

Upper Bounding Diameters of State Spaces of Factored Transition Systems

Friedrich Kurz and Mohammad Abdulaziz

February 6, 2026

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	6
2.1	Sublist Characterization	6
2.2	Main Theorems	11
3	Factored System	14
3.1	Semantics of Plan Execution	14
3.1.1	Characterization of the Set of Possible States	14

3.1.2	State Lists and State Sets	17
3.1.3	Properties of Domain Changes During Plan Execution	18
3.1.4	Properties of Valid Plans	20
3.2	Reachable States	24
3.3	State Spaces	30
3.4	Needed Asses	30
4	Action Sequence Process	34
5	Dependency	44
5.1	Dependent Variables and Variable Sets	44
5.2	Transitive Closure of Dependent Variables and Variable Sets	46
5.3	Sets of Numbers	47
6	Topological Properties	49
6.1	Basic Definitions and Properties	49
6.2	Recurrence Diameter	50
6.3	The Relation between Diameter, Sublist Diameter and Recurrence Diameter Bounds.	62
6.4	Traversal Diameter	64
7	System Abstraction	65
7.1	Projection of Actions, Sequences of Actions and Factored Representations.	66
7.2	Snapshotting	77
7.3	State Space Projection	82
8	Acyclicity	83
8.1	Topological Sorting of Dependency Graphs	83
8.2	The Weightiest Path Function (wlp)	84
9	Acyclic State Spaces	86
9.1	State Space Acyclicity	90

```

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

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

definition valid-states where

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

definition valid-plans where

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

definition state-succ where

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 *length l* ≥ 0

<proof>

lemma len-gt-pref-is-pref:

fixes *l l1 l2*

assumes (*length l2* $>$ *length l1*) (*prefix l1 l*) (*prefix l2 l*)

shows (*prefix l1 l2*)

<proof>

lemma nempty-list-append-length-add:

fixes *l1 l2 l3*

assumes $l2 \neq []$
shows $\text{length } (l1 @ l3) < \text{length } (l1 @ l2 @ l3)$
 $\langle \text{proof} \rangle$

lemma *append-filter*:

fixes $f1 :: 'a \Rightarrow \text{bool}$ **and** $f2 \text{ as1 as2 and } p :: 'a \text{ list}$
assumes $(\text{as1} @ \text{as2} = \text{filter } f1 (\text{map } f2 \text{ p}))$
shows $(\exists p-1 \text{ p-2.}$
 $(p-1 @ p-2 = p)$
 $\wedge (\text{as1} = \text{filter } f1 (\text{map } f2 \text{ p-1}))$
 $\wedge (\text{as2} = \text{filter } f1 (\text{map } f2 \text{ p-2}))$
 $)$
 $\langle \text{proof} \rangle$

lemma *append-eq-as-proj-1*:

fixes $f1 :: 'a \Rightarrow \text{bool}$ **and** $f2 \text{ as1 as2 as3 and } p :: 'a \text{ list}$
assumes $(\text{as1} @ \text{as2} @ \text{as3} = \text{filter } f1 (\text{map } f2 \text{ p}))$
shows $(\exists p-1 \text{ p-2 p-3.}$
 $(p-1 @ p-2 @ p-3 = p)$
 $\wedge (\text{as1} = \text{filter } f1 (\text{map } f2 \text{ p-1}))$
 $\wedge (\text{as2} = \text{filter } f1 (\text{map } f2 \text{ p-2}))$
 $\wedge (\text{as3} = \text{filter } f1 (\text{map } f2 \text{ p-3}))$
 $)$
 $\langle \text{proof} \rangle$

lemma *filter-empty-every-not*: $\bigwedge P \text{ l. } (\text{filter } (\lambda x. P \text{ x}) \text{ l} = []) = \text{list-all } (\lambda x. \neg P \text{ x}) \text{ l}$

$\langle \text{proof} \rangle$

lemma *MEM-SPLIT*:

fixes $x \text{ l}$
assumes $\neg \text{ListMem } x \text{ l}$
shows $\forall l1 \text{ l2. } l \neq l1 @ [x] @ l2$

$\langle \text{proof} \rangle$

lemma *APPEND-EQ-APPEND-MID*:

fixes $l1 \text{ l2 m1 m2 e}$
shows
 $(l1 @ [e] @ l2 = m1 @ m2)$
 \longleftrightarrow
 $(\exists l. (m1 = l1 @ [e] @ l) \wedge (l2 = l @ m2)) \vee$
 $(\exists l. (l1 = m1 @ l) \wedge (m2 = l @ [e] @ l2))$

$\langle \text{proof} \rangle$

lemma *LIST-FRAG-DICHOTOMY*:

fixes $l \text{ la } x \text{ lb}$
assumes $\text{sublist } l \text{ (la @ [x] @ lb)} \neg \text{ListMem } x \text{ l}$
shows $\text{sublist } l \text{ la} \vee \text{sublist } l \text{ lb}$

$\langle \text{proof} \rangle$

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>

```

```

lemma frag-len-filter-le:
fixes P l' l
assumes sublist l' l
shows length (filter P l')  $\leq$  length (filter P l)
<proof>

```

end

```

theory FSSublist
imports Main HOL-Library.Sublist ListUtils
begin

```

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

2 Factored System Sublist

2.1 Sublist Characterization

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

In HOL4 ‘sublist’ is defined as

```

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

```

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

```

subseq xs ys  $\equiv$  subseq xs ys

```

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

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

```

fun sublist-HOL4 where
  sublist-HOL4 [] l1 = True
| (sublist-HOL4 (h # t) [] = False)
| (sublist-HOL4 (x # l1) (y # l2) = ((x = y)  $\wedge$  sublist-HOL4 l1 l2  $\vee$  sublist-HOL4 (x # l1) l2))

```

— NOTE added lemma

lemma *sublist-HOL4-equiv-subseq*:

fixes $l1\ l2$

shows $sublist\text{-}HOL4\ l1\ l2 \longleftrightarrow subseq\ l1\ l2$

<proof>

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**

$list_frag\text{-}HOL4\ l\ frag \equiv \exists\ pfx\ sfx.\ pfx\ @\ frag\ @\ sfx = l$

lemma *list_frag-HOL4-equiv-sublist*:

shows $list_frag\text{-}HOL4\ l\ l' \longleftrightarrow sublist\ l'\ l$

<proof>

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*:

$subseq\ []\ l = True$

$subseq\ (h\ \#\ t)\ [] = False$

<proof>

lemma *sublist-refl*: $subseq\ l\ l$

<proof>

lemma *sublist-cons*:

assumes $subseq\ l1\ l2$

shows $subseq\ l1\ (h\ \#\ l2)$

<proof>

lemma *sublist-NIL*: $subseq\ l1\ [] = (l1 = [])$

<proof>

lemma *sublist-trans*:

fixes $l1\ l2$

assumes $subseq\ l1\ l2\ subseq\ l2\ l3$

shows *subseq l1 l3*
 ⟨*proof*⟩

lemma *sublist-length*:
fixes *l l'*
assumes *subseq l l'*
shows $\text{length } l \leq \text{length } l'$
 ⟨*proof*⟩

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

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

lemma *sublist-antisym*:
assumes *subseq l1 l2 subseq l2 l1*
shows $(l1 = l2)$
 ⟨*proof*⟩

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

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

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

lemma *append-sublist-1*:
assumes *subseq (l1 @ l2) l*
shows $\text{subseq } l1 \ l \wedge \text{subseq } l2 \ l$
 ⟨*proof*⟩

lemma *sublist-prefix*:
shows $\text{subseq } (h \# l1) \ l2 \implies \exists l2a \ l2b. \ l2 = l2a @ [h] @ l2b \wedge \neg \text{ListMem } h \ l2a$
 ⟨*proof*⟩

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

<proof>

lemma *sublist-split-trans*:

fixes $l1\ l2\ h\ l1'$

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

shows $\text{subseq } l1'\ l2b$

<proof>

lemma *sublist-cons-exists*:

shows

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

$\longleftrightarrow (\exists l2a\ l2b. (l2 = l2a @ [h] @ l2b) \wedge \neg \text{ListMem } h\ l2a \wedge \text{subseq } l1\ l2b)$

<proof>

lemma *sublist-append-exists*:

fixes $l1\ l2$

shows $\text{subseq } (l1 @ l2)\ l3 \implies \exists l3a\ l3b. (l3 = l3a @ l3b) \wedge \text{subseq } l1\ l3a \wedge \text{subseq } l2\ l3b$

<proof>

lemma *sublist-append-both-I*:

assumes $\text{subseq } a\ b\ \text{subseq } c\ d$

shows $\text{subseq } (a @ c)\ (b @ d)$

<proof>

lemma *sublist-append*:

assumes $\text{subseq } l1\ l1'\ \text{subseq } l2\ l2'$

shows $\text{subseq } (l1 @ l2)\ (l1' @ l2')$

<proof>

lemma *sublist-append2*:

assumes $\text{subseq } l1\ l2$

shows $\text{subseq } l1\ (l2 @ l3)$

<proof>

lemma *append-sublist*:

shows $\text{subseq } (l1 @ l2 @ l3)\ l \implies \text{subseq } (l1 @ l3)\ l$

<proof>

lemma *sublist-subset*:

assumes $\text{subseq } l1\ l2$

shows $set\ l1 \subseteq set\ l2$
<proof>

lemma *sublist-filter*:
fixes $P\ l$
shows $subseq\ (filter\ P\ l)\ l$
<proof>

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

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

lemma *sublist-SING-MEM*: $subseq\ [h]\ l \longleftrightarrow ListMem\ h\ l$
<proof>

lemma *sublist-append-exists-2*:
fixes $l1\ l2\ l3$
assumes $subseq\ (h\ \#\ l1)\ l2$
shows $(\exists\ l3\ l4. (l2 = l3\ @\ [h]\ @\ l4) \wedge (subseq\ l1\ l4))$
<proof>

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

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

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

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

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

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

2.2 Main Theorems

theorem *sublist-imp-len-filter-le*:
fixes *P l l'*
assumes *subseq l' l*
shows *length* (*filter P l'*) \leq *length* (*filter P l*)
 ⟨*proof*⟩

theorem *list-with-three-types-shorten-type2*:
fixes *P1 P2 P3 k1 f PProbs PProbl s l*
assumes (*PProbs s*) (*PProbl l*)
 ($\forall l s.$
 (*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*)

)
<proof>

lemma *isPREFIX-sublist*:

fixes $x\ y$
assumes $prefix\ x\ y$
shows $subseq\ x\ y$
<proof>

end

theory *HoArithUtils*

imports *Main*

begin

lemma *general-theorem*:

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

end

theory *FmapUtils*

imports *HOL-Library.Finite-Map FactoredSystemLib*

begin

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

— NOTE hide the '++' operator from 'Map' to prevent warnings.

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

— TODO more explicit proof.

lemma *IN-FDOM-DRESTRICT-DIFF*:

fixes $vs\ v\ f$
assumes $\neg(v \in vs) \wedge fdom'\ f \subseteq fdom\ v \in fdom'\ f$
shows $v \in fdom'\ (fmrestrict_set\ (fdom - vs)\ f)$
<proof>

lemma *disj-dom-drest-fupdate-eq*:

$disjnt\ (fdom'\ x)\ vs \implies (fmrestrict_set\ vs\ s = fmrestrict_set\ vs\ (x\ ++\ s))$

<proof>

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)

lemma *exec-drest-5*:

fixes $x \ vs$
assumes $\text{fmdom}' \ x \subseteq \ vs$
shows $(\text{fmrestrict-set } vs \ x = x)$
(proof)

lemma *graph-plan-lemma-5*:

fixes $s \ s' \ vs$
assumes $(\text{fmrestrict-set} (\text{fmdom}' \ s - \ vs) \ s = \text{fmrestrict-set} (\text{fmdom}' \ s' - \ vs) \ s')$
 $(\text{fmrestrict-set } vs \ s = \text{fmrestrict-set } vs \ s')$
shows $(s = s')$
(proof)

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

fixes $x \ s \ vs$
shows $\text{fmrestrict-set } vs \ x \subseteq_f \ s \iff \text{fmrestrict-set } vs \ x \subseteq_f \ \text{fmrestrict-set } vs \ s$
(proof)

lemma *sat-precond-as-proj-1*:

fixes $s \ s' \ vs \ x$
assumes $\text{fmrestrict-set } vs \ s = \text{fmrestrict-set } vs \ s'$
shows $\text{fmrestrict-set } vs \ x \subseteq_f \ s \iff \text{fmrestrict-set } vs \ x \subseteq_f \ s'$
(proof)

lemma *sat-precond-as-proj-4*:

fixes $\text{fm1} \ \text{fm2} \ vs$
assumes $\text{fm2} \subseteq_f \ \text{fm1}$
shows $(\text{fmrestrict-set } vs \ \text{fm2} \subseteq_f \ \text{fm1})$
(proof)

lemma *sublist-as-proj-eq-as-1*:

fixes $x \ s \ vs$
assumes $(x \subseteq_f \ \text{fmrestrict-set } vs \ s)$
shows $(x \subseteq_f \ s)$
(proof)

lemma *limited-dom-neq-restricted-neq*:

assumes $\text{fmdom}' \ f1 \subseteq \ vs \ f1 \ ++ \ f2 \neq \ f2$
shows $\text{fmrestrict-set } vs \ (f1 \ ++ \ f2) \neq \ \text{fmrestrict-set } vs \ f2$
(proof)

lemma *fmlookup-fmrestrict-set-dom*: $\bigwedge \ vs \ s. \ \text{dom} (\text{fmlookup} (\text{fmrestrict-set } vs \ s))$
 $= \ vs \cap (\text{fmdom}' \ s)$
(proof)

end

```

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

```

3 Factored System

```

hide-const (open) Map.map-add
no-notation Map.map-add (infixl <++> 100)

```

3.1 Semantics of Plan Execution

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

The semantics of action execution were previously introduced via the notion of 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*: $state_succ\ s\ (p,\ e) = (if\ (p \subseteq_f\ s)\ then\ (e\ ++\ s)\ else\ s)$
<proof>

```

fun exec-plan where
  exec-plan s [] = s
| exec-plan s (a # as) = exec-plan (state-succ s a) as

```

lemma *exec-plan-Append*:

```

fixes as-a as-b s
shows exec-plan s (as-a @ as-b) = exec-plan (exec-plan s as-a) as-b
<proof>

```

Plan execution effectively eliminates cycles: i.e., if a given plan ‘as’ may be partitioned into plans ‘as1’, ‘as2’ and ‘as3’, s.t. the sequential execution of ‘as1’ and ‘as2’ yields the same state, ‘as2’ may be skipped during plan execution.

lemma *cycle-removal-lemma*:

```

fixes as1 as2 as3
assumes (exec-plan s (as1 @ as2) = exec-plan s as1)
shows (exec-plan s (as1 @ as2 @ as3) = exec-plan s (as1 @ as3))
<proof>

```

3.1.1 Characterization of the Set of Possible States

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

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

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

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⟩

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

fixes $s \ x \ v$

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

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

⟨proof⟩

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

fixes $s \ x \ v$

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

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

⟨proof⟩

lemma *fmap-neq*:

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

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

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

⟨proof⟩

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

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

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

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

⟨proof⟩

lemma *fmsubset-restrict-set*:

fixes $X1 \ X2$ **and** $s :: 'a \text{ state}$

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

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

⟨proof⟩

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⟩

lemma *construction-of-all-possible-states-lemma*:

fixes $v \ X$

assumes $(v \notin X)$

shows $(\{s. \text{fmdom}' s = \text{insert } v \ X\}$

$= ((\lambda s. \text{fmupd } v \ \text{True } s) \ \{s. \text{fmdom}' s = X\})$

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

)

<proof>

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

As lemma ‘*card_of_set_of_all_possible_states*’ shows, for a finite variable set ‘*X*’, the number of possible states is ‘ $2^{\text{card } X}$ ’, 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 $X :: 'a \text{ set}$

shows $\text{finite } X \implies \text{finite } \{(s :: 'a \text{ state}). \text{fmdom}' s = X\}$

<proof>

lemma *bool-update-effect*:

fixes $s X x v b$

assumes $\text{finite } X \ s \in \{s :: 'a \text{ state}. \text{fmdom}' s = X\} \ x \in X \ x \neq v$

shows $\text{fmlookup } ((\lambda s :: 'a \text{ state}. \text{fmupd } v \ b \ s) \ s) \ x = \text{fmlookup } s \ x$

<proof>

lemma *bool-update-inj*:

fixes $X :: 'a \text{ set}$ **and** $v b$

assumes $\text{finite } X \ v \notin X$

shows $\text{inj-on } (\lambda s. \text{fmupd } v \ b \ s) \ \{s :: 'a \text{ state}. \text{fmdom}' s = X\}$

<proof>

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>

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>

lemma *card-of-set-of-all-possible-states*:

fixes $X :: 'a \text{ set}$

assumes $\text{finite } X$

shows $\text{card } \{(s :: 'a \text{ state}). \text{fmdom}' s = X\} = 2^{\text{card } X}$

<proof>

3.1.2 State Lists and State Sets

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

lemma empty-state-list-lemma:

```
  fixes as s  
  shows  $\neg([] = \text{state-list } s \text{ as})$   
<proof>
```

lemma state-list-length-non-zero:

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

lemma state-list-length-lemma:

```
  fixes as s  
  shows  $\text{length } as = \text{length } (\text{state-list } s \text{ as}) - 1$   
<proof>
```

lemma state-list-length-lemma-2:

```
  fixes as s  
  shows  $(\text{length } (\text{state-list } s \text{ as})) = (\text{length } as + 1)$   
<proof>
```

fun state-set **where**

```
  state-set [] = {}  
| state-set (s # ss) = insert [s] (Cons s ' (state-set ss))
```

lemma state-set-thm:

```
  fixes s1  
  shows  $s1 \in \text{state-set } s2 \longleftrightarrow \text{prefix } s1 \ s2 \wedge s1 \neq []$   
<proof>
```

lemma state-set-finite:

```
  fixes X  
  shows finite (state-set X)  
<proof>
```

lemma LENGTH-state-set:

fixes $X e$
assumes $e \in \text{state-set } X$
shows $\text{length } e \leq \text{length } X$
 $\langle \text{proof} \rangle$

lemma *lemma-temp*:
fixes $x s \text{ as } h$
assumes $x \in \text{state-set } (\text{state-list } s \text{ as})$
shows $\text{length } (h \# \text{state-list } s \text{ as}) > \text{length } x$
 $\langle \text{proof} \rangle$

lemma *NIL-NOTIN-stateset*:

fixes X
shows $[] \notin \text{state-set } X$
 $\langle \text{proof} \rangle$

lemma *state-set-card-i*:

fixes $X a$
shows $[a] \notin (\text{Cons } a \text{ ' state-set } X)$
 $\langle \text{proof} \rangle$

lemma *state-set-card-ii*:

fixes $X a$
shows $\text{card } (\text{Cons } a \text{ ' state-set } X) = \text{card } (\text{state-set } X)$
 $\langle \text{proof} \rangle$

lemma *state-set-card-iii*:

fixes $X a$
shows $\text{card } (\text{state-set } (a \# X)) = 1 + \text{card } (\text{state-set } X)$
 $\langle \text{proof} \rangle$

lemma *state-set-card*:

fixes X
shows $\text{card } (\text{state-set } X) = \text{length } X$
 $\langle \text{proof} \rangle$

3.1.3 Properties of Domain Changes During Plan Execution

lemma *FDOM-state-succ*:

assumes $\text{fndom}' (\text{snd } a) \subseteq \text{fndom}' s$
shows $\text{fndom}' (\text{state-succ } s a) = \text{fndom}' s$
 $\langle \text{proof} \rangle$

lemma *FDOM-state-succ-subset*:

$\text{fndom}' (\text{state-succ } s a) \subseteq (\text{fndom}' s \cup \text{fndom}' (\text{snd } a))$
 $\langle \text{proof} \rangle$

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

fixes $p\ e\ s$
assumes $(p, e) \in PROB\ (s \in \text{valid-states } PROB)$
shows $(fndom'\ e \subseteq fndom'\ s)$
 $\langle proof \rangle$

lemma *FDOM-eff-subset-FDOM-valid-states-pair*:
fixes $a\ s$
assumes $a \in PROB\ s \in \text{valid-states } PROB$
shows $fndom'\ (snd\ a) \subseteq fndom'\ s$
 $\langle proof \rangle$

lemma *FDOM-pre-subset-FDOM-valid-states*:
fixes $p\ e\ s$
assumes $(p, e) \in PROB\ s \in \text{valid-states } PROB$
shows $fndom'\ p \subseteq fndom'\ s$
 $\langle proof \rangle$

lemma *FDOM-pre-subset-FDOM-valid-states-pair*:
fixes $a\ s$
assumes $a \in PROB\ s \in \text{valid-states } PROB$
shows $fndom'\ (fst\ a) \subseteq fndom'\ s$
 $\langle proof \rangle$

lemma *action-dom-subset-valid-states-FDOM*:
fixes $p\ e\ s$
assumes $(p, e) \in PROB\ s \in \text{valid-states } PROB$
shows $\text{action-dom } p\ e \subseteq fndom'\ s$
 $\langle proof \rangle$

lemma *FDOM-eff-subset-prob-dom*:
fixes $p\ e$
assumes $(p, e) \in PROB$
shows $fndom'\ e \subseteq \text{prob-dom } PROB$
 $\langle proof \rangle$

lemma *FDOM-eff-subset-prob-dom-pair*:
fixes a
assumes $a \in PROB$
shows $fndom'\ (snd\ a) \subseteq \text{prob-dom } PROB$
 $\langle proof \rangle$

lemma *FDOM-pre-subset-prob-dom*:
fixes $p\ e$
assumes $(p, e) \in PROB$
shows $fndom'\ p \subseteq \text{prob-dom } PROB$
 $\langle proof \rangle$

lemma *FDOM-pre-subset-prob-dom-pair*:
fixes a
assumes $a \in PROB$
shows $fmdom' (fst a) \subseteq prob-dom\ PROB$
 $\langle proof \rangle$

3.1.4 Properties of Valid Plans

lemma *valid-plan-valid-head*:
assumes $(h \# as \in valid-plans\ PROB)$
shows $h \in PROB$
 $\langle proof \rangle$

lemma *valid-plan-valid-tail*:
assumes $(h \# as \in valid-plans\ PROB)$
shows $(as \in valid-plans\ PROB)$
 $\langle proof \rangle$

lemma *valid-plan-pre-subset-prob-dom-pair*:
assumes $as \in valid-plans\ PROB$
shows $(\forall a. ListMem\ a\ as \longrightarrow fmdom' (fst a) \subseteq (prob-dom\ PROB))$
 $\langle proof \rangle$

lemma *valid-append-valid-suff*:
assumes $as1 @ as2 \in (valid-plans\ PROB)$
shows $as2 \in (valid-plans\ PROB)$
 $\langle proof \rangle$

lemma *valid-append-valid-pref*:
assumes $as1 @ as2 \in (valid-plans\ PROB)$
shows $as1 \in (valid-plans\ PROB)$
 $\langle proof \rangle$

lemma *valid-pref-suff-valid-append*:
assumes $as1 \in (valid-plans\ PROB)\ as2 \in (valid-plans\ PROB)$
shows $(as1 @ as2) \in (valid-plans\ PROB)$
 $\langle proof \rangle$

lemma *MEM-statelist-FDOM*:
fixes $PROB\ h\ as\ s0$
assumes $s0 \in (valid-states\ PROB)\ as \in (valid-plans\ PROB)\ ListMem\ h\ (state-list\ s0\ as)$
shows $(fmdom'\ h = fmdom'\ s0)$
 $\langle proof \rangle$

lemma *MEM-statelist-valid-state*:
fixes $PROB\ h\ as\ s0$
assumes $s0 \in valid-states\ PROB\ as \in valid-plans\ PROB\ ListMem\ h\ (state-list$

$s0\ as)$
shows $(h \in \text{valid-states } PROB)$
 $\langle \text{proof} \rangle$

lemma lemma-1-i:
fixes $s\ a\ PROB$
assumes $s \in \text{valid-states } PROB\ a \in PROB$
shows $\text{state-succ } s\ a \in \text{valid-states } PROB$
 $\langle \text{proof} \rangle$

lemma lemma-1-ii:
 $\text{last } '(\#)\ s\ ' \text{state-set } (\text{state-list } (\text{state-succ } s\ a)\ as))$
 $= \text{last } ' \text{state-set } (\text{state-list } (\text{state-succ } s\ a)\ as)$
 $\langle \text{proof} \rangle$

lemma lemma-1:
fixes $as :: (('a, 'b)\ \text{fmap} \times ('a, 'b)\ \text{fmap})\ \text{list}\ \text{and}\ PPROB$
assumes $(s \in \text{valid-states } PROB)\ (as \in \text{valid-plans } PROB)$
shows $((\text{last } '(\text{state-set } (\text{state-list } s\ as))) \subseteq \text{valid-states } PROB)$
 $\langle \text{proof} \rangle$

lemma len-in-state-set-le-max-len:
fixes $as\ x\ PROB$
assumes $(s \in \text{valid-states } PROB)\ (as \in \text{valid-plans } PROB)\ \neg(as = [])$
 $(x \in \text{state-set } (\text{state-list } s\ as))$
shows $(\text{length } x \leq (\text{Suc } (\text{length } as)))$
 $\langle \text{proof} \rangle$

lemma card-state-set-cons:
fixes $as\ s\ h$
shows
 $(\text{card } (\text{state-set } (\text{state-list } s\ (h\ \#)\ as)))$
 $= \text{Suc } (\text{card } (\text{state-set } (\text{state-list } (\text{state-succ } s\ h)\ as))))$
 $\langle \text{proof} \rangle$

lemma card-state-set:
fixes $as\ s$
shows $(\text{Suc } (\text{length } as)) = \text{card } (\text{state-set } (\text{state-list } s\ as))$
 $\langle \text{proof} \rangle$

lemma neq-mems-state-set-neq-len:
fixes $as\ x\ y\ s$
assumes $x \in \text{state-set } (\text{state-list } s\ as)\ (y \in \text{state-set } (\text{state-list } s\ as))\ \neg(x = y)$
shows $\neg(\text{length } x = \text{length } y)$
 $\langle \text{proof} \rangle$

definition inj :: ('a \Rightarrow 'b) \Rightarrow 'a set \Rightarrow 'b set \Rightarrow bool where
 $\text{inj } f\ A\ B \equiv (\forall x \in A. f\ x \in B) \wedge \text{inj-on } f\ A$

— NOTE added lemma; refactored from ‘not_eq_last_diff_paths’.

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

fixes s as $PROB$

assumes $s \in \text{valid-states } PROB$ $as \in \text{valid-plans } PROB$ $x \in \text{state-set } (\text{state-list } s \ as)$

shows $\text{last } x \in \text{valid-states } PROB$

<proof>

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

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

$\neg(\text{inj } (\text{last}) (\text{state-set } (\text{state-list } s \ as)) (\text{valid-states } PROB))$

shows $\exists l1. \exists l2.$

$l1 \in \text{state-set } (\text{state-list } s \ as)$

$\wedge l2 \in \text{state-set } (\text{state-list } s \ as)$

$\wedge \text{last } l1 = \text{last } l2$

$\wedge l1 \neq l2$

<proof>

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

fixes as $PROB$ s

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

$\neg(\text{inj } (\text{last}) (\text{state-set } (\text{state-list } s \ as)) (\text{valid-states } PROB))$

shows $(\exists \text{slist-1 } \text{slist-2}.$

$(\text{slist-1} \in \text{state-set } (\text{state-list } s \ as))$

$\wedge (\text{slist-2} \in \text{state-set } (\text{state-list } s \ as))$

$\wedge ((\text{last } \text{slist-1}) = (\text{last } \text{slist-2}))$

$\wedge \neg(\text{length } \text{slist-1} = \text{length } \text{slist-2}))$

<proof>

lemma *nempty-sl-in-state-set*:

fixes sl

assumes $sl \neq []$

shows $sl \in \text{state-set } sl$

<proof>

lemma *empty-list-nin-state-set*:

fixes h slist as

assumes $(h \# \text{slist}) \in \text{state-set } (\text{state-list } s \ as)$

shows $(h = s)$

<proof>

lemma *cons-in-state-set-2*:

fixes s slist h t

assumes ($slist \neq []$) ($(s \# slist) \in state-set (state-list s (h \# t))$)
shows ($slist \in state-set (state-list (state-succ s h) t)$)
 $\langle proof \rangle$

lemma *valid-action-valid-succ*:
assumes $h \in PROB$ $s \in valid-states PROB$
shows ($state-succ s h \in valid-states PROB$)
 $\langle proof \rangle$

lemma *in-state-set-imp-eq-exec-prefix*:
fixes $slist$ as $PROB$ s
assumes ($as \neq []$) ($slist \neq []$) ($s \in valid-states PROB$) ($as \in valid-plans PROB$)
 $(slist \in state-set (state-list s as))$
shows
 $(\exists as'. (prefix as' as) \wedge (exec-plan s as' = last slist) \wedge (length slist = Suc (length as')))$
 $\langle proof \rangle$

lemma *eq-last-state-imp-append-nempty-as*:
fixes as $PROB$ $slist-1$ $slist-2$
assumes ($as \neq []$) ($s \in valid-states PROB$) ($as \in valid-plans PROB$) ($slist-1 \neq []$)
 $(slist-2 \neq [])$ ($slist-1 \in state-set (state-list s as)$)
 $(slist-2 \in state-set (state-list s as))$ $\neg (length slist-1 = length slist-2)$
 $(last slist-1 = last slist-2)$
shows ($\exists as1 as2 as3.$
 $(as1 @ as2 @ as3 = as)$
 $\wedge (exec-plan s (as1 @ as2) = exec-plan s as1)$
 $\wedge \neg (as2 = [])$
 $)$
 $\langle proof \rangle$

lemma *FINITE-prob-dom*:
assumes $finite PROB$
shows $finite (prob-dom PROB)$
 $\langle proof \rangle$

lemma *CARD-valid-states*:
assumes $finite (PROB :: 'a problem)$
shows ($card (valid-states PROB :: 'a state set) = 2 \wedge card (prob-dom PROB)$)
 $\langle proof \rangle$

lemma *FINITE-valid-states*:
fixes $PROB :: 'a problem$
shows $finite PROB \implies finite ((valid-states PROB) :: 'a state set)$
 $\langle proof \rangle$

lemma *lemma-2*:

fixes $PROB :: 'a$ problem **and** $as :: ('a$ action) list **and** $s :: 'a$ state
assumes finite $PROB$ $s \in (valid-states\ PROB)$ ($as \in valid-plans\ PROB$)
 $((length\ as) > (2 \wedge (card\ (fndom'\ s)) - 1))$
shows $(\exists as1\ as2\ as3.$
 $(as1\ @\ as2\ @\ as3 = as)$
 $\wedge (exec-plan\ s\ (as1\ @\ as2) = exec-plan\ s\ as1)$
 $\wedge \neg(as2 = []))$
 $)$
 $\langle proof \rangle$

lemma *lemma-2-prob-dom:*

fixes $PROB$ **and** $as :: ('a$ action) list **and** $s :: 'a$ state
assumes finite $PROB$ ($s \in valid-states\ PROB$) ($as \in valid-plans\ PROB$)
 $(length\ as > (2 \wedge (card\ (prob-dom\ PROB))) - 1)$
shows $(\exists as1\ as2\ as3.$
 $(as1\ @\ as2\ @\ as3 = as)$
 $\wedge (exec-plan\ s\ (as1\ @\ as2) = exec-plan\ s\ as1)$
 $\wedge \neg(as2 = []))$
 $)$
 $\langle proof \rangle$

lemma *lemma-3:*

fixes $PROB :: 'a$ problem **and** $s :: 'a$ state
assumes finite $PROB$ ($s \in valid-states\ PROB$) ($as \in valid-plans\ PROB$)
 $(length\ as > (2 \wedge (card\ (prob-dom\ PROB))) - 1)$
shows $(\exists as'.$
 $(exec-plan\ s\ as = exec-plan\ s\ as')$
 $\wedge (length\ as' < length\ as)$
 $\wedge (subseq\ as'\ as)$
 $)$
 $\langle proof \rangle$

lemma *sublist-valid-is-valid:*

fixes $as'\ as\ PROB$
assumes ($as \in valid-plans\ PROB$) ($subseq\ as'\ as$)
shows $as' \in valid-plans\ PROB$
 $\langle proof \rangle$

theorem *main-lemma:*

fixes $PROB :: 'a$ problem **and** $as\ s$
assumes finite $PROB$ ($s \in valid-states\ PROB$) ($as \in valid-plans\ PROB$)
shows $(\exists as'.$
 $(exec-plan\ s\ as = exec-plan\ s\ as')$
 $\wedge (subseq\ as'\ as)$
 $\wedge (length\ as' \leq (2 \wedge (card\ (prob-dom\ PROB))) - 1)$
 $)$
 $\langle proof \rangle$

3.2 Reachable States

definition *reachable-s* where

$reachable-s\ PROB\ s \equiv \{exec-plan\ s\ as \mid as.\ as \in valid-plans\ PROB\}$

— 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* \in *valid-plans* *PROB*) (*s* \in *valid-states* *PROB*)
shows (*exec-plan* *s* *as* \in *valid-states* *PROB*)
 $\langle proof \rangle$

lemma *exec-plan-fdom-subset*:
fixes *as* *s* *PROB*
assumes (*as* \in *valid-plans* *PROB*)
shows (*fndom'* (*exec-plan* *s* *as*) \subseteq (*fndom'* *s* \cup *prob-dom* *PROB*)
 $\langle proof \rangle$

lemma *reachable-s-finite-thm-1-a*:
fixes *s* **and** *PROB* :: 'a problem
assumes (*s* :: 'a state) \in *valid-states* *PROB*
shows ($\forall l \in reachable-s\ PROB\ s.\ l \in valid-states\ PROB$)
 $\langle proof \rangle$

lemma *reachable-s-finite-thm-1*:
assumes ((*s* :: 'a state) \in *valid-states* *PROB*)
shows (*reachable-s* *PROB* *s* \subseteq *valid-states* *PROB*)
 $\langle proof \rangle$

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*)
 $\langle proof \rangle$

lemma *empty-plan-is-valid*: $\square \in (valid-plans\ PROB)$
 $\langle proof \rangle$

lemma *valid-head-and-tail-valid-plan*:
assumes (*h* \in *PROB*) (*as* \in *valid-plans* *PROB*)
shows ((*h* $\#$ *as*) \in *valid-plans* *PROB*)
 $\langle proof \rangle$

lemma *lemma-1-reachability-s-i*:
fixes *PROB* *s*
assumes *s* \in *valid-states* *PROB*
shows *s* \in *reachable-s* *PROB* *s*
 $\langle proof \rangle$

lemma *lemma-1-reachability-s*:
fixes *PROB* :: 'a problem **and** *s* :: 'a state **and** *as*
assumes (*s* \in *valid-states* *PROB*) (*as* \in *valid-plans* *PROB*)

shows $((\text{last } \text{state-set } (\text{state-list } s \text{ as})) \subseteq (\text{reachable-s } \text{PROB } s))$
 <proof>

lemma not-eq-last-diff-paths-reachability-s:
fixes $\text{PROB} :: 'a \text{ problem}$ **and** $s :: 'a \text{ state}$ **and** as
assumes $s \in \text{valid-states } \text{PROB}$ $as \in \text{valid-plans } \text{PROB}$
 $\neg(\text{inj last } (\text{state-set } (\text{state-list } s \text{ as})) (\text{reachable-s } \text{PROB } s))$
shows $(\exists \text{slist-1 slist-2.}$
 $\text{slist-1} \in \text{state-set } (\text{state-list } s \text{ as})$
 $\wedge \text{slist-2} \in \text{state-set } (\text{state-list } s \text{ as})$
 $\wedge (\text{last slist-1} = \text{last slist-2})$
 $\wedge \neg(\text{length slist-1} = \text{length slist-2})$
 $)$
 <proof>

lemma lemma-2-reachability-s-i:
fixes $f :: 'a \Rightarrow 'b$ **and** $s \ t$
assumes $\text{finite } t$ $\text{card } t < \text{card } s$
shows $\neg(\text{inj } f \ s \ t)$
 <proof>

lemma lemma-2-reachability-s:
fixes $\text{PROB} :: 'a \text{ problem}$ **and** $as \ s$
assumes $\text{finite } \text{PROB}$ $(s \in \text{valid-states } \text{PROB})$ $(as \in \text{valid-plans } \text{PROB})$
 $(\text{length } as > \text{card } (\text{reachable-s } \text{PROB } s) - 1)$
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>

lemma lemma-3-reachability-s:
fixes as **and** $\text{PROB} :: 'a \text{ problem}$ **and** s
assumes $\text{finite } \text{PROB}$ $(s \in \text{valid-states } \text{PROB})$ $(as \in \text{valid-plans } \text{PROB})$
 $(\text{length } as > (\text{card } (\text{reachable-s } \text{PROB } s) - 1))$
shows $(\exists as'.$
 $(\text{exec-plan } s \ as = \text{exec-plan } s \ as')$
 $\wedge (\text{length } as' < \text{length } as)$
 $\wedge (\text{subseq } as' \ as)$
 $)$
 <proof>

lemma main-lemma-reachability-s:
fixes $\text{PROB} :: 'a \text{ problem}$ **and** as **and** $s :: 'a \text{ state}$
assumes $\text{finite } \text{PROB}$ $(s \in \text{valid-states } \text{PROB})$ $(as \in \text{valid-plans } \text{PROB})$
shows $(\exists as'.$
 $(\text{exec-plan } s \ as = \text{exec-plan } s \ as') \wedge \text{subseq } as' \ as$
 $\wedge (\text{length } as' \leq (\text{card } (\text{reachable-s } \text{PROB } s) - 1)))$
 <proof>

lemma reachable-s-non-empty: $\neg(\text{reachable-s } \text{PROB } s = \{\})$

<proof>

lemma *card-reachable-s-non-zero*:

fixes s

assumes *finite* ($PROB :: 'a$ problem) ($s \in \text{valid-states } PROB$)

shows ($0 < \text{card } (\text{reachable-s } PROB s)$)

<proof>

lemma *exec-fdom-empty-prob*:

fixes s

assumes ($\text{prob-dom } PROB = \{\}$) ($s \in \text{valid-states } PROB$) ($as \in \text{valid-plans } PROB$)

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

<proof>

lemma *reachable-s-empty-prob*:

fixes $PROB :: 'a$ problem **and** $s :: 'a$ state

assumes ($\text{prob-dom } PROB = \{\}$) ($s \in \text{valid-states } PROB$)

shows ($(\text{reachable-s } PROB s) \subseteq \{\text{fmempty}\}$)

<proof>

lemma *sublist-valid-plan--alt*:

assumes ($as1 \in \text{valid-plans } PROB$) ($\text{subseq } as2 as1$)

shows ($as2 \in \text{valid-plans } PROB$)

<proof>

lemma *fmsubset-eq*:

assumes $s1 \subseteq_f s2$

shows ($\forall a. a \in | \text{fmdom } s1 \longrightarrow \text{fmlookup } s1 a = \text{fmlookup } s2 a$)

<proof>

lemma *submap-imp-state-succ-submap-a*:

assumes $s1 \subseteq_f s2$ $s2 \subseteq_f s3$

shows $s1 \subseteq_f s3$

<proof>

lemma *submap-imp-state-succ-submap-b*:

assumes $s1 \subseteq_f s2$

shows ($s0 ++ s1$) \subseteq_f ($s0 ++ s2$)

<proof>

lemma *submap-imp-state-succ-submap*:

fixes $a :: 'a$ action **and** $s1 s2$

assumes ($\text{fst } a \subseteq_f s1$) ($s1 \subseteq_f s2$)

shows ($\text{state-succ } s1 a \subseteq_f \text{state-succ } s2 a$)

<proof>

lemma *pred-dom-subset-succ-submap*:

fixes $a :: 'a$ action **and** $s1 s2 :: 'a$ state

assumes ($\text{fmdom}' (\text{fst } a) \subseteq \text{fmdom}' s1$) ($s1 \subseteq_f s2$)

shows ($\text{state-succ } s1 a \subseteq_f \text{state-succ } s2 a$)

<proof>

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

fixes $s\ a$

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

<proof>

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

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

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

shows $(\text{exec-plan}\ s1\ as \subseteq_f \text{exec-plan}\ s2\ as)$

<proof>

lemma *valid-plan-mems:*

assumes $(as \in \text{valid-plans}\ PROB) (\text{ListMem}\ a\ as)$

shows $a \in PROB$

<proof>

lemma *valid-states-nempty:*

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

assumes *finite* $PROB$

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

<proof>

lemma *empty-prob-dom-single-val-state:*

assumes $(\text{prob-dom}\ PROB = \{\})$

shows $(\exists s. \text{valid-states}\ PROB = \{s\})$

<proof>

lemma *empty-prob-dom-imp-empty-plan-always-good:*

fixes $PROB\ s$

assumes $(\text{prob-dom}\ PROB = \{\}) (s \in \text{valid-states}\ PROB) (as \in \text{valid-plans}\ PROB)$

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

<proof>

lemma *empty-prob-dom:*

fixes $PROB$

assumes $(\text{prob-dom}\ PROB = \{\})$

shows $(PROB = \{(fmempty, fmempty)\} \vee PROB = \{\})$

<proof>

lemma *empty-prob-dom-finite:*

fixes $PROB :: 'a\ \text{problem}$

assumes $\text{prob-dom}\ PROB = \{\}$

shows *finite* $PROB$

<proof>

lemma *disj-imp-eq-proj-exec:*

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

lemma *no-change-vs-eff-submap*:

fixes $a\ vs\ s$
assumes $(fmrestrict\ set\ vs\ s = fmrestrict\ set\ vs\ (state\ succ\ s\ a))\ (fst\ a \subseteq_f\ s)$
shows $(fmrestrict\ set\ vs\ (snd\ a) \subseteq_f\ (fmrestrict\ set\ vs\ s))$
 $\langle proof \rangle$

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 $((fmrestrict\ set\ vs\ (state\ succ\ s\ a)) = (fmrestrict\ set\ vs\ s))$
 $\langle proof \rangle$

lemma *proj-eq-proj-exec-eq*:

fixes $s\ s'\ vs$ **and** $a :: ('a, 'b) fmap \times ('a, 'b) fmap$ **and** a'
assumes $((fmrestrict\ set\ vs\ s) = (fmrestrict\ set\ vs\ s'))\ ((fst\ a \subseteq_f\ s) = (fst\ a' \subseteq_f\ s'))$
 $(fmrestrict\ set\ vs\ (snd\ a) = fmrestrict\ set\ vs\ (snd\ a'))$
shows $(fmrestrict\ set\ vs\ (state\ succ\ s\ a) = fmrestrict\ set\ vs\ (state\ succ\ s'\ a'))$
 $\langle proof \rangle$

lemma *empty-eff-exec-eq*:

fixes $s\ a$
assumes $(fmdom' (snd\ a) = \{\})$
shows $(state\ succ\ s\ a = s)$
 $\langle proof \rangle$

lemma *exec-as-proj-valid-2*:

fixes a
assumes $a \in PROB$
shows $(action\ dom\ (fst\ a)\ (snd\ a) \subseteq prob\ dom\ PROB)$
 $\langle proof \rangle$

lemma *valid-filter-valid-as*:

assumes $(as \in valid\ plans\ PROB)$
shows $(filter\ P\ as \in valid\ plans\ PROB)$
 $\langle proof \rangle$

lemma *sublist-valid-plan*:

assumes $(subseq\ as'\ as)\ (as \in valid\ plans\ PROB)$
shows $(as' \in valid\ plans\ PROB)$
 $\langle proof \rangle$

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

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

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

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

3.3 State Spaces

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

lemma *EQ-SS-DOM*:
assumes $\neg(ss = \{\}) (stateSpace\ ss\ vs1) (stateSpace\ ss\ vs2)$
shows $(vs1 = vs2)$
 $\langle proof \rangle$

lemma *FINITE-SS*:
fixes $ss :: ('a, bool) fmap\ set$
assumes $\neg(ss = \{\}) (stateSpace\ ss\ domain)$
shows *finite* ss
 $\langle proof \rangle$

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

3.4 Needed Asses

definition *action-needed-vars where*

$action\text{-}needed\text{-}vars\ a\ s \equiv \{v. (v \in fmdom'\ s) \wedge (v \in fmdom'\ (fst\ a)) \wedge (fmlookup\ (fst\ a)\ v = fmlookup\ s\ v)\}$

— NOTE name shortened to 'action_needed_asses'.

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

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

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

lemma *act-needed-asses-submap-succ-submap*:

fixes $a\ s1\ s2$

assumes $(action\text{-}needed\text{-}asses\ a\ s2 \subseteq_f action\text{-}needed\text{-}asses\ a\ s1)\ (s1 \subseteq_f s2)$

shows $(state\text{-}succ\ s1\ a \subseteq_f state\text{-}succ\ s2\ a)$

<proof>

lemma *as-needed-asses-submap-exec-i*:

fixes $a\ s$

assumes $v \in fmdom'\ (action\text{-}needed\text{-}asses\ a\ s)$

shows

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

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

<proof>

lemma *as-needed-asses-submap-exec-ii*:

fixes $f\ g\ v$

assumes $v \in fmdom'\ f\ f \subseteq_f g$

shows $fmlookup\ f\ v = fmlookup\ g\ v$

<proof>

lemma *as-needed-asses-submap-exec-iii*:

fixes $f\ g\ v$

shows

$fmdom'\ (action\text{-}needed\text{-}asses\ a\ s)$

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

<proof>

lemma *as-needed-asses-submap-exec-iv*:

fixes $f\ a\ v$

assumes $v \in fmdom'\ (action\text{-}needed\text{-}asses\ a\ s)$

shows

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

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

$\wedge fmlookup\ (fst\ a)\ v = fmlookup\ s\ v$

<proof>

lemma *as-needed-asses-submap-exec-v*:

fixes $f\ g\ v$

assumes $v \in fmdom'\ f\ f \subseteq_f g$

shows $v \in fmdom'\ g$

<proof>

lemma *as-needed-asses-submap-exec-vi*:

fixes $a\ s1\ s2\ v$

assumes $v \in fmdom'\ (action\text{-}needed\text{-}asses\ a\ s1)$

$(action\text{-}needed\text{-}asses\ a\ s1) \subseteq_f (action\text{-}needed\text{-}asses\ a\ s2)$

shows

$(fmlookup (action-needed-asses a s1) v) = fmlookup (fst a) v$
 $\wedge (fmlookup (action-needed-asses a s2) v) = fmlookup (fst a) v \wedge$
 $fmlookup s1 v = fmlookup (fst a) v \wedge fmlookup s2 v = fmlookup (fst a) v$
 <proof>

lemma *as-needed-asses-submap-exec-vii*:
fixes $f g v$
assumes $\forall v \in fmdom' f. fmlookup f v = fmlookup g v$
shows $f \subseteq_f g$
 <proof>

lemma *as-needed-asses-submap-exec-viii*:
fixes $f g v$
assumes $f \subseteq_f g$
shows $\forall v \in fmdom' f. fmlookup f v = fmlookup g v$
 <proof>

lemma *as-needed-asses-submap-exec-viii'*:
fixes $f g v$
assumes $f \subseteq_f g$
shows $fmdom' f \subseteq fmdom' g$
 <proof>

lemma *as-needed-asses-submap-exec-ix*:
fixes $f g$
shows $f \subseteq_f g = (\forall v \in fmdom' f. fmlookup f v = fmlookup g v)$
 <proof>

lemma *as-needed-asses-submap-exec-x*:
fixes $f a v$
assumes $v \in fmdom' (action-needed-asses a f)$
shows $v \in fmdom' (fst a) \wedge v \in fmdom' f \wedge fmlookup (fst a) v = fmlookup f v$
 <proof>

lemma *as-needed-asses-submap-exec-xi*:
fixes $v a f g$
assumes $v \in fmdom' (action-needed-asses a (f ++ g)) \wedge v \in fmdom' f$
shows
 $fmlookup (action-needed-asses a (f ++ g)) v = fmlookup f v$
 $\wedge fmlookup (action-needed-asses a (f ++ g)) v = fmlookup (fst a) v$
 <proof>

lemma *as-needed-asses-submap-exec-xii*:
fixes $f g v$
assumes $v \in fmdom' f$
shows $fmlookup (f ++ g) v = fmlookup f v$
 <proof>

lemma *as-needed-asses-submap-exec-xii'*:
fixes $f g v$
assumes $v \notin fmdom' f \wedge v \in fmdom' g$
shows $fmlookup (f ++ g) v = fmlookup g v$
 <proof>

lemma *as-needed-asses-submap-exec*:
fixes $s1 s2$
assumes $(s1 \subseteq_f s2)$
 $(\forall a. ListMem a as \longrightarrow (action-needed-asses a s2 \subseteq_f action-needed-asses a s1))$

shows (*exec-plan* $s1$ $as \subseteq_f$ *exec-plan* $s2$ as)
(*proof*)

definition *system-needed-vars* **where**

system-needed-vars $PROB$ $s \equiv (\bigcup \{action-needed-vars\ a\ s \mid a. a \in PROB\})$

— NOTE name shortened.

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

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

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

assumes ($a \in PROB$)

shows (*action-needed-vars* $a\ s \subseteq$ *system-needed-vars* $PROB\ s$)

(*proof*)

lemma *action-needed-asses-submap-sys-needed-asses*:

assumes ($a \in PROB$)

shows (*action-needed-asses* $a\ s \subseteq_f$ *system-needed-asses* $PROB\ s$)

(*proof*)

lemma *system-needed-asses-include-action-needed-asses-1*:

assumes ($a \in PROB$)

shows (*action-needed-vars* $a\ (fmrestrict-set\ (system-needed-vars\ PROB\ s)\ s) =$
action-needed-vars $a\ s$)

(*proof*)

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*)

lemma *system-needed-asses-include-action-needed-asses*:

assumes ($a \in PROB$)

shows (*action-needed-asses* $a\ (system-needed-asses\ PROB\ s) =$ *action-needed-asses*
 $a\ s$)

(*proof*)

lemma *system-needed-asses-submap*:

system-needed-asses $PROB\ s \subseteq_f\ s$

(*proof*)

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* $s\ as$)

<proof>

```
end  
theory ActionSeqProcess  
  imports Main HOL-Library.Sublist FactoredSystemLib FactoredSystem FSSub-  
list  
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  
| sat-precond-as s (a # as) = (fst a  $\subseteq_f$  s  $\wedge$  sat-precond-as (state-succ s a) as)
```

— NOTE added lemma.

```
lemma sat-precond-as-pair:  
  sat-precond-as s ((p, e) # as) = (p  $\subseteq_f$  s  $\wedge$  sat-precond-as (state-succ s (p, e))  
as)  
<proof>
```

```
fun rem-effectless-act where  
  rem-effectless-act [] = []  
| rem-effectless-act (a # as) = (if fmdom' (snd a)  $\neq$  {}  
  then (a # rem-effectless-act as)  
  else rem-effectless-act as  
)
```

— NOTE 'fun' because of multiple defining equations.

```
fun no-effectless-act where  
  no-effectless-act [] = True  
| no-effectless-act (a # as) = ((fmdom' (snd a)  $\neq$  {})  $\wedge$  no-effectless-act as)
```

```
lemma graph-plan-lemma-4:  
  fixes s s' as vs P  
  assumes ( $\forall a. (ListMem a as \wedge P a) \longrightarrow ((fmdom' (snd a) \cap vs) = \{\})$ )  
  sat-precond-as s as
```

sat-precond-as s' (filter (λa. ¬(P a)) as) (fmrestrict-set vs s = fmrestrict-set vs s')

shows

*(fmrestrict-set vs (exec-plan s as)
= fmrestrict-set vs (exec-plan s' (filter (λ a. ¬(P a)) as)))*

<proof>

fun *rem-condless-act* **where**

rem-condless-act s pfx-a [] = pfx-a
| *rem-condless-act s pfx-a (a # as) = (if fst a ⊆_f exec-plan s pfx-a*
then rem-condless-act s (pfx-a @ [a]) as
else rem-condless-act s pfx-a as
)

lemma *rem-condless-act-pair:*

rem-condless-act s pfx-a ((p, e) # as) = (if p ⊆_f exec-plan s pfx-a
then rem-condless-act s (pfx-a @ [(p,e)]) as
else rem-condless-act s pfx-a as
)

(rem-condless-act s pfx-a [] = pfx-a)

<proof>

lemma *exec-remcondless-cons:*

fixes *s h as pfx*

shows

exec-plan s (rem-condless-act s (h # pfx) as)
= exec-plan (state-succ s h) (rem-condless-act (state-succ s h) pfx as)

<proof>

lemma *rem-condless-valid-1:*

fixes *as s*

shows *(exec-plan s as = exec-plan s (rem-condless-act s [] as))*

<proof>

lemma *rem-condless-act-cons:*

fixes *h' pfx as s*

shows *(rem-condless-act s (h' # pfx) as) = (h' # rem-condless-act (state-succ s h') pfx as)*

<proof>

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

fixes *h h' as as' s*

assumes $\text{prefix } (h' \# as')$ ($\text{rem-condless-act } s [h] as$)
shows (
 $(\text{prefix } as' (\text{rem-condless-act } (\text{state-succ } s h) [] as))$
 $\wedge h' = h$
 $)$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-2:*

fixes $as s$
shows $\text{sat-precond-as } s (\text{rem-condless-act } s [] as)$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-3:*

fixes $as s$
shows $\text{length } (\text{rem-condless-act } s [] as) \leq \text{length } as$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-4:*

fixes $as A s$
assumes $(\text{set } as \subseteq A)$
shows $(\text{set } (\text{rem-condless-act } s [] as) \subseteq A)$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-6:*

fixes $as s P$
shows $\text{length } (\text{filter } P (\text{rem-condless-act } s [] as)) \leq \text{length } (\text{filter } P as)$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-7:*

fixes $s P as as2$
assumes $(\text{list-all } P as \wedge \text{list-all } P as2)$
shows $\text{list-all } P (\text{rem-condless-act } s as2 as)$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-8:*

fixes $s as$
shows $\text{subseq } (\text{rem-condless-act } s [] as) as$
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid-10:*

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

shows ($\text{rem-condless-act } s \sqcup as \in \text{valid-plans } PROB$)
 $\langle \text{proof} \rangle$

lemma *rem-condless-valid*:

fixes $as A s$
assumes ($\text{exec-plan } s as = \text{exec-plan } s (\text{rem-condless-act } s \sqcup as)$)
 $(\text{sat-precond-as } s (\text{rem-condless-act } s \sqcup as))$
 $(\text{length } (\text{rem-condless-act } s \sqcup as) \leq \text{length } as)$
 $((\text{set } as \subseteq A) \longrightarrow (\text{set } (\text{rem-condless-act } s \sqcup as) \subseteq A))$
shows ($\forall P. (\text{length } (\text{filter } P (\text{rem-condless-act } s \sqcup as)) \leq \text{length } (\text{filter } P as))$)
 $\langle \text{proof} \rangle$

lemma *submap-sat-precond-submap*:

fixes $as :: 'a \text{ action list}$
assumes ($s1 \subseteq_f s2$) ($\text{sat-precond-as } s1 as$)
shows ($\text{sat-precond-as } s2 as$)
 $\langle \text{proof} \rangle$

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

fixes $s1 s2$
assumes ($s1 \subseteq_f s2$) ($\text{sat-precond-as } s1 (a \# as)$)
shows $\text{state-succ } s1 a \subseteq_f \text{state-succ } s2 a$
 $\langle \text{proof} \rangle$

lemma *submap-init-submap-exec*:

fixes $s1 s2$
assumes ($s1 \subseteq_f s2$) ($\text{sat-precond-as } s1 as$)
shows ($\text{exec-plan } s1 as \subseteq_f \text{exec-plan } s2 as$)
 $\langle \text{proof} \rangle$

lemma *sat-precond-drest-sat-precond*:

fixes $vs s$ **and** $as :: 'a \text{ action list}$
assumes $\text{sat-precond-as } (\text{fmrestrict-set } vs s) as$
shows ($\text{sat-precond-as } s as$)
 $\langle \text{proof} \rangle$

definition *varset-action where*

$\text{varset-action } a \text{ varset} \equiv (\text{fmdom}' (\text{snd } a) \subseteq \text{varset})$

for $a :: 'a \text{ action}$

lemma *varset-action-pair*: ($\text{varset-action } (p, e) vs = (\text{fmdom}' e \subseteq vs)$)

$\langle \text{proof} \rangle$

lemma *eq-effect-eq-vset*:

fixes $x y$
assumes ($\text{snd } x = \text{snd } y$)
shows ($(\lambda a. \text{varset-action } a vs) x = (\lambda a. \text{varset-action } a vs) y$)
 $\langle \text{proof} \rangle$

lemma *rem-effectless-works-1:*

fixes $s\ as$

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

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-2:*

fixes $as\ s$

assumes $(\text{sat-precond-as } s\ as)$

shows $(\text{sat-precond-as } s\ (\text{rem-effectless-act } as))$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-3:*

fixes as

shows $\text{length } (\text{rem-effectless-act } as) \leq \text{length } as$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-4:*

fixes $A\ as$

assumes $(\text{set } as \subseteq A)$

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

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-4':*

fixes $A\ as$

assumes $(as \in \text{valid-plans } A)$

shows $(\text{rem-effectless-act } as \in \text{valid-plans } A)$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-5-i:*

shows $\text{subseq } (\text{rem-effectless-act } as)\ as$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-5:*

fixes $P\ as$

shows $\text{length } (\text{filter } P\ (\text{rem-effectless-act } as)) \leq \text{length } (\text{filter } P\ as)$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-6:*

fixes as

shows $\text{no-effectless-act } (\text{rem-effectless-act } as)$

$\langle \text{proof} \rangle$

lemma *rem-effectless-works-7:*

fixes as

shows $no\text{-effectless-act } as = list\text{-all } (\lambda a. fndom' (snd a) \neq \{\}) as$
 $\langle proof \rangle$

lemma *rem-effectless-works-8:*

fixes $P as$

assumes $(list\text{-all } P as)$

shows $list\text{-all } P (rem\text{-effectless-act } as)$
 $\langle proof \rangle$

lemma *rem-effectless-works-9:*

fixes as

shows $subseq (rem\text{-effectless-act } as) as$
 $\langle proof \rangle$

lemma *rem-effectless-works-10:*

fixes $as P$

assumes $(no\text{-effectless-act } as)$

shows $(no\text{-effectless-act } (filter P as))$
 $\langle proof \rangle$

lemma *rem-effectless-works-11:*

fixes $as1 as2$

assumes $subseq as1 (rem\text{-effectless-act } as2)$

shows $(subseq as1 as2)$
 $\langle proof \rangle$

lemma *rem-effectless-works-12:*

fixes $as1 as2$

shows $(no\text{-effectless-act } (as1 @ as2)) = (no\text{-effectless-act } as1 \wedge no\text{-effectless-act}(as2))$
 $\langle proof \rangle$

lemma *rem-effectless-works-13-i:*

fixes $x l$

assumes $ListMem x l list\text{-all } P l$

shows $P x$
 $\langle proof \rangle$

lemma *rem-effectless-works-13:*

fixes $as1 as2$

assumes $(subseq as1 as2) (no\text{-effectless-act } as2)$

shows $(no\text{-effectless-act } as1)$
 $\langle proof \rangle$

lemma *rem-effectless-works-14:*

fixes $PROB as$

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

<proof>

lemma *rem-effectless-works*:

fixes $s A as$

assumes $(exec-plan\ s\ as = exec-plan\ s\ (rem-effectless-act\ as))$

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

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

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

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

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

<proof>

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

<proof>

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

fixes A

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

<proof>

lemma *rem-condless-valid-9*:

fixes $s as$

assumes $no-effectless-act\ as$

shows $no-effectless-act\ (rem-condless-act\ s\ []\ as)$

<proof>

lemma *graph-plan-lemma-17*:

fixes $as-1\ as-2\ as\ s$

assumes $(as-1\ @\ as-2 = as)\ (sat-precond-as\ s\ as)$

shows $((sat-precond-as\ s\ as-1) \wedge sat-precond-as\ (exec-plan\ s\ as-1)\ as-2)$

<proof>

lemma *nempty-eff-every-nempty-act*:

fixes as

assumes $(no-effectless-act\ as)\ (\forall x. \neg(fmdom'\ (snd\ (f\ x)) = \{\}))$

shows $(list-all\ (\lambda a. \neg(f\ a = (fmempty, fmempty)))\ as)$

<proof>

lemma *empty-replace-proj-dual7*:

fixes s as as'
assumes $sat\text{-}precond\text{-}as$ s (as @ as')
shows $sat\text{-}precond\text{-}as$ ($exec\text{-}plan$ s as) as'
 $\langle proof \rangle$

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

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

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

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

lemma $vset\text{-}disj\text{-}eff\text{-}diff$:
fixes s a vs
assumes ($varset\text{-}action$ a vs)
shows $((fmdom' (snd\ a) \cap (s - vs)) = \{\})$
 $\langle proof \rangle$

lemma $list\text{-}all\text{-}list\text{-}mem$:
fixes P **and** $l :: 'a$ $list$
shows $list\text{-}all\ P\ l \longleftrightarrow (\forall e. ListMem\ e\ l \longrightarrow P\ e)$
 $\langle proof \rangle$

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

<proof>

lemma *no-effectless-act-works*:

fixes *as*

assumes (*no-effectless-act as*)

shows (*filter* ($\lambda a. \neg(\text{fmdom}'(\text{snd } a) = \{\})$) *as* = *as*)

<proof>

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)*)

<proof>

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')*)

<proof>

lemma *valid-filter-vset-dom-idempot*:

fixes *PROB as*

assumes (*as \in valid-plans PROB*)

shows (*filter* ($\lambda a. \text{varset-action } a (\text{prob-dom } \text{PROB})$) *as* = *as*)

<proof>

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'*)

<proof>

lemma *sat-precond-as-pfx*:

fixes *s*

assumes (*sat-precond-as s (as @ as')*)

shows (*sat-precond-as s as*)

<proof>

end

theory *RelUtils*

imports *Main HOL.Transitive-Closure*

begin

— NOTE added definition.

definition *reflexive* **where**

reflexive $R \equiv \forall x. R x x$

— NOTE translation of 'TC' in relationScript.sml:69.

— TODO can we replace this with something from 'HOL.Transitive_Closure'?

definition *TC* **where**

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

— NOTE adapts transitive closure definitions of Isabelle and HOL4.

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

<proof>

lemma *TC-IMP-NOT-TC-CONJ-1*:

fixes $R P$ **and** $x y$

assumes $\neg(R^{++} x y)$

shows $\neg((\lambda x y. R x y \wedge P x y)^{++} x y)$

<proof>

lemma *TC-IMP-NOT-TC-CONJ*:

fixes $R R' P x y$

assumes $\forall x y. P x y \longrightarrow R' x y \longrightarrow R x y \neg R^{++} x y$

shows $\neg(\lambda x y. R' x y \wedge P x y)^{++} x y$

<proof>

lemma *TC-INDUCT*:

fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** P

assumes $(\forall x y. R x y \longrightarrow P x y) (\forall x y z. P x y \wedge P y z \longrightarrow P x z)$

shows $\forall u v. (TC R) u v \longrightarrow P u v$

<proof>

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

fixes $R P x y$

assumes $((\lambda x y. R x y \wedge P x y)^{++} x y)$

shows $R^{++} x y$

<proof>

lemma *REFL-IMP-3-CONJ*:

fixes R'

assumes *reflexive* R'

shows $(\forall P x y.$

$(R'^{++} x y) \longrightarrow ((\lambda x y. R' x y \wedge P x \wedge P y)^{++} x y) \vee (\exists z. \neg P z \wedge R'^{++} x z \wedge R'^{++} z y))$

<proof>

lemma *REFL-TC-CONJ*:

fixes $R R' :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $P x y$

assumes *reflexive* $R' \forall x y. P x \wedge P y \longrightarrow (R' x y \longrightarrow R x y) \neg(R^{++} x y)$

```

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

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

— NOTE name shortened to 'dep_var_set'.

definition *dep-var-set where*

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

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

lemma *DEP-REFL*:
fixes *PROB*
shows *reflexive* $(\lambda v v'. dep\ PROB\ v\ v')$
 $\langle proof \rangle$

lemma *NEQ-DEP-IMP-IN-DOM-i*:
fixes *a v*
assumes $a \in PROB\ v \in fmdom'\ (fst\ a)$
shows $v \in prob-dom\ PROB$
 $\langle proof \rangle$

lemma *NEQ-DEP-IMP-IN-DOM-ii*:
fixes *a v*
assumes $a \in PROB\ v \in fmdom'\ (snd\ a)$
shows $v \in prob-dom\ PROB$
 $\langle proof \rangle$

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

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

lemma *dep-union-imp-or-dep*:
fixes *PROB vs vs' vs''*
assumes $(dep-var-set\ PROB\ vs\ (vs' \cup vs''))$
shows $(dep-var-set\ PROB\ vs\ vs' \vee dep-var-set\ PROB\ vs\ vs'')$
 $\langle proof \rangle$

lemma *dep-biunion-imp-or-dep*:
fixes *PROB vs S*
assumes $(dep-var-set\ PROB\ vs\ (\bigcup S))$
shows $(\exists vs'. vs' \in S \wedge dep-var-set\ PROB\ vs\ vs')$
 $\langle proof \rangle$

5.2 Transitive Closure of Dependent Variables and Variable Sets

definition *dep-tc* where

$$\text{dep-tc } PROB = TC (\lambda v1' v2'. \text{dep } PROB v1' v2')$$

— NOTE type of ‘PROB’ had to be fixed for MP on ‘NEQ_DEP_IMP_IN_DOM’:

lemma *dep-tc-imp-in-dom*:

fixes *PROB* :: (*'a*, *'b*) *fmap* × (*'a*, *'b*) *fmap*) *set* **and** *v1 v2*

assumes $\neg(v1 = v2)$ (*dep-tc* *PROB* *v1 v2*)

shows (*v1* ∈ *prob-dom* *PROB*)

⟨*proof*⟩

lemma *not-dep-disj-imp-not-dep*:

fixes *PROB vs-1 vs-2 vs-3*

assumes $((vs-1 \cap vs-2) = \{\})$ (*vs-3* ⊆ *vs-2*) $\neg(\text{dep-var-set } PROB vs-1 vs-2)$

shows $\neg(\text{dep-var-set } PROB vs-1 vs-3)$

⟨*proof*⟩

lemma *dep-slist-imp-mem-dep*:

fixes *PROB vs lvs*

assumes (*dep-var-set* *PROB* (\bigcup (*set* *lvs*)) *vs*)

shows ($\exists vs'. \text{ListMem } vs' lvs \wedge \text{dep-var-set } PROB vs' vs$)

⟨*proof*⟩

lemma *n-bigunion-le-sum-3*:

fixes *PROB vs sv*

assumes $(\forall vs'. vs' \in sv \longrightarrow \neg(\text{dep-var-set } PROB vs' vs))$

shows $\neg(\text{dep-var-set } PROB (\bigcup sv) vs)$

⟨*proof*⟩

lemma *disj-not-dep-vset-union-imp-or*:

fixes *PROB a vs vs'*

assumes (*a* ∈ *PROB*) (*disjnt* *vs vs'*)

$(\neg(\text{dep-var-set } PROB vs' vs) \vee \neg(\text{dep-var-set } PROB vs vs'))$

(*varset-action* *a* (*vs* ∪ *vs'*))

shows (*varset-action* *a* *vs* ∨ *varset-action* *a* *vs'*)

⟨*proof*⟩

end

theory *Invariants*

imports *Main FactoredSystem*

begin

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

— TODO function domain for total function in Isabelle/HOL?

— TODO why is fm total? Shouldn't it be partial and thus needing the the premise 'fm x = Some True' instead of just 'fm x'?

definition $invariant :: ('a \Rightarrow bool) \Rightarrow bool$ **where**

$invariant fm \equiv (\forall x. (x \in fdom fm \wedge fm x) \longrightarrow False) \wedge (\exists x. x \in fdom fm \wedge fm x)$

end

theory *SetUtils*

imports *Main*

begin

— TODO use Inf instead of Min where necessary.

— TODO can be replaced by $card\text{-}Un\text{-}disjoint$ ($\llbracket finite A; finite B; A \cap B = \{\} \rrbracket \implies card (A \cup B) = card A + card B$) ?

lemma $card\text{-}union'$: $(finite s) \wedge (finite t) \wedge (disjnt s t) \implies (card (s \cup t) = card s + card t)$

<proof>

lemma $CARD\text{-}INJ\text{-}IMAGE\text{-}2$:

fixes $f s$

assumes $finite s (\forall x y. ((x \in s) \wedge (y \in s)) \longrightarrow ((f x = f y) \longleftrightarrow (x = y)))$

shows $(card (f ' s) = card s)$

<proof>

lemma $scc\text{-}main\text{-}lemma\text{-}x$: $\bigwedge s t x. (x \in s) \wedge \neg(x \in t) \implies \neg(s = t)$

<proof>

lemma $neq\text{-}funs\text{-}neq\text{-}images$:

fixes s

assumes $\forall x. x \in s \longrightarrow (\forall y. y \in s \longrightarrow f1 x \neq f2 y) \exists x. x \in s$

shows $f1 ' s \neq f2 ' s$

<proof>

5.3 Sets of Numbers

lemma $mems\text{-}le\text{-}finite\text{-}i$:

fixes $s :: nat \text{ set}$ **and** $k :: nat$

shows $(\forall x. x \in s \longrightarrow x \leq k) \implies finite s$

<proof>

lemma $mems\text{-}le\text{-}finite$:

fixes $s :: nat \text{ set}$ **and** $k :: nat$

shows $\bigwedge (s :: nat \text{ set}) k. (\forall x. x \in s \longrightarrow x \leq k) \implies finite s$

<proof>

lemma $mem\text{-}le\text{-}imp\text{-}MIN\text{-}le$:

```

fixes  $s :: \text{nat set}$  and  $k :: \text{nat}$ 
assumes  $\exists x. (x \in s) \wedge (x \leq k)$ 
shows  $(\text{Inf } s \leq k)$ 
<proof>
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>
lemma bound-child-parent-neq-mems-state-set-neq-len:
fixes  $s$  and  $k :: \text{nat}$ 
assumes  $(\forall x. x \in s \longrightarrow x < k)$ 
shows finite s
<proof>

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

<proof>

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>

lemma FINITE-ALL-DISTINCT-LISTS:
fixes  $P$ 
assumes finite P
shows finite  $\{p. \text{distinct } p \wedge \text{set } p \subseteq P\}$ 
<proof>

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

end
theory TopologicalProps
imports Main FactoredSystem ActionSeqProcess SetUtils
begin

```

6 Topological Properties

6.1 Basic Definitions and Properties

definition *PLS-charles* **where**

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

definition *MPLS-charles* **where**

$$\begin{aligned} \text{MPLS-charles } PROB &\equiv \{\text{Inf } (\text{PLS-charles } (fst \ p) \ (snd \ p) \ PROB) \mid p. \\ & ((fst \ p) \in \text{valid-states } PROB) \\ & \wedge ((snd \ p) \in \text{valid-plans } PROB) \\ & \} \end{aligned}$$

— NOTE name shortened to 'problem_plan_bound_charles'.

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

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

— NOTE name shortened to 'PLS_state'.

definition *PLS-state-1* **where**

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

— NOTE name shortened to 'MPLS_stage_1'.

definition *MPLS-stage-1* **where**

$$\begin{aligned} \text{MPLS-stage-1 } PROB &\equiv \\ & (\lambda (s, as). \text{Inf } (\text{PLS-state-1 } s \text{ as})) \\ & \{ (s, as). (s \in \text{valid-states } PROB) \wedge (as \in \text{valid-plans } PROB) \} \end{aligned}$$

— NOTE name shortened to 'problem_plan_bound_stage_1'.

definition *problem-plan-bound-stage-1* **where**

$$\text{problem-plan-bound-stage-1 } PROB \equiv \text{Sup } (\text{MPLS-stage-1 } PROB)$$

for $PROB :: 'a \text{ problem}$

— NOTE name shortened.

definition *PLS* **where**

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

— NOTE added lemma.

— NOTE proof finite PLS for use in 'proof in_MPLS_leq_2_pow_n_i'

lemma *finite-PLS*: *finite* ($\text{PLS } s \text{ as}$)

<proof>

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

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

fixes $PROB$

shows

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

$\langle \text{proof} \rangle$

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

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

shows

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

$\langle \text{proof} \rangle$

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**

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

lemma *valid-path-ITP2015*:

$(\text{valid-path } Pi [] \longleftrightarrow \text{True})$
 $\wedge (\text{valid-path } Pi [s] \longleftrightarrow (s \in \text{valid-states } Pi))$

$$\begin{aligned} & \wedge (\text{valid-path } Pi \ (s1 \ \# \ s2 \ \# \ rest) \longleftrightarrow \\ & \quad (s1 \in \text{valid-states } Pi) \\ & \quad \wedge (\exists a. \\ & \quad \quad (a \in Pi) \\ & \quad \quad \wedge (\text{exec-plan } s1 \ [a] = s2) \\ & \quad \quad) \\ & \quad \wedge (\text{valid-path } Pi \ (s2 \ \# \ rest)) \\ &) \end{aligned}$$

<proof>

definition *RD* **where**

$RD \ Pi \equiv (\text{Sup } \{\text{length } p - 1 \mid p. \text{valid-path } Pi \ p \wedge \text{distinct } p\})$
for $Pi :: 'a \text{ problem}$

lemma *in-PLS-leq-2-pow-n*:

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

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

shows ($\exists x.$

$(x \in \text{PLS } s \ as)$

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

)

<proof>

lemma *in-MPLS-leq-2-pow-n*:

fixes $PROB :: 'a \text{ problem}$ **and** x

assumes *finite* $PROB$ ($x \in \text{MPLS } PROB$)

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

<proof>

lemma *FINITE-MPLS*:

assumes *finite* ($Pi :: 'a \text{ problem}$)

shows *finite* ($\text{MPLS } Pi$)

<proof>

fun *statelist'* **where**

$\text{statelist}' \ s \ [] = [s]$

$| \text{statelist}' \ s \ (a \ \# \ as) = (s \ \# \ \text{statelist}' \ (state-succ \ s \ a) \ as)$

lemma *LENGTH-statelist'*:

fixes $as \ s$

shows $\text{length } (\text{statelist}' \ s \ as) = (\text{length } as + 1)$

<proof>

lemma *valid-path-statelist'*:

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

assumes ($as \in \text{valid-plans } Pi$) ($s \in \text{valid-states } Pi$)
shows ($\text{valid-path } Pi$ ($\text{statelist}' s as$))
 $\langle \text{proof} \rangle$

lemma *statelist'-exec-plan*:
fixes $a s p$
assumes ($\text{statelist}' s as = p$)
shows ($\text{exec-plan } s as = \text{last } p$)
 $\langle \text{proof} \rangle$

lemma *statelist'-EQ-NIL*: $\text{statelist}' s as \neq []$
 $\langle \text{proof} \rangle$

lemma *statelist'-TAKE-i*:
assumes $\text{Suc } m \leq \text{length } (a \# as)$
shows $m \leq \text{length } as$
 $\langle \text{proof} \rangle$

lemma *statelist'-TAKE*:
fixes $as s p$
assumes ($\text{statelist}' s as = p$)
shows ($\forall n. n \leq \text{length } as \longrightarrow (\text{exec-plan } s (\text{take } n as)) = (p ! n)$)
 $\langle \text{proof} \rangle$

lemma *MPLS-nempty*:
fixes $PROB :: (('a, 'b) \text{fmap} \times ('a, 'b) \text{fmap}) \text{set}$
assumes *finite* $PROB$
shows $\text{MPLS } PROB \neq \{\}$
 $\langle \text{proof} \rangle$

theorem *bound-main-lemma*:
fixes $PROB :: 'a \text{problem}$
assumes *finite* $PROB$
shows ($\text{problem-plan-bound } PROB \leq (2 \wedge (\text{card } (\text{prob-dom } PROB))) - 1$)
 $\langle \text{proof} \rangle$

lemma *bound-child-parent-card-state-set-cons*:
fixes $P f$
assumes ($\forall (PROB :: 'a \text{problem}) as (s :: 'a \text{state}).$
 $(P \text{ } PROB)$
 $\wedge (as \in \text{valid-plans } PROB)$
 $\wedge (s \in \text{valid-states } PROB)$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s as = \text{exec-plan } s as')$
 $\wedge (\text{subseq } as' as)$
 $\wedge (\text{length } as' < f \text{ } PROB)$
 $)$
 $)$
shows ($\forall PROB s as.$

$(P \text{ PROB})$
 $\wedge (as \in \text{valid-plans } \text{PROB})$
 $\wedge (s \in (\text{valid-states } \text{PROB}))$
 $\longrightarrow (\exists x.$
 $(x \in \text{PLS } s \text{ as})$
 $\wedge (x < f \text{ PROB})$
 $)$
 $)$
 $\langle \text{proof} \rangle$
lemma *bound-on-all-plans-bounds-MPLS*:
fixes $P f$
assumes $(\forall (\text{PROB} :: 'a \text{ problem}) \text{ as } (s :: 'a \text{ state}).$
 $(P \text{ PROB})$
 $\wedge (s \in \text{valid-states } \text{PROB})$
 $\wedge (as \in \text{valid-plans } \text{PROB})$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s \text{ as} = \text{exec-plan } s \text{ as}')$
 $\wedge (\text{subseq } as' \text{ as})$
 $\wedge (\text{length } as' < f \text{ PROB})$
 $)$
 $)$
shows $(\forall \text{PROB } x. P \text{ PROB}$
 $\longrightarrow (x \in \text{MPLS}(\text{PROB}))$
 $\longrightarrow (x < f \text{ PROB})$
 $)$
 $\langle \text{proof} \rangle$

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

lemma *bound-on-all-plans-bounds-MPLS-finite*:
fixes $P f$
assumes $(\forall \text{PROB } as \text{ s.}$

$P \text{ PROB} \wedge \text{finite } \text{PROB} \wedge s \in (\text{valid-states } \text{PROB}) \wedge as \in (\text{valid-plans } \text{PROB})$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s \text{ } as = \text{exec-plan } s \text{ } as')$
 $\wedge \text{subseq } as' \text{ } as$
 $\wedge \text{length } as' < f(\text{PROB})$
 $)$
 $)$
shows $(\forall \text{PROB } x.$
 $P \text{ PROB} \wedge \text{finite } \text{PROB}$
 $\longrightarrow (x \in \text{MPLS } \text{PROB})$
 $\longrightarrow x < f \text{PROB}$
 $)$
 $\langle \text{proof} \rangle$

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

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

lemma *x-in-MPLS-if*:

fixes x *PROB*

assumes $x \in \text{MPLS } \text{PROB}$

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

<proof>

lemma *bound-on-all-plans-bounds-MPLS-thesis*:

assumes *finite* *PROB* $(\forall \text{ as } s.$

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

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

$\longrightarrow (\exists \text{ as}'.$

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

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

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

$)$

$) (x \in \text{MPLS } \text{PROB})$

shows $(x < k)$

<proof>

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>

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

assumes *finite* *PROB* $(\forall \text{ as } s.$

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

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

$\longrightarrow (\exists \text{ as}'.$

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

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

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

$)$

$)$

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

<proof>

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

assumes *finite* *PROB* $(\forall \text{ as } s.$

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

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

$\longrightarrow (\exists \text{ as}'.$

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

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

$\wedge (\text{length } \text{as}' \leq k)$

$)$

$)$

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

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

fixes $P f PROB$

assumes $(\forall PROB' as s.$

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

$\longrightarrow (\exists as'.$

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

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

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

$)$

$) (P\ PROB)\ finite\ PROB$

shows (*problem-plan-bound* $PROB < f\ PROB$)

⟨*proof*⟩

lemma *S-VALID-AS-VALID-IMP-MIN-IN-PLS*:

fixes $PROB s as$

assumes $(s \in valid-states\ PROB) (as \in valid-plans\ PROB)$

shows $(Inf\ (PLS\ s\ as) \in (MPLS\ PROB))$

⟨*proof*⟩

lemma *problem-plan-bound-ge-min-pls*:

fixes $PROB :: 'a\ problem$ **and** $s :: 'a\ state$ **and** $as\ k$

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

$(problem-plan-bound\ PROB \leq k)$

shows $(Inf\ (PLS\ s\ as) \leq problem-plan-bound\ PROB)$

⟨*proof*⟩

lemma *PLS-NEMPTY*:

fixes $s as$

shows $PLS\ s\ as \neq \{\}$

⟨*proof*⟩

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

fixes $s as$

shows $(\exists x. (x \in PLS\ s\ as) \wedge (x = Inf\ (PLS\ s\ as)))$

⟨*proof*⟩

lemma *PLS-works*:

fixes $x s as$

assumes $(x \in PLS\ s\ as)$

shows $(\exists as'.$

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

```

       $\wedge (\text{length } as' = x)$ 
       $\wedge (\text{subseq } as' as)$ 
    )
  <proof>
lemma problem-plan-bound-works:
  fixes  $PROB :: 'a \text{ problem and } as \text{ and } s :: 'a \text{ state}$ 
  assumes  $\text{finite } PROB (s \in \text{valid-states } PROB) (as \in \text{valid-plans } PROB)$ 
  shows  $(\exists as'.$ 
     $(\text{exec-plan } s as = \text{exec-plan } s as')$ 
     $\wedge (\text{subseq } as' as)$ 
     $\wedge (\text{length } as' \leq \text{problem-plan-bound } PROB)$ 
  )
  <proof>
definition MPLS-s where
   $MPLS-s \text{ } PROB \ s \equiv (\lambda (s, as). \text{Inf } (PLS \ s \ as)) \text{ } \{(s, as) \mid as. as \in \text{valid-plans } PROB\}$ 

— NOTE type of ‘PROB’ had to be fixed (type mismatch in goal).
lemma bound-main-lemma-s-3:
  fixes  $PROB :: (('a, 'b) \text{ fmap} \times ('a, 'b) \text{ fmap}) \text{ set and } s$ 
  shows  $MPLS-s \text{ } PROB \ s \neq \{\}$ 
  <proof>
definition problem-plan-bound-s where
   $\text{problem-plan-bound-s } PROB \ s = \text{Sup } (MPLS-s \text{ } PROB \ s)$ 

— NOTE removed typing from assumption due to matching problems in later proofs.
lemma bound-on-all-plans-bounds-PLS-s:
  fixes  $P \ f$ 
  assumes  $(\forall PROB \ as \ s.$ 
     $\text{finite } PROB \wedge (P \text{ } PROB) \wedge (as \in \text{valid-plans } PROB) \wedge (s \in \text{valid-states } PROB)$ 
     $\longrightarrow (\exists as'.$ 
       $(\text{exec-plan } s \ as = \text{exec-plan } s \ as')$ 
       $\wedge (\text{subseq } as' \ as)$ 
       $\wedge (\text{length } as' < f \text{ } PROB \ s)$ 
    )
  )
  shows  $(\forall PROB \ s \ as.$ 
     $\text{finite } PROB \wedge (P \text{ } PROB) \wedge (as \in \text{valid-plans } PROB) \wedge (s \in \text{valid-states } PROB)$ 
     $\longrightarrow (\exists x.$ 
       $(x \in PLS \ s \ as)$ 
       $\wedge (x < f \text{ } PROB \ s)$ 
    )
  )

```

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

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

lemma *Sup-MPLS-s-lt-if*:
fixes $PROB\ s\ k$
assumes $(\forall x \in \text{MPLS-s } PROB\ s. x < k)$
shows $\text{Sup } (\text{MPLS-s } PROB\ s) < k$
 $\langle proof \rangle$

lemma *bound-child-parent-lemma-s-2*:
fixes $PROB :: 'a\ \text{problem and } P :: 'a\ \text{problem} \Rightarrow \text{bool and } s\ f$
assumes $(\forall (PROB :: 'a\ \text{problem})\ as\ s.$
 $\text{finite } PROB \wedge (P\ PROB) \wedge (s \in \text{valid-states } PROB) \wedge (as \in \text{valid-plans } PROB)$
 $\longrightarrow (\exists as'.$
 $(\text{exec-plan } s\ as = \text{exec-plan } s\ as')$
 $\wedge (\text{subseq } as'\ as)$
 $\wedge (\text{length } as' < f\ PROB\ s)$
 $)$
 $)$
shows $($
 $\text{finite } PROB \wedge (P\ PROB) \wedge (s \in \text{valid-states } PROB)$
 $\longrightarrow \text{problem-plan-bound-s } PROB\ s < f\ PROB\ s$
 $)$
 $\langle proof \rangle$

theorem *bound-main-lemma-reachability-s*:
fixes $PROB :: 'a\ \text{problem and } s$

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

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

lemma *AS-VALID-MPLS-VALID:*
fixes *PROB as*
assumes (*as ∈ valid-plans PROB*)
shows (*Inf (PLS s as) ∈ MPLS-s PROB s*)
⟨*proof*⟩

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

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

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

lemma *problem-plan-bound-s-works:*
fixes *PROB :: 'a problem and as s*
assumes *finite PROB (as ∈ valid-plans PROB) (s ∈ valid-states PROB)*
shows ($\exists as'$.
(*exec-plan s as = exec-plan s as'*)
 \wedge (*subseq as' as*)
 \wedge (*length as' ≤ problem-plan-bound-s PROB s*)
)
⟨*proof*⟩

lemma *PLS-def-ITP2015:*

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

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

fixes $PROB :: 'a\ problem$

shows

$problem-plan-bound-charles\ PROB$

$= Sup\ ($

$\{$

$Inf\ (PLS-charles\ (fst\ p)\ (snd\ p)\ PROB)$

$\mid p. (fst\ p \in valid-states\ PROB) \wedge (snd\ p \in valid-plans\ PROB)\}$

$\langle proof \rangle$

lemma *bound-main-lemma-charles-3:*

fixes $PROB :: 'a\ problem$

assumes $finite\ PROB$

shows $MPLS-charles\ PROB \neq \{\}$

$\langle proof \rangle$

lemma *in-PLS-charles-leq-2-pow-n:*

fixes $PROB :: 'a\ problem$ **and** $s\ as$

assumes $finite\ PROB\ s \in valid-states\ PROB\ as \in valid-plans\ PROB$

shows $(\exists x.$

$(x \in PLS-charles\ s\ as\ PROB)$

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

$\langle proof \rangle$

lemma *x-in-MPLS-charles-then:*

fixes $PROB\ s\ as$

assumes $x \in MPLS-charles\ PROB$

shows $\exists s\ as.$

$s \in valid-states\ PROB \wedge as \in valid-plans\ PROB \wedge x = Inf\ (PLS-charles\ s\ as\ PROB)$

$\langle proof \rangle$

lemma *in-MPLS-charles-leq-2-pow-n:*

fixes $PROB :: 'a\ problem$ **and** x

assumes $finite\ PROB\ x \in MPLS-charles\ PROB$

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

$\langle proof \rangle$

lemma *bound-main-lemma-charles:*

fixes $PROB :: 'a\ problem$

assumes *finite PROB*
shows *problem-plan-bound-charles* $PROB \leq 2 \wedge (\text{card}(\text{prob-dom } PROB) - 1)$
 $\langle \text{proof} \rangle$

lemma *bound-on-all-plans-bounds-PLS-charles:*

fixes P **and** f

assumes $\forall (PROB :: 'a \text{ problem}) \text{ as } s.$

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

$\longrightarrow (\exists \text{ as}'.$

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

shows $(\forall PROB \text{ s as}.$

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

$\longrightarrow (\exists x.$

$(x \in \text{PLS-charles } s \text{ as } PROB)$

$\wedge (x < f \text{ PROB}))$

$\langle \text{proof} \rangle$

lemma *bound-on-all-plans-bounds-MPLS-charles-i:*

assumes $\forall (PROB :: 'a \text{ problem}) \text{ s as}.$

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

$\longrightarrow (\exists \text{ as}'.$

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

shows $\forall (PROB :: 'a \text{ problem}) \text{ s as}.$

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

$\longrightarrow \text{Inf } \{n. n \in \text{PLS-charles } s \text{ as } PROB\} < f \text{ PROB}$

$\langle \text{proof} \rangle$

lemma *bound-on-all-plans-bounds-MPLS-charles:*

fixes P f

assumes $(\forall (PROB :: 'a \text{ problem}) \text{ as } s.$

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

$\longrightarrow (\exists \text{ as}'.$

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

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

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

)

shows $(\forall PROB \text{ x}.$

$(P \text{ PROB}) \wedge \text{finite } PROB$

$\longrightarrow (x \in \text{MPLS-charles } PROB)$

$\longrightarrow (x < f \text{ PROB})$

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

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

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

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

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

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

<proof>
lemma *RD-bounds-sublistD-i-a*:
 fixes $Pi :: 'a$ problem
 assumes *finite* Pi
 shows *finite* $\{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\}$
 <proof>
lemma *RD-bounds-sublistD-i-b*:
 fixes $Pi :: 'a$ problem
 shows $\{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\} \neq \{\}$
 <proof>
lemma *RD-bounds-sublistD-i-c*:
 fixes $Pi :: 'a$ problem and $as :: ((a, bool) \text{fmap} \times (a, bool) \text{fmap}) \text{list}$ and x
 and $s :: (a, bool) \text{fmap}$
 assumes $s \in \text{valid-states } Pi$ and $as \in \text{valid-plans } Pi$
 $(\forall y. y \in \{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\} \longrightarrow y \leq x)$
 $x \in \{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\}$
 shows $\text{Min } (PLS\ s\ as) \leq \text{Max } \{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\}$
 <proof>
lemma *RD-bounds-sublistD-i*:
 fixes $Pi :: 'a$ problem and x
 assumes *finite* Pi $(\forall y. y \in \text{MPLS } Pi \longrightarrow y \leq x)$ and $x \in \text{MPLS } Pi$
 shows $x \leq \text{Max } \{length\ p - 1 \mid p. \text{valid-path } Pi\ p \wedge \text{distinct } p\}$
 <proof>
lemma *RD-bounds-sublistD*:
 fixes $Pi :: 'a$ problem
 assumes *finite* Pi
 shows *problem-plan-bound* $Pi \leq \text{RD } Pi$
 <proof>
theorem *sublistD-bounds-D-and-RD-bounds-sublistD*:
 fixes $PROB :: 'a$ problem
 assumes *finite* $PROB$
 shows
 $\text{problem-plan-bound-charles } PROB \leq \text{problem-plan-bound } PROB$
 $\wedge \text{problem-plan-bound } PROB \leq \text{RD } PROB$

 <proof>
lemma *empty-problem-bound*:
 fixes $PROB :: 'a$ problem
 assumes $(\text{prob-dom } PROB = \{\})$
 shows $(\text{problem-plan-bound } PROB = 0)$
 <proof>

lemma *problem-plan-bound-works'*:
 fixes $PROB :: 'a$ problem and $as\ s$
 assumes *finite* $PROB$ $(s \in \text{valid-states } PROB)$ and $(as \in \text{valid-plans } PROB)$
 shows $(\exists as')$
 $(\text{exec-plan } s\ as' = \text{exec-plan } s\ as)$
 $\wedge (\text{subseq } as'\ as)$

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

6.4 Traversal Diameter

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

lemma *finite-traversed-states*: $\text{finite } (\text{traversed-states } s \ as)$
 $\langle \text{proof} \rangle$

lemma *traversed-states-nempty*: $\text{traversed-states } s \ as \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *traversed-states-geq-1*:
fixes s
shows $1 \leq \text{card } (\text{traversed-states } s \ as)$
 $\langle \text{proof} \rangle$

lemma *init-is-traversed*: $s \in \text{traversed-states } s \ as$
 $\langle \text{proof} \rangle$

definition *td where*
 $\text{td } PROB \equiv \text{Sup } \{$
 $(\text{card } (\text{traversed-states } (\text{fst } p) (\text{snd } p))) - 1$
 $\mid p. (\text{fst } p \in \text{valid-states } PROB) \wedge (\text{snd } p \in \text{valid-plans } PROB)\}$

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

```

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

  <proof>

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

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

begin

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

```

7 System Abstraction

Projection of an object (state, action, sequence of action or factored representation) to a variable set ‘vs’ restricts the domain of the object or its

components—in case of composite objects—to ‘vs’. [Abdulaziz et al., p.12]

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

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

definition *action-proj* **where**

action-proj a $vs \equiv (fmrestrict\text{-}set\ vs\ (fst\ a),\ fmrestrict\text{-}set\ vs\ (snd\ a))$

lemma *action-proj-pair*: *action-proj* (p, e) $vs = (fmrestrict\text{-}set\ vs\ p,\ fmrestrict\text{-}set\ vs\ e)$

<proof>

definition *prob-proj* **where**

prob-proj $PROB$ $vs \equiv (\lambda a.\ action\text{-}proj\ a\ vs)\ 'PROB$

— NOTE using ‘fun’ due to multiple defining equations.

— NOTE name shortened.

fun *as-proj* **where**

as-proj $[]$ $- = []$

| *as-proj* $(a \# as)$ $vs = (if\ fmdom'\ (fmrestrict\text{-}set\ vs\ (snd\ a)) \neq \{\})$
 then *action-proj* a $vs \# as\text{-}proj\ as\ vs$
 else *as-proj* $as\ vs$
)

— TODO the lemma might be superfluous (follows directly from ‘as_proj.simps’).

lemma *as-proj-pair*:

as-proj $((p, e) \# as)$ $vs = (if\ (fmdom'\ (fmrestrict\text{-}set\ vs\ e)) \neq \{\})$

then *action-proj* (p, e) $vs \# as\text{-}proj\ as\ vs$

else *as-proj* $as\ vs$

)

as-proj $[]$ $vs = []$

<proof>

lemma *proj-state-succ*:

fixes $s\ a\ vs$

assumes $(fst\ a \subseteq_f\ s)$

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

<proof>

lemma *graph-plan-lemma-1*:

fixes s vs as
assumes $sat\text{-}precond\text{-}as$ s as
shows $(exec\text{-}plan (fmrestrict\text{-}set\ vs\ s) (as\text{-}proj\ as\ vs) = (fmrestrict\text{-}set\ vs (exec\text{-}plan\ s\ as)))$
 $\langle proof \rangle$

lemma $proj\text{-}action\text{-}dom\text{-}eq\text{-}inter$:

shows
 $action\text{-}dom (fst (action\text{-}proj\ a\ vs)) (snd (action\text{-}proj\ a\ vs))$
 $= (action\text{-}dom (fst\ a) (snd\ a) \cap vs)$

$\langle proof \rangle$

lemma $graph\text{-}plan\text{-}neq\text{-}mems\text{-}state\text{-}set\text{-}neq\text{-}len$:

shows $prob\text{-}dom (prob\text{-}proj\ PROB\ vs) = (prob\text{-}dom\ PROB \cap vs)$

$\langle proof \rangle$

lemma $graph\text{-}plan\text{-}not\text{-}eq\text{-}last\text{-}diff\text{-}paths$:

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

$\langle proof \rangle$

lemma $dom\text{-}eff\text{-}subset\text{-}imp\text{-}dom\text{-}succ\text{-}eq\text{-}proj$:

fixes h s vs
assumes $(fmdom' (snd\ h) \subseteq fmdom' s)$
shows $(fmdom' (state\text{-}succ\ s (action\text{-}proj\ h\ vs)) = fmdom' (state\text{-}succ\ s\ h))$

$\langle proof \rangle$

lemma $drest\text{-}proj\text{-}succ\text{-}eq\text{-}drest\text{-}succ$:

fixes h s vs
assumes $fst\ h \subseteq_f s$ $(fmdom' (snd\ h) \subseteq fmdom' s)$
shows $(fmrestrict\text{-}set\ vs (state\text{-}succ\ s (action\text{-}proj\ h\ vs)) = fmrestrict\text{-}set\ vs (state\text{-}succ\ s\ h))$

$\langle proof \rangle$

lemma $drest\text{-}succ\text{-}proj\text{-}eq\text{-}drest\text{-}succ$:

fixes s vs as
assumes $(fst\ a \subseteq_f s)$
shows $(state\text{-}succ (fmrestrict\text{-}set\ vs\ s) (action\text{-}proj\ a\ vs) = fmrestrict\text{-}set\ vs (state\text{-}succ\ s\ a))$

$\langle proof \rangle$

lemma $exec\text{-}drest\text{-}cons\text{-}proj\text{-}eq\text{-}succ$:

fixes as $PROB$ vs a
assumes $fst\ a \subseteq_f s$
shows (

$$\begin{aligned} & \text{exec-plan } (fmrestrict\text{-set } vs \ s) \ (action\text{-proj } a \ vs \ \# \ as) \\ & = \text{exec-plan } (fmrestrict\text{-set } vs \ (state\text{-succ } s \ a)) \ as \\ &) \\ \langle proof \rangle \end{aligned}$$

lemma *exec-drest*:

fixes $as \ a \ vs$
assumes $(fst \ a \ \subseteq_f \ s)$
shows (

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

 \rangle
 $\langle proof \rangle$

lemma *not-empty-eff-in-as-proj*:

fixes $as \ a \ vs$
assumes $fmdom' \ (fmrestrict\text{-set } vs \ (snd \ a)) \neq \{\}$
shows $(as\text{-proj } (a \ \# \ as) \ vs = (action\text{-proj } a \ vs \ \# \ as\text{-proj } as \ vs))$
 $\langle proof \rangle$

lemma *empty-eff-not-in-as-proj*:

fixes $as \ a \ vs$
assumes $(fmdom' \ (fmrestrict\text{-set } vs \ (snd \ a)) = \{\})$
shows $(as\text{-proj } (a \ \# \ as) \ vs = as\text{-proj } as \ vs)$
 $\langle proof \rangle$

lemma *empty-eff-drest-no-eff*:

fixes s **and** a **and** vs
assumes $(fmdom' \ (fmrestrict\text{-set } vs \ (snd \ a)) = \{\})$
shows $(fmrestrict\text{-set } vs \ (state\text{-succ } s \ (action\text{-proj } a \ vs)) = fmrestrict\text{-set } vs \ s)$
 $\langle proof \rangle$

lemma *sat-precond-exec-as-proj-eq-proj-exec*:

fixes $as \ vs \ s$
assumes $(sat\text{-precond-as } s \ as)$
shows $(\text{exec-plan } (fmrestrict\text{-set } vs \ s) \ (as\text{-proj } as \ vs) = fmrestrict\text{-set } vs \ (\text{exec-plan } s \ as))$
 $\langle proof \rangle$

lemma *action-proj-in-prob-proj*:

assumes $(a \in PROB)$
shows $(action\text{-proj } a \ vs \in prob\text{-proj } PROB \ vs)$
 $\langle proof \rangle$

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*))
 \langle *proof* \rangle

lemma *finite-imp-finite-prob-proj*:
fixes *PROB*
assumes *finite* *PROB*
shows (*finite* (*prob-proj* *PROB vs*))
 \langle *proof* \rangle

lemma
fixes *PROB vs as* **and** *s* :: 'a *state*
assumes *finite* *PROB s* \in *valid-states* *PROB as* \in (*valid-plans* *PROB*) *finite vs*
 $\text{length } (as\text{-proj } as\ vs) > ((2 :: nat) \wedge \text{card } vs) - 1$ *sat-precond-as s as*
shows ($\exists as1\ as2\ as3.$
 $(as1 @ as2 @ as3 = as\text{-proj } as\ vs)$
 $\wedge (exec\text{-plan } (fmrestrict\text{-set } vs\ s) (as1 @ as2) = exec\text{-plan } (fmrestrict\text{-set } vs\ s)$
 $as1)$
 $\wedge (as2 \neq [])$
 $)$
 \langle *proof* \rangle

lemma *as-proj-eq-filter-action-proj*:
fixes *as vs*
shows *as-proj as vs* = *filter* ($\lambda a. fmdom' (snd\ a) \neq \{\}$) (*map* ($\lambda a. action\text{-proj } a$
 $vs)$ *as*)
 \langle *proof* \rangle

lemma *append-eq-as-proj*:
fixes *as1 as2 as3 p vs*
assumes (*as1 @ as2 @ as3* = *as-proj p vs*)
shows ($\exists p-1\ p-2\ p-3.$
 $(p-1 @ p-2 @ p-3 = p)$
 $\wedge (as2 = as\text{-proj } p-2\ vs)$
 $\wedge (as1 = as\text{-proj } p-1\ vs)$
 $)$
 \langle *proof* \rangle

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

<proof>

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>

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>

lemma *graph-plan-lemma-9*:

fixes s as vs

shows

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

<proof>

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>

lemma *exec-as-proj-valid*:

fixes as s $PROB$ vs

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

shows $(exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs)) \in valid\text{-}states\ PROB$

<proof>

lemma *drest-exec-as-proj-eq-drest-exec*:

fixes s as vs

assumes $sat\text{-}precond\text{-}as\ s\ as$

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

$\langle proof \rangle$

lemma *action-proj-idempot*:

fixes $a\ vs$

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

$\langle proof \rangle$

lemma *action-proj-idempot'*:

fixes $a\ vs$

assumes $(action\text{-}dom\ (fst\ a)\ (snd\ a) \subseteq vs)$

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

$\langle proof \rangle$

lemma *action-proj-idempot''*:

fixes $P\ vs$

assumes $prob\text{-}dom\ P \subseteq vs$

shows $prob\text{-}proj\ P\ vs = P$

$\langle proof \rangle$

lemma *sat-precond-as-proj*:

fixes $as\ s\ s'\ vs$

assumes $(sat\text{-}precond\text{-}as\ s\ as)\ (fmrestrict\text{-}set\ vs\ s = fmrestrict\text{-}set\ vs\ s')$

shows $(sat\text{-}precond\text{-}as\ s'\ (as\text{-}proj\ as\ vs))$

$\langle proof \rangle$

lemma *sat-precond-drest-as-proj*:

fixes $as\ s\ s'\ vs$

assumes $(sat\text{-}precond\text{-}as\ s\ as)\ (fmrestrict\text{-}set\ vs\ s = fmrestrict\text{-}set\ vs\ s')$

shows $(sat\text{-}precond\text{-}as\ (fmrestrict\text{-}set\ vs\ s')\ (as\text{-}proj\ as\ vs))$

$\langle proof \rangle$

lemma *as-proj-eq-as*:

assumes $(no\text{-}effectless\text{-}act\ as)\ (as \in valid\text{-}plans\ PROB)\ (prob\text{-}dom\ PROB \subseteq vs)$

shows $(as\text{-}proj\ as\ vs = as)$

$\langle proof \rangle$

lemma *exec-rem-effless-as-proj-eq-exec-as-proj*:

fixes s

shows $exec\text{-}plan\ s\ (as\text{-}proj\ (rem\text{-}effectless\text{-}act\ as)\ vs) = exec\text{-}plan\ s\ (as\text{-}proj\ as\ vs)$

$\langle proof \rangle$

lemma *exec-as-proj-eq-exec-as*:

fixes *PROB as vs s*

assumes (*as* \in *valid-plans* *PROB*) (*prob-dom* *PROB* \subseteq *vs*)

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

<proof>

lemma *dom-prob-proj*: *prob-dom* (*prob-proj* *PROB vs*) \subseteq *vs*

<proof>

lemma *subset-proj-absorb-1-a*:

fixes *f vs1 vs2*

assumes (*vs1* \subseteq *vs2*)

shows *fmrestrict-set* *vs1* (*fmrestrict-set* *vs2 f*) = *fmrestrict-set* *vs1 f*

<proof>

lemma *subset-proj-absorb-1*:

assumes (*vs1* \subseteq *vs2*)

shows (*action-proj* (*action-proj* *a vs2*) *vs1* = *action-proj* *a vs1*)

<proof>

lemma *subset-proj-absorb*:

fixes *PROB vs1 vs2*

assumes *vs1* \subseteq *vs2*

shows *prob-proj* (*prob-proj* *PROB vs2*) *vs1* = *prob-proj* *PROB vs1*

<proof>

lemma *union-proj-absorb*:

fixes *PROB vs vs'*

shows *prob-proj* (*prob-proj* *PROB* (*vs* \cup *vs'*)) *vs* = *prob-proj* *PROB vs*

<proof>

lemma *NOT-VS-IN-DOM-PROJ-PRE-EFF*:

fixes *ROB vs v a*

assumes $\neg(v \in vs)$ (*a* \in *PROB*)

shows (

((*v* \in *fmdom'* (*fst* *a*)) \longrightarrow (*v* \in *fmdom'* (*fst* (*action-proj* *a* (*prob-dom* *PROB* – *vs*))))))

\wedge ((*v* \in *fmdom'* (*snd* *a*)) \longrightarrow (*v* \in *fmdom'* (*snd* (*action-proj* *a* (*prob-dom* *PROB* – *vs*))))))

)

<proof>

lemma *IN-DISJ-DEP-IMP-DEP-DIFF*:

fixes *PROB vs vs' v v'*

assumes $(v \in vs') (v' \in vs) (disjnt\ vs\ vs')$
shows $(dep\ PROB\ v\ v' \longrightarrow dep\ (prob-proj\ PROB\ (prob-dom\ PROB - vs))\ v\ v')$
 $\langle proof \rangle$

lemma *PROB-DOM-PROJ-DIFF:*

fixes $P\ vs$
shows $prob-dom\ (prob-proj\ PROB\ (prob-dom\ PROB - vs)) = (prob-dom\ PROB - vs)$
 $\langle proof \rangle$

lemma *two-children-parent-mems-le-finite:*

fixes $PROB\ vs$
assumes $(vs \subseteq prob-dom\ PROB)$
shows $(prob-dom\ (prob-proj\ PROB\ vs) = vs)$
 $\langle proof \rangle$

lemma *PROJ-DOM-PRE-EFF-SUBSET-DOM:*

fixes $a\ vs$
shows
 $(fmdom'\ (fst\ (action-proj\ a\ vs)) \subseteq fmdom'\ (fst\ a))$
 $\wedge (fmdom'\ (snd\ (action-proj\ a\ vs)) \subseteq fmdom'\ (snd\ a))$
 $\langle proof \rangle$

lemma *NOT-IN-PRE-EFF-NOT-IN-PRE-EFF-PROJ:*

fixes $a\ v\ vs$
shows
 $(\neg(v \in fmdom'\ (fst\ a)) \longrightarrow \neg(v \in fmdom'\ (fst\ (action-proj\ a\ vs))))$
 $\wedge (\neg(v \in fmdom'\ (snd\ a)) \longrightarrow \neg(v \in fmdom'\ (snd\ (action-proj\ a\ vs))))$
 $\langle proof \rangle$

lemma *dep-proj-dep:*

assumes $dep\ (prob-proj\ PROB\ vs)\ v\ v'$
shows $dep\ PROB\ v\ v'$
 $\langle proof \rangle$

lemma *NDEP-PROJ-NDEP:*

fixes $PROB\ vs\ vs'\ vs''$
assumes $(\neg dep-var-set\ PROB\ vs\ vs')$
shows $(\neg dep-var-set\ (prob-proj\ PROB\ vs'')\ vs\ vs')$
 $\langle proof \rangle$

lemma *SUBSET-PROJ-DOM-DISJ:*

fixes $PROB\ vs\ vs'$
assumes $(vs \subseteq (prob-dom\ (prob-proj\ PROB\ (prob-dom\ PROB - vs'))))$
shows $disjnt\ vs\ vs'$
 $\langle proof \rangle$

lemma *NOT-VS-DEP-IMP-DEP-PROJ*:
fixes $PROB\ vs\ v\ v'$
assumes $\neg(v \in vs)\ \neg(v' \in vs)\ (dep\ PROB\ v\ v')$
shows $(dep\ (prob-proj\ PROB\ (prob-dom\ PROB - vs))\ v\ v')$
 $\langle proof \rangle$

lemma *DISJ-PROJ-NDEP-IMP-NDEP*:
fixes $PROB\ vs\ vs'\ vs''$
assumes
 $(disjnt\ vs\ vs'')\ disjnt\ vs\ vs'$
 $\neg(dep-var-set\ (prob-proj\ PROB\ (prob-dom\ PROB - vs))\ vs'\ vs'')$
shows $\neg(dep-var-set\ PROB\ vs'\ vs'')$
 $\langle proof \rangle$

lemma *PROJ-DOM-IDEMPOT*:
fixes $PROB$
shows $prob-proj\ PROB\ (prob-dom\ PROB) = PROB$
 $\langle proof \rangle$

lemma *prob-proj-idempot*:
fixes $vs\ vs'$
assumes $(vs \subseteq vs')$
shows $(prob-proj\ PROB\ vs = prob-proj\ (prob-proj\ PROB\ vs')\ vs)$
 $\langle proof \rangle$

lemma *prob-proj-dom-diff-eq-prob-proj-prob-proj-dom-diff*:
fixes $vs\ vs'$
shows
 $prob-proj\ PROB\ (prob-dom\ PROB - (vs \cup vs'))$
 $=\ prob-proj$
 $(prob-proj\ PROB\ (prob-dom\ PROB - vs))$
 $(prob-dom\ (prob-proj\ PROB\ (prob-dom\ PROB - vs)) - vs')$
 $\langle proof \rangle$

lemma *PROJ-DEP-IMP-DEP*:
fixes $PROB\ vs\ v\ v'$
assumes $dep\ (prob-proj\ PROB\ (prob-dom\ PROB - vs))\ v\ v'$
shows $dep\ PROB\ v\ v'$
 $\langle proof \rangle$

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 (
 $\neg((\lambda v1' v2'. \text{dep } \text{PROB } v1' v2')^{++} v v')$
 $\vee (\exists v''.$
 $v'' \in \text{vs}$
 $\wedge ((\lambda v1' v2'. \text{dep } \text{PROB } v1' v2')^{++} v v'')$
 $\wedge ((\lambda v1' v2'. \text{dep } \text{PROB } v1' v2')^{++} v'' v')$
 $)$
 $)$
 $\langle \text{proof} \rangle$

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

fixes *as vs*
shows *list-all* $(\lambda a. \text{action-proj } a \text{ vs} = a) (\text{as-proj } \text{as } \text{vs})$
 $\langle \text{proof} \rangle$

lemma *empty-eff-not-in-as-proj-2:*

fixes *a as vs*
assumes $\text{fmdom}' (\text{snd } (\text{action-proj } a \text{ vs})) = \{\}$
shows $(\text{as-proj } \text{as } \text{vs} = \text{as-proj } (a \# \text{as}) \text{ vs})$
 $\langle \text{proof} \rangle$

declare $[[\text{smt-timeout}=100]]$

lemma *sublist-as-proj-eq-as:*

fixes *as' as vs*
assumes *subseq* $\text{as}' (\text{as-proj } \text{as } \text{vs})$
shows $(\text{as-proj } \text{as}' \text{ vs} = \text{as}')$
 $\langle \text{proof} \rangle$

lemma *DISJ-EFF-DISJ-PROJ-EFF:*

fixes *a s vs*
assumes $\text{fmdom}' (\text{snd } a) \cap s = \{\}$
shows $(\text{fmdom}' (\text{snd } (\text{action-proj } a \text{ vs})) \cap s = \{\})$

$\langle \text{proof} \rangle$

lemma *state-succ-proj-eq-state-succ:*

fixes *a s vs*
assumes $(\text{varset-action } a \text{ vs}) (\text{fst } a \subseteq_f s) (\text{fmdom}' (\text{snd } a) \subseteq \text{fmdom}' s)$
shows $(\text{state-succ } s (\text{action-proj } a \text{ vs}) = \text{state-succ } s a)$
 $\langle \text{proof} \rangle$

lemma *no-effectless-proj*:

fixes $vs\ as$

shows $no\ effectless\ act\ (as\ proj\ as\ vs)$

$\langle proof \rangle$

lemma *as-proj-valid-in-prob-proj*:

fixes $PROB\ vs\ as$

assumes $(as \in valid\ plans\ PROB)$

shows $(as\ proj\ as\ vs \in valid\ plans\ (prob\ proj\ PROB\ vs))$

$\langle proof \rangle$

lemma *prob-proj-comm*:

fixes $PROB\ vs\ vs'$

shows $prob\ proj\ (prob\ proj\ PROB\ vs)\ vs' = prob\ proj\ (prob\ proj\ PROB\ vs')\ vs$

$\langle proof \rangle$

lemma *vset-proj-imp-vset*:

fixes $vs\ vs'\ a$

assumes $(varset\ action\ a\ vs')\ (varset\ action\ (action\ proj\ a\ vs')\ vs)$

shows $(varset\ action\ a\ vs)$

$\langle proof \rangle$

lemma *vset-imp-vset-act-proj-diff*:

fixes $PROB\ vs\ vs'\ a$

assumes $(varset\ action\ a\ vs)$

shows $(varset\ action\ (action\ proj\ a\ (prob\ dom\ PROB - vs'))\ vs)$

$\langle proof \rangle$

lemma *action-proj-disj-diff*:

assumes $(action\ dom\ (fst\ a)\ (snd\ a) \subseteq vs1)\ (vs2 \cap vs3 = \{\})$

shows $(action\ proj\ (action\ proj\ a\ (vs1 - vs2))\ vs3 = action\ proj\ a\ vs3)$

$\langle proof \rangle$

lemma *disj-proj-proj-eq-proj*:

fixes $PROB\ vs\ vs'$

assumes $(vs \cap vs' = \{\})$

shows $prob\ proj\ (prob\ proj\ PROB\ (prob\ dom\ PROB - vs'))\ vs = prob\ proj\ PROB$

vs

$\langle proof \rangle$

lemma *n-replace-proj-le-n-as-2*:

fixes $a\ vs\ vs'$

assumes $(vs \subseteq vs')\ (varset\ action\ a\ vs')$

shows $(varset\ action\ (action\ proj\ a\ vs')\ vs \longleftrightarrow varset\ action\ a\ vs)$

$\langle proof \rangle$

lemma *empty-problem-proj-bound*:

fixes $PROB :: 'a$ problem
shows $problem-plan-bound (prob-proj PROB \{\}) = 0$
 $\langle proof \rangle$

lemma $problem-plan-bound-works-proj$:
fixes $PROB :: 'a$ problem **and** s as vs
assumes $finite\ PROB (s \in valid-states\ PROB) (as \in valid-plans\ PROB) (sat-precond-as\ s\ as)$
shows $(\exists as')$
 $(exec-plan (fmrestrict-set vs s) as' = exec-plan (fmrestrict-set vs s) (as-proj as\ vs))$
 $\wedge (length\ as' \leq problem-plan-bound (prob-proj\ PROB\ vs))$
 $\wedge (subseq\ as' (as-proj\ as\ vs))$
 $\wedge (sat-precond-as\ s\ as')$
 $\wedge (no-effectless-act\ as')$
 \rangle
 $\langle proof \rangle$

lemma $action-proj-inter-i$: $fmrestrict-set\ V (fmrestrict-set\ W\ f) = fmrestrict-set\ (V \cap W)\ f$
 $\langle proof \rangle$

lemma $action-proj-inter$: $action-proj (action-proj\ a\ vs1)\ vs2 = action-proj\ a (vs1 \cap vs2)$
 $\langle proof \rangle$

lemma $prob-proj-inter$: $prob-proj (prob-proj\ PROB\ vs1)\ vs2 = prob-proj\ PROB (vs1 \cap vs2)$
 $\langle proof \rangle$

7.2 Snapshotting

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

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

definition $agree\ where$

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

— NOTE added lemma.

lemma $state-succ-fixpoint-if$:
fixes $a\ s\ PROB$

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

lemma *agree-state-succ-idempot*:

assumes ($a \in PROB$) ($s \in \text{valid-states } PROB$) ($\text{agree } (\text{snd } a) s$)
shows ($\text{state-succ } s a = s$)

$\langle \text{proof} \rangle$

lemma *fmdom'-fmrestrict-set*:

fixes $X f$
shows $\text{fmdom}' (\text{fmrestrict-set } X f) = X \cap (\text{fmdom}' f)$

$\langle \text{proof} \rangle$

lemma *fmdom'-fmrestrict-set-fmadd*:

fixes $X f g$
shows $\text{fmdom}' (\text{fmrestrict-set } X (f ++_f g)) = X \cap (\text{fmdom}' f \cup \text{fmdom}' g)$

$\langle \text{proof} \rangle$

lemma *fmrestrict-agree*:

fixes $X x f g$
assumes $\text{agree } (\text{fmrestrict-set } X f) (\text{fmrestrict-set } X g)$ $x \in X \cap \text{fmdom}' f \cap \text{fmdom}' g$

shows $\text{fmlookup } (\text{fmrestrict-set } X f) x = \text{fmlookup } (\text{fmrestrict-set } X g) x$

$\langle \text{proof} \rangle$

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

assumes ($a \in PROB$) ($s \in \text{valid-states } PROB$)
 $(\text{agree } (\text{fmrestrict-set } \text{vs } (\text{snd } a)) (\text{fmrestrict-set } \text{vs } s))$
shows $(\text{fmrestrict-set } \text{vs } (\text{state-succ } s a) = \text{fmrestrict-set } \text{vs } s)$

$\langle \text{proof} \rangle$

lemma *agree-exec-idempot*:

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

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

$\langle \text{proof} \rangle$

lemma *agree-restrict-exec-idempot*:

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

$(\forall a. \text{ListMem } a as \longrightarrow \text{agree } (\text{fmrestrict-set } \text{vs } (\text{snd } a)) (\text{fmrestrict-set } \text{vs } s))$
 $(\text{fmrestrict-set } \text{vs } s' = \text{fmrestrict-set } \text{vs } s)$

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

$\langle \text{proof} \rangle$

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

fixes $s s'$
assumes ($as \in \text{valid-plans } PROB$) ($s' \in \text{valid-states } PROB$) ($s \in \text{valid-states } PROB$)
 $(\forall p e. \text{ListMem } (p, e) as \longrightarrow \text{agree } (fmrestrict\text{-set } vs e) (fmrestrict\text{-set } vs s))$
 $(fmrestrict\text{-set } vs s' = fmrestrict\text{-set } vs s)$
shows $(fmrestrict\text{-set } vs (\text{exec-plan } s' as) = fmrestrict\text{-set } vs s)$
 $\langle \text{proof} \rangle$

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

lemma *restricted-agree-imp-agree*:
assumes $(fmdom' s2 \subseteq vs)$ ($\text{agree } (fmrestrict\text{-set } vs s1) s2$)
shows $(\text{agree } s1 s2)$
 $\langle \text{proof} \rangle$

lemma *agree-imp-submap*:
assumes $f1 \subseteq_f f2$
shows $\text{agree } f1 f2$
 $\langle \text{proof} \rangle$

lemma *agree-FUNION*:
assumes $(\text{agree } fm fm1)$ ($\text{agree } fm fm2$)
shows $(\text{agree } fm (fm1 ++ fm2))$
 $\langle \text{proof} \rangle$

lemma *agree-fm-list-union*:
fixes fm
assumes $(\forall fm'. \text{ListMem } fm' fmList \longrightarrow \text{agree } fm fm')$
shows $(\text{agree } fm (\text{foldr } fmap\text{-add-ltr } fmList fmempty))$
 $\langle \text{proof} \rangle$

lemma *DRESTRICT-EQ-AGREE*:
assumes $(fmdom' s2 \subseteq vs2)$ ($fmdom' s1 \subseteq vs1$)
shows $((fmrestrict\text{-set } vs2 s1 = fmrestrict\text{-set } vs1 s2) \longrightarrow \text{agree } s1 s2)$
 $\langle \text{proof} \rangle$

lemma *SUBMAPS-AGREE*: $(s1 \subseteq_f s) \wedge (s2 \subseteq_f s) \implies (\text{agree } s1 s2)$
 $\langle \text{proof} \rangle$

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

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

lemma *action-agree-valid-in-snapshot*:
assumes $(a \in PROB) (\text{agree } (fst\ a)\ s) (\text{agree } (snd\ a)\ s)$
shows $(a \in \text{snapshot } PROB\ s)$
 ⟨proof⟩

lemma *as-mem-agree-valid-in-snapshot*:
assumes $(\forall a. ListMem\ a\ as \longrightarrow \text{agree } (fst\ a)\ s \wedge \text{agree } (snd\ a)\ s) (as \in \text{valid-plans } PROB)$
shows $(as \in \text{valid-plans } (\text{snapshot } PROB\ s))$
 ⟨proof⟩

lemma *fmrestrict-agree-monotonous*:
fixes $f\ g\ X$
assumes $\text{agree } f\ g$
shows $\text{agree } (fmrestrict\text{-set } X\ f) (fmrestrict\text{-set } X\ g)$
 ⟨proof⟩

lemma *SUBMAP-FUNION-DRESTRICT-i*:
fixes $v\ vsa\ vsb\ f\ g$
assumes $v \in vsa$
shows
 $fmlookup\ (fmrestrict\text{-set } ((vsa \cup vsb) \cap vs)\ f)\ v$
 $= fmlookup\ (fmrestrict\text{-set } (vsa \cap vs)\ f)\ v$
 ⟨proof⟩

lemma *SUBMAP-FUNION-DRESTRICT'*:
assumes $(\text{agree } fma\ fmb) (vsa \subseteq fmdom'\ fma) (vsb \subseteq fmdom'\ fmb)$
 $(fmrestrict\text{-set } vsa\ fm = fmrestrict\text{-set } (vsa \cap vs)\ fma)$
 $(fmrestrict\text{-set } vsb\ fm = fmrestrict\text{-set } (vsb \cap vs)\ fmb)$
shows $(fmrestrict\text{-set } (vsa \cup vsb)\ fm = fmrestrict\text{-set } ((vsa \cup vsb) \cap vs)\ (fma$
 $++ fmb))$
 ⟨proof⟩

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

lemma *agree-DRESTRICT*:
assumes $\text{agree } s1\ s2$
shows $\text{agree } (fmrestrict\text{-set } vs\ s1) (fmrestrict\text{-set } vs\ s2)$
 ⟨proof⟩

lemma *agree-DRESTRICT-2*:

assumes $(\text{fmdom}' s1 \subseteq \text{vs1}) (\text{fmdom}' s2 \subseteq \text{vs2}) (\text{agree } s1 s2)$
shows $(\text{agree } (\text{fmrestrict-set } \text{vs2 } s1) (\text{fmrestrict-set } \text{vs1 } s2))$
 $\langle \text{proof} \rangle$

lemma *snapshot-eq-filter*:

shows $\text{snapshot } \text{PROB } s = \text{Set.filter } (\lambda a. \text{agree } (\text{fst } a) s \wedge \text{agree } (\text{snd } a) s) \text{PROB}$
 $\langle \text{proof} \rangle$

corollary *snapshot-subset*:

shows $\text{snapshot } \text{PROB } s \subseteq \text{PROB}$
 $\langle \text{proof} \rangle$

lemma *FINITE-snapshot*:

assumes *finite* PROB
shows *finite* $(\text{snapshot } \text{PROB } s)$
 $\langle \text{proof} \rangle$

lemma *dom-proj-snapshot*:

$\text{prob-dom } (\text{prob-proj } \text{PROB } (\text{prob-dom } (\text{snapshot } \text{PROB } s))) = \text{prob-dom } (\text{snapshot } \text{PROB } s)$
 $\langle \text{proof} \rangle$

lemma *valid-states-snapshot*:

$\text{valid-states } (\text{prob-proj } \text{PROB } (\text{prob-dom } (\text{snapshot } \text{PROB } s))) = \text{valid-states } (\text{snapshot } \text{PROB } s)$
 $\langle \text{proof} \rangle$

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

assumes $(x' \in \text{prob-proj } \text{PROB } \text{vs}) (\text{state-succ } s x' \neq s)$
shows $(\text{fmrestrict-set } \text{vs } (\text{state-succ } s x') \neq \text{fmrestrict-set } \text{vs } s)$
 $\langle \text{proof} \rangle$

lemma *proj-successors*:

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

$\langle \text{proof} \rangle$

lemma *state-in-successor-proj-in-state-in-successor*:

$(s' \in \text{state-successors } (\text{prob-proj } \text{PROB } \text{vs}) s)$
 $\implies (\text{fmrestrict-set } \text{vs } s' \in \text{state-successors } (\text{prob-proj } \text{PROB } \text{vs}) (\text{fmrestrict-set } \text{vs } s))$
 $\langle \text{proof} \rangle$

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 $(\text{fmdom}' e \subseteq \text{fmdom}' s)$
 $\langle \text{proof} \rangle$

lemma *valid-proj-action-valid-succ*:
assumes ($h \in \text{prob-proj } PROB \text{ vs}$) ($s \in \text{valid-states } PROB$)
shows ($\text{state-succ } s \ h \in \text{valid-states } PROB$)
 $\langle \text{proof} \rangle$

lemma *proj-successors-of-valid-are-valid*:
assumes ($s \in \text{valid-states } PROB$)
shows ($\text{state-successors } (\text{prob-proj } PROB \text{ vs}) \ s \subseteq (\text{valid-states } PROB)$)
 $\langle \text{proof} \rangle$

7.3 State Space Projection

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

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

lemma *fmrestrict-set-inter-img*:
fixes $A \ X \ Y$
shows $\text{fmrestrict-set } (X \cap Y) \ ' \ A = (\text{fmrestrict-set } X \circ \text{fmrestrict-set } Y) \ ' \ A$
 $\langle \text{proof} \rangle$

lemma *invariantStateSpace-thm-9*:
fixes $ss \ vs1 \ vs2$
shows $\text{ss-proj } ss \ (vs1 \cap \ vs2) = \text{ss-proj } (ss\text{-proj } ss \ vs2) \ vs1$
 $\langle \text{proof} \rangle$

lemma *FINITE-ss-proj*:
fixes $ss \ vs$
assumes *finite* ss
shows *finite* ($\text{ss-proj } ss \ vs$)
 $\langle \text{proof} \rangle$

lemma *nempty-stateSpace-nempty-ss-proj*:
assumes ($ss \neq \{\}$)
shows ($\text{ss-proj } ss \ vs \neq \{\}$)
 $\langle \text{proof} \rangle$

lemma *invariantStateSpace-thm-5*:
fixes $ss \ vs \ \text{domain}$
assumes ($\text{stateSpace } ss \ \text{domain}$)
shows ($\text{stateSpace } (\text{ss-proj } ss \ vs) \ (\text{domain} \cap \ vs)$)
 $\langle \text{proof} \rangle$

lemma *dom-subset-ssproj-eq-ss*:
fixes $ss \ \text{domain} \ vs$
assumes ($\text{stateSpace } ss \ \text{domain}$) ($\text{domain} \subseteq \ vs$)
shows ($\text{ss-proj } ss \ vs = \ ss$)
 $\langle \text{proof} \rangle$

```

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

```

```

lemma subset-dom-stateSpace-ss-proj:
  fixes vs1 vs2
  assumes (vs1  $\subseteq$  vs2) (stateSpace ss vs2)
  shows (stateSpace (ss-proj ss vs1) vs1)
  <proof>

```

```

lemma card-proj-leq:
  assumes finite PROB
  shows card (prob-proj PROB vs)  $\leq$  card PROB
  <proof>

```

```

end
theory Acyclicity
  imports Main
begin

```

8 Acyclicity

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

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

8.1 Topological Sorting of Dependency Graphs

```

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

```

```

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

```

```

lemma top-sorted-cons:
  assumes top-sorted-abs R (h # l)
  shows (top-sorted-abs R l)
  <proof>

```

8.2 The Weightiest Path Function (wlp)

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

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

fun *wlp* **where**

```

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

```

— NOTE name shortened.

definition *geq_arg* **where**

$$geq_arg\ f \equiv (\forall x\ y. (x \leq f\ x\ y) \wedge (y \leq f\ x\ y))$$

lemma *individual-weight-less-eq-lp*:

```

fixes w :: 'a ⇒ nat
assumes geq_arg g
shows (w x ≤ wlp R w g f x l)
⟨proof⟩

```

lemma *lp-geq-lp-from-successor*:

```

fixes vtx1 and f g :: nat ⇒ nat ⇒ nat
assumes geq_arg f geq_arg g (∀ vtx. ListMem vtx G ⟶ ¬R vtx vtx) R vtx2 vtx1
  ListMem vtx1 G top-sorted-abs R G
shows (f (w vtx2) (wlp R w g f vtx1 G) ≤ (wlp R w g f vtx2 G))
⟨proof⟩

```

definition *increasing* **where**

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

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

```

  (increasing f)
  ⟹ (increasing g)
  ⟹ (∀ y. ListMem y l ⟶ w1 y ≤ w2 y)
  ⟹ (w1 x ≤ w2 x)
  ⟹ (wlp R w1 g f x l ≤ wlp R w2 g f x l)

```

<proof>

lemma *wlp-congruence-rule*:

fixes $l1\ l2\ R1\ R2\ w1\ w2\ g1\ g2\ f1\ f2\ x1\ x2$

assumes $(l1 = l2) (\forall y. ListMem\ y\ l2 \longrightarrow (R1\ x1\ y = R2\ x2\ y))$

$(\forall y. ListMem\ y\ l2 \longrightarrow (R1\ y\ x1 = R2\ y\ x2)) (w1\ x1 = w2\ x2)$

$(\forall y1\ y2. (y1 = y2) \longrightarrow (f1\ (w1\ x1)\ y1 = f2\ (w2\ x2)\ y2))$

$(\forall y1\ y2\ z1\ z2. (y1 = y2) \wedge (z1 = z2) \longrightarrow ((g1\ (f1\ (w1\ x1)\ y1)\ z1) = (g2\ (f2\ (w2\ x2)\ y2)\ z2)))$

$(\forall x\ y. ListMem\ x\ l2 \wedge ListMem\ y\ l2 \longrightarrow (R1\ x\ y = R2\ x\ y))$

$(\forall x. ListMem\ x\ l2 \longrightarrow (w1\ x = w2\ x))$

$(\forall x\ y\ z. ListMem\ x\ l2 \longrightarrow (g1\ (f1\ (w1\ x)\ y)\ z = g2\ (f2\ (w2\ x)\ y)\ z))$

$(\forall x\ y. ListMem\ x\ l2 \longrightarrow (f1\ (w1\ x)\ y = f2\ (w1\ x)\ y))$

shows $((wlp\ R1\ w1\ g1\ f1\ x1\ l1) = (wlp\ R2\ w2\ g2\ f2\ x2\ l2))$

<proof>

lemma *wlp-ite-weights*:

fixes x

assumes $\forall y. ListMem\ y\ l1 \longrightarrow P\ y\ P\ x$

shows $((wlp\ R\ (\lambda y. if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l1) = (wlp\ R\ w1\ g\ f\ x\ l1))$

<proof>

lemma *map-wlp-ite-weights*:

$(\forall x. ListMem\ x\ l1 \longrightarrow P\ x)$

$\implies (\forall x. ListMem\ x\ l2 \longrightarrow P\ x)$

$\implies ($

$map\ (\lambda x. wlp\ R\ (\lambda y. if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l1)\ l2$

$= map\ (\lambda x. wlp\ R\ w1\ g\ f\ x\ l1)\ l2$

$)$

<proof>

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

<proof>

lemma *img-wlp-ite-weights*:

$(\forall x. ListMem\ x\ l \longrightarrow P\ x)$

$\implies (\forall x. x \in s \longrightarrow P\ x)$

$\implies ($

$(\lambda x. wlp\ R\ (\lambda y. if\ P\ y\ then\ w1\ y\ else\ w2\ y)\ g\ f\ x\ l) \text{ ' } s$

$= (\lambda x. wlp\ R\ w1\ g\ f\ x\ l) \text{ ' } s$

$)$

<proof>

```

end
theory AcycSspace
  imports
    FactoredSystem
    ActionSeqProcess
    SystemAbstraction
    Acyclicity
    FmapUtils
begin

```

9 Acyclic State Spaces

value (*state-successors* (*prob-proj* *PROB* *vs*))

definition *S*

```

  where S vs lss PROB s  $\equiv$  wlp
    ( $\lambda x y. y \in (\text{state-successors } (\text{prob-proj } \text{PROB } \text{vs}) x)$ )
    ( $\lambda s. \text{problem-plan-bound } (\text{snapshot } \text{PROB } s)$ )
    ( $\text{max} :: \text{nat} \Rightarrow \text{nat} \Rightarrow \text{nat}$ ) ( $\lambda x y. x + y + 1$ ) 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) ≠ fmrestrict-set
vs s
  then state-succ s a # vars-change as vs (state-succ s a)
  else vars-change as vs (state-succ s a)
)

```

lemma *vars-change-cat*:

fixes *s*

shows

```

  vars-change (as1 @ as2) vs s
= (vars-change as1 vs s @ vars-change as2 vs (exec-plan s as1))

```

<proof>

lemma *empty-change-no-change*:

fixes *s*

assumes (*vars-change as vs s = []*)

shows (*fmrestrict-set vs (exec-plan s as) = fmrestrict-set vs s*)

<proof>

lemma *zero-change-imp-all-effects-submap*:

fixes *s s'*

assumes (*vars-change as vs s = []*) (*sat-precond-as s as*) (*ListMem b as*)

```

  (fmrestrict-set vs s = fmrestrict-set vs s')

```

shows $(fmrestrict\text{-}set\ vs\ (snd\ b) \subseteq_f\ fmrestrict\text{-}set\ vs\ s')$
 $\langle proof \rangle$

lemma *zero-change-imp-all-preconds-submap:*

fixes $s\ s'$
assumes $(vars\text{-}change\ as\ vs\ s = \square)$ $(sat\text{-}precond\text{-}as\ s\ as)$ $(ListMem\ b\ as)$
 $(fmrestrict\text{-}set\ vs\ s = fmrestrict\text{-}set\ vs\ s')$
shows $(fmrestrict\text{-}set\ vs\ (fst\ b) \subseteq_f\ fmrestrict\text{-}set\ vs\ s')$
 $\langle proof \rangle$

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

assumes $(as \in valid\text{-}plans\ PROB)$ $(sat\text{-}precond\text{-}as\ s\ as)$ $(vars\text{-}change\ as\ vs\ s = \square)$
shows $(as \in valid\text{-}plans\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))$
 $\langle proof \rangle$

lemma *no-vs-change-obtain-snapshot-bound-1st-step:*

fixes $PROB :: 'a\ problem$
assumes $finite\ PROB$ $(vars\text{-}change\ as\ vs\ s = \square)$ $(sat\text{-}precond\text{-}as\ s\ as)$
 $(s \in valid\text{-}states\ PROB)$ $(as \in valid\text{-}plans\ PROB)$
shows $(\exists\ as'.$
 $($
 $exec\text{-}plan\ (fmrestrict\text{-}set\ (prob\text{-}dom\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))\ s)$
 as
 $=\ exec\text{-}plan\ (fmrestrict\text{-}set\ (prob\text{-}dom\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))$
 $s)\ as'$
 $)$
 $\wedge\ (subseq\ as'\ as)$
 $\wedge\ (length\ as' \leq\ problem\text{-}plan\text{-}bound\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))$
 $)$
 $\langle proof \rangle$

lemma *no-vs-change-obtain-snapshot-bound-2nd-step:*

fixes $PROB :: 'a\ problem$
assumes $finite\ PROB$ $(vars\text{-}change\ as\ vs\ s = \square)$ $(sat\text{-}precond\text{-}as\ s\ as)$
 $(s \in valid\text{-}states\ PROB)$ $(as \in valid\text{-}plans\ PROB)$
shows $(\exists\ as'.$
 $($
 $exec\text{-}plan\ (fmrestrict\text{-}set\ (prob\text{-}dom\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))\ s)$
 as
 $=\ exec\text{-}plan\ (fmrestrict\text{-}set\ (prob\text{-}dom\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))$
 $s)\ as'$
 $)$
 $\wedge\ (subseq\ as'\ as)$
 $\wedge\ (sat\text{-}precond\text{-}as\ s\ as')$
 $\wedge\ (length\ as' \leq\ problem\text{-}plan\text{-}bound\ (snapshot\ PROB\ (fmrestrict\text{-}set\ vs\ s)))$
 $)$
 $\langle proof \rangle$

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 ($\exists as'$.

(
 $fmrestrict\text{-}set$ (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as*)
 $= fmrestrict\text{-}set$ (*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>

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*)

$fmrestrict\text{-}set$ (*prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*))) (*exec-plan s as*) =
 $fmrestrict\text{-}set$ (*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

$fmrestrict\text{-}set$ (*fmdom'* (*exec-plan s as*) – *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))

(*exec-plan s as*)

$= fmrestrict\text{-}set$ (*fmdom'* (*exec-plan s as*) – *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))

s

$\wedge fmrestrict\text{-}set$ (*fmdom'* (*exec-plan s as'*) – *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))

(*exec-plan s as'*)

$= fmrestrict\text{-}set$ (*fmdom'* (*exec-plan s as'*) – *prob-dom* (*snapshot PROB* (*fmrestrict-set vs s*)))

s

<proof>

lemma *no-vs-change-snapshot-s-vs-is-valid-bound:*

fixes *PROB* :: 'a problem

assumes *finite PROB* (*vars-change as vs s = []*) (*no-effectless-act as*)

(*sat-precond-as s as*) (*s ∈ valid-states PROB*) (*as ∈ valid-plans PROB*)

shows ($\exists as'$.

(*exec-plan s as = exec-plan s as'*)

\wedge (*subseq as' as*)

\wedge (*length as' ≤ problem-plan-bound* (*snapshot PROB* (*fmrestrict-set vs s*)))

)

<proof>

lemma *snapshot-bound-leq-S:*

shows

$problem-plan-bound (snapshot\ PROB\ (fmrestrict-set\ vs\ s))$
 $\leq S\ vs\ lss\ PROB\ (fmrestrict-set\ vs\ s)$

<proof>

lemma *S-geq-S-succ-plus-ell:*

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

$(top-sorted-abs\ (\lambda x\ y.\ y \in state-successors\ (prob-proj\ PROB\ vs)\ x)\ lss)$

$(s' \in state-successors\ (prob-proj\ PROB\ vs)\ s)\ (set\ lss = valid-states\ (prob-proj\ PROB\ vs))$

shows (

$problem-plan-bound (snapshot\ PROB\ (fmrestrict-set\ vs\ s))$

$+ S\ vs\ lss\ PROB\ (fmrestrict-set\ vs\ s')$

$+ (1 :: nat)$

$\leq S\ vs\ lss\ PROB\ (fmrestrict-set\ vs\ s)$

)

<proof>

lemma *vars-change-cons:*

fixes $s\ s'$

assumes $(vars-change\ as\ vs\ s = (s' \# ss))$

shows $(\exists as1\ act\ as2.$

$(as = as1\ @\ (act \# as2))$

$\wedge (vars-change\ as1\ vs\ s = [])$

$\wedge (state-succ\ (exec-plan\ s\ as1)\ act = s')$

$\wedge (vars-change\ as2\ vs\ (state-succ\ (exec-plan\ s\ as1)\ act) = ss)$

)

<proof>

lemma *vars-change-cons-2:*

fixes $s\ s'$

assumes $(vars-change\ as\ vs\ s = (s' \# ss))$

shows $(fmrestrict-set\ vs\ s' \neq fmrestrict-set\ vs\ s)$

<proof>

lemma *problem-plan-bound-S-bound-1st-step:*

fixes $PROB :: 'a\ problem$

assumes $finite\ PROB\ (top-sorted-abs\ (\lambda x\ y.\ y \in state-successors\ (prob-proj\ PROB\ vs)\ x)\ lss)$

$(set\ lss = valid-states\ (prob-proj\ PROB\ vs))\ (s \in valid-states\ PROB)$

$(as \in valid-plans\ PROB)\ (no-effectless-act\ as)\ (sat-precond-as\ s\ as)$

shows $(\exists as'.$

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

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

$\wedge (length\ as' \leq S\ vs\ lss\ PROB\ (fmrestrict-set\ vs\ s))$

)

<proof>
lemma *problem-plan-bound-S-bound-2nd-step*:
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 *S vs lss PROB* (fmrestrict-set *vs s*))
)
 <proof>
lemma *S-in-MPLS-leq-2-pow-n*:
assumes *finite* (*PROB* :: 'a problem)
 (top-sorted-abs ($\lambda x y. y \in \text{state-successors } (\text{prob-proj } \text{PROB } vs) x$) *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>
lemma *problem-plan-bound-S-bound*:
fixes *PROB* :: 'a problem
assumes *finite* *PROB* (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*))
shows
 problem-plan-bound *PROB*
 \leq Sup {*S vs lss PROB* (*s'* :: 'a state) | *s'*. *s'* \in valid-states (prob-proj *PROB* *vs*)}
 <proof>

9.1 State Space Acyclicity

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

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

definition *sspace-DAG* where
sspace-DAG *PROB lss* \equiv (
 (set *lss* = valid-states *PROB*)
 \wedge (top-sorted-abs ($\lambda x y. y \in \text{state-successors } \text{PROB } x$) *lss*)

)

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

assumes *finite* (*PROB* :: 'a problem) (*sspace-DAG* (*prob-proj* *PROB* *vs*) *lss*)

(*s* ∈ *valid-states* *PROB*) (*as* ∈ *valid-plans* *PROB*)

shows (\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*))

)

<proof>

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

theorem *problem-plan-bound-S-bound-thesis:*

assumes *finite* (*PROB* :: 'a problem) (*sspace-DAG* (*prob-proj* *PROB* *vs*) *lss*)

shows (

problem-plan-bound *PROB*

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

)

<proof>

end

References

- [1] M. Abdulaziz, C. Gretton, and M. Norrish. A State Space Acyclicity Property for Exponentially Tighter Plan Length Bounds. In *International Conference on Automated Planning and Scheduling (ICAPS)*. AAAI, 2017.
- [2] M. Abdulaziz, M. Norrish, and C. Gretton. Formally verified algorithms for upper-bounding state space diameters. *Journal of Automated Reasoning*, pages 1–36, 2018.