = filter\ f1\ (map\ f2\ p-2))$

$\wedge (as3 = filter\ f1\ (map\ f2\ p-3))$

)

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*: $\bigwedge P\ l. (filter\ (\lambda x. P\ x)\ l = []) = list-all\ (\lambda x. \neg P\ x)\ l$

proof —

fix $P\ l$

show $(filter\ (\lambda x. P\ x)\ l = []) = list-all\ (\lambda x. \neg P\ x)\ l$

apply(*induction* l)

apply(*auto*)

done

qed

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

lemma *MEM-SPLIT*:

fixes $x\ l$

assumes $\neg ListMem\ x\ l$

shows $\forall\ l1\ l2. l \neq l1\ @\ [x]\ @\ l2$

proof —


```

{
  assume C:  $\neg(\forall l1\ l2. l \neq l1 @ [x] @ l2)$ 
  then have  $\exists l1\ l2. l = l1 @ [x] @ l2$ 
    by blast
  then obtain l1 l2 where 1:  $l = l1 @ [x] @ l2$ 
    by blast
  from assms
  have 2:  $(\forall xs. l \neq x \# xs) \wedge (\forall xs. (\forall y. l \neq y \# xs) \vee \neg ListMem\ x\ xs)$ 
    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  $\exists xs. \exists y. l = y \# xs \wedge ListMem\ x\ xs$ 
      by blast
    then have  $\neg(\forall xs. (\forall y. l \neq y \# xs) \vee \neg ListMem\ x\ xs)$ 
      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

$(l1 @ [e] @ l2 = m1 @ m2)$

\longleftrightarrow

```

      (∃ 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)  $\neg P x$  list-all P l
  shows sublist l la  $\vee$  sublist l lb
proof -
  {
    assume  $\neg P x$  list-all P l
    then have  $\neg$ 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.prem1(2)
          by simp
        then have  $\neg$ ListMem x l
          using Cons.prem1(1) Cons.IH
          by blast
      }
      moreover {
        have P a
          using Cons.prem1(2)
          by simp
        then have  $a \neq x$ 
          using Cons.prem1(1)
          by meson
      }
      ultimately show ?case
        using Cons.prem1(1, 2) ListMem-iff list.pred-set
        by metis
    qed
  }
  then have  $\neg$ 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

$$(\text{sublist } [] \text{ l1} = \text{T}) / (\text{sublist } (\text{h::t}) [] = \text{F}) / (\text{sublist } (\text{x}::\text{l1}) (\text{y}::\text{l2}) = (\text{x} = \text{y}) / \text{sublist } \text{l1 } \text{l2} \text{ sublist } (\text{x}::\text{l1}) \text{ l2})$$

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

$$\text{subseq } \text{xs } \text{ys} \equiv \text{subseq } \text{xs } \text{ys}$$

‘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 tech-

nical 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* \longleftrightarrow *subseq* *l1 l2*

proof —

have *subseq* *l1 l2* = *list-emb* ($\lambda x y. x = y$) *l1 l2*

by *blast*

moreover {

have *sublist-HOL4* *l1 l2* \longleftrightarrow *list-emb* ($\lambda x y. x = y$) *l1 l2*

proof (*induction rule: sublist-HOL4.induct*)

case ($\exists x l1 y l2$)

then show *sublist-HOL4* (*x* # *l1*) (*y* # *l2*) \longleftrightarrow *list-emb* ($\lambda x y. x = y$) (*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 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* $\equiv \exists pfx sfx. pfx @ frag @ sfx = l$

lemma *list-frag-HOL4-equiv-sublist*:
shows *list-frag-HOL4* $l\ l' \longleftrightarrow$ *sublist* $l'\ l$
unfolding *list-frag-HOL4-def* *sublist-def*
by *blast*

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

The remainder 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* $l \leq$ *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 $subseq\ l1\ l \wedge subseq\ l2\ l$
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 $\text{subseq } (h \# l1) \ l2 \implies \exists l2a \ l2b. \ l2 = l2a @ [h] @ l2b \wedge \neg \text{ListMem } h \ l2a$

proof (*induction l2 arbitrary: h l1*)

— NOTE l2 cannot be empty when $h \# l1$ isn't.

case *Nil*

have $\neg(\text{subseq } (h \# l1) \ [])$

by *simp*

then show *?case*

using *Nil.prem*s

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 $\exists l2a \ l2b. \ (\text{Cons } a \ l2) = l2a @ [h] @ l2b \wedge \neg \text{ListMem } h \ l2a$

using *ListMem-iff*

by *force*

next

case *False*

have $\text{subseq } (h \# l1) \ l2$

using *Cons.prem*s *False subseq-Cons2-neq*

by *force*

then obtain $l2a \ l2b$ **where** $l2 = l2a @ [h] @ l2b \wedge \neg \text{ListMem } h \ l2a$

using *Cons.IH Cons.prem*s

by *meson*

moreover have $a \# l2 = (a \# l2a) @ [h] @ l2b$

using *calculation(1)*

by *simp*

moreover have $\neg(\text{ListMem } h \ (a \# l2a))$

using *False calculation(2) ListMem.simp*s

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 $l1 = (h \# l1') \ l2 = l2a @ [h] @ l2b \ \text{subseq } l1 \ l2 \ \neg(\text{ListMem } h \ l2a)$

shows $\text{subseq } l1 \ (h \# l2b)$

using *assms*

proof (*induction l2a arbitrary: l1 l2 h l1'*)

case *Nil*


```

then have l2 = h # l2b
  by fastforce
then show ?case using Nil.premis(3)
  by blast
next
case (Cons a l2a)
have a ≠ h
  using Cons.premis(4) ListMem.simps
  by fast
then have subseq l1 (l2a @ [h] @ l2b)
  using Cons.premis(1, 2, 3) subseq-Cons2-neq
  by force
moreover have ¬ListMem h l2a
  using Cons.premis(4) insert
  by metis
ultimately have subseq l1 (h # l2b)
  using Cons.IH Cons.premis
  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.prem1 True
      by simp
    ultimately show ?thesis
      by blast
  next
  case False
    have 1: subseq (h # l1) l2
      using Cons.prem1 False subseq-Cons2-neq
      by metis
    then obtain l2a l2b where l2 = l2a @ [h] @ l2b ¬ListMem h l2a
      using Cons.IH Cons.prem1
      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 $subseq (l1 @ l2) l3 \implies \exists l3a l3b. (l3 = l3a @ l3b) \wedge subseq l1 l3a \wedge subseq l2 l3b$
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 $subseq (l1 @ l2 @ l3) l \implies subseq (l1 @ l3) l$
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.prem1 append-sublist-1*
by *auto*
next
case (*Cons a list*)
then show *?thesis*
using *Cons.prem1 subseq-append' subseq-order.dual-order.trans*
by *blast*
qed

qed

lemma *sublist-subset*:
 assumes *subseq l1 l2*
 shows *set l1 \subseteq 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) \longleftrightarrow (subseq l1 l2))*
 by *fastforce*

lemma *sublist-every*:
 fixes *l1 l2 P*
 assumes *(subseq l1 l2 \wedge 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 \longleftrightarrow 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 *(\exists l3 l4. (l2 = l3 @ [h] @ l4) \wedge (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 ($\lambda x. \neg(h = x)$) 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 (λx. ¬(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*)

($\forall l s.$
 (*PProbs s*)
 \wedge (*PProbl l*)
 \wedge (*list-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-all P1 l'*)
 \wedge (*subseq l' l*)
)
)
 ($\forall s l1 l2. f (f s l1) l2 = f s (l1 @ l2)$)
 ($\forall s l. (PProbs s) \wedge (PProbl l) \longrightarrow (PProbs (f s l))$)
 ($\forall l1 l2. (subseq l1 l2) \wedge (PProbl l2) \longrightarrow (PProbl l1)$)
 ($\forall l1 l2. PProbl (l1 @ l2) \longleftrightarrow (PProbl l1 \wedge PProbl l2)$)

shows ($\exists l'.$

(*f s l' = f s l*)
 \wedge (*length (filter P3 l') \leq length (filter P3 l)*)
 \wedge ($\forall l''.$
 (*sublist l'' l' \wedge list-all P1 l''*)
 \longrightarrow (*length (filter P2 l'') \leq k1*)
)
 \wedge (*subseq l' l*)
)

using *assms*

proof (*induction filter* ($\lambda x. \neg P1 x$) *l arbitrary: P1 P2 P3 k1 f PProbs PProbl s l*)

case *Nil*

then have *list-all* ($\lambda x. P1 x$) *l*

using *Nil(1) filter-empty-every-not*[*of* $\lambda x. \neg P1 x$ *l*]

by *presburger*

then obtain *l' where 1:*

(*f s l' = f s l*) *length (filter P2 l') \leq k1* *length (filter P3 l') \leq length (filter P3*
l)

list-all P1 l' subseq l' l

using *Nil.prem*s(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 $\text{length} (\text{filter } P2 \ l') \leq \text{length} (\text{filter } P2 \ l)$
using *sublist-imp-len-filter-le*
by blast
then have $\text{length} (\text{filter } P2 \ l') \leq k1$
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*:
 $l = l1 \ @ \ a \ \# \ l2 \ (\forall u \in \text{set } l1. P1 \ u) \ \neg \ P1 \ a \ \wedge \ x = [x \leftarrow l2 \ . \ \neg \ P1 \ x]$
using *Cons(2) filter-eq-Cons-iff[of $\lambda x. \neg P1 \ x$]*
by metis
then have *3: PProbl l2*
using *Cons.prem(2, 6) 2(1) sublist-append-back*
by blast
 — NOTE Use the induction hypothesis to obtain a specific ‘l’’’.
{
have $x = \text{filter} (\lambda x. \neg P1 \ x) \ l2$
using *2(3)*
by blast
moreover have *PProbs (f (f s l1) [a])*
using *Cons.prem(1, 2, 5, 6, 7) 2(1) elem sublist-SING-MEM*
by metis
moreover have $\forall l \ s. PProbs \ s \ \wedge \ PProbl \ l \ \wedge \ \text{list-all } P1 \ l \ \longrightarrow \ (\exists l'. f \ s \ l' = f \ s \ l \ \wedge \ \text{length} (\text{filter } P2 \ l') \leq k1 \ \wedge \ \text{length} (\text{filter } P3 \ l') \leq \text{length} (\text{filter } P3 \ l))$
 $\wedge \ \text{list-all } P1 \ l' \ \wedge \ \text{subseq } l' \ l)$
using *Cons.prem(3)*
by blast
moreover have $\forall s \ l1 \ l2. f \ (f \ s \ l1) \ l2 = f \ s \ (l1 \ @ \ l2)$
 $\forall s \ l. PProbs \ s \ \wedge \ PProbl \ l \ \longrightarrow \ PProbs \ (f \ s \ l)$
 $\forall l1 \ l2. \text{subseq } l1 \ l2 \ \wedge \ PProbl \ l2 \ \longrightarrow \ PProbl \ l1$
 $\forall l1 \ l2. PProbl \ (l1 \ @ \ l2) = (PProbl \ l1 \ \wedge \ PProbl \ l2)$
using *Cons.prem(4, 5, 6, 7)*
by blast+
ultimately have $\exists l'.$
 $f \ (f \ s \ l1) \ [a] \ l' = f \ (f \ (f \ s \ l1) \ [a]) \ l2 \ \wedge \ \text{length} (\text{filter } P3 \ l') \leq \text{length} (\text{filter } P3 \ l2)$
 $\wedge \ (\forall l''. \text{sublist } l'' \ l' \ \wedge \ \text{list-all } P1 \ l'' \ \longrightarrow \ \text{length} (\text{filter } P2 \ l'') \leq k1) \ \wedge \ \text{subseq } l' \ l2$

```

    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.prem(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.prem(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.prem(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.prem(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.prem(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. ∀ p. (n = f p) ∧ P p ⟶ (∃ p'.
P p' ∧ f p' ≤ 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: fmfiltre_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 $\neg(v \in vs) \text{ fmdom}' f \subseteq \text{fdom } v \in \text{fmdom}' f$
 shows $v \in \text{fmdom}' (\text{fmrestrict-set } (\text{fdom} - vs) f)$
 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*:
 $\text{disjnt } (\text{fmdom}' x) vs \implies (\text{fmrestrict-set } vs s = \text{fmrestrict-set } vs (x ++ s))$

proof —

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

— TODO refactor into 'FmapUtils.thy'.

lemma *graph-plan-card-state-set*:
 fixes *PROB vs*
 assumes *finite vs*
 shows $\text{card } (\text{fmdom}' (\text{fmrestrict-set } vs s)) \leq \text{card } vs$
proof —
 let $?vs' = \text{fmdom}' (\text{fmrestrict-set } vs s)$
 have $?vs' \subseteq vs$
 using *fmdom'-restrict-set*
 by *metis*
 moreover have $\text{card } ?vs' \leq \text{card } vs$

```

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

```

lemma *exec-drest-5*:

```

    fixes x vs
    assumes fmdom' x ⊆ vs
    shows (fmrestrict-set vs x = x)
proof –
  — TODO refactor and make into ISAR proof.
  {
    fix v
    have fmlookup (fmrestrict-set vs x) v = fmlookup x v
      apply(cases v ∈ fmdom' x)
      subgoal using assms by auto
      subgoal by (simp add: fmdom'-notD)
      done
    then have fmlookup (fmrestrict-set vs x) v = fmlookup x v
      by fast
  }
  moreover have fmlookup (fmrestrict-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-set (fmdom' s - vs) s = fmrestrict-set (fmdom' s' - vs) s')
      (fmrestrict-set vs s = fmrestrict-set vs s')
    shows (s = s')
proof –
  have  $\forall x. fmlookup\ s\ x = fmlookup\ s'\ x$ 
    using assms(1, 2) fmdom'-notD fminusI fmlookup-restrict-set Diff-iff
    by metis
  then show ?thesis using fmap-ext
    by blast
qed

```

lemma *drest-smap-drest-smap-drest*:

```

    fixes x s vs
    shows fmrestrict-set vs x ⊆f s ⟷ fmrestrict-set vs x ⊆f fmrestrict-set vs s
proof –
  — TODO this could be refactored into standalone lemma since it's very common
  in proofs.
  have 1: fmlookup (fmrestrict-set vs s) ⊆m fmlookup s

```

```

    by (metis fmdom'.rep-eq fmdom'-notI fmllookup-restrict-set map-le-def)
  moreover
  {
    assume P1: fmrestrict-set vs x  $\subseteq_f$  s
    moreover have 2: fmllookup (fmrestrict-set vs x)  $\subseteq_m$  fmllookup s
      using P1 fmsubset.rep-eq by blast
    {
      fix v
      assume v  $\in$  fmdom' (fmrestrict-set vs x)
      then have fmllookup (fmrestrict-set vs x) v = fmllookup (fmrestrict-set vs s) v
        by (metis (full-types) 2 domIff fmdom'-notI fmllookup-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 $fdom' f1 \subseteq vs f1 ++ f2 \neq f2$

shows $fmrestrict\text{-}set\ vs\ (f1\ ++\ f2) \neq fmrestrict\text{-}set\ vs\ f2$

proof –

```
{
  assume  $C: fmrestrict\text{-}set\ vs\ (f1\ ++\ f2) = fmrestrict\text{-}set\ vs\ f2$ 
  then have  $\forall x \in fdom'\ (fmrestrict\text{-}set\ vs\ (f1\ ++\ f2)).$ 
     $fmlookup\ (fmrestrict\text{-}set\ vs\ (f1\ ++\ f2))\ x$ 
     $= fmlookup\ (fmrestrict\text{-}set\ vs\ f2)\ x$ 
  by simp
  obtain  $v$  where  $a: v \in fdom'\ f1\ fmlookup\ (f1\ ++\ f2)\ v \neq fmlookup\ f2\ v$ 
  using assms(2)
  by (metis fmap-add-ltr-def fmap-ext fdom'-notD fdom-notI fmlookup-add)
  then have  $b: v \in vs$ 
  using assms(1)
  by blast
  moreover {
    have  $fdom'\ (fmrestrict\text{-}set\ vs\ (f1\ ++\ f2)) = vs \cap fdom'\ (f1\ ++\ f2)$ 
    by (simp add: fdom'-alt-def fmfilter-alt-defs(4))
    then have  $v \in fdom'\ (fmrestrict\text{-}set\ vs\ (f1\ ++\ f2))$ 
    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*: $\bigwedge vs\ s. dom\ (fmlookup\ (fmrestrict\text{-}set\ vs\ s)) = vs \cap (fdom'\ s)$

by (*auto simp add: fdom'-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 succeeding state (`'state_succ'`). Plan execution (`'exec_plan'`) extends this notion to sequences of actions by calculating the succeeding state from the given state and action pair and then recursively executing the remaining actions on the succeeding 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 defining 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.

$s = \text{fmupd } v \ x \ (\text{fmrestrict_set } X \ s)$

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*: $\{s. \text{fmdom}' s = \{\}\} = \{\text{fmempty}\}$

proof –

let $?A = \{s. \text{fmdom}' s = \{\}\}$

let $?B = \{\text{fmempty}\}$

fix s

show *?thesis* **proof**(*rule ccontr*)

assume $C: ?A \neq ?B$

then show *False* **proof** –

{

assume $C1: ?A \subset ?B$

have $?A = \{\}$ **using** $C1$ **by** *force*

then have *False* **using** *fmdom'-empty* **by** *blast*

}

moreover

{

assume $C2: \neg(?A \subset ?B)$

then have $\text{fmdom}' \text{fmempty} = \{\}$

by *auto*

moreover have $\text{fmempty} \in ?A$

by *auto*

moreover have $?A \neq \{\}$

using *calculation(2)* **by** *blast*

moreover have $\forall a \in ?A. a \notin ?B$

by (*metis (mono-tags, lifting)*)

C *Collect-cong calculation(1) fmrestrict-set-dom fmrestrict-set-null*

singleton-conv)

moreover have $\text{fmempty} \in ?B$ **by** *auto*

moreover have $\exists a \in ?A. a \in ?B$

by *simp*

moreover have $\neg(\forall a \in ?A. a \notin ?B)$

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 $(v \in \text{fmdom}' s)$

shows $(\text{fmdom}' ((\lambda s. \text{fmupd } v x s) s) = \text{fmdom}' s)$

using *assms insert-absorb*

by *auto*

— NOTE added lemma.

lemma *possible-states-set-ii-b*:

fixes $s\ x\ v$

assumes $(v \notin \text{fmdom}'\ s)$

shows $(\text{fmdom}'\ ((\lambda s. \text{fmupd}\ v\ x\ s)\ s) = \text{fmdom}'\ s \cup \{v\})$

by *auto*

— NOTE added lemma.

lemma *fmap-neg*:

fixes $s :: ('a, \text{bool})\ \text{fmap}$ **and** $s' :: ('a, \text{bool})\ \text{fmap}$

assumes $(\text{fmdom}'\ s = \text{fmdom}'\ s')$

shows $((s \neq s') \longleftrightarrow (\exists v \in (\text{fmdom}'\ s). \text{fmlookup}\ s\ v \neq \text{fmlookup}\ s'\ v))$

using *assms fmap-ext fmdom'-notD*

by *metis*

— NOTE added lemma.

lemma *fmdom'-fmsubset-restrict-set*:

fixes $X1\ X2$ **and** $s :: ('a, \text{bool})\ \text{fmap}$

assumes $X1 \subseteq X2\ \text{fmdom}'\ s = X2$

shows $\text{fmdom}'\ (\text{fmrestrict-set}\ X1\ s) = X1$

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** $s :: 'a\ \text{state}$

assumes $X1 \subseteq X2\ s \in \{s. \text{fmdom}'\ s = X2\}$

shows $\text{fmrestrict-set}\ X1\ s \in \{s. \text{fmdom}'\ s = X1\}$

using *assms fmdom'-fmsubset-restrict-set*

by *blast*

— NOTE added lemma.

lemma *fmupd-fmsubset-restrict-set*:

fixes $X\ v\ x$ **and** $s :: 'a\ \text{state}$

assumes $s \in \{s. \text{fmdom}'\ s = \text{insert}\ v\ X\}\ \text{fmlookup}\ s\ v = \text{Some}\ x$

shows $s = \text{fmupd}\ v\ x\ (\text{fmrestrict-set}\ X\ s)$

proof —

— Show that domains of 's' and 'fmupd v x (fmrestrict_set X s)' are identical.

have $1: \text{fmdom}'\ s = \text{insert}\ v\ X$

using *assms(1)*

by *simp*

{

have $X \subseteq \text{insert}\ v\ X$

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 fmllookup s w = fmllookup (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 fmllookup s w = fmllookup (fmupd v x (fmrestrict-set X s)) w
      by (cases w = v)
      (auto simp add: assms calculation)
  }
  ultimately have fmllookup s w = fmllookup (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 =
  (( $\lambda s. \text{fmupd } v \text{ True } s$ ) ‘ $\{s :: 'a \text{ state. } \text{fmdom}' s = X\}$ )
   $\cup$  (( $\lambda s. \text{fmupd } v \text{ False } s$ ) ‘ $\{s :: 'a \text{ state. } \text{fmdom}' s = X\}$ )

```

Show the goal by mutual inclusion. The inclusion $\text{fmupd } v \text{ True } \text{ ‘ } \{s. \text{fmdom}' s = X\} \cup \text{fmupd } v \text{ False } \text{ ‘ } \{s. \text{fmdom}' s = X\} \subseteq \{s. \text{fmdom}' s = \text{insert } v X\}$ is trivial and can be solved by automation. For the complimentary proof $\{s. \text{fmdom}' s = \text{insert } v X\} \subseteq \text{fmupd } v \text{ True } \text{ ‘ } \{s. \text{fmdom}' s = X\} \cup \text{fmupd } v \text{ False } \text{ ‘ } \{s. \text{fmdom}' s = X\}$ however we need to do more work. In our case we choose a proof by contradiction and show that an $s \in \{s. \text{fmdom}' s = \text{insert } v X\}$ 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  $\text{fmdom}' s = X \cup \{v\}$ 
    using obtain-s
    by auto
  moreover have  $\forall s' \in ?B. \text{fmdom}' s' = X \cup \{v\}$ 
    by auto
  moreover have
    ( $s \notin ((\lambda s. \text{fmupd } v \text{ True } s) \text{ ‘ } \{s. \text{fmdom}' s = X\})$ )
    ( $s \notin ((\lambda s. \text{fmupd } v \text{ False } s) \text{ ‘ } \{s. \text{fmdom}' s = X\})$ )
    using obtain-s
    by blast+

```

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

```

moreover
{
  fix s :: 'a state
  assume 1:  $s \in \{s :: 'a \text{ state. } \text{fmdom}' s = \text{insert } v X\}$ 
  then have fmrestrict-set X s  $\in \{s :: 'a \text{ state. } \text{fmdom}' s = X\}$ 
    using subset-insertI fmsubset-restrict-set
    by metis
  moreover
  {
    assume fmlookup s v = Some True
    then have  $s = \text{fmupd } v \text{ True } (\text{fmrestrict-set } X s)$ 
      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 ‘ $2^{\text{card } X}$ ’, 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. Pivotaly, 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 fmllookup ((λs :: 'a state. fmupd v b s) s) x = fmllookup 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 ⟶ fmllookup (?f s1) x = fmllookup s1 x
        ∀x∈X. x ≠ v ⟶ fmllookup (?f s2) x = fmllookup s2 x
      by simp+
    }
    then have

```

```

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

```

— NOTE added lemma.

```

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

```

proof —

```

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

```

— NOTE added lemma.

```

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

```

proof —

```

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

```

```

    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  $\text{card } \{s :: 'a \text{ state}. \text{fmdom}' s = X\} = 2^{\wedge} (\text{card } X)$ 
using assms
proof (induction X)
  case empty
  then have  $1: \{s :: 'a \text{ state}. \text{fmdom}' s = \{\}\} = \{\text{fmempty}\}$ 
    using empty-domain-fmap-set
    by simp
  then have  $\text{card } \{\text{fmempty}\} = 1$ 
    using is-singleton-altdef
    by blast
  then have  $2^{\wedge} (\text{card } \{\}) = 1$ 
    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
       $\{s :: 'a \text{ state}. \text{fmdom}' s = \text{insert } x F\}$ 
       $= (\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\} \cup (\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s.$ 
       $\text{fmdom}' s = F\}$ 
      using False construction-of-all-possible-states-lemma
      by metis
  end

```

then have 2:
 $\text{card} (\{s :: 'a \text{ state. } \text{fmdom}' s = \text{insert } x F\})$
 $= \text{card} ((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\} \cup (\lambda s. \text{fmupd } x \text{ False } s)$
 $\text{ ' } \{s. \text{fmdom}' s = F\})$

by *argo*
then have 3: $2^{\wedge}(\text{card} (\text{insert } x F)) = 2 * 2^{\wedge}(\text{card } F)$
using *False insert.hyps(1)*
by *simp*
then have
 $\text{card} ((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\}) = 2^{\wedge}(\text{card } F)$
 $\text{card} ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\}) = 2^{\wedge}(\text{card } F)$
using *False card-update insert.IH insert.hyps(1)*
by *metis+*
moreover have
 $((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $\cap ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $= \{\}$

using *False insert.hyps(1) updates-disjoint*
by *metis*
moreover have *card* (
 $((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $\cup ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $)$
 $= \text{card} (((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\}))$
 $+ \text{card} ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\})$

using *calculation card-Un-disjoint card.infinite*
power-eq-0-iff rel-simps(76)
by *metis*
then have *card* (
 $((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $\cup ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $)$
 $= 2 * (2^{\wedge}(\text{card } F))$
using *calculation(1, 2)*
by *presburger*
then have *card* (
 $((\lambda s. \text{fmupd } x \text{ True } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $\cup ((\lambda s. \text{fmupd } x \text{ False } s) \text{ ' } \{s. \text{fmdom}' s = F\})$
 $)$
 $= 2^{\wedge}(\text{card} (\text{insert } x F))$
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  $\neg([] = \text{state-list } s \ as)$ 
proof (induction as)
qed auto
```

lemma state-list-length-non-zero:

```
  fixes as s
  shows  $\neg(0 = \text{length } (\text{state-list } s \ as))$ 
proof (induction as)
qed auto
```

lemma state-list-length-lemma:

```
  fixes as s
  shows  $\text{length } as = \text{length } (\text{state-list } s \ as) - 1$ 
proof (induction as arbitrary: s)
next case (Cons a as)
  have  $\text{length } (\text{state-list } s \ (\text{Cons } a \ as)) - 1 = \text{length } (\text{state-list } (\text{state-succ } s \ a) \ as)$ 
  by auto
  — TODO unwrap metis proof.
  then show  $\text{length } (\text{Cons } a \ as) = \text{length } (\text{state-list } s \ (\text{Cons } a \ as)) - 1$ 
  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  $\text{length } (\text{state-list } s \ as) = (\text{length } as + 1)$ 
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 $s1 \in \text{state-set } s2 \longleftrightarrow \text{prefix } s1 \ s2 \wedge s1 \neq []$

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 $s1 \in \text{state-set } s2 \implies \text{prefix } s1 \ s2 \wedge s1 \neq []$

by (*induction s2 arbitrary: s1*) *auto*

moreover {

assume *P*: $\text{prefix } s1 \ s2 \ s1 \neq []$

then have $s1 \in \text{state-set } s2$

proof (*induction s2 arbitrary: s1*)

case (*Cons a s2*)

obtain *s1'* **where** $1: s1 = a \# s1' \ \text{prefix } s1' \ s2$

using *Cons.premis(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 $s1' \in \text{state-set } s2$

using *1 False Cons.IH*

by *blast*

then show *?thesis*

using *1*

by *fastforce*

qed

qed *simp*

}

ultimately show $s1 \in \text{state-set } s2 \longleftrightarrow \text{prefix } s1 \ s2 \wedge s1 \neq []$

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 $e \in \text{state-set } X$

shows $\text{length } e \leq \text{length } X$
using *assms*
by (*induction X arbitrary: e*) *auto*

lemma *lemma-temp*:
fixes $x\ s\ \text{as } h$
assumes $x \in \text{state-set } (\text{state-list } s\ \text{as})$
shows $\text{length } (h \# \text{state-list } s\ \text{as}) > \text{length } x$
using *assms LENGTH-state-set le-imp-less-Suc*
by *force*

lemma *NIL-NOTIN-stateset*:
fixes X
shows $\square \notin \text{state-set } X$
by (*induction X*) *auto*

— NOTE added lemma.

lemma *state-set-card-i*:
fixes $X\ a$
shows $[a] \notin (\text{Cons } a\ \text{'state-set } X)$
by (*induction X*) *auto*

— NOTE added lemma.

lemma *state-set-card-ii*:
fixes $X\ a$
shows $\text{card } (\text{Cons } a\ \text{'state-set } X) = \text{card } (\text{state-set } X)$
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 $\text{card } (\text{state-set } (a \# X)) = 1 + \text{card } (\text{state-set } X)$
proof —
have $\text{card } (\text{state-set } (a \# X)) = \text{card } (\text{insert } [a] (\text{Cons } a\ \text{'state-set } X))$
by *auto*
— TODO unwrap this metis step.
also have $\dots = 1 + \text{card } (\text{Cons } a\ \text{'state-set } X)$
using *state-set-card-i*
by (*metis Suc-eq-plus1-left card-insert-disjoint finite-imageI state-set-finite*)
also have $\dots = 1 + \text{card } (\text{state-set } X)$

```

    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)  $\subseteq$  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)  $\subseteq$  (fmdom' s  $\cup$  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)  $\in$  PROB (s  $\in$  valid-states PROB)
  shows (fmdom' e  $\subseteq$  fmdom' s)
proof -
  {
    have fmdom' e  $\subseteq$  action-dom p e
      unfolding action-dom-def
      by blast
    also have ...  $\subseteq$  prob-dom PROB
      unfolding action-dom-def prob-dom-def

```

```

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

```

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

```

  fixes  $a s$ 
  assumes  $a \in PROB$   $s \in valid-states PROB$ 
  shows  $fmdom' (snd a) \subseteq fmdom' s$ 
proof –
{
  have  $fmdom' (snd a) \subseteq (\lambda(s1, s2). action-dom s1 s2) a$ 
    unfolding action-dom-def
    using case-prod-beta
    by fastforce
  also have  $\dots \subseteq prob-dom PROB$ 
    using assms(1) prob-dom-def Sup-upper
    by fast
  finally have  $fmdom' (snd a) \subseteq fmdom' s$ 
    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  $(p, e) \in PROB$   $s \in valid-states PROB$ 
  shows  $fmdom' p \subseteq fmdom' s$ 
proof –
{
  have  $fmdom' p \subseteq (\lambda(s1, s2). action-dom s1 s2) (p, e)$ 
    using action-dom-def
    by fast
  also have  $\dots \subseteq prob-dom PROB$ 
    using assms(1)
    by (simp add: Sup-upper pair-imageI prob-dom-def)
  finally have  $fmdom' p \subseteq fmdom' s$ 
    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 $a \in PROB\ s \in \text{valid-states } PROB$
shows $\text{fmdom}'(\text{fst } a) \subseteq \text{fmdom}'\ s$
proof –
{
 have $\text{fmdom}'(\text{fst } a) \subseteq (\lambda(s1, s2). \text{action-dom } s1\ s2)\ a$
 using *action-dom-def*
 by *force*
 also have $\dots \subseteq \text{prob-dom } PROB$
 using *assms(1)*
 by (*simp add: Sup-upper pair-imageI prob-dom-def*)
 finally have $\text{fmdom}'(\text{fst } a) \subseteq \text{fmdom}'\ s$
 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 $(p, e) \in PROB\ s \in \text{valid-states } PROB$
shows $\text{action-dom } p\ e \subseteq \text{fmdom}'\ s$
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 $(p, e) \in PROB$
shows $\text{fmdom}'\ e \subseteq \text{prob-dom } PROB$
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 $a \in PROB$
shows $\text{fmdom}'(\text{snd } a) \subseteq \text{prob-dom } PROB$
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 $fmdom' p \subseteq prob-dom\ PROB$

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 $a \in PROB$

shows $fmdom' (fst\ a) \subseteq prob-dom\ PROB$

using *assms FDOM-pre-subset-prob-dom surjective-pairing*

by *metis*

3.1.4 Properties of Valid Plans

lemma *valid-plan-valid-head*:

assumes $(h \# as \in valid-plans\ PROB)$

shows $h \in PROB$

using *assms valid-plans-def*

by *force*

lemma *valid-plan-valid-tail*:

assumes $(h \# as \in valid-plans\ PROB)$

shows $(as \in valid-plans\ PROB)$

using *assms*

by (*simp add: valid-plans-def*)

— TODO unwrap *simp* proof.

lemma *valid-plan-pre-subset-prob-dom-pair*:

assumes $as \in valid-plans\ PROB$

shows $(\forall a. ListMem\ a\ as \longrightarrow fmdom' (fst\ a) \subseteq (prob-dom\ PROB))$

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 $as1\ @\ as2 \in (valid-plans\ PROB)$

shows $as2 \in (valid-plans\ PROB)$

using *assms*

by (*simp add: valid-plans-def*)

lemma *valid-append-valid-pref*:
assumes $as1 @ as2 \in (\text{valid-plans } PROB)$
shows $as1 \in (\text{valid-plans } PROB)$
using *assms*
by (*simp add: valid-plans-def*)

lemma *valid-pref-suff-valid-append*:
assumes $as1 \in (\text{valid-plans } PROB) as2 \in (\text{valid-plans } PROB)$
shows $(as1 @ as2) \in (\text{valid-plans } PROB)$
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 $s0 \in (\text{valid-states } PROB) as \in (\text{valid-plans } PROB) \text{ListMem } h (\text{state-list } s0 as)$
shows $(\text{fmdom}' h = \text{fmdom}' s0)$
using *assms*
proof (*induction as arbitrary: PROB h s0*)
case *Nil*
have $h = s0$
using *Nil.premis(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
' $\text{ListMem } h (\text{state_list } (\text{state_succ } s0 a) as)$ '
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 $a \in PROB$
using *Cons.premis(2) valid-plan-valid-head*
by *fast*
then have $\text{fmdom}' (\text{snd } a) \subseteq \text{fmdom}' s0$
using *Cons.premis(1) FDOM-eff-subset-FDOM-valid-states-pair*
by *blast*
then have $\text{fmdom}' (\text{state-succ } s0 a) = \text{fmdom}' s0$
using *FDOM-state-succ[of - s0] Cons.premis(1) valid-states-def*


```

    by presburger
  }
note 1 = this
{
  have fmdom' s0 = prob-dom PROB
    using Cons.prem1 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.prem3 False
    by (simp add: ListMem-iff)
}
note 3 = this
{
  have as ∈ valid-plans PROB
    using Cons.prem2 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 $s \in \text{valid-states } PROB$ $a \in PROB$
shows $\text{state-succ } s \ a \in \text{valid-states } PROB$
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*:

$\text{last } ' ((\#) \ s \ ' \ \text{state-set } (\text{state-list } (\text{state-succ } \ s \ a) \ as))$
 $= \text{last } ' \ \text{state-set } (\text{state-list } (\text{state-succ } \ s \ a) \ as)$
by (*smt NIL-NOTIN-stateset image-cong image-image last-ConsR*)

lemma *lemma-1*:

fixes $as :: (('a, 'b) \ \text{fmap} \times ('a, 'b) \ \text{fmap}) \ \text{list} \ \mathbf{and} \ PPROB$
assumes $(s \in \text{valid-states } PROB) \ (as \in \text{valid-plans } PROB)$
shows $((\text{last } ' (\text{state-set } (\text{state-list } \ s \ as))) \subseteq \text{valid-states } PROB)$
using *assms*

proof (*induction as arbitrary: s PROB*)

— NOTE Base case simplifies to $\{s\} \subseteq \text{valid-states } PROB$ which itself follows directly from 1st assumption.

case (*Cons a as*)

Split the 'insert' term produced by $\text{state-set } (\text{state-list } \ s \ (a \ \# \ as))$ 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 $\text{last } [s] \in \text{valid-states } PROB$

using *Cons.premis(1)*

by *simp*

moreover {

{

have $a \in PROB$

using *Cons.premis(2) valid-plan-valid-head*

by *fast*

then have $\text{state-succ } \ s \ a \in \text{valid-states } PROB$

using *Cons.premis(1) lemma-1-i*

by *blast*

}

moreover have $as \in \text{valid-plans } PROB$

using *Cons.premis(2) valid-plan-valid-tail*

by *fast*

then have $(\text{last } ' \ \text{state-set } (\text{state-list } (\text{state-succ } \ s \ a) \ as)) \subseteq \text{valid-states } PROB$

using *calculation Cons.IH[of state-succ s a]*

by *presburger*

then have $(\text{last } ' ((\#) \ s \ ' \ \text{state-set } (\text{state-list } (\text{state-succ } \ s \ a) \ as))) \subseteq \text{valid-states } PROB$

```

      using lemma-1-ii
      by metis
    }
  ultimately have
    (last 'insert [s] ((#) s ' state-set (state-list (state-succ s a) as)))  $\subseteq$  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  $\in$  valid-states PROB) (as  $\in$  valid-plans PROB)  $\neg$ (as = [])
    (x  $\in$  state-set (state-list s as))
  shows (length x  $\leq$  (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  $\in$  state-set (state-list s as) (y  $\in$  state-set (state-list s as))  $\neg$ (x = y)
  shows  $\neg$ (length x = length y)
proof -
  have x  $\neq$  [] prefix x (state-list s as)
    using assms(1) state-set-thm
    by blast+
  moreover have y  $\neq$  [] 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 ⇒ 'b) ⇒ 'a set ⇒ 'b set ⇒ bool **where**
inj f A B ≡ (∀ x ∈ A. f x ∈ B) ∧ *inj-on f A*

— NOTE added lemma; refactored from 'not_eq_last_diff_paths'.

lemma *not-eq-last-diff-paths-i*:

fixes *s as PROB*
assumes *s ∈ valid-states PROB as ∈ valid-plans PROB x ∈ state-set (state-list s as)*
shows *last x ∈ valid-states PROB*
proof –
have *last x ∈ 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 ∈ valid-states PROB*) (*as ∈ valid-plans PROB*)
 ¬(*inj (last) (state-set (state-list s as)) (valid-states PROB)*)
shows ∃ l1. ∃ l2.
l1 ∈ state-set (state-list s as)
∧ l2 ∈ state-set (state-list s as)
∧ last l1 = last l2
∧ l1 ≠ l2

proof –

let *?S = state-set (state-list s as)*
have *1: ¬(∀ x ∈ ?S. last x ∈ valid-states PROB) = False*
using *assms(1, 2) not-eq-last-diff-paths-i*
by *blast*
 {
have
(¬(inj (last) ?S (valid-states PROB))) = (¬((∀ x ∈ ?S. ∀ y ∈ ?S. last x = last y
 → *x = y)))*
unfolding *inj-def inj-on-def*
using *1*
by *blast*
then have
(¬(inj (last) ?S (valid-states PROB)))

$$= (\exists x. \exists y. x \in ?S \wedge y \in ?S \wedge \text{last } x = \text{last } y \wedge x \neq y)$$

```

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  $(h \# slist) \in \text{state-set } (\text{state-list } s \text{ as})$ 
shows  $(h = s)$ 
using assms
by (induction as) auto

```

```

lemma cons-in-state-set-2:
fixes s slist h t
assumes  $(slist \neq []) ((s \# slist) \in \text{state-set } (\text{state-list } s (h \# t)))$ 
shows  $(slist \in \text{state-set } (\text{state-list } (\text{state-succ } s \ h) \ t))$ 
using assms
by (induction slist) auto

```

— TODO move up and replace 'FactoredSystem.lemma_1_i'?

```

lemma valid-action-valid-succ:
assumes  $h \in \text{PROB } s \in \text{valid-states } \text{PROB}$ 
shows  $(\text{state-succ } s \ h) \in \text{valid-states } \text{PROB}$ 
using assms lemma-1-i
by blast

```

```

lemma in-state-set-imp-eq-exec-prefix:
fixes slist as PROB s
assumes  $(as \neq []) (slist \neq []) (s \in \text{valid-states } \text{PROB}) (as \in \text{valid-plans } \text{PROB})$ 
 $(slist \in \text{state-set } (\text{state-list } s \ as))$ 
shows
 $(\exists as'. (\text{prefix } as' \ as) \wedge (\text{exec-plan } s \ as' = \text{last } slist) \wedge (\text{length } slist = \text{Suc } (\text{length } as')))$ 
using assms
proof (induction slist arbitrary: as s PROB)
case cons-1: (Cons a slist)
have  $1: s \# slist \in \text{state-set } (\text{state-list } s \ as)$ 
using cons-1.prem5 empty-list-nin-state-set
by auto
then show ?case
using cons-1
proof (cases as)
case cons-2: (Cons a' Ras)
then have  $a: \text{state-succ } s \ a' \in \text{valid-states } \text{PROB}$ 
using cons-1.prem3, 4 valid-action-valid-succ valid-plan-valid-head
by metis
then have  $b: R_{as} \in \text{valid-plans } \text{PROB}$ 
using cons-1.prem4 cons-2 valid-plan-valid-tail
by fast
then show ?thesis
proof (cases slist)
case Nil
then show ?thesis

```

```

    using cons-1.prem5(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  $\neq$  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  $\neq$  []
    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  $\implies$  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) > (2card (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) > 2card (fmdom' s)
    using assms(4)
    by linarith
  then have 1: card (state-set (state-list s as)) > 2card (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) = 2card (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  $\exists x. x \in (\text{last } ' ?S) \wedge x \notin ?T$ 
  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 ∈ last ' (state-set (state-list s as)) ∧ x ∉ valid-states
PROB
    using 5
    by presburger
  then have  $\neg(\forall x \in \text{state-set (state-list s as)}. \text{last } x \in \text{valid-states } \text{PROB})$ 
    by blast
  then have  $\neg \text{inj last (state-set (state-list (s :: 'a state) as)) (valid-states$ 
PROB)
    using inj-def
    by metis
  then have False
    using C
    by simp
}
ultimately have  $\neg \text{inj last (state-set (state-list (s :: 'a state) as)) (valid-states$ 
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  $\neg(2 \wedge \text{card (fmdom } ' s) - 1 < \text{length } as)$ 
    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 (fndom' 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  $\text{length } as \leq (2^{\wedge}(\text{card } (\text{prob-dom } PROB)) - 1)$ 
  using True
  by auto
then show ?thesis
  by blast
next
case False
then have  $\text{length } as > (2^{\wedge}(\text{card } (\text{prob-dom } PROB))) - 1$ 
  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
     $2^{\wedge} \text{card } (\text{prob-dom } PROB) - 1 < \text{length } p$ 
    then have  $(\exists p'. (\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ p}' \wedge \text{subseq } p' \text{ as}) \wedge \text{length } p' <$ 
length p)
    using assms(1, 2, 3) lemma-3 sublist-valid-is-valid
    by fastforce
  }
then have  $\forall p. \text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ p} \wedge \text{subseq } p \text{ as} \longrightarrow$ 
 $(\exists p'. (\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ p}' \wedge \text{subseq } p' \text{ as})$ 
 $\wedge \text{length } p' \leq 2^{\wedge} \text{card } (\text{prob-dom } PROB) - 1)$ 
using general-theorem where
   $P = \lambda(as'' :: 'a \text{ action list}). (\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}'') \wedge \text{subseq } as''$ 
as
  and  $l = (2^{\wedge}(\text{card } (\text{prob-dom } (PROB :: 'a \text{ problem})))) - 1$  and  $f = \text{length}$ 
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.prem*(2) *valid-action-valid-succ*
by *blast*
moreover have *as* ∈ *valid-plans* *PROB*
using *Cons.prem*(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.prem* *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.prem *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) \in *valid-states* $PROB$

shows $(\forall l \in$ *reachable-s* $PROB$ $s. l \in$ *valid-states* $PROB)$

proof –

have $1: \forall l \in$ *reachable-s* $PROB$ $s. \exists as. l =$ *exec-plan* s $as \wedge as \in$ *valid-plans* $PROB$

using *reachable-s-def*

by *fastforce*

{

fix l

assume $P1: l \in$ *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 $as \wedge as \in$ *valid-plans* $PROB$

using 1

by *blast*

then have *exec-plan* s $as \in$ *valid-states* $PROB$

using *assms a valid-as-valid-exec*

by *blast*

then have $l \in$ *valid-states* $PROB$

using a

by *simp*

}

then show $\forall l \in$ *reachable-s* $PROB$ $s. l \in$ *valid-states* $PROB$

by *blast*

qed

lemma *reachable-s-finite-thm-1*:

assumes $((s :: 'a$ state) \in *valid-states* $PROB)$

shows $($ *reachable-s* $PROB$ $s \subseteq$ *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) $(s \in$ *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*: $\square \in$ $($ *valid-plans* $PROB)$

by $($ *simp add: valid-plans-def $)$*

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

— TODO refactor
— NOTE added lemma

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

```

    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.prem(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.prem(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.prem(1, 2) valid-action-valid-succ valid-plan-valid-head
    by metis
    moreover have as ∈ valid-plans PROB
    using cons.prem(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  $\exists as'$ .
      last (a' # list) = exec-plan (state-succ s a) as'  $\wedge$  (as'  $\in$  (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'  $\in$  (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.prem(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  $\in$  valid-states PROB as  $\in$  valid-plans PROB
   $\neg$ (inj last (state-set (state-list s as)) (reachable-s PROB s))
  shows ( $\exists$  slist-1 slist-2.

```

```

    slist-1  $\in$  state-set (state-list s as)
     $\wedge$  slist-2  $\in$  state-set (state-list s as)
     $\wedge$  (last slist-1 = last slist-2)
     $\wedge$   $\neg$ (length slist-1 = length slist-2)
  )

```

proof –

```

  {
    fix x
    assume P1: x  $\in$  state-set (state-list s as)
    have a: last ' state-set (state-list s as)  $\subseteq$  reachable-s PROB s
      using assms(1, 2) lemma-1-reachability-s
      by fast
    then have  $\forall as$  PROB. s  $\in$  (valid-states PROB)  $\wedge$  as  $\in$  (valid-plans PROB)
 $\longrightarrow$  (last ' (state-set (state-list s as))  $\subseteq$  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  $C$ :  $\text{inj } f \text{ } s \text{ } t$ 
then have  $1$ :  $(\forall x \in s. f \ x \in t)$   $\text{inj-on } f \ s$ 
  unfolding  $\text{inj-def}$ 
  by  $\text{blast+}$ 
moreover {
  have  $f \ ' \ s \subseteq t$ 
    using  $1$ 
    by  $\text{fast}$ 
  then have  $\text{card } (f \ ' \ s) \leq \text{card } t$ 
    using  $\text{assms}(1)$   $\text{card-mono}$ 
    by  $\text{auto}$ 
  }
moreover have  $\text{card } (f \ ' \ s) = \text{card } s$ 
  using  $1$   $\text{card-image}$ 
  by  $\text{fast}$ 
ultimately have  $\text{False}$ 
  using  $\text{assms}(2)$ 
  by  $\text{linarith}$ 
}
then show  $?thesis$ 
  by  $\text{blast}$ 
qed

```

lemma $\text{lemma-2-reachability-s}$:

```

fixes  $PROB$  :: 'a  $\text{problem}$  and  $as \ s$ 
assumes  $\text{finite } PROB$   $(s \in \text{valid-states } PROB)$   $(as \in \text{valid-plans } PROB)$ 
   $(\text{length } as > \text{card } (\text{reachable-s } PROB \ s) - 1)$ 
shows  $(\exists as1 \ as2 \ as3.$ 
   $(as1 \ @ \ as2 \ @ \ as3 = as) \wedge (\text{exec-plan } s \ (as1 \ @ \ as2) = \text{exec-plan } s \ as1) \wedge \neg(as2$ 
   $= []))$ 
proof -
{
  have  $\text{Suc } (\text{length } as) > \text{card } (\text{reachable-s } PROB \ s)$ 
    using  $\text{assms}(4)$ 
    by  $\text{fastforce}$ 
  then have  $\text{card } (\text{state-set } (\text{state-list } s \ as)) > \text{card } (\text{reachable-s } PROB \ s)$ 
    using  $\text{card-state-set}$ 
    by  $\text{metis}$ 
}
note  $1 = \text{this}$ 
{
  have  $\text{finite } (\text{reachable-s } PROB \ s)$ 
    using  $\text{assms}(1, 2)$   $\text{reachable-s-finite-thm}$ 
    by  $\text{blast}$ 
  then have  $\neg(\text{inj last } (\text{state-set } (\text{state-list } s \ as)) \ (\text{reachable-s } PROB \ s))$ 
    using  $\text{assms}(4)$   $1$   $\text{lemma-2-reachability-s-i}$ 
    by  $\text{blast}$ 
}
note  $2 = \text{this}$ 

```

```

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 ( $\exists$  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* ∈ *valid-states PROB*) (*as* ∈ *valid-plans PROB*)

shows (∃ *as'*.

(*exec-plan s as* = *exec-plan s as'*) ∧ *subseq as' as*
 ∧ (*length as' ≤ (card (reachable-s PROB s) − 1)*))

proof (*cases length as ≤ 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* ∈ *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 (∃ *p'*. (*exec-plan s as = exec-plan s p' ∧ subseq p' as*) ∧ *length p' < length p*)

using *1*

by *blast*

}

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' ≤ card (reachable-s PROB s) − 1*)

using *general-theorem*[of $\lambda as''$. (*exec-plan s as = exec-plan s as''*) ∧ *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:  $\neg(\text{reachable-s } PROB\ s = \{\})$ 
  using empty-plan-is-valid reachable-s-def
  by blast

```

```

lemma card-reachable-s-non-zero:
  fixes s
  assumes finite (PROB :: 'a problem) (s  $\in$  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  $\in$  valid-states PROB) (as  $\in$  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  $\in$  valid-states PROB)
  shows ((reachable-s PROB s)  $\subseteq$  {fmempty})
proof –
  {
    fix x
    assume P1: x  $\in$  reachable-s PROB s
    then obtain as :: 'a action list where a:
      as  $\in$  valid-plans PROB x = exec-plan s as
    using reachable-s-def
    by blast
    then have as  $\in$  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)  $\subseteq$  {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  $\in$  valid-plans PROB) (subseq as2 as1)
  shows (as2  $\in$  valid-plans PROB)
  using assms
  by (auto simp add: sublist-valid-is-valid)

```

```

lemma fmsubset-eq:
  assumes s1  $\subseteq_f$  s2
  shows ( $\forall a. a \in |fmdom\ s1 \longrightarrow fmllookup\ s1\ a = fmllookup\ s2\ a$ )
  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  $\subseteq_f$  s2 s2  $\subseteq_f$  s3
  shows s1  $\subseteq_f$  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  $\subseteq_f$  s2
  shows (s0 ++ s1)  $\subseteq_f$  (s0 ++ s2)
proof –
{
  assume C:  $\neg((s0 ++ s1) \subseteq_f (s0 ++ s2))$ 
  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) ∧ fmllookup (s1 ++f s0) ≠ fmllookup (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 fmllookup s1 a = fmllookup s2 a
      by (meson assms(1) calculation fmsubset-eq)
    moreover have fmllookup (s0 ++f s1) a = fmllookup s1 a
      by (simp add: True)
    moreover have a |∈| fmdom s2
      using True calculation(2) fmdom-notD by fastforce
    moreover have fmllookup (s0 ++f s2) a = fmllookup s2 a
      by (simp add: calculation(4))
    moreover have fmllookup (s0 ++f s1) a = fmllookup (s0 ++f s2) a
      using calculation(2, 3, 5)
      by auto
    ultimately show ?thesis
      by (smt 1 2 C assms domIff fmllookup-add fmsubset.rep-eq map-le-def)
  next
  case False
  moreover have fmllookup (s0 ++f s1) a = fmllookup 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 ⟨fmllookup (s0 ++f s1) a = fmllookup 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  $(fmdom'\ (fst\ a) \subseteq fmdom'\ s1)\ (s1 \subseteq_f\ s2)$ 
  shows  $(state-succ\ s1\ a \subseteq_f\ state-succ\ s2\ a)$ 
  using assms
  unfolding state-succ-def
proof (auto)
  assume  $P1: fmdom'\ (fst\ a) \subseteq fmdom'\ s1\ s1 \subseteq_f\ s2\ fst\ a \subseteq_f\ s1\ fst\ a \subseteq_f\ s2$ 
  then show  $snd\ a\ ++\ s1 \subseteq_f\ snd\ a\ ++\ s2$ 
    using submap-imp-state-succ-submap-b
    by fast
next
  assume  $P2: fmdom'\ (fst\ a) \subseteq fmdom'\ s1\ s1 \subseteq_f\ s2\ fst\ a \subseteq_f\ s1\ \neg\ fst\ a \subseteq_f\ s2$ 
  then show  $snd\ a\ ++\ s1 \subseteq_f\ s2$ 
    using submap-imp-state-succ-submap-a
    by blast
next
  assume  $P3: fmdom'\ (fst\ a) \subseteq fmdom'\ s1\ s1 \subseteq_f\ s2\ \neg\ fst\ a \subseteq_f\ s1\ fst\ a \subseteq_f\ s2$ 
  {
    have  $a: fmlookup\ s1 \subseteq_m\ fmlookup\ s2$ 
      using  $P3(2)$  fmsubset.rep-eq
      by blast
    {
      have  $\neg(fmlookup\ (fst\ a) \subseteq_m\ fmlookup\ s1)$ 
        using  $P3(3)$  fmsubset.rep-eq

```

```

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

```

— NOTE added lemma.

— TODO refactor.

lemma *valid-as-submap-init-submap-exec-i:*

fixes $s a$

shows $\text{fmdom}' s \subseteq \text{fmdom}' (\text{state-succ } s a)$

proof (*cases* $\text{fst } a \subseteq_f s$)

case *True*

then have $\text{state-succ } s a = s ++_f (\text{snd } a)$

unfolding *state-succ-def*

using *fmap-add-ltr-def*

by *auto*

then have $\text{fmdom}' (\text{state-succ } s a) = \text{fmdom}' s \cup \text{fmdom}' (\text{snd } a)$

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 $s1 s2 :: 'a \text{ state}$

assumes $(s1 \subseteq_f s2) (\forall a. \text{ListMem } a \text{ as} \longrightarrow (\text{fmdom}' (\text{fst } a) \subseteq \text{fmdom}' s1))$

```

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)  $\subseteq$  fmdom' s1
    using Cons.prem(2)
    by blast
  then have state-succ s1 a  $\subseteq_f$  state-succ s2 a
    using Cons.prem(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)  $\subseteq$  fmdom' s1
    using Cons.prem(2)
    by blast
  then have fmdom' s1  $\subseteq$  fmdom' (state-succ s1 a)
    using valid-as-submap-init-submap-exec-i
    by metis
  then have fmdom' (fst b)  $\subseteq$  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* \in *valid-plans PROB*) (*ListMem a as*)

shows *a* \in *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-empty:*

```

fixes PROB :: (('a, 'b) fmap × ('a, 'b) fmap) set
assumes finite PROB
shows  $\exists s. s \in (\text{valid-states } PROB)$ 
unfolding valid-states-def
using fmchoice["OF FINITE-prob-dom[OF assms], where  $Q = \lambda -. \text{True}$ "]
by auto

```

```

lemma empty-prob-dom-single-val-state:
assumes (prob-dom PROB = {})
shows ( $\exists s. \text{valid-states } PROB = \{s\}$ )
proof –
{
  assume C:  $\neg(\exists s. \text{valid-states } PROB = \{s\})$ 
  then have valid-states PROB = {s. fmdom' s = {}}
    using assms
    by (simp add: valid-states-def)
  then have  $\exists s. \text{valid-states } PROB = \{s\}$ 
    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 = {}) ( $s \in \text{valid-states } PROB$ ) ( $as \in \text{valid-plans } PROB$ )
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 ( $PROB = \{(fmempty, fmempty)\} \vee PROB = \{\}$ )
using assms
proof (cases PROB = {})
case False
have  $\bigcup((\lambda(s1, s2). \text{fmdom}' s1 \cup \text{fmdom}' s2) ' PROB) = \{\}$ 
  using assms
  by (simp add: prob-dom-def action-dom-def)
then have  $1:\forall a \in PROB. (\lambda(s1, s2). \text{fmdom}' s1 \cup \text{fmdom}' s2) a = \{\}$ 

```

```

using Union-empty-conv
by auto
{
  fix a
  assume P1: a ∈ PROB
  then have  $(\lambda(s1, s2). \text{fmdom}' s1 \cup \text{fmdom}' s2) a = \{\}$ 
    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  $(\text{fmdom}' (\text{snd } a) \cap vs) = \{\}$ 
  shows  $(\text{fmrestrict-set } vs \ s = \text{fmrestrict-set } vs \ (\text{state-succ } s \ a))$ 
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  $\times$  ('a, 'b) fmap and vs
assumes (fmdom' (fmrestrict-set vs (snd a)) = {})
shows ((fmrestrict-set vs (state-succ s a)) = (fmrestrict-set vs s))

```

proof –

```

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:  $x \in \text{fmdom}' (\text{snd } a) \wedge x \in \text{vs}$ 
  then have a:  $x \in \text{fmdom}' (\text{snd } a) \wedge x \in \text{vs}$ 
    using C
    by blast+
  then have fmlookup (snd a) x  $\neq$  None
    using fmdom'-notI
    by metis
  then have fmlookup (fmrestrict-set vs (snd a)) x  $\neq$  None
    using a(2)
    by force
  then have  $x \in \text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a))$ 
    using fmdom'-notD
    by metis
  then have fmdom' (fmrestrict-set vs (snd a))  $\neq$  {}
    by blast
  then have False
    using assms
    by blast
}
then have  $\forall x. \neg(x \in \text{fmdom}' (\text{snd } a) \wedge x \in \text{vs})$ 
  by blast
then have 1:  $\text{fmdom}' (\text{snd } a) \cap \text{vs} = \{\}$ 
  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 ⊆f s) = (fst a' ⊆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 $PROB1 \subseteq PROB2$
shows $(\text{prob-dom } PROB1 \subseteq \text{prob-dom } PROB2)$
using *assms*
by (auto simp add: prob-dom-def)

lemma *state-succ-valid-act-disjoint*:
assumes $(a \in PROB) (vs \cap (\text{prob-dom } PROB) = \{\})$
shows $(\text{fmrestrict-set } vs (\text{state-succ } s a) = \text{fmrestrict-set } vs s)$
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 $(vs \cap (\text{prob-dom } PROB) = \{\}) (as \in \text{valid-plans } PROB)$
shows $(\text{fmrestrict-set } vs (\text{exec-plan } s as) = \text{fmrestrict-set } vs s)$
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**
 $\text{state-successors } PROB s \equiv ((\text{state-succ } s ' PROB) - \{s\})$

3.3 State Spaces

definition *stateSpace* **where**
 $\text{stateSpace } ss vs \equiv (\forall s. s \in ss \longrightarrow (\text{fmdom}' s = vs))$

lemma *EQ-SS-DOM*:
assumes $\neg(ss = \{\}) (\text{stateSpace } ss vs1) (\text{stateSpace } ss vs2)$
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 $ss :: ('a, \text{bool}) \text{fmap set}$
assumes $\neg(ss = \{\})$ (*stateSpace ss domain*)
shows *finite ss*

proof —

have $1: \text{stateSpace ss domain} = (\forall s. s \in ss \longrightarrow (\text{fmdom}' s = \text{domain}))$
by (*simp add: stateSpace-def*)

{
fix s
assume $P1: s \in ss$
have $\text{fmdom}' s = \text{domain}$
using *assms 1 P1*
by *blast*
then have $s \in \{s. \text{fmdom}' s = \text{domain}\}$
by *auto*

}
then have $2: ss \subseteq \{s. \text{fmdom}' s = \text{domain}\}$
by *blast*

— TODO add lemma (*finite (fmdom' s)*)

then have *finite domain*

using 1 *assms*
by *fastforce*

then have *finite* $\{s :: 'a \text{state}. \text{fmdom}' s = \text{domain}\}$
using *FINITE-states*
by *blast*

then show *?thesis*
using 2 *finite-subset*
by *auto*

qed

lemma *disjoint-effects-no-effects*:

fixes s
assumes $(\forall a. \text{ListMem } a \text{ as} \longrightarrow (\text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a)) = \{\}))$
shows $(\text{fmrestrict-set vs } (\text{exec-plan } s \text{ as}) = (\text{fmrestrict-set vs } s))$
using *assms*

proof (*induction as arbitrary: s vs*)

case (*Cons a as*)

then have *ListMem a (a # as)*

using *elem*
by *fast*

then have $\text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a)) = \{\}$
using *Cons.prems(1)*

by *blast*

then have $\text{fmrestrict-set vs } (\text{state-succ } s \text{ a}) = \text{fmrestrict-set vs } s$

using *sat-precond-as-proj-3*
by *blast*

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

3.4 Needed Asses

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

action-needed-vars a s $\equiv \{v. (v \in \text{fmdom}' s) \wedge (v \in \text{fmdom}' (\text{fst } a))$
 $\wedge (\text{fmlookup } (\text{fst } a) v = \text{fmlookup } s v)\}$
 — NOTE name shortened to 'action_needed_asses'.

definition *action-needed-asses* **where**

action-needed-asses a s $\equiv \text{fmrestrict-set } (\text{action-needed-vars } a) s$

— 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*
 \neg *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* \neg
fst a \subseteq_f *s1*
fst a \subseteq_f *s2*
let *?vs1* $= \{v \in \text{fmdom}' s1. v \in \text{fmdom}' (\text{fst } a) \wedge \text{fmlookup } (\text{fst } a) v = \text{fmlookup } s1 v\}$
let *?vs2* $= \{v \in \text{fmdom}' s2. v \in \text{fmdom}' (\text{fst } a) \wedge \text{fmlookup } (\text{fst } a) v = \text{fmlookup } s2 v\}$
let *?f* $= \text{fmrestrict-set } ?vs1$ *s1*
let *?g* $= \text{fmrestrict-set } ?vs2$ *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 $v \in \text{fmdom}' (\text{action-needed-asses } a s)$

shows

$\text{fmlookup } (\text{action-needed-asses } a s) v = \text{fmlookup } s v$

$\wedge \text{fmlookup } (\text{action-needed-asses } a \ s) \ v = \text{fmlookup } (\text{fst } a) \ v$
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 $v \in \text{fmdom}' \ f \ f \subseteq_f \ g$
shows $\text{fmlookup } f \ v = \text{fmlookup } g \ v$
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
 $\text{fmdom}' (\text{action-needed-asses } a \ s)$
 $= \{v \in \text{fmdom}' \ s. \ v \in \text{fmdom}' (\text{fst } a) \wedge \text{fmlookup } (\text{fst } a) \ v = \text{fmlookup } s \ v\}$
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 $v \in \text{fmdom}' (\text{action-needed-asses } a \ s)$
shows
 $\text{fmlookup } (\text{action-needed-asses } a \ s) \ v = \text{fmlookup } s \ v$
 $\wedge \text{fmlookup } (\text{action-needed-asses } a \ s) \ v = \text{fmlookup } (\text{fst } a) \ v$
 $\wedge \text{fmlookup } (\text{fst } a) \ v = \text{fmlookup } s \ v$
using *assms*

proof —

have 1: $v \in \{v \in \text{fmdom}' \ s. \ v \in \text{fmdom}' (\text{fst } a) \wedge \text{fmlookup } (\text{fst } a) \ v = \text{fmlookup } s \ v\}$

using *assms as-needed-asses-submap-exec-iii*

by *metis*

then have 2: $\text{fmlookup } (\text{action-needed-asses } a \ s) \ v = \text{fmlookup } s \ v$

unfolding *action-needed-asses-def action-needed-vars-def*

by *force*

moreover have 3: $\text{fmlookup } (\text{action-needed-asses } a \ s) \ v = \text{fmlookup } (\text{fst } a) \ v$

using 1 2

by *simp*

moreover have $\text{fmlookup } (\text{fst } a) \ v = \text{fmlookup } s \ v$

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 $v \in \text{fmdom}' f f \subseteq_f g$
 shows $v \in \text{fmdom}' g$

proof —

obtain b where 1: $\text{fmlookup } f v = b \ b \neq \text{None}$
 using *assms*(1)
 by (*meson* *fmdom'-notI*)
 then have $\text{fmlookup } g v = b$
 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 $v \in \text{fmdom}' (\text{action-needed-asses } a s1)$
 $(\text{action-needed-asses } a s1) \subseteq_f (\text{action-needed-asses } a s2)$
 shows
 $(\text{fmlookup } (\text{action-needed-asses } a s1) v) = \text{fmlookup } (\text{fst } a) v$
 $\wedge (\text{fmlookup } (\text{action-needed-asses } a s2) v) = \text{fmlookup } (\text{fst } a) v \wedge$
 $\text{fmlookup } s1 v = \text{fmlookup } (\text{fst } a) v \wedge \text{fmlookup } s2 v = \text{fmlookup } (\text{fst } a) v$
 using *assms*

proof —

have 1:
 $\text{fmlookup } (\text{action-needed-asses } a s1) v = \text{fmlookup } s1 v$
 $\text{fmlookup } (\text{action-needed-asses } a s1) v = \text{fmlookup } (\text{fst } a) v$
 $\text{fmlookup } (\text{fst } a) v = \text{fmlookup } s1 v$
 using *as-needed-asses-submap-exec-iv*[*OF* *assms*(1)]
 by *blast+*

moreover {

have $\text{fmlookup } (\text{action-needed-asses } a s1) v = \text{fmlookup } (\text{action-needed-asses } a s2) v$

using *as-needed-asses-submap-exec-ii*[*OF* *assms*]
 by *simp*

then have $\text{fmlookup } (\text{action-needed-asses } a s2) v = \text{fmlookup } (\text{fst } a) v$

using 1(2)

by *argo*

}

note 2 = *this*

```

moreover {
  have  $v \in \text{fmdom}' (\text{action-needed-asses } a \ s2)$ 
    using as-needed-asses-submap-exec-v[OF assms]
    by simp
  then have  $\text{fmlookup } s2 \ v = \text{fmlookup } (\text{action-needed-asses } a \ s2) \ v$ 
    using as-needed-asses-submap-exec-i
    by metis
  also have  $\dots = \text{fmlookup } (\text{fst } a) \ v$ 
    using  $\mathcal{Q}$ 
    by simp
  finally have  $\text{fmlookup } s2 \ v = \text{fmlookup } (\text{fst } a) \ v$ 
    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  $\forall v \in \text{fmdom}' f. \text{fmlookup } f \ v = \text{fmlookup } g \ v$ 
  shows  $f \subseteq_f g$ 
proof –
  {
    fix  $v$ 
    assume  $a: v \in \text{fmdom}' f$ 
    then have  $v \in \text{dom } (\text{fmlookup } f)$ 
      by simp
    moreover have  $\text{fmlookup } f \ v = \text{fmlookup } g \ v$ 
      using assms a
      by blast
    ultimately have  $v \in \text{dom } (\text{fmlookup } f) \longrightarrow \text{fmlookup } f \ v = \text{fmlookup } g \ v$ 
      by blast
  }
  then have  $\text{fmlookup } f \subseteq_m \text{fmlookup } g$ 
    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  $\forall v \in \text{fmdom}' f. \text{fmlookup } f \ v = \text{fmlookup } g \ v$ 
proof –
  have  $1: \text{fmlookup } f \subseteq_m \text{fmlookup } g$ 

```

```

using assms
by (simp add: fmsubset.rep-eq)
{
  fix v
  assume  $v \in \text{fmdom}' f$ 
  then have  $v \in \text{dom} (\text{fmlookup } f)$ 
    by simp
  then have  $\text{fmlookup } f v = \text{fmlookup } g v$ 
    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  $\text{fmdom}' f \subseteq \text{fmdom}' g$ 
  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  $f \subseteq_f g = (\forall v \in \text{fmdom}' f. \text{fmlookup } f v = \text{fmlookup } g v)$ 
  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  $v \in \text{fmdom}' (\text{action-needed-asses } a f)$ 
  shows  $v \in \text{fmdom}' (\text{fst } a) \wedge v \in \text{fmdom}' f \wedge \text{fmlookup } (\text{fst } a) v = \text{fmlookup } f v$ 
  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  $v \in \text{fmdom}' (\text{action-needed-asses } a (f ++ g)) \wedge v \in \text{fmdom}' f$ 
  shows
     $\text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v = \text{fmlookup } f v$ 
     $\wedge \text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v = \text{fmlookup } (\text{fst } a) v$ 

```

```

proof –
  have 1:  $v \in \{v \in \text{fmdom}' (f ++ g). v \in \text{fmdom}' (\text{fst } a) \wedge \text{fmlookup } (\text{fst } a) v = \text{fmlookup } (f ++ g) v\}$ 
    using as-needed-asses-submap-exec-x[OF assms(1)]
    by blast
  {
    have  $v \in \text{fmdom } f$ 
      using assms(2)
      by (meson fmdom'-notI fmdom-notD)
    then have  $\text{fmlookup } (f ++ g) v = \text{fmlookup } f v$ 
      unfolding fmap-add-ltr-def fmlookup-add
      by simp
  }
  note 2 = this
  {
    have  $\text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v = \text{fmlookup } (f ++ g) v$ 
      unfolding action-needed-asses-def action-needed-vars-def
      using 1
      by force
    then have  $\text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v = \text{fmlookup } f v$ 
      using 2
      by simp
  }
  note 3 = this
  moreover {
    have  $\text{fmlookup } (\text{fst } a) v = \text{fmlookup } (f ++ g) v$ 
      using 1
      by simp
    also have  $\dots = \text{fmlookup } f v$ 
      using 2
      by simp
    also have  $\dots = \text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v$ 
      using 3
      by simp
    finally have  $\text{fmlookup } (\text{action-needed-asses } a (f ++ g)) v = \text{fmlookup } (\text{fst } a) v$ 
      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 $v \in \text{fmdom}' f$

shows $\text{fmlookup } (f ++ g) v = \text{fmlookup } f v$

proof –

```

have  $v \in | fdom\ f$ 
  using  $assms(1)\ fdom'\text{-notI}\ fdom\text{-notD}$ 
  by metis
then show ?thesis
  unfolding  $fmap\text{-add}\text{-ltr}\text{-def}$ 
  using  $fmlookup\text{-add}$ 
  by force
qed

```

— NOTE added lemma.

```

lemma  $as\text{-needed}\text{-asses}\text{-submap}\text{-exec}\text{-xii}'$ :
  fixes  $f\ g\ v$ 
  assumes  $v \notin fdom'\ f\ v \in fdom'\ g$ 
  shows  $fmlookup\ (f\ ++\ g)\ v = fmlookup\ g\ v$ 
proof  $-$ 
  have  $\neg(v \in | fdom\ f)$ 
    using  $assms(1)\ fdom'\text{-notI}\ fdom\text{-notD}$ 
    by fastforce
  moreover have  $v \in | fdom\ g$ 
    using  $assms(2)\ fdom'\text{-notI}\ fdom\text{-notD}$ 
    by metis
  ultimately show ?thesis
    unfolding  $fmap\text{-add}\text{-ltr}\text{-def}$ 
    using  $fmlookup\text{-add}$ 
    by simp
qed

```

— NOTE showcase.

```

lemma  $as\text{-needed}\text{-asses}\text{-submap}\text{-exec}$ :
  fixes  $s1\ s2$ 
  assumes  $(s1 \subseteq_f s2)$ 
   $(\forall a. ListMem\ a\ as \longrightarrow (action\text{-needed}\text{-asses}\ a\ s2 \subseteq_f action\text{-needed}\text{-asses}\ a\ s1))$ 
  shows  $(exec\text{-plan}\ s1\ as \subseteq_f exec\text{-plan}\ s2\ as)$ 
  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  $action\text{-needed}\text{-asses}\ a\ s2 \subseteq_f action\text{-needed}\text{-asses}\ a\ s1$ 
      using  $Cons.prems(2)\ elem$ 
      by metis
    then have  $state\text{-succ}\ s1\ a \subseteq_f state\text{-succ}\ s2\ a$ 
      using  $Cons.prems(1)\ act\text{-needed}\text{-asses}\text{-submap}\text{-succ}\text{-submap}$ 
      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
 $\subseteq_f$  s2
  fst a  $\subseteq_f$  s1
  have i: action-needed-asses a' s2  $\subseteq_f$  action-needed-asses a' s1
  using Cons.prem(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  $I: v \in fmdom'\ s1\ v \in fmdom'\ s2$ 
  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  $\langle fmlookup\ (snd\ a\ ++\ s1)\ v = fmlookup\ s1\ v \rangle$ 
    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
  fmaplookup-restrict-set

```

```

      by simp
    }
  ultimately have fmllookup ?g v = fmllookup (action-needed-asses a' s1)
v
    by argo
  }
  note III = this
  {
    have fmllookup ?f v = fmllookup (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 ⟨fmllookup ?f v = fmllookup (snd a ++ s2) v⟩
    have α: fmllookup ?f v = fmllookup s2 v
      using as-needed-asses-submap-exec-xii'[OF False I(2)]
      by argo
    ultimately have fmllookup (snd a ++ s2) v = fmllookup s2 v
      by argo
    moreover {
      from ⟨fmllookup (snd a ++ s2) v = fmllookup s2 v⟩
      have fmllookup (fst a') v = fmllookup s2 v
        using B(1)
        by argo
      then have fmllookup (action-needed-asses a' s2) v = fmllookup s2 v
        using B(1) I(2)
        unfolding action-needed-asses-def action-needed-vars-def
          fmllookup-restrict-set
        by simp
    }
    ultimately have fmllookup ?f v = fmllookup (action-needed-asses a' s2)
v
      using α
      by argo
  }
  note III = this
  {
    have v ∈ fmdom' (action-needed-asses a' s2)
    proof -
      have fmllookup (fst a') v = fmllookup s1 v
        by (simp add: A False I(1) as-needed-asses-submap-exec-xii')
      then show ?thesis
        by (simp add: A Cons.prem(1) I(1, 2)
          as-needed-asses-submap-exec-ii as-needed-asses-submap-exec-iii)
    qed
    then have
      fmllookup (action-needed-asses a' s2) v
      = fmllookup (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 fmllookup ?f v = fmllookup (action-needed-asses a' s2) v
      using III
      by simp
    also have ... = fmllookup (action-needed-asses a' s1) v
      using IV
      by simp
    finally have ... = fmllookup ?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)
  ∧ fmllookup (fst a') v = fmllookup (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)) ∨ ¬(fmllookup (fst a') v = fmllookup (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
      fmllookup (snd a ++ s2) v = fmllookup (snd a) v
      fmllookup (snd a ++ s1) v = fmllookup (snd a) v
      using as-needed-asses-submap-exec-xii
      by fast+
    then have fmllookup (snd a ++ s1) v = fmllookup (snd a ++ s2) v
      by auto
    also have ... = fmllookup (fst a') v
      using A
  }
  }
end

```

```

    by simp
    finally have fmllookup (snd a ++ s1) v = fmllookup (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 fmllookup ?f v ≠ None
    by (meson fmdom'-notI)
  moreover from false
  have fmllookup ?g v = None
    by (simp add: fmdom'-notD)
  ultimately have fmllookup ?f v ≠ fmllookup ?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
      fmllookup (state-succ s2 a) v = fmllookup s2 v
      using as-needed-asses-submap-exec-xii'[OF False I]
      by simp
    ultimately have fmllookup ?f v = fmllookup (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 fmllookup (state-succ s1 a) v = fmllookup s1 v
      unfolding fmap-add-ltr-def
      using fmllookup-add
      by (simp add: fmdom'-alt-def)
    ultimately have fmllookup ?g v = fmllookup (action-needed-asses a'
s1) v
      unfolding action-needed-asses-def action-needed-vars-def
      using FDOM-state-succ-subset

```

```

    by auto
  }
  moreover {
    have  $v \in \text{fmdom}' (\text{action-needed-asses } a' s2)$ 
    proof -
      have  $v \in \text{fmdom}' s2 \cup \text{fmdom}' (\text{snd } a)$ 
      by (metis (no-types) A FDOM-state-succ-subset P-1-1(2))
    then show ?thesis
  by (simp add: A False as-needed-asses-submap-exec-iii as-needed-asses-submap-exec-xii')
  qed
  then have
     $\text{fmlookup } (\text{action-needed-asses } a' s2) v$ 
    =  $\text{fmlookup } (\text{action-needed-asses } a' s1) v$ 
  using  $i$  as-needed-asses-submap-exec-ix[of  $\text{action-needed-asses } a' s2$ 
     $\text{action-needed-asses } a' s1$ ]
  by blast
  }
  ultimately have  $\text{fmlookup } ?f v = \text{fmlookup } ?g v$ 
  by simp
}
ultimately show ?thesis
by simp
qed
qed
next
assume  $P2: v \in \text{fmdom}' (\text{action-needed-asses } a' (\text{snd } a ++ s2)) \text{fst } a \subseteq_f$ 
 $s2$ 
   $\neg \text{fst } a \subseteq_f s1$ 
then show
   $\text{fmlookup } (\text{action-needed-asses } a' (\text{snd } a ++ s2)) v$ 
  =  $\text{fmlookup } (\text{action-needed-asses } a' s1) v$ 
proof -
  obtain  $aa :: ('a, 'b) \text{fmap} \Rightarrow ('a, 'b) \text{fmap} \Rightarrow 'a$  where
     $\forall x0 x1. (\exists v2. v2 \in \text{fmdom}' x1$ 
       $\wedge \text{fmlookup } x1 v2 \neq \text{fmlookup } x0 v2) = (aa x0 x1 \in \text{fmdom}' x1$ 
       $\wedge \text{fmlookup } x1 (aa x0 x1) \neq \text{fmlookup } x0 (aa x0 x1))$ 
    by moura
  then have  $f1: \forall f fa. aa fa f \in \text{fmdom}' f$ 
     $\wedge \text{fmlookup } f (aa fa f) \neq \text{fmlookup } fa (aa fa f) \vee f \subseteq_f fa$ 
  by (meson as-needed-asses-submap-exec-vii)
  then have  $f2: aa s1 (\text{fst } a) \in \text{fmdom}' (\text{fst } a)$ 
     $\wedge \text{fmlookup } (\text{fst } a) (aa s1 (\text{fst } a)) \neq \text{fmlookup } s1 (aa s1 (\text{fst } a))$ 
  using  $P2(3)$  by blast
  then have  $aa s1 (\text{fst } a) \in \text{fmdom}' s2$ 
  by (metis (full-types)  $P2(2)$  as-needed-asses-submap-exec-v)
  then have  $aa s1 (\text{fst } a) \in \text{fmdom}' (\text{action-needed-asses } a s2)$ 
  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.prem1(2) P2(3) as-needed-asses-submap-exec-vi
elem)
  qed
  next
    assume P3:  $v \in \text{fndom}'(\text{action-needed-asses } a' s2) \neg \text{fst } a \subseteq_f s2 \text{fst } a \subseteq_f$ 
s1
  then show
    fmllookup (action-needed-asses a' s2) v
    = fmllookup (action-needed-asses a' (snd a ++ s1)) v
  using Cons.prem1(1) submap-imp-state-succ-submap-a
  by blast
  next
    assume P4:  $v \in \text{fndom}'(\text{action-needed-asses } a' s2) \neg \text{fst } a \subseteq_f s2 \neg \text{fst } a$ 
 $\subseteq_f s1$ 
  then show
    fmllookup (action-needed-asses a' s2) v
    = fmllookup (action-needed-asses a' s1) v
  by (simp add: Cons.prem1(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 $PROB\ s \equiv (\bigcup \{\text{action-needed-vars } a\ s \mid a. a \in PROB\})$

— NOTE name shortened.

definition *system-needed-asses* **where**

system-needed-asses $PROB\ s \equiv (\text{fmrestrict-set } (\text{system-needed-vars } PROB\ s)\ s)$

lemma *action-needed-vars-subset-sys-needed-vars-subset*:

assumes $(a \in PROB)$

shows $(\text{action-needed-vars } a\ s \subseteq \text{system-needed-vars } PROB\ s)$

using *assms*

by $(\text{auto simp: system-needed-vars-def})$ *(metis surjective-pairing)*

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

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

using *map-le-def*
by *blast*
then show ($action\text{-}needed\text{-}asses\ a\ s \subseteq_f\ system\text{-}needed\text{-}asses\ PROB\ s$)
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 ($action\text{-}needed\text{-}vars\ a\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s) = action\text{-}needed\text{-}vars\ a\ s$)
proof –
let $?A = \{v \in fmdom'\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s)\}.$

```

    v ∈ fmdom' (fst a)
    ∧ fmllookup (fst a) v = fmllookup (fmrestrict-set (system-needed-vars PROB s)
s) v}
let ?B={v ∈ fmdom' s. v ∈ fmdom' (fst a) ∧ fmllookup (fst a) v = fmllookup s v}
{
  fix v
  assume v ∈ ?A
  then have i: v ∈ fmdom' (fmrestrict-set (system-needed-vars PROB s) s) v ∈
fmdom' (fst a)
    fmllookup (fst a) v = fmllookup (fmrestrict-set (system-needed-vars PROB s)
s) v
  by blast+
  then have v ∈ fmdom' s
    by (simp add: fmdom'-restrict-set-precise)
  moreover have fmllookup (fst a) v = fmllookup s v
    using i(2, 3) fmdom'-notI
    by force
  ultimately have v ∈ ?B
    using i
    by blast
}
then have 1: ?A ⊆ ?B
  by blast
{
  fix v
  assume P: v ∈ ?B
  then have ii: v ∈ fmdom' s v ∈ fmdom' (fst a) fmllookup (fst a) v = fmllookup
s v
    by blast+
  moreover {
    have ∃ s'. v ∈ s' ∧ (∃ a. (s' = action-needed-vars a s) ∧ a ∈ PROB)
      unfolding action-needed-vars-def
      using assms P action-needed-vars-def
      bymetis
    then obtain s' where α: v ∈ s' (∃ a. (s' = action-needed-vars a s) ∧ a ∈
PROB)
      by blast
    moreover obtain a' where s' = action-needed-vars a' s a' ∈ PROB
      using α
      by blast
    ultimately have v ∈ 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 fmllookup (fst a) v = fmllookup (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  $?f' = \text{fmrestrict-set } A f$ 
    let  $?f'' = \text{fmrestrict-set } A (\text{fmrestrict-set } B f)$ 
    assume  $C: ?f'' \neq ?f'$ 
    then obtain  $v$  where 1:  $\text{fmlookup } ?f'' v \neq \text{fmlookup } ?f' v$ 
    by (meson fmap-ext)
    then have False
    proof (cases v ∈ A)
    case True
    have  $\text{fmlookup } ?f'' v = \text{fmlookup } (\text{fmrestrict-set } B f) v$ 
    using True fmlookup-restrict-set
    by simp
    moreover have  $\text{fmlookup } (\text{fmrestrict-set } B f) v = \text{fmlookup } ?f' v$ 
    using True assms(1)
    by auto
    ultimately show ?thesis
    using 1
    by argo
  next
    case False
    then have  $\text{fmlookup } ?f' v = \text{None } \text{fmlookup } ?f'' v = \text{None}$ 
    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 ($action\text{-}needed\text{-}asses\ a\ (system\text{-}needed\text{-}asses\ PROB\ s) = action\text{-}needed\text{-}asses\ a\ s$)

proof –

{

have $action\text{-}needed\text{-}vars\ a\ s \subseteq system\text{-}needed\text{-}vars\ PROB\ s$

using $action\text{-}needed\text{-}vars\text{-}subset\text{-}sys\text{-}needed\text{-}vars\text{-}subset[OF\ assms]$

by *simp*

then have

$fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ s)\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s) =$

$fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ s)\ s$

using $system\text{-}needed\text{-}asses\text{-}include\text{-}action\text{-}needed\text{-}asses\text{-}i$

by *fast*

}

moreover

{

have

$action\text{-}needed\text{-}vars\ a\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s) = action\text{-}needed\text{-}vars\ a\ s$

using $system\text{-}needed\text{-}asses\text{-}include\text{-}action\text{-}needed\text{-}asses\text{-}1[OF\ assms]$

by *simp*

then have $fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s))$

$(fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s) =$

$fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ s)\ s$

$\longleftrightarrow fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ s)\ (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s) =$

$fmrestrict\text{-}set\ (action\text{-}needed\text{-}vars\ a\ s)\ s$

by *simp*

}

ultimately show *?thesis*

unfolding $action\text{-}needed\text{-}asses\text{-}def\ system\text{-}needed\text{-}asses\text{-}def$

by *simp*

qed

lemma *system-needed-asses-submap:*

$system\text{-}needed\text{-}asses\ PROB\ s \subseteq_f\ s$

proof –

{

fix x

assume $P: x \in dom\ (fmlookup\ (system\text{-}needed\text{-}asses\ PROB\ s))$

then have $system\text{-}needed\text{-}asses\ PROB\ s = (fmrestrict\text{-}set\ (system\text{-}needed\text{-}vars\ PROB\ s)\ s)$


```

    by (simp add: system-needed-asses-def)
  then have fmllookup (system-needed-asses PROB s) x = fmllookup s x
    using P
    by (auto simp: fmdom'-restrict-set-precise)
}
then have fmllookup (system-needed-asses PROB s)  $\subseteq_m$  fmllookup 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  $\in$  valid-plans PROB)
  shows (exec-plan (system-needed-asses PROB s) as  $\subseteq_f$  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 sequences 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

```

| $\text{sat-precond-as } s (a \# as) = (\text{fst } a \subseteq_f s \wedge \text{sat-precond-as } (\text{state-succ } s a) as)$

— NOTE added lemma.

lemma *sat-precond-as-pair*:

$\text{sat-precond-as } s ((p, e) \# as) = (p \subseteq_f s \wedge \text{sat-precond-as } (\text{state-succ } s (p, e)) as)$

by *simp*

— NOTE 'fun' because of multiple defining equations.

fun *rem-effectless-act* **where**

$\text{rem-effectless-act } [] = []$

| $\text{rem-effectless-act } (a \# as) = (\text{if } \text{fmdom}' (\text{snd } a) \neq \{\} \text{ then } (a \# \text{rem-effectless-act } as)$

$\text{else } \text{rem-effectless-act } as$

$)$

— NOTE 'fun' because of multiple defining equations.

fun *no-effectless-act* **where**

$\text{no-effectless-act } [] = \text{True}$

| $\text{no-effectless-act } (a \# as) = ((\text{fmdom}' (\text{snd } a) \neq \{\}) \wedge \text{no-effectless-act } as)$

lemma *graph-plan-lemma-4*:

fixes $s s' as vs P$

assumes $(\forall a. (\text{ListMem } a as \wedge P a) \longrightarrow ((\text{fmdom}' (\text{snd } a) \cap vs) = \{\}))$

sat-precond-as $s as$

$\text{sat-precond-as } s' (\text{filter } (\lambda a. \neg(P a)) as) (\text{fmrestrict-set } vs s = \text{fmrestrict-set } vs s')$

shows

$(\text{fmrestrict-set } vs (\text{exec-plan } s as)$

$= \text{fmrestrict-set } vs (\text{exec-plan } s' (\text{filter } (\lambda a. \neg(P a)) as)))$

using *assms*

unfolding *exec-plan.simps*

proof(*induction as arbitrary: s s' vs P*)

case (*Cons a as*)

then have $1: \text{fst } a \subseteq_f s \text{ sat-precond-as } (\text{state-succ } s a) as$

by *auto*

then have $2: \forall a'. \text{ListMem } a' as \wedge P a' \longrightarrow \text{fmdom}' (\text{snd } a') \cap vs = \{\}$

by (*simp add: Cons.prem(1) insert*)

then show *?case*

proof (*cases P a*)

case *True*

{

then have $\text{filter } (\lambda a. \neg(P a)) (a \# as) = \text{filter } (\lambda a. \neg(P a)) as$

by *simp*

```

    then have sat-precond-as s' (filter (λa. ¬(P a)) as)
      using Cons.prem3 True
      by argo
  }
note a = this
{
  then have ListMem a (a # as)
    using elem
    by fast
  then have (fmdom' (snd a) ∩ vs) = {}
    using Cons.prem1 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.prem4 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.prem3 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.prem4 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 carried instead of triples.

— NOTE 'fun' because of multiple defining equations.

fun *rem-condless-act* **where**
rem-condless-act s pfx-a [] = pfx-a

| $rem\text{-condless-act } s \text{ pfx-}a \text{ (} a \# as) = (if \text{fst } a \subseteq_f \text{exec-plan } s \text{ pfx-}a$
 then $rem\text{-condless-act } s \text{ (pfx-}a \text{ @ [} a]) as$
 else $rem\text{-condless-act } s \text{ pfx-}a as$
)

lemma *rem-condless-act-pair:*

$rem\text{-condless-act } s \text{ pfx-}a \text{ ((} p, e) \# as) = (if p \subseteq_f \text{exec-plan } s \text{ pfx-}a$
 then $rem\text{-condless-act } s \text{ (pfx-}a \text{ @ [(} p,e]) as$
 else $rem\text{-condless-act } s \text{ pfx-}a as$
)

($rem\text{-condless-act } s \text{ pfx-}a [] = \text{pfx-}a$)
 by *simp+*

lemma *exec-remcondless-cons:*

fixes $s \ h \text{ as pfx}$

shows

$\text{exec-plan } s \text{ (rem-condless-act } s \text{ (} h \# \text{pfx}) as)$
 $= \text{exec-plan (state-succ } s \ h) \text{ (rem-condless-act (state-succ } s \ h) \text{ pfx } as)$

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

lemma *rem-condless-valid-1:*

fixes $as \ s$

shows ($\text{exec-plan } s \ as = \text{exec-plan } s \text{ (rem-condless-act } s \ [] \ as)$)

by (*induction as arbitrary: s*)

(*auto simp add: exec-remcondless-cons FDOM-state-succ state-succ-def*)

lemma *rem-condless-act-cons:*

fixes $h' \ \text{pfx } as \ s$

shows ($rem\text{-condless-act } s \text{ (} h' \# \text{pfx}) as) = (h' \# rem\text{-condless-act (state-succ } s$
 $h') \text{ pfx } as)$

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

lemma *rem-condless-act-cons-prefix:*

fixes $h \ h' \ as \ as' \ s$

assumes $\text{prefix } (h' \# as')$ ($rem\text{-condless-act } s \ [h] \ as$)

shows (

($\text{prefix } as' \text{ (rem-condless-act (state-succ } s \ h) \ [] \ as)$)
 $\wedge h' = h$

)

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 $\text{length } (\text{rem-condless-act } s \ [] \ as) \leq \text{length } as$
by (*induction as arbitrary: s*)
 (*auto simp: rem-condless-act-cons le-SucI*)

lemma *rem-condless-valid-4*:

fixes *as A s*
assumes ($\text{set } as \subseteq A$)
shows ($\text{set } (\text{rem-condless-act } s \ [] \ as) \subseteq A$)
using *assms*
by (*induction as arbitrary: A s*) (*auto simp: rem-condless-act-cons*)

lemma *rem-condless-valid-6*:

fixes *as s P*
shows $\text{length } (\text{filter } P \ (\text{rem-condless-act } s \ [] \ as)) \leq \text{length } (\text{filter } P \ as)$
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 ($\text{list-all } P \ as \wedge \text{list-all } P \ as2$)
shows $\text{list-all } P \ (\text{rem-condless-act } s \ as2 \ as)$
using *assms*
by (*induction as arbitrary: P s as2*) *auto*

lemma *rem-condless-valid-8*:

fixes *s as*
shows $\text{subseq } (\text{rem-condless-act } s \ [] \ as) \ as$
by (*induction as arbitrary: s*) (*auto simp: sublist-cons-4 rem-condless-act-cons*)

lemma *rem-condless-valid-10*:

fixes *PROB as*
assumes $as \in (\text{valid-plans } PROB)$
shows $(\text{rem-condless-act } s \ [] \ as \in \text{valid-plans } PROB)$
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 ⊆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 ⊆f s1
      using Cons.prems(2)
      by simp
    then have fst a ⊆f s2
      using Cons.prems(1) submap-imp-state-succ-submap-a
      by blast
  }
  note 1 = this
  {
    have 2: fst a ⊆f s1 sat-precond-as (state-succ s1 a) as
      using Cons.prems(2)
      by simp+
    then have state-succ s1 a ⊆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 ⊆f s2) (sat-precond-as s1 (a # as))
  shows state-succ s1 a ⊆f state-succ s2 a

```

```

using assms
proof (cases fst a  $\subseteq_f$  s1)
  case true: True
  then show ?thesis
  proof (cases fst a  $\subseteq_f$  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  $\subseteq_f$  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  $\subseteq_f$  s2) (sat-precond-as s1 as)
  shows (exec-plan s1 as  $\subseteq_f$  exec-plan s2 as)
  using assms
proof (induction as arbitrary: s1 s2)
  case (Cons a as)
  have state-succ s1 a  $\subseteq_f$  state-succ s2 a
    using Cons.prem1 submap-init-submap-exec-i
    by blast
  moreover have sat-precond-as (state-succ s1 a) as
    using Cons.prem2(2)
    by simp
  ultimately have exec-plan (state-succ s1 a) as  $\subseteq_f$  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  $\equiv$  (fmdom' (snd a)  $\subseteq$  varset)
for a :: 'a action

```

```

lemma varset-action-pair: (varset-action (p, e) vs) = (fmdom' e  $\subseteq$  vs)
  unfolding varset-action-def
  by auto

```

```

lemma eq-effect-eq-vset:
  fixes x y
  assumes (snd x = snd y)
  shows (( $\lambda a.$  varset-action a vs) x = ( $\lambda a.$  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) ≤ length as*
by (*induction as*) *auto*

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

lemma *rem-effectless-works-4'*:
fixes *A as*
assumes (*as ∈ valid-plans A*)
shows (*rem-effectless-act as ∈ 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)) ≤ 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. fndom' (snd a) ≠ {}) 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* \wedge *no-effectless-act(as2)*)
by (*induction as1*) *auto*

— TODO refactor into ‘*List_Utills.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.premis 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.prem(1) sublist-CONS1-E
      by metis
    then have no-effectless-act as1
      using Cons.prem(2) Cons.IH
      by blast
  }
moreover
  {
    have list-all (λa. fmdom' (snd a) ≠ {}) as2
      using Cons.prem(2) rem-effectless-works-7
      by blast
    moreover have ListMem a as2
      using Cons.prem(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 ($exec-plan\ s\ as = exec-plan\ s\ (rem-effectless-act\ as)$)

($sat-precond-as\ s\ as \longrightarrow sat-precond-as\ s\ (rem-effectless-act\ as)$)

($length\ (rem-effectless-act\ as) \leq length\ as$)

($(set\ as \subseteq A) \longrightarrow (set\ (rem-effectless-act\ as) \subseteq A)$)

($no-effectless-act\ (rem-effectless-act\ as)$)

shows ($\forall P. length\ (filter\ P\ (rem-effectless-act\ as)) \leq length\ (filter\ P\ as)$)

using *assms rem-effectless-works-5*

by *blast*

— NOTE name shortened.

definition *rem-effectless-act-set* **where**

$rem-effectless-act-set\ A \equiv \{a \in A. fmdom'\ (snd\ a) \neq \{\}\}$

lemma *rem-effectless-act-subset-rem-effectless-act-set-thm*:

fixes $as A$

assumes ($set\ as \subseteq A$)

shows ($set\ (rem-effectless-act\ as) \subseteq rem-effectless-act-set\ A$)

unfolding *rem-effectless-act-set-def*

using *assms*

by (*induction as*) *auto*

lemma *rem-effectless-act-set-no-empty-actions-thm*:

fixes A

shows $rem-effectless-act-set\ A \subseteq \{a. fmdom'\ (snd\ a) \neq \{\}\}$

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 ⊆_f exec-plan s []*)

case *True*

```

then have rem-condless-act  $s \ [] (a \# as) = a \# \text{rem-condless-act} (\text{state-succ}$ 
s a)  $\ [] as$ 
using rem-condless-act-cons
by fastforce
moreover
{
have fmdom' (snd a)  $\neq \{\}$  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)  $\wedge$  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) ( $\forall x. \neg(\text{fmdom}' (\text{snd } (f x)) = \{\})$ )
shows (list-all ( $\lambda a. \neg(f a = (\text{fmempty}, \text{fmempty}))$ ) 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) ( $\neg$ varset-action a vs)
  shows  $\neg((\text{fmdom}'(\text{snd } a) \cap ((\text{prob-dom } \text{PROB}) - \text{vs})) = \{\})$ 
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
       $\text{fmdom}'(\text{snd } a) \cap (\text{prob-dom } \text{PROB} - \text{vs})$ 
       $= (\text{fmdom}'(\text{snd } a)) - (\text{fmdom}'(\text{snd } a) \cap \text{vs})$ 
      using Diff-Int-distrib
      by blast
  }
  note 2 = this
  then show ?thesis
    using 1 2
  proof (cases fmdom' (snd a) ∩ vs = {})
  case False
  {
    have  $\neg(\text{fmdom}'(\text{snd } a) \subseteq \text{vs})$ 
      using assms(2) varset-action-def
      by fast
    then have  $(\text{fmdom}'(\text{snd } a) \cap \text{vs} \neq \text{fmdom}'(\text{snd } a))$ 
      by auto
    then have  $(\text{fmdom}'(\text{snd } a) \cap \text{vs}) \subset \text{fmdom}'(\text{snd } a)$ 
      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 $((\text{fmdom}'(\text{snd } a)) \cap (\text{prob-dom } \text{PROB} - \text{vs})) = \{\}$
 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 $((\text{fmdom}'(\text{snd } a)) \cap \text{vs}) = \{\}$
 using *assms*
 unfolding *varset-action-def*
 by *blast*

lemma *vset-nempty-eff-not-disj-eff-vs*:
 fixes *PROB a vs*
 assumes (*varset-action a vs*) $(\text{fmdom}'(\text{snd } a) \neq \{\})$
 shows $\neg((\text{fmdom}'(\text{snd } a) \cap \text{vs}) = \{\})$
 using *assms*
 unfolding *varset-action-def*
 by *auto*

lemma *vset-disj-eff-diff*:
 fixes *s a vs*
 assumes (*varset-action a vs*)
 shows $((\text{fmdom}'(\text{snd } a) \cap (s - \text{vs})) = \{\})$
proof –
 have 1: $\text{fmdom}'(\text{snd } a) \subseteq \text{vs}$
 using *assms*
 by (*simp add: varset-action-def*)
moreover {
 have $\text{fmdom}'(\text{snd } a) \cap (s - \text{vs}) = (\text{fmdom}'(\text{snd } a) \cap s) - (\text{fmdom}'(\text{snd } a)$
 $\cap \text{vs})$
 using *Diff-Int-distrib*
 by *fast*
 also have $\dots = (\text{fmdom}'(\text{snd } a) \cap s) - (\text{fmdom}'(\text{snd } a))$
 using 1
 by *auto*
 finally have $\text{fmdom}'(\text{snd } a) \cap (s - \text{vs}) = \{\}$


```

    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  $\longleftrightarrow$  ( $\forall e. ListMem e l \longrightarrow P e$ )
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.prem1 insert.hyps
        by fastforce+
      then show ?thesis
        using insert.IH
        by blast
      qed simp
    }
  }
  moreover
  {
    assume P2: ( $\forall e. ListMem e l \longrightarrow P e$ )
    then have list-all P l
      proof (induction l arbitrary: P)
      case (Cons a l)
      {
        have  $\forall e. ListMem e l \longrightarrow P e$ 
          using Cons.prem1 insert
          by fast
        then have list-all P l
          using Cons.IH
          by blast
      }
    moreover have P a
      using Cons.prem2
  }

```

```

      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 ( $\lambda a. \text{varset-action } a ((\text{prob-dom } \text{PROB}) - \text{vs})) \text{ as}$ )
  shows (fmrestrict-set vs s = fmrestrict-set vs (exec-plan s as))
proof -
  have 1:  $\forall e. \text{ListMem } e \text{ as} \longrightarrow \text{varset-action } e ((\text{prob-dom } \text{PROB}) - \text{vs})$ 
    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 ( $\lambda a. \text{disjnt } (fmdom' (snd a)) \text{ vs}$ ) as
    using list-all-list-mem
    by blast
  then have list-all ( $\lambda a. \text{disjnt } (fmdom' (snd a)) \text{ vs}$ ) (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  $as = \text{rem-condless-act } s \ [] \ as$  and  $vs = vs$  and
     $P = \lambda a. \text{varset-action } a \ (\text{prob-dom } PROB - vs)$ 
  ] 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 ( $\lambda a. \text{varset-action } a \ (\text{prob-dom } PROB - vs)$ ) (rem-condless-act
   $s \ [] \ as$ )
    using assms(1) rem-condless-valid-7 list.pred-inject(1)
    by blast
  then have filter ( $\lambda a. \neg(\text{varset-action } a \ (\text{prob-dom } PROB - vs))$ ) (rem-condless-act
   $s \ [] \ as$ ) = []
    using filter-empty-every-not
    by fastforce
  then have
    sat-precond-as  $s$  (filter ( $\lambda a. \neg(\text{varset-action } a \ (\text{prob-dom } PROB - vs))$ )
    (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 ( $\lambda a. \neg(\text{fmdom}' (\text{snd } a) = \{\})$ ))  $as = as$ 
  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  $a \ (vs'' - (vs' \cup vs))$ )
  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  $a \ (s - (vs \cup vs'))$ )
  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  $\in$  valid-plans PROB)
  shows (filter ( $\lambda a.$  varset-action a (prob-dom PROB)) as = as)
  using assms
proof (induction as)
  case (Cons a as)
  {
    have as  $\in$  valid-plans PROB
    using Cons.prems valid-plan-valid-tail
    by fast
    then have (filter ( $\lambda a.$  varset-action a (prob-dom PROB)) as = as)
    using Cons.IH
    by blast
  }
  moreover {
    have a  $\in$  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  $\subseteq$  vs') (varset-action a vs)
  shows (varset-action a vs')
  using assms
  unfolding varset-action-def
  by simp

```

```

lemma sat-precond-as-px:
  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  $\subseteq_f$  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  $\equiv (\forall P. (\forall x\ y. R\ x\ y \longrightarrow P\ x\ y) \wedge (\forall x\ y\ z. P\ x\ y \wedge P\ y\ z \longrightarrow P\ x\ z) \longrightarrow P\ a\ b)$ 

```

— NOTE adapts transitive closure definitions of Isabelle and HOL4.

```

lemma TC-equiv-tranclp: TC R a b  $\longleftrightarrow (R^{++}\ a\ b)$ 

```

```

proof –

```

```

{
  have TC R a b  $\implies (R^{++}\ a\ b)$ 
    unfolding TC-def
    using tranclp.r-into-trancl tranclp-trans
    by metis
}
moreover
{
  have  $(R^{++}\ a\ b) \implies TC\ R\ a\ b$  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:  $\neg(\lambda x y. R x y \wedge P x y)^{++} x y$ 
    using TC-IMP-NOT-TC-CONJ-1[where  $P = \lambda x y. P x y$ ]
    by blast
  {
    {
      from 1 have  $\neg TC (\lambda x y. R x y \wedge P x y) x y$ 
        using TC-equiv-tranclp
        by fast
      then have  $\exists Pa.$ 
         $(\forall x y. R x y \wedge P x y \longrightarrow Pa x y) \wedge (\forall x y z. Pa x y \wedge Pa y z \longrightarrow Pa x z)$ 
         $\longrightarrow \neg Pa x y$ 
        unfolding TC-def
        by blast
    }
    then obtain  $Pa$  where  $a:$ 
       $(\forall x y. R x y \wedge P x y \longrightarrow Pa x y) \wedge (\forall x y z. Pa x y \wedge Pa y z \longrightarrow Pa x z)$ 
       $\longrightarrow \neg Pa x y$ 
      by blast
    then have  $\neg(\forall Pa. (\forall x y. R' x y \wedge P x y \longrightarrow Pa x y) \wedge (\forall x y z. Pa x y \wedge Pa$ 
       $y z \longrightarrow Pa x z) \longrightarrow Pa x y)$ 
      by (metis assms(1) assms(2) tranclp.r-into-trancl tranclp-trans)
    then have  $\neg TC (\lambda x y. R' x y \wedge P x y) x y$ 
      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  $R :: 'a \Rightarrow 'a \Rightarrow bool$  and  $P$ 
assumes  $(\forall x y. R x y \longrightarrow P x y) (\forall x y z. P x y \wedge P y z \longrightarrow P x z)$ 
shows  $\forall u v. (TC R) u v \longrightarrow P u v$ 
using assms
unfolding TC-def
by metis

```

lemma *REFL-IMP-3-CONJ-1*:

```

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

```

lemma *REFL-TC-CONJ*:

```

fixes  $R R' :: 'a \Rightarrow 'a \Rightarrow \text{bool}$  and  $P x y$ 
assumes reflexive  $R' \forall x y. P x \wedge P y \longrightarrow (R' x y \longrightarrow R x y) \neg(R^{++} x y)$ 
shows  $(\neg(R'^{++} x y) \vee (\exists z. \neg P z \wedge (R')^{++} x z \wedge (R')^{++} z y))$ 
using assms
proof (cases  $\neg R'^{++} x y$ )
next
  case False
then show ?thesis using assms
   $TC-IMP-NOT-TC-CONJ$ [where  $P=\lambda x y. P x \wedge P y]$ 
   $REFL-IMP-3-CONJ$ [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  $R x z \vee (\exists y :: 'a. \neg(x = y) \wedge \neg(y = z) \wedge R x y \wedge R^{++} y z)$ 
proof –
{
  fix  $u v$ 
have  $\forall x y. R x y \longrightarrow R x y \vee (\exists ya. x \neq ya \wedge ya \neq y \wedge R x ya \wedge R^{++} ya y)$ 
by meson
moreover have  $\forall x y z.$ 
   $(R x y \vee (\exists ya. x \neq ya \wedge ya \neq y \wedge R x ya \wedge R^{++} ya y))$ 
   $\wedge (R y z \vee (\exists ya. y \neq ya \wedge ya \neq z \wedge R y ya \wedge R^{++} ya z))$ 
   $\longrightarrow R x z \vee (\exists y. x \neq y \wedge y \neq z \wedge R x y \wedge R^{++} y z)$ 

```

```

    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  $TC\text{-}INDUCT[\text{where } P = \lambda x z. R\ x\ z \vee (\exists y :: 'a. \neg(x = y) \wedge \neg(y = z)$ 
 $\wedge R\ x\ y \wedge R^{++}\ y\ z)]$ 
  by blast
}
then show ?thesis
  using assms  $TC\text{-}equiv\text{-}tranclp$ 
  by (simp add:  $TC\text{-}equiv\text{-}tranclp$ )
qed
end
theory Dependency
  imports Main HOL-Library.Finite-Map FactoredSystem ActionSeqProcess Re-
LUtils
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*

$$\begin{aligned}
 \text{dep } PROB\ v1\ v2 &\equiv (\exists a. \\
 &a \in PROB \\
 &\wedge (\\
 &\quad ((v1 \in fmdom'\ (fst\ a)) \wedge (v2 \in fmdom'\ (snd\ a))) \\
 &\quad \vee ((v1 \in fmdom'\ (snd\ a)) \wedge v2 \in fmdom'\ (snd\ a))) \\
 &) \\
 &) \\
 &\vee (v1 = v2)
 \end{aligned}$$

— NOTE name shortened to 'dep_var_set'.

definition *dep-var-set where*

$dep\text{-}var\text{-}set\ PROB\ vs1\ vs2 \equiv (disjnt\ vs1\ vs2) \wedge$
 $(\exists\ v1\ v2. (v1 \in vs1) \wedge (v2 \in vs2) \wedge (dep\ PROB\ v1\ v2))$
 $)$

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

lemma *DEP-REFL*:
fixes *PROB*
shows *reflexive* $(\lambda v\ v'. dep\ PROB\ v\ v')$
unfolding *dep-def reflexive-def*
by *presburger*

— NOTE added lemma.

lemma *NEQ-DEP-IMP-IN-DOM-i*:
fixes *a v*
assumes $a \in PROB\ v \in fmdom'\ (fst\ a)$
shows $v \in prob\text{-}dom\ PROB$
proof –
have $v \in fmdom'\ (fst\ a)$
using *assms(2)*
by *simp*
moreover have $fmdom'\ (fst\ a) \subseteq prob\text{-}dom\ PROB$
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  $fmdom'\ (snd\ a) \subseteq prob-dom\ PROB$ 
    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  $PROB :: (('a, 'b)\ fmap \times ('a, 'b)\ fmap)\ set\ \mathbf{and}\ v\ v'$ 
assumes  $\neg(v = v')\ (dep\ PROB\ v\ v')$ 
shows  $(v \in (prob-dom\ PROB) \wedge v' \in (prob-dom\ PROB))$ 
using  $assms$ 
unfolding  $dep-def$ 
using  $FDOM-pre-subset-prob-dom-pair\ FDOM-eff-subset-prob-dom-pair$ 
proof –
  obtain  $a$  where  $1$ :
     $a \in PROB$ 
     $(v \in fmdom'\ (fst\ a) \wedge v' \in fmdom'\ (snd\ a) \vee v \in fmdom'\ (snd\ a) \wedge v' \in fmdom'$ 
     $(snd\ a))$ 
    using  $assms$ 
    unfolding  $dep-def$ 
    by  $blast$ 
  then consider
     $(i)\ v \in fmdom'\ (fst\ a) \wedge v' \in fmdom'\ (snd\ a)$ 
     $| (ii)\ v \in fmdom'\ (snd\ a) \wedge v' \in fmdom'\ (snd\ a)$ 
    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  $v \in \text{prob-dom } PROB$   $v' \in \text{prob-dom } PROB$ 
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 ( $\text{dep-var-set } PROB (\bigcup S) vs$ )
shows ( $\exists vs'. vs' \in S \wedge \text{dep-var-set } PROB vs' vs$ )
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 ( $\text{dep-var-set } PROB vs (vs' \cup vs'')$ )
shows ( $\text{dep-var-set } PROB vs vs' \vee \text{dep-var-set } PROB vs vs''$ )
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**
 $dep\text{-}tc\ PROB = TC (\lambda v1' v2'. dep\ PROB\ v1'\ v2')$

— 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 ($v1 \in prob\text{-}dom\ PROB$)
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** $R = (\lambda v1' v2'. dep\ PROB\ v1'\ v2')$ **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) ( $\exists a.$ 
   $a \in PROB \wedge$ 
  (
     $v1 \in fmdom' (fst a) \wedge v2 \in fmdom' (snd a)$ 
     $\vee v1 \in fmdom' (snd a) \wedge v2 \in fmdom' (snd a)$ 
  )
  | (III)  $v1 = v2$ 
using  $i$ 
unfolding  $dep-def$ 
by  $blast$ 
then have  $?thesis$ 
proof ( $cases$ )
  case  $II$ 
  then obtain  $a$  where  $1$ :
     $a \in PROB (v1 \in fmdom' (fst a) \wedge v2 \in fmdom' (snd a)$ 
     $\vee v1 \in fmdom' (snd a) \wedge v2 \in fmdom' (snd a))$ 
    by  $blast$ 
  then have  $v1 \in fmdom' (fst a) \cup fmdom' (snd a)$ 
    by  $blast$ 
  then have  $2$ :  $v1 \in action-dom (fst a) (snd a)$ 
    unfolding  $action-dom-def$ 
    by  $blast$ 
  then have  $action-dom (fst a) (snd a) \subseteq prob-dom PROB$ 
    using  $1(1) exec-as-proj-valid-2$ 
    by  $fast$ 
  then have  $v1 \in prob-dom PROB$ 
    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  $v1 \neq y \vee y \neq v2 \text{ dep } PROB \vee v1 \neq y \vee y \neq v2 \text{ TC } (dep PROB) \vee v2$ 
  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 $((vs-1 \cap vs-2) = \{\}) (vs-3 \subseteq vs-2) \neg(dep\text{-}var\text{-}set\ PROB\ vs-1\ vs-2)$
shows $\neg(dep\text{-}var\text{-}set\ PROB\ vs-1\ vs-3)$
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  $\forall v1 v2. \neg(v1 \in vs') \vee \neg(v2 \in vs) \vee \neg disjnt vs' vs \vee \neg dep PROB$ 
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:  $\bigwedge aa ab. aa \notin vs \vee ab \notin vs' \vee aa \notin fmdom' (snd a) \vee ab \notin fmdom'$ 
(snd a)
    using <a ∈ PROB> <disjnt vs vs'> disjnt-sym by blast
  obtain aa :: 'a set ⇒ 'a set ⇒ 'a where
    f3:  $\bigwedge A Aa a Ab Ac. (A \subseteq Aa \vee aa A Aa \in A) \wedge (aa A Aa \notin Aa \vee A \subseteq Aa)$ 
  ∧ ((a::'a) ∉ Ab ∨ ¬ Ab ⊆ Ac ∨ a ∈ Ac)
  by (atomize-elim, (subst choice-iff[symmetric])+, blast)
  then have  $\bigwedge A. fmdom' (snd a) \subseteq A \vee aa (fmdom' (snd a)) A \in vs \vee aa (fmdom'$ 
(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 ⇒ 'b) ⇒ 'a set where
  fdom f ≡ {x. ∃ 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 ⇒ bool) ⇒ bool where
  invariant fm ≡ (∀ x. (x ∈ fdom fm ∧ fm x) → False) ∧ (∃ x. x ∈ fdom fm ∧ 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 ∩ B = {}]]
⇒ card (A ∪ B) = card A + card B) ?
lemma card-union': (finite s) ∧ (finite t) ∧ (disjnt s t) ⇒ (card (s ∪ t) = card
s + card t)
  by (simp add: card-Un-disjoint disjnt-def)

lemma CARD-INJ-IMAGE-2:
  fixes f s
  assumes finite s (∀ x y. ((x ∈ s) ∧ (y ∈ s)) → ((f x = f y) ↔ (x = y)))
  shows (card (f ` s) = card s)
proof -
  {
    fix x y
    assume x ∈ s y ∈ s
    then have f x = f y → 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*: $\bigwedge s t x. (x \in s) \wedge \neg(x \in t) \implies \neg(s = t)$
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 $(\forall x. x \in s \longrightarrow x \leq k) \implies \text{finite } s$
proof –
assume *P*: $(\forall x. x \in s \longrightarrow x \leq k)$
let *?f* = *id* :: *nat* \Rightarrow *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 $\bigwedge (s :: \text{nat set})\ k. (\forall x. x \in s \longrightarrow x \leq k) \implies \text{finite } s$
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 $(\text{Inf } s \leq k)$
proof –
from *assms* **obtain** *x* **where** $1: x \in s\ x \leq k$
by *blast*
{
assume *C*: $\text{Inf } s > k$
then **have** $\text{Inf } s > x$ **using** $1(2)$
by *fastforce*
then **have** *False*
using $1(1)$ *cInf-lower leD*
by *fast*
}
then **show** *?thesis*
by *fastforce*

qed

— NOTE $\text{nat} \rightarrow \text{bool}$ 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 $\text{Min } \{n. s n\} \in \{n. s n\}$ is not necessarily true.

lemma *mem-lt-imp-MIN-lt*:

fixes $s :: \text{nat set}$ **and** $k :: \text{nat}$
assumes $(\exists x. x \in s \wedge x < k)$
shows $(\text{Inf } s) < k$

proof —

obtain x **where** $1: x \in s$ $x < k$
using *assms*
by *blast*
then have $2: s \neq \{\}$
by *blast*
then have $\text{Inf } s \in s$
using *Inf-nat-def LeastI*
by *force*
moreover have $\forall x \in s. \text{Inf } s \leq x$
by (*simp add: cInf-lower*)
ultimately show $(\text{Inf } s) < k$
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** $k :: \text{nat}$
assumes $(\forall x. x \in s \rightarrow x < k)$
shows *finite s*
using *assms bounded-nat-set-is-finite*
by *blast*

lemma *bound-main-lemma-2*: $\bigwedge (s :: \text{nat set}) k. (s \neq \{\}) \wedge (\forall x. x \in s \rightarrow x \leq k) \implies \text{Sup } s \leq k$

proof —

fix $s :: \text{nat set}$ **and** k
{
assume $P1: s \neq \{\}$
assume $P2: (\forall x. x \in s \rightarrow x \leq k)$
have *finite s* **using** $P2$ *mems-le-finite* **by** *auto*
moreover have $\text{Max } s \in s$ **using** $P1$ *calculation Max-in* **by** *auto*
moreover have $\text{Max } s \leq k$ **using** $P2$ *calculation* **by** *auto*
}
then show $(s \neq \{\}) \wedge (\forall x. x \in s \rightarrow x \leq k) \implies \text{Sup } s \leq k$
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*: $\wedge s (k :: nat)$.

($s \neq \{\}$)
 $\implies (\forall x. x \in s \longrightarrow x < k)$
 $\implies \text{Sup } s < k$

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

$\{p. \text{distinct } p \wedge \text{set } p \subseteq P\}$
 $= \{\{\}\} \cup (\bigcup ((\lambda e. \{e \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{e\})\}) ' P))$

proof -

let $?A = \{p. \text{distinct } p \wedge \text{set } p \subseteq P\}$

let $?B = \{\{\}\} \cup (\bigcup ((\lambda e. \{e \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{e\})\}) ' P))$

{

{

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** $\text{set } a \subseteq P$

by *simp*

then have $\text{set } list \subseteq (P - \{h\})$

using P *dual-order.trans local.Cons*

by *auto*

}

moreover from P *Cons*

have *distinct list*

by *force*

ultimately have $a \in ((\lambda e. \{e \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{e\})\}) ?b')$

using *Cons*

by *blast*

moreover {

from P *Cons* **have** $?b' \in \text{set } a$

by *simp*

moreover from P **have** $\text{set } a \subseteq P$

by *simp*

ultimately have $?b' \in P$

```

    by auto
  }
  ultimately have
     $\exists b' \in P. a \in ((\lambda e. \{e \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{e\})\}) b')$ 
    by meson
  }
  then obtain  $b'$  where
     $b' \in P a \in ((\lambda e. \{e \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{e\})\}) b')$ 
    by blast
  then show ?thesis
    by blast
qed blast
}
then have  $?A \subseteq ?B$ 
  by auto
}
moreover {
  {
    fix  $b$ 
    assume  $P: b \in ?B$ 
    have  $b \in ?A$ 

```

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  $a:$ 
   $b' \in P b \in \{b' \# p0 \mid p0. \text{distinct } p0 \wedge \text{set } p0 \subseteq (P - \{b'\})\}$ 
  by fast
then obtain  $p0$  where  $b: b = b' \# p0 \text{ distinct } p0 \text{ set } p0 \subseteq (P - \{b'\})$ 
  by blast
then have  $\text{distinct } (b' \# p0)$ 
  by (simp add: subset-Diff-insert)
moreover have  $\text{set } (b' \# p0) \subseteq P$ 
  using  $a(1) b(3)$ 
  by auto
ultimately show ?thesis
  using  $b(1)$ 
  by fast
qed simp
}
then have  $?B \subseteq ?A$ 
  by blast
}
ultimately show ?thesis
  using set-eq-subset
  by blast
qed

```

```

lemma FINITE-ALL-DISTINCT-LISTS:
  fixes  $P$ 
  assumes  $finite\ P$ 
  shows  $finite\ \{p.\ distinct\ p \wedge set\ p \subseteq P\}$ 
  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  $P: e \in P$ 
    have
       $\{e \# p0 \mid p0.\ distinct\ p0 \wedge set\ p0 \subseteq P - \{e\}\}$ 
       $= (\lambda p.\ e \# p)\ \{p.\ distinct\ p \wedge set\ p \subseteq P - \{e\}\}$ 
      by  $blast$ 
    moreover {
      let  $?P'=P - \{e\}$ 
      from  $Suc.prem$ s
      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  $finite\ ((\lambda p.\ e \# p)\ \{p.\ distinct\ p \wedge set\ p \subseteq ?P'\})$ 
  proof ( $cases\ e \in P$ )
    case  $True$ 
    then have  $x = card\ ?P'$  using  $Suc.prem$ s  $Suc(2)$ 
      by  $fastforce$ 
    moreover from  $Suc.prem$ s
    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 ∧ set p ⊆ ?P'}
      using False P by linarith
    then show ?thesis
      using finite-imageI
      by blast
  qed
}
ultimately have finite {e # p0 | p0. distinct p0 ∧ set p0 ⊆ (P - {e})}
  by argo
}
then have finite (⋃ ((λe. {e # p0 | p0. distinct p0 ∧ set p0 ⊆ (P - {e})})
‘ P))
  using Suc.premis
  by blast
}
then have
  finite ({[]} ∪ (⋃ ((λe. {e # p0 | p0. distinct p0 ∧ set p0 ⊆ (P - {e})}) ‘
P)))
  using finite-Un
  by blast
}
then show ?case
  using FINITE-ALL-DISTINCT-LISTS-i[OF Suc.premis]
  by force
qed

```

```

lemma subset-inter-diff-empty:
  assumes s ⊆ t
  shows (s ∩ (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

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

definition *MPLS-charles* where

$$\text{MPLS-charles } PROB \equiv \{\text{Inf } (PLS-charles (fst p) (snd p) PROB) \mid p. \\ ((fst p) \in \text{valid-states } PROB)\}$$

$\wedge ((snd\ p) \in\ valid-plans\ PROB)$
 $\}$

— NOTE name shortened to 'problem_plan_bound_charles'.

definition *problem-plan-bound-charles* **where**
problem-plan-bound-charles $PROB \equiv Sup\ (MPLS-charles\ PROB)$

— NOTE name shortened to 'PLS_state'.

definition *PLS-state-1* **where**
PLS-state-1 $s\ as \equiv length\ '\{as'\}.\ (exec-plan\ s\ as' = exec-plan\ s\ as)$

— NOTE name shortened to 'MPLS_stage_1'.

definition *MPLS-stage-1* **where**
MPLS-stage-1 $PROB \equiv$
 $(\lambda\ (s,\ as).\ Inf\ (PLS-state-1\ s\ as))$
 $'\{(s,\ as).\ (s \in\ valid-states\ PROB) \wedge (as \in\ valid-plans\ PROB)\}$

— NOTE name shortened to 'problem_plan_bound_stage_1'.

definition *problem-plan-bound-stage-1* **where**
problem-plan-bound-stage-1 $PROB \equiv Sup\ (MPLS-stage-1\ PROB)$
for $PROB :: 'a\ problem$

— NOTE name shortened.

definition *PLS* **where**
PLS $s\ as \equiv length\ '\{as'\}.\ (exec-plan\ s\ as' = exec-plan\ s\ as) \wedge (subseq\ as'\ as)$

— 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 $?S = \{as'.\ (exec-plan\ s\ as' = exec-plan\ s\ as) \wedge (subseq\ as'\ as)\}$

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** $length\ '\?S \subseteq (?S1 \cap ?S2)$

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 \equiv
 $(\lambda (s, as). \text{Inf } (PLS\ s\ as))$
 $\text{' } \{(s, as). (s \in \text{valid-states } PROB) \wedge (as \in \text{valid-plans } PROB)\}$

— NOTE name shortened.

definition *problem-plan-bound* **where**

problem-plan-bound PROB $\equiv \text{Sup } (MPLS\ PROB)$

lemma *expanded-problem-plan-bound-thm-1*:

fixes *PROB*

shows

$(\text{problem-plan-bound } PROB) = \text{Sup } ($
 $(\lambda(s,as). \text{Inf } (PLS\ s\ as)) \text{' } \{(s, as). (s \in (\text{valid-states } PROB)) \wedge (as \in \text{valid-plans } PROB)\}$
 $)$

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

by *blast*

lemma *expanded-problem-plan-bound-thm*:

fixes *PROB* :: $(('a, 'b) \text{fmap} \times ('a, 'b) \text{fmap}) \text{ set}$

shows

$\text{problem-plan-bound } PROB = \text{Sup } (\{\text{Inf } (PLS\ s\ as) \mid s \text{ as.}$
 $(s \in \text{valid-states } PROB)$
 $\wedge (as \in \text{valid-plans } PROB)$
 $\})$

proof —

{
have (
 $\{\text{Inf } (PLS\ s\ as) \mid s \text{ as. } (s \in \text{valid-states } PROB) \wedge (as \in \text{valid-plans } PROB)\}$
 $) = ((\lambda(s, as). \text{Inf } (PLS\ s\ as)) \text{' } \{(s, as).$
 $(s \in \text{valid-states } PROB)$
 $\wedge (as \in \text{valid-plans } PROB)$
 $\})$
)

by *fast*

also have ... =

```

    ( $\lambda(s, as). \text{Inf } (PLS\ s\ as)$ ) ‘
    ( $\{s. \text{fmdom}'\ s = \text{prob-dom } PROB\} \times \{as. \text{set } as \subseteq PROB\}$ )

    unfolding valid-states-def valid-plans-def
    by simp
    finally have
       $Sup (\{\text{Inf } (PLS\ s\ as) \mid s\ as. (s \in \text{valid-states } PROB) \wedge (as \in \text{valid-plans } PROB)\})$ 
    =  $Sup ($ 
      ( $\lambda(s, as). \text{Inf } (PLS\ s\ as)$ ) ‘
      ( $\{s. \text{fmdom}'\ s = \text{prob-dom } PROB\} \times \{as. \text{set } as \subseteq PROB\}$ )
    )

    by argo
  }
moreover have
  problem-plan-bound PROB
  =
   $Sup ((\lambda(s, as). \text{Inf } (PLS\ s\ as)) ‘$ 
  ( $\{s. \text{fmdom}'\ s = \text{prob-dom } PROB\} \times \{as. \text{set } as \subseteq PROB\}$ ))

  unfolding problem-plan-bound-def MPLS-def valid-states-def valid-plans-def
  by fastforce
  ultimately show
    problem-plan-bound PROB
  =  $Sup (\{\text{Inf } (PLS\ s\ as) \mid s\ as.$ 
    ( $s \in \text{valid-states } PROB$ )
     $\wedge (as \in \text{valid-plans } PROB)$ 
  })

  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  $Pi [s] = (s \in \text{valid-states } Pi)$ 
| valid-path  $Pi (s1 \# s2 \# rest) = ($ 
  ( $s1 \in \text{valid-states } Pi$ )
   $\wedge (\exists a. (a \in Pi) \wedge (\text{exec-plan } s1 [a] = s2))$ 
   $\wedge (\text{valid-path } Pi (s2 \# rest))$ 
  )

```

lemma *valid-path-ITP2015:*

```

(valid-path Pi []  $\longleftrightarrow$  True)
 $\wedge$  (valid-path Pi [s]  $\longleftrightarrow$  (s  $\in$  valid-states Pi))
 $\wedge$  (valid-path Pi (s1 # s2 # rest)  $\longleftrightarrow$ 
  (s1  $\in$  valid-states Pi)
   $\wedge$  ( $\exists$  a.
    (a  $\in$  Pi)
     $\wedge$  (exec-plan s1 [a] = s2)
  )
   $\wedge$  (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  $\equiv$  (Sup {length p - 1 | p. valid-path Pi p  $\wedge$  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  $\in$  valid-states PROB) (as  $\in$  valid-plans PROB)
shows ( $\exists$  x.
  (x  $\in$  PLS s as)
   $\wedge$  (x  $\leq$  (2  $\wedge$  card (prob-dom PROB)) - 1)
)

```

proof —

obtain as' **where** 1:

```

  exec-plan s as = exec-plan s as' subseq as' as length as'  $\leq$  2  $\wedge$  card (prob-dom
PROB) - 1

```

using assms main-lemma

by blast

let ?x=length as'

have ?x \in PLS s as

unfolding PLS-def

using 1

by simp

moreover have ?x \leq 2 \wedge card (prob-dom PROB) - 1

using 1(3)

by blast

ultimately show (\exists x.

(x \in PLS s as)

\wedge (x \leq (2 \wedge 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* ($x \in \text{MPLS } \text{PROB}$)

shows ($x \leq 2 \wedge \text{card } (\text{prob-dom } \text{PROB}) - 1$)

proof –

let *?mpls* = *MPLS PROB*

– NOTE obtain *p* = (*s*, *as*) where 'x = Inf (PLS *s as*)' from premise.

have *?mpls* =

($\lambda (s, as). \text{Inf } (\text{PLS } s \text{ as})$) ‘

{(*s*, *as*). ($s \in \text{valid-states } \text{PROB}$) \wedge ($as \in \text{valid-plans } \text{PROB}$)}

using *MPLS-def*

by *blast*

then obtain *s* :: ('a, bool) fmap **and** *as* :: (('a, bool) fmap \times ('a, bool) fmap)

list

where *obtain-s-as*: $x \in$

($\lambda (s, as). \text{Inf } (\text{PLS } s \text{ as})$) ‘

{(*s*, *as*). ($s \in \text{valid-states } \text{PROB}$) \wedge ($as \in \text{valid-plans } \text{PROB}$)}

using *assms(2)*

by *blast*

then have

$x \in \{\text{Inf } (\text{PLS } (\text{fst } p) (\text{snd } p)) \mid p. (\text{fst } p \in \text{valid-states } \text{PROB}) \wedge (\text{snd } p \in \text{valid-plans } \text{PROB})\}$

using *assms(1) obtain-s-as*

by *auto*

then have

$\exists p. x = \text{Inf } (\text{PLS } (\text{fst } p) (\text{snd } p)) \wedge (\text{fst } p \in \text{valid-states } \text{PROB}) \wedge (\text{snd } p \in \text{valid-plans } \text{PROB})$

by *blast*

then obtain *p* :: ('a, bool) fmap \times (('a, bool) fmap \times ('a, bool) fmap) *list* **where** *obtain-p*:

$x = \text{Inf } (\text{PLS } (\text{fst } p) (\text{snd } p)) (\text{fst } p \in \text{valid-states } \text{PROB}) (\text{snd } p \in \text{valid-plans } \text{PROB})$

by *blast*

then have $\text{fst } p \in \text{valid-states } \text{PROB}$ $\text{snd } p \in \text{valid-plans } \text{PROB}$

using *obtain-p*

by *blast+*

then obtain *x'* :: nat **where** *obtain-x'*:

$x' \in \text{PLS } (\text{fst } p) (\text{snd } p) \wedge x' \leq 2 \wedge \text{card } (\text{prob-dom } \text{PROB}) - 1$

using *assms(1) in-PLS-leq-2-pow-n* [**where** $s = \text{fst } p$ **and** $as = \text{snd } p$]

by *blast*

then have $1: x' \leq 2 \wedge \text{card } (\text{prob-dom } \text{PROB}) - 1$ $x' \in \text{PLS } (\text{fst } p) (\text{snd } p)$

$x = \text{Inf } (\text{PLS } (\text{fst } p) (\text{snd } p))$ *finite* ($\text{PLS } (\text{fst } p) (\text{snd } p)$)

using *obtain-x' obtain-p finite-PLS*

by *blast+*

moreover have $x \leq x'$
using $1(2, 4)$ *obtain-p(1) cInf-le-finite*
by *blast*
ultimately show $(x \leq 2 \wedge \text{card}(\text{prob-dom } PROB) - 1)$
by *linarith*
qed

lemma *FINITE-MPLS*:
assumes *finite (Pi :: 'a problem)*
shows *finite (MPLS Pi)*
proof –
have $\forall x \in \text{MPLS } Pi. x \leq 2 \wedge \text{card}(\text{prob-dom } Pi) - 1$
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 $\text{length}(\text{statelist}' s as) = (\text{length } as + 1)$
by *(induction as arbitrary: s) auto*

lemma *valid-path-statelist'*:
fixes *as and s :: ('a, 'b) fmap*
assumes $(as \in \text{valid-plans } Pi) (s \in \text{valid-states } Pi)$
shows $(\text{valid-path } Pi (\text{statelist}' s as))$
using *assms*
proof *(induction as arbitrary: s Pi)*
case *cons: (Cons a as)*
then have $1: a \in Pi \text{ as} \in \text{valid-plans } Pi$
using *valid-plan-valid-head valid-plan-valid-tail*
by *metis+*
then show *?case*
proof *(cases as)*
case *Nil*
{
have $\text{state-succ } s a \in \text{valid-states } Pi$
using $1 \text{ cons.prems}(2) \text{ valid-action-valid-succ}$

```

    by blast
  then have valid-path Pi [state-succ s a]
    using 1 cons.prem(2) cons.IH
    by force
  moreover have ( $\exists aa. aa \in Pi \wedge exec-plan s [aa] = state-succ s a$ )
    using 1(1)
    by fastforce
  ultimately have valid-path Pi (statelist' s [a])
    using cons.prem(2)
    by simp
}
then show ?thesis
  using Nil
  by blast
next
case (Cons b list)
{
  have  $s \in valid-states Pi$ 
    using cons.prem(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.prem(2) Cons lemma-1-i
      by fastforce
  moreover have
    ( $\exists aa b. (aa, b) \in Pi \wedge state-succ s (aa, b) = state-succ s a$ )
      using 1(1) surjective-pairing
      by metis
  ultimately have valid-path Pi (statelist' s (a # b # list))
    using cons.prem(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-stalist' One-nat-def add-Suc-right list.size(3) nat.simps(3))
 (metis (no-types) LENGTH-stalist' One-nat-def add-Suc-right list.size(3)
 nat.simps(3))

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

— NOTE added lemma.

lemma *stalist'-TAKE-i*:
 assumes *Suc m ≤ length (a # as)*
 shows *m ≤ length as*
 using *assms*
 by (induction as arbitrary: a m) auto

lemma *stalist'-TAKE*:
 fixes *as s p*
 assumes (*stalist' s as = p*)
 shows ($\forall n. n \leq \text{length } as \longrightarrow (\text{exec-plan } s (\text{take } n \text{ as})) = (p ! n)$)
 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.prem*
 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.prem*
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.prem
  by argo
qed fastforce
}
then show ?case by blast
qed

```

```

lemma MPLS-empty:
  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  $MPLS\ PROB \neq \{\}$ 
  using assms MPLS-nempty
  by auto
moreover have  $(\forall x. x \in MPLS\ PROB \longrightarrow x \leq 2^{\wedge} card\ (prob-dom\ PROB) -$ 
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  $(\forall (PROB :: 'a\ problem)\ as\ (s :: 'a\ state).$ 
   $(P\ PROB)$ 
   $\wedge (as \in valid-plans\ PROB)$ 
   $\wedge (s \in valid-states\ PROB)$ 
   $\longrightarrow (\exists as'.$ 
   $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
   $\wedge (subseq\ as'\ as)$ 
   $\wedge (length\ as' < f\ PROB)$ 
   $)$ 
 $)$ 
shows  $(\forall PROB\ s\ as.$ 
   $(P\ PROB)$ 
   $\wedge (as \in valid-plans\ PROB)$ 
   $\wedge (s \in (valid-states\ PROB))$ 
   $\longrightarrow (\exists x.$ 
   $(x \in PLS\ s\ as)$ 
   $\wedge (x < f\ PROB)$ 
   $)$ 
 $)$ 
proof –
{
  fix  $PROB :: 'a\ problem$  and  $as$  and  $s :: 'a\ state$ 
  assume  $P1: (P\ PROB)$ 
   $(as \in valid-plans\ PROB)$ 
   $(s \in valid-states\ PROB)$ 
   $(\exists as'.$ 
   $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
   $\wedge (subseq\ as'\ as)$ 
   $\wedge (length\ as' < f\ PROB)$ 
   $)$ 
  have  $(exec-plan\ s\ as \in valid-states\ PROB)$ 
  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 $(\forall PROB\ as\ s.$

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

$\longrightarrow (\exists as'.$

$(exec-plan\ s\ as = exec-plan\ s\ as')$

$\wedge subseq\ as'\ as$

$\wedge length\ as' < f(PROB)$

```

)
)
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 ( $\forall$  PROB x.
  P PROB  $\wedge$  finite PROB
   $\rightarrow$  (x  $\in$  MPLS PROB)
   $\rightarrow$  x < f PROB
)
proof -
{
  fix PROB x
  assume P1: P PROB finite PROB x  $\in$  MPLS PROB
  - TODO refactor 'x_in_MPLS_if' and use here.
  then obtain s as where a:
    x = Inf (PLS s as) s  $\in$  valid-states PROB as  $\in$  valid-plans PROB
    unfolding MPLS-def
    by auto
  moreover have ( $\exists$  as'.
    (exec-plan s as = exec-plan s as')
     $\wedge$  (subseq as' as)
     $\wedge$  (length as' < f PROB)
  )
  using P1(1, 2) assms calculation(2, 3)
  by blast
  moreover obtain x' where x'  $\in$  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 ( $\forall$  PROB as s.
  (P PROB)
   $\wedge$  finite PROB
   $\wedge$  (s  $\in$  valid-states PROB)
   $\wedge$  (as  $\in$  valid-plans PROB)
   $\rightarrow$  ( $\exists$  as'.
    (exec-plan s as = exec-plan s as')
     $\wedge$  (subseq as' as)
     $\wedge$  (length as' < f PROB)
  )
)
shows ( $\forall$  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-plan s as = exec-plan s as')
    ∧ subseq as' as
    ∧ length as' < k
  )
) as ∈ (valid-plans PROB) (s ∈ (valid-states PROB))
shows (∃ x. (x ∈ PLS s as) ∧ 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 ∈ MPLS PROB
  shows ∃ s as. s ∈ valid-states PROB ∧ as ∈ valid-plans PROB ∧ x = Inf (PLS
s as)
  using assms
  unfolding MPLS-def
  by fast

```

lemma bound-on-all-plans-bounds-MPLS-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)
  )
) (x ∈ MPLS PROB)
shows (x < k)

```

proof —

```

obtain s as where 1: s ∈ valid-states PROB as ∈ valid-plans PROB x = Inf
(PLS s as)
  using assms(3) x-in-MPLS-if
  by blast
then obtain x' :: nat where x' ∈ 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* ($\exists k. \forall x \in \text{MPLS } \text{PROB}. x < k$)

shows $\text{Sup } (\text{MPLS } \text{PROB}) \in \text{MPLS } \text{PROB}$

proof —

obtain *k* **where** $\forall x \in \text{MPLS } \text{PROB}. x < k$

using *assms(2)*

by *blast*

moreover have *finite (MPLS PROB)*

using *assms(2) finite-nat-set-iff-bounded*

by *presburger*

moreover have $\text{MPLS } \text{PROB} \neq \{\}$

using *assms(1) MPLS-empty*

by *auto*

ultimately show $\text{Sup } (\text{MPLS } \text{PROB}) \in \text{MPLS } \text{PROB}$

unfolding *Sup-nat-def*

by *simp*

qed

lemma *bound-on-all-plans-bounds-problem-plan-bound-thesis'*:

assumes *finite PROB* ($\forall as\ s.$

$s \in (\text{valid-states } \text{PROB})$

$\wedge as \in (\text{valid-plans } \text{PROB})$

$\longrightarrow (\exists as'.$

$(\text{exec-plan } s\ as = \text{exec-plan } s\ as')$

$\wedge \text{subseq } as'\ as$

$\wedge \text{length } as' < k$

$)$

$)$

shows $\text{problem-plan-bound } \text{PROB} < k$

proof —

have $1: \forall x \in \text{MPLS } \text{PROB}. x < k$

using *assms(1, 2) bound-on-all-plans-bounds-MPLS-thesis*

by *blast*

then have $\text{Sup } (\text{MPLS } \text{PROB}) \in \text{MPLS } \text{PROB}$

using *assms(1) bounded-MPLS-contains-supremum*

by *auto*

then have $\text{Sup } (\text{MPLS } \text{PROB}) < k$

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* ($\forall as\ s.$
 $(s \in \text{valid-states } PROB)$
 $\wedge (as \in \text{valid-plans } PROB)$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s\ as = \text{exec-plan } s\ as')$
 $\wedge (\text{subseq } as'\ as)$
 $\wedge (\text{length } as' \leq k)$
 $)$
 $)$
shows (*problem-plan-bound PROB* $\leq k$)
proof –
have 1: $\forall x \in \text{MPLS } PROB. x < k + 1$
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) \in MPLS PROB*
using *assms(1) bounded-MPLS-contains-supremum*
by *fast*
then show (*problem-plan-bound PROB* $\leq k$)
unfolding *problem-plan-bound-def*
using 1
by *fastforce*
qed

lemma *bound-on-all-plans-bounds-problem-plan-bound:*

fixes $P\ f\ PROB$
assumes ($\forall PROB'\ as\ s.$
 $\text{finite } PROB' \wedge (P\ PROB') \wedge (s \in \text{valid-states } PROB')$
 $\wedge (as \in \text{valid-plans } PROB')$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s\ as = \text{exec-plan } s\ as')$
 $\wedge (\text{subseq } as'\ as)$
 $\wedge (\text{length } as' < f\ PROB')$
 $)$
 $) (P\ PROB)\ \text{finite } PROB$
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 ($s \in \text{valid-states } PROB$) ($as \in \text{valid-plans } PROB$)
shows ($\text{Inf } (PLS\ s\ as) \in (\text{MPLS } PROB)$)
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 $PROB :: 'a \text{ problem and } s :: 'a \text{ state and } as \ k$
assumes $finite \ PROB \ (s \in \text{valid-states } PROB) \ (as \in \text{valid-plans } PROB)$
 $(\text{problem-plan-bound } PROB \leq k)$
shows $(Inf \ (PLS \ s \ as) \leq \text{problem-plan-bound } PROB)$

proof –

have $Inf \ (PLS \ s \ as) \in \text{MPLS } PROB$
using $assms(2, 3) \ S\text{-VALID-AS-VALID-IMP-MIN-IN-PLS}$
by *blast*
moreover have $finite \ (\text{MPLS } PROB)$
using $assms(1) \ \text{FINITE-MPLS}$
by *blast*
ultimately have $Inf \ (PLS \ s \ as) \leq \text{Sup } (\text{MPLS } PROB)$
using *le-cSup-finite*
by *blast*
then show *?thesis*
unfolding *problem-plan-bound-def*
by *simp*

qed

lemma *PLS-NEMPTY*:

fixes $s \ as$
shows $PLS \ s \ as \neq \{\}$
unfolding *PLS-def*
by *blast*

lemma *PLS-nempty-and-has-min*:

fixes $s \ as$
shows $(\exists x. (x \in \text{PLS } s \ as) \wedge (x = Inf \ (PLS \ s \ as)))$

proof –

have $PLS \ s \ as \neq \{\}$
using *PLS-NEMPTY*
by *blast*
then have $Inf \ (PLS \ s \ as) \in \text{PLS } s \ as$
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  $(x \in PLS\ s\ as)$ 
  shows  $(\exists\ as'$ 
     $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
     $\wedge (length\ as' = x)$ 
     $\wedge (subseq\ as'\ as)$ 
  )
  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  $PROB$   $(s \in valid-states\ PROB)$   $(as \in valid-plans\ PROB)$ 
  shows  $(\exists\ as'$ 
     $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
     $\wedge (subseq\ as'\ as)$ 
     $\wedge (length\ as' \leq problem-plan-bound\ PROB)$ 
  )

```

```

proof –
  have  $problem-plan-bound\ PROB \leq 2 \wedge card\ (prob-dom\ PROB) - 1$ 
    using assms(1) bound-main-lemma
    by blast
  then have  $1: Inf\ (PLS\ s\ as) \leq problem-plan-bound\ PROB$ 
    using
      assms(1, 2, 3)
      problem-plan-bound-ge-min-pls
    by blast
  then have  $\exists x. x \in PLS\ s\ as \wedge x = Inf\ (PLS\ s\ as)$ 
    using PLS-nempty-and-has-min
    by blast
  then have  $Inf\ (PLS\ s\ as) \in (PLS\ s\ as)$ 
    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  $length\ as' \leq problem-plan-bound\ PROB$ 
    using  $1$ 
    by argo
  then show  $(\exists\ as'$ 
     $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
     $\wedge (subseq\ as'\ as)$ 
     $\wedge (length\ as' \leq problem-plan-bound\ PROB)$ 
  )
    using  $2(1)\ 2(3)$ 

```

by *blast*
qed

— NOTE name shortened.

definition *MPLS-s* **where**

$MPLS-s\ PROB\ s \equiv (\lambda (s, as). Inf (PLS\ s\ as))\ \{ (s, as) \mid as.\ as \in\ valid-plans\ PROB \}$

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

lemma *bound-main-lemma-s-3*:

fixes $PROB :: (('a, 'b)\ fmap \times ('a, 'b)\ fmap)\ set$ **and** s
shows $MPLS-s\ PROB\ s \neq \{\}$

proof –

— TODO $(s, []) \in \{\}$ could be refactored (this is used in ‘MPLS_nempty’ too).

have $[] \in\ valid-plans\ PROB$

using *empty-plan-is-valid*

by *blast*

then have $(s, []) \in \{(s, as).\ as \in\ valid-plans\ PROB\}$

by *simp*

then show $MPLS-s\ PROB\ s \neq \{\}$

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 $(\forall\ PROB\ as\ s.$

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

$\longrightarrow (\exists\ as'.$

$(exec-plan\ s\ as = exec-plan\ s\ as')$

$\wedge (subseq\ as'\ as)$

$\wedge (length\ as' < f\ PROB\ s)$

$)$

shows $(\forall\ PROB\ s\ as.$

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

$\longrightarrow (\exists\ x.$

```

    (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 $s \in \text{valid-states } PROB$ $x \in \text{MPLS-s } PROB$ s

shows $\exists as. x = \text{Inf } (PLS s as) \wedge as \in \text{valid-plans } PROB$

proof —

let $?S = \{(s, as) \mid as. as \in \text{valid-plans } PROB\}$

obtain x' **where** 1:

$x' \in ?S$

$x = (\lambda (s, as). \text{Inf } (PLS s as)) x'$

using *assms*

unfolding *MPLS-s-def*

by *blast*

let $?as = \text{snd } x'$

let $?s = \text{fst } x'$

have $?as \in \text{valid-plans } PROB$

using 1(1)

by *auto*

moreover have $?s = s$

using 1(1)

by *fastforce*

moreover have $x = \text{Inf } (PLS ?s ?as)$

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 $(\forall PROB as s.$

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

$\longrightarrow (\exists as'.$

$(\text{exec-plan } s as = \text{exec-plan } s as')$

$\wedge (\text{subseq } as' as)$

$\wedge (\text{length } as' < f \text{ } PROB s)$

)

)

shows $(\forall PROB\ x\ s.$
 $\text{finite } PROB \wedge (P\ PROB) \wedge (s \in \text{valid-states } PROB) \longrightarrow (x \in \text{MPLS-}s\ PROB$
 $s)$
 $\longrightarrow (x < f\ PROB\ s)$
 $)$
using *assms*
unfolding *MPLS-def*

proof –

have $1: \forall PROB\ s\ as.$
 $\text{finite } PROB \wedge P\ PROB \wedge as \in \text{valid-plans } PROB \wedge s \in \text{valid-states } PROB$
 \longrightarrow
 $(\exists x. x \in \text{PLS } s\ as \wedge x < f\ PROB\ s)$
using *bound-on-all-plans-bounds-PLS-s[OF assms]* .
 $\{$
fix $PROB\ x$ **and** $s :: ('a, 'b)\ fmap$
assume $P1: \text{finite } PROB\ (P\ PROB)\ (s \in \text{valid-states } PROB)$
 $\{$
assume $(x \in \text{MPLS-}s\ PROB\ s)$
then obtain as **where** $i: x = \text{Inf } (\text{PLS } s\ as)\ as \in \text{valid-plans } PROB$
using $P1\ \text{bound-on-all-plans-bounds-MPLS-}s-i$
by *blast*
then obtain x' **where** $x' \in \text{PLS } s\ as\ x' < f\ PROB\ s$
using $P1\ i\ 1$
by *blast*
then have $x < f\ PROB\ s$
using *mem-lt-imp-MIN-lt i(1)*
by *blast*
 $\}$
then have $(x \in \text{MPLS-}s\ PROB\ s) \longrightarrow (x < f\ PROB\ s)$
by *blast*
 $\}$
then show *?thesis*
by *blast*
qed

— NOTE added lemma.

lemma *Sup-MPLS-s-lt-if:*

fixes $PROB\ s\ k$
assumes $(\forall x \in \text{MPLS-}s\ PROB\ s. x < k)$
shows $\text{Sup } (\text{MPLS-}s\ PROB\ s) < k$

proof –

have $\text{MPLS-}s\ PROB\ s \neq \{\}$
using *bound-main-lemma-s-3*
by *fast*
then have $\text{Sup } (\text{MPLS-}s\ PROB\ s) \in \text{MPLS-}s\ PROB\ s$
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  $P :: 'a\ problem \Rightarrow bool$  and  $s\ f$ 
  assumes  $(\forall (PROB :: 'a\ problem)\ as\ s.$ 
     $finite\ PROB \wedge (P\ PROB) \wedge (s \in valid-states\ PROB) \wedge (as \in valid-plans$ 
     $PROB)$ 
     $\longrightarrow (\exists as'.$ 
       $(exec-plan\ s\ as = exec-plan\ s\ as')$ 
       $\wedge (subseq\ as'\ as)$ 
       $\wedge (length\ as' < f\ PROB\ s)$ 
     $)$ 
   $)$ 
  shows  $($ 
     $finite\ PROB \wedge (P\ PROB) \wedge (s \in valid-states\ PROB)$ 
     $\longrightarrow problem-plan-bound-s\ PROB\ s < f\ PROB\ s$ 
   $)$ 
proof —
  — NOTE manual instantiation is required (automation fails otherwise).
  have  $\forall (PROB :: 'a\ problem)\ x\ s.$ 
     $finite\ PROB \wedge P\ PROB \wedge s \in valid-states\ PROB$ 
     $\longrightarrow x \in MPLS-s\ PROB\ s$ 
     $\longrightarrow x < f\ PROB\ s$ 

    using assms bound-on-all-plans-bounds-MPLS-s[of P f]
    by simp
  then show
     $finite\ PROB \wedge (P\ PROB) \wedge (s \in valid-states\ PROB) \longrightarrow (problem-plan-bound-s$ 
     $PROB\ s < f\ PROB\ s)$ 
    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  $finite\ PROB\ s \in valid-states\ PROB$ 
  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  $P1: finite\ PROB\ s \in valid-states\ PROB\ as \in valid-plans\ PROB$ 
  }

```

then obtain as' **where** a : $exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as' \text{ subseq } as'$ as
 $length\ as' \leq card\ (reachable\text{-}s\ PROB\ s) - 1$
using $P1\ main\text{-}lemma\text{-}reachability\text{-}s$
by $blast$
then have $length\ as' < card\ (reachable\text{-}s\ PROB\ s)$
using $P1(1, 2)\ card\text{-}reachable\text{-}s\text{-}non\text{-}zero$
by $fastforce$
then have $(\exists as')$
 $exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as' \wedge subseq\ as'\ as \wedge length\ as' < card\ (reachable\text{-}s$
 $PROB\ s))$

using a
by $blast$
}
then have
 $finite\ PROB \wedge True \wedge s \in valid\text{-}states\ PROB$
 $\rightarrow problem\text{-}plan\text{-}bound\text{-}s\ PROB\ s < card\ (reachable\text{-}s\ PROB\ s)$

using $bound\text{-}child\text{-}parent\text{-}lemma\text{-}s\text{-}2$ [**where** $PROB = PROB$ **and** $P = \lambda\cdot. True$
and $s = s$
and $f = \lambda PROB\ s. card\ (reachable\text{-}s\ PROB\ s)$]
by $blast$
then show $?thesis$
using $assms(1, 2)$
by $blast$
qed

lemma $problem\text{-}plan\text{-}bound\text{-}s\text{-}LESS\text{-}EQ\text{-}problem\text{-}plan\text{-}bound\text{-}thm$:

fixes $PROB :: 'a\ problem$ **and** $s :: 'a\ state$
assumes $finite\ PROB\ (s \in valid\text{-}states\ PROB)$
shows $(problem\text{-}plan\text{-}bound\text{-}s\ PROB\ s < problem\text{-}plan\text{-}bound\ PROB + 1)$
proof –
{
fix $PROB :: 'a\ problem$ **and** $s :: 'a\ state$ **and** as
assume $finite\ PROB\ s \in valid\text{-}states\ PROB\ as \in valid\text{-}plans\ PROB$
then obtain as' **where** a : $exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as' \text{ subseq } as'$ as
 $length\ as' \leq problem\text{-}plan\text{-}bound\ PROB$
using $problem\text{-}plan\text{-}bound\text{-}works$
by $blast$
then have $length\ as' < problem\text{-}plan\text{-}bound\ PROB + 1$
by $linarith$
then have $\exists as'$
 $exec\text{-}plan\ s\ as = exec\text{-}plan\ s\ as' \wedge subseq\ as'\ as \wedge length\ as' \leq prob\text{-}$
 $lem\text{-}plan\text{-}bound\ PROB + 1$

using a
by $fastforce$
}

— TODO unsure why a proof is needed at all here.

then have $\forall (PROB :: 'a \text{ problem}) \text{ as } s.$
finite $PROB \wedge \text{True} \wedge s \in \text{valid-states } PROB \wedge \text{as} \in \text{valid-plans } PROB$
 $\longrightarrow (\exists \text{as}'.$
 $\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}' \wedge \text{subseq } \text{as}' \text{ as} \wedge \text{length } \text{as}' < \text{problem-plan-bound } PROB + 1)$

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

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

lemma *AS-VALID-MPLS-VALID*:
fixes $PROB \text{ as}$
assumes ($\text{as} \in \text{valid-plans } PROB$)
shows ($\text{Inf } (PLS \text{ s as}) \in \text{MPLS-s } PROB \text{ s}$)
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 $PROB :: 'a \text{ problem}$ **and** $s :: 'a \text{ state}$ **and** x
assumes *finite* $PROB \text{ s} \in (\text{valid-states } PROB) \text{ x} \in \text{MPLS-s } PROB \text{ s}$
shows ($x \leq (2 \wedge \text{card } (\text{prob-dom } PROB)) - 1$)
proof –
obtain $\text{as} :: ((a, \text{bool}) \text{ fmap} \times (a, \text{bool}) \text{ fmap}) \text{ list}$ **where** $\text{as} \in \text{valid-plans } PROB$
using *empty-plan-is-valid*
by *blast*
then obtain x **where** $1: x \in PLS \text{ s as}$ $x \leq 2 \wedge \text{card } (\text{prob-dom } PROB) - 1$
using *assms in-PLS-leq-2-pow-n*
by *blast*
then have $\text{Inf } (PLS \text{ s as}) \leq 2 \wedge \text{card } (\text{prob-dom } PROB) - 1$
using *mem-le-imp-MIN-le* [**where** $s = PLS \text{ s as}$ **and** $k = 2 \wedge \text{card } (\text{prob-dom } PROB) - 1$]
by *blast*
then have $x \leq 2 \wedge \text{card } (\text{prob-dom } PROB) - 1$
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 $\forall x \in \text{MPLS-}s \text{ } PROB \ s. \ x \leq 2^{\wedge} \text{card}(\text{prob-dom } PROB) - 1$
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* $PROB\ s \leq 2^{\wedge} \text{card}(\text{prob-dom } PROB) - 1$
 unfolding *problem-plan-bound-s-def*
 by *blast*
 qed

lemma *problem-plan-bound-s-works*:

fixes $PROB :: 'a\ \text{problem}$ and $as\ s$

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

shows $(\exists as')$

$(\text{exec-plan } s\ as = \text{exec-plan } s\ as')$

$\wedge (\text{subseq } as'\ as)$

$\wedge (\text{length } as' \leq \text{problem-plan-bound-s } PROB\ s)$

)

proof –

have *problem-plan-bound-s* $PROB\ s \leq 2^{\wedge} \text{card}(\text{prob-dom } PROB) - 1$

using *assms*(1, 3) *bound-main-lemma-s*

by *blast*

then have 1: $\text{Inf}(\text{PLS } s\ as) \leq \text{problem-plan-bound-s } PROB\ s$

using *assms* *problem-plan-bound-s-ge-min-pls*[of $PROB\ s\ as\ 2^{\wedge} \text{card}(\text{prob-dom } PROB) - 1$]

by *blast*

then obtain x **where** *obtain-x*: $x \in \text{PLS } s\ as \wedge x = \text{Inf}(\text{PLS } s\ as)$

using *PLS-nempty-and-has-min*

by *blast*

then have $\exists as'. \text{exec-plan } s\ as = \text{exec-plan } s\ as' \wedge \text{length } as' = \text{Inf}(\text{PLS } s\ as)$
 $\wedge \text{subseq } as'\ as$

using *PLS-works*[**where** $s = s$ and $as = as$ and $x = \text{Inf}(\text{PLS } s\ as)$]

obtain-x

by *fastforce*

then show $(\exists as')$

$(\text{exec-plan } s\ as = \text{exec-plan } s\ as') \wedge (\text{subseq } as'\ as)$

$\wedge (\text{length } as' \leq \text{problem-plan-bound-s } PROB\ s)$

)

using 1

by *metis*

qed

— NOTE skipped second declaration (declared twice in source).

lemma *PLS-def-ITP2015*:

fixes $s\ as$

shows $\text{PLS } s\ as = \{\text{length } as' \mid as'. (\text{exec-plan } s\ as' = \text{exec-plan } s\ as) \wedge (\text{subseq } as'\ as)\}$

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( $\beta$ ) 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( $\beta$ )
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  $\wedge$  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( $\beta$ ) x-in-MPLS-charles-then
by blast
then obtain x' where 2: x'  $\in$  PLS-charles s as PROB x'  $\leq$  2  $\wedge$  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* $PROB \leq 2^{\wedge}(\text{card } (\text{prob-dom } PROB)) - 1$
proof –
have *1: $\forall x \in \text{MPLS-charles } PROB. x \leq 2^{\wedge}(\text{card } (\text{prob-dom } PROB)) - 1$*
using *assms in-MPLS-charles-leq-2-pow-n*
by *blast*
then have *MPLS-charles* $PROB \neq \{\}$
using *assms bound-main-lemma-charles-3*
by *blast*
then have *Sup (MPLS-charles* $PROB) \leq 2^{\wedge}(\text{card } (\text{prob-dom } PROB)) - 1$
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 $\forall (PROB :: 'a \text{ problem}) \text{ as } s.$
 $(P \text{ } PROB) \wedge \text{finite } PROB \wedge (\text{as} \in \text{valid-plans } PROB) \wedge (s \in \text{valid-states } PROB)$
 $\longrightarrow (\exists \text{ as}'.$
 $(\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}') \wedge (\text{subseq as}' \text{ as}) \wedge (\text{length as}' < f \text{ } PROB))$
shows $(\forall \text{ } PROB \text{ } s \text{ as}.$
 $(P \text{ } PROB) \wedge \text{finite } PROB \wedge (\text{as} \in \text{valid-plans } PROB) \wedge (s \in \text{valid-states } PROB)$
 $\longrightarrow (\exists x.$
 $(x \in \text{PLS-charles } s \text{ as } PROB)$
 $\wedge (x < f \text{ } PROB)))$

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  $\wedge$  finite PROB  $\wedge$  as  $\in$  valid-plans PROB  $\wedge$  s  $\in$  valid-states PROB
     $\longrightarrow$  Inf {n. n  $\in$  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 ( $\forall$ (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 x.
    (P PROB)  $\wedge$  finite PROB
     $\longrightarrow$  (x  $\in$  MPLS-charles PROB)
     $\longrightarrow$  (x < f PROB)
  )

```

proof –

```

  have 1:  $\forall$ (PROB :: 'a problem) s as.
    P PROB  $\wedge$  finite PROB  $\wedge$  as  $\in$  valid-plans PROB  $\wedge$  s  $\in$  valid-states PROB
     $\longrightarrow$  Inf {n. n  $\in$  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  $\in$  MPLS-charles PROB
  then obtain s as where a:
    as  $\in$  valid-plans PROB s  $\in$  valid-states PROB x = Inf (PLS-charles s as
PROB)
    using x-in-MPLS-charles-then
    by blast
  then have Inf {n. n  $\in$  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  $\forall x \in \text{MPLS-charles } \text{PROB}. x < k$ 
  shows Sup (MPLS-charles PROB)  $\in \text{MPLS-charles } \text{PROB}$ 

```

proof –

```

  have 1:  $\text{MPLS-charles } \text{PROB} \neq \{\}$ 
  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 ( $\forall (\text{PROB} :: 'a \text{ problem}) \text{ as } s.$ 
    ( $P \text{ PROB} \wedge \text{finite } \text{PROB} \wedge (s \in \text{valid-states } \text{PROB}) \wedge (\text{as} \in \text{valid-plans } \text{PROB})$ 
       $\longrightarrow (\exists \text{ as}'.$ 
        ( $\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}'$ )
         $\wedge (\text{subseq } \text{as}' \text{ as})$ 
         $\wedge (\text{length } \text{as}' < f \text{ PROB}))$ 
    ))

```

```

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

```

proof –

```

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

```

```

moreover have 2:  $\forall x. x \in \text{MPLS-charles } \text{PROB} \longrightarrow x < f \text{PROB}$ 
  using 1 P
  by blast
moreover
{
  fix x
  assume P1:  $x \in \text{MPLS-charles } \text{PROB}$ 
  moreover have  $x < f \text{PROB}$ 
    using P(1, 2) P1 1
    by presburger
  moreover have  $\text{MPLS-charles } \text{PROB} \neq \{\}$ 
    using P1
    by blast
  moreover have  $\text{Sup } (\text{MPLS-charles } \text{PROB}) < f \text{PROB}$ 
  using calculation(3) 2 bound-child-parent-not-eq-last-diff-paths[of MPLS-charles
PROB f PROB]
    by blast
  ultimately have  $(\text{problem-plan-bound-charles } \text{PROB} < f \text{PROB})$ 
    unfolding problem-plan-bound-charles-def
    by blast
}
moreover have  $\text{Sup } (\text{MPLS-charles } \text{PROB}) \in \text{MPLS-charles } \text{PROB}$ 
  using P(2) 2 bound-on-all-plans-bounds-problem-plan-bound-charles-i
  by blast
ultimately have  $\text{problem-plan-bound-charles } \text{PROB} < f \text{PROB}$ 
  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.

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

where $d \delta$, $l \delta$ and $rd \delta$ 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  $\text{problem-plan-bound-charles } \text{PROB} \leq \text{problem-plan-bound } \text{PROB}$ 
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' ≤ prob-
    lem-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' < prob-
    lem-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 =
  λPROB. 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 exclusion 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  $P: y \in P$ 
then have  $?x \leq y$ 
  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 $finite\ \{length\ p - 1\ |\ p.\ valid_path\ Pi\ p \wedge\ distinct\ p\}$

proof —

```

{
  let  $?ss = \{length\ p - 1\ |\ p.\ valid\_path\ Pi\ p \wedge\ distinct\ p\}$ 
  let  $?ss' = \{p.\ valid\_path\ Pi\ p \wedge\ distinct\ p\}$ 
  have  $1: ?ss = (\lambda x.\ length\ x - 1)\ ' ?ss'$ 
  by blast

```

```

{

```

— NOTE type of ‘valid_states Pi’ had to be asserted to match ‘FINITE_valid_states’.

```

  let  $?S = \{p.\ distinct\ p \wedge\ set\ p \subseteq (valid\_states\ Pi :: 'a\ state\ set)\}$ 

```

```

{

```

```

  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  $x \in ?ss'$ 

```

```

  then have  $x \in ?S$ 

```

```

  proof ( $induction\ x$ )

```

```

    case ( $Cons\ a\ x$ )

```

```

    then have  $a: valid\_path\ Pi\ (a\ \# x)\ distinct\ (a\ \# x)$ 

```

```

    by blast+

```

```

    moreover {

```

```

      fix  $x'$ 

```

```

      assume  $P: x' \in set\ (a\ \# x)$ 

```

```

      then have  $x' \in valid\_states\ Pi$ 

```

```

      proof ( $cases\ x$ )

```

```

        case  $Nil$ 

```

```

        from  $a(1)\ Nil$ 

```

```

      }

```

```

    }

```

```

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

```

```

    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 = (λp. 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  $y \in ?Q \ y = \text{length } p' - 1$ 
  by blast
    — NOTE we cannot infer  $\text{length } p' - 1 \leq \text{length } p - 1$  since ‘length p =
0’ might be true.
  then have a:  $\text{length } p' - 1 \leq \text{length } p - 1$ 
    using assms(3) 1(1)
    by meson
  }
note 2 = this
{
  from finite-PLS PLS-NEMPTY
  have finite (PLS s as) PLS s as ≠ {}
    by blast+
  moreover {
    fix n
    assume P:  $(\forall y. y \in \text{PLS } s \text{ as} \longrightarrow n \leq y) \ n \in \text{PLS } s \text{ as}$ 
    from P(2) obtain as' where i:
       $n = \text{length } as' \ \text{exec-plan } s \ as' = \text{exec-plan } s \ \text{subseq } as' \ as$ 
    unfolding PLS-def
    by blast
    let ?p'=stalist' s as'
    {
      have  $\text{length } as' = \text{length } ?p' - 1$ 
        by (simp add: LENGTH-stalist')
        — MARKER (topologicalPropsScript.sml:195)
      have  $1 + (\text{length } p - 1) = \text{length } p - 1 + 1$ 
        by presburger
        — MARKER (topologicalPropsScript.sml:200)
    }
    {
      from assms(2) i(3) sublist-valid-plan
      have as' ∈ valid-plans Pi
        by blast
      then have valid-path Pi ?p'
        using assms(1) valid-path-stalist'
        by auto
    }
  }
  moreover {
    {
      assume C:  $\neg \text{distinct } ?p'$ 
      — 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-stalist'
have C-3: length as' + 1 = length ?p'
  by metis
then have ?pfx ≤ length as'
  using C-1
  by fastforce
then have C-4: exec-plan s (take ?pfx as') = rs
  using C-2 statelist'-TAKE
  by blast
let ?prsd = length (pfx @ [rs] @ drop')
let ?ap1 = take ?pfx 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 = ?pfx) ∧ (length ?ap2 = ?prsd)
  by fastforce
then have C-11: length (?ap1 @ ?asfx) < length (?ap2 @ ?asfx)
  by auto
{
  from C-10
  have ?pfx + 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)  $\in$  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  $\leq$  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  $\dots \leq 1 + (\text{length } p - 1)$ 
      using ii
      by linarith
    finally have n  $\leq$  length p - 1
      by fastforce
  }
  then have n  $\leq$  length p - 1
    by blast
  }
  ultimately have Min ?P  $\leq$  length p - 1
    using MIN-SET-ELIM'[where P=?P and Q= $\lambda$ x. x  $\leq$  length p - 1]

```

```

    by blast
  }
  note  $\beta = \text{this}$ 
  {
    have  $\text{length } p - 1 \leq \text{Max } \{\text{length } p - 1 \mid p. \text{valid-path } Pi \ p \wedge \text{distinct } p\}$ 
      using  $\text{assms}(\beta, 4) 1(1)$ 
      by ( $\text{smt Max.coboundedI bdd-aboveI bdd-above-nat}$ )
    moreover
    have  $\text{Min } (PLS \ s \ as) \leq \text{length } p - 1$ 
      using  $\beta$ 
      by blast
    ultimately
    have  $\text{Min } (PLS \ s \ as) \leq \text{Max } \{\text{length } p - 1 \mid p. \text{valid-path } Pi \ p \wedge \text{distinct } p\}$ 
      by  $\text{linarith}$ 
  }
  then show  $?thesis$ 
    by blast
qed

```

— NOTE added lemma (refactored from ‘RD_bounds_sublistD’).

```

lemma  $RD\text{-bounds-sublistD-i}$ :
  fixes  $Pi :: 'a \text{ problem and } x$ 
  assumes  $\text{finite } Pi \ (\forall y. y \in MPLS \ Pi \longrightarrow y \leq x) \ x \in MPLS \ Pi$ 
  shows  $x \leq \text{Max } \{\text{length } p - 1 \mid p. \text{valid-path } Pi \ p \wedge \text{distinct } p\}$ 
proof –
  {
    let  $?P = MPLS \ Pi$ 
    let  $?Q = \{\text{length } p - 1 \mid p. \text{valid-path } Pi \ p \wedge \text{distinct } p\}$ 
    from  $\text{assms}(\beta)$ 
    obtain  $s \ as$  where 1:
       $s \in \text{valid-states } Pi \ as \in \text{valid-plans } Pi \ x = \text{Inf } (PLS \ s \ as)$ 
    unfolding  $MPLS\text{-def}$ 
    by  $\text{fast}$ 
    have  $x \leq \text{Max } ?Q$  proof –

```

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

```

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

```

```

    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 ≠ {}
      using RD-bounds-sublistD-i-b
      by fast
    from 1(1, 2)
    have c: ∀ x. (∀ y. y ∈ ?Q → y ≤ x) ∧ x ∈ ?Q → ?limit ≤ Max ?Q
      using RD-bounds-sublistD-i-c
      by blast
    have ?limit ≤ Max ?Q
      using MAX-SET-ELIM'[where P=?Q and R=λx. ?limit ≤ Max ?Q, OF
a b c]
      by blast
  }
  note b = this
  from a b show x ≤ 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 ≤ 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 ≠ {}*

using *MPLS-nempty*

```

    by blast
  from assms
  have  $\exists: \forall x. (\forall y. y \in ?P \longrightarrow y \leq x) \wedge x \in ?P \longrightarrow x \leq \text{Max } ?Q$ 
    using RD-bounds-sublistD-i
    by blast
  have  $\text{Max } ?P \leq \text{Max } ?Q$ 
    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  $\leq$  problem-plan-bound PROB
     $\wedge$  problem-plan-bound PROB  $\leq$  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  $\times$  ('a, 'b) fmap) list and s :: ('a, 'b) fmap
    assume
      finite PROB prob-dom PROB' = {} s  $\in$  valid-states PROB' as  $\in$  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 ( $\exists as'. \text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}' \wedge \text{subseq } as' \text{ as} \wedge \text{length } as' < 1$ )
      by force
  }
  then show ?thesis
    using bound-on-all-plans-bounds-problem-plan-bound-[where  $P = \lambda P. \text{prob-dom } P = \{\}$  and  $f = \lambda P. 1$ , 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 $(\exists as'.$
 $(exec-plan\ s\ as' = exec-plan\ s\ as)$
 $\wedge (subseq\ as'\ as)$
 $\wedge (length\ as' \leq problem-plan-bound\ PROB)$
 $\wedge (sat-precond-as\ s\ as')$
 $)$

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 implicitly 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 $(\forall as\ s.$

$(s \in valid-states\ PROB)$

$\wedge (as \in valid-plans\ PROB)$

$\longrightarrow (\exists as'.$

$(exec-plan\ s\ as = exec-plan\ s\ as')$

```

       $\wedge$  subseq  $as'$   $as$ 
       $\wedge$  (length  $as' < f$   $PROB$ )
    )
  ) finite  $PROB$ 
  shows (problem-plan-bound  $PROB < f$   $PROB$ )
  proof –
  let  $?P = \lambda Pr. PROB = Pr$ 
  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 $as \equiv$ 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 $as \neq \{\}$
unfolding *traversed-states-def*
 by (induction as) auto

lemma *traversed-states-geq-1*:
 fixes s
 shows $1 \leq$ card (traversed-states s as)
 proof –
 have card (traversed-states s as) $\neq 0$
 using *traversed-states-nempty* *finite-traversed-states* *card-0-eq*
 by blast
 then show $1 \leq$ card (traversed-states s as)
 by linarith
 qed

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

— NOTE name shortened.

definition *td* **where**
td $PROB \equiv$ Sup {

$$\{ \text{card } (\text{traversed-states } (fst p) (snd p)) - 1 \mid p. (fst p \in \text{valid-states } PROB) \wedge (snd p \in \text{valid-plans } PROB) \}$$

lemma *traversed-states-rem-condless-act*: $\bigwedge s.$
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
 $\{ \text{card } (\text{traversed-states } (fst p) (snd p)) - 1 \mid p. (fst p \in \text{valid-states } PROB) \wedge (snd p \in \text{valid-plans } PROB) \}$
 $\neq \{ \}$

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 $[] \in \text{valid-plans } PROB$
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* \equiv (*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* \equiv ($\lambda a.$ *action-proj a vs*) ' *PROB*

— NOTE using 'fun' due to multiple defining equations.

— NOTE name shortened.

fun *as-proj* **where**

as-proj [] - = []

| *as-proj (a # as) vs* = (*if fmdom' (fmrestrict-set vs (snd a)) \neq {}*
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 (fndom' (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 ⊆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 ⊆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) ⊆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) ⊆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.prem1 proj-state-succ
      by fastforce
    then show ?thesis
      unfolding exec-plan.simps sat-precond-as.simps as-proj.simps
      using Cons.IH Cons.prem1 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.prem1 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 $\text{prob-dom } (\text{prob-proj } PROB \text{ vs}) = (\text{prob-dom } PROB \cap \text{vs})$
proof –
have
 $\text{prob-dom } (\text{prob-proj } PROB \text{ vs})$
 $= ($
 $\bigcup (s1, s2) \in (\lambda a. (\text{fmrestrict-set vs } (\text{fst } a), \text{fmrestrict-set vs } (\text{snd } a)))$
 $\quad \text{' } PROB. \text{ action-dom } s1 \text{ } s2$
 $)$

unfolding *prob-dom-def prob-proj-def action-proj-def*
by *blast*
moreover
{
have
 $(\text{prob-dom } PROB \cap \text{vs})$
 $= (\bigcup a \in PROB. \text{ action-dom } (\text{fst } a) (\text{snd } a) \cap \text{vs})$

unfolding *prob-dom-def prob-proj-def*
using *SUP-cong*
by *auto*
also have $\dots = (\bigcup a \in PROB. \text{ action-dom } (\text{fst } (\text{action-proj } a \text{ vs})) (\text{snd } (\text{action-proj } a \text{ vs})))$
using *proj-action-dom-eq-inter[symmetric]*
by *fast*
finally have
 $(\text{prob-dom } PROB \cap \text{vs})$
 $= (\bigcup a \in PROB. \text{ fmdom}' (\text{fmrestrict-set vs } (\text{fst } a)) \cup \text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a)))$

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 $PROB \text{ vs}$
assumes $(s \in \text{valid-states } PROB)$
shows $((\text{fmrestrict-set vs } s) \in \text{valid-states } (\text{prob-proj } PROB \text{ vs}))$

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 $\dots = fmrestrict_set\ vs\ s\ ++_f\ fmrestrict_set\ vs\ (fmrestrict_set\ vs\ (snd\ h))$

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 $\dots = fmrestrict_set\ vs\ s\ ++_f\ fmrestrict_set\ vs\ (snd\ h)$
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))  $\neq$  {}
  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 ( $a \in PROB$ )
  shows (action-proj  $a$   $vs \in prob-proj$   $PROB$   $vs$ )
  unfolding action-proj-def prob-proj-def
  using assms
  by simp

```

```

lemma valid-as-valid-as-proj:
  fixes  $PROB$   $vs$ 
  assumes ( $as \in valid-plans$   $PROB$ )
  shows ( $as-proj$   $as$   $vs \in valid-plans$  ( $prob-proj$   $PROB$   $vs$ ))
  using assms
proof (induction  $as$  arbitrary:  $PROB$   $vs$ )
  case (Cons  $a$   $as$ )
  then show ?case
    using assms Cons
  proof (cases fndom' (fmrestrict-set  $vs$  (snd  $a$ ))  $\neq \{\}$ )
    case True
    then have  $1$ :  $as-proj$  ( $a \# as$ )  $vs = action-proj$   $a$   $vs \# as-proj$   $as$   $vs$ 
      using True
      by simp
    then have  $as \in valid-plans$   $PROB$ 
      using Cons.prems valid-plan-valid-tail
      by fast
    then have  $as-proj$   $as$   $vs \in valid-plans$  ( $prob-proj$   $PROB$   $vs$ )
      using Cons.IH  $1$ 

```

```

    by simp
  then have action-proj a vs # as-proj as vs ∈ valid-plans (prob-proj PROB vs)
  using Cons.premis 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 explicitly 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 $\dots < \text{length } (as\text{-proj } as \text{ vs})$
using *assms(5)*
by *blast*
finally have $2 \wedge \text{card } (fmdom' (fmrestrict\text{-set } vs \ s)) - 1 < \text{length } (as\text{-proj } as$
 $vs)$
by *blast*
}
note $1 = \text{this}$
moreover have $fmrestrict\text{-set } vs \ s \in \text{valid}\text{-states } (prob\text{-proj } PROB \ vs)$
using *assms(2) graph-plan-not-eq-last-diff-paths*
by *blast*
moreover have $as\text{-proj } as \ vs \in \text{valid}\text{-plans } (prob\text{-proj } PROB \ vs)$
using *assms(3) valid-as-valid-as-proj*
by *blast*
moreover have *finite* $(prob\text{-proj } PROB \ vs)$
using *assms(1) finite-imp-finite-prob-proj*
by *blast*
ultimately show *?thesis*
using *lemma-2* [**where** $PROB = prob\text{-proj } PROB \ vs$ **and** $as = as\text{-proj } as \ vs$ **and**
 $s = fmrestrict\text{-set } vs \ s]$
by *blast*
qed

lemma *as-proj-eq-filter-action-proj*:
fixes $as \ vs$
shows $as\text{-proj } as \ vs = \text{filter } (\lambda a. fmdom' (snd \ a) \neq \{\}) (\text{map } (\lambda a. \text{action}\text{-proj } a$
 $vs) \ as)$
by $(\text{induction } as) (\text{auto simp add: action}\text{-proj}\text{-def})$

lemma *append-eq-as-proj*:
fixes $as1 \ as2 \ as3 \ p \ vs$
assumes $(as1 \ @ \ as2 \ @ \ as3 = as\text{-proj } p \ vs)$
shows $(\exists p\text{-1 } p\text{-2 } p\text{-3}.$
 $(p\text{-1} \ @ \ p\text{-2} \ @ \ p\text{-3} = p)$
 $\wedge (as2 = as\text{-proj } p\text{-2} \ vs)$
 $\wedge (as1 = as\text{-proj } p\text{-1} \ vs)$
 $)$
using *assms append-eq-as-proj-1 as-proj-eq-filter-action-proj*
by $(metis \ (no\text{-types}, \ \text{lifting}))$

lemma *succ-drest-eq-drest-succ*:
fixes $a \ s \ vs$
shows

$$\begin{aligned} & \text{state-succ (fmrestrict-set vs s) (action-proj a vs)} \\ & = \text{fmrestrict-set vs (state-succ s (action-proj a vs))} \end{aligned}$$

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))  $\subseteq_f$  fmre-
    strict-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:  $\neg(\text{fst (fmrestrict-set vs (fst a), fmrestrict-set vs (snd a))} \subseteq_f \text{fmre-}$ 
    strict-set vs s)
    then have a:  $\neg(\text{fst (fmrestrict-set vs (fst a), fmrestrict-set vs (snd a))} \subseteq_f s)$ 
      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
 $fmrestrict\text{-}set\ vs\ (exec\text{-}plan\ (fmrestrict\text{-}set\ vs\ s)\ (as\text{-}proj\ as\ vs))$
 $=\ exec\text{-}plan\ (fmrestrict\text{-}set\ vs\ s)\ (as\text{-}proj\ as\ vs)$

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

lemma *proj-exec-proj-eq-exec-proj'*:
 fixes s as vs
 shows
 $fmrestrict\text{-}set\ vs\ (exec\text{-}plan\ (fmrestrict\text{-}set\ vs\ s)\ (as\text{-}proj\ as\ vs))$
 $=\ fmrestrict\text{-}set\ vs\ (exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs))$

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

lemma *graph-plan-lemma-9*:
 fixes s as vs
 shows
 $fmrestrict\text{-}set\ vs\ (exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs))$
 $=\ exec\text{-}plan\ (fmrestrict\text{-}set\ vs\ s)\ (as\text{-}proj\ as\ vs)$
 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 $fmdom'\ (snd\ (action\text{-}proj\ a\ vs)) \subseteq fmdom'\ (snd\ a)$
proof –
 have $snd\ (action\text{-}proj\ a\ vs) = fmrestrict\text{-}set\ vs\ (snd\ a)$
 unfolding *action-proj-def*
 by *simp*

```

then have  $fmlookup (fmrestrict\text{-}set\ vs\ (snd\ a)) \subseteq_m fmlookup (snd\ a)$ 
  by (simp add: map-le-def fmdom'-restrict-set-precise)
then have  $dom (fmlookup (fmrestrict\text{-}set\ vs\ (snd\ a))) \subseteq dom (fmlookup (snd\ a))$ 
  using map-le-implies-dom-le
  by blast
then have  $fmdom' (fmrestrict\text{-}set\ vs\ (snd\ a)) \subseteq fmdom' (snd\ a)$ 
  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  $s \in valid\text{-}states\ PROB (as \in valid\text{-}plans\ PROB)$ 
shows  $(exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs) \in valid\text{-}states\ PROB)$ 
using assms
proof (induction as arbitrary: s PROB vs)
case (Cons a as)
then have  $1: as \in valid\text{-}plans\ PROB$ 
  using Cons.prem(2) valid-plan-valid-tail
  by fast
then have  $2: exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs) \in valid\text{-}states\ PROB$ 
  using Cons.prem(1) Cons.IH(1)
  by blast
  — NOTE split on the if-then branch introduced by 'as_proj'.
moreover {
  assume  $P: fmdom' (fmrestrict\text{-}set\ vs\ (snd\ a)) \neq \{\}$ 
  then have
     $exec\text{-}plan\ s\ (as\text{-}proj\ (a\ \# \ as)\ vs)$ 
     $= exec\text{-}plan (state\text{-}succ\ s (action\text{-}proj\ a\ vs)) (as\text{-}proj\ as\ vs)$ 

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

    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  $s \in valid\text{-}states\ PROB \implies fmdom' s = prob\text{-}dom$ 

```

PROB.

```

{
  have  $s \in \text{valid-states } PROB$ 
    using Cons.prems
    by simp
  then have  $s \in \{s'. \text{fmdom}' s' = \text{prob-dom } PROB\}$ 
    unfolding valid-states-def
    by simp
  then obtain  $s'$  where  $s' = s$   $\text{fmdom}' s' = \text{prob-dom } PROB$ 
    by auto
  then have  $\text{fmdom}' s = \text{prob-dom } PROB$ 
    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  $(\text{snd } (\text{action-proj } a \text{ vs}) ++ s) = (s ++_f \text{fmrestrict-set vs } (\text{snd } a))$ 
    unfolding action-proj-def fmap-add-ltr-def
    by simp
  then have  $a: a \in PROB$ 
    using Cons.prems(2) valid-plan-valid-head
    by fast
  then have  $\text{action-dom } (\text{fst } a) (\text{snd } a) \subseteq \text{prob-dom } PROB$ 
    using exec-as-proj-valid-2
    by blast
  then have  $\text{fmdom}' (\text{snd } a) \subseteq \text{action-dom } (\text{fst } a) (\text{snd } a)$ 
    unfolding action-dom-def
    by simp
  then have  $\text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a)) \subseteq \text{fmdom}' (\text{snd } a)$ 
    using action-proj-def act-dom-proj-eff-subset-act-dom-eff snd-conv
    by metis
  then have  $\text{fmdom}' (\text{fmrestrict-set vs } (\text{snd } a)) \subseteq \text{prob-dom } PROB$ 
    using FDOM-eff-subset-prob-dom-pair a
    by blast
  then have  $\text{fmdom}' (s ++_f \text{fmrestrict-set vs } (\text{snd } a)) = \text{fmdom}' s$ 
    by (simp add: calculation sup.absorb-iff1)
}
ultimately have  $(\text{snd } (\text{action-proj } a \text{ vs}) ++ s) \in \text{valid-states } PROB$ 
  unfolding action-proj-def fmap-add-ltr-def valid-states-def
  by simp
}
then have  $\text{exec-plan } s (\text{as-proj } (a \# \text{as}) \text{ vs}) \in \text{valid-states } PROB$ 
  using 1 3 calculation(1) Cons.IH [where  $s = \text{snd } (\text{action-proj } a \text{ vs}) ++ s$ ]
  by presburger
}
moreover {
  assume  $\neg(\text{fst } (\text{action-proj } a \text{ vs}) \subseteq_f s)$ 
  then have
     $\text{exec-plan } (\text{state-succ } s (\text{action-proj } a \text{ vs})) (\text{as-proj } \text{as } \text{vs})$ 

```



```

    = 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: fmfiter-alt-defs(4)*)

lemma *action-proj-idempot'*:
 fixes $a\ vs$
 assumes $(action\ dom\ (fst\ a)\ (snd\ a) \subseteq vs)$
 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: $(fmdom'\ (fst\ a) \cup fmdom'\ (snd\ a)) \subseteq vs$
 unfolding *action-dom-def*
 using *assms*
 by (*auto simp add: action-dom-def*)
 — NOTE Show that both components of 'a' remain unchanged.
 {
 then have $fmdom'\ (fst\ a) \subseteq vs$
 by *blast*
 then have $fmrestrict\ set\ vs\ (fst\ a) = (fst\ a)$
 using *exec-drest-5*
 by *auto*
 }
 moreover {
 have $fmdom'\ (snd\ a) \subseteq vs$
 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 $prob\ dom\ P \subseteq vs$

```

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 ⊆f s sat-precond-as (state-succ s a) as
    using Cons.premis(1)
    by simp+
  then have 2: fmrestrict-set vs (fst a) ⊆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) ⊆f s'
        ∧ sat-precond-as (state-succ s' (action-proj a vs)) (as-proj as vs)
      )
    using calculation
    by simp
  }

```

```

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

        using P1 Cons.IH Cons.prem(1, 2) Cons2
      by (metis sat-precond-as-proj-3 empty-eff-not-in-as-proj sat-precond-as.simps(2))
      then have  $b: \text{fst } a' \subseteq_f s'$ 
        unfolding sat-precond-as.simps(2)
      using P1 Cons.IH Cons.prem(1, 2) sat-precond-as-proj-3 empty-eff-not-in-as-proj
      by (metis sat-precond-as.simps(2))
      then have  $\text{sat-precond-as } (\text{state-succ } s' a') \text{ list}$ 
        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.prem*s
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.prem*s(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.prem*s(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.prem*s(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* \in *valid-plans PROB*) (*prob-dom PROB* \subseteq *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 explicitly because it's so pivotal.
then have *fmdom' (fmrestrict-set vs (snd a))* \neq $\{\}$

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.prem(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.prem(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.prem(1)*
by *simp*
then have *as ∈ valid-plans PROB*
using *Cons.prem(2) valid-plan-valid-tail*
by *fast*
then have *as-proj as vs = as*
using *Cons.prem(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 True
            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 $(vs1 \subseteq vs2)$
shows $fmrestrict\text{-}set\ vs1\ (fmrestrict\text{-}set\ vs2\ f) = fmrestrict\text{-}set\ vs1\ f$
using *assms*
proof –
{
 fix v
 have $fmlookup\ (fmrestrict\text{-}set\ vs1\ (fmrestrict\text{-}set\ vs2\ f))\ v = fmlookup\ (fmrestrict\text{-}set\ vs1\ f)\ v$
 using *assms*
 proof (*cases* $v \in vs1$)
 case *False*
 then show *?thesis*
 proof (*cases* $v \in vs2$)
 case *False*
 then have $v \notin vs1$
 using *False assms*
 by *blast*
 then have
 $fmlookup\ (fmrestrict\text{-}set\ vs1\ (fmrestrict\text{-}set\ vs2\ f))\ v = None$
 $fmlookup\ (fmrestrict\text{-}set\ vs1\ f)\ v = None$
 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 $(vs1 \subseteq vs2)$
shows $(action\text{-}proj\ (action\text{-}proj\ a\ vs2)\ vs1 = action\text{-}proj\ a\ vs1)$
using *assms*
proof –
 have
 $fmrestrict\text{-}set\ vs1\ (fmrestrict\text{-}set\ vs2\ (fst\ a)) = fmrestrict\text{-}set\ vs1\ (fst\ a)$
 $fmrestrict\text{-}set\ vs1\ (fmrestrict\text{-}set\ vs2\ (snd\ a)) = fmrestrict\text{-}set\ vs1\ (snd\ a)$
 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 $vs1 \subseteq vs2$
shows $prob\text{-}proj (prob\text{-}proj\ PROB\ vs2)\ vs1 = prob\text{-}proj\ PROB\ vs1$
proof –
{
 have
 $prob\text{-}proj (prob\text{-}proj\ PROB\ vs2)\ vs1$
 $= ((\lambda a. action\text{-}proj\ a\ vs1) \circ (\lambda a. action\text{-}proj\ a\ vs2)) \text{ ‘ } PROB$

 unfolding *prob-proj-def*
 by *fastforce*
 also have $\dots = (\lambda a. action\text{-}proj (action\text{-}proj\ a\ vs2)\ vs1) \text{ ‘ } PROB$
 by *fastforce*
 also have $\dots = (\lambda a. action\text{-}proj\ a\ vs1) \text{ ‘ } PROB$
 using *assms subset-proj-absorb-1*
 by *metis*
 also have $\dots = prob\text{-}proj\ PROB\ vs1$
 unfolding *prob-proj-def*
 by *simp*
 finally have $prob\text{-}proj (prob\text{-}proj\ PROB\ vs2)\ vs1 = prob\text{-}proj\ PROB\ vs1$
 by *simp*
}
then show *?thesis*
by *simp*
qed

lemma *union-proj-absorb*:
fixes *PROB vs vs'*
shows $prob\text{-}proj (prob\text{-}proj\ PROB\ (vs \cup vs'))\ vs = prob\text{-}proj\ PROB\ vs$
by (*simp add: subset-proj-absorb*)

lemma *NOT-VS-IN-DOM-PROJ-PRE-EFF*:
fixes *ROB vs v a*
assumes $\neg(v \in vs) (a \in ROB)$
shows (
 $((v \in fmdom' (fst\ a)) \longrightarrow (v \in fmdom' (fst (action\text{-}proj\ a (prob\text{-}dom\ ROB - vs))))))$
 $\wedge ((v \in fmdom' (snd\ a)) \longrightarrow (v \in fmdom' (snd (action\text{-}proj\ a (prob\text{-}dom\ ROB - vs))))))$
)
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 ( $v \in vs'$ ) ( $v' \in vs'$ ) (disjnt vs vs')
  shows ( $dep\ PROB\ v\ v' \longrightarrow dep\ (prob-proj\ PROB\ (prob-dom\ PROB - vs))\ v\ v'$ )
  using assms
  proof (cases v = v')
  case False
  {
    assume P: dep PROB v v'
    then obtain a where a:
      ( $v \in fmdom'\ (fst\ a) \wedge v' \in fmdom'\ (snd\ a) \vee v \in fmdom'\ (snd\ a) \wedge v' \in$ 
fmdom' (snd a))
       $a \in PROB$ 
      unfolding dep-def
      using False
      by blast
    {
      have  $v \notin vs$ 
      using assms(1, 3)
      unfolding disjnt-def
      by blast
      then have ( $v \in fmdom'\ (fst\ a) \longrightarrow v \in fmdom'\ (fst\ (action-proj\ a\ (prob-dom\$ 
PROB - vs)))))
      ( $v \in fmdom'\ (snd\ a) \longrightarrow v \in fmdom'\ (snd\ (action-proj\ a\ (prob-dom\ PROB$ 
- vs)))))
      using a NOT-VS-IN-DOM-PROJ-PRE-EFF
      by metis+
    }
    note  $b = this$ 
    then consider (i)  $v \in fmdom'\ (fst\ a) \wedge v' \in fmdom'\ (snd\ a)$ 
      | (ii)  $v \in fmdom'\ (snd\ a) \wedge v' \in fmdom'\ (snd\ a)$ 
      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 $(vs \subseteq prob-dom\ PROB)$
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
 $(fmdom' (fst (action-proj\ a\ vs)) \subseteq fmdom' (fst\ a))$
 $\wedge (fmdom' (snd (action-proj\ a\ vs)) \subseteq fmdom' (snd\ a))$
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
 $(\neg(v \in fmdom' (fst\ a)) \longrightarrow \neg(v \in fmdom' (fst (action-proj\ a\ vs))))$
 $\wedge (\neg(v \in fmdom' (snd\ a)) \longrightarrow \neg(v \in fmdom' (snd (action-proj\ a\ vs))))$
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 $(\neg \text{dep-var-set } \text{PROB } \text{vs } \text{vs}')$
shows $(\neg \text{dep-var-set } (\text{prob-proj } \text{PROB } \text{vs}'') \text{ vs } \text{vs}')$
using *assms dep-proj-dep*
unfolding *dep-var-set-def*
by *metis*

lemma *SUBSET-PROJ-DOM-DISJ*:

fixes *PROB vs vs'*
assumes $(\text{vs} \subseteq (\text{prob-dom } (\text{prob-proj } \text{PROB } (\text{prob-dom } \text{PROB } - \text{vs}'))))$
shows *disjnt vs vs'*
using *assms*
by $(\text{auto simp add: } \text{PROB-DOM-PROJ-DIFF subset-iff disjnt-iff})$

— TODO showcase (lemma which is solved effortlessly by automation).

lemma *NOT-VS-DEP-IMP-DEP-PROJ*:

fixes *PROB vs v v'*
assumes $\neg(v \in \text{vs}) \neg(v' \in \text{vs}) (\text{dep } \text{PROB } v v')$
shows $(\text{dep } (\text{prob-proj } \text{PROB } (\text{prob-dom } \text{PROB } - \text{vs})) v v')$
using *assms*
by $(\text{metis } \text{Diff-disjoint Diff-iff disjnt-def insertCI IN-DISJ-DEP-IMP-DEP-DIFF})$

lemma *DISJ-PROJ-NDEP-IMP-NDEP*:

fixes *PROB vs vs' vs''*
assumes
 $(\text{disjnt } \text{vs } \text{vs}'') \text{ disjnt } \text{vs } \text{vs}'$
 $\neg(\text{dep-var-set } (\text{prob-proj } \text{PROB } (\text{prob-dom } \text{PROB } - \text{vs})) \text{vs}' \text{vs}'')$
shows $\neg(\text{dep-var-set } \text{PROB } \text{vs}' \text{vs}'')$
proof –
{
assume *C*: *dep-var-set* *PROB vs' vs''*
then obtain *v1 v2* **where** $v1 \in \text{vs}' v2 \in \text{vs}'' \text{disjnt } \text{vs}' \text{vs}'' \text{dep } \text{PROB } v1 v2$
unfolding *dep-var-set-def*
by *blast*
then have $\exists v1 v2.$
 $v1 \in \text{vs}' \wedge v2 \in \text{vs}'' \wedge \text{disjnt } \text{vs}' \text{vs}'' \wedge \text{dep } (\text{prob-proj } \text{PROB } (\text{prob-dom } \text{PROB } - \text{vs})) v1 v2$

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

```

lemma *every-action-proj-eq-as-proj:*

```

fixes as vs
shows list-all ( $\lambda a. \text{action-proj } a \text{ vs} = a$ ) (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  $\text{fndom}' (\text{snd } (\text{action-proj } a \text{ vs})) = \{\}$ 
shows ( $\text{as-proj } a \text{ vs} = \text{as-proj } (a \# \text{as}) \text{ vs}$ )
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 ( $\text{as-proj } as' \text{ vs} = as'$ )
using assms
proof (induction as arbitrary: as' vs)
case Nil
moreover have  $as' = []$ 
using Nil.premis 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.prem1 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 $(\text{varset-action } a\ vs)\ (\text{fst } a \subseteq_f s)\ (\text{fmdom}'(snd\ a) \subseteq \text{fmdom}'\ s)$

shows $(\text{state-succ } s\ (\text{action-proj } a\ vs) = \text{state-succ } s\ a)$

proof –

have 1: $\text{fmdom}'(snd\ a) \cap (\text{fmdom}'\ s - vs) = \{\}$

using *assms(1) vset-disj-eff-diff*

by *blast*

then have 2:

$\text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ s = \text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ (\text{state-succ } s\ a)$

using *disj-imp-eq-proj-exec[where vs = fmdom' s - vs]*

by *blast*

then have $\text{fmdom}'(snd\ (\text{action-proj } a\ vs)) \cap (\text{fmdom}'\ s - vs) = \{\}$

using 1 *DISJ-EFF-DISJ-PROJ-EFF[where s = (fmdom' s - vs)]*

by *blast*

then have

$\text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ s$

$= \text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ (\text{state-succ } s\ (\text{action-proj } a\ vs))$

using *disj-imp-eq-proj-exec[where a = (action-proj a vs) and vs = fmdom' s - vs]*

by *blast*

then have $\text{fmdom}'(snd\ (\text{action-proj } a\ vs)) \cap (\text{fmdom}'\ s - vs) = \{\}$

using 1 *DISJ-EFF-DISJ-PROJ-EFF[where s = (fmdom' s - vs)]*

by *blast*

then have

$\text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ s =$

$\text{fmrestrict-set } (\text{fmdom}'\ s - vs)\ (\text{state-succ } s\ (\text{action-proj } a\ vs))$

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 $s' = \text{state-succ } s\ a$ **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 fmfiter-alt-defs(4))
  also have ... ⊆ fmdom' (snd a)
    by simp
  finally have fmdom' (snd (
    fmrrestrict-set (prob-dom PROB - vs') (fst a)
    , fmrrestrict-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)  $\subseteq$  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  $\subseteq$  vs') (varset-action a vs')
  shows (varset-action (action-proj a vs') vs  $\longleftrightarrow$  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  $\neq$  {}
    have  $\forall a. (fmrestrict-set \{ \} (fst a), fmrestrict-set \{ \} (snd a)) = (fmempty,
fmempty)
      using fmrestrict-set-null
      by simp
    then have prob-proj PROB {} = {(fmempty, fmempty)}
      unfolding prob-proj-def action-proj-def
      using P
      by auto
  }
  ultimately consider
  (i) prob-proj PROB {} = {}
  | (ii) prob-proj PROB {} = {(fmempty, fmempty)}
  by (cases PROB = {}) force+$ 
```

```

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'
 $subseq\ as'\ (as-proj\ as\ vs) \wedge length\ as' \leq problem-plan-bound (prob-proj\ PROB$
 $vs)$
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 $length (rem-condless-act (fmrestrict-set\ vs\ s) []\ as') \leq length\ as'$
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
 $length (rem-effectless-act (rem-condless-act (fmrestrict-set\ vs\ s) []\ as'))$
 $\leq length (rem-condless-act (fmrestrict-set\ vs\ s) []\ as')$

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 $\equiv (\forall v. (v \in \text{fmdom}' s1) \wedge (v \in \text{fmdom}' s2) \longrightarrow (\text{fmlookup } s1 \ v = \text{fmlookup } s2 \ v))$

— 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  $\forall x. x \in \text{fmdom}' (\text{snd } a) \longrightarrow \text{fmlookup } (\text{snd } a) \ x = \text{fmlookup } s \ x$ 
    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  $fmdom' (fmrestrict-set X f) = X \cap (fmdom' f)$ 
  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  $fmdom' (fmrestrict-set X (f ++_f g)) = X \cap (fmdom' f \cup fmdom' g)$ 
proof –
  have  $fmrestrict-set X (f ++_f g) = fmrestrict-set X f ++_f fmrestrict-set X g$ 
    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  $agree (fmrestrict-set X f) (fmrestrict-set X g) x \in X \cap fmdom' f \cap fmdom' g$ 
  shows  $fmlookup (fmrestrict-set X f) x = fmlookup (fmrestrict-set X g) x$ 
proof –
  {
    fix v
    assume  $v \in X \cap fmdom' f \cap fmdom' g$ 
    then have  $v \in fmdom' (fmrestrict-set X f) \wedge v \in fmdom' (fmrestrict-set X g)$ 
      using fmdom'-fmrestrict-set
      by force
    then have  $fmlookup (fmrestrict-set X f) v = fmlookup (fmrestrict-set X g) v$ 
      using assms(1)
      unfolding agree-def
      by blast
  }
then show ?thesis

```


using *assms*
by *blast*
qed

lemma *agree-restrict-state-succ-idempot:*

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

```

        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.prem(1) valid-plan-valid-head
    by fast
  then have 2: as ∈ valid-plans PROB
    using Cons.prem(1) valid-plan-valid-tail
    by fast
  then have 3: ∀ a. ListMem a as → agree (snd a) s
    using Cons.prem(3) ListMem.simps
    by metis
  then have ListMem a (a # as)
    using elem
    by fast
  then have agree (snd a) s
    using Cons.prem(3)
    by blast
  then have 4: state-succ s a = s
    using Cons.prem(1, 2) 1 agree-state-succ-idempot
    by blast
  then have exec-plan s as = s
    using Cons.IH Cons.prem(2) 2 3
    by blast
  then show ?case

```

using 4
by *simp*
qed *simp*

lemma *agree-restrict-exec-idempot*:

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

```

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

```

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

```

  fixes  $s s'$ 
  assumes  $(as \in \text{valid-plans } PROB) (s' \in \text{valid-states } PROB) (s \in \text{valid-states } PROB)$ 
    ( $\forall p e. \text{ListMem } (p, e) as \longrightarrow \text{agree } (\text{fmrestrict-set vs } e) (\text{fmrestrict-set vs } s)$ )
    ( $\text{fmrestrict-set vs } s' = \text{fmrestrict-set vs } s$ )
  shows  $\text{fmrestrict-set vs } (\text{exec-plan } s' as) = \text{fmrestrict-set vs } s$ 
    using assms agree-restrict-exec-idempot
    by fastforce

```

lemma *agree-comm*: $\text{agree } x x' = \text{agree } x' x$

```

  unfolding agree-def
  by fastforce

```

lemma *restricted-agree-imp-agree*:

```

assumes ( $fmdom' s2 \subseteq vs$ ) ( $agree (fmrestrict-set vs s1) s2$ )
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 ( $\forall fm'. ListMem fm' fmList \longrightarrow agree fm fm'$ )
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  $\forall fm'. ListMem fm' fmList \longrightarrow agree fm fm'$ 
using Cons.premis 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.premis 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 $(fmdom' s2 \subseteq vs2) (fmdom' s1 \subseteq vs1)$
shows $((fmrestrict\text{-}set\ vs2\ s1 = fmrestrict\text{-}set\ vs1\ s2) \longrightarrow agree\ s1\ s2)$
using *assms fmdom'-restrict-set restricted-agree-imp-agree*
by *(metis agree-def)*

lemma *SUBMAPS-AGREE*: $(s1 \subseteq_f s) \wedge (s2 \subseteq_f s) \implies (agree\ s1\ s2)$

unfolding *agree-def*
by *(metis as-needed-asses-submap-exec-ii)*

— NOTE name shortened.

definition *snapshot* **where**

snapshot $PROB\ s = \{a \mid a. a \in PROB \wedge agree\ (fst\ a)\ s \wedge agree\ (snd\ a)\ s\}$

lemma *snapshot-pair*: $snapshot\ PROB\ s = \{(p, e). (p, e) \in PROB \wedge agree\ p\ s \wedge agree\ e\ s\}$

unfolding *snapshot-def*
by *fastforce*

lemma *action-agree-valid-in-snapshot*:

assumes $(a \in PROB) (agree\ (fst\ a)\ s) (agree\ (snd\ a)\ s)$
shows $(a \in snapshot\ PROB\ s)$
unfolding *snapshot-def*
using *assms*
by *blast*

lemma *as-mem-agree-valid-in-snapshot*:

assumes $(\forall a. ListMem\ a\ as \longrightarrow agree\ (fst\ a)\ s \wedge agree\ (snd\ a)\ s) (as \in valid\text{-}plans\ PROB)$

shows $(as \in valid\text{-}plans\ (snapshot\ PROB\ s))$

using *assms*

proof *(induction as)*

case *Nil*

then show *?case*

using *empty-plan-is-valid*

by *blast*

next

case $(Cons\ a\ as)$

{

have $\forall a. ListMem\ a\ as \longrightarrow agree\ (fst\ a)\ s \wedge agree\ (snd\ a)\ s$

using *Cons.prems(1) insert*

```

    by fast
  moreover have (as ∈ valid-plans PROB)
    using Cons.prem(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.prem(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.prem(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 fmllookup (fmrestrict-set X f) v = fmllookup (fmrestrict-set X g) v
    unfolding fmllookup-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 ∈ vsa
  shows
    fmllookup (fmrestrict-set ((vsa ∪ vsb) ∩ vs) f) v
    = fmllookup (fmrestrict-set (vsa ∩ vs) f) v

```

```

  unfolding fmllookup-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 fmllookup ?f v = fmllookup ?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 fmllookup-add fmllookup-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 fmllookup (fmrestrict-set (vsa ∪ vsb) fm) v = None
  using assms(4, 5)
  by (metis Int-iff Un-iff fmllookup-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)  $v \in vs1$ 
  | (ii)  $v \in vs2$ 
  | (iii)  $\neg v \in vs1 \wedge \neg v \in vs2$ 
  by blast
then have  $fmlookup\ ?f\ v = fmlookup\ s\ v$ 
proof (cases)
  case i
  then have  $v \in fmdom'\ fma$ 
    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  $\dots = fmlookup\ (fmrestrict-set\ vs1\ fma)\ v$ 
    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* \subseteq *vs1*) (*fmdom' s2* \subseteq *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* ($\lambda a.$ *agree* (*fst a*) *s* \wedge *agree* (*snd a*) *s*) *PROB*
unfolding *snapshot-def Set.filter-def*
by *presburger*

— NOTE moved up.

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

lemma *proj-successors*:

$((\lambda s. fmrestrict\text{-set } vs \ s) \ ' (state\text{-successors } (prob\text{-proj } PROB \ vs) \ s))$
 $\subseteq (state\text{-successors } (prob\text{-proj } PROB \ vs) \ (fmrestrict\text{-set } vs \ s))$

proof –

let $?A = ((\lambda s. fmrestrict\text{-set } vs \ s) \ ' (state\text{-successors } (prob\text{-proj } PROB \ vs) \ s))$
let $?B = (state\text{-successors } (prob\text{-proj } PROB \ vs) \ (fmrestrict\text{-set } vs \ s))$

{

fix x

assume $P: x \in ?A$

then obtain $x' \ x''$ **where** a :

$x'' \in \text{prob-proj } PROB \ \text{vs} \ x' = \text{state-succ } s \ x'' \ x' \neq s \ x = fmrestrict\text{-set } vs \ x'$

unfolding *state-successors-def subset-iff*

by *blast*

moreover {

have $(\exists x'')$.

$x'' \in \text{prob-proj } PROB \ \text{vs} \wedge x = \text{state-succ } (fmrestrict\text{-set } vs \ s) \ x''$

$\wedge x \neq fmrestrict\text{-set } vs \ s)$

proof $(cases \ fst \ x'' \subseteq_f \ s)$

case *true: True*

then show *?thesis*

proof $(cases \ fst \ x'' \subseteq_f \ fmrestrict\text{-set } vs \ s)$

case *True*

{

have $fmdom' (snd \ x'') \subseteq vs$

using $a(1) \ \text{FDOM-eff-subset-prob-dom-pair } dom\text{-prob-proj } dual\text{-order}.\text{trans}$

by *metis*

then have $fmrestrict\text{-set } vs \ (snd \ x'') = snd \ x''$

using *exec-drest-5*

by *fast*

}

note $i = this$

{

have $x = fmrestrict\text{-set } vs \ (snd \ x'' \ ++ \ s)$

using $a(2, 4) \ \text{true}$

unfolding *state-succ-def*

by *simp*

then have $x = fmrestrict\text{-set } vs \ (snd \ x'') \ ++ \ fmrestrict\text{-set } vs \ s$

unfolding *fmap-add-ltr-def*

using *fmrestrict-set-add-distrib*

by *simp*

then have $x = snd \ x'' \ ++ \ fmrestrict\text{-set } vs \ s$

```

      using i
      by simp
    then have  $x = \text{state-succ } (\text{fmrestrict-set } vs \ s) \ x''$ 
      unfolding state-succ-def
      using True
      by argo
  }
  moreover have  $x \neq \text{fmrestrict-set } vs \ s$ 
    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  $x'' \in (\lambda p. \text{action-proj } p \ vs) \ \text{PROB}$ 
    using calculation(1) prob-proj-def
    by auto
  then have  $\text{action-proj } x'' \ vs = x''$ 
    using action-proj-idempot
    by blast
  then show ?thesis
    by (metis (no-types) False action-proj-pair fmsubset-restrict-set-mono
      surjective-pairing true)
qed
qed
next
case False
then show ?thesis
proof (cases  $\text{fst } x'' \subseteq_f \text{fmrestrict-set } vs \ s$ )
case True
then have  $\text{fmdom}' (\text{snd } x'') \subseteq vs$ 
  using FDOM-eff-subset-prob-dom-pair dom-prob-proj
  using a(1) dual-order.trans
  by metis
then have  $\text{fmrestrict-set } vs \ (\text{snd } x'') = \text{snd } x''$ 
  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  $\text{fmdom}' (\text{fst } x'') \subseteq vs$ 
  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
}
}
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*:
 $(s' \in \text{state-successors } (\text{prob-proj } \text{PROB } \text{vs}) s)$
 $\implies (\text{fmrestrict-set } \text{vs } s' \in \text{state-successors } (\text{prob-proj } \text{PROB } \text{vs}) (\text{fmrestrict-set } \text{vs } s))$
 using *proj-successors*
 by *force*

lemma *proj-FDOM-eff-subset-FDOM-valid-states*:
 fixes $p \ e \ s$
 assumes $((p, e) \in \text{prob-proj } \text{PROB } \text{vs}) (s \in \text{valid-states } \text{PROB})$
 shows $(\text{fmdom}' e \subseteq \text{fmdom}' s)$
 using *assms*
proof –
 {
 obtain $p' \ e'$ where $(p', e') \in \text{PROB } (p, e) = \text{action-proj } (p', e') \ \text{vs}$
 using *assms*(1)
 unfolding *prob-proj-def*
 by *fast*
 then have $\text{fmdom}' e \subseteq \text{prob-dom } (\text{prob-proj } \text{PROB } \text{vs})$
 using *assms* *FDOM-eff-subset-prob-dom*
 by *blast*
 also have $\dots = \text{prob-dom } \text{PROB} \cap \text{vs}$
 using *graph-plan-neq-mems-state-set-neq-len*
 by *fast*
 finally have $\text{fmdom}' e \subseteq \text{prob-dom } \text{PROB}$
 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  $(h \in prob-proj\ PROB\ vs) (s \in valid-states\ PROB)$ 
  shows  $(state-succ\ s\ h \in valid-states\ PROB)$ 
proof –
  have  $fmdom' (snd\ h) \subseteq fmdom' s$ 
    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  $(s \in valid-states\ PROB)$ 
  shows  $(state-successors\ (prob-proj\ PROB\ vs)\ s \subseteq (valid-states\ PROB))$ 
  unfolding  $state-successors-def$ 
  using  $assms\ valid-proj-action-valid-succ$ 
  by  $blast$ 

```

7.3 State Space Projection

```

definition  $ss-proj\ where$ 
   $ss-proj\ ss\ vs \equiv (\lambda s.\ fmrestrict-set\ vs\ s) \text{ ‘ } ss$ 

```

— NOTE added lemma.
 — TODO refactor into 'Fmap_Utills'.

```

lemma  $fmrestrict-set-inter-img$ :
  fixes  $A\ X\ Y$ 
  shows  $fmrestrict-set\ (X \cap Y) \text{ ‘ } A = (fmrestrict-set\ X \circ fmrestrict-set\ Y) \text{ ‘ } A$ 
proof –
  — NOTE Proof by mutual inclusion.
  let  $?lhs = fmrestrict-set\ (X \cap Y) \text{ ‘ } A$ 
  let  $?rhs = (fmrestrict-set\ X \circ fmrestrict-set\ Y) \text{ ‘ } A$ 
  {
    fix  $a$ 
    assume  $a \in A$ 
    have  $(fmrestrict-set\ X \circ fmrestrict-set\ Y)\ a = fmrestrict-set\ X\ (fmrestrict-set\ Y\ a)$ 

```

```

    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 ≠ {})
shows (ss-proj ss vs ≠ {})
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 ∩ 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 ⊆ vs)
shows (ss-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 ≠ {}) (stateSpace ss vs) (vs1 ⊆ vs) (vs2 ⊆ vs) (vs1 ≠ vs2)
shows (ss-proj ss vs1 ≠ ss-proj ss vs2)
proof –
{
  have 1: ∃f. f ∈ ss
    using assms(1)
    by blast
  then obtain x where (x ∈ vs1 ∧ x ∉ vs2) ∨ (x ∈ vs2 ∧ x ∉ vs1)
    using assms(5)
    by blast
  then consider (i) x ∈ vs1 ∧ x ∉ vs2 | (ii) x ∈ vs2 ∧ x ∉ vs1
    by blast
  then have fmrestrict-set vs1 ‘ ss ≠ fmrestrict-set vs2 ‘ ss proof (cases)
  case i
  {
    fix s' t'

```

```

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

```

```

lemma top-sorted-abs-mem:

```

```

assumes (top-sorted-abs R (h # l)) (ListMem x l)
shows ( $\neg$  R x h)
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  $\equiv$  ( $\forall$  x y. (x  $\leq$  f x y)  $\wedge$  (y  $\leq$  f x y))

```

```

lemma individual-weight-less-eq-lp:
fixes w :: 'a  $\Rightarrow$  nat
assumes geq-arg g
shows (w x  $\leq$  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.prem
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.prem(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.prem(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:  $\forall n \text{ na. } n \leq g \ n \ \text{na} \wedge \text{na} \leq g \ n \ \text{na}$ 
using Cons.prem(2) by blast
have f2:  $\text{vtx1} = a \vee \text{vtx1} \in \text{set } G$ 
by (meson Cons.prem(5) ListMem-iff set-ConsD)
obtain aa :: ('a  $\Rightarrow$  'a  $\Rightarrow$  bool)  $\Rightarrow$  'a where
 $\forall x2. (\exists v5. \text{ListMem } v5 \ G \wedge x2 \ v5 \ v5) = (\text{ListMem } (aa \ x2) \ G \wedge x2 \ (aa$ 
x2) (aa x2))
by moura
then have
ListMem (aa R) G  $\wedge$  R (aa R) (aa R)
 $\vee \neg \text{ListMem } \text{vtx1} \ G \vee f \ (w \ \text{vtx2}) \ (\text{wlp } R \ w \ g \ f \ \text{vtx1} \ G) \leq \text{wlp } R \ w \ g \ f$ 
vtx2 G
using f1 by (metis (no-types) Cons.IH Cons.prem(1, 4, 6) top-sorted-cons)
then show ?thesis
using f2 f1 by (meson Cons.prem(3) ListMem-iff insert le-trans)
qed
next
assume P4:  $\neg R \ \text{vtx1} \ a \ \neg R \ \text{vtx2} \ a$ 
then show  $f \ (w \ \text{vtx2}) \ (\text{wlp } R \ w \ g \ f \ \text{vtx1} \ G) \leq \text{wlp } R \ w \ g \ f \ \text{vtx2} \ G$ 
proof –
have f1: top-sorted-abs R G
using Cons.prem(6) by fastforce
have ListMem vtx1 G
by (metis Cons.prem(4) Cons.prem(5) ListMem-iff P4(2) set-ConsD)
then show ?thesis
using f1 by (simp add: Cons.IH Cons.prem(1, 2, 3, 4) insert)
qed
qed
qed

```

definition *increasing where*

increasing $f \equiv (\forall e \ b \ c \ d. (e \leq c) \wedge (b \leq d) \longrightarrow (f \ e \ b \leq f \ c \ d))$

lemma *weight-fun-leq-imp-lp-leq*: $\bigwedge x.$

(*increasing* f)
 \implies (*increasing* g)
 $\implies (\forall y. \text{ListMem } y \ l \longrightarrow w1 \ y \leq w2 \ y)$
 $\implies (w1 \ x \leq w2 \ x)$
 $\implies (\text{wlp } R \ w1 \ g \ f \ x \ l \leq \text{wlp } R \ w2 \ g \ f \ x \ l)$

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.prem1(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.prem1(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.prem2(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.prem1(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-lambda-exp:* $\bigwedge x. wlp R w g f x l = wlp R (\lambda y. w y) g f x l$

proof –

```

  fix x
  show wlp R w g f x l = wlp R (\lambda y. 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  $P1: \forall x. ListMem\ x\ l \longrightarrow P\ x$ 
  assume  $P2: \forall x. x \in s \longrightarrow P\ x$ 
  show (
     $(\lambda x. wlp\ R\ (\lambda y. if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l)\ 's$ 
     $= (\lambda x. wlp\ R\ w1\ g\ f\ x\ l)\ 's$ 
  )
  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  $S\ vs\ lss\ PROB\ s \equiv wlp$ 
     $(\lambda x\ y. y \in (state\ successors\ (prob\ proj\ PROB\ vs)\ x))$ 
     $(\lambda s. problem\ plan\ bound\ (snapshot\ PROB\ s))$ 
     $(max :: nat \Rightarrow nat \Rightarrow nat)\ (\lambda x\ y. x + y + 1)\ s\ lss$ 

```

— 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)  $\neq$  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 ($\text{vars-change } as \text{ vs } s = []$)
shows ($\text{fmrestrict-set } vs \text{ (exec-plan } s \text{ as)} = \text{fmrestrict-set } vs \text{ } s$)
using $assms$
proof (induction as arbitrary: s vs)
case ($\text{Cons } a \text{ as}$)
then show ?case
proof (cases $\text{fmrestrict-set } vs \text{ (state-succ } s \text{ a)} \neq \text{fmrestrict-set } vs \text{ } s$)
case True
— NOTE This case violates the induction premise $\text{vars-change } (a \# as) \text{ vs } s = []$ since the empty list is impossible.
then have $\text{state-succ } s \text{ a } \# \text{ vars-change } as \text{ vs (state-succ } s \text{ a)} = []$
using $\text{Cons.prem } \text{True}$
by simp
then show $\text{fmrestrict-set } vs \text{ (exec-plan } s \text{ (a } \# as)) = \text{fmrestrict-set } vs \text{ } s$
by blast
next
case False
then have $\text{vars-change } as \text{ vs (state-succ } s \text{ a)} = []$
using $\text{Cons.prem } \text{False}$
by force
then have
 $\text{fmrestrict-set } vs \text{ (exec-plan (state-succ } s \text{ a) as)} = \text{fmrestrict-set } vs \text{ (state-succ } s \text{ a)}$
using $\text{Cons.IH[of } vs \text{ (state-succ } s \text{ a)]}$
by blast
then show $\text{fmrestrict-set } vs \text{ (exec-plan } s \text{ (a } \# as)) = \text{fmrestrict-set } vs \text{ } s$
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 $s \text{ } s'$
assumes ($\text{vars-change } as \text{ vs } s = []$) ($\text{sat-precond-as } s \text{ as}$) ($\text{ListMem } b \text{ as}$)
($\text{fmrestrict-set } vs \text{ } s = \text{fmrestrict-set } vs \text{ } s'$)
shows ($\text{fmrestrict-set } vs \text{ (snd } b) \subseteq_f \text{fmrestrict-set } vs \text{ } s'$)
using $assms$
proof (induction as arbitrary: s s' vs b)
case ($\text{Cons } a \text{ as}$)
— NOTE Having either $\text{fmrestrict-set } vs \text{ (state-succ } s \text{ a)} \neq \text{fmrestrict-set } vs \text{ } s$

or $\neg \text{ListMem } b \text{ as}$ leads to simpler propositions so we split here.

```

then show (fmrestrict-set vs (snd b)  $\subseteq_f$  fmrestrict-set vs s')
  using Cons.prem(1)
proof (cases fmrestrict-set vs (state-succ s a) = fmrestrict-set vs s  $\wedge$  ListMem b
as)
  case True
  let ?s=state-succ s a
  have vars-change as vs ?s = []
    using True Cons.prem(1)
    by auto
  moreover have sat-precond-as ?s as
    using Cons.prem(2) sat-precond-as.simp(2)
    by blast
  ultimately show ?thesis
    using True Cons.prem(4) Cons.IH
    by auto
next
case False
then consider
  (i) fmrestrict-set vs (state-succ s a)  $\neq$  fmrestrict-set vs s
  | (ii)  $\neg$ ListMem b as
  by blast
then show ?thesis
  using Cons.prem(1)
proof (cases)
  case ii
  then have a = b
    using Cons.prem(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.prem(1, 2, 4) no-change-vs-eff-submap
    by (metis list.distinct(1) sat-precond-as.simp(2) vars-change.simp(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)  $\neq$  fmrestrict-set vs s
or  $\neg \text{ListMem } b \text{ as}$  leads to simpler propositions so we split here.

```

```

then show (fmrestrict-set vs (fst b)  $\subseteq_f$  fmrestrict-set vs s')
  using Cons.prem(1)
proof (cases fmrestrict-set vs (state-succ s a) = fmrestrict-set vs s  $\wedge$  ListMem b
as)
  case True
  let ?s=state-succ s a
  have vars-change as vs ?s = []
    using True Cons.prem(1)
    by auto
  moreover have sat-precond-as ?s as
    using Cons.prem(2) sat-precond-as.simp(2)
    by blast
  ultimately show ?thesis
    using True Cons.prem(4) Cons.IH
    by auto
next
  case False
  then consider
    (i) fmrestrict-set vs (state-succ s a)  $\neq$  fmrestrict-set vs s
    | (ii)  $\neg$ ListMem b as
    by blast
  then show ?thesis
    using Cons.prem(1)
  proof (cases)
    case ii
    then have a = b
      using Cons.prem(3) ListMem-iff set-ConsD
      by metis
    then show ?thesis
      using Cons.prem(2, 4) fmsubset-restrict-set-mono
      by (metis sat-precond-as.simp(2))
    qed simp
  qed
qed (simp add: ListMem-iff)

```

lemma no-vs-change-valid-in-snapshot:

```

assumes (as  $\in$  valid-plans PROB) (sat-precond-as s as) (vars-change as vs s =
[])
shows (as  $\in$  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 $(\exists as'$.

$($
 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'
 $)$
 \wedge *(subseq as' as)*
 \wedge *(sat-precond-as s as')*
 \wedge *(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 $length\ as'' \leq$ *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 ∈ valid-states PROB*) (*as ∈ 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' ≤ problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))
)
proof –
obtain *as' :: ('a, bool) fmap × ('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' ≤ 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 ∈ valid-states PROB*) (*as ∈ valid-plans PROB*)
fmrestrict-set (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as*) =

```

      fmrestrict-set (prob-dom (snapshot PROB (fmrestrict-set vs s))) (exec-plan
s as')
      subseq as' as length as' ≤ problem-plan-bound (snapshot PROB (fmrestrict-set
vs s))
shows
      fmrestrict-set (fndom' (exec-plan s as) – prob-dom (snapshot PROB (fmrestrict-set
vs s)))
        (exec-plan s as)
        = fmrestrict-set (fndom' (exec-plan s as) – prob-dom (snapshot PROB
(fmrestrict-set vs s)))
          s
      ∧ fmrestrict-set (fndom' (exec-plan s as') – prob-dom (snapshot PROB (fmrestrict-set
vs s)))
        (exec-plan s as')
        = fmrestrict-set (fndom' (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'=(fndom' (exec-plan s as) – prob-dom (snapshot PROB (fmrestrict-set
vs s)))
  let ?vs''=(fndom' (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 ∈ 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 fndom' (snd a) ⊆ prob-dom (snapshot PROB (fmrestrict-set vs
s))
    using 1 FDOM-eff-subset-prob-dom-pair valid-plan-mems
    by metis
  then have fndom' (fmrestrict-set (fndom' (exec-plan s as)
– prob-dom (snapshot PROB (fmrestrict-set vs s))) (snd a))
    = {}
    using subset-inter-diff-empty[of fndom' (snd a)
prob-dom (snapshot PROB (fmrestrict-set vs s))] fndom'-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  $\alpha$ : as'  $\in$  valid-plans (snapshot PROB (fmrestrict-set vs s))
    using assms(8) 1 sublist-valid-plan
    by blast
  moreover have a  $\in$  PROB
    using P  $\alpha$  snapshot-subset subsetCE valid-plan-mems
    by fast
  ultimately have fndom' (snd a)  $\subseteq$  prob-dom (snapshot PROB (fmrestrict-set
vs s))
    using FDOM-eff-subset-prob-dom-pair valid-plan-mems
    by metis
  then have fndom' (fmrestrict-set (fndom' (exec-plan s as')
    - prob-dom (snapshot PROB (fmrestrict-set vs s))) (snd a))
    = {}
    using subset-inter-diff-empty[of fndom' (snd a)
    prob-dom (snapshot PROB (fmrestrict-set vs s))] fndom'-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 \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 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' \leq 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 (fndom' (exec-plan s as) - prob-dom (snapshot PROB
(fmrestrict-set vs s)))
    (exec-plan s as)
    = fmrestrict-set (fndom' (exec-plan s as) - prob-dom (snapshot PROB
(fmrestrict-set vs s)))
      s
    fmrestrict-set (fndom' (exec-plan s as') - prob-dom (snapshot PROB (fmrestrict-set
vs s)))
      (exec-plan s as')
    = fmrestrict-set (fndom' (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 sub-
list-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  $\forall x. ListMem\ x\ lss \longrightarrow \neg ?R\ x\ x$ 
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.premis
      by simp+
    then show ?thesis
      by fastforce
  next
    case False
    then have vars-change as vs (state-succ s a) = s' # ss
      using Cons.premis
      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

```

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 ($\exists as'$.
(exec-plan s as' = exec-plan s as)
 \wedge (*subseq as' as*)
 \wedge (*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.premis(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))
 \leq *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.prem5(5) 1(1) valid-append-valid-pref
      by blast
    moreover have act ∈ PROB
      using Cons.prem5(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.prem4(4) valid-as-valid-exec lemma-1-i
      by blast
  }
  moreover have as2 ∈ valid-plans PROB
    using Cons.prem5(5) 1(1) valid-append-valid-suff valid-plan-valid-tail
    by fast
  moreover have no-effectless-act as2
    using Cons.prem6(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.prem7(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.prem1(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.prem6(6) 1(1) rem-effectless-works-12
      by blast
    moreover have sat-precond-as s as1
      using Cons.prem7(7) 1(1) sat-precond-as-pfx
      by blast
    moreover have as1 ∈ valid-plans PROB
      using Cons.prem5(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.premis(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.premis(7)
          by blast
        then have fst act ⊆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.prem5(5) 1 valid-append-valid-suff valid-plan-valid-head
      by fast
    then have  $\aleph$ : 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) ⊆f s)
        case True
          then show ?thesis
            unfolding state-successors-def
            using Cons.hyps(2) 1(3) b(1) A B C  $\aleph$  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  $\aleph$  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.prem5(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 ( $\lambda x y. y \in \text{state-successors (prob-proj PROB vs) } x$ ) lss)
  (set lss = valid-states (prob-proj PROB vs)) (s  $\in$  valid-states PROB)
  (as  $\in$  valid-plans PROB)
shows ( $\exists as'$ .
  (exec-plan s as' = exec-plan s as)
   $\wedge$  (subseq as' as)
   $\wedge$  (length as'  $\leq$  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  $\sqcap$  (rem-effectless-act as)  $\in$  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  $\sqcap$  (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  $\sqcap$  (rem-effectless-act as))
    using assms rem-condless-valid-2
    by blast
  then have  $\exists as'$ .
    exec-plan s as' = exec-plan s (rem-condless-act s  $\sqcap$  (rem-effectless-act as))
     $\wedge$  subseq as' (rem-condless-act s  $\sqcap$  (rem-effectless-act as))
     $\wedge$  length as'  $\leq$  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  $\sqcap$  (rem-effectless-act as))
  subseq as' (rem-condless-act s  $\sqcap$  (rem-effectless-act as))
  length as'  $\leq$  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 ( $\lambda x y. y \in \text{state-successors} (\text{prob-proj } \text{PROB } vs) x$ ) lss)
  (set lss = valid-states (prob-proj PROB vs)) (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'  $\leq$  Sup {S vs lss PROB s' | s'. s'  $\in$  valid-states (prob-proj PROB
vs)}))
)

```

proof –

obtain as' where

```

  exec-plan s as' = exec-plan s as subseq as' as
  length as'  $\leq$  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' \in 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'  $\in$  valid-states (prob-proj PROB vs)}
{
  have fmrestrict-set vs s  $\in$  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)  $\in$  ?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  $PROB$  ( $fmrestrict\text{-}set$  vs  $s$ )  $\leq$   $Sup$  ? $S$ 
      using  $le\text{-}cSup\text{-}finite$  by blast
  }
  ultimately show ?thesis
    using  $le\text{-}trans$ 
    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  $PROB$  ( $top\text{-}sorted\text{-}abs$  ( $\lambda x y. y \in state\text{-}successors$  ( $prob\text{-}proj$ 
 $PROB$  vs)  $x$ ) lss)
    ( $set$  lss =  $valid\text{-}states$  ( $prob\text{-}proj$   $PROB$  vs))
  shows
     $problem\text{-}plan\text{-}bound$   $PROB$ 
     $\leq$   $Sup$  { $S$  vs lss  $PROB$  ( $s' :: 'a$  state) |  $s'. s' \in valid\text{-}states$  ( $prob\text{-}proj$   $PROB$ 
vs)}

```

proof –

```

  let ? $f$ = $\lambda$  $PROB$ .
     $Sup$  { $S$  vs lss  $PROB$  ( $s' :: 'a$  state) |  $s'. s' \in valid\text{-}states$  ( $prob\text{-}proj$   $PROB$  vs)}
+ 1
  {
    fix  $as$  and  $s :: 'a$  state
    assume  $s \in valid\text{-}states$   $PROB$   $as \in valid\text{-}plans$   $PROB$ 
    then obtain  $as'$  where  $a$ :
       $exec\text{-}plan$   $s$   $as' = exec\text{-}plan$   $s$  as subseq  $as'$  as
       $length$   $as' \leq Sup$  { $S$  vs lss  $PROB$   $s' | s'. s' \in valid\text{-}states$  ( $prob\text{-}proj$   $PROB$  vs)}
      using  $assms$   $S\text{-in}\text{-}MPLS\text{-}leq\text{-}2\text{-}pow\text{-}n$ 
      by blast
    then have  $length$   $as' < ?f$   $PROB$ 
      by  $linarith$ 
    moreover have  $exec\text{-}plan$   $s$   $as = exec\text{-}plan$   $s$   $as'$ 
      using  $a(1)$ 
      by  $simp$ 
    ultimately have
       $\exists as'. exec\text{-}plan$   $s$   $as = exec\text{-}plan$   $s$   $as' \wedge subseq$   $as'$   $as \wedge length$   $as' < ?f$   $PROB$ 
      using  $a(2)$ 
      by blast
  }
  then show ?thesis
    using  $assms(1)$   $problem\text{-}plan\text{-}bound\text{-}UBound$ [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_the-sis’). [Abdulaziz et al., p.29]

definition *sspace-DAG* where

$$\begin{aligned} & \textit{sspace-DAG} \textit{ PROB lss} \equiv (\\ & \quad (\textit{set lss} = \textit{valid-states} \textit{ PROB}) \\ & \quad \wedge (\textit{top-sorted-abs} (\lambda x y. y \in \textit{state-successors} \textit{ PROB} x) \textit{lss}) \\ &) \end{aligned}$$

lemma *problem-plan-bound-S-bound-2nd-step-thesis*:

assumes *finite* (*PROB* :: 'a *problem*) (*sspace-DAG* (*prob-proj* *PROB* *vs*) *lss*)
(s ∈ *valid-states* *PROB*) (*as* ∈ *valid-plans* *PROB*)

shows (\exists *as'*. (*exec-plan* *s as'* = *exec-plan* *s as*)

\wedge (*subseq* *as'* *as*)

\wedge (*length* *as'* ≤ *S vs lss PROB (fmrestrict-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-DAG* (*prob-proj* *PROB* *vs*) *lss*)

shows (

problem-plan-bound *PROB*

≤ *Sup* {*S vs lss PROB s'* | *s'*. *s'* ∈ *valid-states* (*prob-proj* *PROB* *vs*)}

)

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.