We present an executable formally verified SAT encoding of classical AI planning that is based on the encodings by Kautz and Selman [2] and the one by Rintanen et al. [3]. The encoding was experimentally tested and shown to be usable for reasonably sized standard AI planning benchmarks. We also use it as a reference to test a state-of-the-art SAT-based planner, showing that it sometimes falsely claims that problems have no solutions of certain lengths. The formalisation in this submission was described in an independent publication [1].

Contents

1 State-Variable Representation 3
2 STRIPS Representation 3
3 STRIPS Semantics 5
   3.1 Serial Plan Execution Semantics 5
   3.2 Parallel Plan Semantics 15
   3.3 Serializable Parallel Plans 46
   3.4 Auxiliary lemmas about STRIPS 58
4 SAS+ Representation 59
5 SAS+ Semantics 64
   5.1 Serial Execution Semantics 64
   5.2 Parallel Execution Semantics 66
   5.3 Serializable Parallel Plans 77
   5.4 Auxiliary lemmas on SAS+ 83
6 SAS+/STRIPS Equivalence 85
   6.1 Translation of SAS+ Problems to STRIPS Problems 85
   6.2 Equivalence of SAS+ and STRIPS 166

*Author names are alphabetically ordered.
# The Basic SATPlan Encoding

7.1 Encoding Function Definitions .................................. 177
7.2 Decoding Function Definitions .................................. 181
7.3 Soundness of the Basic SATPlan Algorithm .................. 213
7.4 Completeness ...................................................... 251

# Serializable SATPlan Encodings

8.1 Soundness ......................................................... 299
8.2 Completeness ...................................................... 302

# SAT-Solving of SAS+ Problems

9 ................................................................. 309

# Adding Noop actions to the SAS+ problem

10 .............................................................. 311

# Proving Equivalence of SAS+ representation and Fast-Downward’s Multi-Valued Problem Representation

11.1 Translating Fast-Downward’s representation to SAS+ ... 313
11.2 Translating SAS+ representation to Fast-Downward’s ... 329
11.3 SAT encoding works for Fast-Downward’s representation ... 335

# DIMACS-like semantics for CNF formulae

12.1 Going from Formulae to DIMACS-like CNF ................. 341

# Code Generation

13 .............................................................. 347
theory State-Variable-Representation

Propositional-Proof-Systems.CNF
begin

1 State-Variable Representation

Moving on to the Isabelle implementation of state-variable representation, we first add a more concrete representation of states using Isabelle maps. To this end, we add a type synonym for maps of variables to values. Since maps can be conveniently constructed from lists of assignments—i.e. pairs \((v, a) :: \text{variable} \times \text{domain}\)—we also add a corresponding type synonym.

type-synonym \((\text{variable}, \text{domain})\) state = \text{variable} \rightarrow \text{domain}

type-synonym \((\text{variable}, \text{domain})\) assignment = \text{variable} \times \text{domain}

Effects and effect condition (see ??) are implemented in a straightforward manner using a datatype with constructors for each effect type.

type-synonym \((\text{variable}, \text{domain})\) Effect = \((\text{variable} \times \text{domain})\) list

end

theory STRIPS-Representation
imports State-Variable-Representation
begin

2 STRIPS Representation

We start by declaring a record for STRIPS operators. This which allows us to define a data type and automatically generated selector operations. ¹

The record specification given below closely resembles the canonical representation of STRIPS operators with fields corresponding to precondition, add effects as well as delete effects.

record \((\text{variable})\) strips-operator =
precondition-of :: \text{variable} list
add-effects-of :: \text{variable} list
delete-effects-of :: \text{variable} list

— This constructor function is sometimes a more descriptive and replacement for the record syntax and can moreover be helpful if the record syntax leads to type ambiguity.

¹For the full reference on records see [?, 11.6, pp.260-265]
abbreviation operator-for
:: 'variable list ⇒ 'variable list ⇒ 'variable list ⇒ 'variable strips-operator
where operator-for prec add delete ≡
   precondition-of = prec
   , add-effects-of = add
   , delete-effects-of = delete []

definition to-precondition
:: 'variable strips-operator ⇒ ('variable, bool) assignment list
where to-precondition op ≡ map (λv. (v, True)) (precondition-of op)

definition to-effect
:: 'variable strips-operator ⇒ ('variable, bool) Effect
where to-effect op = [(v, True). v ← add-effects-of op] @ [(v, False). v ←
   delete-effects-of op]

Similar to the operator definition, we use a record to represent STRIPS
problems and specify fields for the variables, operators, as well as the initial
and goal state.

record ('variable) strips-problem =
   variables-of :: 'variable list ((-V) [1000] 999)
   operators-of :: 'variable strips-operator list ((-O) [1000] 999)
   initial-of :: 'variable strips-state ((-I) [1000] 999)
   goal-of :: 'variable strips-state ((-G) [1000] 999)

value stop

As discussed in ??, the effect of a STRIPS operator can be normalized to
a conjunction of atomic effects. We can therefore construct the successor
state by simply converting the list of add effects to assignments to True
resp. converting the list of delete effect to a list of assignments to False and
then adding the map corresponding to the assignments to the given state s
as shown below in definition ??.

definition execute-operator
:: 'variable strips-state
⇒ 'variable strips-state
⇒ 'variable strips-state (infixl ⇒ 52)
where execute-operator s op
≡ s ++ map-of (effect-to-assignments op)
end

theory STRIPS-Semantics
imports STRIPS-Representation
   List-Supplement
   Map-Supplement
begin

2 Function effect_to_assignments converts the operator effect to a list of assignments.
3 STRIPS Semantics

Having provided a concrete implementation of STRIPS and a corresponding locale strips, we can now continue to define the semantics of serial and parallel STRIPS plan execution (see ?? and ??).

3.1 Serial Plan Execution Semantics

Serial plan execution is defined by primitive recursion on the plan. Definition ?? returns the given state if the state argument does not satisfy the precondition of the next operator in the plan. Otherwise it executes the rest of the plan on the successor state $s \Rightarrow op$ of the given state and operator.

```
primrec execute-serial-plan
  where
  execute-serial-plan s [] = s
  | execute-serial-plan s (op # ops) = (if is-operator-applicable-in s op
    then execute-serial-plan (execute-operator s op) ops
    else s)
```

Analogously, a STRIPS trace either returns the singleton list containing only the given state in case the precondition of the next operator in the plan is not satisfied. Otherwise, the given state is prepended to trace of the rest of the plan for the successor state of executing the next operator on the given state.

```
fun trace-serial-plan-strips :: 'variable strips-state ⇒ 'variable strips-plan ⇒ 'variable strips-state list
  where
  trace-serial-plan-strips s [] = [s]
  | trace-serial-plan-strips s (op # ops) = s # (if is-operator-applicable-in s op
    then trace-serial-plan-strips (execute-operator s op) ops
    else [])
```

Finally, a serial solution is a plan which transforms a given problems initial state into its goal state and for which all operators are elements of the problem’s operator list.

```
definition is-serial-solution-for-problem
  where
  is-serial-solution-for-problem Π π ≡ (goal-of Π) ⊆m execute-serial-plan (initial-of Π) π
  ∧ list-all (λop. ListMem op (operators-of Π)) π
```

lemma is-valid-problem-strips-initial-of-dom: fixes II:: 'a strips-problem
  assumes is-valid-problem-strips II
  shows dom ((II)₁) = set ((II)Y)
  proof –
{  
    let ?I = strips-problem.initial-of Π 
    let ?vs = strips-problem.variables-of Π 
    fix v  
    have ?I v ≠ None ⟷ ListMem v ?vs  
        using assms(1)  
        unfolding is-valid-problem-strips-def  
        by meson  
    hence v ∈ dom ?I ⟷ v ∈ set ?vs  
        using ListMem-iff  
        by fast  
}  
thus ?thesis  
    by auto  
qed 

lemma is-valid-problem-dom-of-goal-state-is:  
fixes Π:: 'a strips-problem  
assumes is-valid-problem-strips Π  
shows dom ((Π)G) ⊆ set ((Π)V)  
proof −  
    let ?vs = strips-problem.variables-of Π 
    let ?G = strips-problem.goal-of Π  
    have nb: ∀ v. ?G v ≠ None ⟷ ListMem v ?vs  
        using assms(1)  
        unfolding is-valid-problem-strips-def  
        by meson  
    {  
        fix v  
        assume v ∈ dom ?G  
        then have ?G v ≠ None  
            by blast  
        hence v ∈ set ?vs  
            using nb  
            unfolding ListMem-iff  
            by blast  
    }  
thus ?thesis  
    by auto  
qed 

lemma is-valid-problem-strips-operator-variable-sets:  
fixes Π:: 'a strips-problem  
assumes is-valid-problem-strips Π  
and op ∈ set ((Π)O)  
shows set (precondition-of op) ⊆ set ((Π)V)  
    and set (add-effects-of op) ⊆ set ((Π)V)  
    and set (delete-effects-of op) ⊆ set ((Π)V)  
    and disjnt (set (add-effects-of op)) (set (delete-effects-of op))
proof –
let \(?ops\) = strips-problem.operators-of \(\Pi\)
and \(?vs\) = strips-problem.variables-of \(\Pi\)
have list-all \((\text{is-valid-operator-strips} \ \Pi) \ ?ops\)
  using assms(1)
unfolding is-valid-problem-strips-def
by meson
moreover have \(\forall \ v \in \ \text{set} \ \text{(precondition-of } op) \ v \in \ \text{set} \ \text{(}(\Pi)V)\)
and \(\forall \ v \in \ \text{set} \ \text{(add-effects-of } op) \ v \in \ \text{set} \ \text{(}(\Pi)V)\)
and \(\forall \ v \in \ \text{set} \ \text{(delete-effects-of } op) \ v \notin \ \text{set} \ \text{(delete-effects-of } op)\)
and \(\forall \ v \in \ \text{set} \ \text{(delete-effects-of } op) \ v \notin \ \text{set} \ \text{(add-effects-of } op)\)
using assms(2)
calculation
unfolding is-valid-operator-strips-def list-all-iff Let-def ListMem-iff
using variables-of-def
by auto+
ultimately show \(\text{set} \ \text{(precondition-of } op) \subseteq \ \text{set} \ \text{(}(\Pi)V)\)
and \(\text{set} \ \text{(add-effects-of } op) \subseteq \ \text{set} \ \text{(}(\Pi)V)\)
and \(\text{set} \ \text{(delete-effects-of } op) \subseteq \ \text{set} \ \text{(}(\Pi)V)\)
and \(\text{disjnt} \ \text{(set} \ \text{(add-effects-of } op)) \ \text{(set} \ \text{(delete-effects-of } op)\)
unfolding disjnt-def
by fast+
qed

lemma effect-to-assignments-i:
assumes as = effect-to-assignments \(op\)
shows as = \(\text{map} \ (\lambda v. \ (v, \ \text{True})) \ \text{(add-effects-of } op)\)
\(@\ \text{map} \ (\lambda v. \ (v, \ \text{False})) \ \text{(delete-effects-of } op)\)
using assms
unfolding effect-to-assignments-def effect--strips-def
by auto

lemma effect-to-assignments-ii:
— NOTE effect-to-assignments can be simplified drastically given that only atomic
effects and the add-effects as well as delete-effects lists only consist of variables.
assumes as = effect-to-assignments \(op\)
obtains as1 as2
where as = as1 \(\oplus\) as2
  and as1 = map \(\lambda v. \ (v, \ \text{True})\) \ (add-effects-of \(op\))
  and as2 = map \(\lambda v. \ (v, \ \text{False})\) \ (delete-effects-of \(op\))
by \((\text{simp add: assms effect--strips-def effect-to-assignments-def})\)

— NOTE Show that for every variable \(v\) in either the add effect list or the delete
effect list, there exists an assignment in representing setting \(v\) to true respectively
setting \(v\) to false. Note that the first assumption amounts to saying that the add
effect list is not empty. This also requires us to split lemma into two separate
lemmas since add and delete effect lists are not required to both contain at least
one variable simultaneously.
lemma effect-to-assignments-iii-a:
fixes $v$
assumes $v \in \text{set (add-effects-of op)}$
and $\text{as} = \text{effect-to-assignments op}$
obtains $\text{a where a } \in \text{set as a} = (v, \text{True})$
proof
let $\text{add-assignments} = (\lambda v. (v, \text{True})) \triangledown \text{set (add-effects-of op)}$
let $\text{delete-assignments} = (\lambda v. (v, \text{False})) \triangledown \text{set (delete-effects-of op)}$
obtain $\text{as}_1 \text{as}_2$
where $a1: \text{as} = \text{as}_1 \circ \text{as}_2$
and $a2: \text{as}_1 = \text{map} (\lambda v. (v, \text{True})) \text{add-effects-of op}$
and $a3: \text{as}_2 = \text{map} (\lambda v. (v, \text{False})) \text{delete-effects-of op}$
using assms(2) effect-to-assignments-ii
by blast
then have $b: \text{set as}$
= $\text{add-assignments} \cup \text{delete-assignments}$
by auto
— NOTE The existence of an assignment as proposed can be shown by the following sequence of set inclusions.
\{ 
from $b$ have $\text{add-assignments} \subseteq \text{set as}$
by blast
moreover have $\{(v, \text{True})\} \subseteq \text{add-assignments}$
using assms(1) $a2$
by blast
ultimately have $\exists a. a \in \text{set as} \land a = (v, \text{True})$
by blast
\}
then show $\exists \text{thesis}$
using that
by blast
qed

lemma effect-to-assignments-iii-b:
— NOTE This proof is symmetrical to the one above.
fixes $v$
assumes $v \in \text{set (delete-effects-of op)}$
and $\text{as} = \text{effect-to-assignments op}$
obtains $\text{a where a } \in \text{set as a} = (v, \text{False})$
proof
let $\text{delete-assignments} = (\lambda v. (v, \text{True})) \triangledown \text{set (add-effects-of op)}$
let $\text{add-assignments} = (\lambda v. (v, \text{False})) \triangledown \text{set (delete-effects-of op)}$
obtain $\text{as}_1 \text{as}_2$
where $a1: \text{as} = \text{as}_1 \circ \text{as}_2$
and $a2: \text{as}_1 = \text{map} (\lambda v. (v, \text{True})) \text{add-effects-of op}$
and $a3: \text{as}_2 = \text{map} (\lambda v. (v, \text{False})) \text{delete-effects-of op}$
using assms(2) effect-to-assignments-ii
by blast
then have $b: \text{set as}$
= $\text{add-assignments} \cup \text{delete-assignments}$
by auto

— NOTE The existence of an assignment as proposed can be shown by the following sequence of set inclusions.

\[
\begin{align*}
&\text{from } b \text{ have } ?\text{delete-assignments} \subseteq \text{set as} \\
&\text{by blast} \\
&\text{moreover have } \{ (v, False) \} \subseteq ?\text{delete-assignments} \\
&\text{using } \text{assms}(1) \ a \ 2 \\
&\text{by blast} \\
&\text{ultimately have } \exists a. a \in \text{set as } \land a = (v, False) \\
&\text{by blast} \\
\end{align*}
\]

then show ?thesis \\
using that \\
by blast \\
qed

lemma effect--strips-i:

fixes op 
assumes \( e = \text{effect--strips op} \) 
obtains \( es_1, es_2 \) 
where \( e = (es_1 \circ es_2) \) 
and \( es_1 = \text{map } (\lambda v. (v, True)) (\text{add-effects-of op}) \) 
and \( es_2 = \text{map } (\lambda v. (v, False)) (\text{delete-effects-of op}) \)
proof 
  obtain \( es_1, es_2 \) where \( a: e = (es_1 \circ es_2) \) 
  and \( b: es_1 = \text{map } (\lambda v. (v, True)) (\text{add-effects-of op}) \) 
  and \( c: es_2 = \text{map } (\lambda v. (v, False)) (\text{delete-effects-of op}) \) 
  using \( \text{assms}(1) \) 
  unfolding effect--strips-def 
  by blast 
  then show ?thesis 
  using that 
  by force 
qed

lemma effect--strips-ii:

fixes op 
assumes \( e = \text{ConjunctiveEffect } (es_1 \circ es_2) \) 
and \( es_1 = \text{map } (\lambda v. (v, True)) (\text{add-effects-of op}) \) 
and \( es_2 = \text{map } (\lambda v. (v, False)) (\text{delete-effects-of op}) \) 
shows \( \forall v \in \text{set } (\text{add-effects-of op}). (\exists e' \in \text{set } es_1. e' = (v, True)) \) 
and \( \forall v \in \text{set } (\text{delete-effects-of op}). (\exists e' \in \text{set } es_2. e' = (v, False)) \)
proof 
— NOTE Show that for each variable \( v \) in the add effect list, we can obtain an atomic effect with true value.
fix \( v \) 
\{ 
  assume a: \( v \in \text{set } (\text{add-effects-of op}) \)
have set es₁ = (λv. (v, True)) ' set (add-effects-of op)
using assms(2) List.set-map
by auto
then obtain e'
  where e' ∈ set es₁
  and e' = (λv. (v, True)) v
  using a
  by blast
then have ∃e' ∈ set es₁. e' = (v, True)
by blast }
thus v ∈ set (add-effects-of op) ⇒ ∃e' ∈ set es₁. e' = (v, True)
by fast
— NOTE the proof is symmetrical to the one above: for each variable v in the
delete effect list, we can obtain an atomic effect with v being false.

next
{
fix v
assume a: v ∈ set (delete-effects-of op)
have set es₂ = (λv. (v, False)) ' set (delete-effects-of op)
  using assms(3) List.set-map
  by force
then obtain e''
  where e'' ∈ set es₂
  and e'' = (λv. (v, False)) v
  using a
  by blast
then have ∃e'' ∈ set es₂. e'' = (v, False)
  by blast
}
thus ∀v ∈ set (delete-effects-of op). ∃e'∈set es₂. e' = (v, False)
by fast
qed

lemma map-of-constant-assignments-dom:
— NOTE ancillary lemma used in the proof below.
assumes m = map-of (map (λv. (v, d)) vs)
shows dom m = set vs
proof
  let ?vs' = map (λv. (v, d)) vs
  have dom m = fst ' set ?vs'
    using assms(1) dom-map-of-comp-image-fst
    by metis
  moreover have fst ' set ?vs' = set vs
    by force
  ultimately show ?thesis
    by argo
qed
lemma effect-strips-iii-a:
assumes $s' = (s \gg op)$
shows $\forall v. v \in \text{set} (\text{add-effects-of op}) \Rightarrow s' v = \text{Some True}$
proof
  fix $v$
assume $a: v \in \text{set} (\text{add-effects-of op})$
let $?as = \text{effect-to-assignments op}$
obtain $as_1 as_2$ where $b: ?as = as_1 @ as_2$
  and $c: as_1 = \text{map} (\lambda v. (v, \text{True})) (\text{add-effects-of op})$
  and $as_2 = \text{map} (\lambda v. (v, \text{False})) (\text{delete-effects-of op})$
using effect-to-assignments-ii
by blast
have $d: \text{map-of } ?as = \text{map-of } as_2 ++ \text{map-of } as_1$
  using $b \text{ Map.map-of-append}$
  by auto
{
  -- TODO refactor?
  let $?vs = \text{add-effects-of op}$
  have $?vs \neq []$
    using $a$
    by force
  then have $\text{dom} (\text{map-of } as_1) = \text{set} (\text{add-effects-of op})$
    using $c \text{ map-of-constant-assignments-dom}$
    by metis
  then have $v \in \text{dom} (\text{map-of } as_1)$
    using $a$
    by blast
  then have $\text{map-of } ?as v = \text{map-of } as_1 v$
    using $d$
    by force
}
moreover {
  let $?f = \lambda_. \text{True}$
  from $c$ have $\text{map-of } as_1 = (\text{Some } ?f) \mid (\text{set} (\text{add-effects-of op}))$
    using map-of-map-restrict
    by fast
  then have $\text{map-of } as_1 v = \text{Some True}$
    using $a$
    by auto
}
moreover have $s' = s ++ \text{map-of } as_2 ++ \text{map-of } as_1$
  using assms(1)
  unfolding execute-operator-def
  using $b$
  by simp
ultimately show $s' v = \text{Some True}$
  by simp
qed
lemma effect-strips-iii-b:
assumes $s' = (s \Rightarrow op)$
shows $\forall v. v \in \text{set (delete-effects-of op)} \land v \notin \text{set (add-effects-of op)} \implies s' v = \text{Some False}$
proof (auto)
  fix v
  assume a1: $v \notin \text{set (add-effects-of op)}$ and a2: $v \in \text{set (delete-effects-of op)}$
  let ?as = effect-to-assignments op
  obtain as1 as2 where b: ?as = as1 @ as2
    and c: as1 = map ($\lambda v. (v, \text{True})$) (add-effects-of op)
    and d: as2 = map ($\lambda v. (v, \text{False})$) (delete-effects-of op)
  using effect-to-assignments-ii
  by blast
  have c: map-of ?as = map-of as2 ++ map-of as1
    using b Map.map-of-append
    by auto
  { have dom (map-of as1) = set (add-effects-of op)
      using c map-of-constant-assignments-dom
      by metis
      then have $v \notin \text{dom (map-of as1)}$
        using a1
        by blast
    } note f = this
  { let ?vs = delete-effects-of op
    have ?vs \neq []
      using a2
      by force
    then have dom (map-of as2) = set ?vs
      using d map-of-constant-assignments-dom
      by metis
    } note g = this
  { have $s' = s ++ \text{map-of as2} ++ \text{map-of as1}$
      using assms(1)
      unfolding execute-operator-def
      using b
      by simp
    thm f map-add-dom-app-simps(3)[OF f, of s ++ map-of as2]
    moreover have $s' v = (s ++ \text{map-of as2}) v$
      using calculation map-add-dom-app-simps(3)[OF f, of s ++ map-of as2]
      by blast
    moreover have $v \in \text{dom (map-of as2)}$
      using a2 g
      by argo
    ultimately have $s' v = \text{map-of as2} v$
      by fastforce
  }
moreover

\{ 
  let \( ?f = \lambda - \cdot \text{False} \)

  from \( d \) have map-of \( a_{s_2} \) = (Some \( \circ ?f \)) \( \mid (\text{set} (\text{delete-effects-of op})) \)
  using map-of-map-restrict
  by fast

  then have map-of \( a_{s_2} \) \( v = \text{Some False} \)
  using \( a_2 \)
  by force
\}

ultimately show \( s' v = \text{Some False} \)
by argo
qed

lemma effect-strips-iii-c:
assumes \( s' = (s \gg op) \)
shows \( \forall v. \, v \notin \text{set} (\text{add-effects-of op}) \land v \notin \text{set} (\text{delete-effects-of op}) \Rightarrow s' v = s v \)
proof (auto)
fix \( v \)
assume \( a1: v \notin \text{set} (\text{add-effects-of op}) \) and \( a2: v \notin \text{set} (\text{delete-effects-of op}) \)
let \( ?as = \text{effect-to-assignments op} \)
obtain \( a_{s_1} \, a_{s_2} \) where \( ?as = a_{s_1} @ a_{s_2} \)
  and \( c: \, a_{s_1} = (\lambda v. (v, \text{True})) (\text{add-effects-of op}) \)
  and \( d: \, a_{s_2} = (\lambda v. (v, \text{False})) (\text{delete-effects-of op}) \)
  using effect-to-assignments-ii
  by blast
have \( c: \, \text{map-of} \, ?as = \text{map-of} \, a_{s_2} \, + + \, \text{map-of} \, a_{s_1} \)
using \( b \) Map.map-of-append
by auto
\{ 
  have \( \text{dom} (\text{map-of} \, a_{s_1}) = \text{set} (\text{add-effects-of op}) \)
    using \( c \) map-of-constant-assignments-dom
    by metis
  then have \( v \notin \text{dom} (\text{map-of} \, a_{s_1}) \)
    using \( a1 \)
    by blast
\}
moreover \{ 
  have \( \text{dom} (\text{map-of} \, a_{s_2}) = \text{set} (\text{delete-effects-of op}) \)
    using \( d \) map-of-constant-assignments-dom
    by metis
  then have \( v \notin \text{dom} (\text{map-of} \, a_{s_2}) \)
    using \( a2 \)
    by blast
\}
ultimately show \( s' v = s v \)
using assms(1)
unfolding execute-operator-def
by (simp add: b map-add-dom-app-simps(3))
qed

The following theorem combines three preceding sublemmas which show that the following properties hold for the successor state \( s' \equiv \text{execute-operator op s} \) obtained by executing an operator \( \text{op} \) in a state \( s \)\(^3\):

- every add effect is satisfied in \( s' \) (sublemma); and,
- every delete effect that is not also an add effect is not satisfied in \( s' \) (sublemma); and finally
- the state remains unchanged—i.e. \( s' \, v = s \, v \)—for all variables which are neither an add effect nor a delete effect.

**Theorem** operator-effect--strips:

**Assumes** \( s' = (s \, \triangleright \, \text{op}) \)

**Shows**

\[
\forall v. \quad v \in \text{set (add-effects-of op)} \quad \implies s' \, v = \text{Some True}
\]

and

\[
\forall v. \quad v \notin \text{set (add-effects-of op)} \land v \in \text{set (delete-effects-of op)} \quad \implies s' \, v = \text{Some False}
\]

and

\[
\forall v. \quad v \notin \text{set (add-effects-of op)} \land v \notin \text{set (delete-effects-of op)} \quad \implies s' \, v = s \, v
\]

**Proof** (auto)

**Show** \( \forall v. \quad v \in \text{set (add-effects-of op)} \quad \implies s' \, v = \text{Some True} \)

**Using** asms effect--strips-iii-a

**By** fast

**Next**

**Show** \( \forall v. \quad v \notin \text{set (add-effects-of op)} \quad \implies v \in \text{set (delete-effects-of op)} \quad \implies s' \, v = \text{Some False} \)

**Using** asms effect--strips-iii-b

**By** fast

**Next**

**Show** \( \forall v. \quad v \notin \text{set (add-effects-of op)} \quad \implies v \notin \text{set (delete-effects-of op)} \quad \implies s' \, v = s \, v \)

---

\(^3\)Lemmas effect__strips_iii_a, effect__strips_iii_b, and effect__strips_iii_c (not shown).
using assms effect--strips-iii-c
by metis
qed

3.2 Parallel Plan Semantics

definition are-all-operators-applicable s ops
≡ list-all (λop. is-operator-applicable-in s op) ops

definition are-operator-effects-consistent op1 op2 ≡ let
add1 = add-effects-of op1
; add2 = add-effects-of op2
; del1 = delete-effects-of op1
; del2 = delete-effects-of op2
in ¬list-ex (λv. list-ex ((=) v) del2) add1 ∧ ¬list-ex (λv. list-ex ((=) v) add2) del1

definition are-all-operator-effects-consistent ops ≡ list-all (λop. list-all (are-operator-effects-consistent op) ops) ops

definition execute-parallel-operator :: 'variable strips-state ⇒ 'variable strips-operator list ⇒ 'variable strips-state
where execute-parallel-operator s ops
≡ foldl (++) s (map (map-of ◦ effect-to-assignments) ops)

The parallel STRIPS execution semantics is defined in similar way as the serial STRIPS execution semantics. However, the applicability test is lifted to parallel operators and we additionally test for operator consistency (which was unnecessary in the serial case).

fun execute-parallel-plan :: 'variable strips-state ⇒ 'variable strips-parallel-plan ⇒ 'variable strips-state
where execute-parallel-plan s [] = s
| execute-parallel-plan s (ops # opss) = (if
are-all-operators-applicable s ops
∧ are-all-operator-effects-consistent ops
then execute-parallel-plan (execute-parallel-operator s ops) opss
else s)

definition are-operators-interfering op1 op2
≡ list-ex (λv. list-ex ((=) v) (delete-effects-of op1)) (precondition-of op2)
∨ list-ex (λv. list-ex ((=) v) (precondition-of op1)) (delete-effects-of op2)

primrec are-all-operators-non-interfering :: 'variable strips-operator list ⇒ bool

15
where are-all-operators-non-interfering [] = True
| are-all-operators-non-interfering (op ≠ ops)
  = (list-all (λop'. ¬are-operators-interfering op op') ops
     ∧ are-all-operators-non-interfering ops)

Since traces mirror the execution semantics, the same is true for the definition of parallel STRIPS plan traces.

fun trace-parallel-plan-strips
:: 'variable strips-state ⇒ 'variable strips-parallel-plan ⇒ 'variable strips-state list
where trace-parallel-plan-strips s [] = [s]
  | trace-parallel-plan-strips s (ops ≠ opss) = s ≠ (if
    are-all-operators-applicable s ops
    ∧ are-all-operator-effects-consistent ops
    then trace-parallel-plan-strips (execute-parallel-operator s ops) opss
    else [])

Similarly, the definition of parallel solutions requires that the parallel execution semantics transforms the initial problem into the goal state of the problem and that every operator of every parallel operator in the parallel plan is an operator that is defined in the problem description.

definition is-parallel-solution-for-problem
where is-parallel-solution-for-problem Π π ≡ (strips-problem.goal-of Π) ⊆ m
  execute-parallel-plan
  (strips-problem.initial-of Π) π
  ∧ list-all (λops. list-all (λop.
    ListMem op (strips-problem.operators-of Π)) ops) π

lemma are-all-operators-applicable-set:
are-all-operators-applicable s ops
  ←→ (∀ op ∈ set ops. ∀ v ∈ set (precondition-of op). s v = Some True)
unfolding are-all-operators-applicable-def
  STRIPS-Representation.is-operator-applicable-in-def list-all-iff
by presburger

lemma are-all-operators-applicable-cons:
assumes are-all-operators-applicable s (op ≠ ops)
shows is-operator-applicable-in s op
  and are-all-operators-applicable s ops
proof –
  from assms have a: list-all (λop. is-operator-applicable-in s op) (op ≠ ops)
    unfolding are-all-operators-applicable-def is-operator-applicable-in-def
  STRIPS-Representation.is-operator-applicable-in-def
by blast
then have is-operator-applicable-in s op
  by fastforce
moreover {
from a have list-all (λop. is-operator-applicable-in s op) ops  
  by simp
then have are-all-operators-applicable s ops  
  using are-all-operators-applicable-def is-operator-applicable-in-def  
  STRIPS-Representation.is-operator-applicable-in-def  
  by blast
}
ultimately show is-operator-applicable-in s op  
  and are-all-operators-applicable s ops  
  by fast+
qed

lemma are-operator-effects-consistent-set:
  assumes op_1 ∈ set ops  
  and op_2 ∈ set ops  
  shows are-operator-effects-consistent op_1 op_2  
  = (set (add-effects-of op_1) ∩ set (delete-effects-of op_2) = {})  
  ∧ set (delete-effects-of op_1) ∩ set (add-effects-of op_2) = {})
proof −
  have (¬list-ex (λv. list-ex ((=) v) (delete-effects-of op_2)) (add-effects-of op_1))  
  = (set (add-effects-of op_1) ∩ set (delete-effects-of op_2) = {})  
  using list-ex-intersection[of delete-effects-of op_2 add-effects-of op_1]  
  by meson
  moreover have (¬list-ex (λv. list-ex ((=) v) (add-effects-of op_2)) (delete-effects-of op_1))  
  = (set (delete-effects-of op_1) ∩ set (add-effects-of op_2) = {})  
  using list-ex-intersection[of add-effects-of op_2 delete-effects-of op_1]  
  by meson
  ultimately show are-operator-effects-consistent op_1 op_2  
  = (set (add-effects-of op_1) ∩ set (delete-effects-of op_2) = {})  
  ∧ set (delete-effects-of op_1) ∩ set (add-effects-of op_2) = {})
  unfolding are-operator-effects-consistent-def  
  by presburger
qed

lemma are-all-operator-effects-consistent-set:
  are-all-operator-effects-consistent ops  
  ⟷ (∀ op_1 ∈ set ops. ∀ op_2 ∈ set ops.  
  (set (add-effects-of op_1) ∩ set (delete-effects-of op_2) = {})  
  ∧ (set (delete-effects-of op_1) ∩ set (add-effects-of op_2) = {}))
proof −
  {  
    fix op_1 op_2  
    assume op_1 ∈ set ops and op_2 ∈ set ops  
    hence are-operator-effects-consistent op_1 op_2  
    = (set (add-effects-of op_1) ∩ set (delete-effects-of op_2) = {})  
    ∧ set (delete-effects-of op_1) ∩ set (add-effects-of op_2) = {})
    using are-operator-effects-consistent-set[of op_1 ops op_2]  
    by fast
thus \$\text{thesis}\$

unfolding \$\text{are-all-operator-effects-consistent-def}\$
by \$\text{force}\$

qed

lemma \$\text{are-all-effects-consistent-tail}:\$
assumes \$\text{are-all-operator-effects-consistent (op \# ops)}\$
shows \$\text{are-all-operator-effects-consistent ops}\$
proof -
from \$\text{assms}\$
have \$a: \text{list-all (} \lambda \text{op'. list-all (are-operator-effects-consistent op')} (\text{Cons op ops}) (\text{Cons op ops})\$
unfolding \$\text{are-all-operator-effects-consistent-def}\$
by \$\text{blast}\$
then have \$b-1: \text{list-all (are-operator-effects-consistent op) (op \# ops)}\$
and \$b-2: \text{list-all (} \lambda \text{op'. list-all (are-operator-effects-consistent op')} (op \# ops)\$
ops
by \$\text{force}\$
then have \$\text{list-all (are-operator-effects-consistent op) ops}\$
by \$\text{simp}\$
moreover
{
{
fix \text{z}
assume \$z \in \text{set (Cons op ops)}\$
and \$\text{list-all (are-operator-effects-consistent z) (op \# ops)}\$
then have \$\text{list-all (are-operator-effects-consistent z) ops}\$
by \$\text{auto}\$
}
then have \$\text{list-all (} \lambda \text{op'. list-all (are-operator-effects-consistent op')} (op \# ops)\$
using \$\text{list.pred-mono-strong[of}\$
(\$\lambda \text{op'. list-all (are-operator-effects-consistent op')} (op \# ops)\$
Cons op ops (\$\lambda \text{op'. list-all (are-operator-effects-consistent op')} (op \# ops))\$
] \$a\$
by \$\text{fastforce}\$
}
ultimately have \$\text{list-all (are-operator-effects-consistent op) ops} \wedge \text{list-all (} \lambda \text{op'. list-all (are-operator-effects-consistent op')} (op \# ops)\$
by \$\text{blast}\$
then show \$\text{thesis}\$
using \$\text{are-all-operator-effects-consistent-def}\$
by \$\text{fast}\$
qed

lemma \$\text{are-all-operators-non-interfering-tail}:\$
assumes \$\text{are-all-operators-non-interfering (op \# ops)}\$
shows \$\text{are-all-operators-non-interfering ops}\$
using \$\text{assms}\$
unfolding are-all-operators-non-interfering-def
by simp

lemma are-operators-interfering-symmetric:
  assumes are-operators-interfering op1 op2
  shows are-operators-interfering op2 op1
  using assms
  unfolding are-operators-interfering-def list-ex-iff
  by fast

— A small technical characterizing operator lists with property .
We show that pairs of distinct operators which interfere with one another
cannot both be contained in the corresponding operator set.

lemma are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs:
  assumes are-all-operators-non-interfering ops
  and are-operators-interfering op op1 op2
  and op1 ≠ op2
  shows op1 /∈ set ops ∨ op2 /∈ set ops
  using assms
proof (induction ops)
case (Cons op ops)
  thm Cons.IH[OF - Cons.prems(2, 3)]
  have nb1: ∀ op' ∈ set ops. ~are-operators-interfering op op'
    and nb2: are-all-operators-non-interfering ops
    using Cons.prems(1)
    unfolding are-all-operators-non-interfering.simps(2) list-all-iff
    by blast+
  then consider (A) op = op1
    | (B) op = op2
    | (C) op1 ≠ op1 ∧ op ≠ op2
    by blast
thus ?case
proof (cases)
case A
  { assume op2 ∈ set (op ≠ ops)
    then have op2 ∈ set ops
      using Cons.prems(3) A
      by force
    then have ~are-operators-interfering op1 op2
      using nb2 A
      by fastforce
    hence False
      using Cons.prems(2).+
  }
thus ?thesis
  by blast
next
  case B
\{ 
  \textbf{assume} \( op_1 \in \text{set} \ (\text{op} \neq \text{ops}) \)  
  \textbf{then have} \( op_1 \in \text{set} \ \text{ops} \)  
  \textbf{using} \( \text{Cons} . \text{prems}(3) \ B \)  
  \textbf{by} \ \text{force}  
  \textbf{then have} \( \neg \text{are-operators-interfering} \ \text{op}_1 \ \text{op}_2 \)  
  \textbf{using} \( \text{nb}_1 \ B \ \text{are-operators-interfering-symmetric} \)  
  \textbf{by} \ \text{blast}  
  \textbf{hence} \ False  
  \textbf{using} \( \text{Cons} . \text{prems}(2) \).  
\}

\textbf{thus} \ ?\text{thesis}  
\textbf{by} \ \text{blast}  
\textbf{next}  
\textbf{case} \ C  
\textbf{thus} \ ?\text{thesis}  
\textbf{using} \( \text{Cons} . \text{IH} \{ \text{OF} \ \text{nb}_2 \ \text{Cons} . \text{prems}(2, 3) \} \)  
\textbf{by} \ \text{force}  
\textbf{qed}  
\textbf{qed} \ \text{simp}

\textbf{lemma} \text{execute-parallel-plan-precondition-cons-i} :  
\textbf{fixes} \( s \:: \ (\text{variable}, \ \text{bool}) \ \text{state} \)  
\textbf{assumes} \( \neg \text{are-operators-interfering} \ \text{op} \ \text{op}' \)  
\textbf{and} \( \text{is-operator-applicable-in} \ \text{s} \ \text{op} \)  
\textbf{and} \( \text{is-operator-applicable-in} \ \text{s} \ \text{op}' \)  
\textbf{shows} \( \text{is-operator-applicable-in} \ (\text{s} \ ++ \ \text{map-of} \ (\text{effect-to-assignments} \ \text{op})) \ \text{op}' \)  
\textbf{proof} \ –  
\textbf{let} \( ?s' = \text{s} \ ++ \ \text{map-of} \ (\text{effect-to-assignments} \ \text{op}) \)  
\text{— TODO slightly hackish to exploit the definition of execute-operator, but}  
\text{we otherwise have to rewrite theorem operator-effect--strips (which is a todo as of}  
\text{now).}  
\{  
  \textbf{have} \( a : \ ?s' = \text{s} \ \gg \ \text{op} \)  
  \textbf{by} \ (\text{simp add: execute-operator-def})  
  \textbf{then have} \( \bigwedge v . \ v \in \text{set} \ (\text{add-effects-of} \ \text{op}) \implies \ ?s' \ v = \text{Some True} \)  
  \text{and} \( \bigwedge v . \ v \notin \text{set} \ (\text{add-effects-of} \ \text{op}) \land v \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \implies ?s' \ v = \text{Some False} \)  
  \text{and} \( \bigwedge v . \ v \notin \text{set} \ (\text{add-effects-of} \ \text{op}) \land v \notin \text{set} \ (\text{delete-effects-of} \ \text{op}) \implies ?s' \ v = \text{s v} \)  
  \textbf{using} \ \text{operator-effect--strips}  
  \textbf{by} \ \text{metis+}  
\}  
\textbf{note} \( a = \text{this} \)  
\text{— TODO refactor lemma not-have-interference-set.}  
\{  
  \textbf{fix} \ v  
  \textbf{assume} \( \alpha : \ v \in \text{set} \ (\text{precondition-of} \ \text{op}') \)
{ fix \ v \\
have \ \neg \text{list-ex} \ ( (\ = ) \ v) \ (\text{delete-effects-of} \ \text{op}) \\
= \text{list-all} \ (\ \lambda v'. \ \neg v = v') \ (\text{delete-effects-of} \ \text{op}) \\
using \ \text{not-list-ex-equals-list-all-not}[ \\
\ \ \ \ \text{where} \ P= (=) \ v \ \text{and} \ xs=\text{delete-effects-of} \ \text{op}] \\
by \ \text{blast} \\
} \\
moreover { 
from \ \text{assms}(1) 
have \ \neg \text{list-ex} \ (\ \lambda v. \ \text{list-ex} \ ( (\ = ) \ v) \ (\text{delete-effects-of} \ \text{op})) \ (\text{precondition-of} \ \text{op}') 
unfolding \ \text{are-operators-interfering-def} 
by \ \text{blast} 
then have \ \text{list-all} \ (\ \lambda v. \ \neg \text{list-ex} \ ( (\ = ) \ v) \ (\text{delete-effects-of} \ \text{op})) \ (\text{precondition-of} \ \text{op}') 
using \ \text{not-list-ex-equals-list-all-not}[ \\
\ \ \ \ \text{where} \ P=\lambda v. \ \text{list-ex} \ ( (\ = ) \ v) \ (\text{delete-effects-of} \ \text{op}) \ \text{and} \ xs=\text{precondition-of} \ \text{op}'] 
by \ \text{blast} 
} \\
ultimately have \ \beta: 
\text{list-all} \ (\ \lambda v. \ \text{list-all} \ (\ \lambda v'. \ \neg v = v')) \ (\text{delete-effects-of} \ \text{op}) 
by \ \text{presburger} 
moreover { 
fix \ \ v 
have \ \text{list-all} \ (\ \lambda v'. \ \neg v = v') \ (\text{delete-effects-of} \ \text{op}) 
= (\forall v' \in \ \text{set} \ (\text{delete-effects-of} \ \text{op}). \ \neg v = v') 
using \ \text{list-all-iff}[ \ \text{where} \ P=\lambda v'. \ \neg v = v' \ \text{and} \ x=\text{delete-effects-of} \ \text{op}] 
} \\
ultimately have \ \forall v \in \ \text{set} \ (\text{precondition-of} \ \text{op}'). \ \forall v' \in \ \text{set} \ (\text{delete-effects-of} \ \text{op}). \ \neg v = v' 
using \ \beta \ \text{list-all-iff}[ \\
\ \ \ \ \text{where} \ P=\lambda v. \ \text{list-all} \ (\ \lambda v'. \ \neg v = v') \ (\text{delete-effects-of} \ \text{op}) 
\ \ \ \text{and} \ x=\text{precondition-of} \ \text{op}'] 
by \ \text{presburger} 
then have \ v \notin \ \text{set} \ (\text{delete-effects-of} \ \text{op}) 
using \ \alpha 
by \ \text{fast} 
} \\
\text{note} \ \ b = \ this 

{ 
fix \ \ v 
assume \ a: \ v \in \ \text{set} \ (\text{precondition-of} \ \text{op}') 
have \ \text{list-all} \ (\ \lambda v. \ s \ v = \ \text{Some True}) \ (\text{precondition-of} \ \text{op}') 
using \ \text{assms}(3) 
unfolding \ \text{is-operator-applicable-in-def} 
\ \ \ \text{STRIPS-Representation.is-operator-applicable-in-def} 
by \ \text{presburger} 
then have \ \forall v \in \ \text{set} \ (\text{precondition-of} \ \text{op}'). \ s \ v = \ \text{Some True} 

21
using list-all-iff[where \( P = \lambda v. s \ v = \text{Some True} \) and \( x = \text{precondition-of} \) op]

by blast
then have \( s \ v = \text{Some True} \)
  using a
  by blast
}

note \( c = \text{this} \)
{
fix \( v \)
assume \( d \): \( v \in \text{set} \ (\text{precondition-of} \ op) \)
then have \( ?s' \ v = \text{Some True} \)
proof \( (\text{cases} \ v \in \text{set} \ (\text{add-effects-of} \ op)) \)
case True
  then show \( ?\text{thesis} \)
    using a
    by blast
next
case \( c \): False
  then show \( ?\text{thesis} \)
proof \( (\text{cases} \ v \in \text{set} \ (\text{delete-effects-of} \ op)) \)
case True
  then show \( ?\text{thesis} \)
    using assms(1) b d
    by fast
next
case False
  then have \( ?s' \ v = s \ v \)
    using a e
    by blast
  then show \( ?\text{thesis} \)
    using c d
    by presburger
qed
qed

then have list-all \( (\lambda v. \ ?s' \ v = \text{Some True}) \ (\text{precondition-of} \ op) \)
  using list-all-iff[where \( P = \lambda v. \ ?s' \ v = \text{Some True} \) and \( x = \text{precondition-of} \) op]
  by blast
  then show \( ?\text{thesis} \)
unfolding is-operator-applicable-in-def
  STRIPS-Representation.is-operator-applicable-in-def
  by auto
qed

— The third assumption \( \text{are-all-operators-non-interfering} \ (a \ # \ \text{ops}) \)” is not part of the precondition of but is required for the proof of the subgoal hat applicable is maintained.
lemma execute-parallel-plan-precondition-cons:
  fixes a :: 'variable strips-operator
  assumes are-all-operators-applicable s (a ≠ ops)
    and are-all-operator-effects-consistent (a ≠ ops)
    and are-all-operators-non-interfering (a ≠ ops)
  shows are-all-operators-applicable (s ++ map-of (effect-to-assignments a)) ops
    and are-all-operator-effects-consistent ops
    and are-all-operators-non-interfering ops
  using are-all-effects-consistent-tail[OF assms(2)]
    are-all-operators-non-interfering-tail[OF assms(3)]
  proof (–
    let ?s' = s ++ map-of (effect-to-assignments a)
    have nb1: ∀ op ∈ set (a ≠ ops). is-operator-applicable-in s op
      using assms(1) are-all-operators-applicable-set
      unfolding are-all-operators-applicable-def is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def list-all-iff
      by blast
    have nb2: ∀ op ∈ set ops. ¬are-operators-interfering a op
      using assms(3)
      unfolding are-all-operators-non-interfering-def list-all-iff
      by simp
    have nb3: is-operator-applicable-in s a
      using assms(1) are-all-operators-applicable-set
      unfolding are-all-operators-applicable-def is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def list-all-iff
      by force
    { fix op
      assume op-in-ops: op ∈ set ops
      hence is-operator-applicable-in ?s' op
        using execute-parallel-plan-precondition-cons-i[of a op] nb1 nb2 nb3
        by force
    }
    then show are-all-operators-applicable ?s' ops
      unfolding are-all-operators-applicable-def list-all-iff
      is-operator-applicable-in-def
      by blast
  qed

lemma execute-parallel-operator-cons[simp]:
  execute-parallel-operator s (op ≠ ops)
  = execute-parallel-operator (s ++ map-of (effect-to-assignments op)) ops
  unfolding execute-parallel-operator-def
  by simp

lemma execute-parallel-operator-cons-equals:
  assumes are-all-operators-applicable s (a ≠ ops)
    and are-all-operator-effects-consistent (a ≠ ops)
    and are-all-operators-non-interfering (a ≠ ops)
shows execute-parallel-operator s (a # ops)
= execute-parallel-operator (s ++ map-of (effect-to-assignments a)) ops
proof –
let ?s’ = s ++ map-of (effect-to-assignments a)
{ from assms(1, 2)
  have execute-parallel-operator s (Cons a ops)
    = foldl (++) s (map (map-of o effect-to-assignments) (Cons a ops))
    unfolding execute-parallel-operator-def
    by presburger
  also have ... = foldl (++) (?s’)
    (map (map-of o effect-to-assignments) ops)
    by auto
  finally have execute-parallel-operator s (Cons a ops)
    = foldl (++) (?s’)
    (map (map-of o effect-to-assignments) ops)
    using execute-parallel-operator-def
    by blast
}
— NOTE the precondition of for a # ops is also true for the tail list and state ?s’ as shown in lemma. Hence the precondition for the r.h.s. of the goal also holds.
moreover have execute-parallel-operator ?s’ ops
= foldl (++) (s ++ (map-of o effect-to-assignments) a)
  (map (map-of o effect-to-assignments) ops)
  by (simp add: execute-parallel-operator-cons-equals)
ultimately show ?thesis
  by force
qed

— We show here that following the lemma above, executing one operator of a parallel operator can be replaced by a (single) STRIPS operator execution.
corollary execute-parallel-operator-cons-equals-corollary:
  assumes are-all-operators-applicable s (a # ops)
  shows execute-parallel-operator s (a # ops)
  = execute-parallel-operator (s >> a) ops
proof –
let ?s’ = s ++ map-of (effect-to-assignments a)
from assms
  have execute-parallel-operator s (a # ops)
    = execute-parallel-operator (s ++ map-of (effect-to-assignments a)) ops
    using execute-parallel-operator-cons-equals
    by simp
  moreover have ?s’ = s >> a
    unfolding execute-operator-def
    by simp
  ultimately show ?thesis
    by argo
qed
lemma effect-to-assignments-simp[simp]:
effect-to-assignments op
= map (λv. (v, True)) (add-effects-of op) @ map (λv. (v, False)) (delete-effects-of op)
by (simp add: effect-to-assignments-i)

lemma effect-to-assignments-set-is[simp]:
set (effect-to-assignments op) = { ((v, a), True) | v a. (v, a) ∈ set (add-effects-of op) }
∪ { ((v, a), False) | v a. (v, a) ∈ set (delete-effects-of op) }
proof –
obtain as where effect--strips op = as
and as = map (λv. (v, True)) (add-effects-of op)
@ map (λv. (v, False)) (delete-effects-of op)
unfolding effect--strips-def
by blast
moreover have as
= map (λv. (v, True)) (add-effects-of op) @ map (λv. (v, False)) (delete-effects-of op)
using calculation(2)
unfolding map-append map-map comp-apply
by auto
moreover have effect-to-assignments op = as
unfolding effect-to-assignments-def calculation(1, 2)
by auto
ultimately show thesis
unfolding set-map
by auto
qed

corollary effect-to-assignments-construction-from-function-graph:
assumes set (add-effects-of op) ∩ set (delete-effects-of op) = { }
shows effect-to-assignments op = map
(λv. (v, if ListMem v (add-effects-of op) then True else False))
(add-effects-of op @ delete-effects-of op)
and effect-to-assignments op = map
(λv. (v, if ListMem v (delete-effects-of op) then False else True))
(add-effects-of op @ delete-effects-of op)
proof –
let ?f = λv. (v, if ListMem v (add-effects-of op) then True else False)
and ?g = λv. (v, if ListMem v (delete-effects-of op) then False else True)
{ have map ?f (add-effects-of op @ delete-effects-of op)
= map ?f (add-effects-of op) @ map ?f (delete-effects-of op)
using map-append
by fast
— TODO slow.
hence effect-to-assignments op = map ?f (add-effects-of op @ delete-effects-of op)
using ListMem-iff assms
by fastforce
}

moreover {
  have map ?g (add-effects-of op @ delete-effects-of op)
    = map ?g (add-effects-of op) @ map ?g (delete-effects-of op)
  using map-append
  by fast
  — TODO slow.
hence effect-to-assignments op = map ?g (add-effects-of op @ delete-effects-of op)
  using ListMem-iff assms
  by fastforce
}

ultimately show effect-to-assignments op = map (λ v. if ListMem v (add-effects-of op) then True else False)
  (add-effects-of op @ delete-effects-of op)
and effect-to-assignments op = map (λ v. if ListMem v (delete-effects-of op) then False else True)
  (add-effects-of op @ delete-effects-of op)
by blast+
qed

corollary map-of-effect-to-assignments-is-none-if:
assumes ¬ v ∈ set (add-effects-of op)
  and ¬ v ∈ set (delete-effects-of op)
shows map-of (effect-to-assignments op) v = None
proof –
let ¯l = effect-to-assignments op
{
  have set ¯l = { (v, True) | v. v ∈ set (add-effects-of op) }
    ∪ { (v, False) | v. v ∈ set (delete-effects-of op) }
  by auto
  then have fst ' set ¯l
    = (fst ' {(v, True) | v. v ∈ set (add-effects-of op)})
      ∪ (fst ' {(v, False) | v. v ∈ set (delete-effects-of op)})
  using image-Un[of fst {(v, True) | v. v ∈ set (add-effects-of op)}]
    {(v, False) | v. v ∈ set (delete-effects-of op)}]
  by presburger
  — TODO slow.
  also have ... = (fst ' (λv. (v, True)) ' set (add-effects-of op))
    ∪ (fst ' (λv. (v, False)) ' set (delete-effects-of op))
  using setcompr-eq-image[of λv. (v, True) λv. v ∈ set (add-effects-of op)]
    setcompr-eq-image[of λv. (v, False) λv. v ∈ set (delete-effects-of op)]
  by simp
  — TODO slow.
  also have ... = id ' set (add-effects-of op) ∪ id ' set (delete-effects-of op)
  by force
  — TODO slow.
  finally have fst ' set ¯l = set (add-effects-of op) ∪ set (delete-effects-of op)
by auto
hence \( v \notin \text{fst ' } \text{set } \ ?l \)
using assms(1, 2)
by blast
}
thus \( \text{thesis} \)
using map-of-eq-None-iff(of \( ?l \ v \))
by blast
qed

lemma execute-parallel-operator-positive-effect-if-i:
assumes are-all-operators-applicable s ops
and are-all-operator-effects-consistent ops
and \( op \in \text{set ops} \)
and \( v \in \text{set (add-effects-of op)} \)
shows \( \text{map-of (effect-to-assignments op)} \ v = \text{Some True} \)
proof
let \( ?f = \lambda x. \text{if ListMem } x \text{ (add-effects-of op) then True else False} \)
and \( ?l' = \text{map (} \lambda v. \text{if ListMem } v \text{ (add-effects-of op) then True else False})) \)
(\( \text{add-effects-of op} \cap \text{delete-effects-of op} \))
have set (add-effects-of op) \( \neq \) \{\}
using assms(4)
by fastforce
moreover \{ 
have set (add-effects-of op) \( \cap \) set (delete-effects-of op) = \{\}
using are-all-operator-effects-consistent-set assms(2, 3)
by fast
moreover have effect-to-assignments op = \( ?l' \)
using effect-to-assignments-construction-from-function-graph(1) calculation
by fast
ultimately have \( \text{map-of (effect-to-assignments op)} = \text{map-of } \ ?l' \)
by argo
\}
ultimately have \( \text{map-of (effect-to-assignments op)} \ v = \text{Some (?f v)} \)
using Map-Supplement.map-of-from-function-graph-is-some-if[\( \text{of - - ?f, OF - assms(4)} \)]
by simp
thus \( \text{thesis} \)
using ListMem-iff assms(4)
by metis
qed

lemma execute-parallel-operator-positive-effect-if:
fixes ops
assumes are-all-operators-applicable s ops
and are-all-operator-effects-consistent ops
and \( op \in \text{set ops} \)
and \( v \in \text{set (add-effects-of op)} \)
shows execute-parallel-operator s ops \( v = \text{Some True} \)
proof

let \( l = \text{map-of } \circ \text{effect-to-assignments} \) ops

have \( \text{set-l-is: set } l = (\text{map-of } \circ \text{effect-to-assignments}) \ ' \) set ops
  using set-map
  by fastforce

\{ 
  let \( m = (\text{map-of } \circ \text{effect-to-assignments}) \) op
  have \( m \in \text{set } l \)
    using assms(3) set-l-is
    by blast
  moreover have \( m v = \text{Some True} \)
    using execute-parallel-operator-positive-effect-if-i[OF assms]
    by fastforce
  ultimately have \( \exists m \in \text{set } l. m v = \text{Some True} \)
    by blast
\}

moreover 

fix \( m' \)

assume \( m' \in \text{set } l \)
then obtain \( op' \)
  where \( op'-in-set-ops; op' \in \text{set ops} \)
  and \( m'-is: m' = (\text{map-of } \circ \text{effect-to-assignments}) \) op'
  by auto
then have \( \text{set } (\text{add-effects-of op}) \cap \text{set } (\text{delete-effects-of op'}) = \{} \)
  using assms(2, 3) are-all-operator-effects-consistent-set[of ops]
  by blast
then have \( v \notin \text{set } (\text{delete-effects-of op'}) \)
  using assms(4)
  by blast
then consider \( (v-in-set-add-effects) v \in \text{set } (\text{add-effects-of op'}) \)
  \mid \( \text{otherwise} \) \( \neg v \in \text{set } (\text{add-effects-of op'}) \land \neg v \in \text{set } (\text{delete-effects-of op'}) \)
  by blast
hence \( m' v = \text{Some True} \lor m' v = \text{None} \)

proof (cases)
  case \( v-in-set-add-effects \)
    — TODO slow.
    thus \( \text{thesis} \)
      using execute-parallel-operator-positive-effect-if-i[OF assms(1, 2) op'-in-set-ops, of v] m'-is
      by simp
  next
  case otherwise
  then have \( \neg v \in \text{set } (\text{add-effects-of op'}) \)
    and \( \neg v \in \text{set } (\text{delete-effects-of op'}) \)
    by blast
  thus \( \text{thesis} \)
    using map-of-effect-to-assignments-is-none-if[of v op'] m'-is
    by fastforce
qed
ultimately show ?thesis
unfolding execute-parallel-operator-def
using foldl-map-append-is-some-if[of s v True ?l]
by meson

lemma execute-parallel-operator-negative-effect-if-i:
assumes are-all-operators-applicable s ops
and are-all-operator-effects-consistent ops
and op ∈ set ops
and v ∈ set (delete-effects-of op)
shows map-of (effect-to-assignments op) v = Some False

proof –
let ?f = λx. if ListMem x (delete-effects-of op) then False else True
and ?l' = map (λv. (v, if ListMem v (delete-effects-of op) then False else True))
(add-effects-of op @ delete-effects-of op)
have set (delete-effects-of op @ add-effects-of op) ≠ {}
using assms(4)
by fastforce
moreover have v ∈ set (delete-effects-of op @ add-effects-of op)
using assms(4)
by simp
moreover {
  have set (add-effects-of op) ∩ set (delete-effects-of op) = {}
    using are-all-operator-effects-consistent-set assms(2, 3)
  by fast
  moreover have effect-to-assignments op = ?l'
    using effect-to-assignments-construction-from-function-graph(2) calculation
    by blast
  ultimately have map-of (effect-to-assignments op) = map-of ?l'
    by argo
}
ultimately have map-of (effect-to-assignments op) v = Some (?f v)
using Map-Supplement.map-of-from-function-graph-is-some-if[of add-effects-of op @ delete-effects-of op v ?f]
by force
thus ?thesis
unfolding ListMem-iff
by presburger
qed

lemma execute-parallel-operator-negative-effect-if:
assumes are-all-operators-applicable s ops
and are-all-operator-effects-consistent ops
and op ∈ set ops

} — TODO slow.
and $v \in \text{set} \ (\text{delete-effects-of op})$

shows $\text{execute-parallel-operator} \ s \ \text{ops} \ v = \text{Some False}$

proof ~

let $\ ?l = \text{map} \ (\text{map-of} \circ \text{effect-to-assignments}) \ \text{ops}$

have $\text{set-l-is}: \ ?l = (\text{map-of} \circ \text{effect-to-assignments}) \ l \ \text{ops}$

using $\text{set-map}$

by fastforce

{ 
  let $\ ?m = (\text{map-of} \circ \text{effect-to-assignments}) \ \text{op}$
  have $\ ?m \in \ ?l$
    using assms(3) set-l-is
    by blast
  moreover have $\ ?m \ v = \text{Some False}$
    using $\text{execute-parallel-operator-negative-effect-if-i}[OF \ \text{assms}]$
    by fastforce
  ultimately have $\exists \ m \in \ ?l. \ m \ v = \text{Some False}$
    by blast
}

moreover { 
  fix $m'$
  assume $m' \in \ ?l$
  then obtain $\text{op'}$
    where $\text{op'}$-$\text{in-set-ops}: \ \text{op'} \in \ \text{ops}$
      and $m'$-$\text{is}: \ m' = (\text{map-of} \circ \text{effect-to-assignments}) \ \text{op'}$
      by auto
  then have $\text{set} \ (\text{delete-effects-of op}) \cap \ \text{set} \ (\text{add-effects-of \text{op'}}) = \{\}$
    using assms(2, 3) are-all-operator-effects-consistent-set[of \ \text{ops}]
    by blast
  then have $v \notin \ \text{set} \ (\text{add-effects-of \text{op'}})$
    using assms(4)
    by blast
  then consider $(\text{v-in-set-delete-effects}) \ v \in \ \text{set} \ (\text{delete-effects-of \text{op'}})$
    $| \ (\text{otherwise}) \ \neg v \in \ \text{set} \ (\text{add-effects-of \text{op'}}) \land \neg v \in \ \text{set} \ (\text{delete-effects-of \text{op'}})$
    by blast
  hence $m' \ v = \text{Some False} \lor m' \ v = \text{None}$

  proof (cases)
    case $\text{v-in-set-delete-effects}$
    — TODO slow.
    thus $\ ?thesis$
      using $\text{execute-parallel-operator-negative-effect-if-i}[OF \ \text{assms}(1, 2) \ \text{op'}$-$\text{in-set-ops}, \ \text{of} \ v] \ m'$-$\text{is}$
      by simp
  next
    case otherwise
    then have $\neg v \in \ \text{set} \ (\text{add-effects-of \text{op'}})$
      and $\neg v \in \ \text{set} \ (\text{delete-effects-of \text{op'}})$
      by blast+
    thus $\ ?thesis$
      using $\text{map-of-effect-to-assignments-is-none-if}[\text{of} \ v \ \text{op'}] \ m'$-$\text{is}$
by fastforce

qed

— TODO slow.

ultimately show thesis

unfolding execute-parallel-operator-def

using foldl-map-append-is-some-if[of s v False ?l]

by meson

qed

lemma execute-parallel-operator-no-effect-if:

assumes ∀ op ∈ set ops. ¬v ∈ set (add-effects-of op) ∧ ¬v ∈ set (delete-effects-of op)

shows execute-parallel-operator s ops v = s v

using assms

unfolding execute-parallel-operator-def

proof (induction ops arbitrary: s)

case (Cons a ops)

let ?f = map-of ◦ effect-to-assignments

{ have v ∉ set (add-effects-of a) ∧ v ∉ set (delete-effects-of a)
  using Cons.prems(1)
  by force
  then have ?f a v = None
  using map-of-effect-to-assignments-is-none-if[of v a]
  by fastforce
  then have v ∉ dom (?f a)
  by blast
  hence (s ++ ?f a) v = s v
  using map-add-dom-app-simps(3)[of v ?f a s]
  by blast
}

moreover { have ∀ op ∈ set ops. v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of op)
  using Cons.prems(1)
  by simp
  hence foldl (++) (s ++ ?f a) (map ?f ops) v = (s ++ ?f a) v
  using Cons.IH[of s ++ ?f a]
  by blast
}

moreover { have map ?f (a # ops) = ?f a # map ?f ops
  by force
  then have foldl (++) s (map ?f (a # ops))
  = foldl (++) (s ++ ?f a) (map ?f ops)
  using foldl-Cons
  by force
}

ultimately show ?case

31
corollary execute-parallel-operators-strips-none-if:
assumes ∀ op ∈ set ops. ¬v ∈ set (add-effects-of op) ∧ ¬v ∈ set (delete-effects-of op)
and s v = None
shows execute-parallel-operator s ops v = None
using execute-parallel-operator-no-effect-if[OF assms(1)] assms(2)
by simp

corollary execute-parallel-operators-strips-none-if-contraposition:
assumes ¬execute-parallel-operator s ops v = None
shows (∃ op ∈ set ops. v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op))
∧ s v ≠ None
proof –
let ?P = (∀ op ∈ set ops. ¬v ∈ set (add-effects-of op) ∧ ¬v ∈ set (delete-effects-of op))
∧ s v = None
and ?Q = execute-parallel-operator s ops v = None
have ?P ⇒ ?Q
using execute-parallel-operators-strips-none-if[of ops v s]
by blast
then have ¬?P
using contrapos-nn[of ?Q ?P]
using assms
by argo
thus ?thesis
by meson
qed

We will now move on to showing the equivalent to theorem in . Under the condition that for a list of operators ops all operators in the corresponding set are applicable in a given state s and all operator effects are consistent, if an operator op exists with op ∈ set ops and with v being an add effect of op, then the successor state

s' ≡ execute-parallel-operator s ops

will evaluate v to true, that is

execute-parallel-operator s ops v = Some True

Symmetrically, if v is a delete effect, we have

execute-parallel-operator s ops v = Some False
under the same condition as for the positive effect. Lastly, if \( v \) is neither an add effect nor a delete effect for any operator in the operator set corresponding to \( \text{ops} \), then the state after parallel operator execution remains unchanged, i.e.

\[
\text{execute-parallel-operator } s \ \text{ops} \ v = s \ v
\]

**Theorem** execute-parallel-operator-effect:

**Assumes** are-all-operators-applicable \( s \ \text{ops} \)

**and** are-all-operator-effects-consistent \( \text{ops} \)

**Shows** \( \text{op} \in \text{set} \ \text{ops} \ \land \ v \in \text{set} \ (\text{add-effects-of } \text{op}) \)

\[\rightarrow \text{execute-parallel-operator } s \ \text{ops} \ v = \text{Some True} \]

\( \text{and} \ (\forall \ \text{op} \in \text{set} \ \text{ops}.

\ v \notin \text{set} \ (\text{add-effects-of } \text{op}) \ \land \ v \notin \text{set} \ (\text{delete-effects-of } \text{op})\)

\[\rightarrow \text{execute-parallel-operator } s \ \text{ops} \ v = \text{Some False} \]

**Using** execute-parallel-operator-positive-effect-if \([\text{OF assms}]\)

execute-parallel-operator-negative-effect-if \([\text{OF assms}]\)

execute-parallel-operator-no-effect-if \([\text{of ops v s}]\)

**By** blast*

**Lemma** is-parallel-solution-for-problem-operator-set:

**Fixes** \( \Pi::'a \text{strips}\text{-problem} \)

**Assumes** is-parallel-solution-for-problem \( \Pi \ \pi \)

**and** \( \text{ops} \in \text{set} \ \pi \)

**and** \( \text{op} \in \text{set} \ \text{ops} \)

**Shows** \( \text{op} \in \text{set} \ ((\Pi)\circ) \)

**Proof**

\[\text{have} \ (\forall \ \text{ops} \in \text{set} \ \pi. \ (\forall \ \text{op} \in \text{set} \ \text{ops}. \ \text{op} \in \text{set} \ (\text{strips}\text{-problem.operators-of } \Pi))} \]

**Using** assms \((1)\)

**Unfolding** is-parallel-solution-for-problem-def list-all-iff ListMem-iff..

**Thus** ?thesis

**Using** assms \((2, 3)\)

**By** fastforce

**Qed**

**Lemma** trace-parallel-plan-strips-not-nil: trace-parallel-plan-strips \( I \ \pi \neq [] \)

**Proof** (cases \( \pi \))

**Case** (Cons a list)

**Then** show ?thesis

**By** (cases are-all-operators-applicable \( I \ (\text{hd} \ \pi) \ \land \ are-all-operator-effects-consistent \ (\text{hd} \ \pi) \))

**Simp**

**Qed**

**Corollary** length-trace-parallel-plan-gt-0[simp]: \( \theta < \text{length} \ (\text{trace-parallel-plan-strips } I \ \pi) \)

33
using trace-parallel-plan-strips-not-nil.. 

**corollary** length-trace-minus-one-lt-length: 
\[ \text{length (trace-parallel-plan-strips } I \pi) - 1 < \text{length (trace-parallel-plan-strips } I \pi) \]

**using** diff-less[OF - length-trace-parallel-plan-gt-0] 
**by auto** 

**lemma** trace-parallel-plan-strips-head-is-initial-state: 
\[ \text{trace-parallel-plan-strips } I \pi ! 0 = I \]

**proof** (cases \( \pi \)) 
\[ \text{case } (\text{Cons } a \text{ list}) \]
**then show** ?thesis 
\[ \text{by } (\text{cases } \text{are-all-operators-applicable } I a \land \text{are-all-operator-effects-consistent } a, \text{simp+}) \]
**qed simp** 

**lemma** trace-parallel-plan-strips-length-gt-one-if: 
\[ \text{assumes } k < \text{length (trace-parallel-plan-strips } I \pi) - 1 \]
**shows** \( 1 < \text{length (trace-parallel-plan-strips } I \pi) \)
**using** assms 
**by linarith** 

— This lemma simply shows that the last element of a trace-parallel-plan-strips execution step \( s \neq \text{trace-parallel-plan-strips } s' \pi \) always is the last element of \( \text{trace-parallel-plan-strips } s' \pi \) since trace-parallel-plan-strips always returns at least a singleton list (even if \( \pi = [] \)).

**lemma** trace-parallel-plan-strips-last-cons-then: 
\[ \text{last } (s \neq \text{trace-parallel-plan-strips } s' \pi) = \text{last } (\text{trace-parallel-plan-strips } s' \pi) \]
**by** (cases \( \pi, \text{simp, force} \)) 

Parallel plan traces have some important properties that we want to confirm before proceeding. Let \( \tau = \text{trace-parallel-plan-strips } I \pi \) be a trace for a parallel plan \( \pi \) with initial state \( I \).

First, all parallel operators \( \text{ops} = \pi ! k \) for any index \( k \) with \( k < \text{length } \tau - 1 \) (meaning that \( k \) is not the index of the last element). must be applicable and their effects must be consistent. Otherwise, the trace would have terminated and \( \text{ops} \) would have been the last element. This would violate the assumption that \( k < \text{length } \tau - 1 \) is not the last index since the index of the last element is \( \text{length } \tau - 1 \). \(^4\)

**lemma** trace-parallel-plan-strips-operator-preconditions: 
\[ \text{assumes } k < \text{length (trace-parallel-plan-strips } I \pi) - 1 \]
**shows** \( \text{are-all-operators-applicable } (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k) \land \text{are-all-operator-effects-consistent } (\pi ! k) \)
**using** assms 

\(^4\)More precisely, the index of the last element is \( \text{length } \tau - 1 \) if \( \tau \) is not empty which is however always true since the trace contains at least the initial state.
proof \(\text{(induction } \pi \text{ arbitrary: } I \ k)\)
— NOTE Base case yields contradiction with assumption and can be left to automation.

case \((\text{Cons } a \, \pi)\)
then show \(?\text{case}\)
proof \((\text{cases are-all-operators-applicable } I \ a \land \text{are-all-operator-effects-consistent } a)\)

case \(\text{True}\)
have \(\text{trace-parallel-plan-strips-cons: trace-parallel-plan-strips } I \ (a \neq \pi) = I \neq \text{trace-parallel-plan-strips } (\text{execute-parallel-operator } I \ a) \, \pi\)
using \(\text{True}\)
by simp
then show \(?\text{thesis}\)
proof \((\text{cases } k)\)

case \(0\)
have \(\text{trace-parallel-plan-strips } I \ (a \neq \pi) ! 0 = I\)
using \(\text{trace-parallel-plan-strips-cons}\)
by simp
moreover have \((a \neq \pi) ! 0 = a\)
by simp
ultimately show \(?\text{thesis}\)
using \(\text{True } 0\)
by presburger

next
case \((\text{Suc } k')\)
let \(?I' = \text{execute-parallel-operator } I \ a\)
have \(\text{trace-parallel-plan-strips } I \ (a \neq \pi) \neq \text{Suc } k' = \text{trace-parallel-plan-strips } ?I' \, \pi \neq k'\)
using \(\text{trace-parallel-plan-strips-cons}\)
by simp
moreover have \((a \neq \pi) \neq \text{Suc } k' = \pi \neq k'\)
by simp
moreover {
have \(\text{length } (\text{trace-parallel-plan-strips } I \ (a \neq \pi)) = 1 + \text{length } (\text{trace-parallel-plan-strips } ?I' \, \pi)\)
unfolding \(\text{trace-parallel-plan-strips-cons}\)
by simp
then have \(k' < \text{length } (\text{trace-parallel-plan-strips } ?I' \, \pi) - 1\)
using \(\text{Suc Cons.prems}\)
by fastforce
hence \(\text{are-all-operators-applicable } (\text{trace-parallel-plan-strips } ?I' \, \pi \neq k')\)
\((\pi \neq k')\)
\(\land \text{are-all-operator-effects-consistent } (\pi \neq k')\)
using \(\text{Cons.IH[of } k'\}\)
by blast
}
ultimately show \(?\text{thesis}\)
using \(\text{Suc}\)
by argo
Another interesting property that we verify below is that elements of the trace store the result of plan prefix execution. This means that for an index \( k \) with 
\[ k < \text{length} (\text{trace-parallel-plan-strips} I \pi) \], the \( k \)-th element of the trace is state reached by executing the plan prefix \( \text{take} k \pi \) consisting of the first \( k \) parallel operators of \( \pi \).

**Lemma** \( \text{trace-parallel-plan-plan-prefix} \):
- **Assumes** \( k < \text{length} (\text{trace-parallel-plan-strips} I \pi) \)
- **Shows** \( \text{trace-parallel-plan-strips} I \pi \! k = \text{execute-parallel-plan} I (\text{take} k \pi) \)
- **Using** \( \text{assms} \)
- **Proof** (induction \( \pi \) arbitrary: \( I \ k \))
  - **Case** \( \text{Cons} \ a \pi \)
    - **Then show** \( ?\text{thesis} \)
      - **Proof** (cases \( \text{are-all-operators-applicable} \ a \) \( \land \text{are-all-operator-effects-consistent} \ a \))
        - **Case** True
          - Let \( ?\sigma = \text{trace-parallel-plan-strips} I (a \# \pi) \)
          - And \( ?I' = \text{execute-parallel-operator} I a \)
          - Have \( ?\sigma\)-equals: \( ?\sigma = I \# \text{trace-parallel-plan-strips} ?I' \pi \)
            - **Using** True
            - **By** auto
          - **Then show** \( ?\text{thesis} \)
            - **Proof** (cases \( k = 0 \))
              - **Case** False
                - Obtain \( k' \) where \( k\)-is-suc-of-\( k' \): \( k = \text{Suc} k' \)
                  - **Using** \( \text{not0-implies-Suc}[\text{OF} \ False] \)
                    - **By** blast
                  - **Then have** \( \text{execute-parallel-plan} I (\text{take} k (a \# \pi)) = \text{execute-parallel-plan} ?I' (\text{take} k' \pi) \)
                    - **Using** True
                      - **By** simp
                - **Moreover have** \( \text{trace-parallel-plan-strips} I (a \# \pi) ! k = \text{trace-parallel-plan-strips} ?I' \pi ! k' \)
                  - **Using** \( ?\sigma\)-equals \( k\)-is-suc-of-\( k' \)
                    - **By** simp
moreover {
  have $k' < \text{length} (\text{trace-parallel-plan-strips} (\text{execute-parallel-operator} I a))$
  using $\text{Cons.prems} \sigma$-equals $k$-is-suc-of-$k'$
  by force
  hence $\text{trace-parallel-plan-strips} ?I' \pi ! k'$
  $= \text{execute-parallel-plan} ?I' (\text{take} k' \pi)$
  using $\text{Cons.IH}[\text{of} k' ?I']$
  by blast
}
ultimately show $?thesis$
by presburger
qed simp

next

case $\text{operator-precondition-violated} : \text{False}$
than show $?thesis$
proof (cases $k = 0$)

case $\text{False}$
then have $\text{trace-parallel-plan-strips} I (a \# \pi) = [I]$
  using $\text{operator-precondition-violated}$
  by force
moreover have $\text{execute-parallel-plan} I (\text{take} k (a \# \pi)) = I$
  using $\text{Cons.prems} \text{operator-precondition-violated}$
  by force
ultimately show $?thesis$
  using $\text{Cons.prems} \text{nth-Cons-0}$
  by auto
qed simp

qed simp

lemma $\text{length-trace-parallel-plan-strips-lte-length-plan-plus-one}$:
shows $\text{length} (\text{trace-parallel-plan-strips} I \pi) \leq \text{length} \pi + 1$
proof (induction $\pi$ arbitrary: $I$)

case (Cons $a \pi$)
then show $?case$
proof (cases $\text{are-all-operators-applicable} I a \land \text{are-all-operator-effects-consistent} a$)

case $\text{True}$
let $?I' = \text{execute-parallel-operator} I a$
{
  have $\text{trace-parallel-plan-strips} I (a \# \pi) = I \# \text{trace-parallel-plan-strips} ?I' \pi$
  using $\text{True}$
  by auto
then have $\text{length} (\text{trace-parallel-plan-strips} I (a \# \pi))$
  $= \text{length} (\text{trace-parallel-plan-strips} ?I' \pi) + 1$
  by simp
}
moreover have \( \text{length} (\text{trace-parallel-plan-strips} ?I' \pi) \leq \text{length} \pi + 1 \)
using Cons.IH[of ?I']
by blast
ultimately have \( \text{length} (\text{trace-parallel-plan-strips} I (a \# \pi)) \leq \text{length} (a \# \pi) + 1 \)
by simp
}
thus \?thesis
by blast
qed auto
qed simp
— Show that \( \pi \) is at least a singleton list.

**lemma** plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements:

**assumes** \( k < \text{length} (\text{trace-parallel-plan-strips} I \pi) - 1 \)

**obtains** \( \text{ops} \pi' \quad \text{where} \quad \pi = \text{ops} \# \pi' \)

**proof —**
let \( ?\tau = \text{trace-parallel-plan-strips} I \pi \)
have \( \text{length} ?\tau \leq \text{length} \pi + 1 \)
using length-trace-parallel-plan-strips-lte-length-plan-plus-one
by fast
then have \( 0 < \text{length} \pi \)
using trace-parallel-plan-strips-length-gt-one-if assms
by force
then obtain \( k' \) where \( \text{length} \pi = \text{Suc} k' \)
using gr0-implies-Suc
by meson
thus \?thesis using that
using length-Suc-conv[of \pi k']
by blast
qed

— Show that if a parallel plan trace does not have maximum length, in the last state reached through operator execution the parallel operator execution condition was violated.

**corollary** length-trace-parallel-plan-strips-lt-length-plan-plus-one-then:

**assumes** \( \text{length} (\text{trace-parallel-plan-strips} I \pi) < \text{length} \pi + 1 \)

**shows** \( \neg\text{are-all-operators-applicable} \)
\( (\text{execute-parallel-plan} I (\text{take} (\text{length} (\text{trace-parallel-plan-strips} I \pi) - 1) \pi)) \)
\( (\pi \# (\text{length} (\text{trace-parallel-plan-strips} I \pi) - 1)) \)
\( \lor \neg\text{are-all-operator-effects-consistent} (\pi \# (\text{length} (\text{trace-parallel-plan-strips} I \pi) - 1)) \)

**using** assms
**proof** (induction \pi arbitrary: \( I \))

**case** (Cons \ops \pi)
let \( ?\tau = \text{trace-parallel-plan-strips} I (\ops \# \pi) \)
and \( ?I' = \text{execute-parallel-operator} I \ops \)
show \?case

**proof** (cases are-all-operators-applicable \I \ops \land are-all-operator-effects-consistent)
case True
then have \( \tau \text{-is: } ?\tau = I \# \text{trace-parallel-plan-strips } ?I' \pi \)
by fastforce

show \( \text{?thesis} \)
proof (cases length (trace-parallel-plan-strips ?I' \pi) < length \pi + 1)
case True
then have \( \neg \text{are-all-operators-applicable} \)
\((\text{execute-parallel-plan } ?I')\)
\((\pi ! (\text{length (trace-parallel-plan-strips } ?I' \pi) - 1))\)
\( \lor \neg \text{are-all-operator-effects-consistent} \)
\((\pi ! (\text{length (trace-parallel-plan-strips } ?I' \pi) - 1))\)
using Cons.IH[of ?I']
by blast
moreover have \( \text{trace-parallel-plan-strips } ?I' \pi \neq [] \)
using trace-parallel-plan-strips-not-nil
by blast
ultimately show \( \text{?thesis} \)
unfolding take-Cons'
by simp

next
case False
then have length (trace-parallel-plan-strips ?I' \pi) \geq length \pi + 1
by fastforce

thm Cons.prems
moreover have length (trace-parallel-plan-strips I (ops \# \pi))
= 1 + length (trace-parallel-plan-strips ?I' \pi)
using True
by force
moreover have length (trace-parallel-plan-strips ?I' \pi)
< length (ops \# \pi)
using Cons.prems calculation(2)
by force
ultimately have False
by fastforce
thus \( \text{?thesis} .. \)
qed

next
case False
then have \( \tau \text{-is-singleton: } ?\tau = [I] \)
using False
by auto
then have ops = (ops \# \pi)! (length ?\tau - 1)
by fastforce
moreover have execute-parallel-plan I (take (length ?\tau - 1) \pi) = I
using \( \tau \text{-is-singleton} \)
by auto
— TODO slow.
ultimately show \( \textit{thesis} \)
using \( \text{False} \)
by auto
qed
qed simp

\textbf{lemma} trace-parallel-plan-step-effect-is:
\textbf{assumes} \( k < \text{length} (\text{trace-parallel-plan-strips } I \pi) - 1 \)
\textbf{shows} \( \text{trace-parallel-plan-strips } I \pi ! \text{Suc } k \)
= \( \text{execute-parallel-operator} (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k) \)
\textbf{proof} —
— \textbf{NOTE} Rewrite the proposition using \textit{lemma} trace-parallel-plan-strips-subplan.
{ let \( \tau = \text{trace-parallel-plan-strips } I \pi \)
have \( \text{Suc } k < \text{length } \tau \)
using \textit{assms} by linarith
hence \( \text{trace-parallel-plan-strips } I \pi ! \text{Suc } k \)
= \( \text{execute-parallel-plan } I \text{(take } (\text{Suc } k) \pi) \)
using \textit{trace-parallel-plan-plan-prefix[of } \text{Suc } k I \pi]\)
by blast
}
moreover have \( \text{execute-parallel-plan } I \text{(take } (\text{Suc } k) \pi) \)
= \( \text{execute-parallel-operator} (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k) \)
using \textit{assms}
\textbf{proof} \( \text{(induction } k \text{ arbitrary: } I \pi) \)
case \( 0 \)
then have \( \text{execute-parallel-operator} (\text{trace-parallel-plan-strips } I \pi ! 0) (\pi ! 0) \)
= \( \text{execute-parallel-operator } I (\pi ! 0) \)
using \textit{trace-parallel-plan-strips-head-is-initial-state[of } I \pi]\)
by argo
moreover { obtain \( \text{ops } \pi' \text{ where } \pi = \text{ops } \neq \pi' \)
using \textit{plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF 0.prem]}]
by blast
then have \( \text{take } (\text{Suc } 0) \pi = [\pi ! 0] \)
by simp
hence \( \text{execute-parallel-plan } I \text{(take } (\text{Suc } 0) \pi) \)
= \( \text{execute-parallel-plan } I [\pi ! 0] \)
by argo
}
moreover { have \( 0 < \text{length} (\text{trace-parallel-plan-strips } I \pi) - 1 \)
using \textit{trace-parallel-plan-strips-length-gt-one-if 0.prem}
by fastforce
hence \( \text{are-all-operators-applicable } I (\pi ! 0) \)
\land \text{are-all-operator-effects-consistent } (\pi ! 0)
using \textit{trace-parallel-plan-strips-operator-preconditions}[of 0 I \pi]

\textit{trace-parallel-plan-strips-head-is-initial-state}[of I \pi]

by \textit{argo}

\}

ultimately show ?case

by \textit{auto}

next

case (Suc k)

obtain ops \pi' \textbf{where} \pi\text{-split}: \pi = ops \# \pi'

using \textit{plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements}[OF Suc.prems]

by \textit{blast}

let ?I' = execute-parallel-operator I ops

\{

have length (\textit{trace-parallel-plan-strips} I \pi) =

\hspace{1em} 1 + length (\textit{trace-parallel-plan-strips} \ ?I' \ \pi')

using Suc.prems \pi\text{-split}

by \textit{fastforce}

then have \( k < \) length (\textit{trace-parallel-plan-strips} \ ?I' \ \pi')

using Suc.prems

by \textit{fastforce}

moreover have \textit{trace-parallel-plan-strips} I \pi ! Suc k

= \textit{trace-parallel-plan-strips} \ ?I' \ \pi' ! k

using Suc.prems \pi\text{-split}

by \textit{force}

ultimately have \textit{trace-parallel-plan-strips} I \pi ! Suc k

= execute-parallel-plan \ ?I' (\textit{take} k \ \pi')

using \textit{trace-parallel-plan-plan-prefix}[of k \ ?I' \ \pi']

by \textit{argo}

\}

moreover have \textit{execute-parallel-plan} \ I \ (\textit{take} (Suc \ (Suc k)) \ \pi)

= \textit{execute-parallel-plan} \ ?I' (\textit{take} \ (Suc k) \ \pi')

using Suc.prems \pi\text{-split}

by \textit{fastforce}

moreover { 

have 0 < length (\textit{trace-parallel-plan-strips} I \pi) - 1

using Suc.prems

by \textit{linarith}

hence \textit{are-all-operators-applicable} \ I \ (\pi ! 0)

\hspace{1em} \land \textit{are-all-operator-effects-consistent} \ (\pi ! 0)

using \textit{trace-parallel-plan-strips-operator-preconditions}[of 0 I \pi]

\textit{trace-parallel-plan-strips-head-is-initial-state}[of I \pi]

by \textit{argo}

\}

ultimately show ?case

using Suc.IH Suc.prems \pi\text{-split}

by \textit{auto}

qed

ultimately show \ ?thesis
— Show that every state in a plan execution trace of a valid problem description is defined for all problem variables. This is true because the initial state is defined for all problem variables—by definition of is-valid-problem-strips Π—and no operator can remove a previously defined variable (only positive and negative effects are possible).

**lemma** trace-parallel-plan-strips-none-if:

**fixes** Π:: 'a strips-problem

**assumes** is-valid-problem-strips Π

and is-parallel-solution-for-problem Π π

and k < length (trace-parallel-plan-strips ((Π)₁) π)

**shows** (trace-parallel-plan-strips ((Π)₁) π ! k) v = None ←→ v /∈ set ((Π)ᵥ)

**proof**

let ?vs = strips-problem.variables-of Π

and ?ops = strips-problem.operators-of Π

and ?τ = trace-parallel-plan-strips ((Π)₁) π

and ?I = strips-problem.initial-of Π

show ?thesis

using assms

proof (induction k)

case 0

have ?τ ! 0 = ?I

using trace-parallel-plan-strips-head-is-initial-state

by auto

then show ?case

using is-valid-problem-strips-initial-of-dom[OF assms(1)]

by auto

next

case (Suc k)

have k-lt-length-τ-minus-one: k < length ?τ − 1

using Suc.prems(3)

by linarith

then have IH: (trace-parallel-plan-strips ?I π ! k) v = None ←→ v /∈ set ((Π)ᵥ)

((Π)ᵥ)

using Suc.IH[OF Suc.prems(1, 2)]

by force

have τ-Suc-k-is: (?τ ! Suc k) = execute-parallel-operator (?τ ! k) (π ! k)

using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one],

have all-operators-applicable: are-all-operators-applicable (?τ ! k) (π ! k)

and all-effects-consistent: are-all-operator-effects-consistent (π ! k)

using trace-parallel-plan-strips-operator-preconditions[OF k-lt-length-τ-minus-one]

by simp

show ?case

proof (rule iffI)

assume τ-Suc-k-of-v-is-None: (?τ ! Suc k) v = None
show $v \notin \text{set} \ (\Pi V)$
proof (rule ccontr)
assume $\neg v \notin \text{set} \ (\Pi V)$
then have v-in-set-vs: $v \in \text{set}(\Pi V)$
  by blast
show False
proof (cases $\exists \op \in \text{set} \ (\pi ! k)$.
v $\in \text{set} \ (\text{add-effects-of} \ \op) \lor v \in \text{set} \ (\text{delete-effects-of} \ \op)$)
  case True
  then obtain $\op$
    where $\op \in \text{set} \ (\pi ! k)$
    and $v \in \text{set} \ (\text{add-effects-of} \ \op) \lor v \in \text{set} \ (\text{delete-effects-of} \ \op)$.
  then consider (A) $v \in \text{set} \ (\text{add-effects-of} \ \op)$
    | (B) $v \in \text{set} \ (\text{delete-effects-of} \ \op)$
    by blast
  thus False
  using execute-parallel-operator-positive-effect-if $\tau$-Suc-k-of-v-is-None $\tau$-Suc-k-is
  by (cases, fastforce+)
next
  case False
  then have $\forall \op \in \text{set} \ (\pi ! k)$.
v $\notin \text{set} \ (\text{add-effects-of} \ \op) \land v \notin \text{set} \ (\text{delete-effects-of} \ \op)$
  by blast
  then have ($\tau ! \text{Suc} \ k \ v = (? \tau ! k \ v$
    using execute-parallel-operator-no-effect-if $\tau$-Suc-k-is
  by fastforce
  then have $v \notin \text{set} \ (\Pi V)$
    using IH $\tau$-Suc-k-of-v-is-None
    by simp
  thus False
  using v-in-set-vs
  by blast
qed
next
assume v-notin-vs: $v \notin \text{set} \ (\Pi V)$
{
  fix $\op$
  assume op-in-\pi_k: $\op \in \text{set} \ (\pi ! k)$
  {
    have $1 < \text{length} \ ? \tau$
      using trace-parallel-plan-strips-length-gt-one-if [OF $k$-lt-length-$\tau$-minus-one].
  then have $0 < \text{length} \ ? \tau - 1$
    using k-lt-length-$\tau$-minus-one
    by linarith
moreover have \( \text{length } \tau - 1 \leq \text{length } \pi \)

using \( \text{length-trace-parallel-plan-strips-lte-length-plan-plus-one} \)

le-diff-conv

by blast

then have \( k < \text{length } \pi \)

using \( k-\lt\text{-length-}\tau-\text{-minus-one} \)

by force

hence \( \pi ! k \in \text{set } \pi \)

by simp

\}

then have \( \text{op-in-ops: } \text{op} \in \text{set } \tau\text{ops} \)

using \( \text{is-parallel-solution-for-problem-operator-set} \)

by force

hence \( v \notin \text{set } (\text{add-effects-of } \text{op}) \) and \( v \notin \text{set } (\text{delete-effects-of } \text{op}) \)

subgoal

using \( \text{is-valid-problem-strips-operator-variable-sets} \)

by auto

subgoal

using \( \text{is-valid-problem-strips-operator-variable-sets} \)

by auto

\}

then have \( (?) \tau ! \text{Suc } k \) v = \( (?) \tau ! k \) v

using \( \text{execute-parallel-operator-no-effect-if } \tau-\text{Suc-}k\text{-is} \)

by metis

thus \( (?) \tau ! \text{Suc } k \) v = None

using \( \text{IH } v\\text{-notin-vs} \)

by fastforce

qed

qed

Finally, given initial and goal states \( I \) and \( G \), we can show that it’s equivalent to say that \( \pi \) is a solution for \( I \) and \( G \)—i.e. \( G \subseteq_m \text{execute-parallel-plan } I \pi \)—and that the goal state is subsumed by the last element of the trace of \( \pi \) with initial state \( I \).

lemma \( \text{execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace} \):

\( G \subseteq_m \text{execute-parallel-plan } I \pi \)

\( \iff \) \( G \subseteq_m \text{last (trace-parallel-plan-strips } I \pi) \)

proof

let \( \text{LHS} = G \subseteq_m \text{execute-parallel-plan } I \pi \)

and \( \text{RHS} = G \subseteq_m \text{last (trace-parallel-plan-strips } I \pi) \)

show \( \text{thesis} \)

proof (rule iffI)
assume \(?LHS\)
thus \(?RHS\)
proof (induction \(\pi\) arbitrary: \(I\))
— NOTE Nil case follows from simplification.
case (\(\text{Cons} \ a \ \pi\))
thus \(?case\)
using Cons.prem
proof (cases are-all-operators-applicable \(I\) \(a\) \(\land\) are-all-operator-effects-consistent)
a)

\[
\text{case } \text{True} \\
\text{let } ?I' = \text{execute-parallel-operator } I \ a \\
\{
\text{have } \text{execute-parallel-plan } I \ (a \ # \ \pi) = \text{execute-parallel-plan } ?I' \ \pi \\
\text{using } \text{True} \\
\text{by } \text{auto} \\
\text{then have } G \subseteq_m \text{execute-parallel-plan } ?I' \ \pi \\
\text{using Cons.prem} \\
\text{by presburger} \\
\text{hence } G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } ?I' \ \pi) \\
\text{using Cons.IH[of } ?I'] \\
\text{by blast} \\
\}
\]
moreover {
\text{have } \text{trace-parallel-plan-strips } I \ (a \ # \ \pi) \\
= I \ # \ \text{trace-parallel-plan-strips } ?I' \ \pi \\
\text{using } \text{True} \\
\text{by simp} \\
\text{then have } \text{last } (\text{trace-parallel-plan-strips } I \ (a \ # \ \pi)) \\
= \text{last } (I \ # \ \text{trace-parallel-plan-strips } ?I' \ \pi) \\
\text{by argo} \\
\text{hence last } (\text{trace-parallel-plan-strips } I \ (a \ # \ \pi)) \\
= \text{last } (\text{trace-parallel-plan-strips } ?I' \ \pi) \\
\text{using trace-parallel-plan-strips-last-cons-then[of } ?I' \ ?I'] \\
\text{by argo} \\
}
ultimately show \(?thesis\)
by argo
qed force
qed simp
next
assume \(?RHS\)
thus \(?LHS\)
proof (induction \(\pi\) arbitrary: \(I\))
— NOTE Nil case follows from simplification.
case (\(\text{Cons} \ a \ \pi\))
thus \(?case\)
proof (cases are-all-operators-applicable \(I\) \(a\) \(\land\) are-all-operator-effects-consistent)
a)

\[
\text{case } \text{True} \\
\]
let \( I' = \text{execute-parallel-operator } I \ a \)
{
\begin{align*}
\text{have} & \quad \text{trace-parallel-plan-strips} \ I \ (a \ # \ \pi) = I \ # \ (\text{trace-parallel-plan-strips} \ ?I' \ \pi) \\
\text{using} & \quad \text{True} \\
\text{by} & \quad \text{simp} \\
\text{then} & \quad \text{have} \quad \text{last} \ (\text{trace-parallel-plan-strips} \ I \ (a \ # \ \pi)) = \text{last} \ (\text{trace-parallel-plan-strips} \ ?I' \ \pi) \\
\text{using} & \quad \text{trace-parallel-plan-strips-last-cons-then}[\text{of } I \ ?I' \ \pi] \\
\text{by} & \quad \text{argo} \\
\text{hence} & \quad G \subseteq_m \text{last} \ (\text{trace-parallel-plan-strips} \ ?I' \ \pi) \\
\text{using} & \quad \text{Cons}. \text{prems} \\
\text{by} & \quad \text{argo} \\
\end{align*}
\}
\text{thus} \ \?\text{thesis} \\
\text{using} \quad \text{True} \ \text{Cons} \\
\text{by} \quad \text{simp} \\
\text{next} \\
\text{case} \quad \text{False} \\
\text{then} & \quad \text{have} \quad \text{last} \ (\text{trace-parallel-plan-strips} \ I \ (a \ # \ \pi)) = I \\
\text{and} & \quad \text{execute-parallel-plan} \ I \ (a \ # \ \pi) = I \\
\text{by} & \quad (\text{fastforce}, \text{force}) \\
\text{thus} & \quad \?\text{thesis} \\
\text{using} \quad \text{Cons}. \text{prems} \\
\text{by} & \quad \text{argo} \\
\text{qed} \\
\text{qed \ fastforce} \\
\text{qed} \\
\text{qed}

3.3 Serializable Parallel Plans

With the groundwork on parallel and serial execution of STRIPS in place we can now address the question under which conditions a parallel solution to a problem corresponds to a serial solution and vice versa. As we will see (in theorem ??), while a serial plan can be trivially rewritten as a parallel plan consisting of singleton operator list for each operator in the plan, the condition for parallel plan solutions also involves non interference.

— Given that non interference implies that operator execution order can be switched arbitrarily, it stands to reason that parallel operator execution can be serialized if non interference is mandated in addition to the regular parallel execution condition (applicability and effect consistency). This is in fact true as we show in the lemma below

\[ ^5 \text{In the source literature it is required that app}_o \ (s) \text{ is defined which requires that app}_o \ (s) \text{ is defined for every } o \in S. \text{ This again means that the preconditions hold in } s \text{ and the set of effects is consistent which translates to the execution condition in execute-parallel-operator.} \]

\[ ^3 \text{Lemma 2.11., p.1037} \]

\[ ^3 \text{Lemma 2.11., p.1037} \text{ is in fact proposed to be true for any total} \]
lemma execute-parallel-operator-equals-execute-sequential-strips-if:
  fixes s :: ('variable, bool) state
  assumes are-all-operators-applicable s ops
      and are-all-operator-effects-consistent ops
      and are-all-operators-non-interfering ops
  shows execute-parallel-operator s ops = execute-serial-plan s ops
  using assms
proof (induction ops arbitrary: s)
  case Nil
  have execute-parallel-operator s Nil
    = foldl (++) s (map (map-of ◦ effect-to-assignments) Nil)
    using Nil.prems(1,2)
    unfolding execute-parallel-operator-def
    by presburger
  also have ... = s
    by simp
  finally have execute-parallel-operator s Nil = s
    by blast
  moreover have execute-serial-plan s Nil = s
    by auto
  ultimately show ?case
    by simp
next
  case (Cons a ops)
    — NOTE Use the preceding lemmas to show that the premises hold for
    the sublist and use the IH to obtain the theorem for the sublist ops.
    have a: is-operator-applicable-in s a
      using are-all-operators-applicable-cons Cons.prems(1)
      by blast
    let ?s' = s ++ map-of (effect-to-assignments a)
    { from Cons.prems
      have are-all-operators-applicable ?s' ops
        and are-all-operator-effects-consistent ops
        and are-all-operators-non-interfering ops
        using execute-parallel-plan-precondition-cons
        by blast
      then have execute-serial-plan ?s' ops
        = execute-parallel-operator ?s' ops
        using Cons.IH
        by presburger
    }
  moreover from Cons.prems
  have execute-parallel-operator s (Cons a ops)
    = execute-parallel-operator ?s' ops
  using execute-parallel-operator-cons-equals-corollary
ordering of the operator set but we only proof it for the implicit total ordering induced
by the specific order in the operator list of the problem statement.
unfolding execute-operator-def
by simp
moreover
from a have execute-serial-plan s (Cons a ops)
  = execute-serial-plan ?s' ops
unfolding execute-serial-plan-def execute-operator-def
  is-operator-applicable-in-def
by fastforce
ultimately show ?case
by argo
qed

lemma execute-serial-plan-split-i:
assumes are-all-operators-applicable s (op ≠ π)
  and are-all-operators-non-interfering (op ≠ π)
shows are-all-operators-applicable (s >>= op) π
using assms
proof (induction π arbitrary: s)
case Nil
then show ?case
  unfolding are-all-operators-applicable-def
  by simp
next
case (Cons op' π)
let ?t = s >>= op
{
  fix x
  assume x ∈ set (op' ≠ π)
  moreover have op ∈ set (op ≠ op' ≠ π)
    by simp
  moreover have ¬are-operators-interfering op x
    using Cons.prems(2) calculation(1)
  unfolding are-all-operators-non-interfering-def list-all-iff
  by fastforce
  moreover have is-operator-applicable-in s op
    using Cons.prems(1)
  unfolding are-all-operators-applicable-def list-all-iff
    is-operator-applicable-in-def
    by force
  moreover have is-operator-applicable-in s x
    using are-all-operators-applicable-cons(2)[OF Cons.prems(1)] calculation(1)
  unfolding are-all-operators-applicable-def list-all-iff
    is-operator-applicable-in-def
    by fast
  ultimately have is-operator-applicable-in ?t x
    using execute-parallel-plan-precondition-cons-i[of op x s]
    by (auto simp: execute-operator-def)
} thus ?case
— Show that plans \( \pi \) can be split into separate executions of partial plans \( \pi_1 \) and \( \pi_2 \) with \( \pi = \pi_1 \@ \pi_2 \), if all operators in \( \pi_1 \) are applicable in the given state \( s \) and there is no interference between subsequent operators in \( \pi_1 \). This is the case because non interference ensures that no precondition for any operator in \( \pi_1 \) is negated by the execution of a preceding operator. Note that the non interference constraint excludes partial plans where a precondition is first violated during execution but later restored which would also allow splitting but does not meet the non interference constraint (which must hold for all possible executing orders).

**Lemma execute-serial-plan-split:**

fixes \( s :: (\text{variable}, \text{bool}) \) state
assumes are-all-operators-applicable \( s \) \( \pi_1 \)
and are-all-operators-non-interfering \( \pi_1 \)
shows execute-serial-plan \( s \) \( (\pi_1 \@ \pi_2) \)
using assms

**Proof** (induction \( \pi_1 \) arbitrary: \( s \))

let \(?t = s \gg op\)

\{ have are-all-operators-applicable \( s \gg op \) \( \pi_1 \)
using execute-serial-plan-split-i[OF Cons.prems(1, 2)],
moreover have are-all-operators-non-interfering \( \pi_1 \)
using are-all-operators-non-interfering-tail[OF Cons.prems(2)],
ultimately have execute-serial-plan \(?t \ (\pi_1 \@ \pi_2) \) =
execute-serial-plan \( (execute-serial-plan \(?t \pi_1) \pi_2 \)
using Cons.IH[of \(?t\)]
by blast \}
moreover have STRIPS-Representation.is-operator-applicable-in \( s \) \( op \)
using Cons.prems(1)
unfolding are-all-operators-applicable-def list-all-iff
by fastforce
ultimately show \(?case
unfolding execute-serial-plan-def
by simp
qed simp

**Lemma embedding-lemma-i:**

fixes \( I :: (\text{variable}, \text{bool}) \) state
assumes is-operator-applicable-in \( I \) \( op \)
and are-operator-effects-consistent \( op \) \( op \)
shows $I \Rightarrow op = \text{execute-parallel-operator } I \ [op]$

proof

have are-all-operators-applicable $I \ [op]$
  using assms(1)
unfolding are-all-operators-applicable-def list-all-iff is-operator-applicable-in-def
  by fastforce
moreover have are-all-operator-effects-consistent $[op]$
  unfolding are-all-operator-effects-consistent-def list-all-iff
  using assms(2)
  by fastforce
moreover have are-all-operators-non-interfering $[op]$
  by simp
moreover have $I \Rightarrow op = \text{execute-serial-plan } I \ [op]$
  using assms(1)
  unfolding is-operator-applicable-in-def
  by (simp add: assms(1) execute-operator-def)
ultimately show ?thesis
  using execute-parallel-operator>equals-execute-sequential-strips-if
  by force
qed

lemma execute-serial-plan-is-execute-parallel-plan-ii:

fixes $I :: \{\text{variable strips-state}\}$

assumes $\forall op \in \text{set } \pi. \text{are-operator-effects-consistent } op \ [op]$
and $G \subseteq_m \text{execute-serial-plan } I \ [\pi]$

shows $G \subseteq_m \text{execute-parallel-plan } I \ (\text{embed } \pi)$

proof

show ?thesis
  using assms

proof (induction $\pi$ arbitrary: $I$)

  case (Cons $op \ \pi$)

  then show ?case
  proof (cases is-operator-applicable-in $I \ [op]$)
    case True
    let $?J = I \Rightarrow op$
    and $?J' = \text{execute-parallel-operator } I \ [op]$
    
    have $G \subseteq_m \text{execute-serial-plan } ?J \ [\pi]$
      using Cons.prems(2) True
      unfolding is-operator-applicable-in-def
      by (simp add: True)
    hence $G \subseteq_m \text{execute-parallel-plan } ?J (\text{embed } \pi)$
      using Cons.IH[of $?J'] Cons.prems(1)
      by fastforce
  
moreover 

  have are-all-operators-applicable $I \ [op]$
    using True
    unfolding are-all-operators-applicable-def list-all-iff
is-operator-applicable-in-def
by fastforce
moreover have are-all-operator-effects-consistent [op]
  unfolding are-all-operator-effects-consistent-def list-all-iff
  using Cons.prems(1)
  by fastforce
moreover have ?J = ?J'
  using execute-parallel-operator-equals-execute-sequential-strips-if[OF
    calculation(1, 2)] Cons.prems(1) True
  unfolding is-operator-applicable-in-def
  by (simp add: True)
ultimately have execute-parallel-plan I (embed (op # π))
  = execute-parallel-plan ?J (embed π)
  by fastforce
}
ultimately show ?thesis
  by presburger
next
  case False
    then have G ⊆m I
      using Cons.prems is-operator-applicable-in-def
      by simp
    moreover {  
      have ¬are-all-operators-applicable I [op]
        using False
      unfolding are-all-operators-applicable-def list-all-iff
        is-operator-applicable-in-def
        by force
        hence execute-parallel-plan I (embed (op # π)) = I
          by simp
      }
    ultimately show ?thesis
      by presburger
    qed
  qed simp
qed

lemma embedding-lemma-iii:
  fixes II:: 'a strips-problem
  assumes ∀op ∈ set π. op ∈ set ((II)σ)
  shows ∀ops ∈ set (embed π). ∀op ∈ set ops. op ∈ set ((II)σ)
  proof –
    have nb: set (embed π) = { [op] | op. op ∈ set π }
      by (induction π; force)
    {  
      fix ops
      assume ops ∈ set (embed π)
      moreover obtain op where op ∈ set π and ops = [op]
    }
We show in the following theorem that—as mentioned—a serial solution \( \pi \) to a STRIPS problem \( \Pi \) corresponds directly to a parallel solution obtained by embedding each operator in \( \pi \) in a list (by use of function \texttt{List-Supplement.embed}). The proof shows this by first confirming that

\[
G \subseteq_m \text{execute-serial-plan} ((\Pi)_I) \pi
\]

\[
\implies G \subseteq_m \text{execute-serial-plan} ((\Pi)_I) (\text{embed } \pi)
\]

using lemma ; and moreover by showing that

\[
\forall \text{ops } \subseteq \text{set } (\text{embed } \pi). \forall \text{op } \subseteq \text{set ops. op } \subseteq (\Pi)_O
\]

meaning that under the given assumptions, all parallel operators of the embedded serial plan are again operators in the operator set of the problem.

\textbf{Theorem embedding-lemma:}
\begin{itemize}
  \item \textbf{Assumes} \texttt{is-valid-problem-strips} \( \Pi \)
  \item \textbf{and} \texttt{is-serial-solution-for-problem} \( \Pi \) \( \pi \)
  \item \textbf{shows} \texttt{is-parallel-solution-for-problem} \( \Pi \) (\text{embed } \pi)
\end{itemize}
\textbf{Proof} —
\begin{itemize}
  \item \textbf{have} \texttt{nb}_1: \forall \text{op } \subseteq \text{set } \pi. \text{op } \subseteq \text{set } ((\Pi)_O)
    \item \textbf{using} \texttt{assms}(2)
  \item \textbf{unfolding} \texttt{is-serial-solution-for-problem-def} \texttt{list-all-iff} \texttt{ListMem-iff} \texttt{operators-of-def}
    \item \textbf{by} \texttt{blast}
    \begin{itemize}
      \item \textbf{fix} \text{op}
      \item \textbf{assume} \text{op } \subseteq \text{set } \pi
      \item \textbf{moreover have} \text{op } \subseteq \text{set } ((\Pi)_O)
        \item \textbf{using} \texttt{nb}_1 \texttt{calculation}
        \item \textbf{by} \texttt{fast}
      \item \textbf{moreover have} \texttt{is-valid-operator-strips} \( \Pi \) \text{op}
        \item \texttt{using} \texttt{assms}(1) \texttt{calculation}(2)
        \item \textbf{unfolding} \texttt{is-valid-problem-strips-def} \texttt{is-valid-problem-strips-def} \texttt{list-all-iff} \texttt{operators-of-def}
          \item \textbf{by} \texttt{meson}
          \item \textbf{moreover have} \texttt{list-all} (\lambda v. \neg \texttt{ListMem} v (\texttt{delete-effects-of} \text{op})) (\texttt{add-effects-of} \text{op})
            \item \textbf{and} \texttt{list-all} (\lambda v. \neg \texttt{ListMem} v (\texttt{add-effects-of} \text{op})) (\texttt{delete-effects-of} \text{op})
              \item \textbf{using} \texttt{calculation}(3)
              \item \textbf{unfolding} \texttt{is-valid-operator-strips-def}
    \end{itemize}
\end{itemize}
by meson+
moreover have \( \neg \text{list-ex} (\lambda v. \text{ListMem} v \text{ (delete-effects-of } \text{op})) \) (add-effects-of op)
and \( \neg \text{list-ex} (\lambda v. \text{ListMem} v \text{ (add-effects-of } \text{op})) \) (delete-effects-of op)
using calculation(4, 5) not-list-ex-equals-list-all-not
by blast+
moreover have \( \neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) \text{ (delete-effects-of } \text{op})) \) (add-effects-of op)
and \( \neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) \text{ (add-effects-of } \text{op})) \) (delete-effects-of op)
using calculation(6, 7) not-list-ex-equals-list-all-not
ultimately have are-operator-effects-consistent op op
unfolding are-operator-effects-consistent-def Let-def
by blast
} note nb2 = this
moreover have
have \((\Pi)_{G} \subseteq_{m} \text{execute-serial-plan} \ ((\Pi)_{I}) \) \(\pi\)
using assms(2)
unfolding is-serial-solution-for-problem-def
by simp
hence \((\Pi)_{G} \subseteq_{m} \text{execute-parallel-plan} \ ((\Pi)_{I}) \) \((\text{embed } \pi)\)
using execute-serial-plan-is-execute-parallel-plan-ii nb2
by blast
}
moreover have \(\forall \text{ops} \in \text{set} \ (\text{embed } \pi). \forall \text{op} \in \text{set ops. op} \in \text{set} \ ((\Pi)_{O})\)
using embedding-lemma-iii[of nb1]
ultimately show \(\forall \text{op} \in \text{set} \ (\text{concat } \pi). \text{op} \in \text{set} \ ((\Pi)_{O})\)
by blast

lemma flattening-lemma-i:
fixes II:: 'a strips-problem
assumes \(\forall \text{ops} \in \text{set } \pi. \forall \text{op} \in \text{set ops. op} \in \text{set} \ ((\Pi)_{O})\)
shows \(\forall \text{op} \in \text{set} \ (\text{concat } \pi). \text{op} \in \text{set} \ ((\Pi)_{O})\)
proof –

{ fix \text{op}
assume \text{op} \in \text{set} \ (\text{concat } \pi)
moreover have \text{op} \in \bigcup \text{ops} \in \text{set } \pi. \text{set ops}
using calculation
unfolding set-concat.
then obtain \text{ops} \ where \text{ops} \in \text{set } \pi \text{ and } \text{op} \in \text{set ops}
using UN-iff
by blast
ultimately have \text{op} \in \text{set} \ ((\Pi)_{O})
using assms

53
by blast
)
thus ?thesis..
qed

lemma flattening-lemma-ii:
fixes I :: 'variable strips-state
assumes \( \forall \text{ops} \in \text{set } \pi. \exists \text{op}. \text{ops} = [\text{op}] \land \text{is-valid-operator-strips } \Pi \text{ op} \)
and \( G \subseteq_m \text{execute-parallel-plan } I \pi \)
shows \( G \subseteq_m \text{execute-serial-plan } I (\text{concat } \pi) \)
proof -
let \(?\pi' = \text{concat } \pi\
{
{
\{  
  fix op  
  assume is-valid-operator-strips \Pi op  
  moreover have list-all (\( \lambda v. \neg \text{ListMem } v \) (\text{delete-effects-of op})) (\text{add-effects-of op})  
  and list-all (\( \lambda v. \neg \text{ListMem } v \) (\text{add-effects-of op})) (\text{delete-effects-of op})  
  using calculation(1)  
  unfolding is-valid-operator-strips-def  
  by meson+  
  moreover have \( \neg \text{list-ex } (\lambda v. \text{ListMem } v \) (\text{delete-effects-of op})) (\text{add-effects-of op})  
  and \( \neg \text{list-ex } (\lambda v. \text{ListMem } v \) (\text{add-effects-of op})) (\text{delete-effects-of op})  
  using calculation(2, 3) \text{not-list-ex-equals-list-all-not}  
  by blast+  
  moreover have \( \neg \text{list-ex } (\lambda v. \text{list-ex } (=) v \) (\text{delete-effects-of op})) (\text{add-effects-of op})  
  and \( \neg \text{list-ex } (\lambda v. \text{list-ex } (=) v \) (\text{delete-effects-of op})) (\text{delete-effects-of op})  
  using calculation(4, 5) \text{not-list-ex-equals-list-all-not}  
  by blast+  
  ultimately have are-operator-effects-consistent op op  
  unfolding are-operator-effects-consistent-def \text{Let-def}  
  by blast  
}\}  
note nb1 = this
show ?thesis
proof (induction \pi \text{ arbitrary: } I)
case (Cons \text{ops } \pi)
  obtain op where \text{ops-is: } \text{ops} = [\text{op}] \land \text{is-valid-op: } \text{is-valid-operator-strips } \Pi \text{ op} 
II op
  using Cons.prems(1)
  by fastforce
show ?case
proof (cases are-all-operators-applicable I \text{ops})
case True
  let ?J = \text{execute-parallel-operator } I [\text{op}]

and \( ?J' = I \supset op \)

have \( nb_2: \text{is-operator-applicable-in } I \ op \)
  using \( True \ ops-is \)
  unfolding \( \text{are-all-operators-applicable-def list-all-iff} \)
  \( \text{is-operator-applicable-in-def} \)
  by \( \text{simp} \)

have \( nb_3: \text{are-operator-effects-consistent op op} \)
  using \( nb_1[OF \text{is-valid-op}] \).

\{
  then have \( \text{are-all-operator-effects-consistent ops} \)
    unfolding \( \text{are-all-operator-effects-consistent-def list-all-iff} \)
    using \( ops-is \)
    by \( \text{fastforce} \)
  hence \( G \subseteq m \text{ execute-parallel-plan } ?J \pi \)
    using \( \text{Cons.prems(2)} ops-is True \)
    by \( \text{fastforce} \)
\}

moreover have \( \text{execute-serial-plan } I \ (\text{concat} \ (ops \# \pi)) \)
  \( = \text{execute-serial-plan } ?J' \ (\text{concat} \ \pi) \)
  using \( ops-is nb_2 \)
  unfolding \( \text{is-operator-applicable-in-def} \)
  by \( \text{(simp add: execute-operator-def nb_2)} \)

moreover have \( ?J = ?J' \)
  unfolding \( \text{execute-parallel-operator-def execute-operator-def comp-apply} \)
  by \( \text{fastforce} \)

ultimately show \( \text{thesis} \)
  using \( \text{Cons.IH Cons.prems} \)
  by \( \text{force} \)

next

\text{case False}

moreover have \( G \subseteq m \ I \)
  using \( \text{Cons.prems(2)} \ calculation \)
  by \( \text{force} \)

moreover \{ 
  have \( \text{~is-operator-applicable-in } I \ op \)
    using \( ops-is False \)
    unfolding \( \text{are-all-operators-applicable-def list-all-iff} \)
    \( \text{is-operator-applicable-in-def} \)
    by \( \text{fastforce} \)
  hence \( \text{execute-serial-plan } I \ (\text{concat} \ (ops \# \pi)) = I \)
    using \( ops-is \text{ is-operator-applicable-in-def} \)
    by \( \text{simp} \)
\}

ultimately show \( \text{thesis} \)
  by \( \text{argo} \)

\text{qed}

\text{qed \ force}

\text{qed}

The opposite direction is also easy to show if we can normalize the parallel
plan to the form of an embedded serial plan as shown below.

**lemma** flattening-lemma:

**assumes** is-valid-problem-strips II
  and \( \forall \text{ops} \in \text{set } \pi. \exists \text{op}. \text{ops} = \text{[op]} \)
  and is-parallel-solution-for-problem II \( \pi \)

**shows** is-serial-solution-for-problem II (concat \( \pi \))

**proof**

let \( ?\pi' = \text{concat } \pi \)

\{ 
  have \( \forall \text{ops} \in \text{set } \pi. \forall \text{op} \in \text{set } \text{ops}. \text{op} \in \text{set } ((\Pi)_{\sigma}) \)
    using assms(3)
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  hence \( \forall \text{op} \in \text{set } ?\pi'. \text{op} \in \text{set } ((\Pi)_{\sigma}) \)
    using flattening-lemma-i
    by blast
\}

moreover { 

  fix \text{ops}
  assume \( \text{ops} \in \text{set } \pi \)
  moreover obtain \( \text{op} \) where \( \text{ops} = \text{[op]} \)
    using assms(2)
    calculation
    by blast
  moreover have \( \text{op} \in \text{set } ((\Pi)_{\sigma}) \)
    using assms(3)
    calculation
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  moreover have \( \text{is-valid-operator-strips } \Pi \text{ op} \)
    using assms(1)
    calculation(3)
    unfolding is-valid-problem-strips-def Let-def list-all-iff ListMem-iff
    by simp
  ultimately have \( \exists \text{op}. \text{ops} = \text{[op]} \land \text{is-valid-operator-strips } \Pi \text{ op} \)
    by blast
\}

moreover have \( (\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi \)
  using assms(3)
  unfolding is-parallel-solution-for-problem-def
  by simp
  ultimately have \( (\Pi)_G \subseteq_m \text{execute-serial-plan } ((\Pi)_I) ?\pi' \)
    using flattening-lemma-ii
    by blast

ultimately show is-serial-solution-for-problem II \( ?\pi' \)
  unfolding is-serial-solution-for-problem-def list-all-iff ListMem-iff
  by simp

qed

Finally, we can obtain the important result that a parallel plan with a trace
that reaches the goal state of a given problem \( \Pi \), and for which both the parallel operator execution condition as well as non interference is assured at every point \( k \leq \text{length} \, \pi \), the flattening of the parallel plan \( \text{concat} \, \pi \) is a serial solution for the initial and goal state of the problem. To wit, by lemma ?? we have

\[
(G \subseteq_m \text{execute-parallel-plan} \, I \, \pi) = (G \subseteq_m \text{last} \, (\text{trace-parallel-plan-strips} \, I \, \pi))
\]

so the second assumption entails that \( \pi \) is a solution for the initial state and the goal state of the problem. (which implicitly means that \( \pi \) is a solution for the initial state and goal state of the problem). The trace formulation is used in this case because it allows us to write the—state dependent—applicability condition more succinctly. The proof (shown below) is by structural induction on \( \pi \) with arbitrary initial state.

**Theorem** \( \text{execute-parallel-plan-is-execute-sequential-plan-if} \):

- **Fixes** \( I :: (\text{'variable}, \text{bool}) \) state
- **Assumes** \( \text{is-valid-problem} \, \Pi \)
- and \( G \subseteq_m \text{last} \, (\text{trace-parallel-plan-strips} \, I \, \pi) \)
- and \( \forall \, k < \text{length} \, \pi. \)
  - \( \text{are-all-operators-applicable} \, (\text{trace-parallel-plan-strips} \, I \, \pi \, ! \, k) \, (\pi \, ! \, k) \)
  - \( \land \text{are-all-operator-effects-consistent} \, (\pi \, ! \, k) \)
  - \( \land \text{are-all-operators-non-interfering} \, (\pi \, ! \, k) \)
- **Shows** \( G \subseteq_m \text{execute-serial-plan} \, I \, \text{(concat} \, \pi) \)

**Proof** (induction \( \pi \) arbitrary; \( I \))

**Case** (\( \text{Cons} \, \text{ops} \, \pi \))

- let \( ?\text{ops}' = \text{take} \, (\text{length} \, \text{ops}) \, (\text{concat} \, (\text{ops} \# \, \pi)) \)
- let \( ?\text{J}' = \text{execute-parallel-operator} \, I \, \text{ops} \)
  - and \( ?\text{J}' = \text{execute-serial-plan} \, I \, ?\text{ops}' \)

\{ 
  
  have \( \text{trace-parallel-plan-strips} \, I \, \pi \, ! \, 0 = I \) and \( \text{ops} \# \, \pi \) ! \, 0 = \text{ops}
  unfolding \text{trace-parallel-plan-strips-head-is-initial-state}
  by \text{simp+}
  then have \text{are-all-operators-applicable} \, I \, \text{ops}
  and \text{are-all-operator-effects-consistent} \, \text{ops}
  and \text{are-all-operators-non-interfering} \, \text{ops}
  using \text{Cons.prems}(3)
  by \text{auto+}
  then have \( \text{trace-parallel-plan-strips} \, I \, \text{(ops} \# \, \pi) \)
  = \( I \# \, \text{trace-parallel-plan-strips} \, ?\text{J} \, \pi \)
  by \text{fastforce}
\} note \( \text{nb} = \text{this} \)

\{ 
  
  have \text{last} \, (\text{trace-parallel-plan-strips} \, I \, \text{(ops} \# \, \pi))
  = \text{last} \, (\text{trace-parallel-plan-strips} \, ?\text{J} \, \pi)
  using \text{trace-parallel-plan-strips-last-cons-then nb}
  by \text{metis}
\}
hence $G \subseteq_m \text{last (trace-parallel-plan-strips } ?J \pi)$
using Cons.prems(2)
by force
}
moreover {
  fix $k$
  assume $k < \text{length } \pi$
  moreover have $k + 1 < \text{length } (\text{ops } \# \pi)$
  using calculation
  by force
  moreover have $\pi ! k = (\text{ops } \# \pi)! (k + 1)$
  by simp
  ultimately have are-all-operators-applicable
    (trace-parallel-plan-strips ?J $\pi ! k$) ($\pi ! k$)
  and are-all-operator-effects-consistent ($\pi ! k$)
  and are-all-operators-non-interfering ($\pi ! k$)
  using Cons.prems(3) nb
  by force+
}
ultimately have $G \subseteq_m \text{execute-serial-plan } ?J (\text{concat } \pi)$
using Cons.IH[OF Cons.prems(1), of $?J$
by blast
moreover {
  have execute-serial-plan $I (\text{concat } (\text{ops } \# \pi))$
    = execute-serial-plan $?J' (\text{concat } \pi)$
  using execute-serial-plan-split[of $I \text{ops}$] Cons.prems(3)
  by auto
  thm execute-parallel-operator-equals-execute-sequential-strips-if[of $I$
  moreover have $?J = $?J'
  using execute-parallel-operator-equals-execute-sequential-strips-if Cons.prems(3)
  by fastforce
  ultimately have execute-serial-plan $I (\text{concat } (\text{ops } \# \pi))$
    = execute-serial-plan $?J (\text{concat } \pi)$
  using execute-serial-plan-split[of $I \text{ops}$] Cons.prems(3)
  by argo
}
ultimately show $?case$
by argo
qed force

3.4 Auxiliary lemmas about STRIPS

lemma set-to-precondition-of-op-is[simp]: set (to-precondition op)
  = { (v, True) | v. v \in set (precondition-of op) }

unfolding to-precondition-def STRIPS-Representation.to-precondition-def set-map
by blast

end
theory SAS-Plus-Representation
imports State-Variable-Representation
begin

4 SAS+ Representation

We now continue by defining a concrete implementation of SAS+.

SAS+ operators and SAS+ problems again use records. In contrast to STRIPS, the operator effect is contracted into a single list however since we now potentially deal with more than two possible values for each problem variable.

record ('variable, 'domain) sas-plus-operator =
  precondition-of :: ('variable, 'domain) assignment list
  effect-of :: ('variable, 'domain) assignment list

record ('variable, 'domain) sas-plus-problem =
  variables-of :: 'variable list ((v+)[1000]999)
  operators-of :: ('variable, 'domain) sas-plus-operator list ((o+)[1000]999)
  initial-of :: ('variable, 'domain) state ((i+)[1000]999)
  goal-of :: ('variable, 'domain) state ((g+)[1000]999)
  range-of :: 'variable ↦ 'domain list

definition range-of': ('variable, 'domain) sas-plus-problem ⇒ 'variable ⇒ 'domain
set (R+ - - 52)
where
  range-of' Ψ v ≡
  (case sas-plus-problem.range-of Ψ v of None ⇒ {} |
   Some as ⇒ set as)

definition to-precondition
  :: ('variable, 'domain) sas-plus-operator ⇒ ('variable, 'domain) assignment list
where to-precondition ≡ precondition-of

definition to-effect
  :: ('variable, 'domain) sas-plus-operator ⇒ ('variable, 'domain) Effect
where to-effect op ≡ [(v, a) . (v, a) ← effect-of op]

type-synonym ('variable, 'domain) sas-plus-plan
  = ('variable, 'domain) sas-plus-operator list

type-synonym ('variable, 'domain) sas-plus-parallel-plan
  = ('variable, 'domain) sas-plus-operator list list

abbreviation empty-operator
  :: ('variable, 'domain) sas-plus-operator (ϱ)
where empty-operator ≡ [] precondition-of = [], effect-of = []

59
\textbf{definition} \textit{is-valid-operator-sas-plus} :: (\texttt{\{variable, domain\}} \texttt{sas-plus-problem} \Rightarrow (\texttt{\{variable, domain\}} \texttt{sas-plus-operator}) \Rightarrow \texttt{bool} \\
\textbf{where} \textit{is-valid-operator-sas-plus} \Psi \texttt{op} \equiv \texttt{let} \\
\quad \texttt{pre} = \text{precondition-of op} \\
\quad \texttt{eff} = \text{effect-of op} \\
\quad \texttt{vs} = \text{variables-of } \Psi \\
\quad \texttt{D} = \text{range-of } \Psi \\
\texttt{in} \texttt{list-all} (\lambda (v, a). \text{ListMem} v \texttt{vs}) \texttt{pre} \\
\quad \land \texttt{list-all} (\lambda (v, a). (D v \neq \texttt{None}) \land \text{ListMem} a (\texttt{the} (D v))) \texttt{pre} \\
\quad \land \texttt{list-all} (\lambda (v, a). \text{ListMem} v \texttt{vs}) \texttt{eff} \\
\quad \land \texttt{list-all} (\lambda (v, a). (D v \neq \texttt{None}) \land \text{ListMem} a (\texttt{the} (D v))) \texttt{eff} \\
\quad \land \texttt{list-all} (\lambda (v, a). \texttt{list-all} (\lambda (v', a'). v \neq v' \lor a = a') \texttt{pre}) \texttt{pre} \\
\quad \land \texttt{list-all} (\lambda (v, a). \texttt{list-all} (\lambda (v', a'). v \neq v' \lor a = a') \texttt{eff}) \texttt{eff} \\

\textbf{definition} \textit{is-valid-problem-sas-plus} \Psi \equiv \texttt{let} \texttt{ops} = \text{operators-of } \Psi \\
\quad \texttt{vs} = \text{variables-of } \Psi \\
\quad \texttt{I} = \text{initial-of } \Psi \\
\quad \texttt{G} = \text{goal-of } \Psi \\
\quad \texttt{D} = \text{range-of } \Psi \\
\texttt{in} \texttt{list-all} (\lambda v. D v \neq \texttt{None}) \texttt{vs} \\
\quad \land \texttt{list-all} (\textit{is-valid-operator-sas-plus} \Psi) \texttt{ops} \\
\quad \land (\forall v. I v \neq \texttt{None} \iff \text{ListMem} v \texttt{vs}) \\
\quad \land (\forall v. I v \neq \texttt{None} \implies \text{ListMem} (\texttt{the} (I v)) (\texttt{the} (D v))) \\
\quad \land (\forall v. G v \neq \texttt{None} \implies \text{ListMem} v (\text{variables-of } \Psi)) \\
\quad \land (\forall v. G v \neq \texttt{None} \implies \text{ListMem} (\texttt{the} (G v)) (\texttt{the} (D v))) \\

\textbf{definition} \textit{is-operator-applicable-in} :: (\texttt{\{variable, domain\}} \texttt{state} \\
\Rightarrow (\texttt{\{variable, domain\}} \texttt{sas-plus-operator} \\
\Rightarrow \texttt{bool} \\
\textbf{where} \textit{is-operator-applicable-in} \texttt{s} \texttt{op} \equiv \texttt{map-of} (\texttt{precondition-of op}) \subseteq_{m} \texttt{s} \\

\textbf{definition} \textit{execute-operator-sas-plus} :: (\texttt{\{variable, domain\}} \texttt{state} \\
\Rightarrow (\texttt{\{variable, domain\}} \texttt{sas-plus-operator} \\
\Rightarrow (\texttt{\{variable, domain\}} \texttt{state} (\texttt{infixl} \gg> 52) \\
\textbf{where} \textit{execute-operator-sas-plus} \texttt{s} \texttt{op} \equiv \texttt{s} \gg> \texttt{map-of} (\texttt{effect-of op}) \\

— Set up simp rules to keep use of local parameters transparent within proofs (i.e. automatically substitute definitions). \\
\textbf{lemma}[\textit{simp}]: \\
\textit{is-operator-applicable-in} \texttt{s} \texttt{op} = (\texttt{map-of} (\texttt{precondition-of op}) \subseteq_{m} \texttt{s}) \\
\texttt{s} \gg> \texttt{op} = \texttt{s} \gg> \texttt{map-of} (\texttt{effect-of op}) \\
\textbf{unfolding} initial-of-def goal-of-def variables-of-def range-of-def operators-of-def
lemma range-of-not-empty:

(sas-plus-problem.range-of \( \Psi \ v \neq None \land \) sas-plus-problem.range-of \( \Psi \ v \neq Some \)

\[ \leftarrow (R_+ \ \Psi \ v \neq \{\}) \]

apply (cases sas-plus-problem.range-of \( \Psi \ v \))

by (auto simp add: SAS-Plus-Representation.range-of'-def)

lemma is-valid-operator-sas-plus-then:

fixes \( \Psi::(v',d) \) sas-plus-problem

assumes is-valid-operator-sas-plus \( \Psi \) op

shows \( \forall (v, a) \in \) set \( \) precondition-of \( op \), \( v \in \) set \( (\Psi)V_+ \)

and \( \forall (v, a) \in \) set \( \) precondition-of \( op \), \( R_+ \ \Psi \ v \neq \{\} \land a \in R_+ \ \Psi \ v \)

and \( \forall (v, a) \in \) set \( \) effect-of \( op \), \( v \in \) set \( (\Psi)V_+ \)

and \( \forall (v, a) \in \) set \( \) effect-of \( op \), \( R_+ \ \Psi \ v \neq \{\} \land a \in R_+ \ \Psi \ v \)

and \( \forall (v, a) \in \) set \( \) precondition-of \( op \), \( \forall (v', a') \in \) set \( \) precondition-of \( op \). \( v \neq v' \lor a = a' \)

and \( \forall (v, a) \in \) set \( \) effect-of \( op \), \( v \neq v' \lor a = a' \)

proof --

let \( ?vs = \) sas-plus-problem.variables-of \( \Psi \)

and \( ?pre = \) precondition-of \( op \)

and \( ?eff = \) effect-of \( op \)

and \( ?D = \) sas-plus-problem.range-of \( \Psi \)

have \( \forall (v, a) \in \) set \( ?pre. \ v \in \) set \( ?vs \)

and \( \forall (v, a) \in \) set \( ?pre. \ (\ ?D v \neq None) \land \ a \in \) set \( \) (the \( \ ?D v \))

and \( \forall (v, a) \in \) set \( ?eff. \ v \in \) set \( ?vs \)

and \( \forall (v, a) \in \) set \( ?eff. \ (\ ?D v \neq None) \land \ a \in \) set \( \) (the \( \ ?D v \))

and \( \forall (v, a) \in \) set \( ?pre. \ \forall (v', a') \in \) set \( ?pre. \ v \neq v' \lor a = a' \)

and \( \forall (v, a) \in \) set \( ?eff. \ \forall (v', a') \in \) set \( ?eff. \ v \neq v' \lor a = a' \)

using asms

unfolding is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff

by meson+

moreover have \( \forall (v, a) \in \) set \( ?pre. \ v \in \) set \( (\Psi)V_+ \)

and \( \forall (v, a) \in \) set \( ?pre. \ v \in \) set \( (\Psi)V_+ \)

and \( \forall (v, a) \in \) set \( ?pre. \ \forall (v', a') \in \) set \( ?pre. \ v \neq v' \lor a = a' \)

and \( \forall (v, a) \in \) set \( ?eff. \ \forall (v', a') \in \) set \( ?eff. \ v \neq v' \lor a = a' \)

using calculation

unfolding variables-of-def

61
by blast+
moreover {
  have (∀(v, a) ∈ set ?pre. (?D v ≠ None) ∧ a ∈ set (the (?D v))
    using assms
    unfolding is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff
    by argo
  hence (∀(v, a) ∈ set ?pre. ((plaintext + v) ≠ {}) ∧ a ∈ + v)
    using range-of ′-def
    by fastforce
}
moreover {
  have (∀(v, a) ∈ set ?eff. (?D v ≠ None) ∧ a ∈ set (the (?D v))
    using assms
    unfolding is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff
    by argo
  hence (∀(v, a) ∈ set ?eff. ((plaintext + v) ≠ {}) ∧ a ∈ + v)
    using range-of ′-def
    by fastforce
}
ultimately show (∀(v, a) ∈ set (precondition-of op). v ∈ set ((plaintext + v))
  and (∀(v, a) ∈ set (precondition-of op). (plaintext + v) ≠ {}) ∧ a ∈ + v
  and (∀(v, a) ∈ set (effect-of op). v ∈ set ((plaintext + v))
  and (∀(v, a) ∈ set (effect-of op). (plaintext + v) ≠ {}) ∧ a ∈ + v
  and (∀(v, a) ∈ set (precondition-of op). (∀v′, a′) ∈ set (precondition-of op). v
    ≠ v′ ∨ a = a′
    and (∀(v, a) ∈ set (effect-of op).
      (∀v′, a′) ∈ set (effect-of op). v ≠ v′ ∨ a = a′
    by blast+
qed

lemma is-valid-problem-sas-plus-then:
fixes ?ψ::('v,'d) sas-plus-problem
assumes is-valid-problem-sas-plus ?ψ
shows ∀v ∈ set ((plaintext + v)). (plaintext + v) ≠ {}
  and ∀op ∈ set ((plaintext + v)). is-valid-operator-sas-plus ?op
  and dom ((plaintext + v)) = set ((plaintext + v))
  and ∀v ∈ dom ((plaintext + v)). the (((plaintext + v)) v) ∈ + v
  and dom ((plaintext + v)) ⊆ set ((plaintext + v))
  and ∀v ∈ dom ((plaintext + v)). the (((plaintext + v)) v) ∈ + v
proof –
let ?as = sas-plus-problem.variables-of ?ψ
  and ?ops = sas-plus-problem/operators-of ?ψ
  and ?I = sas-plus-problem/initial-of ?ψ
  and ?G = sas-plus-problem/goal-of ?ψ
  and ?D = sas-plus-problem/range-of ?ψ
{ fix v have (?D v ≠ None ∧ ?D v ≠ Some []) ⟷ ((plaintext + v) ≠ {})
by (cases ?D v; (auto simp: range-of'‐def))

} note nb = this

have nb1: "\forall v \in \text{set ?vs}. ?D v \neq \text{None}
and \forall op \in \text{set ?ops. is-valid-operator-sas-plus }\Psi op
and \forall v\. (?I v \neq \text{None} = (v \in \text{set ?vs})
and nb2: "\forall v\. ?I v \neq \text{None} \longrightarrow (\text{the } (?I v) \in \text{set } (\text{the } (?D v)))
and \forall v\. ?G v \neq \text{None} \longrightarrow v \in \text{set ?vs}
and nb3: "\forall v\. ?G v \neq \text{None} \longrightarrow (\text{the } (?G v) \in \text{set } (\text{the } (?D v)))

using assms

unfolding \text{SAS-Plus-Representation.is-valid-problem-sas-plus-def Let-def list-all-iff ListMem-iff}

by argo+

then have G3: "\forall op \in \text{set } ((\Psi)_{\mathcal{O}+}). is-valid-operator-sas-plus \Psi op
and G4: dom ((\Psi)_{\mathcal{T}+}) = set ((\Psi)_{\mathcal{V}+})
and G5: dom ((\Psi)_{\mathcal{G}+}) \subseteq set ((\Psi)_{\mathcal{V}+})

unfolding \text{variables-of-def operators-of-def}

by auto+

moreover {
fix v
assume v \in \text{set } ((\Psi)_{\mathcal{V}+})
then have ?D v \neq \text{None}

using nb1

by force+
}

} note G6 = this

moreover {
fix v
assume v \in \text{dom } ((\Psi)_{\mathcal{T}+})

moreover have ((\Psi)_{\mathcal{T}+}) v \neq \text{None}

using calculation

by blast+

moreover {
fix v \in \text{set } ((\Psi)_{\mathcal{V}+})

using G4 calculation(1)

by argo

then have sas-plus-problem.range-of \Psi v \neq \text{None}

using range-of-not-empty

unfolding range-of'‐def

using G6

by fast+

hence set (\text{the } (?D v)) = \mathcal{R}_+ \Psi v

by (simp add: \text{sas-plus-problem.range-of }\Psi v \neq \text{None \ option.case-eq-if range-of'‐def})

}

ultimately have the ((\Psi)_{\mathcal{T}+}) v \in \mathcal{R}_+ \Psi v

using nb2

by force

}

moreover {
fix v

}
assume \( v \in \text{dom} \ (\Psi)_{G^+} \) 
then have \( ((\Psi)_{G^+}) \ v \neq \text{None} \) 
by blast

moreover {
have \( v \in \text{set} \ (\Psi)_{V^+} \)
using \( G5 \) calculation(1)
by fast
then have \( \text{sas-plus-problem.range-of} \ (\Psi) \ v \neq \text{None} \)
using \( \text{range-of-not-empty} \)
using \( G6 \)
by fast+
hence set (the \((\Psi) \ v\)) = \( R^+_+ \Psi \ v \)
by (simp add: \( \text{sas-plus-problem.range-of} \ (\Psi) \ v \neq \text{None} \) option.case-eq-if range-of'-def)

} ultimately have the \( (((\Psi)_{G^+}) \ v) \in R^+_+ \Psi \ v \)
using \( nb_3 \)
by auto

} ultimately show \( \forall v \in \text{set} \ (\Psi)_{V^+} \). (\( R^+_+ \Psi \ v \) \neq \{ \} 
and \( \forall \text{op} \in \text{set} \ (\Psi)_{O^+} \). \( \text{is-valid-operator-sas-plus} \ (\Psi) \text{ op} \)
and \( \text{dom} \ (\Psi)_{I^+} \) = \( \text{set} \ (\Psi)_{V^+} \)
and \( \forall v \in \text{dom} \ (\Psi)_{I^+} \). \( \text{the} \ ((\Psi)_{I^+}) \ v \in R^+_+ \Psi \ v \)
and \( \text{dom} \ (\Psi)_{G^+} \subseteq \text{set} \ (\Psi)_{V^+} \)
and \( \forall v \in \text{dom} \ (\Psi)_{G^+} \). \( \text{the} \ ((\Psi)_{G^+}) \ v \in R^+_+ \Psi \ v \)
by blast+
qed

end

theory SAS-Plus-Semantics
imports SAS-Plus-Representation List-Supplement
Map-Supplement
begin

5 SAS+ Semantics

5.1 Serial Execution Semantics

Serial plan execution is implemented recursively just like in the STRIPS case. 
By and large, compared to definition ??, we only substitute the operator 
applicability function with its SAS+ counterpart.

primrec execute-serial-plan-sas-plus
where execute-serial-plan-sas-plus \( s \ [] = s \)
\begin{align*}
| \text{execute-serial-plan-sas-plus} \ s \ (\text{op} \neq \text{ops}) & = \text{if is-operator-applicable-in} \ s \ \text{op} \\
\text{then} \text{execute-serial-plan-sas-plus} \ (\text{execute-operator-sas-plus} \ s \ \text{op}) \ \text{ops} \\
\text{else} \ s
\end{align*}

end
Similarly, serial SAS+ solutions are defined just like in STRIPS but based on the corresponding SAS+ definitions.

**definition** is-serial-solution-for-problem
:: ('variable, 'domain) sas-plus-problem ⇒ ('variable, 'domain) sas-plus-plan ⇒ bool

**where** is-serial-solution-for-problem Ψ ψ
≡ let
  I = sas-plus-problem.initial-of Ψ
  ; G = sas-plus-problem.goal-of Ψ
  ; ops = sas-plus-problem.operators-of Ψ
in G ⊆ m execute-serial-plan-sas-plus I ψ ∧ list-all (λop. ListMem op ops) ψ

**context**

**begin**

**private lemma** execute-operator-sas-plus-effect-i:

assumes is-operator-applicable-in s op and ∀(v, a) ∈ set (effect-of op). ∀(v', a') ∈ set (effect-of op).

shows (s ⊳ op) v = Some a

**proof** –

let ?effect = effect-of op

have map-of ?effect v = Some a using map-of-constant-assignments-defined-if[OF assms(2, 3)] try0 by blast

thus ?thesis

unfolding execute-operator-sas-plus-def map-add-def by fastforce

**qed**

**private lemma** execute-operator-sas-plus-effect-ii:

assumes is-operator-applicable-in s op and ∀(v', a') ∈ set (effect-of op). v' ≠ v

shows (s ⊳ op) v = s v

**proof** –

let ?effect = effect-of op

{ have v ∉ fst ' set ?effect
  using assms(2)
  by fastforce
  then have v ∉ dom (map-of ?effect)
  using dom-map-of-conv-image-fst[of ?effect]
  by argo
  hence (s ++ map-of ?effect) v = s v
  using map-add-dom-app-simps(3)[of v map-of ?effect s]
  by blast

65
Given an operator \( op \) that is applicable in a state \( s \) and has a consistent set of effects (second assumption) we can now show that the successor state \( s' \equiv s \gg_+ op \) has the following properties:

- \( s' v = \text{Some } a \) if \( (v, a) \) exist in \( \text{set (effect-of } op) \); and,
- \( s' v = s v \) if no \( (v, a') \) exist in \( \text{set (effect-of } op) \).

The second property is the case if the operator doesn’t have an effect for a variable \( v \).

**Theorem execute-operator-sas-plus-effect:**

** Assumes ** is-operator-applicable-in \( s \) \( op \)

and \( \forall (v, a) \in \text{set (effect-of } op) \).

\( \forall (v', a') \in \text{set (effect-of } op) \). \( v \neq v' \, \forall \ a = a' \)

** Shows ** \( (v, a) \in \text{set (effect-of } op) \)

\( \rightarrow (s \gg_+ op) \, v = \text{Some } a \)

and \( (\forall a, (v, a) \notin \text{set (effect-of } op)) \)

\( \rightarrow (s \gg_+ op) \, v = s v \)

** Proof **

show \( (v, a) \in \text{set (effect-of } op) \)

\( \rightarrow (s \gg_+ op) \, v = \text{Some } a \)

using execute-operator-sas-plus-effect-i[OF assms(1, 2)]

by blast

next

show \( (\forall a, (v, a) \notin \text{set (effect-of } op)) \)

\( \rightarrow (s \gg_+ op) \, v = s v \)

using execute-operator-sas-plus-effect-ii[OF assms(1)]

by blast

qed

end

5.2 Parallel Execution Semantics

— Define a type synonym for SAS+ parallel plans and add a definition lifting SAS+ operator applicability to parallel plans.

** Type-synonym ** \( (\text{variable, domain}) \text{ sas-plus-parallel-plan} \)

\( = (\text{variable, domain}) \text{ sas-plus-operator list list} \)

** Definition ** are-all-operators-applicable-in

\( : (\text{variable, domain}) \text{ state} \)

\( \Rightarrow (\text{variable, domain}) \text{ sas-plus-operator list} \)

\( \Rightarrow \text{bool} \)
where are-all-operators-applicable-in s ops
≡ list-all (is-operator-applicable-in s) ops

definition are-operator-effects-consistent
:: ('variable, 'domain) sas-plus-operator
⇒ ('variable, 'domain) sas-plus-operator
⇒ bool
where are-operator-effects-consistent op op'
≡ let
  effect = effect-of op
 ; effect' = effect-of op
in list-all (λ(v, a). list-all (λ(v', a'). v ≠ v' ∨ a = a') effect') effect

definition are-all-operator-effects-consistent
:: ('variable, 'domain) sas-plus-operator list
⇒ bool
where are-all-operator-effects-consistent ops
≡ list-all (λop. list-all (are-operator-effects-consistent op) ops) ops

definition execute-parallel-operator-sas-plus
:: ('variable, 'domain) state
⇒ ('variable, 'domain) sas-plus-operator list
⇒ ('variable, 'domain) state
where execute-parallel-operator-sas-plus s ops
≡ foldl (++) s (map (map-of ◦ effect-of) ops)

We now define parallel execution and parallel traces for SAS+ by lifting
the tests for applicability and effect consistency to parallel SAS+ operators.
The definitions are again very similar to their STRIPS analogs (definitions ?? and ??).

fun execute-parallel-plan-sas-plus
:: ('variable, 'domain) state
⇒ ('variable, 'domain) sas-plus-parallel-plan
⇒ ('variable, 'domain) state
where execute-parallel-plan-sas-plus s [] = s
| execute-parallel-plan-sas-plus s (ops # opss) = (if
  are-all-operators-applicable-in s ops
 ∧ are-all-operator-effects-consistent ops
then execute-parallel-plan-sas-plus
  (execute-parallel-operator-sas-plus s ops) opss
else s)

fun trace-parallel-plan-sas-plus
:: ('variable, 'domain) state
⇒ ('variable, 'domain) sas-plus-parallel-plan
⇒ ('variable, 'domain) state list
where trace-parallel-plan-sas-plus s [] = [s]
| trace-parallel-plan-sas-plus s (ops # opss) = s # (if
  are-all-operators-applicable-in s ops
∧ are-all-operator-effects-consistent ops
\text{then trace-parallel-plan-sas-plus}
\text{(execute-parallel-operator-sas-plus s ops) opss}
else []\)

A plan \( \psi \) is a solution for a SAS+ problem \( \Psi \) if

1. starting from the initial state \( \Psi \), SAS+ parallel plan execution reaches a state which satisfies the described goal state \( \Psi_G^+ \); and,

2. all parallel operators \( \text{ops} \) in the plan \( \psi \) only consist of operators that are specified in the problem description.

definition is-parallel-solution-for-problem
:: ('variable, 'domain) sas-plus-problem \Rightarrow ('variable, 'domain) sas-plus-parallel-plan \Rightarrow bool
where is-parallel-solution-for-problem \( \Psi \ \psi \)
eq let
\( G = \text{sas-plus-problem}.\text{goal-of} \ \Psi \);
\( I = \text{sas-plus-problem}.\text{initial-of} \ \Psi \);
\( \text{Ops} = \text{sas-plus-problem}.\text{operators-of} \ \Psi \);
in \( G \subseteq_m \text{execute-parallel-plan-sas-plus I} \ \psi \)
\& list-all (\lambda \text{ops}. list-all (\lambda \text{op}. \text{ListMem op Ops}) \text{ops}) \psi

context
begin

\textbf{lemma} execute-parallel-operator-sas-plus-cons\text{[simp]}:
\text{execute-parallel-operator-sas-plus s (op \# opss)}
= \text{execute-parallel-operator-sas-plus (s ++ map-of (effect-of op)) opss}
\textbf{unfolding} execute-parallel-operator-sas-plus-def
\textbf{by} simp

The following lemmas show the properties of SAS+ parallel plan execution traces. The results are analogous to those for STRIPS. So, let \( \tau \equiv \text{trace-parallel-plan-sas-plus I} \ \psi \) be a trace of a parallel SAS+ plan \( \psi \) with initial state \( I \), then

- the head of the trace \( \tau ! 0 \) is the initial state of the problem (lemma ??); moreover,

- for all but the last element of the trace—i.e. elements with index \( k < \text{length} \ \tau - 1 \)—the parallel operator \( \pi ! k \) is executable (lemma ??); and finally,

- for all \( k < \text{length} \ \tau \), the parallel execution of the plan prefix \( \text{take k} \ \psi \) with initial state \( I \) equals the \( k \)-th element of the trace \( \tau ! k \) (lemma ??).
lemma \textit{trace-parallel-plan-sas-plus-head-is-initial-state:}
\textit{trace-parallel-plan-sas-plus} I \psi ! 0 = I
definition \textit{proof (cases }\psi\textit{)}
deinduction \textit{case (Cons a list)}
\textit{then show }\psi\textit{thesis}
\textit{by (cases are-all-operators-applicable-in I a \& are-all-operator-effects-consistent a;}
\textit{simp+)}
\textit{qed simp}

lemma \textit{trace-parallel-plan-sas-plus-length-gt-one-if:}
\textit{assumes }k < \textit{length }\textit{(trace-parallel-plan-sas-plus} I \psi) - 1\textit{ shows }I < \textit{length }\textit{(trace-parallel-plan-sas-plus} I \psi)
definition \textit{using assms by linarith}

lemma \textit{length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one:}
\textit{shows }\textit{length }\textit{(trace-parallel-plan-sas-plus} I \psi) \leq \textit{length }\psi + 1\textit{ proves (induction }\psi\textit{ arbitrary: I)}
deinduction \textit{case (Cons a }\psi\textit{)}
\textit{then show }\psi\textit{case}
definition \textit{proof (cases are-all-operators-applicable-in I a \& are-all-operator-effects-consistent a;}
\textit{simp+)}
\textit{let }\textit{?I}′ = \textit{execute-parallel-operator-sas-plus} I a
\textit{\{ have }\textit{trace-parallel-plan-sas-plus} I (a \# }\psi\textit{) = I \# }\textit{trace-parallel-plan-sas-plus} ?I′ \psi
\textit{using True by auto then have }\textit{length }\textit{(trace-parallel-plan-sas-plus} I (a \# }\psi\textit{)) = \textit{length }\textit{(trace-parallel-plan-sas-plus} ?I′ \psi) + 1
\textit{by simp moreover have }\textit{length }\textit{(trace-parallel-plan-sas-plus} ?I′ \psi) \leq \textit{length }\psi + 1
\textit{using Cons.IH[of }?I′\textit{] by blast ultimately have }\textit{length }\textit{(trace-parallel-plan-sas-plus} I (a \# }\psi\textit{)) \leq \textit{length (a \# }\psi\textit{)} + 1
\textit{by simp \}}
\textit{thus }\psi\textit{thesis by blast qed auto}
\textit{qed simp}

lemma \textit{plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements:}
\textit{assumes }k < \textit{length }\textit{(trace-parallel-plan-sas-plus} I \psi) - 1\textit{ obtains }\textit{ops }\psi' \textit{where }\psi = \textit{ops \# }\psi'\textit{ proves –}
let \( ?\tau = \text{trace-parallel-plan-sas-plus } I \psi \)

have length \( ?\tau \leq \text{length } \psi + 1 \)
  using length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one
  by fast
then have \( 0 < \text{length } \psi \)
  using trace-parallel-plan-sas-plus-length-gt-one-if[of assms]
  by fastforce
then obtain \( k' \) where length \( \psi = \text{Suc } k' \)
  using gr0-implies-Suc
  by meson
thus \( ?\text{thesis} \) using that
  using length-Suc-conv[of \( \psi \) \( k' \)]
  by blast
qed

lemma trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds:
assumes \( k < \text{length } (\text{trace-parallel-plan-sas-plus } I \pi) - 1 \)
shows are-all-operators-applicable-in (trace-parallel-plan-sas-plus I \pi ! k) (\pi ! k)
\land are-all-operator-effects-consistent (\pi ! k)
using assms
proof (induction \( \pi \) arbitrary: \( I k \))
  — NOTE Base case yields contradiction with assumption and can be left to automation.
case (Cons a \( \pi \))
then show \( ?\text{case} \)
  proof (cases are-all-operators-applicable-in \( I a \cap \text{are-all-operator-effects-consistent } a \))
    case True
    have trace-parallel-plan-sas-plus-cons: trace-parallel-plan-sas-plus I (a \# \pi)
      = I \# trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) \pi
      using True
      by simp
    then show \( ?\text{thesis} \)
      proof (cases \( k \))
        case 0
        have trace-parallel-plan-sas-plus I (a \# \pi) ! 0 = I
          using trace-parallel-plan-sas-plus-cons
          by simp
        moreover have (a \# \pi) ! 0 = a
          by simp
        ultimately show \( ?\text{thesis} \)
          using True 0
          by presburger
next
case (Suc \( k' \))
have trace-parallel-plan-sas-plus I (a \# \pi) ! Suc \( k' \)
  = trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) \pi ! k'
  using trace-parallel-plan-sas-plus-cons
  by simp

70
moreover have \( (a \neq \pi) \land Suc \ k' = \pi \land k' \)
  by simp

moreover \{
  let \(?I' = \text{execute-parallel-operator-sas-plus} \ I \ a\)
  have \(\text{length} (\text{trace-parallel-plan-sas-plus} \ ?I') = I + \text{length} (\text{trace-parallel-plan-sas-plus} \ ??I' \ \pi)\)
  using \(\text{trace-parallel-plan-sas-plus-cons}\)
  by auto
  then have \(k' < \text{length} (\text{trace-parallel-plan-sas-plus} \ ?I' \ \pi) - 1\)
  using Cons.prems Suc
  unfolding Suc.eq-plus1
  by fastforce
  hence \(\text{are-all-operators-applicable-in} \ (\text{trace-parallel-plan-sas-plus} (\text{execute-parallel-operator-sas-plus} \ I \ a) \ \pi \land k') \land \text{are-all-operator-effects-consistent} (\pi \land k')\)
  using Cons.IH[of \(k' \ \text{execute-parallel-operator-sas-plus} \ I \ a\] Cons.prems Suc
  trace-parallel-plan-sas-plus-cons
  by simp
  }
ultimately show \(?thesis\)
  using Suc
  by argo
qed

next
  case False
  then have \(\text{trace-parallel-plan-sas-plus} \ I \ (a \neq \pi) = [I]\)
  by force
  then have \(\text{length} (\text{trace-parallel-plan-sas-plus} \ I \ (a \neq \pi)) - 1 = 0\)
  by simp
  — NOTE Thesis follows from contradiction with assumption.
  then show \(?thesis\)
  using Cons.prems
  by force
qed

lemma trace-parallel-plan-sas-plus-prefix:
  assumes \(k < \text{length} (\text{trace-parallel-plan-sas-plus} \ I \ \psi)\)
  shows \(\text{trace-parallel-plan-sas-plus} \ I \ \psi ! k = \text{execute-parallel-plan-sas-plus} \ I \ (\text{take} \ k \ \psi)\)
  using assms
proof (induction \(\psi\) arbitrary: \(I \ k\))
  case (Cons \(a \ \psi\))
  then show \(?case\)
  proof (cases \(\text{are-all-operators-applicable-in} \ I \ a \land \text{are-all-operator-effects-consistent} \ a\))
    case True
    let \(?\sigma = \text{trace-parallel-plan-sas-plus} \ I \ (a \neq \psi)\)
and $\sigma' = \text{execute-parallel-operator-sas-plus } I a$

have $\sigma$-equals: $\sigma = I \# \text{trace-parallel-plan-sas-plus } ?I' \psi$
using True
by auto
then show $?thesis$
proof (cases $k = 0$)
case False
obtain $k'$ where $k$-is-suc-of-$k'$: $k = \text{Suc } k'$
using not0-implies-Suc[OF False]
by blast
then have execute-parallel-plan-sas-plus $I$ (take $k \ (a \ # \ \psi))$
using True
by simp
moreover have trace-parallel-plan-sas-plus $I$ (a $\# \ \psi) ! k$
= execute-parallel-plan-sas-plus $?I' \ (take k' \ \psi)$
using $\sigma$-equals k-is-suc-of-$k'$
by simp
moreover {
have $k' < \text{length } (\text{trace-parallel-plan-sas-plus } ?I' \ \psi)$
using Cons.prems $\sigma$-equals k-is-suc-of-$k'$
by force
hence trace-parallel-plan-sas-plus $?I' \ \psi \ ! \ k'$
= execute-parallel-plan-sas-plus $?I' \ (take k' \ \psi)$
using Cons.IH[of $k' \ ?I' \ ?I'$]
by blast
}
ultimately show $?thesis$
by presburger
qed simp
next
case operator-precondition-violated: False
then show $?thesis$
proof (cases $k = 0$)
case False
then have trace-parallel-plan-sas-plus $I$ (a $\# \ \psi) = [I]$
using operator-precondition-violated
by force
moreover have execute-parallel-plan-sas-plus $I$ (take $k \ (a \ # \ \psi)) = I$
using Cons.prems operator-precondition-violated
by force
ultimately show $?thesis$
using Cons.prems nth-Cons-0
by auto
qed simp
qed simp

lemma trace-parallel-plan-sas-plus-step-effect-is:
assumes $k < length (trace-parallel-plan-sas-plus I \psi) − 1$

shows trace-parallel-plan-sas-plus I \psi \! Suc k

= execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I \psi \! k) (\psi \! k)

proof

let $?\tau = trace-parallel-plan-sas-plus I \psi$

let $?\tau_k = ?\tau \! k$

and $?\tau_k' = ?\tau \! Suc k$

— NOTE rewrite the goal using the subplan formulation to be able. This allows us to make the initial state arbitrary.

{ have suc-k-lt-length-\tau: Suc k < length $?\tau$
  using assms by linarith
  hence $?\tau_k' = execute-parallel-plan-sas-plus I (take (Suc k) \psi)$
  using trace-parallel-plan-sas-plus-prefix[of Suc k]
  by blast
  } note rewrite-goal = this

have execute-parallel-plan-sas-plus I (take (Suc k) \psi)

= execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I \psi \! k) (\psi \! k)

using assms

proof (induction k arbitrary: I \psi)

  case 0
  obtain ops $\psi'$ where $\psi$-is: $\psi = ops \# \psi'$
    using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF 0.prems]
    by force
    { have take (Suc 0) $\psi = [\psi \! 0]$
      using $\psi$-is
      by simp
      hence execute-parallel-plan-sas-plus I (take (Suc 0) \psi)
      = execute-parallel-plan-sas-plus I [\psi \! 0]
      by argo
    }
  moreover 
  have trace-parallel-plan-sas-plus I \psi \! 0 = I
    using trace-parallel-plan-sas-plus-head-is-initial-state.
  moreover 
  have are-all-operators-applicable-in I (\psi \! 0)
    and are-all-operator-effects-consistent (\psi \! 0)
  using trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds[OF 0.prems] calculation
    by argo+
  then have execute-parallel-plan-sas-plus I [\psi \! 0]
    = execute-parallel-operator-sas-plus I (\psi \! 0)
    by simp
  }

73
ultimately have \texttt{execute-parallel-operator-sas-plus} (\texttt{trace-parallel-plan-sas-plus} \\
\qquad I \psi ! 0) \\
\qquad \psi ! 0) \\
\qquad = \texttt{execute-parallel-plan-sas-plus} I [\psi ! 0] \\
\qquad \texttt{by argo} \\
\} \\
\texttt{ultimately show} ?\texttt{case} \\
\qquad \texttt{by argo} \\
\texttt{next} \\
\texttt{case} (\texttt{Suc} \ k) \\
\texttt{obtain} \ \texttt{ops} \ \psi' \ \texttt{where} \ \psi-\texttt{is}: \ \psi = \texttt{ops} \neq \psi' \\
\qquad \texttt{using} \ \texttt{plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements}[\texttt{OF} \ \texttt{Suc.prems}] \\
\qquad \texttt{by blast} \\
\texttt{let} \ ?I' = \texttt{execute-parallel-operator-sas-plus} \ \texttt{ops} \ \texttt{I} \\
\texttt{have} \ \texttt{execute-parallel-plan-sas-plus} \ \texttt{I} \ (\texttt{take} \ (\texttt{Suc} \ (\texttt{Suc} \ k)) \ \psi) \\
\qquad = \texttt{execute-parallel-plan-sas-plus} \ ?I' \ (\texttt{take} \ (\texttt{Suc} \ k) \ \psi') \\
\qquad \texttt{using} \ \texttt{Suc.prems} \ \psi-\texttt{is} \\
\qquad \texttt{by fastforce} \\
\texttt{moreover} \{ \\
\qquad \texttt{thm} \ \texttt{Suc.IH}[\texttt{of}] \\
\qquad \texttt{have} \ \texttt{length} \ (\texttt{trace-parallel-plan-sas-plus} \ I \ \psi) \\
\qquad \quad = 1 + \texttt{length} \ (\texttt{trace-parallel-plan-sas-plus} \ ?I' \ \psi') \\
\qquad \quad \texttt{using} \ \psi-\texttt{is} \ \texttt{Suc.prems} \\
\qquad \quad \texttt{by fastforce} \\
\qquad \texttt{moreover have} \ k < \texttt{length} \ (\texttt{trace-parallel-plan-sas-plus} \ ?I' \ \psi') - 1 \\
\qquad \quad \texttt{using} \ \texttt{Suc.prems} \ \texttt{calculation} \\
\qquad \quad \texttt{by fastforce} \\
\qquad \texttt{ultimately have} \ \texttt{execute-parallel-plan-sas-plus} \ ?I' \ (\texttt{take} \ (\texttt{Suc} \ k) \ \psi') = \\
\qquad \quad \texttt{execute-parallel-operator-sas-plus} \ (\texttt{trace-parallel-plan-sas-plus} ?I' \ \psi' ! k) \\
\qquad \quad (\psi' ! k) \\
\qquad \quad \texttt{using} \ \texttt{Suc.IH}[\texttt{of} \ ?I' \ \psi'] \\
\qquad \quad \texttt{by blast} \\
\} \\
\texttt{moreover have} \ \texttt{execute-parallel-operator-sas-plus} \ (\texttt{trace-parallel-plan-sas-plus} \\
\qquad ?I' \ \psi' ! k) \\
\qquad (\psi' ! k) \\
\qquad = \texttt{execute-parallel-operator-sas-plus} \ (\texttt{trace-parallel-plan-sas-plus} \ I \ \psi ! \texttt{Suc} \ k) \\
\qquad (\psi ! \texttt{Suc} \ k) \\
\qquad \texttt{using} \ \texttt{Suc.prems} \ \psi-\texttt{is} \\
\qquad \texttt{by auto} \\
\texttt{ultimately show} ?\texttt{case} \\
\qquad \texttt{by argo} \\
\texttt{qed} \\
\texttt{thus} ?\texttt{thesis} \\
\qquad \texttt{using} \ \texttt{rewrite-goal} \\
\qquad \texttt{by argo} \\
\texttt{qed} \\

Finally, we obtain the result corresponding to lemma ?? in the SAS+ case:
it is equivalent to say that parallel SAS+ execution reaches the problem’s goal state and that the last element of the corresponding trace satisfies the goal state.

**Lemma** execute-parallel-plan-sas-plus-reaches-goal-iff-goal-is-last-element-of-trace:

\[ G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi \quad \iff \quad G \subseteq_m \text{last (trace-parallel-plan-sas-plus } I \psi) \]

**Proof**

let \(?\tau = \text{trace-parallel-plan-sas-plus } I \psi\)

**Show** \(?\text{thesis}\)

**Proof** (rule \(\text{iffI}\))

assume \(G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi\)

thus \(G \subseteq_m \text{last } \tau\)

**Proof** (induction \(\psi\) arbitrary: \(I\))

— NOTE Base case follows from simplification.

case (\(\text{Cons } ops \psi\))

**Show** \(?\text{thesis}\)

**Proof** (cases \(\text{are-all-operators-applicable-in } I \text{ ops}\) \(\land\) \(\text{are-all-operator-effects-consistent } ops\))

case True

let \(?s = \text{execute-parallel-operator-sas-plus } I \text{ ops}\)

\{ have \(G \subseteq_m \text{execute-parallel-plan-sas-plus } ?s \psi\)

using True Cons.prems

by simp

hence \(G \subseteq_m \text{last } \tau\)

by auto

\} moreover \{

have \(\text{trace-parallel-plan-sas-plus } I \text{ (ops } \# \psi)\)

\(= I \# \text{trace-parallel-plan-sas-plus } ?s \psi\)

using True

by simp

moreover have \(\text{trace-parallel-plan-sas-plus } ?s \psi \neq []\)

using \(\text{trace-parallel-plan-sas-plus.elims}\)

by blast

ultimately have \(\text{last } \tau\)

by simp

ultimately show \(?\text{thesis}\)

by argo

next

case False

then have \(G \subseteq_m I\)

using Cons.prems

by force

thus \(?\text{thesis}\)
using False
by force
qed
qed force
next
assume \( G \subseteq_m \tau \)
thus \( G \subseteq_m \text{execute-parallel-plan-sas-plus} \ I \ \psi \)
proof (induction \( \psi \) arbitrary: \( I \))
case (\( \text{Cons} \ ops \ \psi \))
thus ?case
proof (cases are-all-operators-applicable-in \( I \) \( \text{ops} \) \( \land \) are-all-operator-effects-consistent \( \text{ops} \))
case True
let ?s = \( \text{execute-parallel-operator-sas-plus} \ I \ \text{ops} \)\{ have trace-parallel-plan-sas-plus \( \text{ops} \ # \ \psi \) = \( I \ # \ \text{trace-parallel-plan-sas-plus} \ ?s \ \psi \) using True by simp
moreover have trace-parallel-plan-sas-plus ?s \( \psi \) \( \neq \) [] using trace-parallel-plan-sas-plus.elims by blast
ultimately have last (trace-parallel-plan-sas-plus \( \text{ops} \ # \ \psi \)) = last (trace-parallel-plan-sas-plus ?s \( \psi \)) using last-ConsR by simp hence \( G \subseteq_m \text{execute-parallel-plan-sas-plus} \ ?s \ \psi \) using Cons.IH[of ?s] Cons.prems by argo\} moreover have \( \text{execute-parallel-plan-sas-plus} \ \text{ops} \ # \ \psi \) = \( \text{execute-parallel-plan-sas-plus} \ ?s \ \psi \) using True by force ultimately show ?thesis by argo
next
case False
have \( G \subseteq_m \ I \) using Cons.prems False by simp thus ?thesis using False by force qed qed simp qed
lemma is-parallel-solution-for-problem-plan-operator-set:

fixes Ψ :: (′v, ′d) sas-plus-problem
assumes is-parallel-solution-for-problem Ψ ψ
shows ∀ ops ∈ set ψ. ∀ op ∈ set ops. op ∈ set ((Ψ)O+)
using assms
unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff operators-of-def
by presburger
end

5.3 Serializable Parallel Plans

Again we want to establish conditions for the serializability of plans. Let Ψ be a SAS+ problem instance and let ψ be a serial solution. We obtain the following two important results, namely that

1. the embedding List-Supplement.embed ψ of ψ is a parallel solution for Ψ (lemma ??); and conversely that,

2. a parallel solution to Ψ that has the form of an embedded serial plan can be concatenated to obtain a serial solution (lemma ??).

class context
begin

lemma execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i:
assumes is-operator-applicable-in s op
are-operator-effects-consistent op op
shows s □ s op = execute-parallel-operator-sas-plus s [op]
proof –
have are-all-operators-applicable-in s [op]
unfolding are-all-operators-applicable-in-def
SAS-Plus-Representation.execute-operator-sas-plus-def
is-operator-applicable-in-def SAS-Plus-Representation.is-operator-applicable-in-def
list-all-iff
using assms(1)
by fastforce
moreover have are-all-operator-effects-consistent [op]
unfolding are-all-operator-effects-consistent-def list-all-iff
using assms(2)
by fastforce
ultimately show ?thesis
unfolding execute-parallel-operator-sas-plus-def execute-operator-sas-plus-def
by simp
qed

77
lemma execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii:
fixes I :: ('variable, 'domain) state
assumes ∀ op ∈ set ψ. are-operator-effects-consistent op op
and G ⊆ m execute-serial-plan-sas-plus I ψ
shows G ⊆ m execute-parallel-plan-sas-plus I (embed ψ)
using assms
proof (induction ψ arbitrary: I)
case (Cons op ψ)
show ?case
proof (cases are-all-operators-applicable-in I [op])
case True
let ?J = execute-operator-sas-plus I op
let ?J' = execute-parallel-operator-sas-plus I [op]
have SAS-Plus-Representation.is-operator-applicable-in I op
  using True
  unfolding are-all-operators-applicable-in-def list-all-iff
  by force
moreover have G ⊆ m execute-serial-plan-sas-plus ?J ψ
  using Cons.prems(2) calculation(1)
  by simp
moreover have are-all-operator-effects-consistent [op]
  unfolding are-all-operator-effects-consistent-def list-all-iff Let-def
  using Cons.prems(1)
  by simp
moreover have execute-parallel-plan-sas-plus I ([op] # embed ψ)
  = execute-parallel-plan-sas-plus ?J' (embed ψ)
  using True calculation(3)
  by simp
moreover {
  have is-operator-applicable-in I op
    are-operator-effects-consistent op op
    using True Cons.prems(1)
    unfolding are-all-operators-applicable-in-def
    SAS-Plus-Representation.is-operator-applicable-in-def list-all-iff
    by fastforce+
  hence ?J = ?J'
  using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i
  calculation(1)
  by blast
}
ultimately show ?thesis
  using Cons.IH[of ?J] Cons.prems(1)
  by simp
next
case False
moreover have ~is-operator-applicable-in I op
  using calculation
  unfolding are-all-operators-applicable-in-def
  SAS-Plus-Representation.is-operator-applicable-in-def list-all-iff
by fastforce
moreover have $G \subseteq I$
  using Cons.prems(2) calculation(2)
  unfolding is-operator-applicable-in-def
by simp
moreover have $execute-parallel-plan-sas-plus I ([op] \# embed \psi) = I$
  using calculation(1)
  by fastforce
ultimately show ?thesis
  by force
qed
qed simp

lemma $execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus$-iii:
assumes is-valid-problem-sas-plus $\Psi$
  and is-serial-solution-for-problem $\Psi \psi$
  and $op \in \text{set } \psi$
shows are-operator-effects-consistent $op \ op$
proof –
have $op \in \text{set } ((\Psi)_{O+})$
  using assms(2) assms(3)
  unfolding is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
  by fastforce
then have is-valid-operator-sas-plus $\Psi \ op$
  using is-valid-problem-sas-plus-then(2) assms(1, 3)
  by auto
thus ?thesis
  unfolding are-operator-effects-consistent-def Let-def list-all-iff ListMem-iff
  using is-valid-operator-sas-plus-then(6)
  by fast
qed

lemma $execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus$-iv:
fixes $\Psi :: (\\'v, \\'d) sas-plus-problem$
assumes $\forall \ op \in \text{set } \psi. \ op \in \text{set } ((\Psi)_{O+})$
shows $\forall \ op \in \text{set } \{ \text{embed } \psi \}. \ \forall \ op \in \text{set } \opps. \ op \in \text{set } ((\Psi)_{O+})$
proof –
let $\psi' = \text{embed } \psi$
have nb: set $?\psi' = \{ [op] | \ op. \ op \in \text{set } \psi \}$
  by (induction $\psi$; force)
{  
  fix $\ op$
  assume $\ op \in \text{set } ?\psi'$
  moreover obtain $\ op$ where $\ opss = [\ op] \ and \ \ op \in \text{set } ((\Psi)_{O+})$
    using assms(1) nb calculation
    by blast
  ultimately have $\forall \ op \in \text{set } \opps. \ op \in \text{set } ((\Psi)_{O+})$
  by fastforce
}
thus \( \text{?thesis} \).

\textbf{qed}

\textbf{theorem} \text{execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus}:
\begin{itemize}
  \item \text{assumes} \text{is-valid-problem-sas-plus} \( \Psi \)
  \item \text{and} \text{is-serial-solution-for-problem} \( \Psi, \psi \)
  \item \text{shows} \text{is-parallel-solution-for-problem} \( \Psi (\text{embed} \ \psi) \)
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item \text{let} \( \psi' = \text{embed} \ \psi \)
  \item \text{have} \( (\Psi) G^+ \subseteq_m \text{execute-serial-plan-sas-plus} (\Psi I^+) \psi \)
    \begin{itemize}
      \item \text{using} \text{assms}(2)
      \item \text{unfolding} \text{is-serial-solution-for-problem-def Let-def}
      \item \text{by simp}
    \end{itemize}
  \end{itemize}

\textbf{moreover} \text{have} \( \forall \text{op} \in \text{set} \ \psi. \text{arc-operator-effects-consistent} \ \text{op} \ \text{op} \)

\textbf{ultimately have} \( (\Psi) G^+ \subseteq_m \text{execute-parallel-plan-sas-plus} (\Psi I^+) \ ?\psi' \)

\textbf{by blast}

\textbf{moreover} \( \forall \text{op} \in \text{set} \ \psi. \text{op} \in \text{set} (\Psi O^+) \)

\textbf{by fastforce}

\textbf{hence} \( \forall \text{ops} \in ?\psi'. \forall \text{op} \in \text{ops} \ \text{op} \in \text{set} (\Psi O^+) \)

\textbf{by blast}

\textbf{ultimately show} \( \text{?thesis} \)

\textbf{goal-of-def}
\begin{itemize}
  \item \text{initial-of-def}
  \item \text{by fastforce}
\end{itemize}

\textbf{qed}

\textbf{lemma} \text{flattening-lemma-i}:
\begin{itemize}
  \item \text{fixes} \( \Psi :: (\nu, \ d) \text{sas-plus-problem} \)
  \item \text{assumes} \( \forall \text{ops} \in \text{set} \ \pi. \ \forall \text{op} \in \text{ops} \ \text{op} \in \text{set} (\Psi O^+) \)
  \item \text{shows} \( \forall \text{op} \in \text{set} (\text{concat} \ \pi). \ \text{op} \in \text{set} (\Psi O^+) \)
\end{itemize}

\textbf{proof} –
\begin{itemize}
  \item \text{fix} \ \text{op}
  \item \text{assume} \( \text{op} \in \text{set} (\text{concat} \ \pi) \)
  \item \text{moreover have} \( \text{op} \in (\bigcup \text{ops} \in \text{set} \ \pi. \ \text{set} \ \text{ops}) \)
    \begin{itemize}
      \item \text{using} \text{calculation}
      \item \text{unfolding} \text{set-concat.}
    \end{itemize}
\end{itemize}
then obtain \( \text{ops where ops} \in \text{set } \pi \text{ and op} \in \text{set ops} \)

using \( \text{UN-iff} \)
by \( \text{blast} \)

ultimately have \( \text{op} \in \text{set } ((\Psi)\sigma_+) \)
using \( \text{assms} \)
by \( \text{blast} \)

\}

thus ?\( \text{thesis} \).

\] qed

\]

**Lemma**: **flattening-lemma-ii**:

fixes \( I :: (\text{'variable, 'domain}) \text{ state} \)
assumes \( \forall \text{ ops} \in \text{set } \psi. \exists \text{ op. ops} = [\text{op}] \land \text{is-valid-operator-sas-plus } \Psi \text{ op} \)
and \( G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi \)
shows \( G \subseteq_m \text{execute-serial-plan-sas-plus } I (\text{concat } \psi) \)

**Proof** –
show ?\( \text{thesis} \)
using \( \text{assms} \)

proof (induction \( \psi \) arbitrary: \( I \))
case (\( \text{Cons ops } \psi \))
obtain \( \text{op where ops-is: ops} = [\text{op}] \text{ and is-valid-op: is-valid-operator-sas-plus } \Psi \text{ op} \)
using \( \text{Cons.prem}(1) \)
by \( \text{auto} \)
then show ?case
proof (cases are-all-operators-applicable-in \( I \) \( \text{ops} \))
case True
let \( ?J = \text{execute-parallel-operator-sas-plus } I [\text{op}] \)
and \( ?J' = \text{execute-operator-sas-plus } I \text{ op} \)
have \( \text{nb}_1: \text{is-operator-applicable-in } I \text{ op} \)
using \( \text{True ops-is} \)
unfolding \( \text{are-all-operators-applicable-in-def} \text{ is-operator-applicable-in-def} \)

list-all-iff
by \( \text{force} \)
have \( \text{nb}_2: \text{are-operator-effects-consistent op op} \)
unfolding \( \text{are-operator-effects-consistent-def list-all-iff Let-def} \)
using \( \text{is-valid-operator-sas-plus-then(6)[OF is-valid-op]} \)
by \( \text{blast} \)
have \( \text{are-all-operator-effects-consistent ops} \)
using \( \text{ops-is} \)
unfolding \( \text{are-all-operator-effects-consistent-def list-all-iff} \)
using \( \text{nb}_2 \)
by \( \text{force} \)
moreover have \( G \subseteq_m \text{execute-parallel-plan-sas-plus } ?J \psi \)
using \( \text{Cons.prem}(2) \text{ True calculation ops-is} \)
by \( \text{fastforce} \)
moreover have \( \text{execute-serial-plan-sas-plus } I (\text{concat } (\text{ops } \# \psi)) \)
\( = \text{execute-serial-plan-sas-plus } ?J' (\text{concat } \psi) \)
using ops-is nb \_1 is-operator-applicable-in-def
by simp
moreover have ?J = ?J'
  using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i[OF
nb \_1 nb \_2]
by simp
ultimately show ?thesis
  using Cons.IH[of ?J] Cons.prems(1)
by force

next
case False
moreover have G \subseteq m I
  using Cons.prems(2) calculation
by fastforce
moreover {
  have \neg is-operator-applicable-in I op
    using False ops-is
  unfolding are-all-operators-applicable-in-def
    is-operator-applicable-in-def list-all-iff
    by force
  moreover have execute-serial-plan-sas-plus I (concat (ops \# \psi))
    = execute-serial-plan-sas-plus I (op \# concat \psi)
    using ops-is
    by force
  ultimately have execute-serial-plan-sas-plus I (concat (ops \# \psi)) = I
    using False
    unfolding is-operator-applicable-in-def
    by fastforce
}
ultimately show ?thesis
  by argo
qed
qed force

lemma flattening-lemma:
assumes is-valid-problem-sas-plus \Psi
  and \ \forall ops \in set \psi. \exists op. ops = [op]
  and is-parallel-solution-for-problem \Psi \psi
shows is-serial-solution-for-problem \Psi (concat \psi)

proof -
let ?\psi' = concat \psi
{
  have \forall ops \in set \psi. \forall op \in set ops. op \in set ((\Psi)_{\O_+})
    using assms(3)
  unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  hence \forall op \in set ?\psi'. op \in set ((\Psi)_{\O_+})
    using flattening-lemma-i
by blast

moreover {
  
  fix ops
  assume ops ∈ set ψ
  moreover obtain op where ops = [op]
    using assms(2) calculation
    by blast
  moreover have op ∈ set ((ψ)₀⁺)
    using assms(3) calculation
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  moreover have is-valid-operator-sas-plus Ψ op
    using assms(1) calculation(3)
    unfolding is-valid-problem-sas-plus-def Let-def list-all-iff
    ListMem-iff
    by simp
  ultimately have ∃ op. ops = [op] ∧ is-valid-operator-sas-plus Ψ op
    by blast
}

moreover have (ψ)₀⁺ ⊆ₘ execute-parallel-plan-sas-plus ((ψ)₁⁺) ψ
  using assms(3)
  unfolding is-parallel-solution-for-problem-def
  by fastforce
  ultimately have (ψ)₀⁺ ⊆ₘ execute-serial-plan-sas-plus ((ψ)₁⁺) ?ψ'
    using flattening-lemma-ii
    by blast
}

ultimately show is-serial-solution-for-problem Ψ ?ψ'
  unfolding is-serial-solution-for-problem-def list-all-iff ListMem-iff
  by fastforce
qed

end

5.4 Auxiliary lemmata on SAS+

context
begin

— Relate the locale definition range-of with its corresponding implementation for
valid operators and given an effect (v, a).

lemma is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op:
  assumes is-valid-operator-sas-plus Ψ op
  and (v, a) ∈ set (precondition-of op) ∨ (v, a) ∈ set (effect-of op)
  shows (R⁺ Ψ v) = set (the (sas-plus-problem.range-of Ψ v))
proof —
  consider (A) (v, a) ∈ set (precondition-of op)
  | (B) (v, a) ∈ set (effect-of op)
using assms(2).

thus ?thesis

proof (cases)
case A
then have \((R_+ \Psi v) \neq \{\}\) and \(a \in R_+ \Psi v\)
  using assms
  unfolding range-of-def
  using is-valid-operator-sas-plus-then(2)
  by fast+
thus ?thesis
  unfolding range-of'-def option.case-eq-if
  by auto

next
case B
then have \((R_+ \Psi v) \neq \{\}\) and \(a \in R_+ \Psi v\)
  using assms
  unfolding range-of-def
  using is-valid-operator-sas-plus-then(4)
  by fast+
thus ?thesis
  unfolding range-of'-def option.case-eq-if
  by auto
qed

lemma set-the-range-of-is-range-of-sas-plus-if:
  fixes \(\Psi :: (\prime v, \prime d) sas-plus-problem\)
  assumes is-valid-problem-sas-plus \(\Psi\)
  \(v \in \text{set } ((\Psi)_{\nu_+})\)
  shows \(\text{set } (\text{the } (sas-plus-problem.\text{range-of }\Psi v)) = R_+ \Psi v\)
proof
  have \(v \in \text{set } ((\Psi)_{\nu_+})\)
    using assms(2)
  unfolding variables-of-def.
  moreover have \((R_+ \Psi v) \neq \{\}\) 
    using assms(1) calculation is-valid-problem-sas-plus-then(1)
    by blast
  moreover have \(sas-plus-problem.\text{range-of }\Psi v \neq \text{None}\)
    and \(sas-plus-problem.\text{range-of }\Psi v \neq \text{Some } []\)
    using calculation(2) range-of-not-empty
  unfolding range-of-def
    by fast+
  ultimately show ?thesis
    unfolding option.case-eq-if range-of'-def
    by force
qed

lemma sublocale-sas-plus-finite-domain-representation-ii:
  fixes \(\Psi :: (\prime v,\prime d) sas-plus-problem\)
assumes is-valid-problem-sas-plus $\Psi$
shows $\forall v \in \text{set } ((\Psi)_v^+) \cdot (R_+ \Psi v) \neq \{\}$
and $\forall \text{op} \in \text{set } ((\Psi)_o^+), \text{is-valid-operator-sas-plus } \Psi \text{ op}$
and $\text{dom } ((\Psi)_t^+) = \text{set } ((\Psi)_v^+)$
and $\forall v \in \text{dom } ((\Psi)_t^+), \text{the } ((\Psi)_t^+) v \in R_+ \Psi v$
and $\text{dom } ((\Psi)_g^+) \subseteq \text{set } ((\Psi)_v^+)$
and $\forall v \in \text{dom } ((\Psi)_g^+), \text{the } ((\Psi)_g^+) v \in R_+ \Psi v$
using is-valid-problem-sas-plus-then[OF assms]
by auto
end
end

theory SAS-Plus-STRIPS
imports STRIPS-Semantics SAS-Plus-Semantics
Map-Supplement
begin

6 SAS+/STRIPS Equivalence

The following part is concerned with showing the equivalent expressiveness of SAS+ and STRIPS as discussed in ??.

6.1 Translation of SAS+ Problems to STRIPS Problems

definition possible-assignments-for
:: ('variable, 'domain) sas-plus-problem \Rightarrow 'variable \Rightarrow ('variable \times 'domain) list

where possible-assignments-for $\Psi v \equiv [(v, a). a \leftarrow \text{the } \text{range-of } \Psi v]$

definition all-possible-assignments-for
:: ('variable, 'domain) sas-plus-problem \Rightarrow ('variable \times 'domain) list

where all-possible-assignments-for $\Psi$
\equiv concat [possible-assignments-for $\Psi v. v \leftarrow \text{variables-of } \Psi$

definition state-to-strips-state
:: ('variable, 'domain) sas-plus-problem
\Rightarrow ('variable, 'domain) state
\Rightarrow ('variable, 'domain) assignment strips-state
($\varphi_S$ - - 99)

where state-to-strips-state $\Psi s$
\equiv let defined = filter ($\lambda v. s v \neq \text{None}$) (variables-of $\Psi$) in
map-of (map ($\lambda (v, a). ((v, a), \text{the } (s v) = a)$
\Rightarrow concat [possible-assignments-for $\Psi v. v \leftarrow \text{defined}]$

definition sasp-op-to-strips
:: ('variable, 'domain) sas-plus-problem
\[ \Rightarrow (\text{'variable}, \text{'domain}) \text{sas-plus-operator} \]
\[ \Rightarrow (\text{'variable}, \text{'domain}) \text{assignment strips-operator} \]
\[ (\varphi_O - 99) \]
\[ \text{where sas-op-to-strips } \Psi \text{ op } \equiv \text{ let} \]
\[ \text{pre } = \text{precondition-of op} \]
\[ \text{add } = \text{effect-of op} \]
\[ \text{delete } = \{ (v, a'), (v, a) \leftarrow \text{effect-of op}, a' \leftarrow \text{filter } ((\neq) \text{ a}) \text{ (the range-of } \Psi \} \]
\[ \text{in STRIPS-Representation.operator-for pre add delete} \]

\text{definition sas-plus-problem-to-strips-problem} :: (\text{'variable}, \text{'domain}) \text{sas-plus-problem } \Rightarrow (\text{'variable}, \text{'domain}) \text{assignment strips-problem} \]
\[ (\varphi - 99) \]
\[ \text{where sas-plus-problem-to-strips-problem } \Psi \equiv \text{ let} \]
\[ \text{vs } = \{ \text{as } v \leftarrow \text{variables-of } \Psi, \text{as } \leftarrow \text{possible-assignments-for } \Psi \} \]
\[ \text{ops } = \text{map } (\text{sas-op-to-strips } \Psi) \text{ (operators-of } \Psi) \]
\[ I = \text{state-to-strips-state } \Psi \text{ (initial-of } \Psi) \]
\[ G = \text{state-to-strips-state } \Psi \text{ (goal-of } \Psi) \]
\[ \text{in STRIPS-Representation.problem-for vs ops I G} \]

\text{definition sas-plus-parallel-plan-to-strips-parallel-plan} :: (\text{'variable}, \text{'domain}) \text{sas-plus-problem} \\ \Rightarrow (\text{'variable}, \text{'domain}) \text{sas-plus-parallel-plan} \\ \Rightarrow (\text{'variable } \times \text{'domain}) \text{strips-parallel-plan} \]
\[ (\varphi - 99) \]
\[ \text{where sas-plus-parallel-plan-to-strips-parallel-plan } \Psi \psi \equiv \{ \text{sas-op-to-strips } \Psi \text{ op, op } \leftarrow \text{ops}, \text{ops } \leftarrow \psi \} \]

\text{definition strips-state-to-state} :: (\text{'variable}, \text{'domain}) \text{sas-plus-problem} \\ \Rightarrow (\text{'variable}, \text{'domain}) \text{assignment strips-state} \\ \Rightarrow (\text{'variable}, \text{'domain}) \text{state} \]
\[ (\varphi_s^{-1} - 99) \]
\[ \text{where strips-state-to-state } \Psi s \equiv \text{map-of } (\text{filter } \lambda(v, a). s (v, a) = \text{Some True}) \text{ (all-possible-assignments-for } \Psi) \]

\text{definition strips-op-to-sasp} :: (\text{'variable}, \text{'domain}) \text{sas-plus-problem} \\ \Rightarrow (\text{'variable } \times \text{'domain}) \text{strips-operator} \\ \Rightarrow (\text{'variable}, \text{'domain}) \text{sas-plus-operator} \]
\[ (\varphi_O^{-1} - 99) \]
\[ \text{where strips-op-to-sasp } \Psi \text{ op } \equiv \text{ let} \]
\[ \text{precondition } = \text{strips-operator.precondition-of op} \]
\[ \text{effect } = \text{strips-operator.add-effects-of op} \]

86
\[
\text{definition strips-parallel-plan-to-sas-plus-parallel-plan}:: ('\text{variable, \text{'domain}}) \text{sas-plus-problem} \\
\quad \Rightarrow ('\text{variable} \times \text{'domain}) \text{strips-parallel-plan} \\
\quad \Rightarrow ('\text{variable, \text{'domain}}) \text{sas-plus-parallel-plan} \\
\quad (\varphi \sigma^{-1} - 99)
\]

\text{where strips-parallel-plan-to-sas-plus-parallel-plan} \Pi \pi \\
\equiv [[\text{strips-op-to-sasp} \Pi \text{op. op} \leftarrow \text{ops}]. \text{ops} \leftarrow \pi]

To set up the equivalence proof context, we declare a common locale for both the STRIPS and SAS+ formalisms and make it a sublocale of both locale as well as . The declaration itself is omitted for brevity since it basically just joins locales and while renaming the locale parameter to avoid name clashes. The sublocale proofs are shown below. 6

\text{definition range-of-strips} \Pi x \equiv \{ \text{True, False} \}

context

begin

— Set-up simp rules.

lemma[simp]:

\((\varphi \Psi) = (let \vs = [as. v \leftarrow \text{variables-of } \Psi, \text{as} \leftarrow \text{possible-assignments-for } \Psi) \text{ v}] \; ; \text{ops} = \text{map} (\text{sasp-op-to-strips } \Psi) (\text{operators-of } \Psi) \; ; \text{I} = \text{state-to-strips-state } \Psi (\text{initial-of } \Psi) \; ; \text{G} = \text{state-to-strips-state } \Psi (\text{goal-of } \Psi) \; in \text{STRIPS-Representation.problem-for } \vs \text{ ops I G})\)

and \((\varphi_S \Psi s)\)

\(= (let \text{defined} = \text{filter} (\lambda v. s v \neq None) \text{ (variables-of } \Psi) \text{ in} \) \text{map-of} (\text{map} (\lambda (v, a). ((v, a), \text{the } (s v) = a)) \text{ (concat [possible-assignments-for } \Psi v. v \leftarrow \text{defined}])))

and \((\varphi_O \Psi \text{ op})\)

\(= (let \text{pre} = \text{precondition-of } \text{op} \; ; \text{add} = \text{effect-of } \text{op} \; ; \text{delete} = [(v, a'). (v, a) \leftarrow \text{effect-of } \text{op}, a' \leftarrow \text{filter} ((\neq) a) (\text{the } \text{(range-of } \Psi v))] \) \text{ in STRIPS-Representation.operator-for } \text{pre add delete})

and \((\varphi_S^{-1} \Psi \psi) = [[\varphi_O \Psi \text{ op. op} \leftarrow \text{ops}]. \text{ops} \leftarrow \psi])\)

and \((\varphi_S^{-1} \Psi s') = \text{map-of} (\text{filter} (\lambda (v, a). \text{ s'} (v, a) = \text{Some True})\)

\(\text{We append a suffix identifying the respective formalism to the the parameter names passed to the parameter names in the locale. This is necessary to avoid ambiguous names in the sublocale declarations. For example, without addition of suffixes the type for initial-of is ambiguous and will therefore not be bound to either strips-problem.initial-of or sas-plus-problem.initial-of. Isabelle in fact considers it to be a a free variable in this case. We also qualify the parent locales in the sublocale declarations by adding strips: and sas_plus: before the respective parent locale identifiers.}\)

87
\[(\text{all-possible-assignments-for } \Psi))\]

and \((\varphi_\Psi^{-1} \Psi \text{ op'}) = (\text{let} \quad \text{precondition} = \text{strips-operator.precondition-of \text{ op'}}
  \quad \text{effect} = \text{strips-operator.add-effects-of \text{ op'}}
  \quad \text{in } (\text{precondition-of } = \text{precondition}, \text{effect-of } = \text{effect}))\]

and \((\varphi_\Psi^{-1} \Psi \pi) = \left[\left[ \varphi_\Psi^{-1} \Psi \text{ op. op } \leftarrow \text{ops. ops } \leftarrow \pi \right]\right]\)

unfolding

\text{SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def}
\text{SAS-Plus-STRIPS.state-to-strips-state-def}
\text{SAS-Plus-STRIPS.sasp-op-to-strips-def}
\text{SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def}
\text{SAS-Plus-STRIPS.strips-state-to-state-def}
\text{SAS-Plus-STRIPS.strips-op-to-sasp-def}
\text{SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def}
\text{SAS-Plus-STRIPS.strips-state-to-state-def}
\text{SAS-Plus-STRIPS.strips-op-to-sasp-def}
\text{SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def}
\text{SAS-Plus-STRIPS.strips-state-to-state-def}
\text{SAS-Plus-STRIPS.strips-op-to-sasp-def}
\text{SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def}
\text{SAS-Plus-STRIPS.strips-state-to-state-def}

by blast+

lemmas \([\text{simp}] = \text{range-of}'-\text{def}\)

\textbf{lemma} \text{is-valid-problem-sas-plus-dom-sas-plus-problem-range-of}:
\textbf{assumes} \text{is-valid-problem-sas-plus } \Psi
\textbf{shows} \(\forall v \in \text{set } ((\Psi)_v^+). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)\)
\textbf{using} \text{assms}(1) \text{ is-valid-problem-sas-plus-then}(1)
\textbf{unfolding} \text{is-valid-problem-sas-plus-def}

by \text{(meson domIff list.pred-set)}

\textbf{lemma} \text{possible-assignments-for-set-is}:
\textbf{assumes} \(v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)\)
\textbf{shows} \(\text{set } (\text{possible-assignments-for } \Psi v) = \{ (v, a) \mid a. a \in R_+ \Psi v \}\)
\textbf{proof} –
\textbf{have} \text{sas-plus-problem.range-of } \Psi v \neq \text{None}
\textbf{using} \text{assms}(1)
by auto
\textbf{thus} \(?thesis\)
\textbf{unfolding} \text{possible-assignments-for-def}
by \text{fastforce}
\textbf{qed}

\textbf{lemma} \text{all-possible-assignments-for-set-is}:
\textbf{assumes} \(\forall v \in \text{set } ((\Psi)_v^+). \text{range-of } \Psi v \neq \text{None}\)
\textbf{shows} \(\text{set } (\text{all-possible-assignments-for } \Psi) = (\bigcup v \in \text{set } ((\Psi)_v^+). \{ (v, a) \mid a. a \in R_+ \Psi v \})\)
proof

let ?vs = variables-of Ψ
have set (all-possible-assignments-for Ψ) = 
  (⋃ (set ' (λv. map (λ(v, a). (v, a)) (possible-assignments-for Ψ v)) ' set ?vs))
unfolding all-possible-assignments-for-def set-concat
using set-map
by auto
also have ... = (⋃ ((λv. set (possible-assignments-for Ψ v)) ' set ?vs))
  using image-comp set-map
by simp
also have ... = (⋃ ((λv. { (v, a) | a. a ∈ R+ Ψ v }) ' set ?vs))
  using possible-assignments-for-set-is assms
by fastforce
finally show ?thesis
by force
qed

lemma state-to-strips-state-dom-is-i[simp]:
assumes ∀ v ∈ set (((Ψ) v+). v ∈ dom (sas-plus-problem.range-of Ψ)
shows set (concat
  [possible-assignments-for Ψ v. v ← filter (λv. s v ≠ None) (variables-of Ψ)])
= (⋃ {v ∈ set (((Ψ) v+). v ∈ set (sas-plus-problem.range-of Ψ) ∧ s v ≠ None).})
by blast
proof

let ?vs = variables-of Ψ
let ?defined = filter (λv. s v ≠ None) ?vs
let ?l = concat [possible-assignments-for Ψ v. v ← ?defined]
have nb: set ?defined = {v. v ∈ set (((Ψ) v+) ∧ s v ≠ None).}
  unfolding set-filter
by force
have set ?l = (⋃ (set ' set (map (possible-assignments-for Ψ) ?defined ')))
  unfolding set-concat image-Union
by blast
also have ... = (⋃ (set ' (possible-assignments-for Ψ) ' set ?defined))
unfolding set-map
by blast
also have ... = (⋃ v ∈ set ?defined. set (possible-assignments-for Ψ v))
by blast
also have ... = (⋃ v ∈ set (possible-assignments-for Ψ v))
  by blast
also have ... = (⋃ v ∈ set (possible-assignments-for Ψ v))
  set (possible-assignments-for Ψ v))
using nb
by argo
finally show ?thesis
  using possible-assignments-for-set-is
  is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)
by fastforce
qed

89
lemma state-to-strips-state-dom-is:
— NOTE A transformed state is defined on all possible assignments for all variables defined in the original state.

assumes is-valid-problem-sas-plus $\Psi$

shows $\text{dom} (\varphi_S \Psi s)$

\[ = \left( \bigcup v \in \{ v \mid v, v \in \text{set } ((\Psi)_{\varphi_S}) \land s v \neq \text{None} \} \setminus \{ (v, a) \mid a, a \in \mathcal{R}_+ \Psi v \} \right) \]

proof –

let $?vs = \text{variables-of } \Psi$

let $?l = \text{concat } \left[ \text{possible-assignments-for } \Psi v, v \leftarrow \text{filter } (\lambda v. s v \neq \text{None}) ?vs \right]$

have $\text{nb: } \forall v \in \text{set } ((\Psi)_{\varphi_S}). v \in \text{dom } (\text{sas-plus\text{-}problem}\text{-}range\text{-}of } \Psi)$

using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms (1)

by fastforce

have $\text{dom} (\varphi_S \Psi s, \Psi s) = \text{fst } \left( \text{set } (\lambda (v, a). ((v, a), \text{the } (s v) = a)) \right) ?l$

unfolding state-to-strips-state-def

SAS-Plus-STRIPS.state-to-strips-state-def

using dom-map-of-conv-image-fst[of map (\lambda (v, a). ((v, a), \text{the } (s v) = a)) ?l]

by presburger

also have $\ldots = \text{fst } (\lambda (v, a). ((v, a), \text{the } (s v) = a)) \text{ set } ?l$

unfolding set-map

by blast

also have $\ldots = (\lambda (v, a), \text{fst } ((v, a), \text{the } (s v) = a)) \text{ set } ?l$

unfolding image-comp[of \lambda (v, a). ((v, a), \text{the } (s v) = a)] \text{ comp\text{-}apply[of}

\text{fst } \lambda (v, a). (v, a), \text{the } (s v) = a]) \text{ prod\text{-}case\text{-}distrib

by blast

finally show $\text{thesis}$

unfolding state-to-strips-state-dom-is-i[OF \text{nb}]

by force

qed

corollary state-to-strips-state-dom-element-iff:

assumes is-valid-problem-sas-plus $\Psi$

shows $(v, a) \in \text{dom } (\varphi_S \Psi s) \iff v \in \text{set } ((\Psi)_{\varphi_S})$

$\land s v \neq \text{None}$

$\land a \in \mathcal{R}_+ \Psi v$

proof –

let $?vs = \text{variables-of } \Psi$

and $?s' = \varphi_S \Psi s$

show $\text{thesis}$

proof (rule iff1)

assume $(v, a) \in \text{dom } (\varphi_S \Psi s)$

then have $v \in \{ v \mid v, v \in \text{set } ((\Psi)_{\varphi_S}) \land s v \neq \text{None} \}$

and $a \in \mathcal{R}_+ \Psi v$

unfolding state-to-strips-state-dom-is[OF assms(1)]

by force

moreover have $v \in \text{set } ?vs$ and $s v \neq \text{None}$

using calculation(1)

by fastforce

ultimately show
\[ v \in \text{set } ((\Psi)_{V+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v \]

by force

next

assume \[ v \in \text{set } ((\Psi)_{V+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v \]
then have \[ v \in \text{set } ((\Psi)_{V+}) \]
and \[ s v \neq \text{None} \]
and \[ a\text{-in-range-of-}v: a \in \mathcal{R}_+ \Psi v \]
by simp+
then have \[ v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V+}) \land s v \neq \text{None} \} \]
by force
thus \[(v, a) \in \text{dom } (\varphi_S \Psi s)\]
unfolding state-to-strips-state-dom-is[\(\text{OF assms(1)}\)]
using a-in-range-of-v
by blast
qed

qed

lemma state-to-strips-state-range-is:
assumes \[ \text{is-valid-problem-sas-plus } \Psi \]
and \[(v, a) \in \text{dom } (\varphi_S \Psi s)\]
shows \[ (\varphi_S \Psi s) (v, a) = \text{Some } (\text{the } (s v) = a)\]
proof –
let \(\text{?vs} = \text{variables-of } \Psi\)
let \(\text{?s'} = \varphi_S \Psi s\)
and \(\text{?defined} = \text{filter } (\lambda v. s v \neq \text{None}) \text{?vs}\)
let \(\text{?l} = \text{concat } \{\text{possible-assignments-for } \Psi v. v \leftarrow \text{?defined}\}\)
have \(\text{v-in-set-vs}\): \(v \in \text{set } \text{?vs}\)
and \(\text{s-of-v-is-not-None}: s v \neq \text{None}\)
and \(\text{a-in-range-of-}v: a \in \mathcal{R}_+ \Psi v\)
using assms(1)
unfolding state-to-strips-state-dom-is[\(\text{OF assms(1)}\)]
by fastforce+
moreover \{ 
have \(\forall v \in \text{set } ((\Psi)_{V+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)\)
using assms(1) is-valid-problem-sas-plus-then(1)
unfolding is-valid-problem-sas-plus-def
by fastforce
moreover have \((v, a) \in \text{set } \text{?l}\)
unfolding state-to-strips-state-dom-is-i[\(\text{OF calculation(1)}\)]
using s-of-v-is-not-None a-in-range-of-v v-in-set-vs
by fastforce
moreover have \(\text{set } \text{?l} \neq \{\}\)
using calculation
by fastforce
— TODO slow.
ultimately have \((\varphi_S \Psi s) (v, a) = \text{Some } (\text{the } (s v) = a)\)
using map-of-from-function-graph-is-some-if[\(\text{of \(\text{?l} \(v, a\) } \lambda (v, a). \text{the } (s v) = a\)\]
unfolding SAS-Plus-STRIPS.state-to-strips-state-def
Show that a STRIPS state corresponding to a SAS+ state via transformation is consistent w.r.t. to the variable subset with same left component (i.e. the original SAS+ variable). This is the consistency notion corresponding to SAS+ consistency: i.e. if no two assignments with different values for the same variable exist in the SAS+ state, then assigning the corresponding assignment both to True is impossible. Vice versa, if both are assigned to True then the assignment variables must be the same SAS+ variable/SAS+ value pair.

**Lemma state-to-strips-state-effect-consistent:**

assumes is-valid-problem-sas-plus $\Psi$
and $(v, a) \in \text{dom}(\varphi_S \Psi s)$
and $(v, a') \in \text{dom}(\varphi_S \Psi s)$
and $(\varphi_S \Psi s)(v, a) = \text{Some True}$
and $(\varphi_S \Psi s)(v, a') = \text{Some True}$
shows $(v, a) = (v, a')$

proof –

have the $(s v) = a$ and the $(s v) = a'$
using state-to-strips-state-range-is[OF assms(1)] assms(2, 3, 4, 5)
by fastforce
thus ?thesis
by argo

qed

**Lemma sasp-op-to-strips-set-delete-effects-is:**

assumes is-valid-operator-sas-plus $\Psi$ op
shows set (strips-operator.delete-effects-of ($\varphi_O \Psi$ op))
  = (\bigcup (v, a) \in \text{set (effect-of op)}. \{ (v, a') | a', a' \in (\mathcal{R}_+ \Psi v) \land a' \neq a \})

proof –

let $?D = \text{range-of } \Psi$
and $?effect = \text{effect-of op}$
let $?delete = [(v, a'). (v, a) \leftarrow $?effect, a' \leftarrow \text{filter } ((\neq) a) \text{ (the } (?D v))]$

{ fix $v$ a
assume $(v, a) \in \text{set } $?effect$
then have $(\mathcal{R}_+ \Psi v) = \text{set (the } (?D v))$
using assms
using is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op
by fastforce
hence set (filter $((\neq) a) \text{ (the } (?D v))) = \{ a' \in \mathcal{R}_+ \Psi v. a' \neq a \}$
unfolding set-filter
by blast
} note nb = this

92
{— TODO slow.

have set ?delete = ∪(set ' (λ(v, a). map (Pair v) (filter ((≠) a) (the (?D v)))))
  (set ?effect))
using set-concat
by simp
also have ... = ∪((λ(v, a). Pair v ' set (filter ((≠) a) (the (?D v)))))
  (set ?effect))
unfolding image-comp[of set] set-map
by auto
— TODO slow.
also have ... = (∪(v, a) ∈ set ?effect. Pair v ' { a' ∈ R v. a' ≠ a })
  using nb
by fast
finally have set ?delete = (∪(v, a) ∈ set ?effect.
  { (v, a') | a', a' ∈ (R+ Ψ v) ∧ a' ≠ a })
by blast
}
thus ?thesis
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def Let-def
by force
qed

lemma sas-plus-problem-to-strips-problem-variable-set-is:
— The variable set of Π is the set of all possible assignments that are possible
using the variables of V and the corresponding domains.
assumes is-valid-problem-sas-plus Ψ
shows set ((ϕ Ψ)V) = (∪ v ∈ set ((Ψ)V+). { (v, a) | a. a ∈ R+ Ψ v })
proof —
let ?Π = ϕ Ψ
and ?vs = variables-of Ψ
{ have set (strips-problem.variables-of ?Π)
  = set [as, v ← ?vs, as ← possible-assignments-for Ψ v]
  unfolding sas-plus-problem-to-strips-problem-def
  SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
  by force
also have ... = (∪ (set ' (λv. possible-assignments-for Ψ v) ' set ?vs))
  using set-concat
  by auto
also have ... = (∪ ((set o possible-assignments-for Ψ) ' set ?vs))
  using image-comp[of set λv. possible-assignments-for Ψ v set ?vs]
  by argo
finally have set (strips-problem.variables-of ?Π)
  = (∪ v ∈ set ?vs, set (possible-assignments-for Ψ v))
  unfolding o-apply
  by blast

93
moreover have \( \forall v \in \text{set } \Psi \) using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms by force
ultimately show \( \text{thesis} \) using possible-assignments-for-set-is by force qed

**corollary** sas-plus-problem-to-strips-problem-variable-set-element-iff:
assumes is-valid-problem-sas-plus \( \Psi \) shows \((v, a) \in \text{set } ((\varphi \Psi)_v) \iff v \in \text{set } ((\Psi)_{\Psi^+}) \land a \in R^+ \Psi v\)
unfolding sas-plus-problem-to-strips-problem-variable-set-is[OF assms] by fast

**lemma** sasp-op-to-strips-effect-consistent:
assumes \( \varphi = \varphi_{\text{OP}} \) \( \Psi \) op \( \Psi \) op \( \Psi \) and \( \text{is-valid-operator-sas-plus } \Psi \) op \( \Psi \) shows \((v, a) \in \text{set } (\text{add-effects-of } op) \rightarrow (v, a) \notin \text{set } (\text{delete-effects-of } op)\) and \( (v, a) \in \text{set } (\text{add-effects-of } op) \rightarrow (v, a) \notin \text{set } (\text{add-effects-of } op)\)
proof
- have \( \text{nb: } (\forall (v, a) \in \text{set } (\text{effect-of } op^\prime). \forall (v', a') \in \text{set } (\text{effect-of } op^\prime). v \neq v' \lor a = a') \)
  using assms
  unfolding is-valid-operator-sas-plus-def
  Let-def by argo
  \{ fix \( v \) \( a \) \\
  assume \( v-a \).in-add-effects-of-op: \( (v, a) \in \text{set } (\text{add-effects-of } op) \)
  have \( (v, a) \notin \text{set } (\text{delete-effects-of } op) \)
  proof (rule ccontr)
    assume \( \neg(v, a) \notin \text{set } (\text{delete-effects-of } op) \)
    moreover have \((v, a) \in (\bigcup\{(v, a') \in \text{set } (\text{effect-of } op^\prime). \{ (v, a'') \\
    \{ a'', a' \in (R^+ \Psi v) \land a'' \neq a' \})\)\)
    using calculation sasp-op-to-strips-set-delete-effects-is assms
    by blast
    moreover obtain \( a' \) where \((v, a') \in \text{set } (\text{effect-of } op^\prime) \) and \( a \neq a' \)
    using calculation
    by blast
    moreover have \((v, a') \in \text{set } (\text{add-effects-of } op) \)
    using assms(1) calculation(3)
    unfolding sasp-op-to-strips-def
    SAS-Plus-STRIPS,sasp-op-to-strips-def
    Let-def

  \}
by fastforce
moreover have \((v, a) \in \text{set}(\text{effect-of } \text{op})\) and \((v, a') \in \text{set}(\text{effect-of } \text{op})\)
using assms(1) v-a-in-add-effects-of-op calculation(5)
unfolding sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
Let-def
by force+
ultimately show False
using nb
by fast
qed
}
moreover {
  fix \(v\ a\)
assume v-a-in-delete-effects-of-op: \((v, a) \in \text{set}(\text{delete-effects-of } \text{op})\)
have \((v, a) \notin \text{set}(\text{add-effects-of } \text{op})\)
proof (rule ccontr)
  assume \(\neg(v, a) \notin \text{set}(\text{add-effects-of } \text{op})\)
moreover have \((v, a) \in \text{set}(\text{add-effects-of } \text{op})\)
  using calculation
  by blast
moreover have \((v, a) \in\)
(\(\bigcup(v, a') \in \text{set}(\text{effect-of } \text{op}'), \{ (v, a'') | a'' \cdot a'' \in (\mathcal{R}_+ \Psi v) \land a'' \neq a'\}\))
using sasp-op-to-strips-set-delete-effects-is
nb assms(1, 3) v-a-in-delete-effects-of-op
by force
moreover obtain \(a'\) where \((v, a') \in \text{set}(\text{effect-of } \text{op})\) and \(a \neq a'\)
using calculation
by blast
moreover have \((v, a') \in \text{set}(\text{add-effects-of } \text{op})\)
using assms(1) calculation(4)
unfolding sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
Let-def
by fastforce
moreover have \((v, a) \in \text{set}(\text{effect-of } \text{op})\) and \((v, a') \in \text{set}(\text{effect-of } \text{op})\)
using assms(1) calculation(2, 6)
unfolding sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def Let-def
by force+
ultimately show False
using nb
by fast
qed
}
ultimately show \((v, a) \in \text{set}(\text{add-effects-of } \text{op})\)
  \(\rightarrow (v, a) \notin \text{set}(\text{delete-effects-of } \text{op})\)
and \((v, a) \in \text{set}(\text{delete-effects-of } \text{op})\)

95
\[
(v, a) \notin \text{set (add-effects-of op)}
\]
by blast+

qed

lemma is-valid-problem-sas-plus-then-strips-transformation-too-iii:
assumes is-valid-problem-sas-plus \( \Psi \)
shows list-all \((\text{is-valid-operator-strips } (\varphi \ \Psi))\)
(strips-problem.operators-of \((\varphi \ \Psi))\)
proof –

let \( ?\Pi = \varphi \ \Psi \)
let \( ?\mathit{vs} = \text{strips-problem.variables-of } ?\Pi \)
\{
fix op
assume \( \text{op } \in \text{set (strips-problem.operators-of } ?\Pi) \)
— TODO slow.
then obtain \( \text{op}' \)
where \( \text{op-is: } \text{op } = \varphi_{\mathcal{O}} \ \Psi \ \text{op}' \)
and \( \text{op}'-\text{in-operators: } \text{op}' \in \text{set } (\Psi_{\mathcal{O}+}) \)
unfolding \( \text{SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def} \)
\( \text{sas-plus-problem-to-strips-problem-def} \)
\( \text{sasp-op-to-strips-def} \)
by auto
then have \( \text{is-valid-op}' \); \( \text{is-valid-operator-sas-plus } \Psi \text{ op}' \)
using sublocale-sas-plus-finite-domain-representation-ii(2)[OF assms]
by blast

moreover \{
fix \( v \ a \)
assume \( (v, a) \in \text{set (strips-operator.precondition-of op)} \)
— TODO slow.
then have \( (v, a) \in \text{set (sas-plus-operator.precondition-of op')} \)
using op-is
unfolding \( \text{SAS-Plus-STRIPS.sasp-op-to-strips-def} \)
\( \text{sasp-op-to-strips-def} \)
by force
moreover have \( v \in \text{set } ((\Psi)_{\mathcal{V}+}) \)
using is-valid-op' calculation
using is-valid-operator-sas-plus-then(1)
by fastforce
moreover have \( a \in \mathcal{R}_+ \ \Psi \ v \)
using is-valid-op' calculation(1)
using is-valid-operator-sas-plus-then(2)
by fast
ultimately have \( (v, a) \in \text{set } ?\mathit{vs} \)
using sas-plus-problem-to-strips-problem-variable-set-element-iff'[OF assms(1)]
by force
\}
moreover \{
fix \( v \ a \)
assume \( (v, a) \in \text{set (strips-operator.add-effects-of op)} \)
then have \((v, a) \in \text{set}\ (\text{effect-of }\text{op}')\)
  using \text{op-is}
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
  by force
then have \(v \in ((\Psi)_{\mathbb{R}^+})\) and \(a \in \mathbb{R}^+ \Psi v\)
  using is-valid-operator-sas-plus-then is-valid-op'
  by fastforce+
hence \((v, a) \in \text{set}\ ?\text{vs}\)
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[\text{OF assms}(1)]
  by force
}
moreover {
  fix \(v\ a'\)
  assume \(v\cdot a'\text{-in}\text{-delete-effects}: (v, a') \in \text{set} (\text{strips-operator.delete-effects-of op})\)
  moreover have \(\text{set}\ (\text{strips-operator.delete-effects-of op})\)
    \(= \bigcup \{ (v, a) \in \text{set} (\text{effect-of op}')\cdot\{ (v, a') \mid a', a' \in (\mathbb{R}^+ \Psi v) \land a' \neq a \}\}\)
  using sasp-op-to-strips-set-delete-effects-is[\text{OF is-valid-op}]
  \text{op-is}
  by simp
  — TODO slow.
ultimately obtain \(a\)
  where \((v, a) \in \text{set} (\text{effect-of op}')\)
    and \(a'\text{-in}: a' \in \{ a' \in \mathbb{R}^+ \Psi v. a' \neq a \}\)
  by blast
moreover have \(\text{is-valid-operator-sas-plus }\Psi \text{ op'}\)
  using \text{op'}-in-operators assms(1)
  \text{is-valid-problem-sas-plus-then}(2)
  by blast
moreover have \(v \in ((\Psi)_{\mathbb{R}^+})\)
  using is-valid-operator-sas-plus-then calculation(1, 3)
  by fast
moreover have \(a' \in \mathbb{R}^+ \Psi v\)
  using \(a'\text{-in}\)
  by blast
ultimately have \((v, a') \in \text{set}\ ?\text{vs}\)
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[\text{OF assms}(1)]
  by force
}
ultimately have \(\text{set}\ (\text{strips-operator.precondition-of op}) \subseteq \text{set}\ ?\text{vs}\)
  \(\land\ \text{set}\ (\text{strips-operator.add-effects-of op}) \subseteq \text{set}\ ?\text{vs}\)
  \(\land\ \text{set}\ (\text{strips-operator.delete-effects-of op}) \subseteq \text{set}\ ?\text{vs}\)
  \(\land\ (\forall v \in \text{set}\ (\text{add-effects-of op}), v \notin \text{set}\ (\text{delete-effects-of op}))\)
  \(\land\ (\forall v \in \text{set}\ (\text{delete-effects-of op}), v \notin \text{set}\ (\text{add-effects-of op}))\)
  using sasp-op-to-strips-effect-consistent[\text{OF}
  \text{op-is}\ \text{op'}-in-operators\ \text{is-valid-op}']
  by fast+
\begin{proof}
\let\proofproof\proof
\end{proof}
using list-mem-x-vs'
unfolding ListMem-iff
by blast
then have \( v \in \text{set } ?v \text{ and } a \in R_+ \Psi v \)
using sas-plus-problem-to-strips-problem-variable-set-element-iff[of assms(1)]
by force+
moreover have \(?I v \neq \text{None}\)
using is-valid-problem-sas-plus-then(3) assms(1) calculation(1)
by auto
ultimately have \((v, a) \in \text{dom } ?I'\)
using state-to-strips-state-dom-element-iff[of assms(1), of v a ?I]
unfolding SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
sas-plus-problem-to-strips-problem-def
SAS-Plus-STRIPS.state-to-strips-state-def
state-to-strips-state-def
by force
thus \(?I' x \neq \text{None}\)
using x-is
by fastforce
qed

Thus \(?thesis\)
by simp
qed

private lemma is-valid-problem-sas-plus-then-strips-transformation-too-v:
assumes is-valid-problem-sas-plus \(\Psi\)
shows \(\forall x. ((\phi \Psi) G) x \neq \text{None} \rightarrow \text{ListMem } x \text{(strips-problem.variables-of } (\phi \Psi))\)
proof -
let ?vs = variables-of \(\Psi\)
and ?D = range-of \(\Psi\)
and ?G = goal-of \(\Psi\)
let ?P = \(\phi \Psi\)
let ?vs' = strips-problem.variables-of ?P
and ?G' = strips-problem.goal-of ?P
have nb: ?G' = \(\phi_S \Psi') ?G\)
by simp
{
fix x
assume ?G' x \neq \text{None}
moreover obtain \(v a \text{ where } x = (v, a)\)
by fastforce
moreover have \((v, a) \in \text{dom } ?G'\)
using domIff calculation(1, 2)
by blast
moreover have \(v \in \text{set } ?v \text{ and } a \in R_+ \Psi v\)
using state-to-strips-state-dom-is[of assms(1), of ?G] nb calculation(3)
by auto+
ultimately have $x \in \text{set } \bar{v}s'$

using \text{sas-plus-problem-to-strips-problem-variable-set-element-iff}[OF assms(1)]

by \text{auto}
}

thus \text{thesis}

unfolding \text{ListMem-iff}

by \text{simp}

\text{qed}

We now show that given $\Psi$ is a valid SASPlus problem, then $\Pi \equiv \phi \Psi$ is a valid STRIPS problem as well. The proof unfolds the definition of \text{is-valid-problem-strips} and then shows each of the conjuncts for $\Pi$. These are:

- $\Pi$ has at least one variable;
- $\Pi$ has at least one operator;
- all operators are valid STRIPS operators;
- $\Pi_I$ is defined for all variables in $\Pi_V$; and finally,
- if $(\Pi_G) x$ is defined, then $x$ is in $\Pi_V$.

\text{theorem}

\text{is-valid-problem-sas-plus-then-strips-transformation-too};

\text{assumes} \hspace{1em} \text{is-valid-problem-sas-plus } \Psi

\text{shows} \hspace{1em} \text{is-valid-problem-strips} (\phi \Psi)

\text{proof} \hspace{1em} 

\text{let} \hspace{1em} ?\Pi = \phi \Psi

\text{have} \hspace{1em} \text{list-all} (\text{is-valid-operator-strips} (\phi \Psi))

(\text{strips-problem-operators-of} (\phi \Psi))

\text{using} \hspace{1em} \text{is-valid-problem-sas-plus-then-strips-transformation-too-iii}[OF assms].

\text{moreover have} \hspace{1em} \forall \hspace{1em} x. ((\phi \Psi)_I) x \neq \text{None} =

\text{ListMem} x (\text{strips-problem-variables-of} (\phi \Psi))

\text{using} \hspace{1em} \text{is-valid-problem-sas-plus-then-strips-transformation-too-iv}[OF assms].

\text{moreover have} \hspace{1em} \forall \hspace{1em} x. ((\phi \Psi)_G) x \neq \text{None} \rightarrow

\text{ListMem} x (\text{strips-problem-variables-of} (\phi \Psi))

\text{using} \hspace{1em} \text{is-valid-problem-sas-plus-then-strips-transformation-too-v}[OF assms].

\text{ultimately show} \hspace{1em} \text{thesis}

\text{using} \hspace{1em} \text{is-valid-problem-strips-def}

\text{unfolding} \hspace{1em} \text{STRIPS-Representation.is-valid-problem-strips-def}

\text{by} \hspace{1em} \text{fastforce}

\text{qed}

\text{lemma} \hspace{1em} \text{set-filter-all-possible-assignments-true-is};

\text{assumes} \hspace{1em} \text{is-valid-problem-sas-plus } \Psi

\text{shows} \hspace{1em} \text{set} (\text{filter} (\lambda (v, a). s (v, a) = \text{Some True})

(\text{all-possible-assignments-for } \Psi))

= \left( \bigcup v \in \text{set } ((\Psi)_v^+) \right). \text{Pair} v ^+ \{ a \in R_+ \Psi \ v. s (v, a) = \text{Some True} \} \right)
proof –
let ?vs = sas-plus-problem.variables-of Ψ
    and ?P = (λ(v, a), s (v, a) = Some True)
let ?l = filter ?P (all-possible-assignments-for Ψ)
have set ?l = set (concat (map (filter ?P) (map (possible-assignments-for Ψ) ?vs)))
    unfolding all-possible-assignments-for-def
    filter-concat[of ?P map (possible-assignments-for Ψ) (sas-plus-problem.variables-of Ψ)]
    by simp
also have ... = set (concat (map (λv. filter ?P (possible-assignments-for Ψ v)) ?vs))
    unfolding map-map comp-apply
    by blast
also have ... = ∪(set ' ((∀v. map (Pair v) (filter (λa. s (v, a) = Some True)) (the (range-of Ψ v)))) ?vs))
    unfolding set-concat set-map...
also have ... = (∪v ∈ set ?vs. Pair v ' set (filter (λa. s (v, a) = Some True) (the (range-of Ψ v))))
    unfolding image-comp[of set] comp-apply set-map...
also have ... = (∪v ∈ set ?vs. Pair v ' (∀a ∈ set (the (range-of Ψ v)). s (v, a) = Some True )
    unfolding set-filter..
finally show ?thesis
    using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
    by auto
qed

lemma strips-state-to-state-dom-is:
assumes is-valid-problem-sas-plus Ψ
shows dom (φ⁻¹ Ψ s)
  = (∪v ∈ set ((Ψ)ψ⁺).
    { v | a ∈ (ψ⁺ Ψ v) ∧ s (v, a) = Some True })
proof –
let ?vs = variables-of Ψ
    and ?s' = φ⁻¹ Ψ s
    and ?P = (λ(v, a), s (v, a) = Some True)
let ?l = filter ?P (all-possible-assignments-for Ψ)
{ have fst ' set ?l = fst ' (∪v ∈ set ?vs. Pair v

101
\(a \in \mathbb{R}^+ \Psi v. s(v, a) = \text{Some True}\)

unfolding \(\text{set-filter-all-possible-assignments-true-is[OF assms]}\) by auto

also have \(\ldots = (\bigcup v \in \text{set } \text{vs} \cdot \text{fst } (\text{Pair } v)\) by blast

also have \(\ldots = (\bigcup v \in \text{set } \text{vs} \cdot (\lambda a. \text{fst } (\text{Pair } v a))\) unfolding \(\text{image-comp[of fst]} \text{ comp-apply}\) by blast

finally have \(\text{fst } \text{set } \text{l} = (\bigcup v \in \text{set } ((\Psi)^{+} V) \cdot \{ v \mid a. a \in (\mathbb{R}^+ \Psi v) \wedge s(v, a) = \text{Some True}\})\) unfolding \(\text{setcompr-eq-image fst-conv}\) by simp

thus \(?\text{thesis}\)
unfolding \(\text{SAS-Plus-STRIPS.strips-state-to-state-def}\)
strips-state-to-state-def \(\text{dom-map-of-conv-image-fst}\) by blast

qed

lemma \(\text{strips-state-to-state-range-is}\):
assumes \(\text{is-valid-problem-sas-plus } \Psi\)
and \(v \in \text{set } ((\Psi)^{+} V)\)
and \(a \in \mathbb{R}^+ \Psi v\)
and \((v, a) \in \text{dom } s'\)
and \(\forall (v, a) \in \text{dom } s'. \forall (v, a') \in \text{dom } s'. s'(v, a) = \text{Some True} \wedge s'(v, a') = \text{Some True}\)

shows \((\varphi_{S^{-1}} \Psi s') v = \text{Some } a \iff \text{the } (s'(v, a))\)

proof –
let \(?\text{vs} = \text{variables-of } \Psi\)
and \(?D = \text{range-of } \Psi\)
and \(?s = \varphi_{S^{-1}} \Psi s'\)
let \(?\text{as} = \text{all-possible-assignments-for } \Psi\)
let \(?l = \text{filter } (\lambda(v, a). s'(v, a) = \text{Some True}) \?\text{as}\)
show \(?\text{thesis}\)
proof (rule iffI)
assume \(s\text{-of-}v\text{-is-}a: ?s v = \text{Some } a\)

\begin{align*}
&\{\
&\text{have } (v, a) \in \text{set } \?l \\
&\text{using s-of-}v\text{-is-}a \\
&\text{unfolding } \text{SAS-Plus-STRIPS.strips-state-to-state-def} \\
&\text{strips-state-to-state-def} \\
&\text{using map-of-}s \text{D} \\
&\text{by fast} \\
&\text{hence } s'(v, a) = \text{Some True} \\
&\text{unfolding all-possible-assignments-for-set-is set-filter} \\
&\text{by blast}
\end{align*}
thus the \((s' (v, a))\)
by simp

next
assume the-of-s'-of-v-a-is: the \((s' (v, a))\)
then have s'-of-v-a-is-true: \(s' (v, a) = \text{Some True}\)
using assms(4) domiff
by force
— TODO slow.

moreover { 
fix \(v v' a a'\)
assume \((v, a) \in \text{set } ?l \text{ and } (v', a') \in \text{set } ?l\)
then have \(v \neq v' \lor a = a'\)
using assms(5)
by fastforce
}

moreover { 
have \(\forall v \in \text{set } ((\Psi)V_+) \text{. sas-plus-problem.range-of } \Psi \ v \neq \text{None}\)
using is-valid-problem-sas-plus-then(1) assms(1)
range-of-not-empty
by force
moreover have \(\text{set } ?l = \text{Set.filter } (\lambda(v, a). \ s' (v, a) = \text{Some True})\)
(\(\bigcup v \in \text{set } ((\Psi)V_+). \left\{ (v, a) \mid a. a \in R_+ \Psi v \right\}\))
using all-possible-assignments-for-set-is
by force
ultimately have \((v, a) \in \text{set } ?l\)
using assms(2, 3) s'-of-v-a-is-true
by simp
}
ultimately show ?s v = \text{Some a}
using map-of-constant-assignments-defined-if[of ?l v a]
unfolding SAS-Plus-STRIPS.strips-state-to-state-def
strips-state-to-state-def
by blast
qed

qed

— NOTE A technical lemma which characterizes the return values for possible
assignments \((v, a)\) when used as variables on a state \(s\) which was transformed
from.

lemma strips-state-to-state-inverse-is-i:
assumes is-valid-problem-sas-plus \(\Psi\)
and \(v \in \text{set } ((\Psi)V_+)\)
and \(s v \neq \text{None}\)
and \(a \in R_+ \Psi v\)
shows \((\varphi_S \Psi s) (v, a) = \text{Some (the } (s v) = a)\)
proof –
let \(\Psi v s = \text{sas-plus-problem.variables-of } \Psi\)
let $s'$ = $\varphi_S \Psi s$

and $f = \lambda(v, a). \text{the } (s v) = a$

and $?l = \text{concat } (\text{map } (\text{possible-assignments-for } \Psi) (\text{filter } (\lambda v. s v \neq \text{None} ))(\text{?es}))$

have $(v, a) \in \text{dom } ?s'$

using $\text{state-to-strips-state-dom-element-iff}$

by presburger

{ have $v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V^+}) \land s v \neq \text{None } \}$

using $\text{assms}(2, 3)$

by blast

moreover have $\forall v \in \text{set } ((\Psi)_{V^+}). v \in \text{dom } (\text{sas-plus-problem-range-of } \Psi)$

using $\text{is-valid-problem-sas-plus-dom-sas-plus-problem-range-of}[\text{OF } \text{assms}(1)]$

moreover have set $?l = (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V^+}) \land s v \neq \text{None } \}).$

{ $(v, a) \mid a \in \mathcal{R}_+ \Psi v$ }

unfolding $\text{state-to-strips-state-dom-is-i}[\text{OF calculation}(2)]$

by blast

ultimately have $(v, a) \in \text{set } ?l$

using $\text{assms}(4)$

by blast

}

moreover have set $?l \neq \{ \}$

using $\text{calculation}$

by force

— TODO slow.

ultimately show ?thesis

unfolding $\text{SAS-Plus-STRIPS.state-to-strips-state-def}$

$\text{state-to-strips-state-def}$

using $\text{map-of-from-function-graph-is-some-if}[\text{OF } ?l (v, a) f]$

unfolding $\text{split-def}$

by fastforce

qed

— NOTE Show that the transformed strips state is consistent for pairs of assignments $(v, a)$ and $(v, a')$ in the same variable domain.

corollary $\text{strips-state-to-state-inverse-is-ii}$:

assumes $\text{is-valid-problem-sas-plus } \Psi$

and $v \in \text{set } ((\Psi)_{V^+})$

and $s v = \text{Some } a$

and $a \in \mathcal{R}_+ \Psi v$

and $a' \in \mathcal{R}_+ \Psi v$

and $a' \neq a$

shows $(\varphi_S \Psi s) (v, a') = \text{Some False}$

proof — have $s v \neq \text{None}$

using $\text{assms}(3)$
by simp
moreover have the \((s \; v) \neq a'\)
using assms(3, 6)
by simp
ultimately show \(?thesis\)
using strips-state-to-state-inverse-is-i[OF assms(1, 2) - assms(5)]
by force
qed

— NOTE Follows from the corollary above by contraposition.

corollary strips-state-to-state-inverse-is-iii:
assumes is-valid-problem-sas-plus \(Ψ\)
and \(v \in \text{set } ((Ψ)V_+ )\)
and \(s \; v = \text{Some } a\)
and \(a \in \mathbb{R}_+ Ψ \; v\)
and \(a' \in \mathbb{R}_+ Ψ \; v\)
and \((ϕS \; Ψ \; s) \; (v, a) = \text{Some True}\)
and \((ϕS \; Ψ \; s) \; (v, a') = \text{Some True}\)
shows \(a = a'\)
proof –
have \(s \; v \neq \text{None}\)
using assms(3)
by blast
thus \(?thesis\)
using strips-state-to-state-inverse-is-i[OF assms(1, 2)] assms(4, 5, 6, 7)
by auto
qed

lemma strips-state-to-state-inverse-is-iv:
assumes is-valid-problem-sas-plus \(Ψ\)
and \(\text{dom } s \subseteq \text{set } ((Ψ)V_+ )\)
and \(v \in \text{set } ((Ψ)V_+ )\)
and \(s \; v = \text{Some } a\)
and \(a \in \mathbb{R}_+ Ψ \; v\)
shows \((ϕ^{-1}_S Ψ \; (ϕS \; Ψ \; s) ) \; v = \text{Some } a\)
proof –
let \(?vs = \text{variables-of } Ψ\)
and \(?s' = ϕS Ψ \; s\)
let \(?s'' = ϕ^{-1}_S Ψ \; s'\)
let \(?P = \lambda(v, a). \; ?s' \; (v, a) = \text{Some True}\)
let \(?as = \text{filter } ?P \; (\text{all-possible-assignments-for } Ψ)\)
and \(?As = \text{Set.filter } ?P \; ( \bigcup v \in \text{set } ((Ψ)V_+ ). \; \{ (v, a) | \; a. \; a \in \mathbb{R}_+ Ψ \; v \})\)
\{ 
have \(\forall v \in \text{set } ((Ψ)V_+)\). \; \text{range-of } Ψ \; v \neq \text{None}\)
using sublocale-sas-plus-finite-domain-representation-ii(1)[OF assms(1)]
\text{range-of-not-empty}

105
by force

hence set ?as = ?As

unfolding set-filter
using all-possible-assignments-for-set-is
by force

} note nb = this

moreover {

{ fix v v' a a'
  assume (v, a) ∈ set ?as
  and (v', a') ∈ set ?as
  then have (v, a) ∈ ?As and (v', a') ∈ ?As
  using nb
  by blast+
  then have v-in-set-vs: v ∈ set ?vs and v'-in-set-vs: v' ∈ set ?vs
  and a-in-range-of-v: a ∈ ℜ⁺ v
  and a'-in-range-of-v: a' ∈ ℜ⁺ v'
  and s'-of-v-a-is: ?s' (v, a) = Some True and s'-of-v'-a'-is: ?s' (v', a') = Some True
  by fastforce+
  then have (v, a) ∈ dom ?s'
  by blast
  then have s-of-v-is-None-a: s v = Some a
  using state-to-strips-state-dom-element-iff[OF assms(1)]
  state-to-strips-state-range-is[OF assms(1)] s'-of-v-a-is
  by auto
  have v ≠ v' ∨ a = a'
  proof (rule ccontr)
    assume ~(v ≠ v' ∨ a = a')
    then have v = v' and a ≠ a'
    by simp+
    thus False
    using a'-in-range-of-v a-in-range-of-v assms(1) v'-in-set-vs s'-of-v'-a'-is
  s'-of-v-a-is s-of-v-is-None-a strips-state-to-state-inverse-is-iii
  by force
  qed
}

moreover {
  have s v ≠ None
  using assms(4)
  by simp
  then have ?s' (v, a) = Some True
  using strips-state-to-state-inverse-is-i[OF assms(1, 3) - assms(5)]
  assms(4)
  by simp

  hence (v, a) ∈ set ?as
using all-possible-assignments-for-set-is assms(3, 5) nb
by simp
}
ultimately have map-of ?as v = Some a
using map-of-constant-assignments-defined-if[of ?as v a]
by blast
}
— TODO slow.
thusthesis
unfolding SAS-Plus-STRIPS.strips-state-to-state-def
strips-state-to-state-def all-possible-assignments-for-def
by simp
qed

— Show that that $\varphi_{S^{-1}} \Psi$ is the inverse of $\varphi_{S} \Psi$. The additional constraints
dom $s = \text{set (}$Ψ$ \text{)}$ and $\forall v \in \text{dom s}$. $\text{the (}$s v$\text{)} \in \mathcal{R}_{+} \Psi v$ are needed because the
transformation functions only take into account variables and domains declared
in the problem description. They also sufficiently characterize a state that was
transformed from SAS+ to STRIPS.

lemma strips-state-to-state-inverse-is:
assumes is-valid-problem-sas-plus $\Psi$
and dom $s \subseteq \text{set (}$Ψ$ \text{)}$
and $\forall v \in \text{dom s}$. $\text{the (}$s v$\text{)} \in \mathcal{R}_{+} \Psi v$
shows $s = (\varphi_{S^{-1}} \Psi (\varphi_{S} \Psi s))$

proof —
let $?vs = \text{variables-of } \Psi$
and $?D = \text{range-of } \Psi$
let $?s' = \varphi_{S} \Psi s$
let $?s'' = \varphi_{S^{-1}} \Psi ?s'$
— NOTE Show the thesis by proving that $s$ and $?s'$ are mutual submaps.
{
fix $v$
assume v-in-dom-s: $v \in \text{dom s}$
then have v-in-set-vs: $v \in \text{set } ?vs$
using assms(2)
by auto
then obtain $a$
where the-s-v-is-a: $s v = \text{Some } a$
and a-in-dom-v: $a \in \mathcal{R}_{+} \Psi v$
using assms(2, 3) v-in-dom-s
by force
moreover have $?s'' v = \text{Some } a$
using strips-state-to-state-inverse-is-iv[of assms(1, 2)] v-in-set-vs
the-s-v-is-a a-in-dom-v
by force
ultimately have $s v = $?s'' $v$
by argo
moreover { 
  fix v
  assume \( v \in \text{dom } ?s'' \)
  then obtain a
    where \( a \in \mathbb{R}_+ \Psi v \)
    and \( ?s'(v, a) = \text{Some True} \)
    using strips-state-to-state-dom-is[OF assms(1)]
    by blast
  then have \( (v, a) \in \text{dom } ?s' \)
    by blast
  then have \( s \circ v \neq \text{None} \)
    using state-to-strips-state-dom-is[OF assms(1)]
    by simp
  then obtain a where \( s \circ v = \text{Some } a \)
    by blast
  hence \( ?s'' \circ v = s \circ v \)
    using nb
    by fastforce
}

— TODO slow.

ultimately show \( ?\text{thesis} \)
  using map-le-antisym[of s ?s''] map-le-def
  unfolding strips-state-to-state-def
  state-to-strips-state-def
  by blast

qed

— An important lemma which shows that the submap relation does not change if we transform the states on either side from SAS+ to STRIPS.

**lemma** state-to-strips-state-map-le-iff:

assumes is-valid-problem-sas-plus \( \Psi \)
  and \( \text{dom } s \subseteq \text{set } ((\Psi)_V)^+ \)
  and \( \forall v \in \text{dom } s. \text{the } (s \circ v) \in \mathbb{R}_+ \Psi v \)
shows \( s \subseteq_m t \longleftrightarrow (\varphi_S \Psi s) \subseteq_m (\varphi_S \Psi t) \)

**proof** —

let \( ?us = \text{variables-of } \Psi \)
  and \( ?D = \text{range-of } \Psi \)
  and \( ?s' = \varphi_S \Psi s \)
  and \( ?t' = \varphi_S \Psi t \)

show \( ?\text{thesis} \)

**proof** (rule iffI)

assume \( s\text{-map-le-t: } s \subseteq_m t \)

{ 
  fix \( v \ a \)
  assume \( (v, a) \in \text{dom } ?s' \)
  moreover have \( v \in \text{set } ((\Psi)_V)^+ \)
    and \( s \circ v \neq \text{None} \)
    and \( a \in \mathbb{R}_+ \Psi v \)
    using state-to-strips-state-dom-is[OF assms(1)]
    calculation
    by blast
}
moreover have \(?s' \(v, a\) = \text{Some (the } (s \ v) = a)\)
using state-to-strips-state-range-is[OF assms(1)] calculation(1)
by meson
moreover have \(v \in \text{dom } s\)
using calculation(3)
by auto
moreover have \(s \ v = t \ v\)
using s-map-le-t calculation(6)
unfolding map-le-def
by blast
moreover have \(t \ v \neq \text{None}\)
using calculation(3, 7)
by argo
moreover have \((v, a) \in \text{dom } ?t'\)
using state-to-strips-state-dom-is[OF assms(1)] calculation(2, 4, 8)
by blast
moreover have \(?t' \(v, a\) = \text{Some (the } (t \ v) = a)\)
using state-to-strips-state-range-is[OF assms(1)] calculation(9)
by simp
ultimately have \(\?s' \(v, a\) = \?t' \(v, a\)\)
by presburger
}

thus \(\?s' \subseteq \text{m } \?t'\)
unfolding map-le-def
by fast

next
assume \(s'\text{-map-le-t'}: \?s' \subseteq \text{m } \?t'\)
{
fix \(v\)
assume \(v\text{-in-dom-s: } v \in \text{dom } s\)
moreover obtain \(a\) where the-of-s-of-v-is-a: the \((s \ v) = a\)
by blast
moreover have \(v\text{-in-us: } v \in \text{set } ((\Psi)_v)\)
and s-of-v-is-not-None: \(s \ v \neq \text{None}\)
and a-in-range-of-v: \(a \in \mathcal{R}_+ \Psi v\)
using assms(2, 3) v-in-dom-s calculation
by blast+
moreover have \((v, a) \in \text{dom } ?s'\)
using state-to-strips-state-dom-is[OF assms(1)]
calculation(3, 4, 5)
by simp
moreover have \(?s' \(v, a\) = \?t' \(v, a\)\)
using \(s'\text{-map-le-t'}\) calculation
unfolding map-le-def
by blast
moreover have \((v, a) \in \text{dom } ?t'\)
using calculation
unfolding domIff
by argo

109
moreover have \(?s'(v, a) = \text{Some (the (s v) = a)\)
and \(?t'(v, a) = \text{Some (the (t v) = a)\)
using state-to-strips-state-range-is[OF assms(1)] calculation
by fast+
moreover have \(s v = \text{Some a\)
by force
moreover have \(?s'(v, a) = \text{Some True\)
using calculation(9, 11)
by fastforce
moreover have \(?t'(v, a) = \text{Some True\)
using calculation(7, 12)
by argo
moreover have the (t v) = a
using calculation(10, 13) try0
by force
moreover { have \(v \in \text{dom } t\)
using state-to-strips-state-dom-element-iff[OF assms(1)]
calculation(8)
by auto
hence \(t v = \text{Some a\)
using calculation(14)
by force
}
ultimately have \(s v = t v\)
by argo
}
thus \(s \subseteq_m t\)
unfolding map-le-def
by simp
qed
qed

— We also show that \(\varphi_{O}^{-1} \Pi\) is the inverse of \(\varphi_{O} \Psi\). Note that this proof is completely mechanical since both the precondition and effect lists are simply being copied when transforming from SAS+ to STRIPS and when transforming back from STRIPS to SAS+.

lemma sas-plus-operator-inverse-is:
assumes is-valid-problem-sas-plus \(\Psi\)
and \(op \in \text{set ((}\Psi)_{O}+)\)
shows \((\varphi_{O}^{-1} \Psi (\varphi_{O} \Psi op)) = op\)
proof -
let \(?op = \varphi_{O}^{-1} \Psi (\varphi_{O} \Psi op)\)
have precondition-of \(?op = \text{precondition-of op\)
unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
SAS-Plus-STRIPS.sasp-op-to-strips-def

by fastforce

moreover have effect-of ?op = effect-of op

unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def

strips-op-to-sasp-def

SAS-Plus-STRIPS.sasp-op-to-strips-def

by force

ultimately show ?thesis

by simp

qed

— Note that we have to make the assumption that op’ is a member of the operator set of the induced STRIPS problem \( \varphi \Psi \). This implies that op’ was transformed from an \( op \in \text{operators-of } \Psi \). If we don’t make this assumption, then multiple STRIPS operators of the form (\( \emptyset \text{ precondition-of } = \emptyset \), \( \emptyset \text{ add-effects-of } = \emptyset \), \( [(v, a), ...] \)) correspond to one SAS+ operator (since the delete effects are being discarded in the transformation function).

lemma strips-operators-inverse-is:

assumes is-valid-problem-sas-plus \( \Psi \)

and \( \text{ops’} \subseteq \text{set} (\{\varphi (\Psi)_{\text{O}}\}) \)

shows (\( \varphi_{\text{O}^{-1}} \Psi \text{ ops’} \)) = \( \text{op’} \)

proof —

let \( ?\Pi = \varphi \Psi \)

obtain \( op \) where \( op \in \text{set} (\{\text{ops}+\}) \) and \( \text{op’} = \varphi_{\text{O}^{-1}} \Psi \text{ op} \)

using assms

by auto

moreover have \( \varphi_{\text{O}^{-1}} \Psi \text{ op’} = \text{op} \)

using sas-plus-operator-inverse-is[OF assms(1) calculation(1)] calculation(2)

by blast

ultimately show ?thesis

by argo

qed

lemma sas-plus-equivalent-to-strips-i-a-I:

assumes is-valid-problem-sas-plus \( \Psi \)

and \( \text{set ops’} \subseteq \text{set} (\{\varphi (\Psi)_{\text{O}}\}) \)

and STRIPS-Semantics.are-all-operators-applicable (\( \varphi_{\text{S}^{-1}} \Psi \text{ s} \)) \( \text{ops}’ \)

and \( \text{op} \in \text{set} [\varphi_{\text{O}^{-1}} \Psi \text{ op’} \leftarrow \text{ops’}] \)

shows \( \text{map-of} (\text{precondition-of } \text{op}) \subseteq m (\varphi_{\text{S}^{-1}} \Psi (\varphi_{\text{S}^{-1}} \Psi \text{ s})) \)

proof —

let \( ?\Pi = \varphi \Psi \)

and \( ?s’ = \varphi_{\text{S}} \Psi \text{ s} \)

let \( ?s = \varphi_{\text{S}^{-1}} \Psi ?s’ \)

and \( ?D = \text{range-of } \Psi \)

and \( ?\text{ops} = [\varphi_{\text{O}^{-1}} \Psi \text{ op’}, \text{op’} \leftarrow \text{ops’}] \)

and \( ?\text{pre} = \text{precondition-of } \text{op} \)
have \( nb_1 \): \( \forall (v, a) \in \text{dom} \ ?s' \).
\( \forall (v, a') \in \text{dom} \ ?s' \):
\( ?s' (v, a) = \text{Some True} \land ?s' (v, a') = \text{Some True} \)
\[ (v, a) = (v, a') \]
using \text{state-to-strips-state-effect-consistent}[\text{OF assms}(1)]
by blast

\[
\{
\text{fix op'}
\text{assume op' \in set ops'}
\text{moreover have op' \in set } ((\bar{\Pi})_{\land})
\text{using assms(2) calculation}
\text{by blast}
\text{ultimately have } \exists \text{ op } \in \text{ set } ((\Psi)_{O+}). \text{ op'} = (\varphi_O \ Psi \ op)
\text{by auto}
\}
\text{note nb_2 = this}
\{
\text{fix op}
\text{assume op \in set ?ops}
\text{then obtain op'' where op'' \in set ops' and op} = \varphi_O^{-1} \ Psi \ op'
\text{using assms(4)}
\text{by auto}
\text{moreover obtain op'' where op'' \in set } ((\Psi)_{O+}) \text{ and op'} = \varphi_O \ Psi \ op''
\text{using nb_2 calculation(1)}
\text{by blast}
\text{moreover have op = op''}
\text{using sas-plus-operator-inverse-is}[\text{OF assms(1) calculation(3)}] \text{ calculation(2, 4)}
\text{by blast}
\text{ultimately have op \in set } ((\Psi)_{O+})
\text{by blast}
\}
\text{note nb_3 = this}
\{
\text{fix op \ v \ a}
\text{assume op \in set ?ops}
ad v-a-in-precondition-of-op': (v, a) \in \text{ set } (\text{precondition-of op})
\text{moreover obtain op' where op' \in set ops' and op} = \varphi_O^{-1} \ Psi \ op'
\text{using calculation(1)}
\text{by auto}
\text{moreover have strips-operator.precondition-of op' = precondition-of op}
\text{using calculation(4)}
\text{unfolding \text{SAS-Plus-STRIPS.strips-op-to-sasp-def}}
\text{strips-op-to-sasp-def}
\text{by simp}
\text{ultimately have } \exists \text{ op' \in set ops'. op} = (\varphi_O^{-1} \ Psi \ op')
\land (v, a) \in \text{ set } (\text{strips-operator.precondition-of op'})
\text{by metis}
\}
\text{note nb_4 = this}
\{
\text{fix op' \ v \ a}
assume \( op' \in \text{set ops'} \)
and \( v, a \in \text{set (strips-operator.precondition-of op')} \)

moreover have \( s' \text{-of-v-a-is-}\text{Some-True}: ?s'(v, a) = \text{Some True} \)
using \( \text{assms}(3) \) calculation(1, 2)
unfolding \( \text{are-all-operators-applicable-set} \)
by blast

moreover have \( \{ \)
\( \text{obtain} \ op \text{ where} \ \\ \ \ \ ao \in (\Psi)O+ \text{ and} \ ao' = \varphi_O \Psi \ \ \ \ \ \ \ ) \)
using \( \text{nb}_2 \) calculation(1)
by blast
moreover have \( \text{strips-operator.precondition-of op'} = \text{precondition-of op} \)
using \( \text{calculation(2)} \)
unfolding \( \text{sas-plus-STRIPS.sasp-op-to-strips-def} \)
\( \text{sasp-op-to-strips-def} \)
by simp
moreover have \( (v, a) \in \text{set (precondition-of op)} \)
using \( \text{v-a-in-precondition-of-op'} \) calculation(3)
by argo
moreover have \( \text{is-valid-operator-sas-plus} \Psi \ \ \ \ \ \ \ ) \)
using \( \text{is-valid-problem-sas-plus-then(2) assms(1) calculation(1))} \)
unfolding \( \text{is-valid-operator-sas-plus-def} \)
by auto
moreover have \( v \in \text{set ((\Psi)v+)} \text{ and} \ \ \ a \in \mathbb{R}_+ \Psi \ v \)
using \( \text{is-valid-operator-sas-plus-then(1,2) calculation(4, 5))} \)
unfolding \( \text{is-valid-operator-sas-plus-def} \)
by fastforce+ 
mmoreover have \( v \in \text{dom ?s} \)
using \( \text{strips-state-to-state-dom-is[OF assms(1), of ?s']} \)
\( \text{s'\text{-of-v-a-is-}\text{Some-True calculation(6, 7))} \)
by blast
moreover have \( (v, a) \in \text{dom ?s'} \)
using \( \text{s'\text{-of-v-a-is-}\text{Some-True domIff} \ \ \ \ \ \ \ )} \)
by blast
ultimately have \( ?s \ v = \text{Some a} \)
using \( \text{strips-state-to-state-range-is[OF assms(1) - - nb_1]} \)
\( \text{s'\text{-of-v-a-is-}\text{Som} \ e-True} \)
by simp
\( \} \)
hence \( ?s \ v = \text{Some a} \).
\} note \( \text{nb}_5 = \text{this} \)
\{ 
fix \( v \)
assume \( v \in \text{dom (map-of ?pre)} \)
then obtain \( a \text{ where map-of ?pre} \ v = \text{Some a} \)
by fast
moreover have \( (v, a) \in \text{set ?pre} \)
using \( \text{map-of-}\text{SomeD calculation} \)
by fast

113
moreover { 
  have \( \text{op} \in \text{set} \((\Psi)_O+)\)  
  using assms(4) nb3  
  by blast  
  then have is-valid-operator-sas-plus \(\Psi\) \(\text{op}\)  
  using is-valid-problem-sas-plus-then(2) assms(1)  
  unfolding is-valid-operator-sas-plus-def  
  by auto  
  hence \(\forall (v, a) \in \text{set} \ ?\text{pre} \ . \ \forall (v', a') \in \text{set} \ ?\text{pre} \ . v \neq v' \lor a = a'\)  
  using is-valid-operator-sas-plus-then(5)  
  unfolding is-valid-operator-sas-plus-def  
  by fast 
}

moreover have map-of \(?\text{pre}\) \(v\) = Some \(a\)  
  using map-of-constant-assignments-defined-if[\of \ ?\text{pre}] calculation(2, 3)  
  by blast  

moreover obtain \(\text{op}'\) where \(\text{op}' \in \text{set} \text{ops}'\)  
  and \((v, a) \in \text{set} \text{strips-operator.precondition-of \text{op}'}\)  
  using nb4[OF assms(4) calculation(2)]  
  by blast  

moreover have \(?s\ v\) = Some \(a\)  
  using nb5 calculation(5, 6)  
  by fast  

ultimately have map-of \(?\text{pre}\) \(v\) = \(?s\ v\)  
  by argo 
}

thus \(?\text{thesis}\)  
unfolding map-le-def  
by blast  

qed 

lemma to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure: 
assumes is-valid-problem-sas-plus \(\Psi\)  
and set \text{ops}' \subseteq \text{set} \((\varphi \ ?\Psi)_O)\)  
and \(\text{op} \in \text{set} \ \{\varphi_{O^{-1}} \ ?\Psi \ \text{op}', \ \text{op}' \leftarrow \text{ops}'\}\)  
shows \(\text{op} \in \text{set} \ ((\Psi)_O+) \land (\exists \text{op}' \in \text{set} \ \text{ops}'. \ \text{op}' = \varphi_O \ ?\Psi \ \text{op})\)

proof –  
let \(?\Pi = \varphi \ ?\Psi\)  
obtain \(\text{op}'\) where \(\text{op}' \in \text{set} \ \text{ops}'\) and \(\text{op} = \varphi_{O^{-1}} \ ?\Psi \ \text{op}'\)  
  using assms(3)  
  by auto  

moreover have \(\text{op}' \in \text{set} \ ((?\Pi)_O)\)  
  using assms(2) calculation(1)  
  by blast  

moreover obtain \(\text{op}''\) where \(\text{op}'' \in \text{set} \((\Psi)_O+)\) and \(\text{op}' = \varphi_O \ ?\Psi \ \text{op}''\)  
  using calculation(3)  
  by auto  

moreover have \(\text{op} = \text{op}''\)  
  using sas-plus-operator-inverse-is[OF assms(1) calculation(4)] calculation(2),
5)

by presburger
ultimately show ?thesis
by blast
qed

lemma sas-plus-equivalent-to-strips-i-a-II:
fixes Ψ :: ('variable, 'domain) sas-plus-problem
fixes s :: ('variable, 'domain) state
assumes is-valid-problem-sas-plus Ψ
and set ops' ⊆ set ((φ Ψ)ο)
and STRIPS-Semantics.are-all-operators-applicable (φ s Ψ) ops'
∧ STRIPS-Semantics.are-all-operator-effects-consistent ops'
shows are-all-operator-effects-consistent [φO⁻¹ Ψ op', op' ← ops']
proof –
let ?s' = φS Ψ s
let ?s = φS⁻¹ Ψ ?s'
and ?ops = [φO⁻¹ Ψ op', op' ← ops']
and ?Π = φ Ψ
have nb: ∀ (v, a) ∈ dom ?s'.
∀ (v', a') ∈ dom ?s'.
?s'(v, a) = Some True ∧ ?s'(v', a') = Some True
→ (v, a) = (v', a')
using state-to-strips-state-effect-consistent[OF assms(1)]
by blast

{ fix op1' op2'
assume op1' ∈ set ops' and op2' ∈ set ops'
hence STRIPS-Semantics.are-operator-effects-consistent op1' op2'
using assms(3)
unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
by blast
}

note nb1 = this

{ fix op1 op1' op2 op2'
assume op1'-in-ops: op1' ∈ set ?ops
and op1'-in-ops': op1' ∈ set ops'
and op1'-is: op1' = φO Ψ op1
and is-valid-op1: is-valid-operator-sas-plus Ψ op1
and op2-in-ops: op2 ∈ set ?ops
and op2'-in-ops': op2' ∈ set ops'
and op2'-is: op2' = φO Ψ op2
and is-valid-op2: is-valid-operator-sas-plus Ψ op2
have ∀ (v, a) ∈ set (add-effects-of op1'). ∀ (v', a') ∈ set (add-effects-of op2').
v ≠ v' ∨ a = a'
proof (rule ccontr)
assume ¬(∀ (v, a) ∈ set (add-effects-of op1'). ∀ (v', a') ∈ set (add-effects-of op2')).
\[
v \neq v' \lor a = a'
\]
then obtain \( v v' a a' \) where \((v, a) \in \text{set (add-effects-of } op_1')\)
and \((v', a') \in \text{set (add-effects-of } op_2')\)
and \(v = v'\)
and \(a \neq a'\)
by blast
— TODO slow.
moreover have \((v, a) \in \text{set (effect-of } op_1)\)
using \( op_1'-\text{is } op_2'-\text{is calculation}(1, 2)\)
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
by force
moreover have \{(v', a') \in \text{set (effect-of } op_2)\}
using \( op_2'-\text{is calculation}(2)\)
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
by force
hence \(a' \in \mathcal{R}_+ \Psi v\)
using is-valid-operator-sas-plus-then is-valid-op_2 calculation(3)
by fastforce
moreover have \((v, a') \in \text{set (delete-effects-of } op_1')\)
using sasp-op-to-strips-set-delete-effects-is
\( op_1'-\text{is valid-op_1 calculation}(3, 4, 5, 6)\)
by blast
moreover have \(\sim \text{STRIPS-Semantics.are-operator-effects-consistent } op_1'\)
\( op_2'\)
unfolding STRIPS-Semantics.are-operator-effects-consistent-def list-ex-iff
using calculation(2, 3, 7)
by meson
ultimately show False
using assms(3) \( op_1'-\text{in-ops}' \ op_2'-\text{in-ops}'\)
unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
by blast
qed

note \(nb_3 = \text{this}\)

\{ fix \( op_1 \ op_2 \)
assume \( op_1'-\text{in-ops}: op_1 \in \text{set } \mathcal{O} \ \text{ops and } op_2'-\text{in-ops}: op_2 \in \text{set } \mathcal{O} \ \text{ops} \)
moreover have \( op_1'-\text{in-operators-of-}\Psi: op_1 \in \text{set } ((\Psi)_{\mathcal{O}}+)\)
and \( op_2'-\text{in-operators-of-}\Psi: op_2 \in \text{set } ((\Psi)_{\mathcal{O}}+)\)
using to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure[OF
assms(1, 2)] calculation
by blast+
moreover have \(\text{is-valid-operator-op_1}: \text{is-valid-operator-sas-plus } \Psi op_1\)
and \(\text{is-valid-operator-op_2}: \text{is-valid-operator-sas-plus } \Psi op_2\)
using \(\text{is-valid-problem-sas-plus-then}(2) \ op_1'-\text{in-operators-of-}\Psi \ op_2'-\text{in-operators-of-}\Psi\)
assms(1)

unfolding is-valid-operator-sas-plus-def
by auto+

moreover obtain op_1' op_2'
where op_1-in-ops': op_1' ∈ set ops'
and op_1-is: op_1' = φ_O Ψ op_1
and op_2-in-ops': op_2' ∈ set ops'
and op_2-is: op_2' = φ_O Ψ op_2
using to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure[OF
assms(1, 2)] op_1-in-ops op_2-in-ops
by blast
— TODO slow.

ultimately have ∀ (v, a) ∈ set (add-effects-of op_1'). ∀ (v', a') ∈ set (add-effects-of op_2').
  v ≠ v' ∨ a = a'
using nb_3
by auto

hence are-operator-effects-consistent op_1 op_2
using op_1-is op_2-is
unfolding are-operator-effects-consistent-def
sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
list-all-iff Let-def
by simp

)}
thus ?thesis

unfolding are-all-operator-effects-consistent-def list-all-iff
by fast

qed

— A technical lemmas used in sas-plus-equivalent-to-strips-i-a showing that the execution precondition is linear w.r.t. to STRIPS transformation to SAS+. The second premise states that the given STRIPS state corresponds to a consistent SAS+ state (i.e. no two assignments of the same variable to different values exist).

lemma sas-plus-equivalent-to-strips-i-a-IV:
assumes is-valid-problem-sas-plus Ψ
and set ops' ⊆ set ((φ Ψ)Ω)
and STRIPS-Semantics.are-all-operators-applicable (φ_S Ψ s) ops'
∧ STRIPS-Semantics.are-all-operator-effects-consistent ops'
shows are-all-operators-applicable-in (φ_S⁻¹ Ψ (φ_S Ψ s)) [φ_O⁻¹ Ψ op'. op' ←
ops'] ∧
are-all-operator-effects-consistent [φ_O⁻¹ Ψ op'. op' ← ops']

proof —
let ?Π = φ Ψ
and ?s' = φ_S Ψ s
let ?ops' = strips-problem.variables-of ?Π
and ?ops' = strips-problem.operators-of ?Π
and ?vs = variables-of Ψ

117
and \( \mathcal{D} = \text{range-of } \Psi \)
and \( \mathcal{S} = \varphi\Psi^{-1} \mathcal{S} \mathcal{S}' \)
and \( \mathcal{O} = [\varphi\Omega^{-1} \Psi \mathcal{O}', \mathcal{O}' \leftarrow \mathcal{O}] \)

have \( \text{nb}: \forall (v, a) \in \text{dom } \mathcal{S}'\).
\( \forall (v, a') \in \text{dom } (\varphi\Psi \mathcal{S}) \).
\( \mathcal{S}'(v, a) = \text{Some True} \land \mathcal{S}'(v, a') = \text{Some True} \)
\( \rightarrow (v, a) = (v, a') \)

using state-to-strips-state-effect-consistent[\text{OF assms}(1)]
by blast

have STRIPS-Semantics.are-all-operators-applicable \( \mathcal{S}' \mathcal{O}' \)
using assms(3)
by simp

moreover have list-all \( (\lambda \mathcal{O}. \text{map-of (precondition-of } \mathcal{O}) \subseteq_m \mathcal{S}) \mathcal{O} \)
using sas-plus-equivalent-to-strips-i-a-I[\text{OF assms}(1) \text{ assms(2)] calculation}

unfolding list-all-iff
by blast

moreover have list-all \( (\lambda \mathcal{O}. \text{list-all (arc-operator-effects-consistent op) } \mathcal{O} \mathcal{S}) \mathcal{O} \)

using sas-plus-equivalent-to-strips-i-a-II assms nb

unfolding are-all-operators-applicable-in \( \mathcal{S} \mathcal{O} \)

by blast

ultimately have are-all-operators-applicable-in \( \mathcal{S} \mathcal{O} \)

by simp

ultimately show \( ?\text{thesis} \)
by simp

qed

\textbf{lemma sas-plus-equivalent-to-strips-i-a-VI:}

assumes is-valid-problem-sas-plus \( \Psi \)
and \( \text{dom } \mathcal{S} \subseteq \text{set } (\varphi\Psi)_+ \)
and \( \forall v \in \text{dom } \mathcal{S}. \text{the } (s v) \in R_+ \Psi v \)
and \( \text{set } \mathcal{O}' \subseteq \text{set } (\varphi\Psi)_\Omega \)
and are-all-operators-applicable-in \( s [\varphi\Omega^{-1} \Psi \mathcal{O}' \mathcal{O} \mathcal{O}', \mathcal{O}' \leftarrow \mathcal{O}] \mathcal{O} \land \)
are-all-operator-effects-consistent \( [\varphi\Omega^{-1} \Psi \mathcal{O}' \mathcal{O} \mathcal{O}', \mathcal{O}' \leftarrow \mathcal{O}] \mathcal{O} \)

shows STRIPS-Semantics.are-all-operators-applicable \( (\varphi\Psi s) \mathcal{O} \mathcal{S}' \)

proof –

let \( ?\mathcal{S} = \text{variables-of } \Psi \)
and \( \mathcal{D} = \text{range-of } \Psi \)
and \( \mathcal{II} = \varphi\Psi \)
and \( \mathcal{O} = [\varphi\Omega^{-1} \Psi \mathcal{O}' \mathcal{O}', \mathcal{O}' \leftarrow \mathcal{O}] \)
and \( ?\mathcal{S}' = \varphi\Psi s \)

118
— TODO refactor.

\[
\begin{align*}
\{ & \text{fix } op' \\
& \text{assume } op' \in \text{set } ops' \\
& \text{moreover obtain } op \text{ where } op \in \text{set } ?ops \text{ and } op = \varphi_O^{-1} \Psi op' \\
& \quad \text{using calculation} \\
& \quad \text{by force} \\
& \text{moreover obtain } op'' \text{ where } op'' \in \text{set } (\Psi)_O^+ \text{ and } op' = \varphi_O \Psi op'' \\
& \quad \text{using assms(4) calculation(1)} \\
& \quad \text{by auto} \\
& \text{moreover have } \text{is-valid-operator-sas-plus } \Psi op'' \\
& \quad \text{using is-valid-problem-sas-plus-then(2) assms(1) calculation(4)} \\
& \quad \text{unfolding is-valid-operator-sas-plus-def} \\
& \quad \text{by auto} \\
& \text{moreover have } op = op'' \\
& \quad \text{using sas-plus-operator-inverse-is[OF assms(1)] calculation(3, 4, 5)} \\
& \quad \text{by blast} \\
\end{align*}
\]

ultimately have \( \exists op \in \text{set } ?ops. \text{ op } \in \text{set } ?ops \wedge op = (\varphi_O^{-1} \Psi op') \wedge \text{is-valid-operator-sas-plus } \Psi op \)

by blast

\[
\{ \text{note nb}_1 = \text{this} \}
\]

have \( \text{nb}_2: \forall (v, a) \in \text{dom } ?s'. \forall (v', a') \in \text{dom } ?s'. \forall (v, a) = \text{Some True} \wedge ?s'(v, a') = \text{Some True} \)

\[\rightarrow (v, a) = (v', a') \]

using state-to-strips-state-effect-consistent[OF assms(1), of - - s]

by blast

\[
\{ \text{note nb}_3 = \text{this} \}
\]

\[
\{ \text{fix } op' \\
\text{assume } op' \in \text{set } ops' \\
\text{then obtain } op \text{ where } op \text{-in-ops: } op \in \text{set } ?ops \\
\quad \text{and } op-is: op = (\varphi_O^{-1} \Psi op') \\
\quad \text{and is-valid-operator-op: is-valid-operator-sas-plus } \Psi op \\
\quad \text{using nb}_1 \\
\quad \text{by force} \\
\text{moreover have } \text{preconditions-are-consistent:} \\
\forall (v, a) \in \text{set } (\text{precondition-of op}). \forall (v', a') \in \text{set } (\text{precondition-of op}). v \neq v' \vee a = a' \\
\quad \text{using is-valid-operator-sas-plus-then(5) calculation(3)} \\
\quad \text{unfolding is-valid-operator-sas-plus-def} \\
\}
\]

119
by fast
moreover 
  fix v a
  assume (v, a) ∈ set (strips-operator.precondition-of op')
moreover have v-a-in-precondition-of-op: (v, a) ∈ set (precondition-of op)
  using op-is calculation
  unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
by auto
moreover have map-of (precondition-of op) v = Some a
  using map-of-constant-assignments-defined-if[OF
    preconditions-are-consistent calculation(2)]
by blast
moreover have s-of-v-is: s v = Some a
  using nb3[OF op-in-ops] calculation(3)
  unfolding map-let-def
by force
moreover have v ∈ set ((ψ)Ψ+) and a ∈ Ψ v
  using is-valid-operator-sas-plus-then(1, 2) is-valid-operator-op
v-a-in-precondition-of-op
  unfolding is-valid-operator-sas-plus-def
SAS-Plus-Representation.is-valid-operator-sas-plus-def Let-def list-all-iff
ListMem-iff
  by auto+
moreover have (v, a) ∈ dom ?s'
  using state-to-strips-state-dom-is[OF assms(1)] s-of-v-is
  calculation
by simp
moreover have (φ⁻¹Ψ ?s') v = Some a
  using strips-state-to-state-inverse-is[OF assms(1, 2, 3)] s-of-v-is
by argo
— TODO slow.
ultimately have ?s' (v, a) = Some True
  using strips-state-to-state-range-is[OF assms(1)] nb2
by auto
}
ultimately have ∀ (v, a) ∈ set (strips-operator.precondition-of op'). ?s' (v, a) = Some True
  by fast
}
thus thesis
  unfolding are-all-operators-applicable-def is-operator-applicable-in-def
  STRIPS-Representation.is-operator-applicable-in-def list-all-iff
by simp
qed

lemma sas-plus-equivalent-to-strips-i-a-VII:
assumes is-valid-problem-sas-plus Ψ
and $\text{dom } s \subseteq \text{set } ((\varPsi)_V)$
and $\forall v \in \text{dom } s. \ (s v) \in \mathcal{R}_+ \ \varPsi \ v$
and set ops' $\subseteq \text{set } ((\varphi \ \varPsi)_O)$
and are-all-operators-applicable-in $s$ $[\varphi_0^{-1} \ \varPsi \ \text{ops'} \ \text{op'} \leftarrow \text{ops'}] \land$
are-all-operator-effects-consistent $[\varphi_0^{-1} \ \varPsi \ \text{op'} \ \text{op'} \leftarrow \text{ops'}]$
shows STRIPS-Semantics.are-all-operator-effects-consistent ops'

proof –
let $?s' = \varphi_S \ \varPsi \ s$
and $?\text{ops} = [\varphi_0^{-1} \ \varPsi \ \text{op'} \ \text{op'} \leftarrow \text{ops'}]
and $?D = \text{range-of } \varPsi$
and $?\Pi = \varphi \ \varPsi$
— TODO refactor.

{ 
fix \text{op'}
assume \text{op'} $\in \text{set } \text{ops'}$
morerover obtain \text{op} where \text{op} $\in \text{set } \text{?ops}$ and \text{op} = $\varphi_0^{-1} \ \varPsi \ \text{op'}$
using calculation
by force
morerover obtain \text{op''} where \text{op''} $\in \text{set } ((\varPsi)_O)$ and \text{op'} = $\varphi_0 \ \varPsi \ \text{op''}$
using \text{assms}(4) calculation(1)
by auto
morerover have is-valid-operator-sas-plus $\varPsi \ \text{op''}$
using is-valid-problem-sas-plus-then(2) \text{assms}(1) calculation(4)
unfolding is-valid-operator-sas-plus-def
by auto
morerover have \text{op} = \text{op''}
using sas-plus-operator-inverse-is[OF \text{assms}(1)] calculation(3, 4, 5)
by blast
ultimately have $\exists \text{op} \in \text{set } \text{?ops}. \ \text{op} \in \text{set } \text{?ops} \land \text{op'} = (\varphi_0 \ \varPsi \ \text{op})$
$\land$ is-valid-operator-sas-plus $\varPsi \ \text{op}$
by blast
}
note nb1 = this

{ 
fix $\text{op}_1$ $\text{op}_2'$
assume $\text{op}_1'$ $\in \text{set } \text{ops'}$
and $\text{op}_2'$ $\in \text{set } \text{ops'}$
and $\exists (v, a) \in \text{set } \text{(add-effects-of } \text{op}_1'). \ \exists (v', a') \in \text{set } \text{(delete-effects-of } \text{op}_2').$
$(v, a) = (v', a')$
morerover obtain $\text{op}_1$ $\text{op}_2$
where $\text{op}_1$ $\in \text{set } \text{?ops}$
and $\text{op}_1' = \varphi_0 \ \varPsi \ \text{op}_1$
and is-valid-operator-sas-plus $\varPsi \ \text{op}_1$
and $\text{op}_2$ $\in \text{set } \text{?ops}$
and $\text{op}_2' = \varphi_0 \ \varPsi \ \text{op}_2$
and is-valid-op2: is-valid-operator-sas-plus $\varPsi \ \text{op}_2$
using nb1 calculation(1, 2)
by meson
morerover obtain $v v' a a'$
where $(v, a) \in \text{set } \text{(add-effects-of } \text{op}_1')$
and \((v', a') \in \text{set (delete-effects-of op_2')}\)
and \((v, a) = (v', a')\)
using calculation
by blast
moreover have \((v, a) \in \text{set (effect-of op_1)}\)
using calculation(5, 10)
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
by fastforce
moreover have \(v = v'\) and \(a = a'\)
using calculation(12)
by simp+
— The next proof block shows that \((v', a')\) is constructed from an effect \((v'', a'')\) s.t. \(a' \neq a''\).
moreover {

have \((v', a') \in \bigcup \{ (v'', a'') \in \text{set (effect-of op_2')} \mid \exists a'''. (v'', a''') \in \text{set (effect-of op_2')} \land a''' \in (\mathcal{R}_+ \Psi v'') \land a''' \neq a''')\}\)
using sasp-op-to-strips-set-delete-effects-is

by blast
then obtain \(v'' a''\) where \((v'', a'') \in \text{set (effect-of op_2')}\)
and \((v', a') \in \{ (v'', a''') \mid a'''. a''' \in (\mathcal{R}_+ \Psi v'') \land a''' \neq a''')\}\)
by blast
moreover have \((v', a'') \in \text{set (effect-of op_2')}\)
using calculation
by blast
moreover have \(a' \in \mathcal{R}_+ \Psi v''\) and \(a' \neq a''\)
using calculation(1, 2)
by fast+
ultimately have \(\exists a''. (v', a'') \in \text{set (effect-of op_2')} \land a' \in (\mathcal{R}_+ \Psi v')\)
\wedge \(a' \neq a''\)
by blast
}
moreover obtain \(a''\) where \((v', a'') \in \text{set (effect-of op_2')}\)
and \(a' \in \mathcal{R}_+ \Psi v'\)
and \(a' \neq a''\)
using calculation(16)
by blast
moreover have \(\exists (v, a) \in \text{set (effect-of op_1)}. (\exists (v', a') \in \text{set (effect-of op_2)} . v = v' \wedge a \neq a')\)
using calculation(13, 14, 15, 17, 19)
by blast
moreover have \(\neg \text{are-operator-effects-consistent op_1 op_2}\)
unfolding are-operator-effects-consistent-def list-all-iff
using calculation(20)
by fastforce
ultimately have \(\neg \text{are-all-operator-effects-consistent ?ops}\)
unfolding are-all-operator-effects-consistent-def list-all-iff
by meson 
} note nb2 = this 
{
fix op₁' op₂'
assume op₁'-in-ops: op₁' ∈ set ops' and op₂'-in-ops: op₂' ∈ set ops'
have STRIPS-Semantics. are-operator-effects-consistent op₁' op₂'
proof (rule ccontr)
assume ¬STRIPS-Semantics. are-operator-effects-consistent op₁' op₂'
then consider (A) ∃(v, a) ∈ set (add-effects-of op₁').
| (B) ∃(v, a) ∈ set (add-effects-of op₂').
| ∃(v', a') ∈ set (delete-effects-of op₁'). (v, a) = (v', a')
unfolding STRIPS-Semantics. are-operator-effects-consistent-def list-ex-iff
by fastforce
thus False
using nb2[OF op₁'-in-ops op₂'-in-ops] nb2[OF op₂'-in-ops op₁'-in-ops]
assms(5)
by (cases, argo, force)
qed
}

thus ?thesis
unfolding STRIPS-Semantics. are-all-operator-effects-consistent-def
STRIPS-Semantics. are-operator-effects-consistent-def list-all-iff
by blast
qed

lemma sas-plus-equivalent-to-strips-i-a-VIII:
assumes is-valid-problem-sas-plus Ψ
and dom s ⊆ set ((Ψ)Υ⁺)
and ∀ v ∈ dom s. the (s v) ∈ R⁺ Ψ v
and set ops' ⊆ set ((φ Ψ)Ο)
and are-all-operators-applicable-in s [φ⁻¹ Ψ op', op' ← ops'] ∧
are-all-operator-effects-consistent [φ⁻¹ Ψ op', op' ← ops']
shows STRIPS-Semantics. are-all-operators-applicable (φ Ψ s) ops'
∧ STRIPS-Semantics. are-all-operator-effects-consistent ops'
using sas-plus-equivalent-to-strips-i-a-VI sas-plus-equivalent-to-strips-i-a-VII assms
by fastforce

lemma sas-plus-equivalent-to-strips-i-a-IX:
assumes dom s ⊆ V
and ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op). v ∈ V
shows dom (execute-parallel-operator-sas-plus s ops) ⊆ V
proof –
show ?thesis
using assms
proof (induction ops arbitrary: s)
case Nil
then show ?case

123
unfolding \texttt{execute-parallel-operator-sas-plus-def} by \texttt{simp}

\textbf{next}

\textbf{case} (\texttt{Cons op ops})

\textbf{let} \( \texttt{?s'} = \texttt{s ++ map-of (effect-of \texttt{op})} \)

— TODO Wrap IH instantiation in block.

\{ \hspace{1cm}

\textbf{have} \( \forall (v, a) \in \text{set (effect-of \texttt{op})}. v \in V \)

using \texttt{Cons.prems(2)}

by \texttt{fastforce}

\textbf{moreover have} \( \texttt{fst ' set (effect-of \texttt{op}) \subseteq V} \)

using \texttt{calculation}

by \texttt{fastforce}

\textbf{ultimately have} \( \texttt{dom \ ?s' \subseteq V} \)

unfolding \texttt{dom-map-add dom-map-of-conv-image-fst}

using \texttt{Cons.prems(1)}

by \texttt{blast}

\}

\textbf{moreover have} \( \forall \texttt{op} \in \texttt{set ops}. \forall (v, a) \in \text{set (effect-of \texttt{op})}. v \in V \)

using \texttt{Cons.prems(2)}

by \texttt{fastforce}

\textbf{ultimately have} \( \texttt{dom \ (execute-parallel-operator-sas-plus \ ?s' \ ops) \subseteq V} \)

using \texttt{Cons.IH[of \ ?s']}

by \texttt{fast}

\textbf{thus } \texttt{?case}

\textbf{unfolding} \texttt{execute-parallel-operator-sas-plus-cons}.

\textbf{qed}

\textbf{qed}

— NOTE Show that the domain value constraint on states is monotonous w.r.t. to valid operator execution. I.e. if a parallel operator is executed on a state for which the domain value constraint holds, the domain value constraint will also hold on the resultant state.

\textbf{lemma} \texttt{sas-plus-equivalent-to-strips-i-a-X:}

\textbf{assumes} \texttt{dom s \subseteq V}

\textbf{and} \( \texttt{V \subseteq dom D} \)

\textbf{and} \( \forall v \in \texttt{dom s}. \texttt{the (s v) \in set (the (D v))} \)

\textbf{and} \( \forall \texttt{op} \in \texttt{set ops}. \forall (v, a) \in \text{set (effect-of \texttt{op})}. \texttt{v \in V \land a \in set (the (D v))} \)

\textbf{shows} \( \forall v \in \texttt{dom \ (execute-parallel-operator-sas-plus \ s \ ops)}. \texttt{the (execute-parallel-operator-sas-plus \ s \ ops \ v) \in set (the (D v))} \)

\textbf{proof} —

\textbf{show } \texttt{?thesis}

using \texttt{assms}

\textbf{proof} (\texttt{induction \texttt{ops arbitrary: \texttt{s})}

\textbf{case} \texttt{Nil}

\textbf{then show } \texttt{?case}

\textbf{unfolding} \texttt{execute-parallel-operator-sas-plus-def}

by \texttt{simp}
next
  case (Cons op ops)
  let ?s' = s ++ map-of (effect-of op)
  |
  have \( \forall (v, a) \in \text{set (effect-of op)}, v \in V \)
    using Cons.prems(4)
    by fastforce
  moreover have \( \text{fst '} \set (effect-of op) \subseteq V \)
    using calculation
    by fastforce
  ultimately have \( \text{dom} \ ?s' \subseteq V \)
    unfolding dom-map-add dom-map-of-conv-image-fst
    using Cons.prems(1)
    by blast
  }
moreover {
  fix v
  assume v-in-dom-s': v \in \text{dom} \ ?s'
  hence the \( (\ ?s' \ v) \in \text{set (the (D v))} \)
  proof (cases v \in \text{dom} \ (map-of (effect-of op)))
    case True
    moreover have \( ?s' \ v = \text{(map-of (effect-of op)) v} \)
      unfolding map-add-dom-app-simps(1)[OF True]
      by blast
    moreover obtain a where \( \text{(map-of (effect-of op)) v = Some a} \)
      using calculation(1)
      by fast
    moreover have \( (v, a) \in \text{set (effect-of op)} \)
      using map-of-SomeD calculation(3)
      by fast
    moreover have \( a \in \text{set (the (D v))} \)
      using Cons.prems(4) calculation(4)
      by fastforce
    ultimately show ?thesis
      by force
  next
  case False
  then show ?thesis
    unfolding map-add-dom-app-simps(3)[OF False]
    using Cons.prems(3) v-in-dom-s'
    by fast
  qed
}
moreover have \( \forall \text{ op } \in \text{ops}, \forall (v, a) \in \text{set (effect-of op)}, v \in V \land a \in \text{set (the (D v))} \)
  using Cons.prems(4)
  by auto
ultimately have \( \forall v \in \text{dom (execute-parallel-operator-sas-plus \ ?s' \ ops)} \).
the (execute-parallel-operator-sas-plus ?s' op ops) ∈ set (the (D v))

using Cons.IH[of s ++ map-of (effect-of op), OF - Cons.prems(2)]
by meson
}

thus ?case

unfolding execute-parallel-operator-sas-plus-cons
by blast

qed

lemma transform-sas-plus-problem-to-strips-problem-operators-valid:
assumes is-valid-problem-sas-plus \(\Psi\)
and \(op' \in \text{set}\ ((\varphi \Psi)_{\sigma})\)

obtains \(op\)
where \(op \in \text{set}\ ((\Psi)_{\sigma+})\)
and \(op' = (\varphi_{\sigma} \Psi op)\) is-valid-operator-sas-plus \(\Psi\) \(op\)

proof –

\{ obtain \(op\) where \(op \in \text{set}\ ((\Psi)_{\sigma+})\) and \(op' = \varphi_{\sigma} \Psi op\)

using assms
by auto

moreover have is-valid-operator-sas-plus \(\Psi\) \(op\)

using is-valid-problem-sas-plus-then(2) assms(1) calculation(1)
by auto

ultimately have \(\exists \, op \in ((\Psi)_{\sigma+}). \, op' = (\varphi_{\sigma} \Psi op)\)
∧ is-valid-operator-sas-plus \(\Psi\) \(op\)
by blast
\}

thus ?thesis

using that
by blast

qed

lemma sas-plus-equivalent-to-strips-i-a-XI:
assumes is-valid-problem-sas-plus \(\Psi\)
and \(op' \in \text{set}\ ((\varphi \Psi)_{\sigma})\)

shows \((\varphi_{\Psi} \Psi \, s) \, ++ \, \text{map-of\ (effect-to-assignments \(op'\))} = \varphi_{\Psi} \, (s \, ++ \, \text{map-of\ (effect-of\ \(\varphi_{\Psi}^{-1} \Psi \, op'\))})\)

proof –

let \(\Pi = \varphi \Psi\)
let \(?us = \text{variables-of } \Psi\)
and \?ops = \text{operators-of } \Psi\)
and \?ops' = \text{strips-problem.operators-of } \Pi\)
let \(?s' = \varphi_{\Psi} \Psi \, s\)
let \(?t = ?s' \, ++ \, \text{map-of\ (effect-to-assignments \(op'\))} \)
and \(?t' = \varphi_{\Psi} \Psi \, (s \, ++ \, \text{map-of\ (effect-of\ \(\varphi_{\Psi}^{-1} \Psi \, op'\))})\)

obtain \(op\) where \(op'\)-is: \(op' = \varphi_{\sigma} \Psi \, op\)
and \(op\)-in-ops: \(op \in \text{set}\ ((\Psi)_{\sigma+})\)
and is-valid-operator-op: is-valid-operator-sas-plus \(\Psi\) \(op\)
using \textit{transform-sas-plus-problem-to-strips-problem-operators-valid}[OF \textit{assms}]
by auto
have \texttt{nb1}: \((\varphi_{O}^{-1} \Psi \text{ op}') = \text{ op})
  using \textit{sas-plus-operator-inverse-is}[OF \textit{assms}(1)] \text{ op}'-is \text{ op-in-ops}
by blast
— TODO refactor.
{
  have \texttt{dom} (map-of (effect-to-assignments \text{ op}'))
  = set (strips-operator.add-effects-of \text{ op}') \cup set (strips-operator.delete-effects-of \text{ op}')
  unfolding dom-map-of-conv-image-fst
by force
— TODO slow.
also have \ldots = set (effect-of \text{ op}) \cup set (strips-operator.delete-effects-of \text{ op}')
  using \text{ op}'-is
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
by auto
— TODO slow.
finally have \texttt{dom} (map-of (effect-to-assignments \text{ op}'))
  = set (effect-of \text{ op}) \cup set (\bigcup \text{ v}, \text{ a} \in \text{ set (effect-of \text{ op})} . \{ \text{ (v, a') | a'. a' } \in (R_{+} \Psi \text{ v}) \land a' \neq a \})
  using sasp-op-to-strips-set-delete-effects-is[OF \textit{is-valid-operator-op} \text{ op}'-is]
by argo
} note \texttt{nb2 = this}
have \texttt{nb3}: \texttt{dom} ?t = \texttt{dom} ?s' \cup set (effect-of \text{ op})
  \cup (\bigcup \text{ (v, a) } \in \text{ set (effect-of \text{ op})} . \{ \text{ (v, a') | a'. a' } \in (R_{+} \Psi \text{ v}) \land a' \neq a \})
  unfolding \texttt{nb2 dom-map-add}
by blast
— TODO refactor.
have \texttt{nb4}: \texttt{dom} (s ++ map-of (effect-of (\varphi_{O}^{-1} \Psi \text{ op}')))
  = \texttt{dom} s \cup \texttt{fst ' set (effect-of \text{ op})}
  unfolding dom-map-add dom-map-of-conv-image-fst \texttt{nb1}
by fast
{
  let \texttt{?u = s ++ map-of (effect-of (\varphi_{O}^{-1} \Psi \text{ op}'))}
  have \texttt{dom ?t'} = (\bigcup \text{ v } \in \{ \text{ v | v, v } \in \text{ set ((\Psi)_{v+}) \land ?u v \neq None } \} . \{ \text{ (v, a) | a, a } \in R_{+} \Psi \text{ v } \})
    using state-to-strips-state-dom-is[OF \textit{assms}(1)]
  by presburger
} note \texttt{nb5 = this}
— TODO refactor.
have \texttt{nb6}: set (add-effects-of \text{ op}') = set (effect-of \text{ op})
  using \text{ op}'-is
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
by auto
— TODO refactor.
\[\text{have } \text{nb}_7: \text{set } \{ (v, a) \in (\mathcal{R}_+ v) : a' \neq a \} = \bigcup \{ (v, a') \mid a' \in (\mathcal{R}_+ v) \} \]

\[\text{using } \text{sasp-op-to-strips-set-delete-effects-is}\]

\[\text{is-valid-operator-op} \text{ op}'-\text{is} \]

\[\text{by argo} \]

— TODO refactor.

\[
\{ \begin{align*}
\text{let } \text{?Add} &= \text{set } \{ (v, a) \in \text{set } \{ (v, a') \mid a' \in (\mathcal{R}_+ v) \} : a' \neq a \} \\
\text{let } \text{?Delete} &= \text{set } \{ (v, a) \in \text{set } \{ (v, a') \mid a' \in (\mathcal{R}_+ v) \} : a' \neq a \} \\
\text{have dom-add: dom } \text{map-of } \{ (v, \text{True}) \} (\text{add-effects-of op'}) = \text{?Add} \\
\text{unfolding } \text{dom-map-of-conv-image-fst set-map image-comp comp-apply} \\
\text{by simp} \\
\text{have dom-delete: dom } \text{map-of } \{ (v, \text{False}) \} (\text{delete-effects-of op'}) = \text{?Delete} \\
\text{unfolding } \text{dom-map-of-conv-image-fst set-map image-comp comp-apply} \\
\text{by auto} \\
\end{align*} \]

\[
\begin{align*}
\text{fix } v \ a \\
\text{assume } \text{v-a-in-dom-add: } (v, a) \in \text{dom } \text{map-of } \{ (v, \text{True}) \} (\text{add-effects-of op'}) \\
\text{have } (v, a) \notin \text{dom } \text{map-of } \{ (v, \text{False}) \} (\text{delete-effects-of op'}) \\
\text{proof } \text{(rule ccontr)} \\
\text{assume } \neg (v, a) \notin \text{dom } \text{map-of } \{ (v, \text{False}) \} (\text{delete-effects-of op'}) \\
\end{align*} \]

\[
\begin{align*}
\text{then have } (v, a) \in \text{?Delete } \text{and } (v, a) \in \text{?Add} \\
\text{using dom-add dom-delete v-a-in-dom-add} \\
\text{by argo}+ \\
\text{moreover have } \forall (v', a') \in \text{?Add}. \ v' \neq v \lor a = a' \\
\text{using is-valid-operator-sas-plus-then(6) is-valid-operator-op} \\
\text{calculation(2)} \\
\text{unfolding is-valid-operator-sas-plus-def} \\
\text{by fast} \\
\text{ultimately show } False \\
\text{by fast} \\
\text{qed} \\
\end{align*} \]

\[
\text{hence } \text{disjnt } \text{map-of } \{ (v, \text{True}) \} (\text{add-effects-of op'}) \}
\]

\[
\begin{align*}
\text{unfolding } \text{disjnt-def Int-def} \\
\text{using nb}_7 \\
\text{by simp} \\
\end{align*} \]

\[
\begin{align*}
\text{hence } \text{dom } \text{map-of } \{ (v, \text{True}) \} (\text{add-effects-of op'}) = \text{?Add} \\
\text{and } \text{dom } \text{map-of } \{ (v, \text{False}) \} (\text{delete-effects-of op'}) = \text{?Delete} \\
\end{align*} \]
and disjnt (dom (map-of (map (\(v\). (v, True))) (add-effects-of op'))))
(dom (map-of (map (\(v\). (v, False))) (delete-effects-of op'))))
using dom-add dom-delete
by blast
}

\textbf{note} \(nb_8 = \text{this}\)
— TODO refactor.
\{  
  let \(?\text{Add} = \text{set} (\text{effect-of } op)\)
  let \(?\text{Delete} = (\bigcup(v, a) \in \text{set} (\text{effect-of } op). \{ (v, a') \mid a', a' \in (R_+ \Psi v) \land a' \neq a \})\)
  — TODO slow.
  have \(\forall (v, a) \in ?\text{Add}. \text{map-of} (\text{effect-to-assignments } op') (v, a) = \text{Some True}\)
  and \(\forall (v, a) \in ?\text{Delete}. \text{map-of} (\text{effect-to-assignments } op') (v, a) = \text{Some False}\)
proof —
  \{  
  fix \(v\ a\)
  assume \((v, a) \in ?\text{Add}\)
  hence \(\text{map-of} (\text{effect-to-assignments } op') (v, a) = \text{Some True}\)
  unfolding effect-to-assignments-simp
  using \(nb_0\) map-of-defined-if-constructed-from-list-of-constant-assignments[of
    \(\text{map} (\lambda v. (v, True)) (\text{add-effects-of } op') \text{ True add-effects-of } op'\)
  by force
  \}
  moreover \{  
  fix \(v\ a\)
  assume \((v, a) \in ?\text{Delete}\)
  moreover have \((v, a) \in \text{dom} (\text{map-of} (\text{map} (\lambda v. (v, False))) (\text{delete-effects-of}\)
  \(op'))))\)
  using \(nb_8(2)\) calculation(1)
  by argo
  moreover have \((v, a) \notin \text{dom} (\text{map-of} (\text{map} (\lambda v. (v, True))) (\text{add-effects-of}
  \(op'))))\)
  using \(nb_8\)
  unfolding disjnt-def
  using calculation(1)
  by blast
  moreover have \(\text{map-of} (\text{effect-to-assignments } op') (v, a) = \text{map-of} (\text{map} (\lambda v. (v, False))) (\text{delete-effects-of}\)
  \(op'))))\)
  unfolding effect-to-assignments-simp map-of-append
  using map-add-dom-app-simps(3)[OF calculation(3)]
  by presburger
  — TODO slow.
  ultimately have \(\text{map-of} (\text{effect-to-assignments } op') (v, a) = \text{Some False}\)
  using map-of-defined-if-constructed-from-list-of-constant-assignments[
    \text{of map} (\lambda v. (v, False)) (\text{delete-effects-of } op') \text{ False delete-effects-of}
  \(op')\]
  \(nb_7\)
\[ 129 \]

by auto

ultimately show \( \forall (v, a) \in \text{?Add. map-of (effect-to-assignments op')} (v, a) = \text{Some True} \)
and \( \forall (v, a) \in \text{?Delete. map-of (effect-to-assignments op')} (v, a) = \text{Some False} \)

by blast+

qed

\}
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
\)
using state-to-strips-state-dom-is[OF assms(1), of
s +++ map-of (effect-of (\varphi_O^{-1} \Psi op'))]
by simp
}

note nb_{11} = this

{ fix v a
  assume (v, a) \in set (effect-of op)
moreover have v \in dom (map-of (effect-of op))
  unfolding dom-map-of-conv-image-fst
  using calculation
  by force
moreover have (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op')))) v = Some a
  unfolding map-add-dom-app-simps(1)[OF calculation(2)] nb_{10}
  by blast
moreover have (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op')))) v \neq None
  using calculation(3)
  by auto
moreover have (v, a) \in dom (\varphi_S \Psi (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op')))))
  using nb_{11} calculation(1, 4)
  by presburger
ultimately have (\varphi_S \Psi (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op'))))) (v, a) =
Some True
  using state-to-strips-state-range-is[OF assms(1)]
  by simp
}

note nb_{12} = this

{ fix v a'
  assume (v, a') \in dom (map-of (effect-to-assignments op'))
  and (v, a') \in \big( \bigcup (v, a) \in set (effect-of op).
  \{ (v, a') | a', a' \in (\mathcal{R}_+ \Psi v) \land a' \neq a \} \big)
moreover have v \in dom (map-of (effect-of op))
  unfolding dom-map-of-conv-image-fst
  using calculation(2)
  by force
moreover have v \in set ?vs
  using calculation(3) is-valid-operator-sas-plus-then(3) is-valid-operator-op
  unfolding dom-map-of-conv-image-fst is-valid-operator-sas-plus-def
  by fastforce
moreover obtain a where (v, a) \in set (effect-of op)
  and a' \in \mathcal{R}_+ \Psi v
  and a' \neq a
  using calculation(2)
  by blast
moreover have (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op')))) v = Some a
  unfolding map-add-dom-app-simps(1)[OF calculation(3)] nb_{10}
  by blast
moreover have (s +++ map-of (effect-of (\varphi_O^{-1} \Psi op')))) v \neq None

131
using calculation(8)
by auto
— TODO slow.
moreover have \((v, a') \in \text{dom } (\varphi S \Psi (s ++ \text{map-of } (\varphi O^{-1} \Psi \text{op}')))\)
using state-to-strips-state-dom-is[OF assms(1), of \(s ++ \text{map-of } (\varphi O^{-1} \Psi \text{op}'))] calculation(4, 6, 9)
by simp
— TODO slow.
ultimately have \((\varphi S \Psi (s ++ \text{map-of } (\varphi O^{-1} \Psi \text{op}'))) (v, a') = \text{Some False}\)
using state-to-strips-state-range-is[OF assms(1), of v a']
calculation(4, 6, 9)
by simp

moreover have \((v, a) \in \text{dom } ?t\)
and \((v, a) \notin \text{dom } \text{(map-of } (\text{effect-to-assignments op}'))\)
moreover have \((v, a) \in \text{dom } ?s'\)
using calculation(1, 2)
unfolding dom-map-add
by blast
moreover have \(?t (v, a) = ?s' (v, a)\)
unfolding map-add-dom-app-simps[OF calculation(2)]
ultimately have \(?t (v, a) = \text{Some (the } (s v) = a)\)
using state-to-strips-state-range-is[OF assms(1)]
by presburger
}

note nb13 = this
{
fix v a
assume \((v, a) \in \text{dom } ?t\)
and \((v, a) \notin \text{dom } \text{(map-of } (\text{effect-to-assignments op}'))\)
moreover have \((v, a) \in \text{dom } ?s'\)
using calculation(1, 2)
unfolding dom-map-add
by blast
moreover have \((v, a) \in \bigcup v \in \{ v \mid v \in \text{set } ((\Psi)_{v+}) \land s v \neq \text{None } \}.\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi \ v \}\)\)
using state-to-strips-state-dom-is[OF assms(1)] calculation(3)
by presburger
moreover have \(v \in \text{set } ((\Psi)_{v+})\) and \(s v \neq \text{None }\) and \(a \in \mathcal{R}_+ \Psi v\)
using calculation(4)
by blast+
— NOTE Hasn’t this been proved before?
moreover { 
have \(\text{dom } \text{(map-of } (\text{effect-to-assignments op}')) = \bigcup (v, a) \in \text{set } \text{(effect-of op)}.\{ (v, a) \}\)
\cup \bigcup (v, a) \in \text{set } \text{(effect-of op)}.\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \land a' \neq a \}\)

132
unfolding \(nb_2\)
by blast
also have \(\ldots = (\bigcup (v, a) \in \text{set (effect-of \(op\)).} \{ (v, a) \}
\cup \{ (v, a') | a', a' \in (R_+ \Psi v) \land a' \neq a \})\)
by blast
finally have \(\text{dom (map-of (effect-to-assignments \(op')\))} =
(\bigcup (v, a) \in \text{set (effect-of \(op\)).} \{ (v, a) \}
\cup \{ (v, a | a, a \in R_+ \Psi v) \})\)
by auto
then have \((v, a) \notin (\bigcup (v, a) \in \text{set (effect-of \(op\)).} \{ (v, a) \})\)
using \(?v-a-not-in\)
by blast
moreover have \(v \notin \text{dom (map-of (effect-of \(op\)))}\)
using \(?v-a-not-in\)
by fastforce
moreover have \((s++ \text{map-of (effect-of (\(\varphi_O^{-1} \Psi \(op')\)))) v = s v\)
unfolding \(nb_1 \map-add-dom-app-simps{3}[\text{OF calculation}(9)]\)
by simp
— TODO slow.
moreover have \((v, a) \in \text{dom ?t}'\)
using \(?v-a-not-in\)
by fastforce
ultimately have \(?t' (v, a) = \text{Some (the (s v) = a)}\)
using \(?v-a-not-in\)
by presburger
— TODO refactor.

have \(nb_{16} \colon \text{dom ?t} = \bigcup v \in \{ v | v, v \in \text{set ((\(\Psi)_{V+}\))} \land s v \neq \text{None} \} \}
\cup \{ (v, a) | a, a \in (R_+ \Psi v) \}\)
\cup \text{set (effect-of \(op\))}
\cup (\bigcup (v, a) \in \text{set (effect-of \(op\))}.
\{ (v, a') | a', a' \in (R_+ \Psi v) \land a' \neq a \})\)
unfolding \(\text{dom-map-add nb}_2\)
using \(\text{state-to-strips-state-dom-is[\text{OF assms}(1), of s]}\)
by auto

\begin{verbatim}
{ fix \(v a\)
  assume \((v, a) \in \text{dom ?t}\)
  then consider \((A) (v, a) \in \text{dom (}\varphi_S \Psi s)\)
  \( | (B) (v, a) \in \text{dom (map-of (effect-to-assignments \(op')\))}\)
  by fast
  hence \((v, a) \in \text{dom ?t}'\)
  proof (cases)
  case A

\end{verbatim}

133
then have $v \in \text{set}((\Psi)V^+)$ and $s \neq \text{None}$ and $a \in \mathcal{R}_+ \Psi v$

unfolding $\text{state-to-strips-state-dom-element-iff}[\text{OF assms}(1)]$
by blast+

thm $\text{map-add-None state-to-strips-state-dom-element-iff}[\text{OF assms}(1)]$
moreover have $(s ++ \text{map-of}(\text{effect-of}(\varphi_{O-1} \Psi \text{op}'))) v \neq \text{None}$
using calculation(2)
by simp
ultimately show $\text{thesis}$
unfolding $\text{state-to-strips-state-dom-element-iff}[\text{OF assms}(1)]$
by blast

next
case $B$
then have $(v, a) \in$
set $(\text{effect-of op})$
$\cup (\bigcup (v, a) \in\text{set}(\text{effect-of op}). \{ (v, a') | a' \in \mathcal{R}_+ \Psi v \land a' \neq a \})$
unfolding $\text{nb}_2$
by blast
then consider $(B_1) (v, a) \in\text{set}(\text{effect-of op})$
$|(B_2) (v, a) \in (\bigcup (v, a) \in\text{set}(\text{effect-of op}).$
$\{ (v, a') | a' \in \mathcal{R}_+ \Psi v \land a' \neq a \})$
by blast
thm $\text{nb}_{12} \text{nb}_{13} \text{nb}_2$
thus $\text{thesis}$
proof (cases)

case $B_1$
then show $\text{thesis}$
using $\text{nb}_{12}$
by fast
next
case $B_2$
then show $\text{thesis}$
using $\text{nb}_{13} B$
by blast
qed
qed

moreover {
let $?u = s ++ \text{map-of}(\text{effect-of}(\varphi_{O-1} \Psi \text{op}'))$
fix $v$ $a$
assume $v\text{-a-in-dom-}t'; (v, a) \in\text{dom} \text{ ?t'}$

thm $\text{nb}_5$
then have $v\text{-in-vs}: v \in\text{set}((\Psi)V^+)$
and $u\text{-of-v-is-not-None}: ?a v \neq \text{None}$
and $a\text{-in-range-of-v}: a \in \mathcal{R}_+ \Psi v$
using $\text{state-to-strips-state-dom-element-iff}[\text{OF assms}(1)]$
$v\text{-a-in-dom-}t'$
by meson+
}
assume $(v, a) \notin\text{dom} \text{ ?t}$
then have contradiction: \((v, a) \notin \)
\[
\left( \bigcup v \in \{ v \mid v. v \in \text{set} ((\Psi)_{v^+}) \wedge s v \neq \text{None}\}. \ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \right)
\]
\[
\bigcup \text{set (effect-of op)}
\]
\[
\bigcup (\bigcup (v, a) \in \text{set (effect-of op)}. \ \{(v, a') \mid a'. a' \in \mathcal{R}_+ \Psi v \land a' \neq a\})
\]
unfolding nb16
by fast
hence False
proof (cases map-of (effect-of \((\varphi_O^{-1} \Psi \ op')\)) \(v = \text{None}\))
case True
then have \(s v \neq \text{None}\)
using \(u-of-v-is-not-None\)
by simp
then have \((v, a) \in \bigcup \{ v \mid v. v \in \text{set} ((\Psi)_{v^+}) \wedge s v \neq \text{None}\}. \ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \}\)
using \(v-in-vs a-in-range-of-v\)
by blast
thus \(?thesis\)
using \(contradiction\)
by blast
next
case False
then have \(v \in \text{dom (map-of (effect-of op))}\)
using \(u-of-v-is-not-None\ nb1\)
by blast
then obtain \(a'\) where \(map-of-effect-of-op-v-is: \text{map-of (effect-of op) v = Some a'}\)
by blast
then have \(v-a'-in: (v, a') \in \text{set (effect-of op)}\)
using \(map-of-SomeD\)
by fast
then show \(?thesis\)
proof (cases \(a = a'\))
case True
then have \((v, a) \in \text{set (effect-of op)}\)
using \(v-a'-in\)
by blast
then show \(?thesis\)
using \(contradiction\)
by blast
next
case False
then have \((v, a) \in \bigcup \{(v, a) \in \text{set (effect-of op)}. \ \{(v, a') \mid a'. a' \in \mathcal{R}_+ \Psi v \land a' \neq a\}\}\)
using \(v-a'-in\) calculation \(a-in-range-of-v\)
by blast
thus \(?thesis\)
using \(contradiction\)
by fast

135
hence \((v, a) \in \text{dom } ?t\)
by argo

moreover have \(\text{dom } ?t \subseteq \text{dom } ?t'\) and \(\text{dom } ?t' \subseteq \text{dom } ?t\)
subgoal
using calculation(1) subrelI[of dom ?t dom ?t']
by fast
subgoal
using calculation(2) subrelI[of dom ?t' dom ?t]
by argo
done
ultimately have \(\text{dom } ?t = \text{dom } ?t'\)
by force

note \(nb_{17} = \text{this}\)
{
fix \(v, a\)
assume \(v \text{-in-dom-} ?t: (v, a) \in \text{dom } ?t\)
hence \(?t (v, a) = ?t' (v, a)\)
proof (cases \((v, a) \in \text{dom } (\text{map-of } \text{effect-to-assignments } \text{op'})\))
case \(\text{True}\)
— TODO slow.
— NOTE Split on the (disjunct) domain variable sets of \(\text{map-of } \text{effect-to-assignments } \text{op'}\).
then consider \((A1) (v, a) \in \text{set } \text{effect-of } \text{op})\)
| \((A2) (v, a) \in (\bigcup (v, a) \in \text{set } \text{effect-of } \text{op}).\)
\{ \((v, a') | a', a' \in (\mathcal{R}_+ \Psi v) \land a' \neq a \}\}
using \(nb_2\)
by fastforce
then show \(?\text{thesis}\)
proof (cases)
case \(A1\)
then have \(?t (v, a) = \text{Some True}\)
unfolding map-add-dom-app-simps(1)[OF True]
using \(nb_9(1)\)
by fast
moreover have \(?t' (v, a) = \text{Some True}\)
using \(nb_{12}[OF A1]\).
ultimately show \(?\text{thesis}..\)
next
case \(A2\)
then have \(?t (v, a) = \text{Some False}\)
unfolding map-add-dom-app-simps(1)[OF True]
using \(nb_9(2)\)
by blast
moreover have \(?t' (v, a) = \text{Some False}\)
using \(nb_{13}[OF True A2]\).
ultimately show \( ?thesis \)...

qed

next

case False

moreover have \( ?t (v, a) = \text{Some} (\text{the} (s v) = a) \)

using \( \text{nb14} \{ \text{OF} v-a-in-dom-t \text{ False} \} \).

moreover have \( ?t' (v, a) = \text{Some} (\text{the} (s v) = a) \)

using \( \text{nb15} \{ \text{OF} v-a-in-dom-t \text{ False} \} \).

ultimately show \( ?thesis \)

by \text{argo}

qed

} 

note \( \text{nb18} = \text{this} \)

moreover \{

fix \( v, a \)

assume \( (v, a) \in \text{dom} \ ?t' \)

hence \( ?t (v, a) = ?t' (v, a) \)

using \( \text{nb17 \ nb18} \)

by \text{presburger}

\}

— TODO slow.

ultimately have \( ?t \subseteq_m ?t' \) and \( ?t' \subseteq_m ?t \)

unfolding \text{map-le-def}

by \text{fastforce+}

thus \( ?thesis \)

using \text{map-le-antisym} \{ \text{af} ?t \ ?t' \}

by \text{fast}

qed

— NOTE This is the essential step in the SAS+/STRIPS equivalence theorem. We show that executing a given parallel STRIPS operator \( \text{ops}' \) on the corresponding STRIPS state \( s' = \varphi_S \Psi s \) yields the same state as executing the transformed SAS+ parallel operator \( \text{ops} = [\varphi_O^{-1} (\varphi \Psi) \text{ op'} \ \text{op'} \leftarrow \text{ops}' ] \) on the original SAS+ state \( s \) and the transforming the resultant SAS+ state to its corresponding STRIPS state.

\textbf{lemma} \text{sas-plus-equivalent-to-strips-i-a-XII:}

\text{assumes} \ is-valid-problem-sas-plus \( \Psi \)

\and \( \forall \text{ op'} \in \text{set \ ops}', \text{ op'} \in \text{set \ ((\varphi \Psi)O) } \)

\text{shows} \ execute-parallel-operator \( (\varphi_S \Psi s) \text{ ops'} \)

\ = \( \varphi_S \Psi \text{ (execute-parallel-operator-sas-plus \ s \ [\varphi_O^{-1} \Psi \text{ op'} \ \text{op'} \leftarrow \text{ops}' ] )} \)

using \ assms

\text{proof} \ (\text{induction \ ops'} \ arbitrary: s)

\text{case} \ Nil

\text{then show} \ ?\text{case}

\text{unfolding} \ execute-parallel-operator-def execute-parallel-operator-sas-plus-def

by \text{simp}

\text{next}

\text{case} \ (\text{Cons op' \ ops'})

\text{let} \ ?\Pi \text{ = } \varphi \Psi

137
let \( ?t' = (\varphi_S \Psi \ s) \mapplus \text{map-of} \ (\text{effect-to-assignments} \ \text{op}') \)
and \( ?t = s \mapplus \text{map-of} \ (\varphi_O^{-1} \Psi \ \text{op}') \)

have \( nb_1: \ ?t' = \varphi_S \Psi \ ?t \)
using \text{sas-plus-equivalent-to-strips-i-a-XI}[\text{OF assms(1)}] \text{Cons.prems(2)}
by \text{force} \{ 
  have \( \forall \text{op}' \in \text{set \ ops'}. \ \text{op}' \in \text{set} \ ((\varphi \ \Psi)_O) \)
using \text{Cons.prems(2)}
by \text{simp} 
then have \( \text{execute-parallel-operator} \ (\varphi_S \Psi \ ?t) \ \text{ops'} = \varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ ?t \ [\varphi_O^{-1} \Psi \ x, \ x \leftarrow \text{ops'}]) \)
using \text{Cons.IH}[\text{OF Cons.prems(1)}, \ \text{of} \ ?t]
by \text{fastforce} 

hence \( \text{execute-parallel-operator} \ ?t' \ \text{ops'} = \varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ ?t \ [\varphi_O^{-1} \Psi \ x, \ x \leftarrow \text{ops'}]) \)
using \( nb_1 \)
by \text{argo} 
\}
thus \( ?\text{case} \)
by \text{simp} 

qed 

lemma \text{sas-plus-equivalent-to-strips-i-a-XIII}: 
assumes \( \text{is-valid-problem-sas-plus} \ \Psi \)
and \( \forall \text{op}' \in \text{set \ ops'}. \ \text{op}' \in \text{set} \ ((\varphi \ \Psi)_O) \)
and \( (\varphi_S \Psi \ G) \subseteq_m \text{execute-parallel-plan} \ (\text{execute-parallel-operator} \ (\varphi_S \Psi \ I) \ \text{ops'}) \pi \)

shows \( (\varphi_S \Psi \ G) \subseteq_m \text{execute-parallel-plan} \ (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ I \ [\varphi_O^{-1} \Psi \ \text{op}', \ \text{op}' \leftarrow \text{ops'}])) \pi \)

proof --
let \( ?I' = (\varphi_S \Psi \ I) \)
and \( ?G' = \varphi_S \Psi \ G \)
and \( ?\text{ops} = [\varphi_O^{-1} \Psi \ \text{op}', \ \text{op}' \leftarrow \text{ops'}] \)
and \( ?\Pi = \varphi \Psi \)

let \( ?I = \text{execute-parallel-operator-sas-plus} \ I \ ?\text{ops} \)
{ 
fix \ v \ a 
assume \( (v, \ a) \in \text{dom} \ ?G' \)
then have \( ?G' (v, \ a) = \text{execute-parallel-plan} \ (\text{execute-parallel-operator} \ ?I' \ \text{ops'}) \pi (v, \ a) \)
using \text{assms(3)}
unfolding \text{map-le-def}
by \text{auto} 

hence \( ?G' (v, \ a) = \text{execute-parallel-plan} \ (\varphi_S \Psi \ ?I) \pi (v, \ a) \)
using \text{sas-plus-equivalent-to-strips-i-a-XII}[\text{OF assms(1, 2)}]
by \text{simp} 
\}
thus \( ?\text{thesis} \)
unfolding \text{map-le-def}
— NOTE This is a more abstract formulation of the proposition in `sas-plus-equivalent-to-strips-i` which is better suited for induction proofs. We essentially claim that given a plan the execution in STRIPS semantics of which solves the problem of reaching a transformed goal state \( \varphi S \Psi G \) from a transformed initial state \( \varphi S \Psi I \)—such as the goal and initial state of an induced STRIPS problem for a SAS+ problem—is equivalent to an execution in SAS+ semantics of the transformed plan \( \varphi P^{-1} (\varphi \Psi) \pi \) w.r.t to the original initial state \( I \) and original goal state \( G \).

**Lemma** `sas-plus-equivalent-to-strips-i-a`:

assumes is-valid-problem-sas-plus \( \Psi \)

and \( \text{dom } I \subseteq \text{set } (\varphi \Psi) \cup \)

and \( \forall v \in \text{dom } I. \text{the } (I v) \in \mathcal{R}_+ \Psi v \)

and \( \text{dom } G \subseteq \text{set } (\varphi \Psi) \cup \)

and \( \forall v \in \text{dom } G. \text{the } (G v) \in \mathcal{R}_+ \Psi v \)

and \( \forall \text{ops'} \in \pi. \forall \text{op'} \in \text{set } \text{ops'}. \text{op'} \in \text{set } ((\varphi \Psi)_0) \)

and \( \varphi S \Psi G \subseteq_m \text{execute-parallel-plan } (\varphi S \Psi I) \pi \)

shows \( G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\varphi P^{-1} \Psi \pi) \)

**Proof**

let \( ?\psi = \text{variables-of } \Psi \)

and \( \Psi = \varphi P^{-1} \Psi \pi \)

show \( ?\thesis \)

using \( \text{assms} \)

proof (induction \( \pi \) arbitrary: \( I \))

case \( \text{Nil} \)

then have \( \varphi S \Psi G \subseteq_m (\varphi S \Psi I) \)

using \( \text{state-to-strips-state-map-le-iff}[\text{OF } \text{assms}(1, 4, 5)] \)

by \( \text{blast} \)

thus \( ?\case \)


`strips-parallel-plan-to-sas-plus-parallel-plan-def`

by \( \text{fastforce} \)

next

case \( (\text{Cons } \text{ops'} \pi) \)

let \( ?D = \text{range-of } \Psi \)

and \( ?\Pi = \varphi \Psi \)

and \( ?I' = \varphi S \Psi I \)

and \( ?G' = \varphi S \Psi G \)

let \( ?\ops = [\varphi O^{-1} \Psi \text{op'}. \text{op'} \leftarrow \text{ops}'] \)

let \( ?J = \text{execute-parallel-operator-sas-plus } I ?\ops \)

and \( ?J' = \text{execute-parallel-operator } ?I' \text{ops'} \)

have \( \text{nb'}: \text{set } \text{ops'} \subseteq \text{set } ((?\Pi)_0) \)

using \( \text{Cons.prems(6)} \)

unfolding `STRIPS-Semantics.is-parallel-solution-for-problem-def list-all-iff` `ListMem-iff`

by \( \text{fastforce} \)


\{ 
  \textbf{fix \ op} \\
  \textbf{assume \ op} \in \textit{set \ \textit{\{ops\}}} \\
  \textbf{moreover obtain \ op' where \ op' \in \ textit{set \ ops'} \ and \ op = \varphi_{O}^{-1} \ \Psi \ \op'} \\
  \textbf{using \ calculation} \\
  \textbf{by \ auto} \\
  \textbf{moreover have \ op' \in \ ((\oplus_{O})_{\Theta})} \\
  \textbf{using \ nb_1 \ calculation(2)} \\
  \textbf{by \ blast} \\
  \textbf{moreover obtain \ op'' \ where \ op'' \in \ ((\oplus_{O})_{\Theta+}) \ and \ op' = \varphi_{O} \ \Psi \ \op''} \\
  \textbf{using \ calculation(4)} \\
  \textbf{by \ auto} \\
  \textbf{moreover have \ op = \ op''} \\
  \textbf{using \ \textit{sas-plus-operator-inverse-is[OF \ \textit{assms}(1) \ calculation(5)] \ calculation(3, \ 6)}} \\
  \textbf{by \ presburger} \\
  \textbf{ultimately have \ op \in \ ((\oplus_{O})_{\Theta+})} \land (\exists \ op' \in \ \textit{set \ ops'}. \ op' = \varphi_{O} \ \Psi \ \op) \\
  \textbf{by \ blast} 
\} \textbf{note nb_2 = this} 
\} 
\{ 
  \textbf{fix \ op, v, a} \\
  \textbf{assume \ op} \in \textit{set \ ((\oplus_{O})_{\Theta+})} \land (v, a) \in \textit{set \ (effect-of \ op)} \\
  \textbf{moreover have \ op} \in \textit{set \ ((\oplus_{O})_{\Theta+})} \\
  \textbf{using \ nb_2 \ calculation(1)} \\
  \textbf{by \ blast} \\
  \textbf{moreover have \ is-valid-operator-sas-plus \ \Psi \ \op} \\
  \textbf{using \ is-valid-problem-sas-plus-then(2) \ Cons.prems(1) \ calculation(3)} \\
  \textbf{by \ blast} \\
  \textbf{ultimately have \ v} \in \textit{set \ ((\oplus_{O})_{\Theta+})} \\
  \textbf{using \ is-valid-operator-sas-plus-then(3)} \\
  \textbf{by \ fastforce} 
\} \textbf{note nb_3 = this} 
\} 
\{ 
  \textbf{fix \ op} \\
  \textbf{assume \ op} \in \textit{set \ \textit{\{ops\}}} \\
  \textbf{then have \ op} \in \textit{set \ ((\oplus_{O})_{\Theta+})} \\
  \textbf{using \ nb_2} \\
  \textbf{by \ blast} \\
  \textbf{then have \ is-valid-operator-sas-plus \ \Psi \ \op} \\
  \textbf{using \ is-valid-problem-sas-plus-then(2) \ Cons.prems(1)} \\
  \textbf{by \ blast} \\
  \textbf{hence} \forall (v, a) \in \textit{set \ (effect-of \ op)}. \ v \in \textit{set \ ((\oplus_{O})_{\Theta+})} \\
  \land a \in R_{\Theta} \ \Psi \ v \\
  \textbf{using \ is-valid-operator-sas-plus-then(3, 4)} \\
  \textbf{by \ fast} 
\} \textbf{note nb_4 = this} 
\} 
\textbf{show \ ?case} \\
\textbf{proof} \ (\textit{cases \ STRIPS-Semantics.are-all-operators-applicable \ ?I' \ ops' \\
\land \ STRIPS-Semantics.are-all-operator-effects-consistent \ ops'})
case True
{
  
  have dom I ⊆ set ((Ψ)V+)
    using Cons.prems(2)
    by blast
  hence (ϕ⁻¹Ψ ?I) = I
    using strips-state-to-state-inverse-is[OF Cons.prems(1) - Cons.prems(3)]
    by argo
  }
then have are-all-operators-applicable-in I ?ops
    ∧ are-all-operator-effects-consistent ?ops
    using sas-plus-equivalent-to-strips-i-a-IV[OF assms nb_1, of I] True
    by simp
moreover have (ϕ⁻¹Ψ (ops' # π)) = ?ops # (ϕ⁻¹Ψ π)
unfolding SAS-Plus-STRIPS strips-parallel-plan-to-sas-plus-parallel-plan-def
  strips-op-to-sasp-def
by simp
ultimately have execute-parallel-plan-sas-plus I (ϕ⁻¹Ψ (ops' # π))
  = execute-parallel-plan-sas-plus ?J (ϕ⁻¹Ψ π)
by force
}

note nb_5 = this
— Show the goal using the IH.
{
  have dom-J-subset-eq-vs: dom ?J ⊆ set ((Ψ)V+)
    using sas-plus-equivalent-to-strips-i-a-IX[OF Cons.prems(2)] nb_2 nb_4
    by blast
  moreover {
    have set ((Ψ)V+) ⊆ dom (range-of Ψ)
      using is-valid-problem-sas-plus-then[OF assms nb_1]
      by fastforce
    moreover have ∀ v ∈ dom I. the (I v) ∈ set (the (range-of Ψ v))
      using Cons.prems(2, 3) assms(1) set-the-range-of-is-range-of-sas-plus-if
      by force
    moreover have ∀ op ∈ set ?ops, ∀ (v, a) ∈ set (effect-of op).
      v ∈ set ((Ψ)V+) ∧ a ∈ set (the (?D v))
      using set-the-range-of-is-range-of-sas-plus-if assms nb_4
      by fastforce
    moreover have v in dom-J-range: ∀ v ∈ dom ?J. the (?J v) ∈ set (the (?D v))
      using sas-plus-equivalent-to-strips-i-a-X[OF I set ((Ψ)V+) ?D ?ops, OF Cons.prems(2)] calculation(1, 2, 3)
      by fastforce
  }
  fix v

141
assume \( v \in \text{dom} \ ?J \)
moreover have \( v \in \text{set} \ ((\Psi)v_+) \)
    using \( nb_2 \) calculation \( \text{dom-J-subset-eq-vs} \)
    by blast
moreover have \( \text{set} \ (\text{the range of } \Psi \ v) = R_+ \Psi \ v \)
    using \( \text{set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]} \)
    calculation(2)
    by presburger
ultimately have \( (\Psi \ v) \in R_+ \Psi \ v \)
    using \( \text{sas-plus-equivalent-to-strips-i-a-XIII[OF Cons.prems(1)]} \)
    \( nb_3 \) \( v \in \text{dom-J-range} \)
    by blast

ultimately have \( \forall v \in \text{dom} \ ?J. \ \text{the} \ (\Psi \ v) \in R_+ \Psi \ v \)
by fast

moreover have \( \forall \text{ops'} \in \text{set } \pi. \ \forall \text{op'} \in \text{set } \text{ops'}. \ \text{op'} \in \text{set} \ ((\varphi \ \Psi)\circ) \)
using \( \text{Cons.prems(6)} \)
by simp
moreover {
    have \( \Psi \subseteq_m \text{execute-parallel-plan} \ ?J' \pi \)
        using \( \text{Cons.prems(7)} \) True
        by auto
    hence \( (\varphi_S \Psi \ G) \subseteq_m \text{execute-parallel-plan} \ (\varphi_S \Psi \ ?J) \pi \)
        using \( \text{sas-plus-equivalent-to-strips-i-a-XIII[OF Cons.prems(1)]} \)
        \( nb_1 \)
        by blast
}
ultimately have \( G \subseteq_m \text{execute-parallel-plan-sas-plus I} \ (\varphi_{P^{-1}} \Psi \ (\text{ops'} \ # \ \pi)) \)
    using \( \text{Cons.IH[of \ ?J, OF Cons.prems(1) - - Cons.prems(4, 5)]} \)
    Cons.prems(6) \( nb_5 \)
    by presburger
}
thus \( \text{thesis} \).
next
case False
then have \( \Psi' \subseteq_m \ ?I' \)
    using \( \text{Cons.prems(7)} \)
    by force
moreover {
    have \( \text{dom } I \subseteq \text{set } \text{?vs} \)
        using \( \text{Cons.prems(2)} \)
        by simp
    hence \( \neg(\text{are-all-operators-applicable-in } I \ ?\text{ops} \land \text{are-all-operator-effects-consistent } \text{?ops}) \)
        using \( \text{sas-plus-equivalent-to-strips-i-a-VIII[OF Cons.prems(1)]} \)
        \( \text{Cons.prems(3)} \) \( nb_1 \)
        False
    by force
}
moreover \{
  \text{have } (\varphi_p^{-1} \Psi (\text{ops'} \# \pi)) = ?\text{ops' } (\varphi_p^{-1} \Psi \pi)
}\text{ unfolding } \text{SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def}
  \text{strips-parallel-plan-to-sas-plus-parallel-plan-def}
  \text{SAS-Plus-STRIPS.strips-op-to-sas-p-def}
  \text{strips-op-to-sasp-def}
  \text{by simp}
  \text{hence } G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\text{?ops' } (\varphi_p^{-1} \Psi \pi))
  \text{ using calculation(2)}
  \text{by force}
\}

\text{ultimately show } ?\text{thesis}
\text{ using state-to-strips-state-map-iff}[\text{OF Cons.prems(1, 4, 5)}]
\text{ unfolding } \text{SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def}
  \text{strips-parallel-plan-to-sas-plus-parallel-plan-def}
  \text{SAS-Plus-STRIPS.strips-op-to-sas-p-def}
  \text{strips-op-to-sasp-def}
  \text{by force}

\text{qed}
\text{qed}
\text{qed}

— NOTE Show that a solution for the induced STRIPS problem for the given valid SAS+ problem, corresponds to a solution for the given SAS+ problem.

Note that in the context of the SAS+ problem solving pipeline, we

1. convert the given valid SAS+ Ψ problem to the corresponding STRIPS problem Π (this is implicitly also valid by lemma is-valid-problem-sas-plus-then-strips-transformation-too);
2. get a solution π—if it exists—for the induced STRIPS problem by executing SATPlan; and finally,
3. convert π back to a solution ψ for the SAS+ problem.

lemma sas-plus-equivalent-to-strips-i:
assumes is-valid-problem-sas-plus Ψ
  and STRIPS-Semantics.is-parallel-solution-for-problem
    (\varphi \Psi) \pi
shows goal-of Ψ \subseteq_m execute-parallel-plan-sas-plus
    (sas-plus-problem.initial-of Ψ) (\varphi_p^{-1} \Psi \pi)
proof –
let ?\text{vs} = variables-of Ψ
  and ?I = initial-of Ψ
  and ?G = goal-of Ψ
let ?\text{II} = \varphi \Psi
let ?G' = strips-problem.goal-of ?\text{II}
  and ?I' = strips-problem.initial-of ?\text{II}
let ?\psi = \varphi_p^{-1} \Psi \pi
have dom ?I \subseteq set ?\text{vs}
using is-valid-problem-sas-plus-then(3) \text{assms}(1)
by auto
moreover have \( \forall v \in \text{dom } ?I. \text{ the } (?I v) \in \mathcal{R}_+ \Psi v \)
using is-valid-problem-sas-plus-then(4) \text{assms}(1)\text{ calculation}
by auto
moreover have dom ?G \subseteq \text{set } ?v \text{ and } \forall v \in \text{dom } ?G \text{ the } (?G v) \in \mathcal{R}_+ \Psi v
using is-valid-problem-sas-plus-then(5, 6) \text{assms}(1)
by blast+
moreover have \( \forall \text{ops}' \in \text{set } \pi. \forall \text{op}' \in \text{set ops}'. \text{op}' \in \text{set ((?\Pi)\sigma)} \)
using is-parallel-solution-for-problem-operator-set[\text{OF} \text{assms}(2)]
by simp
moreover have \( ?G' \subseteq_m \text{execute-parallel-plan } ?I' \pi \)
using assms(2)
unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def.,
moreover have \( ?G' = \varphi_S \Psi ?G \text{ and } ?I' = \varphi_S \Psi ?I \)
by simp+
ultimately have \( (\varphi_S \Psi ?G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi ?I) \pi \)
by simp
}
ultimately show \( ?\text{thesis} \)
using sas-plus-equivalent-to-strips-i-a[\text{OF} \text{assms}(1)]
by simp
qed

— NOTE Show that the operators for a given solution \( \pi \) to the induced STRIPS problem for a given SAS+ problem correspond to operators of the SAS+ problem.

lemma sas-plus-equivalent-to-strips-ii:
assumes is-valid-problem-sas-plus \( \Psi \)
and STRIPS-Semantics.is-parallel-solution-for-problem \( (\varphi \Psi) \pi \)
shows list-all \( (\text{list-all } (\lambda \text{op. ListMem op } \text{operators-of } \Psi) ) ) (\varphi_{p^-1} \Psi \pi) \)
proof –
let \( ?\Pi = \varphi \Psi \)
let \( ?\text{ops} = \text{operators-of } \Psi \)
and \( ?\psi = \varphi_{p^-1} \Psi \pi \)
have is-valid-problem-strips ?\Pi
using is-valid-problem-sas-plus-then-strips-transformation-too[\text{OF} \text{assms}(1)]
by simp
have \text{nb1: } \forall \text{op}' \in \text{set } ((?\Pi)\sigma). (\exists \text{op } \in \text{set } ?\text{ops}. \text{op}' = (\varphi_O \Psi \text{op}))
by auto
{\text{fix } \text{ops}' \text{op}' \text{op}}
assume \( \text{ops}' \in \text{set } \pi \text{ and } \text{op}' \in \text{set } \text{ops}' \)
then have \( \text{op}' \in \text{set } (\text{strips-problem.operators-of } ?\Pi) \)
using is-parallel-solution-for-problem-operator-set[\text{OF} \text{assms}(2)]
by simp
then obtain \( \text{op where } \text{op } \in \text{set } ((\Psi)\sigma+) \text{ and } \text{op}' = (\varphi_O \Psi \text{op}) \)
by auto
then have \( (\varphi_{O^-1} \Psi \text{op}') \in \text{set } ((\Psi)\sigma+) \)
using sas-plus-operator-inverse-is[OF assms(1)]
by presburger
}
 thus ?thesis
 unfolding list-all-iff ListMem-iff
 strips-parallel-plan-to-sas-plus-parallel-plan-def
 SAS-Plus-STRIPS.strips-op-to-sasp-def
 strips-op-to-sasp-def
 by auto
qed

We now show that for a parallel solution \( \pi \) of \( \Pi \) the SAS+ plan \( \psi \equiv \varphi_{P^{-1}} \Psi \pi \) yielded by the STRIPS to SAS+ plan transformation is a solution for \( \Psi \). The proof uses the definition of parallel STRIPS solutions and shows that the execution of \( \psi \) on the initial state of the SAS+ problem yields a state satisfying the problem’s goal state, i.e.

\[
G \subseteq m \text{ execute-parallel-plan-sas-plus } I \psi
\]

and by showing that all operators in all parallel operators of \( \psi \) are operators of the problem.

**theorem**

**sas-plus-equivalent-to-strips:**

**assumes** is-valid-problem-sas-plus \( \Psi \)

and STRIPS-Semantics.is-parallel-solution-for-problem \( (\varphi \Psi) \pi \)

**shows** is-parallel-solution-for-problem \( \Psi (\varphi_{P^{-1}} \Psi \pi) \)

**proof**

let \( \varphi = \text{initial-of } \Psi \)
and \( \varphi = \text{goal-of } \Psi \)
and \( \varphi = \text{operators-of } \Psi \)
and \( \varphi = \varphi_{P^{-1}} \Psi \pi \)

shows \( \text{thesis} \)

**unfolding** is-parallel-solution-for-problem-def Let-def

**proof** (rule conjI)

show \( \varphi \subseteq m \text{ execute-parallel-plan-sas-plus } I \psi \)
using sas-plus-equivalent-to-strips-i[OF assms].

next

show list-all (list-all (\lambda op. ListMem op \varphi) op) \psi
using sas-plus-equivalent-to-strips-ii[OF assms].

qed

**private lemma**

**strips-equivalent-to-sas-plus-i-a-I:**

**assumes** is-valid-problem-sas-plus \( \Psi \)

and \( \forall op \in \text{set ops. op} \in \text{set } (\varphi_{O+}) \)
and \( \forall op \in \text{set } [\varphi_{O} \Psi. op \leftarrow \text{ops}] \)

obtains \( \text{op where } op \in \text{set ops} \)
and \( \text{op} = \varphi_{O} \Psi \text{ op} \)
proof
  let \( ?\Pi = \varphi \Psi \)
  let \( ?\text{ops} = \text{operators-of} \Psi \)
  obtain \( \text{op} \) where \( \text{op} \in \text{set ops} \) and \( \text{op}' = \varphi_O \Psi \text{ op} \)
  using \( \text{assms}(3) \)
  by auto
thus \( ?\text{thesis} \)
using that
by blast
qed

private corollary strips-equivalent-to-sas-plus-i-a-II:
assumes is-valid-problem-sas-plus \Psi
  and \( \forall \text{op} \in \text{set ops}, \text{op} \in \text{set } ((\Psi)_O+) \)
  and \( \text{op}' \in \text{set } [\varphi_O \Psi \text{ op}, \text{op} \leftarrow \text{ops}] \)
shows \( \text{op}' \in \text{set } ((\varphi \Psi)_O) \)
  and is-valid-operator-strips \( (\varphi \Psi) \text{ op}' \)
proof
  let \( ?\Pi = \varphi \Psi \)
  let \( ?\text{ops} = \text{operators-of} \Psi \)
  and \( ?\text{ops}' = \text{strips-problem.operators-of} ?\Pi \)
  obtain \( \text{op} \) where \( \text{op-in: op} \in \text{set ops and op}'-is: op' = \varphi_O \Psi \text{ op} \)
  using strips-equivalent-to-sas-plus-i-a-I[OF \text{assms}].
then have \( \text{nb: op'} \in \text{set } ((\varphi \Psi)_O) \)
  using \( \text{assms}(2) \text{ op-in op}'-is \)
  by fastforce
thus \( \text{op}' \in \text{set } ((\varphi \Psi)_O) \)
  and is-valid-operator-strips \( ?\Pi \text{ op}' \)
proof
  have \( \forall \text{op}' \in \text{set } ?\text{ops}', \text{is-valid-operator-strips } ?\Pi \text{ op}' \)
  using is-valid-problem-sas-plus-then-strips-transformation-too-iii[OF \text{assms}(1)]
  unfolding list-all-iff.
thus \( \text{is-valid-operator-strips } ?\Pi \text{ op}' \)
  using \( \text{nb} \)
  by fastforce
qed fastforce

lemma strips-equivalent-to-sas-plus-i-a-III:
assumes is-valid-problem-sas-plus \Psi
  and \( \forall \text{op} \in \text{set ops}, \text{op} \in \text{set } ((\Psi)_O+) \)
shows execute-parallel-operator \( (\varphi_S \Psi \text{ s}) [\varphi_O \Psi \text{ op} \leftarrow \text{ops}] \)
  = \( (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus s ops})) \)
proof
  \{ 
  fix \( \text{op s} \)
  assume \( \text{op} \in \text{set } ((\Psi)_O+) \)
  moreover have \( (\varphi_O \Psi \text{ op}) \in \text{set } ((\varphi \Psi)_O) \)
using calculation by simp

moreover have \((\varphi_S \Psi s) ++ \text{map-of (effect-to-assignments} (\varphi_O \Psi op))\) = \((\varphi_S \Psi (s ++ \text{map-of (effect-of} (\varphi_O^{-1} \Psi (\varphi_O \Psi op))))))\)

using sas-plus-equivalent-to-strips-i-a-XI[OF assms(1) calculation(2)] by blast

moreover have \((\varphi_O^{-1} \Psi (\varphi_O \Psi op)) = op\)

using sas-plus-operator-inverse-is[OF assms(1) calculation(1)].

ultimately have \((\varphi_S \Psi s) \gg (\varphi_O \Psi op)\)

unfolding execute-operator-def execute-operator-sas-plus-def

by simp

}\note nb\_1 = this

show ?thesis

using assms

proof (induction ops arbitrary: s)

\begin{enumerate}
\item \begin{enumerate}
\item \begin{enumerate}
\item case Nil

then show ?case

unfolding execute-parallel-operator-def execute-parallel-operator-sas-plus-def

by simp

next

\begin{enumerate}
\item case (Cons op ops)

let \(?t = s \ggop op\)

let \(?s' = \varphi_S \Psi s\)

\begin{enumerate}
\item \begin{enumerate}
\item and \(?ops' = [\varphi_O \Psi op. op \leftarrow op \# ops]\)

let \(?t' = \?s' \gg (\varphi_O \Psi op)\)

have \(\text{execute-parallel-operator \?s' \?ops'}\)

\begin{enumerate}
\item unfolding execute-operator-def

by simp

moreover have \((\varphi_S \Psi (\text{execute-parallel-operator-sas-plus s} (op \# ops)))\)

= \((\varphi_S \Psi (\text{execute-parallel-operator-sas-plus \?t ops}))\)

unfolding execute-operator-sas-plus-def

by simp

moreover \begin{enumerate}
\item have \(?t' = (\varphi_S \Psi \?t)\)

using \(\text{nb}_1 \text{Cons.prems}(2)\)

by simp

hence \(\text{execute-parallel-operator \?t'}[\varphi_O \Psi x. x \leftarrow ops]\)

= \((\varphi_S \Psi (\text{execute-parallel-operator-sas-plus \?t ops}))\)

using Cons.IH[of \(?t\) Cons.prems]

by simp

\end{enumerate}

\end{enumerate}

ultimately show ?case

by argo

qed

qed

\end{enumerate}

\end{enumerate}

\end{enumerate}

\end{enumerate}

\end{enumerate}
private lemma strips-equivalent-to-sas-plus-i-a-IV:

assumes is-valid-problem-sas-plus \( \Psi \)
and \( \forall \text{op} \in \text{set ops}, \text{op} \in \text{set } ((\Psi)_{O+}) \)
and are-all-operators-applicable-in I ops
\( \land \) are-all-operator-effects-consistent ops

shows STRIPS-Semantics.are-all-operators-applicable \( (\varphi_S \Psi I) [\varphi_O \Psi \text{ op. op} \leftarrow \text{ops}] \)
\( \land \) STRIPS-Semantics.are-all-operator-effects-consistent \( [\varphi_O \Psi \text{ op. op} \leftarrow \text{ops}] \)

proof –

let \( \text{?vs} = \text{variables-of } \Psi \)
and \( \text{?ops} = \text{operators-of } \Psi \)

let \( \text{?I'} = \varphi_S \Psi I \)
and \( \text{?ops'} = [\varphi_O \Psi \text{ op. op} \leftarrow \text{ops}] \)

have \( nb_1: \forall \text{op} \in \text{set ops. is-operator-applicable-in I op} \)
using assms(3)

unfolding are-all-operators-applicable-in-def list-all-iff
by blast

have \( nb_2: \forall \text{op} \in \text{set ops. is-valid-operator-sas-plus } \Psi \text{ op} \)
using is-valid-problem-sas-plus-then(2) assms(1, 2)
unfolding is-valid-operator-sas-plus-def
by auto

have \( nb_3: \forall \text{op} \in \text{set ops. map-of (precondition-of op) } \subseteq_m I \)
using nb1
unfolding is-operator-applicable-in-def list-all-iff
by blast

\{ 
  fix \( \text{op}_1 \) \( \text{op}_2 \)
assume \( \text{op}_1 \in \text{set ops and } \text{op}_2 \in \text{set ops} \)
hence are-all-operator-effects-consistent \( \text{op}_1 \) \( \text{op}_2 \)
using assms(3)
unfolding are-all-operator-effects-consistent-def list-all-iff
by blast
\}

note \( nb_4 = \text{this} \)

\{ 
  fix \( \text{op}_1 \) \( \text{op}_2 \)
assume \( \text{op}_1 \in \text{set ops and } \text{op}_2 \in \text{set ops} \)
hence \( \forall (v, \text{a}) \in \text{set (effect-of } \text{op}_1). \forall (v', \text{a'}) \in \text{set (effect-of } \text{op}_2). \)
\( v \neq v' \lor \text{a} = \text{a'} \)
using nb4
unfolding are-operator-effects-consistent-def Let-def list-all-iff
by presburger
\}

note \( nb_5 = \text{this} \)

\{ 
  fix \( \text{op}_1' \) \( \text{op}_2' \) \( I \)
assume \( \text{op}_1' \in \text{set } \text{?ops'} \)
and \( \text{op}_2' \in \text{set } \text{?ops'} \)
and \( \exists (v, \text{a}) \in \text{set (add-effects-of } \text{op}_1'). \exists (v', \text{a'}) \in \text{set (delete-effects-of } \text{op}_2'). \)
\( (v, \text{a}) = (v', \text{a'}) \)
moreover obtain \( \text{op}_1 \) \( \text{op}_2 \)
where \( op_1 \in \text{set ops} \)

and \( op_1' = \varnothing \Psi \) \( op_1 \)

and \( op_2 \in \text{set ops} \)

and \( op_2' = \varnothing \Psi \) \( op_2 \)

using strips-equivalent-to-sas-plus-i-a-1[\text{OF assms}\](1, 2) calculation\((1, 2)\)
by auto

moreover have is-valid-operator-sas-plus \( \Psi \) \( op_1 \)

and is-valid-operator-op_2: is-valid-operator-sas-plus \( \Psi \) \( op_2 \)
using calculation\((4, 6)\) \( nb_2 \)
by blast+

moreover obtain \( v \ v' \ a \ a' \)
where \( (v, a) \in \text{set (add-effects-of} \ op_1' \)
and \( (v', a') \in \text{set (delete-effects-of} \ op_2' \)
and \( (v, a) = (v', a') \)
using calculation
by blast

moreover have \( (v, a) \in \text{set (effect-of} \ op_1 \)
using calculation\((5, 10)\)

unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def sasp-op-to-strips-def Let-def
by fastforce

moreover have \( v = v' \) \( and \ a = a' \)
using calculation\((12)\)
by simp+

moreover have { \( (v', a') \in \bigcup (v, a) \in \text{set (effect-of} \ op_2). \)
\{ (v, a') | a', a' \in (R^+ \Psi v) \land a' \neq a \} \}
using sasp-op-to-strips-set-delete-effects-is calculation\((7, 9, 11)\)
by blast

then obtain \( v'' a'' \) where \( (v'', a'') \in \text{set (effect-of} \ op_2 \)
and \( (v', a') \in \{ (v'', a''') | a'', a''' \in (R^+ \Psi v'') \land a''' \neq a'' \} \)
by blast

moreover have \( (v', a'') \in \text{set (effect-of} \ op_2 \)
using calculation
by blast

moreover have \( a' \in R^+ \Psi v'' \) \( and \ a' \neq a'' \)
using calculation\((1, 2)\)
by fast+

ultimately have \( \exists a''. (v', a'') \in \text{set (effect-of} \ op_2) \land a' \in (R^+ \Psi v') \land a' \neq a'' \)
by blast

} moreover obtain \( a'' \) where \( a' \in R^+ \Psi v' \)
and \( (v', a'') \in \text{set (effect-of} \ op_2 \)
and \( a' \neq a'' \)
using calculation\((16)\)
by blast

moreover have \( \exists (v, a) \in \text{set (effect-of} \ op_1). (\exists (v', a') \in \text{set (effect-of} \ op_2). \)

149
\[ v = v' \land a \neq a' \]

using calculation(13, 14, 15, 17, 18, 19)
by blast
— TODO slow.
ultimately have \( \exists \text{op}_1 \in \text{set ops.} \exists \text{op}_2 \in \text{set ops.} \neg \text{are-operator-effects-consistent} \text{op}_1 \text{op}_2 \)
unfolding are-operator-effects-consistent-def list-all-iff
by fastforce

\} note nb_6 = this
show \(?thesis
proof (rule conjI)
{

fix \text{op}'
assume \text{op}' \in \text{set ?ops'}
moreover obtain \text{op} where \text{op-in:} \text{op} \in \text{set ops}
and \text{op}'-is: \text{op}' = \varphi_o \Psi \text{op}
and \text{op}'-in: \text{op}' \in \text{set (}((\varphi \Psi)_o)\text{)}
and is-valid-op: is-valid-operator-strips (\varphi \Psi) \text{op}'
using strips-equivalent-to-sas-plus-i-a-I[\text{OF assms(1, 2)}]
strips-equivalent-to-sas-plus-i-a-II[\text{OF assms(1, 2)}] calculation
by metis
moreover have is-valid-op: is-valid-operator-sas-plus \Psi \text{op}
using nb_2 calculation(2).

fix \text{v, a}
assume \text{v-a-in-preconditions':} \( (\text{v, a}) \in \text{set (strips-operator.precondition-of \text{op}')} \)

have \text{v-a-in-preconditions:} \( (\text{v, a}) \in \text{set (precondition-of \text{op})} \)
using \text{op}'-is
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def Let-def
using \text{v-a-in-preconditions'}
by force
moreover have \( \text{v} \in \text{set ?vs and} \text{a} \in \mathcal{R}_+ \Psi \text{v} \)
using is-valid-operator-sas-plus-then(1,2) is-valid-op calculation(1)
by fastforce
moreover {

have \( \forall (\text{v, a}) \in \text{set (precondition-of \text{op}).} \forall (\text{v'}, \text{a'}) \in \text{set (precondition-of \text{op})}. \)
\( v \neq v' \lor a = a' \)
using is-valid-operator-sas-plus-then(5) is-valid-op
by fast
hence map-of (precondition-of \text{op}) v = Some a
using map-of-constant-assignments-defined-if[\text{OF - v-a-in-preconditions}]
by blast
}
moreover have \( \text{v} \in \text{dom (map-of (precondition-of \text{op}))} \)
using calculation(4)
by blast
moreover have $I \cdot v = \text{Some } a$

using $nb_3$

unfolding $\text{map-le-def}$

using $\text{op-in calculation}(4, 5)$

by $\text{metis}$

moreover have $(v, a) \in \text{dom } ?I'$

using $\text{state-to-strips-state-dom-element-iff}[OF \text{assms}(1)]$

$\text{calculation}(2, 3, 6)$

by $\text{simp}$

ultimately have $?I' (v, a) = \text{Some } \text{True}$

using $\text{state-to-strips-state-range-is}[OF \text{assms}(1)]$

by $\text{simp}$

}\]

hence $\text{STRIPS-Representation.is-operator-applicable-in } ?I' \text{ op}'$

unfolding

$\text{STRIPS-Representation.is-operator-applicable-in-def}$

Let-def list-all-iff

by $\text{fast}$

}\]

thus $\text{are-all-operators-applicable ?I' ops'}$

unfolding $\text{are-all-operators-applicable-def list-all-iff}$

by $\text{blast}$

next

{  \[ 151

  \text{fix } op_1', op_2'$

  \text{assume } op_1'^{-in-ops'}: \text{op}_1' \in \text{set } ops' \text{ and } op_2'^{-in-ops'}: \text{op}_2' \in \text{set } ops'$

  \text{have } \text{STRIPS-Semantics.are-operator-effects-consistent op}_1' \text{ op}_2'$

  unfolding \text{STRIPS-Semantics.are-operator-effects-consistent-def Let-def}

  — TODO proof is symmetrical... refactor into nb.

  proof (rule conjI)

  show $\neg \text{list-ex } (\lambda x. \text{list-ex } ((=) x) (\text{delete-effects-of op}_2'))$

  (\text{add-effects-of op}_1')$

  proof (rule ccontr)

  assume $\neg \text{list-ex } (\lambda v. \text{list-ex } ((=) v) (\text{delete-effects-of op}_2'))$

  (\text{add-effects-of op}_1')$

  then have $\exists (v, a) \in \text{set } (\text{delete-effects-of op}_2').$

  $\exists (v', a') \in \text{set } (\text{add-effects-of op}_1').$ $(v, a) = (v', a')$

  unfolding list-ex-iff

  by $\text{fastforce}$

  then obtain $op_1$ $op_2$ where $op_1 \in \text{set ops}$

  and $op_2 \in \text{set ops}$

  and $\neg \text{are-operator-effects-consistent op}_1 \text{ op}_2$

  using $\text{nb}_6[\text{OF } \text{op}_1'^{-in-ops'} \text{ op}_2'^{-in-ops'}]$

  by $\text{blast}$

  thus $\text{False}$

  using $\text{nb}_4$

  by $\text{blast}$

  qed

next

151
show \( \neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{add-effects-of } op_2')) \) (\text{delete-effects-of } op_1')

proof (rule ccontr)
assume \( \neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{add-effects-of } op_2')) \)
then have \( \exists (v, a) \in \text{set } (\text{delete-effects-of } op_1'). \)
\( \exists (v', a') \in \text{set } (\text{add-effects-of } op_2'). (v, a) = (v', a') \)
unfolding list-ex-iff
by fastforce
then obtain \( op_1 \) \( op_2 \) where
\( op_1 \in \text{set } \text{ops} \)
and \( op_2 \in \text{set } \text{ops} \)
and \( \neg \text{are-operator-effects-consistent } op_1 \) \( op_2 \)
using \( \text{nb}_6[\text{OF } op_2'\text{-in-ops' } op_1'\text{-in-ops'}] \)
by blast
thus False
using \( \text{nb}_4 \)
by blast
qed

qed

private lemma strips-equivalent-to-sas-plus-i-a-V:
assumes is-valid-problem-sas-plus \( \Psi \)
and \( \forall op \in \text{set } \text{ops}, op \in (\Psi)_{\text{O}} \)
and \( \neg (\text{are-all-operators-applicable-in } s \text{ ops} \wedge \text{are-all-operator-effects-consistent ops}) \)
shows \( \neg (\text{STRIPS-Semantics. are-all-operators-applicable } (\phi_S \Psi s) [\phi_o \Psi op \leftarrow op] \wedge \text{STRIPS-Semantics. are-all-operator-effects-consistent } [\phi_o \Psi op \leftarrow op]) \)
proof
let \( ?vs = \text{variables-of } \Psi \)
and \( ?ops = \text{operators-of } \Psi \)
let \( ?s' = \phi_S \Psi s \)
and \( ?ops' = [\phi_o \Psi op \leftarrow op] \)
{ fix \( op \)
assume \( op \in \text{set } \text{ops} \)
hence \( \exists op' \in \text{set } ?ops'. op' = \phi_o \Psi op \)
by simp
}
reveal \( \text{nb}_4 \)
{ fix \( op \)
assume \( op \in \text{set } \text{ops} \)
then have \( op \in (\Psi)_{\text{O}} \)

152
using assms(2)
by blast
then have is-valid-operator-sas-plus $\Psi$ $\psi$
  using is-valid-problem-sas-plus-then(2) assms(1)
unfolding is-valid-operator-sas-plus-def
by auto
hence $\forall (v, a) \in \text{set (precondition-of op)}. \forall (v', a') \in \text{set (precondition-of op)}.$
  $v \neq v' \lor a = a'$
  using is-valid-operator-sas-plus-then(5)
unfolding is-valid-operator-sas-plus-def
by fast
} note nb$_2$ = this
{
  consider (A) $\neg$are-all-operators-applicable-in s ops
  | (B) $\neg$are-all-operator-effects-consistent ops
using assms(3)
by blast
hence $\neg$STRIPS-Semantics.are-all-operators-applicable $\psi s' ?ops'$
  $\lor \neg$STRIPS-Semantics.are-all-operator-effects-consistent $\psi ops'$
proof (cases)
  case A
  then obtain $\psi op$ where op-in: $\psi op \in \text{set ops}$
    and not-precondition-map-le-s: $\neg$ (map-of (precondition-of $\psi op$) $\subseteq_m s$)
  using A
  unfolding are-all-operators-applicable-in-def list-all-iff
  is-operator-applicable-in-def
by blast
  then obtain $\psi op'$ where op'-in: $\psi op' \in \text{set } ?ops'$ and op'-is: $\psi op' = \varphi_\Omega \Psi \psi op$
  using nb$_1$
  by blast
  have $\neg$are-all-operators-applicable $\psi s' ?ops'$
  proof (rule ccontr)
    assume $\neg$are-all-operators-applicable $\psi s' ?ops'$
    then have all-operators-applicable: are-all-operators-applicable $\psi s' ?ops'$
      by simp
    moreover {
      fix $v$
      assume $v \in \text{dom (map-of (precondition-of } \psi op))$
      moreover obtain $\psi a$ where map-of (precondition-of $\psi op$) $v = \text{Some } a$
        using calculation
by blast
      moreover have $(v, a) \in \text{set (precondition-of } \psi op)$
        using map-of-SomeD[OF calculation(2)].
      moreover have $(v, a) \in \text{set (strips-operator.precondition-of } \psi op')$
        using op'-is
      unfolding sasp-op-to-strips-def
      SAS-Plus-STRIPS,sasp-op-to-strips-def
        using calculation(3)
      by auto
    }
moreover have \(?s'(v, a) = \text{Some \: True}\)
using all-operators-applicable calculation
unfolding arc-all-operators-applicable-def
STRIPS-Representation.is-operator-applicable-in-def
is-operator-applicable-in-def Let-def list-all-iff
using \(op'\)-in
by fast
moreover have \((v, a) \in \text{dom } ?s'\)
using calculation(5)
by blast
moreover have \((v, a) \in \text{set (precondition-of } op)\)
using \(op'\)-is calculation(3)
unfolding sasp-op-to-strips-def Let-def
by fastforce
moreover have \(v \in \text{set } ?vs\)
and \(a \in \mathbb{R}_+ \Psi v\)
and \(s v \neq \text{None}\)
using state-to-strips-state-domain-element-iff[OF assms(1)]
calculation(6)
by simp
moreover have \(?s'(v, a) = \text{Some (the } (s v) = a)\)
using state-to-strips-state-range-is[OF assms(1) calculation(6)].
moreover have \(\text{the } (s v) = a\)
using calculation(5, 11)
by fastforce
moreover have \(s v = \text{Some } a\)
using calculation(12) option.collapse[OF calculation(10)]
by argo
moreover have \(\text{map-of (precondition-of } op) v = \text{Some } a\)
using map-of-constant-assignments-defined-if[OF nb2[OF op-in]
calculation(7)].
ultimately have \(\text{map-of (precondition-of } op) v = s v\)
by argo
}
then have \(\text{map-of (precondition-of } op) \subseteq_m s\)
unfolding map-le-def
by blast
thus False
using not-precondition-map-le-s
by simp
qed
thus \(?thesis\)
by simp
next
case B
{
obtain \(op_1 \: op_2 \: v \: v' \: a \: a'\)
where \(op_1 \in \text{set ops}\)
\[\text{and } \text{op}_2 \text{-in: } \text{op}_2 \in \text{set ops}\]
\[\text{and } v \cdot a \text{-in: } (v, a) \in \text{set (effect-of } \text{op}_1)\]
\[\text{and } v' \cdot a' \text{-in: } (v', a') \in \text{set (effect-of } \text{op}_2)\]
\[\text{and } v \text{-is: } v = v' \text{ and } a \cdot a' \text{-is: } a \neq a'\]

using \(B\)

unfolding are-all-operator-effects-consistent-def

\(\text{are-operator-effects-consistent-def list-all-iff Let-def}\)

by blast

moreover obtain \(\text{op}_1 \text{' op}_2 \text{' where } \text{op}_1 \text{' } \in \text{ set } ?\text{ops}' \text{ and } \text{op}_1 \text{' } = \varphi_0 \Psi\)

\[\text{op}_1\]
\[\text{and } \text{op}_1 \text{' } \in \text{ set } ?\text{ops}' \text{ and } \text{op}_2 \text{' } -\text{is: } \text{op}_2 \text{' } = \varphi_0 \Psi \text{ op}_2\]

using \(\text{nb}_1[\text{OF calculation}(1)] \text{ nb}_1[\text{OF calculation}(2)]\)

by blast

moreover have \((v, a) \in \text{ set (add-effects-of } \text{op}_1')\)

using calculation\((3, 8)\)

unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def

\(\text{sasp-op-to-strips-def Let-def}\)

by force

moreover \{\}

have \(\text{is-valid-operator-sas-plus } \Psi \text{ op}_1\)

using \(\text{assms}(2) \text{ calculation}(1) \text{ is-valid-problem-sas-plus-then}(2) \\text{assms}(1)\)

unfolding \(\text{is-valid-operator-sas-plus-def}\)

by auto

moreover have \(\text{is-valid-operator-sas-plus } \Psi \text{ op}_2\)

using \(\text{sublocale-sas-plus-finite-domain-representation-ii}(2)[\text{OF assms}(1)] \text{ assms}(2) \text{ op}_2 \text{-in}\)

by blast

moreover have \(a \in R_+ \Psi v\)

using \(\text{is-valid-operator-sas-plus-then}(4) \text{ calculation } v \cdot a \text{-in}\)

unfolding \(\text{is-valid-operator-sas-plus-def}\)

by fastforce

ultimately have \((v, a) \in \text{ set (delete-effects-of } \text{op}_2')\)

using \(\text{sasp-op-to-strips-set-delete-effects-is}[\text{of } \Psi \text{ op}_2]\)

\(v' \cdot a' \text{-in v-is a-is}\)

using \(\text{op}_2' \text{-is}\)

by blast

}\}

— TODO slow.

ultimately have \(\exists \text{op}_1' \in \text{ set } ?\text{ops}' \cdot \exists \text{op}_2' \in \text{ set } ?\text{ops}'\)

\((v, a) \in \text{ set (delete-effects-of } \text{op}_2') \cdot (v', a') \in \text{ set (add-effects-of } \text{op}_1').\)

\((v, a) = (v', a')\)

by fastforce

}\}

then have \(\neg \text{STRIPS-Semantics.are-all-operator-effects-consistent } ?\text{ops}'\)

unfolding \(\text{STRIPS-Semantics.are-all-operator-effects-consistent-def}\)

\(\text{STRIPS-Semantics.are-operator-effects-consistent-def list-all-iff list-ex-iff}\)

Let-def

by blast

thus \(\text{?thesis}\)
by simp
qed
}
thus ?thesis
by blast
qed

lemma strips-equivalent-to-sas-plus-i-a:
assumes is-valid-problem-sas-plus $\Psi$
and $\text{dom } I \subseteq \text{set } (\langle \Psi \rangle_{\nu}^\bot)$
and $\forall v \in \text{dom } I. \text{ the } (I v) \in \mathcal{R}_+ \Psi v$
and $\text{dom } G \subseteq \text{set } (\langle \Psi \rangle_{\nu}^\bot)$
and $\forall v \in \text{dom } G. \text{ the } (G v) \in \mathcal{R}_+ \Psi v$
and $\forall \text{ ops } \in \text{ set } \psi. \forall op \in \text{ set ops. op } \in \text{ set } (\langle \Psi \rangle_{\sigma}^\bot)$
and $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi$
shows $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi I) (\varphi_P \Psi \psi)$

proof
let $?\Pi = \varphi \Psi$
and $?G' = \varphi_S \Psi G$
show $?\text{thesis}$
using assms

proof (induction $\psi$ arbitrary: $I$

| case Nil |
| let $?I' = \varphi_S \Psi I$
| have $G \subseteq_m I$
| using Nil
| by simp
| moreover have $?G' \subseteq_m ?I'$
| using state-to-strips-state-map-le-iff[OF Nil.prems(1, 4, 5)]
| calculation..
| ultimately show $?\text{case}$
| unfolding SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
| sas-plus-parallel-plan-to-strips-parallel-plan-def
| by simp
|
| next |
| case (Cons $\text{ops } \psi$)
| let $?\text{vs } = \text{variables-of } \Psi$
| and $?\text{ops } = \text{operators-of } \Psi$
| and $?J = \text{execute-parallel-operator-sas-plus } I \text{ ops}$
| and $?\pi = \varphi_P \Psi (\text{ops } \# \psi)$
| let $?I' = \varphi_S \Psi I$
| and $?J' = \varphi_S \Psi ?J$
| and $?\text{ops'} = [\varphi_O \Psi \text{ op. op } \leftarrow \text{ops}]$
| |
| \{ fixation $\text{op } v a$
| assume $\text{op } \in \text{ set } \text{ops and } (v, a) \in \text{ set } \text{effect-of } \text{op}$
| moreover have $\text{op } \in \text{ set } ?\text{ops}$
| using Cons.prems(6) calculation(1)
| |
| 156
by simp
moreover have is-valid-operator-sas-plus \( \Psi \) \( op \)
  using is-valid-problem-sas-plus-then(2) Cons.prems(1) calculation(3)
unfolding is-valid-operator-sas-plus-def
by auto
ultimately have \( v \in \text{set } ((\Psi)_V^+) \)
  and \( a \in \mathbb{R}_+ \\Psi v \)
  using is-valid-operator-sas-plus-then(3,4)
  by fastforce+
} note \( nb_1 = \text{this} \)
show \( \forall \text{case} \)
proof (cases are-all-operators-applicable-in \( I \) \( \text{ops} \)
  \& are-all-operator-effects-consistent \( \text{ops} \))
case True
  \{ have \( (\varphi_P \Psi \text{ (ops } \# \psi)) = \ ?\text{ops}' \# (\varphi_P \Psi \psi) \)
    unfolding sas-plus-parallel-plan-to-strips-parallel-plan-def
    SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def
    by simp
  moreover have \( \forall \text{op} \in \text{set } \text{ops}. \ \text{op} \in \text{set } ((\Psi)_O^+) \)
    using Cons.prems(6)
    by simp
  moreover have STRIPS-Semantics.are-all-operators-applicable \(?I' \ ?\text{ops}' \)
    and STRIPS-Semantics.are-all-operator-effects-consistent \(?\text{ops}' \)
    using strips-equivalent-to-sas-plus-i-a-IV[of Cons.prems(1) - True]
calculation
    by blast+
  ultimately have execute-parallel-plan \(?I' \ ?\pi \)
    = execute-parallel-plan (execute-parallel-operator \(?I' \ ?\text{ops}' \) (\varphi_P \Psi \psi))
    by fastforce
  \} — NOTE Instantiate the IH on the next state of the SAS+ execution
execute-parallel-operator-sas-plus \( I \) \( \text{ops} \).
moreover
  \{ have \( \text{dom } I \subseteq \text{set } \text{(sas-plus-problem.variables-of } \Psi) \)
    using Cons.prems(2)
    by blast
  moreover have \( \forall \text{op} \in \text{set } \text{ops}. \ \forall (v, a) \in \text{set } \text{(effect-of } \text{op}). \)
    \( v \in \text{set } ((\Psi)_V^+) \)
    using \( nb_1(1) \)
    by blast
  ultimately have \( \text{dom } ?J \subseteq ((\Psi)_V^+) \)
    using sas-plus-equivalent-to-strips-i-a-IX[of I \ set \ ?vs]
    by simp
  \} note \( nb_2 = \text{this} \)
moreover \{ 
  \textbf{have} \ \text{dom } I \subseteq \text{sas-plus-problem.variables-of } \Psi \\
  \text{using} \ \text{Cons.prems(2)} \\
  \text{by} \ \text{blast} \\
  \textbf{moreover have} \ \text{sas-plus-problem.variables-of } \Psi \\
  \subseteq \text{dom } \text{range-of } \Psi \\
  \text{using} \ \text{is-valid-problem-sas-plus-dom-sas-plus-problem-range-of } \text{assms(1)} \\
  \text{by} \ \text{auto}  \\
  \textbf{moreover} \{ \\
    \text{fix } v \\
    \text{assume } v \in \text{dom } I \\
    \textbf{moreover have} v \in \text{sas-plus-problem.variables-of } \Psi \\
    \text{using} \ \text{Cons.prems(2)} \text{ calculation} \\
    \text{by} \ \text{blast} \\
    \textbf{ultimately have} \ (I \ v) \in \text{set } \text{range-of } \Psi \\
    \text{using} \ \text{Cons.prems(3)} \\
    \text{using} \ \text{set-the-range-of-is-range-of-sas-plus-if } \text{OF } \text{assms(1)} \\
    \text{by} \ \text{blast}  \\
  }  \\
  \textbf{moreover have} \ \forall \text{op}\in \text{set ops} \ \forall (v, a)\in \text{set } \text{effect-of } \text{op}, \\
  v \in \text{sas-plus-problem.variables-of } \Psi \land a \in \text{set } \text{range-of } \Psi \\
  \text{using} \ \text{set-the-range-of-is-range-of-sas-plus-if } \text{OF } \text{assms(1)} \\
  \text{by} \ \text{force} \\
  \textbf{moreover have} nb_1: \forall v \in \text{dom } ?J. \ \text{the } (?J \ v) \in \text{set } \text{range-of } \Psi \\
  \text{using} \ \text{sas-plus-equivalent-to-strips-i-a-X } \text{of } \text{set } ?\text{vs range-of } \Psi \ \text{ops} \\
  \text{calculation} \\
  \text{by} \ \text{fast} \\
  \textbf{moreover} \{ \\
    \text{fix } v \\
    \text{assume } v \in \text{dom } ?J \\
    \textbf{moreover have} v \in \text{sas-plus-problem.variables-of } \Psi \\
    \text{using} \ \text{nb}_2\ \text{ calculation} \\
    \text{by} \ \text{blast} \\
    \textbf{moreover have} \ (\text{the } \text{range-of } \Psi) = \mathcal{R} \Psi \ v \\
    \text{using} \ \text{set-the-range-of-is-range-of-sas-plus-if } \text{OF } \text{assms(1)} \\
    \text{calculation}(2) \\
    \text{by} \ \text{presburger} \\
    \textbf{ultimately have} \ (\text{?J } v) \in \mathcal{R} \Psi \ v \\
    \text{using} \ \text{nb}_3 \\
    \text{by} \ \text{blast} \\
  }  \\
  \textbf{ultimately have} \forall v \in \text{dom } ?J. \ \text{the } (?J \ v) \in \mathcal{R} \Psi \ v \\
  \text{by} \ \text{fast}  \\
  \textbf{moreover have} \ \forall \text{ops}\in \text{set } \psi \ \forall \text{op}\in \text{set ops} \ \text{op} \in \text{set } ?\text{ops} \\
  \text{using} \ \text{Cons.prems(6)}
by auto
moreover have \( G \subseteq_m \text{execute-parallel-plan-sas-plus} \ ?J \ \psi \)
using Cons.prems\(7\) True
by simp
ultimately have \((\varphi_S \Psi \ G) \subseteq_m \text{execute-parallel-plan} \ ?J' (\varphi_P \Psi \ \psi)\)
using Cons.IH[of ?J', OF Cons.prems\(1\) - - Cons.prems\(4, 5\)]
by fastforce
}
moreover have \( \text{execute-parallel-operator} \ ?I' \ ?\text{ops}' = \ ?J' \)
using assms\(1\) strips-equivalent-to-sas-plus-i-a-III[OF assms\(1\)] Cons.prems\(6\)
by auto
ultimately show \(?\text{thesis}\)
by argo
next
case False
then have \( \text{nb} : \ G \subseteq_m \ I \)
using Cons.prems\(7\)
by force
moreover {
have \( ?\pi = ?\text{ops}' \neq (\varphi_P \Psi \ \psi)\)
unfolding sas-plus-parallel-plan-to-strips-parallel-plan-def
SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
sas-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def Let-def
by auto
moreover have set \( ?\text{ops}' \subseteq \text{set} (\text{strips-problem.operators-of} \ ?\Pi)\)
using strips-equivalent-to-sas-plus-i-a-II[OF assms\(1\)] Cons.prems\(6\)
by auto
moreover have \( \neg (\text{STRIPS-Semantics.are-all-operators-applicable} \ ?I' \ ?\text{ops}') \)
\wedge STRIPS-Semantics. are-all-operator-effects-consistent \( ?\text{ops}' \)
using strips-equivalent-to-sas-plus-i-a-V[OF assms\(1\) - False] Cons.prems\(6\)
by force
ultimately have \( \text{execute-parallel-plan} \ ?I' \ ?\pi = ?I' \)
by auto
}
moreover have \( G' \subseteq_m \ ?I' \)
using state-to-strips-state-map-le-iff[OF Cons.prems\(1, 4, 5\)] \( \text{nb} \)
by blast
ultimately show \(?\text{thesis}\)
by presburger
qed
qed

lemma strips-equivalent-to-sas-plus-i:
assumes \( \text{is-valid-problem-sas-plus} \ \Psi \)
and \( \text{is-parallel-solution-for-problem} \ \Psi \ \psi \)
shows \((\text{strips-problem.goal-of } (\varphi \Psi)) \subseteq_m \text{execute-parallel-plan}(\text{strips-problem.initial-of } (\varphi \Psi)) (\varphi P \Psi \psi)\)

**proof** –

let \(\text{?vs} = \text{variables-of } \Psi\)
and \(\text{?ops} = \text{operators-of } \Psi\)
and \(\text{?I} = \text{initial-of } \Psi\)
and \(\text{?G} = \text{goal-of } \Psi\)

let \(\text{?I}' = \text{strips-problem.initial-of } ?I\)
and \(\text{?G}' = \text{strips-problem.goal-of } ?I\)

have \(\text{dom } ?I \subseteq \text{set } ?\text{vs}\)
using \(\text{is-valid-problem-sas-plus-then}(3) \text{ asms}(1)\)
by auto

moreover have \(\forall \, v \in \text{dom } ?I. \, \text{the } (\text{?I } v ) \in R_+ \Psi \, v\)
using \(\text{is-valid-problem-sas-plus-then}(4) \text{ asms}(1) \) calculation
by auto

moreover have \(\text{dom } ?G \subseteq \text{set } ((\Psi)\cup)\)
using \(\text{is-valid-problem-sas-plus-then}(5) \text{ asms}(1)\)
by auto

moreover have \(\forall \, v \in \text{dom } ?G. \, \text{the } (\text{?G } v ) \in R_+ \Psi \, v\)
using \(\text{is-valid-problem-sas-plus-then}(6) \text{ asms}(1)\)
by auto

moreover have \(\forall \, \text{ops } \in \text{set } \psi. \, \forall \, \text{op } \in \text{set } \text{ops}. \, \text{op } \in \text{set } ?\text{ops}\)
using \(\text{is-parallel-solution-for-problem-plan-operator-set}[\text{OF asms}(2)]\)
by fastforce

moreover have \(\text{?G } \subseteq_m \text{execute-parallel-plan-sas-plus } ?I \psi\)
using \(\text{asms}(2)\)
unfolding \(\text{is-parallel-solution-for-problem-def}\)
by simp

ultimately show \(\text{?thesis}\)
using \(\text{strips-equivalent-to-sas-plus-i-a}[\text{OF asms}(1), \, \text{of } ?I \, ?G \, \psi]\)
unfolding \(\text{sas-plus-problem-to-strips-problem-def}\)
\(\text{SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def}\)
\(\text{state-to-strips-state-def}\)
\(\text{SAS-Plus-STRIPS.state-to-strips-state-def}\)
by force

ged

**lemma** \(\text{strips-equivalent-to-sas-plus-ii}:\)

assumes \(\text{is-valid-problem-sas-plus } \Psi\)
and \(\text{is-parallel-solution-for-problem } \Psi \, \psi\)
shows \(\text{list-all } \text{(list-all } \lambda \text{op. } \text{ListMem op } (\text{strips-problem.operators-of } (\varphi \Psi)))\)
(\(\varphi P \Psi \psi\))

**proof** –

let \(\text{?ops } = \text{operators-of } \Psi\)
let \(\text{?I} = \varphi \Psi\)
let \(\text{?ops}' = \text{strips-problem.operators-of } ?I\)
and \( ?\pi = \varphi_P \Psi \psi \)

have is-valid-problem-strips \(?\Pi\)
    using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]
    by simp

have \( nb_1: \forall \text{op} \in \text{set } ?\text{ops}, (\exists \text{op}' \in \text{set } ?\text{ops}'. \text{op}' = (\varphi_O \Psi \text{op})) \)
    unfolding sas-plus-problem-to-strips-problem-def
    SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def Let-def
    sasp-op-to-strips-def
    by force

\{
  fix \text{ops op op'}
  assume \text{ops} \in \text{set } \psi \text{ and } \text{op} \in \text{set } \text{ops}
  moreover have \text{op} \in \text{set } ((\Psi)_O+)
    using is-parallel-solution-for-problem-plan-operator-set[OF assms(2)]
    calculation
    by blast
  moreover obtain \text{op'} where \text{op'} \in \text{set } ?\text{ops}' \text{ and } \text{op}' = (\varphi_O \Psi \text{op})
    using \( nb_1 \) calculation(3)
    by auto
  ultimately have \((\varphi_O \Psi \text{op}) \in \text{set } ?\text{ops}'\)
    by blast
\}

thus ?thesis
    unfolding list-all-iff ListMem-iff Let-def
    sas-plus-problem-to-strips-problem-def
    SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
    sas-plus-parallel-plan-to-strips-parallel-plan-def
    SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def
    Let-def
    by auto

qed

The following lemma proves the complementary proposition to theorem ??.
Namely, given a parallel solution \( \psi \) for a SAS+ problem, the transformation
to a STRIPS plan \( \varphi_P \Psi \psi \) also is a solution to the corresponding STRIPS
problem \( \Pi \equiv \varphi \Psi \). In this direction, we have to show that the execution of
the transformed plan reaches the goal state \( G' \equiv \Pi_G \) of the corresponding
STRIPS problem, i.e.

\( G' \subseteq_m \text{execute-parallel-plan } I' \pi \)

and that all operators in the transformed plan \( \pi \) are operators of \( \Pi \).

theorem
    strips-equivalent-to-sas-plus:
    assumes is-valid-problem-sas-plus \( \Psi \)
        and is-parallel-solution-for-problem \( \Psi \psi \)
    shows STRIPS-Semantics.is-parallel-solution-for-problem \((\varphi \Psi) (\varphi_P \Psi \psi)\)
proof
let $\Pi = \phi \Psi$
let $?I' = \text{strips-problem\.initial-of} \ ?\Pi$
and $?G' = \text{strips-problem\.goal-of} \ ?\Pi$
and $?\text{ops}' = \text{strips-problem\.operators-of} \ ?\Pi$
and $?\pi = \phi P \Psi \psi$

show $?\text{thesis}$
using unfolding $\text{STRIPS-Semantics}.\text{is-parallel-solution-for-problem-def}$
proof (rule conjI)
show $?G' \subseteq_m \text{execute-parallel-plan} \ ?I' \ ?\pi$
using $\text{strips-equivalent-to-sas-plus-i}[OF \ \text{assms}]$
by simp

next
show list-all $\text{(list-all (\lambda op. \text{ListMem} op \ ?\text{ops}'}) \ ?\pi}$
using $\text{strips-equivalent-to-sas-plus-ii}[OF \ \text{assms}]$.
qed

qed

lemma $\text{embedded-serial-sas-plus-plan-operator-structure}$:
assumes $\text{ops} \in \text{set (embed} \ \psi)$
obtains $\text{op}$
where $\text{op} \in \text{set} \ \psi$
and $[\varphi_\alpha \Psi \text{ op. op} \leftarrow \text{ops}] = [\varphi_\alpha \Psi \text{ op}]$
proof
let $?\psi' = \text{embed} \ \psi$
{  
have $?\psi' = [[\text{op}], \text{op} \leftarrow \psi]$  
by (induction $\psi$; force)
moreover obtain $\text{op} \ where \ \text{ops} = [\text{op}] \ \text{and} \ \text{op} \in \text{set} \ \psi$
using $\text{assms calculation}$
by fastforce
ultimately have $\exists \text{op} \in \text{set} \ \psi. \ [\varphi_\alpha \Psi \text{ op. op} \leftarrow \text{ops}] = [\varphi_\alpha \Psi \text{ op}]$
by auto
}
thus $?\text{thesis}$
using that
by meson

qed

private lemma $\text{serial-sas-plus-equivalent-to-serial-strips-i}$:
assumes $\text{ops} \in \text{set (}\varphi_\beta \Psi (\text{embed} \ \psi))$
obtains $\text{op} \ where \ \text{op} \in \text{set} \ \psi \ \text{and} \ \text{ops} = [\varphi_\alpha \Psi \text{ op}]$
proof
let $?\psi' = \text{embed} \ \psi$
{  
have set $(\varphi_\beta \Psi (\text{embed} \ \psi)) = \{ [\varphi_\alpha \Psi \text{ op. op} \leftarrow \text{ops}] \ | \ \text{ops. ops} \in \text{set} \ ?\psi' \}$

unfolding $\text{sas-plus-parallel-plan-to-strips-parallel-plan-def}$
$\text{SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def}$

162
Having established the equivalence of parallel STRIPS and SAS+, we can now show the equivalence in the serial case. The proof combines the embedding theorem for serial SAS+ solutions (??), the parallel plan equivalence theorem ??, and the flattening theorem for parallel STRIPS plans (??). More precisely, given a serial SAS+ solution $\psi$ for a SAS+ problem $\Psi$, the embedding theorem confirms that the embedded plan $\text{List-Supplement}.\text{embed} \psi$ is an equivalent parallel solution to $\Psi$. By parallel plan equivalence, $\pi \equiv \varphi \Psi \text{List-Supplement}.\text{embed} \psi$ is a parallel solution for the corresponding STRIPS problem $\varphi \Psi$. Moreover, since $\text{List-Supplement}.\text{embed} \psi$ is a plan consisting of singleton parallel operators, the same is true for $\pi$. Hence, the flattening lemma applies and $\text{concat} \pi$ is a serial solution for $\varphi \Psi$. Since $\text{concat}$ moreover can be shown to be the inverse of $\text{List-Supplement}.\text{embed}$, the term

$$\text{concat} \pi = \text{concat} (\varphi \Psi \text{ (embed} \psi))$$
can be reduced to the intuitive form

\[ \pi = [\varphi_O \Psi \ op. \ op \leftarrow \psi] \]

which concludes the proof.

**Theorem**

**serial-sas-plus-equivalent-to-serial-strips:**

**Assumes** is-valid-problem-sas-plus \( \Psi \)

**And** SAS-Plus-Semantics.is-serial-solution-for-problem \( \Psi \ \psi \)

**Shows** STRIPS-Semantics.is-serial-solution-for-problem (\( \varphi \ \Psi \)) [\( \varphi_O \Psi \ op. \ op \leftarrow \psi \)]

**Proof**

- **let** \( ?\psi' = \text{embed} \ \psi \)
- **and** \( ?\Pi = \varphi \ \Psi \)
- **let** \( ?\pi' = \varphi_P \ \Psi \ ?\psi' \)
- **let** \( ?\pi = \text{concat} \ ?\pi' \)
  
  \{ 
  
  **have** SAS-Plus-Semantics.is-parallel-solution-for-problem \( \Psi \ ?\psi' \)
  
  **using** execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus[\( \text{OF assms} \)]
  
  **by** simp
  
  **hence** STRIPS-Semantics.is-parallel-solution-for-problem \( ?\Pi \ ?\pi' \)
  
  **using** strips-equivalent-to-sas-plus[\( \text{OF assms}(1) \)]
  
  **by** simp
  
  \}

**Moreover have** \( ?\pi = [\varphi_O \Psi \ op. \ op \leftarrow \psi] \)

**by** simp

**Moreover have** is-valid-problem-strips \( ?\Pi \)

**using** is-valid-problem-sas-plus-then-strips-transformation-too[\( \text{OF assms}(1) \)].

**Moreover have** \( \forall \ \text{ops} \in \text{set} \ ?\pi'. \ \exists \ \text{op} \in \text{set} \ \psi. \ \text{ops} = [\varphi_O \Psi \ \text{op}] \)

**using** serial-sas-plus-equivalent-to-serial-strips-i[\( \text{of} - \Psi \ \psi \)]

**by** metis

**Ultimately show** ?thesis

**using** STRIPS-Semantics.flattening-lemma[\( \text{of} ?\Pi \)]

**by** metis

**QED**

**Lemma** embedded-serial-strips-plan-operator-structure:

**Assumes** \( \text{ops}' \in \text{set} \ (\text{embed} \ \pi) \)

**Obtains** \( \text{op} \)

**where** \( \text{op} \in \text{set} \ \pi \ \text{and} \ [\varphi_O^{-1} \ \Pi \ \text{op} \leftarrow \text{ops}'] = [\varphi_O^{-1} \ \Pi \ \text{op}] \)

**Proof**

- **let** \( ?\pi' = \text{embed} \ \pi \)
  
  \{ 
  
  **have** \( ?\pi' = [\text{[\text{op}]. \ \text{op} \leftarrow \text{[\text{op}]}]} \)
  
  **by** (\text{induction} \ \pi; \ \text{force})
  
  **Moreover obtain** \( \text{op} \ \text{where} \ \text{ops}' = [\text{op}] \ \text{and} \ \text{op} \in \text{set} \ \pi \)
  
  **using** calculation assms
  
  **by** fastforce

164
ultimately have $\exists \, \text{op} \in \text{set} \, \pi. \, [\varphi_O^{-1} \Pi \text{op} \leftarrow \text{ops}] = [\varphi_O^{-1} \Pi \text{op}]

by auto

} 

thus $?\text{thesis}

using that

by meson

qed

private lemma serial-strips-equivalent-to-serial-sas-plus-i:

assumes $\text{ops} \in \text{set} \, (\varphi_P^{-1} \Pi \text{embed} \, \pi))$

obtains $\text{op} \, \text{where} \, \text{op} \in \text{set} \, \pi \, \text{and} \, \text{ops} = [\varphi_O^{-1} \Pi \text{op}]

proof –

let $?\pi' = \text{embed} \, \pi

{ 

have set (\varphi_P^{-1} \Pi (\text{embed} \, \pi)) = \{ [\varphi_O^{-1} \Pi \text{op} \leftarrow \text{ops} \mid \text{ops. ops} \in \text{set} \, ?\pi'] 

unfolding strips-parallel-plan-to-sas-plus-parallel-plan-def
strips-op-to-sasp-def set-map
using setcompr-eq-image
by blast

moreover obtain $\text{ops}' \, \text{where} \, \text{ops}' \in \text{set} \, ?\pi' \, \text{and} \, \text{ops} = [\varphi_O^{-1} \Pi \text{op} \leftarrow \text{ops}']

using assms(1) calculation
by blast

moreover obtain $\text{op} \, \text{where} \, \text{op} \in \text{set} \, \pi \, \text{and} \, \text{ops} = [\varphi_O^{-1} \Pi \text{op}]

by blast

ultimately have $\exists \, \text{op} \in \text{set} \, \pi. \, \text{ops} = [\varphi_O^{-1} \Pi \text{op}]

by meson

} 

thus $?\text{thesis}

using that...

qed

private lemma serial-strips-equivalent-to-serial-sas-plus-ii[simp]:

concat (var_P^{-1} \Pi (\text{embed} \, \pi)) = [\varphi_O^{-1} \Pi \text{op} \leftarrow \pi]

proof –

let $?\pi' = \text{List-Supplement}.\text{embed} \, \pi

have concat ($\varphi_P^{-1} \Pi ?\pi' \rangle = \text{map} \, (\lambda \text{op.} \, \varphi_O^{-1} \Pi \text{op}) \, \text{(concat ?\pi')}

unfolding strips-parallel-plan-to-sas-plus-parallel-plan-def
strips-op-to-sasp-def set-map
SAS-Plus-STRIPS.strips-op-to-sasp-def Let-def
map-concat
by simp
also have $\ldots = \text{map} \, (\lambda \text{op.} \varphi_O^{-1} \Pi \text{op}) \, \pi

unfolding concat-is-inverse-of-embed[of \, \pi]..

finally show concat (\varphi_P^{-1} \Pi (\text{embed} \, \pi)) = [\varphi_O^{-1} \Pi \text{op} \leftarrow \pi].
Using the analogous lemmas for the opposite direction, we can show the counterpart to theorem ?? which shows that serial solutions to STRIPS solutions can be transformed to serial SAS+ solutions via composition of embedding, transformation and flattening.

**theorem**

*serial-strips-equivalent-to-serial-sas-plus:*

**assumes** is-valid-problem-sas-plus $\Psi$

**and** STRIPS-Semantics.is-serial-solution-for-problem ($\varphi \Psi$) $\pi$

**shows** SAS-Plus-Semantics.is-serial-solution-for-problem $\Psi [\varphi^{-1}_O \Psi . \ op \ op \leftarrow \pi]$

**proof** –

let $\pi' = \text{embed } \pi$

and $\Pi = \varphi \Psi$

let $\psi' = \varphi^{-1}_P \Psi ? \pi'$

let $\psi = \text{concat } ? \psi'$

{ have STRIPS-Semantics.is-parallel-solution-for-problem $\Pi \ ? \pi'$

using embedding-lemma[OF is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]

assms(2)].

hence SAS-Plus-Semantics.is-parallel-solution-for-problem $\Psi \ ? \psi'$

using sas-plus-equivalent-to-strips[OF assms(1)]

by simp
}

moreover have $? \psi = [\varphi^{-1}_O \Psi . \ op \ op \leftarrow \pi]$ by simp

moreover have is-valid-problem-strips $\Pi$

using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)].

moreover have $\forall \ ops \in \ set \ ? \psi'. \ \exists \ op \in \ set \ \pi. \ ops = [\varphi^{-1}_O \ Psi \ op]$

using serial-strips-equivalent-to-serial-sas-plus-i by metis

ultimately show $? \text{thesis}$

using flattening-lemma[OF assms(1)]

by metis

qed

6.2 Equivalence of SAS+ and STRIPS

— Define the sets of plans with upper length bound as well as the sets of solutions with upper length bound for SAS problems and induced STRIPS problems.

We keep this polymorphic by not specifying concrete types so it applies to both STRIPS and SAS+ plans.

**abbreviation** bounded-plan-set

**where** bounded-plan-set $\text{ops} \ k \equiv \{ \ \pi. \ \text{set } \pi \subseteq \text{set } \text{ops} \wedge \text{length } \pi = k \ \}$

**definition** bounded-solution-set-sas-plus'

:: (variable, domain) sas-plus-problem


⇒ nat
⇒ (variable, domain) sas-plus-plan set

where bounded-solution-set-sas-plus′ Ψ k
≡ { ψ. is-serial-solution-for-problem Ψ ψ ∧ length ψ = k }

abbreviation bounded-solution-set-sas-plus
:: (variable, domain) sas-plus-problem
⇒ nat
g⇒ (variable, domain) sas-plus-plan set

where bounded-solution-set-sas-plus Ψ N
≡ (∪ k ∈ {0..N}. bounded-solution-set-sas-plus′ Ψ k)

definition bounded-solution-set-strips'
:: (variable × domain) strips-problem
⇒ nat
g⇒ (variable × domain) strips-plan set

where bounded-solution-set-strips' Π k
≡ { π. STRIPS-Semantics.is-serial-solution-for-problem Π π ∧ length π = k }

abbreviation bounded-solution-set-strips
:: (variable × domain) strips-problem
⇒ nat
g⇒ (variable × domain) strips-plan set

where bounded-solution-set-strips Π N
≡ (∪ k ∈ {0..N}. bounded-solution-set-strips' Π k)

— Show that plan transformation for all SAS Plus solutions yields a STRIPS solution for the induced STRIPS problem with same length.

We first show injectiveness of plan transformation λψ. [φ O Ψ op. op ← ψ] on the set of plans Pk ≡ bounded-plan-set (operators-of Ψ) k with length bound k. The injectiveness of Solk ≡ bounded-solution-set-sas-plus Ψ k—the set of solutions with length bound k—then follows from the subset relation Solk ⊆ Pk.

lemma sasp-op-to-strips-injective:
assumes (φ O Ψ op1) = (φ O Ψ op2)
shows op1 = op2

proof −

let ?op1' = φ O Ψ op1
and ?op2' = φ O Ψ op2

{ have strips-operator.precondition-of ?op1' = strips-operator.precondition-of ?op2'

using assms
by argo

hence sas-plus-operator.precondition-of op1 = sas-plus-operator.precondition-of op2

unfolding sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
Let-def
by simp

167
moreover { 
    have strips-operator.add-effects-of ?op_1' = strips-operator.add-effects-of ?op_2'
        using assms
        unfolding sasp-op-to-strips-def Let-def
        by argo
    hence sas-plus-operator.effect-of op_1 = sas-plus-operator.effect-of op_2
        unfolding sasp-op-to-strips-def Let-def
        SAS-Plus-STRIPS.sasp-op-to-strips-def
        by simp
}
ultimately show ?thesis
    by simp
qed

lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a:
assumes is-valid-problem-sas-plus Ψ
shows inj-on (λψ. [ϕ_0 Ψ op. op ← ψ]) (bounded-plan-set (sas-plus-problem.operators-of Ψ)) k
proof –
    let ?ops = sas-plus-problem.operators-of Ψ
    and ?ϕ_P = λψ. [ϕ_0 Ψ op. op ← ψ]
    let ?P = bounded-plan-set ?ops
    { 
      fix ψ_1 ψ_2
      assume ψ_1-in: ψ_1 ∈ ?P k
      and ψ_2-in: ψ_2 ∈ ?P k
      and ϕ_P-of-ψ_1-is-ϕ_P-of-ψ_2: (?ϕ_P ψ_1) = (?ϕ_P ψ_2)
      hence ψ_1 = ψ_2
      proof (induction k arbitrary: ψ_1 ψ_2)
        case 0
        then have length ψ_1 = 0
        and length ψ_2 = 0
        using ψ_1-in ψ_2-in
        unfolding bounded-solution-set-sas-plus'-def
        by blast
        then show ?case
        by blast
      next
      case (Suc k)
        moreover have length ψ_1 = Suc k and length ψ_2 = Suc k
            using length-Suc-conv Suc(2, 3)
        unfolding bounded-solution-set-sas-plus'-def
            by blast
        moreover obtain op_1 ψ_1' where ψ_1 = op_1 # ψ_1'
            and set (op_1 # ψ_1') ⊆ set ?ops
            and length ψ_1' = k
            using calculation(5) Suc(2)
      }
unfolding length-Suc-conv
by blast
moreover obtain op_2 \psi_2' where \psi_2 = op_2 \# \psi_2'
and set (op_2 \# \psi_2') \subseteq set ?ops
and length \psi_2' = k
using calculation(6) Suc(3)
unfolding length-Suc-conv
by blast
moreover have set \psi_1' \subseteq set ?ops and set \psi_2' \subseteq set ?ops
using calculation(8, 11)
by auto+
moreover have \psi_1' \in \?P k and \psi_2' \in \?P k
using calculation(9, 12, 13, 14)
by fast+
moreover have \varphi_P \psi_1' = \varphi_P \psi_2'
using Suc.prems(3) calculation(7, 10)
by fastforce
moreover have \psi_1' = \psi_2'
using Suc.IH[of \psi_1' \psi_2', OF calculation(15, 16, 17)]
by simp
moreover have \varphi_P \psi_1 = (\varphi_O \Psi op_1) \# \varphi_P \psi_1'
and \varphi_P \psi_2 = (\varphi_O \Psi op_2) \# \varphi_P \psi_2'
using Suc.prems(3) calculation(7, 10)
by fastforce+
moreover have (\varphi_O \Psi op_1) = (\varphi_O \Psi op_2)
using Suc.prems(3) calculation(17, 19, 20)
by simp
moreover have op_1 = op_2
using sasp-op-to-strips-injective[OF calculation(21)].
ultimately show \?case
by argo
qed

thus \?thesis
unfolding inj-on-def
by blast
qed

private corollary sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b:
assumes is-valid-problem-sas-plus \Psi
shows inj-on (\lambda \psi. [\varphi_O \Psi op. \psi]) (bounded-solution-set-sas-plus' \Psi k)
proof -
let \?ops = sas-plus-problem.operators-of \Psi
and \?varphi_P = \lambda \psi. [\varphi_O \Psi op. \psi]
{ fix \psi
assume \psi \in bounded-solution-set-sas-plus' \Psi k
then have set \psi \subseteq set \?ops
and length \psi = k

Let-def

list-all-iff ListMem-iff
by fast+

hence ψ ∈ bounded-plan-set ?ops k
by blast

}\n
hence bounded-solution-set-sas-plus′ Ψ k ⊆ bounded-plan-set ?ops k
by blast

moreover have inj-on ?φP (bounded-plan-set ?ops k)
using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a[OF
assms(1)]

ultimately show ?thesis
Ψ k]
by fast

qed

— Show that mapping plan transformation λψ. [φO Ψ op. op ← ψ] over the solution
set for a given SAS+ problem yields the solution set for the induced STRIPS
problem.

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c:
assumes is-valid-problem-sas-plus Ψ
shows (λψ. [φO Ψ op. op ← ψ]) ′ (bounded-solution-set-sas-plus′ Ψ k)
= bounded-solution-set-strips′ (φ Ψ) k

proof —

let ?Π = ψ

and ?φP = λψ. [φO Ψ op. op ← ψ]

let ?Solk = bounded-solution-set-sas-plus′ Ψ k

and ?Sol′ k = bounded-solution-set-strips′ ?Π k

{ 
assume ?φP ′ ?Solk ≠ ?Sol′ k
then consider (A) ∃π ∈ ?φP ′ ?Solk, π ∉ ?Sol′ k

| (B) ∃π ∈ ?Sol′ k, π ∉ ?φP ′ ?Solk

by blast

hence False

proof (cases)

case A

moreover obtain π where π ∈ ?φP ′ ?Solk and π ∉ ?Sol′ k

using calculation
by blast

moreover obtain ψ where length ψ = k

and SAS-Plus-Semantics.is-serial-solution-for-problem Ψ ψ

and π = ?φP ψ

using calculation(2)

unfolding bounded-solution-set-sas-plus′-def

by blast
moreover have length $\pi = k$ and \textit{STRIPS-Semantics.is-serial-solution-for-problem} \ref{Pi}\, $\pi$

subgoal
using calculation(4, 6) by auto

subgoal
using serial-sas-plus-equivalent-to-serial-strips
assms(1) calculation(5) calculation(6)
by blast

moreover have $\pi \in \text{Sol}_k'$
unfolding bounded-solution-set-strips'-def
using calculation(7, 8)
by simp

ultimately show \ref{thesis}
by fast

next

moreover obtain $\pi$ where $\pi \in \text{Sol}_k'$ and $\pi \not\in \varphi_P \land \text{Sol}_k$

using calculation
by blast

moreover have \textit{STRIPS-Semantics.is-serial-solution-for-problem} \ref{Pi}\, $\pi$
and length $\pi = k$

using calculation(2)

moreover have \ref{thesis}

Construct the counter example $\psi \equiv [\varphi_O^{-1} \, \ref{Pi} \, \text{op. op} \leftarrow \pi]$ and show that $\psi \in \text{Sol}_k$ as well as $\varphi_P \, \psi = \pi$ hence $\pi \in \varphi_P \land \text{Sol}_k$

moreover have length $[\varphi_O^{-1} \, \psi \, \text{op. op} \leftarrow \pi] = k$
and \textit{SAS-Plus-Semantics.is-serial-solution-for-problem} $\psi \varphi_O^{-1} \Psi \, \text{op. op}$

moreover have $[\varphi_O^{-1} \, \psi \, \text{op. op} \leftarrow \pi] \in \text{Sol}_k$

unfolding bounded-solution-set-sas-plus'-def
using calculation(6, 7)
by blast

moreover { have $\forall \, \text{op} \in \text{set. op} \in \text{set.} (\text{\ref{Pi}})$

using calculation(4)

unfolding \textit{STRIPS-Semantics.is-serial-solution-for-problem-def list-all-iff}
ListMem-iff
by simp
hence $\varphi_P [\varphi_O^{-1} \Psi \ op. \ op \leftarrow \pi] = \pi$

proof (induction $\pi$)
  case (Cons $op \ \pi$)
    moreover have $\varphi_P [\varphi_O^{-1} \Psi \ op. \ op \leftarrow \pi]$
    \hspace{1em} = $(\varphi_O \Psi (\varphi_O^{-1} \Psi \ op)) \# \varphi_P [\varphi_O^{-1} \Psi \ op. \ op \leftarrow \pi]$
    \hspace{1em} by simp
    moreover have $op \in \text{set} \ ((?\Pi)_{O})$
    \hspace{1em} using $\text{Cons.prem}$
    \hspace{1em} by simp
    moreover have $(\varphi_O \Psi (\varphi_O^{-1} \Psi \ op)) = op$
    \hspace{1em} using $\text{strips-operator-inverse-is}$ $\text{OF assms(1) calculation(4)}$.
    moreover have $\varphi_P [\varphi_O^{-1} \Psi \ op. \ op \leftarrow \pi] = \pi$
    \hspace{1em} using $\text{Cons.IH Cons.prem}$
    \hspace{1em} by auto
    ultimately show $?\text{case}$
    \hspace{1em} by argo
  qed simp

moreover have $\pi \in \varphi_P \cdot ?\text{Sol}_k$
  \hspace{1em} using $\text{calculation(8, 9)}$
  \hspace{1em} by force
  ultimately show $?\text{thesis}$
  \hspace{1em} by blast
  qed

thus $?\text{thesis}$
  \hspace{1em} by blast
  qed

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-d:
  assumes is-valid-problem-sas-plus $\Psi$
  shows card (bounded-solution-set-sas-plus' $\Psi \ k$) $\leq$ card (bounded-solution-set-strips' $(\varphi \Psi) \ k$)
  \hspace{1em} proof
    \hspace{1em} let $?\Pi = \varphi \Psi$
    \hspace{1em} and $\varphi_P = \lambda\psi. \ [\varphi_O \Psi \ op. \ op \leftarrow \psi]$
    \hspace{1em} let $?\text{Sol}_k = \text{bounded-solution-set-sas-plus}' \Psi \ k$
    \hspace{1em} and $?\text{Sol}_k' = \text{bounded-solution-set-strips}' ?\Pi \ k$
    \hspace{1em} have card ($?\varphi_P \cdot ?\text{Sol}_k = \text{card} \ (?\text{Sol}_k)$
    \hspace{1em} using $\text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b}$ $\text{OF assms(1)}$
    \hspace{1em} card-image
    \hspace{1em} by blast
    moreover have $\varphi_P \cdot ?\text{Sol}_k = ?\text{Sol}_k'$
    \hspace{1em} using $\text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c}$ $\text{OF assms(1)}$.
    ultimately show $?\text{thesis}$
    \hspace{1em} by simp
  qed
— The set of fixed length plans with operators in a given operator set is finite.

**lemma** bounded-plan-set-finite:

**shows** finite { π. set π ⊆ set ops ∧ length π = k }

**proof** (induction k)

**case** (Suc k)

let ?P = { π. set π ⊆ set ops ∧ length π = k }

and ?P’ = { π. set π ⊆ set ops ∧ length π = Suc k }

let ?P'' = (⋃ op ∈ set ops. (⋃ π ∈ ?P. { op # π } ))

{ have ∀ op π. finite { op # π }
  by simp
  then have ∀ op. finite (⋃ π ∈ ?P. { op # π })
  using finite-UN[of ?P] Suc
  by blast
  hence finite ?P''
  using finite-UN[of set ops]
  by blast }

moreover {
  { fix π
    assume π ∈ ?P’
    moreover have set π ⊆ set ops
      and length π = Suc k
      using calculation
      by simp+
    moreover obtain op π’ where π = op # π’
      using calculation (3)
      unfolding length-Suc-conv
      by fast
    moreover have set π’ ⊆ set ops and op ∈ set ops
      using calculation(2, 4)
      by simp+
    moreover have length π’ = k
      using calculation(3, 4)
      by auto
    moreover have π’ ∈ ?P
      using calculation(5, 7)
      by blast
    ultimately have π ∈ ?P''
    by blast
  }
  hence ?P’ ⊆ ?P''
  by blast
}

ultimately show ?case
  by blast

173
— The set of fixed length SAS+ solutions are subsets of the set of plans with fixed length and therefore also finite.

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a:
  assumes is-valid-problem-sas-plus \( \Psi \)
  shows finite (bounded-solution-set-sas-plus' \( \Psi \) \( k \))
proof
  let \( ?Ops = \text{set } ((\Psi)_{O+}) \)
  let \( ?Sol_k = \text{bounded-solution-set-sas-plus' } \Psi \ k \)
  and \( ?P_k = \{ \pi. \text{ set } \pi \subseteq ?Ops \land \text{length } \pi = k \} \)
  { 
    fix \( \psi \)
    assume \( \psi \in ?Sol_k \)
    then have length \( \psi = k \) and set \( \psi \subseteq ?Ops \)
      unfolding bounded-solution-set-sas-plus'-def
      SAS-Plus-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
      by fastforce+
    hence \( \psi \in ?P_k \)
      by blast
  }
  then have \( ?Sol_k \subseteq ?P_k \)
    by force
  thus \( \text{thesis} \)
    using bounded-plan-set-finite rev-finite-subset[of \( ?P_k \) \( ?Sol_k \)]
    by auto
qed

— The set of fixed length STRIPS solutions are subsets of the set of plans with fixed length and therefore also finite.

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b:
  assumes is-valid-problem-sas-plus \( \Psi \)
  shows finite (bounded-solution-set-strips' \( \varphi \) \( \Psi \) \( k \))
proof
  let \( ?\Pi = \varphi \Psi \)
  let \( ?Ops = \text{set } ((?\Pi)_O) \)
  let \( ?Sol_k = \text{bounded-solution-set-strips' } \varphi \ k \)
  and \( ?P_k = \{ \pi. \text{ set } \pi \subseteq ?Ops \land \text{length } \pi = k \} \)
  { 
    fix \( \pi \)
    assume \( \pi \in ?Sol_k \)
    then have length \( \pi = k \) and set \( \pi \subseteq ?Ops \)
      unfolding bounded-solution-set-strips'-def
      STRIPS-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
      by fastforce+
    hence \( \pi \in ?P_k \)
      by blast
  }
  then have \( ?Sol_k \subseteq ?P_k \)
by force
thus ?thesis
unfolding state-to-strips-state-def
\[ SAS-Plus-STRIPS.state-to-strips-state-def \text{ operators-of-def} \]
by blast
qed

With the results on the equivalence of SAS+ and STRIPS solutions, we can now show that given problems in both formalisms, the solution sets have the same size. This is the property required by the definition of planning formalism equivalence presented earlier in theorem ?? (??) and thus end up with the desired equivalence result.

The proof uses the finiteness and disjunctiveness of the solution sets for either problem to be able to equivalently transform the set cardinality over the union of sets of solutions with bounded lengths into a sum over the cardinality of the sets of solutions with bounded length. Moreover, since we know that for each SAS+ solution with a given length an equivalent STRIPS solution exists in the solution set of the transformed problem with the same length, both sets must have the same cardinality.

Hence the cardinality of the SAS+ solution set over all lengths up to a given upper bound \( N \) has the same size as the solution set of the corresponding STRIPS problem over all length up to a given upper bound \( N \).

theorem
assumes is-valid-problem-sas-plus \( \Psi \)
shows \( \text{card} \ (\text{bounded-solution-set-sas-plus} \ \Psi \ N) \)
= \( \text{card} \ (\text{bounded-solution-set-strips} \ (\varphi \ \Psi) \ N) \)
proof –
let \( \pi = \varphi \ \Psi \)
and \( \ \ \ ?R = \{0..N\} \)
— Due to the disjoint nature of the bounded solution sets for fixed plan length for different lengths, we can sum the individual set cardinality to obtain the cardinality of the overall SAS+ resp. STRIPS solution sets.

have finite-\( R \): finite \?R
by simp
moreover {
have \( \forall \ k \in \ ?R, \ \text{finite} \ (\text{bounded-solution-set-sas-plus}' \ \Psi \ k) \)
using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a[OF
assms(1)]...
moreover have \( \forall \ j \in \ ?R, \ \forall \ k \in \ ?R, \ j \neq k \)
\[ \rightarrow \text{bounded-solution-set-sas-plus}' \ \Psi \ j \]
\[ \cap \text{bounded-solution-set-sas-plus}' \ \Psi \ k = \{\} \]
unfolding bounded-solution-set-sas-plus'-def
by blast
ultimately have \( \text{card} \ (\text{bounded-solution-set-sas-plus} \ \Psi \ N) \)
= (∑ k ∈ ?R. card (bounded-solution-set-sas-plus′ Ψ k))
using card-UN-disjoint
by blast
}
moreover {
  have ∀ k ∈ ?R. finite (bounded-solution-set-strips′ ?Π k)
  using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b[OF assistms(1)]
moreover have ∀ j ∈ ?R. ∀ k ∈ ?R. j ≠ k
  → bounded-solution-set-strips′ ?Π j
  ∩ bounded-solution-set-strips′ ?Π k = {}
  unfolding bounded-solution-set-strips′-def
  by blast
ultimately have card (bounded-solution-set-strips ?Π N)
  = (∑ k ∈ ?R. card (bounded-solution-set-strips′ ?Π k))
using card-UN-disjoint
by blast
}
moreover {
  fix k
  have card (bounded-solution-set-sas-plus′ Ψ k)
  = card ((∀ψ. [ϕ ° op ← ψ])
  ∪ bounded-solution-set-sas-plus′ Ψ k)
  using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b[OF assistms]
  card-image[symmetric]
  by blast
  hence card (bounded-solution-set-sas-plus′ Ψ k)
  = card (bounded-solution-set-strips′ ?Π k)
  using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c[OF assistms]
  by presburger
}
ultimately show ?thesis
by presburger
qed

dep
end
dep
theory SAT-Plan-Base
imports List—Index.List-Index
  Propositional-Proof-Systems.Formulas
  STRIPS-Semantics
  Map-Supplement List-Supplement
The Basic SATPlan Encoding

We now move on to the formalization of the basic SATPlan encoding (see ??).

The two major results that we will obtain here are the soundness and completeness result outlined in ?? in ??.

Let in the following \( \Phi \equiv \text{encode-to-sat} \Pi t \) denote the SATPlan encoding for a STRIPS problem \( \Pi \) and makespan \( t \). Let \( k < t \) and \( I \equiv (\Pi)_I \) be the initial state of \( \Pi \), \( G \equiv (\Pi)_G \) be its goal state, \( V \equiv (\Pi)_V \) its variable set, and \( O \equiv (\Pi)_O \) its operator set.

7.1 Encoding Function Definitions

Since the SATPlan encoding uses propositional variables for both operators and state variables of the problem as well as time points, we define a datatype using separate constructors — \( \text{State} k n \) for state variables resp. \( \text{Operator} k n \) for operator activation—to facilitate case distinction. The natural number values store the time index resp. the indexes of the variable or operator within their lists in the problem representation.

\[
\text{datatype} \quad \text{sat-plan-variable} = \\
\quad \text{State} \; \text{nat} \; \text{nat} \\
\quad \text{Operator} \; \text{nat} \; \text{nat}
\]

A SATPlan formula is a regular propositional formula over SATPlan variables. We add a type synonym to improve readability.

\[
\text{type-synonym} \quad \text{sat-plan-formula} = \text{sat-plan-variable} \; \text{formula}
\]

We now continue with the concrete definitions used in the implementation of the SATPlan encoding. State variables are encoded as literals over SATPlan variables using the \( \text{State} \) constructor of .
definition encode-state-variable :: nat ⇒ nat ⇒ bool option ⇒ sat-plan-variable formula
where encode-state-variable t k v ≡ case v of
  Some True ⇒ Atom (State t k)
  | Some False ⇒ ¬ (Atom (State t k))

The initial state encoding (definition ??) is a conjunction of state variable encodings
\( A \equiv encode-state-variable 0 n b \) with \( n \equiv index vs v \) and \( b \equiv I v = Some True \) for all \( v \in V \). As we can see below, the same function but substituting the initial state with the goal state and zero with the makespan \( t \) produces the goal state encoding (??). Note that both functions construct a conjunction of clauses \( A \lor \bot \) for which it is easy to show that we can normalize to conjunctive normal form (CNF).

definition encode-initial-state :: 'variable strips-problem ⇒ sat-plan-variable formula (ΦI - 99)
where encode-initial-state Π ≡ let I = initial-of Π ; vs = variables-of Π
  in \( \land \) (map (λv. encode-state-variable 0 (index vs v) (I v) \lor \bot)
  (filter (λv. I v \neq None) vs))

definition encode-goal-state :: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula (ΦG - 99)
where encode-goal-state Π t ≡ let
  vs = variables-of Π ; G = goal-of Π
  in \( \land \) (map (λv. encode-state-variable t (index vs v) (G v) \lor \bot)
  (filter (λv. G v \neq None) vs))

Operator preconditions are encoded using activation-implies-precondition formulation as mentioned in ??: i.e. for each operator \( op \in O \) and \( p \in set (precondition-of op) \) we have to encode
\[ Atom \ (Operator \ k \ (index \ ops \ op)) \rightarrow Atom \ (State \ k \ (index \ vs \ v)) \]

We use the equivalent disjunction in the formalization to simplify conversion to CNF.

definition encode-operator-precondition :: 'variable strips-problem ⇒ nat ⇒ 'variable strips-operator ⇒ sat-plan-variable formula
where encode-operator-precondition Π t op ≡ let
  vs = variables-of Π ; ops = operators-of Π
  in \( \land \) (map (λv. ¬ (Atom (Operator t (index ops op))) \lor Atom (State t (index vs v))))
(precondition-of op))

**definition**  encode-all-operator-preconditions
:: 'variable strips-problem
   ⇒ 'variable strips-operator list
   ⇒ nat
   ⇒ sat-plan-variable formula
**where**  encode-all-operator-preconditions Π ops t ≡ let
            l = List.product [0..<t] ops
            in foldr (∧) (map (λ(t, op). encode-operator-precondition Π t op) l) (¬⊥)

Analogously to the operator precondition, add and delete effects of operators have to be implied by operator activation. That being said, we have to encode both positive and negative effects and the effect must be active at the following time point: i.e.

Atom (Operator k m) → Atom (State (Suc k) n)

for add effects respectively

Atom (Operator k m) → ¬Atom (State (Suc k) n)

for delete effects. We again encode the implications as their equivalent disjunctions in definition ??.

**definition**  encode-operator-effect
:: 'variable strips-problem
   ⇒ nat
   ⇒ 'variable strips-operator
   ⇒ sat-plan-variable formula
**where**  encode-operator-effect Π t op ≡ let
            vs = variables-of Π
            ; ops = operators-of Π
            in \{(map (λv.
                      ¬(Atom (Operator t (index ops op))))
                    ∨ Atom (State (Suc t) (index vs v)))
                (add-effects-of op)
                \@ map (λv.
                      ¬(Atom (Operator t (index ops op))))
                    ∨ ¬ (Atom (State (Suc t) (index vs v))))
                (delete-effects-of op))

**definition**  encode-all-operator-effects
:: 'variable strips-problem
   ⇒ 'variable strips-operator list
   ⇒ nat
   ⇒ sat-plan-variable formula
**where**  encode-all-operator-effects Π ops t ≡ let l = List.product [0..<t] ops

179
in foldr (\&) (map (\(\lambda (t, op).\ encode-operator-effect \Pi t\ op\)) l) \(\neg\perp\)

definition encode-operators
:: 'variable strips-problem \Rightarrow \nat \Rightarrow sat-plan-variable formula

where encode-operators \Pi t
\equiv let ops = operators-of \Pi
      in encode-all-operator-preconditions \Pi ops t \& encode-all-operator-effects \Pi ops t

Definitions ?? and ?? similarly encode the negative resp. positive transition frame axioms as disjunctions.

definition encode-negative-transition-frame-axiom
:: 'variable strips-problem
\Rightarrow \nat
\Rightarrow 'variable
\Rightarrow sat-plan-variable formula

where encode-negative-transition-frame-axiom \Pi t v
\equiv let vs = variables-of \Pi
    ; ops = operators-of \Pi
    ; deleting-operators = filter (\(\lambda op.\ ListMem v (delete-effects-of op))\) ops
    in \(\neg(\text{Atom}\ (\text{State}\ t (\text{index}\ vs\ v)))\)
      \lor (\text{Atom}\ (\text{State}\ (\text{Suc}\ t) (\text{index}\ vs\ v)))
      \lor \left(\bigvee (\text{map}\ (\lambda op.\ \text{Atom}\ (\text{Operator}\ t (\text{index}\ ops\ op))))\ \text{deleting-operators})\right)

definition encode-positive-transition-frame-axiom
:: 'variable strips-problem
\Rightarrow \nat
\Rightarrow 'variable
\Rightarrow sat-plan-variable formula

where encode-positive-transition-frame-axiom \Pi t v
\equiv let vs = variables-of \Pi
    ; ops = operators-of \Pi
    ; adding-operators = filter (\(\lambda op.\ ListMem v (add-effects-of op))\) ops
    in (\text{Atom}\ (\text{State}\ t (\text{index}\ vs\ v)))
      \lor (\neg (\text{Atom}\ (\text{State}\ (\text{Suc}\ t) (\text{index}\ vs\ v))))
      \lor \left(\bigvee (\text{map}\ (\lambda op.\ \text{Atom}\ (\text{Operator}\ t (\text{index}\ ops\ op))))\ \text{adding-operators})\right)

definition encode-all-frame-axioms
:: 'variable strips-problem \Rightarrow \nat \Rightarrow sat-plan-variable formula

where encode-all-frame-axioms \Pi t
\equiv let l = List.product [0..<t] (variables-of \Pi)
      in \(\bigwedge (\text{map}\ (\lambda (k, v).\ encode-negative-transition-frame-axiom \Pi v k\ l))\ l\)
      @ \text{map}\ (\lambda (k, v).\ encode-positive-transition-frame-axiom \Pi v k\ l\ l)

Finally, the basic SATPlan encoding is the conjunction of the initial state, goal state, operator and frame axiom encoding for all time steps. The functions and ?? take care of mapping the operator precondition, effect and frame

??Not shown.
axiom encoding over all possible combinations of time point and operators resp. time points, variables, and operators.

definition encode-problem (Φ - - 99)
where encode-problem Π t
≡ encode-initial-state Π
∧ (encode-operators Π t
∧ (encode-all-frame-axioms Π t
∧ (encode-goal-state Π t))))

7.2 Decoding Function Definitions

Decoding plans from a valuation \(A\) of a SATPlan encoding entails extracting all activated operators for all time points except the last one. We implement this by mapping over all \(k < t\) and extracting activated operators—i.e. operators for which the model valuates the respective operator encoding at time \(k\) to true—into a parallel operator (see definition ??).  

---

Note that due to the implementation based on lists, we have to address the problem of duplicate operator declarations in the operator list of the problem. Since \(\text{index } op = \text{index } op'\) for equal operators, the parallel operator obtained from \(\) will contain duplicates in case the problem’s operator list does. We therefore remove duplicates first using \(\text{remdups ops}\) and then filter out activated operators.

definition decode-plan' :: 'variable strips-problem
⇒ sat-plan-variable valuation
⇒ nat
⇒ 'variable strips-operator list
where decode-plan' Π A i
≡ let ops = operators-of Π
 ; vs = map (λop. Operator i (index ops op)) (remdups ops)
 in map (λv. case v of Operator - k ⇒ ops ! k) (filter A vs)

We decode maps over range \(0, \ldots, t - 1\) because the last operator takes effect in \(t\) and must therefore have been applied in step \(t - (1::a)\).

definition decode-plan :: 'variable strips-problem
⇒ sat-plan-variable valuation
⇒ nat
⇒ 'variable strips-parallel-plan (Φ⁻¹ - - 99)
where decode-plan Π A t ≡ map (decode-plan' Π A) [0..<t]

Similarly to the operator decoding, we can decode a state at time \(k\) from a valuation of of the SATPlan encoding \(A\) by constructing a map from list of assignments \((v, A (\text{State } k (\text{index } vs v)))\) for all \(v \in \mathcal{V}\).

definition decode-state-at

---

This is handled by function \(\text{decode_plan'}\) (not shown).
:: 'variable strips-problem
⇒ sat-plan-variable valuation
⇒ nat
⇒ 'variable strips-state (ΦS⁻¹ - - 99)
where decode-state-at II A k
≡ let
vs = variables-of II
; state-encoding-to-assignment = λv. (A (State k (index vs v)))
in map-of (map state-encoding-to-assignment vs)

We continue by setting up the context for the proofs of soundness and completeness.

definition encode-transitions :: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable
formula (ΦT-99) where
encode-transitions II t
≡ SAT-Plan-Base.encode-operators II t ∧
SAT-Plan-Base.encode-all-frame-axioms II t

— Immediately proof the sublocale proposition for strips in order to gain access to definitions and lemmas.

— Setup simp rules.
lemma [simp]:
encode-transitions II t
≡ SAT-Plan-Base.encode-operators II t ∧
SAT-Plan-Base.encode-all-frame-axioms II t

unfolding encode-problem-def encode-initial-state-def encode-transitions-def
encode-goal-state-def decode-plan-def decode-state-at-def
by simp+

context begin

lemma encode-state-variable-is-lit-plus-if:
assumes is-valid-problem-strips II
and v ∈ dom s
shows is-lit-plus (encode-state-variable k (index (strips-problem.variables-of II) v) (s v))
proof
have s v ≠ None
using is-valid-problem-strips-initial-of-dom assms(2)
by blast
then consider (s-of-v-is-some-true) s v = Some True
| (s-of-v-is-some-false) s v = Some False
by fastforce
thus ?thesis
unfolding encode-state-variable-def
by (cases, simp+)
qed
lemma is-cnf-encode-initial-state:
  assumes is-valid-problem-strips Π
  shows is-cnf (Φ I Π)
proof -
  let ?I = (Π) I
  and ?vs = strips-problem.variables-of Π
  let ?l = map (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥)
          (filter (λv. ?I v ≠ None) ?vs)
  { fix C
    assume c-in-set-l:C ∈ set ?l
    have set ?l = (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥) ' 
      set (filter (λv. ?I v ≠ None) ?vs)
    using set-map[of λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥ 
                   filter (λv. ?I v ≠ None) ?vs]
    by blast
    then have set ?l = (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥) ' 
      {v ∈ set ?vs. ?I v ≠ None}
    using set-filter[of λv. ?I v ≠ None ?vs]
    by argo
    then obtain v
      where c-is: C = encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥ 
      and v-in-set-vs: v ∈ set ?vs 
      and I-of-v-is-not-None: ?I v ≠ None 
      using c-in-set-l
      by auto
  }
  { have v ∈ dom ?I
    using I-of-v-is-not-None
    by blast
    moreover have is-lit-plus (encode-state-variable 0 (index ?vs v) (?I v)) 
      using encode-state-variable-is-lit-plus-if [OF - calculation] assms(1)
    by blast
    moreover have is-lit-plus ⊥ 
      by simp
    ultimately have is-disj C 
      using c-is 
      by force
  }
  hence is-cnf C
  unfolding encode-state-variable-def 
  using c-is 
  by fastforce
  }
  thus ?thesis
  unfolding encode-initial-state-def SAT-Plan-Base.encode-initial-state-def Let-def
  initial-of-def

183
using is-cnf-BigAnd[of ?l]
  by (smt is-cnf-BigAnd)
qed

lemma encode-goal-state-is-cnf:
  assumes is-valid-problem-strips II
  shows is-cnf (encode-goal-state II t)
proof –
let ?I = (Π)t
  and ?G = (Π)G
  and ?vs = strips-problem.variables-of II
let ?l = map (λv. encode-state-variable t (index ?vs v) (?G v) ∨ ⊥)
          (filter (λv. ?G v ≠ None) ?vs)
{
  fix C
  assume C ∈ set ?l
  moreover {  
    have set ?l = (λv. encode-state-variable t (index ?vs v) (?G v) ∨ ⊥)
                   ' set (filter (λv. ?G v ≠ None) ?vs)
      unfolding set-map
      by blast
    then have set ?l = { encode-state-variable t (index ?vs v) (?G v) ∨ ⊥} 
                      | v. v ∈ set ?vs ∧ ?G v ≠ None 
      by auto
  }
  moreover obtain v where C-is: C = encode-state-variable t (index ?vs v)
   (?G v) ∨ ⊥
    and v ∈ set ?vs
    and G-of-v-is-not-None: ?G v ≠ None
    using calculation(1)
    by auto
  moreover {  
    have v ∈ dom ?G
      using G-of-v-is-not-None
      by blast
    moreover have is-lit-plus (encode-state-variable t (index ?vs v) (?G v))
      using assms(1) calculation
      by (simp add: encode-state-variable-is-lit-plus-if)
    moreover have is-lit-plus ⊥
      by simp
    ultimately have is-disj C
      unfolding C-is
      by force
  }
  ultimately have is-cnf C
    by simp
}
thus \( \text{thesis} \)
unfolding encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def
using is-cnf-BigAnd[of \(?l\)]
by simp

qed

private lemma encode-operator-precondition-is-cnf:
is-cnf (encode-operator-precondition II k op)
proof ~
  let \( ?\text{vs} = \text{strips-problem.variables-of II } \)
  and \( ?\text{ops} = \text{strips-problem.operators-of II } \)
  let \( ?l = \text{map (\( \lambda v. \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))) \)\)}}\)
  (precondition-of op)
  {
    have set \( ?l = \{ \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))\)\}}\)
      using set-map
      by force
    then have set \( ?l = \{ \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))\)\}}\)
      using setcompr-eq-image[of \( \lambda v. \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))\)\)
      using setcompr-eq-image[of \( \lambda v. \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))\)\)
      by simp
  } note set-l-is = this
  { fix \( C \)
    assume \( C \in \text{set } ?l \)
    then obtain \( v \)
      where \( v \in \text{set (precondition-of op)} \)
      and \( C = \neg(\text{Atom (}\text{Operator k (index } ?\text{ops op)}) \lor \text{Atom (}\text{State k (index } ?\text{vs v}))\)\)
      using set-l-is
      by blast
    hence is-cnf \( C \)
      by simp
  }
thus \( \text{thesis} \)
unfolding encode-operator-precondition-def
using is-cnf-BigAnd[of \(?l\)]
by meson

qed

private lemma set-map-operator-precondition[simp]:
set (map (\( \lambda (k, \text{op}). \text{encode-operator-precondition II k op} \)) (List.product [0..<t]\( \text{ops})\))

185
\begin{verbatim}
proof -
let ?l' = List.product [0..<t] ops
let ?fs = map (λ(k, op). encode-operator-precondition II k op) ?l'
have set-l'-is: set ?l' = {0..<t} × set ops
  by simp
moreover {
  have set ?fs = (λ(k, op). encode-operator-precondition II k op)
    · ( {0..<t} × set ops)
    using set-map set-l'-is
    by simp
  also have ... = { encode-operator-precondition II k op | k op. (k, op) ∈ {0..<t}
  × set ops}
    using setcompr-eq-image
    by fast
finally have set ?fs = { encode-operator-precondition II k op
  | k op. (k, op) ∈ {0..<t} } by blast
}
thus ?thesis
  by blast
qed

private lemma is-cnf-encode-all-operator-preconditions:
  is-cnf (encode-all-operator-preconditions II (strips-problem.operators-of II) t)
proof -
let ?l' = List.product [0..<t] (strips-problem.operators-of II)
let ?fs = map (λ(k, op). encode-operator-precondition II k op) ?l'
have ∀f ∈ set ?fs. is-cnf f
  using encode-operator-precondition-is-cnf
  by fastforce
thus ?thesis
unfolding encode-all-operator-preconditions-def
using is-cnf-foldr-and-if[of ?fs]
by presburger
qed

private lemma set-map-or[simp]:
  set (map (λv. A v ∨ B v) vs) = { A v ∨ B v | v. v ∈ set vs }
proof -
let ?l = map (λv. A v ∨ B v) vs
have set ?l = (λv. A v ∨ B v) ' set vs
  using set-map
  by force
thus ?thesis
  using setcompr-eq-image
  by auto
\end{verbatim}
qed

private lemma encode-operator-effects-is-cnf-i:
is-cnf (\(\bigwedge\) (map (\(\lambda\) v. (\(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op))))) 
\(\lor\) Atom (State (Suc t) (index (strips-problem.variables-of II) v)))) (add-effects-of op))
proof –
let ?fs = map (\(\lambda\) v. \(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op)))) 
\(\lor\) Atom (State (Suc t) (index (strips-problem.variables-of II) v))) (add-effects-of op)
{ 
fix C
assume C \(\in\) set ?fs
then obtain v 
  where v \(\in\) set (add-effects-of op)
  and C = \(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op))) 
  \(\lor\) Atom (State (Suc t) (index (strips-problem.variables-of II) v))
  by auto
hence is-cnf C
  by fastforce
} thus ?thesis
  using is-cnf-BigAnd
by blast
qed

private lemma encode-operator-effects-is-cnf-ii:
is-cnf (\(\bigwedge\) (map (\(\lambda\) v. \(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op))))) 
\(\lor\) \(\neg\) (Atom (State (Suc t) (index (strips-problem.variables-of II) v)))) (delete-effects-of op))
proof –
let ?fs = map (\(\lambda\) v. \(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op)))) 
\(\lor\) \(\neg\) (Atom (State (Suc t) (index (strips-problem.variables-of II) v)))) (delete-effects-of op)
{ 
fix C
assume C \(\in\) set ?fs
then obtain v 
  where v \(\in\) set (delete-effects-of op)
  and C = \(\neg\) (Atom (Operator t (index (strips-problem.operators-of II) op))) 
  \(\lor\) \(\neg\) (Atom (State (Suc t) (index (strips-problem.variables-of II) v)))
  by auto
hence is-cnf C
  by fastforce
}
thus \(\text{?thesis}\)
using \(\text{is-cnfs}\)-\(\text{BigAnd}\)
by blast
qed

private lemma \(\text{encode-operator-effect-is-cnfs}\):
shows \(\text{is-cnfs} (\text{encode-operator-effect} \Pi t \operatorname{op})\)
proof –
let \(\operatorname{ops} = \text{strips-problem.operators-of} \Pi\)
and \(\operatorname{vs} = \text{strips-problem.variables-of} \Pi\)
let \(\operatorname{fs} = \text{map}(\lambda \operatorname{v}. \neg(\text{Atom} (\text{Operator} t (\text{index} \operatorname{ops} \operatorname{op})) ∨ \text{Atom} (\text{State} (\text{Suc} t) (\text{index} \operatorname{vs} \operatorname{v}))))\)
\(\text{(add-effects-of}\ \operatorname{op})\)
and \(\operatorname{fs}' = \text{map}(\lambda \operatorname{v}. \neg(\text{Atom} (\text{Operator} t (\text{index} \operatorname{ops} \operatorname{op})) ∨ \neg(\text{Atom} (\text{State} (\text{Suc} t) (\text{index} \operatorname{vs} \operatorname{v})))))\)
\(\text{(delete-effects-of}\ \operatorname{op})\)
have \(\text{encode-operator-effect} \Pi t \operatorname{op} = \bigwedge (\operatorname{fs} @ \operatorname{fs'})\)
unfolding \(\text{encode-operator-effect-def}[\Pi t \operatorname{op}]\)
by metis
moreover {
have \(\forall f \in \operatorname{set} \operatorname{fs}. \text{is-cnfs} f \forall f' \in \operatorname{set} \operatorname{fs}'. \text{is-cnfs} f'\)
using \(\text{encode-operator-effects-is-cnfs-ii}[\Pi t \operatorname{op}]\)
by (simp+)

hence \(\forall f \in \operatorname{set} (\operatorname{fs} @ \operatorname{fs'})\). \text{is-cnfs} f\)
by auto
}
ultimately show \(\text{?thesis}\)
using \(\text{is-cnfs-BigAnd}[\Pi t \operatorname{op}]\)
by presburger
qed

private lemma \(\text{set-map-encode-operator-effect[simp]}\):
set \(\text{(map}(\lambda (t, \operatorname{op}). \text{encode-operator-effect} \Pi t \operatorname{op}) \ (\text{List.product} [0..<t])\)
\(\text{(strips-problem.operators-of} \Pi))\))
\(\{ \text{encode-operator-effect} \Pi k \operatorname{op} \mid k \operatorname{op}. (k, \operatorname{op}) \in \{(0..<t) \times \operatorname{set} \text{(strips-problem.operators-of} \Pi)\} \}\)
proof –
let \(\operatorname{ops} = \text{strips-problem.operators-of} \Pi\)
and \(\operatorname{vs} = \text{strips-problem.variables-of} \Pi\)
let \(\operatorname{fs} = \text{map}(\lambda (t, \operatorname{op}). \text{encode-operator-effect} \Pi t \operatorname{op}) \ (\text{List.product} [0..<t])\)
\(\text{?ops}\)
have \(\text{set} \operatorname{fs} = (\lambda (t, \operatorname{op}). \text{encode-operator-effect} \Pi t \operatorname{op}) \ (\{0..<t\} \times \text{set} \operatorname{ops})\)
unfolding \(\text{encode-operator-effect-def}[\Pi t]\)
by force
thus \(\text{?thesis}\)
using \(\text{setcompr-eq-image}[\lambda (t, \operatorname{op}). \text{encode-operator-effect} \Pi t \operatorname{op}\]
\(\lambda (k, \operatorname{op}). (k, \operatorname{op}) \in \{0..<t\} \times \text{set} \operatorname{ops}\)
by force
qed

private lemma encode-all-operator-effects-is-cnf:
  assumes is-valid-problem-strips II
  shows is-cnf (encode-all-operator-effects II (strips-problem.operators-of II) t)
proof —
  let ?ops = strips-problem.operators-of II
  let ?l = List.product [0..<t] ?ops
  let ?fs = map (λ(t, op). encode-operator-effect II t op) ?l
  have ∀ f ∈ set ?fs. is-cnf f
    using encode-operator-effect-is-cnf
    by force
  thus ?thesis
    unfolding encode-all-operator-effects-def
    using is-cnf-foldr-and-if[of ?fs]
    by presburger
qed

lemma encode-operators-is-cnf:
  assumes is-valid-problem-strips II
  shows is-cnf (encode-operators II t)
unfolding encode-operators-def
using is-cnf-encode-all-operator-preconditions[of II t]
  encode-all-operator-effects-is-cnf[of assms, of t]
  encode-all-operator-effects II (strips-problem.operators-of II) t
by meson
— Simp flag alone did not do it, so we have to assign a name to this lemma as well.

private lemma set-map-to-operator-atom[simp]:
  set (map (λop. Atom (Operator t (index (strips-problem.operators-of II) op)))
    (filter (λop. ListMem v vs) (strips-problem.operators-of II)))
  = \{ Atom (Operator t (index (strips-problem.operators-of II) op)) | op. op ∈ set (strips-problem.operators-of II) ∧ v ∈ set vs \}
proof —
  let ?ops = strips-problem.operators-of II
  { have set (filter (λop. ListMem v vs) ?ops)
      = \{ op. op ∈ set ?ops. ListMem v vs \}
      using set-filter
      by force
      then have set (filter (λop. ListMem v vs) ?ops)
        = \{ op. op ∈ set ?ops ∧ v ∈ set vs \}
        using ListMem-iff[of v]
        by blast
  }
  then have set (map (λop. Atom (Operator t (index ?ops op))))
(filter (\op. ListMem v vs ?ops))
= (\op. Atom (Operator t (index ?ops op))) \{ op \in set ?ops. v \in set vs \}
using set-map[of \op. Atom (Operator t (index ?ops op))]
by presburger
thus ?thesis
by blast
qed

lemma is-disj-big-or-if:
  assumes \forall f \in set fs. is-lit-plus f
  shows is-disj \bigvee fs
  using assms
proof (induction fs)
case (Cons f fs)
  have is-lit-plus f
  using Cons.prems
  by simp
moreover have is-disj \bigvee fs
  using Cons
  by fastforce
ultimately show ?case
  by simp
qed simp

lemma is-cnf-encode-negative-transition-frame-axiom:
  shows is-cnf (encode-negative-transition-frame-axiom \Pi t v)
proof –
let ?vs = strips-problem.variables-of \Pi
and ?ops = strips-problem.operators-of \Pi
let ?deleting = filter (\op. ListMem v (delete-effects-of op)) ?ops
let ?fs = map (\op. Atom (Operator t (index ?ops op))) ?deleting
and ?A = (~\(Atom (State t (index ?vs v))))
and ?B = Atom (State (Suc t) (index ?vs v))
{
  fix f
  assume f \in set ?fs
  then obtain op
    where op \in set ?ops
    and v \in set (delete-effects-of op)
    and f = Atom (Operator t (index ?ops op))
    using set-map-to-operator-atom[of t \Pi v]
    by fastforce
  hence is-lit-plus f
  by simp
  } note nb = this
  {
  have is-disj \bigvee ?fs

using is-disj-big-or-if nb
by blast
then have is-disj (?B ∨ ⋁ ?fs)
by force
then have is-disj (?A ∨ (?B ∨ ⋁ ?fs))
by fastforce
hence is-cnf (?A ∨ (?B ∨ ⋁ ?fs))
by fastforce
}
thus ?thesis
unfolding encode-negative-transition-frame-axiom-def
by meson
qed

lemma is-cnf-encode-positive-transition-frame-axiom:
shows is-cnf (encode-positive-transition-frame-axiom Π t v)
proof –
let ?vs = strips-problem.variables-of Π
and ?ops = strips-problem.operators-of Π
let ?adding = filter (λop. ListMem v (add-effects-of op) ?ops)
let ?fs = map (λop. Atom (Operator t (index ?ops op))) ?adding
and ?A = Atom (State t (index ?vs v))
and ?B = ¬(Atom (State (Suc t) (index ?vs v)))
{
fix f
assume f ∈ set ?fs
then obtain op
where op ∈ set ?ops
and v ∈ set (add-effects-of op)
and f = Atom (Operator t (index ?ops op))
using set-map-to-operator-atom[of t Π v]
by fastforce
hence is-lit-plus f
by simp
} note nb = this
{
have is-disj ⋁ ?fs
using is-disj-big-or-if nb
by blast
then have is-disj (?B ∨ ⋁ ?fs)
by force
then have is-disj (?A ∨ (?B ∨ ⋁ ?fs))
by fastforce
hence is-cnf (?A ∨ (?B ∨ ⋁ ?fs))
by fastforce
}
thus ?thesis
unfolding encode-positive-transition-frame-axiom-def
by meson

qed

private lemma encode-all-frame-axioms-set[simp]:

set (map (λ(k, v). encode-negative-transition-frame-axiom Π k v) (List.product [0..<t] (strips-problem.variables-of Π)) @ (map (λ(k, v). encode-positive-transition-frame-axiom Π k v) (List.product [0..<t] (strips-problem.variables-of Π)))) = \{ encode-negative-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set (strips-problem.variables-of Π)) \} ∪ \{ encode-positive-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set (strips-problem.variables-of Π)) \}

proof –

let ?l = List.product [0..<t] (strips-problem.variables-of Π)
and ?fs = map (λ(k, v). encode-negative-transition-frame-axiom Π k v) ?l
@ (map (λ(k, v). encode-positive-transition-frame-axiom Π k v) ?l)
and ?vs = strips-problem.variables-of Π

have set-l-is: set ?l = {0..<t} × set ?vs
  by simp
  have set ?fs = ?A ∪ ?B
  using set-append
  by force
  moreover have ?A = \{ encode-negative-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set ?vs) \}
  using set-l-is setcompr-eq-image[of λ(k, v). encode-negative-transition-frame-axiom Π k v]
  by fast
  moreover have ?B = \{ encode-positive-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set ?vs) \}
  using set-l-is setcompr-eq-image[of λ(k, v). encode-positive-transition-frame-axiom Π k v]
  by fast
ultimately show ?thesis
  by argo

qed

lemma encode-frame-axioms-is-cnf:

shows is-cnf (encode-all-frame-axioms Π t)

proof –

let ?l = List.product [0..<t] (strips-problem.variables-of Π)
and ?vs = strips-problem.variables-of Π
let ?A = \{ encode-negative-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set ?vs) \}
and ?B = \{ encode-positive-transition-frame-axiom Π k v | k v. (k, v) ∈ ({0..<t} × set ?vs) \}
\( (k, v) \in (\{0..t\} \times \text{set } \text{?vs}) \)

and \( ?fs = \text{map } (\lambda (k, v). \text{encode-negative-transition-frame-axiom } \Pi ?l (\text{map } (\lambda (k, v). \text{encode-positive-transition-frame-axiom } \Pi ?l)) \)

{  
  fix \( f \)
  assume \( f \in \text{set } \text{?fs} \)
  
  then consider \( (f\text{-encodes-negative-frame-axiom}) f \in \text{?A} \)
  
  | \( (f\text{-encodes-positive-frame-axiom}) f \in \text{?B} \)
  
  by fastforce

  hence \( \text{is-cnf } f \)
  
  using \( \text{is-cnf-encode-negative-transition-frame-axiom} \)
  
  \( \text{is-cnf-encode-positive-transition-frame-axiom} \)
  
  by (smt mem-Collect-eq)

}  

thus \( ?\text{thesis} \)

unfolding \( \text{encode-all-frame-axioms-def} \)

using \( \text{is-cnf-BigAnd[of } \text{?fs} \)

by meson

qed

lemma \( \text{is-cnf-encode-problem}: \)

assumes \( \text{is-valid-problem-strips } \Pi \)

shows \( \text{is-cnf } (\Phi \Pi t) \)

proof –

have \( \text{is-cnf } (\Phi_I \Pi) \)

using \( \text{is-cnf-encode-initial-state } \text{assms} \)

by auto

moreover have \( \text{is-cnf } (\text{encode-goal-state } \Pi t) \)

using \( \text{encode-goal-state-is-cnf}[OF \text{assms}] \)

by simp

moreover have \( \text{is-cnf } (\text{encode-operators } \Pi t \land \text{encode-all-frame-axioms } \Pi t) \)

using \( \text{encode-operators-is-cnf}[OF \text{assms}] \text{ encode-frame-axioms-is-cnf} \)

unfolding \( \text{encode-transitions-def} \)

by simp

ultimately show \( ?\text{thesis} \)

unfolding \( \text{encode-problem-def SAT-Plan-Base.encode-problem-def} \)

\( \text{encode-transitions-def encode-initial-state-def[symmetric] encode-goal-state-def[symmetric]} \)

by simp

qed

lemma \( \text{encode-problem-has-model-then-also-partial-encodings}: \)

assumes \( \mathcal{A} \models \text{SAT-Plan-Base.encode-problem-def } \Pi t \)

shows \( \mathcal{A} \models \text{SAT-Plan-Base.encode-initial-state } \Pi \)

and \( \mathcal{A} \models \text{SAT-Plan-Base.encode-goal-state } \Pi \)

and \( \mathcal{A} \models \text{SAT-Plan-Base.encode-operators } \Pi t \)

and \( \mathcal{A} \models \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t \)

using \( \text{assms} \)

unfolding \( \text{SAT-Plan-Base.encode-problem-def} \)
by simp+

lemma cnf-of-encode-problem-structure:
  shows cnf (SAT-Plan-Base.encode-initial-state \Pi) 
    \subseteq cnf (SAT-Plan-Base.encode-problem \Pi t)
  and cnf (SAT-Plan-Base.encode-goal-state \Pi \Pi)
    \subseteq cnf (SAT-Plan-Base.encode-problem \Pi \Pi)
  and cnf (SAT-Plan-Base.encode-operators \Pi \Pi)
    \subseteq cnf (SAT-Plan-Base.encode-problem \Pi \Pi)
  and cnf (SAT-Plan-Base.encode-all-frame-axioms \Pi \Pi)
    \subseteq cnf (SAT-Plan-Base.encode-problem \Pi \Pi)

unfolding SAT-Plan-Base.encode-problem-def
  SAT-Plan-Base.encode-goal-state-def[of \Pi \Pi]
  SAT-Plan-Base.encode-operators-def[of \Pi \Pi]
  SAT-Plan-Base.encode-all-frame-axioms-def[of \Pi \Pi]

subgoal by auto
subgoal by force
subgoal by auto
subgoal by force

done

— A technical lemma which shows a simpler form of the CNF of the initial state encoding.

private lemma cnf-of-encode-initial-state-set-i:
  shows cnf \((\Phi_0, \Pi) = \bigcup \{ cnf (\text{encode-state-variable } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v)) | v \in \text{set } (\text{strips-problem.variables-of } \Pi) \land (\Pi I) v \neq \text{None} \} \)

proof –
let ?vs = \text{strips-problem.variables-of } \Pi
and ?I = \text{strips-problem.initial-of } \Pi
let ?ls = \text{map } (\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (\Pi I v) \lor \bot)
  (\text{filter } (\lambda v. \Pi v \neq \text{None} ) ?vs )
  \{ have cnf ' set ?ls = cnf ' (\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (\Pi I v) \lor \bot) 
    \{ set (\text{filter } (\lambda v. \Pi v \neq \text{None} ) ?vs )
      \text{using set-map[of lambda } \lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (\Pi I v) \lor \bot\}
      \text{by prebsburger}
    also have ... = (\lambda v. \text{cnf } (\text{encode-state-variable } 0 (\text{index } ?vs v) (\Pi I v) \lor \bot))
      \{ v \in \text{set } ?vs, \Pi v \neq \text{None } \}
      \text{using set-filter[of lambda } \lambda v. \Pi v \neq \text{None } ?vs\}
      \text{by auto}
    finally have cnf ' set ?ls = \{ cnf (\text{encode-state-variable } 0 (\text{index } ?vs v) (\Pi I v))\}
  \}
\}
\[
\begin{align*}
| v, v & \in \text{set } ?vs \land \neg I v \neq \text{None} \\
\text{using } \text{setcompr-eq-image[of } \lambda v. \text{cnf (encode-state-variable 0 (index } ?vs v) \\neg I v)]] \text{ by presburger} & \\
\text{}\}
\end{align*}
\]

moreover have \( \text{cnf (} \Phi I \Pi) = \bigcup (\text{cnf } \text{' set } ?ls) \)

unfolding \( \text{encode-initial-state-def SAT-Plan-Base, encode-initial-state-def} \)

using \( \text{cnf-BigAnd[of } ?ls] \)

by meson

ultimately show \( ?\text{thesis} \)

by auto

qed

— A simplification lemma for the above one.

corollary \( \text{cnf-of-encode-initial-state-set-ii:} \)

assumes \( \text{is-valid-problem-strips } \Pi \)

shows \( \text{cnf (} \Phi I \Pi) = (\bigcup v \in \text{set (strips-problem.variables-of } \Pi) \cdot (\{} \)

\( \text{literal-formula-to-literal (encode-state-variable 0 (index (strips-problem.variables-of } \Pi) v) \}

\( (\text{strips-problem.initial-of } \Pi v) \}) \}) \)

proof –

let \( ?vs = \text{strips-problem.variables-of } \Pi \)

and \( ?I = \text{strips-problem.initial-of } \Pi \)

have \( \text{nb1: } \{ v, v \in \text{set } ?vs \land \neg I v \neq \text{None } \} = \text{set } ?vs \)

using \( \text{is-valid-problem-strips-initial-of-dom assms(1)} \)

by auto

\{
\}

fix \( v \)

assume \( v \in \text{set } ?vs \)

then have \( \neg I v \neq \text{None} \)

using \( \text{is-valid-problem-strips-initial-of-dom assms(1)} \)

by auto

then consider \( (I-v-\text{is-Some-True}) ?I v = \text{Some True} \)

| \( (I-v-\text{is-Some-False}) ?I v = \text{Some False} \)

by fastforce

hence \( \text{cnf (encode-state-variable 0 (index } ?vs v) (?I v)) \)

\( = (\{} \text{literal-formula-to-literal (encode-state-variable 0 (index } ?vs v) (?I v)) \}) \)

unfolding \( \text{encode-state-variable-def} \)

by (cases, simp+)

} note \( \text{nb2 = this} \)

\{
\}

have \( \text{cnf (encode-state-variable 0 (index } ?vs v) (?I v)) \mid v, v \in \text{set } ?vs \land ?I v \neq \text{None } \)

\( = (\lambda v. \text{cnf (encode-state-variable 0 (index } ?vs v) (?I v))) \cdot \text{set } ?vs \)

using \( \text{setcompr-eq-image[of } \lambda v. \text{cnf (encode-state-variable 0 (index } ?vs v) (?I v)) \}

\lambda v. v \in \text{set } ?vs \land ?I v \neq \text{None} \) using \( \text{nb1} \)

195
by presburger

hence \{ cnf (encode-state-variable \theta (index \?vs v) (?I v)) | v. v \in set \?vs \land
\?I v \neq None \}
= (\lambda v. \{ \{ literal-formula-to-literal (encode-state-variable \theta (index \?vs v) (?I v)) \} \} )
  \set \?vs
using nb2
by force

} thus \?thesis
using cnf-of-encode-initial-state-set-i
by (smt Collect-cong)

qed

lemma cnf-of-encode-initial-state-set:
assumes is-valid-problem-strips II
and v \in dom (strips-problem.initial-of II)
shows strips-problem.initial-of II v = Some True \implies (\exists ! C. C \in cnf (\Phi_I II)
\land C = \{ (State 0 (index (strips-problem.variables-of II) v)) ^+ \} )
and strips-problem.initial-of II v = Some False \implies (\exists ! C. C \in cnf (\Phi_I II)
\land C = \{ (State 0 (index (strips-problem.variables-of II) v)) ^- 1 \} )

proof –
let ?I = (II)_I
let \?vs = strips-problem.variables-of II
let \?\Phi_I = \Phi_I II
have nb1: cnf (\Phi_I II) = \bigcup \{ cnf (encode-state-variable \theta (index \?vs v) (\set \?vs)
(strips-problem.initial-of II v)) | v. v \in set \?vs \land \?I v \neq None \}
using cnf-of-encode-initial-state-set-i
by blast

have v \in set \?vs
  using is-valid-problem-strips-initial-of-dom assms(1, 2)
by blast
hence v \in \{ v. v \in set \?vs \land \?I v \neq None \}
using assms(2)
by auto

} note nb2 = this
show strips-problem.initial-of II v = Some True \implies (\exists ! C. C \in cnf (\Phi_I II)
\land C = \{ (State 0 (index (strips-problem.variables-of II) v)) ^+ \} )
and strips-problem.initial-of II v = Some False \implies (\exists ! C. C \in cnf (\Phi_I II)
\land C = \{ (State 0 (index (strips-problem.variables-of II) v)) ^- 1 \} )

proof (auto)
assume i-v-is-some-true: strips-problem.initial-of II v = Some True
then have \{ (State 0 (index (strips-problem.variables-of II) v)) ^+ \}
  \in cnf (encode-state-variable \theta (index (strips-problem.variables-of II) v) (?I v))
unfolding encode-state-variable-def
using i-v-is-some-true

196
by auto
thus \{ (State 0 (index (strips-problem.variables-of II) v)) \} 
∈ cnf (Φ II)
using nb1 nb2
by auto

next
assume i-v-is-some-false: strips-problem.initial-of II v = Some False
then have \{ (State 0 (index (strips-problem.variables-of II) v))^-1 \} 
∈ cnf (encode-state-variable 0 (index (strips-problem.variables-of II) v) (??I)
v))
unfolding encode-state-variable-def
using i-v-is-some-false
by auto
thus \{ (State 0 (index (strips-problem.variables-of II) v))^-1 \} 
∈ cnf (Φ II)
using nb1 nb2
by auto
qed

lemma cnf-of-operator-encoding-structure:
cnf (encode-operators II t) = cnf (encode-all-operator-preconditions II
 (strips-problem.operators-of II) t)
unfolding encode-operators-def
using cnf.simps(5)
by metis

corollary cnf-of-operator-precondition-encoding-subset-encoding:
cnf (encode-all-operator-preconditions II (strips-problem.operators-of II) t) 
⊆ cnf (Φ II t)
using cnf-of-operator-encoding-structure cnf-of-encode-problem-structure subset-trans
unfolding encode-problem-def
by blast

lemma cnf-foldr-and[simp]:
cnf (foldr (\&\&) fs (\&\&)) = (\bigcup f \in set fs. cnf f)
proof (induction fs)
case (Cons f fs)
have ih: cnf (foldr (\&\&) fs (\&\&)) = (\bigcup f \in set fs. cnf f)
using Cons.IH
by blast
{ have cnf (foldr (\&\&) (f \# fs) (\&\&)) = cnf (f \& foldr (\&\&) fs (\&\&))
  by simp
also have \ldots = cnf f \cup cnf (foldr (\&\&) fs (\&\&))
  by force
finally have cnf (foldr (\&\&) (f \# fs) (\&\&)) = cnf f \cup (\bigcup f \in set fs. cnf f)
private lemma cnf-of-encode-operator-precondition[simp]:
cnf (encode-operator-precondition Π t op) = (∪v ∈ set (precondition-of op).
{((Operator t (index (strips-problem.operators-of Π) op))−1
 , (State t (index (strips-problem.variables-of Π) v))+)\})).

proof –
let ?vs = strips-problem.variables-of Π
and ?ops = strips-problem.operators-of Π
and ?Φ p = encode-operator-precondition Π t op
let ?fs = map (λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t (index ?vs v)))
(precondition-of op)
and ?A = (λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t (index ?vs v)))

' set (precondition-of op)

have cnf (encode-operator-precondition Π t op) = cnf (\?fs)
  unfolding encode-operator-precondition-def
  by presburger
also have . . = ∪ (cnf ∘ set ?fs)
  using cnf-BigAnd
  by blast
also have . . = ∪ (cnf ∘ ?A)
  using set-map[af λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t (index ?vs v))]
  (precondition-of op)
  by argo
also have . . = (∪ v ∈ set (precondition-of op). cnf (¬(Atom (Operator t (index ?ops op))) ∨ Atom (State t (index ?vs v))))
  by blast

finally show ?thesis
  by auto
qed

lemma cnf-of-encode-all-operator-preconditions-structure[simp]:
cnf (encode-all-operator-preconditions Π (strips-problem.operators-of Π) t)
= (∪(t, op) ∈ ({<t} × set (operators-of Π)).
(∪ v ∈ set (precondition-of op).
{((Operator t (index (strips-problem.operators-of Π) op))−1
 , (State t (index (strips-problem.variables-of Π) v))+)\})).

proof –
let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
let ?l = List.product [0..<t] ?ops
    and ?ΦP = encode-all-operator-preconditions Π (strips-problem.operators-of Π) t
let ?A = set (map (λ(t, op). encode-operator-precondition Π t op) ?l)
    { have set ?l = {0..<t} × set ((Π)O)
      by auto
      then have ?A = (λ(t, op). encode-operator-precondition Π t op) · ({0..<t} × set ((Π)O))
        using set-map
        by force
    } note nb = this
have cnf ?ΦP = cnf (foldr (∧) (map (λ(t, op). encode-operator-precondition Π t op) ?l) (¬⊥))
  unfolding encode-all-operator-preconditions-def
  by presburger
also have ... = (⋃f ∈ ?A. cnf f)
  by simp
also have ... = (⋃(k, op) ∈ ({0..<t} × set ((Π)O)).
  cnf (encode-operator-precondition Π k op))
  using nb
  by fastforce
finally show ?thesis
  by fastforce
qed

corollary cnf-of-encode-all-operator-preconditions-contains-clause-if:
  fixes Π::'variable STRIPS-Representation.strips-problem
  assumes is-valid-problem-strips (Π::'variable STRIPS-Representation.strips-problem)
    and k < t
    and op ∈ set ((Π)O)
    and v ∈ set (precondition-of op)
  shows { (Operator k (index (strips-problem.operators-of Π) op))⁻¹
    , (State k (index (strips-problem.variables-of Π) v))⁺ } ∈ cnf (encode-all-operator-preconditions Π (strips-problem.operators-of Π) t)
proof –
  let ?ops = strips-problem.operators-of Π
  and ?vs = strips-problem.variables-of Π
  let ?ΦP = encode-all-operator-preconditions Π ?ops t
  and ?C = { (Operator k (index (strips-problem.operators-of Π) op))⁻¹
    , (State k (index (strips-problem.variables-of Π) v))⁺ } { have nb: (k, op) ∈ {..<t} × set ((Π)O)
      using assms(2, 3)
      by blast
    }
moreover {  
  have \( \exists C \in (\bigcup v \in \text{set (precondition-of } \text{op})^{-1}, \)
  \( (\text{State } k (\text{index } \text{strips-problem.variables-of } \Pi) v)^+ \}} \)
  
  using UN-iff [where \( A = \text{set (precondition-of } \text{op}) \)
  
  and \( B = \lambda v. \{((\text{Operator } t (\text{index } \text{strips-problem.operators-of } \Pi) \text{op}))^{-1}, \)
  \( (\text{State } t (\text{index } \text{strips-problem.variables-of } \Pi) v)^+ \} \} \) asms(4)
  
  by blast
  
  hence \( \exists x \in \{..<t\} \times \text{set ((}\Pi) O) \).
  
  \( \forall C \in (\text{case } x \text{ of } (k, \text{op}) \Rightarrow \bigcup v \in \text{set (precondition-of } \text{op})^{-1}, \)
  \( (\text{State } k (\text{index } \text{strips-problem.variables-of } \Pi) v)^+ \} \}) \)
  
  using nb
  
  by blast
  
}
  
ultimately have \( \exists C \in (\bigcup (t, \text{op}) \in \{..<t\} \times \text{set ((}\Pi) O) \).
  
  \( \bigcup v \in \text{set (precondition-of } \text{op})^{-1}, \)
  \( \{ (\text{Operator } t (\text{index } \text{operators-of } \Pi) \text{op}))^{-1}, (\text{State } t (\text{index } \text{variables-of } \Pi) v)^+ \} \}) \)
  
  by blast
  
}
  
thus \( \text{thesis} \)
  
using cnf-of-encode-all-operator-preconditions-structure[of \Pi t]
  
by argo
  
qed

\begin{corollary}
cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem:
cnf (encode-all-operator-effects \Pi (\text{strips-problem.operators-of } \Pi) t)
  \subseteq cnf (\Phi \Pi t)
  
using cnf-of-encode-problem-structure(3) cnf-of-operator-encoding-structure
  
unfolding encode-problem-def
  
by blast
\end{corollary}

\begin{private lemma}
cnf-of-encode-operator-effect-structure[simp]:
cnf (encode-operator-effect \Pi t op)
  = (\bigcup v \in \text{set (add-effects-of } \text{op}) \} \{ (\text{Operator } t (\text{index } \text{operators-of } \Pi) \text{op}))^{-1}
  \cup (\bigcup v \in \text{set (delete-effects-of } \text{op}) \} \{ (\text{State } (\text{Suc } t) (\text{index } \text{variables-of } \Pi) v)^+ \})
  \cup (\bigcup v \in \text{set (delete-effects-of } \text{op}) \} \{ (\text{State } (\text{Suc } t) (\text{index } \text{variables-of } \Pi) v)^- \}) \}
  
proof =
  
let \( \text{o}_{s1} = \text{map (}(\lambda v. \neg(\text{Atom (Operator } t (\text{index } \text{operators-of } \Pi) \text{op}))))(\text{add-effects-of } \text{op}) \)
  
\( \forall \text{Atom (State } (\text{Suc } t) (\text{index } \text{variables-of } \Pi) v) \))
  
\( \text{and } \text{o}_{s2} = \text{map (}(\lambda v. \neg(\text{Atom (Operator } t (\text{index } \text{operators-of } \Pi) \text{op}))))(\text{delete-effects-of } \text{op}) \)
  
\( \forall \neg (\text{Atom (State } (\text{Suc } t) (\text{index } \text{variables-of } \Pi) v))) \)
  
by blast

200
{  
  have cnf ' set ?fs1 = cnf
    (λv. ¬(Atom (Operator t (index (strips-problem.operators-of II) op)))
       ∨ Atom (State (Suc t) (index (strips-problem.variables-of II) v)))
    set (add-effects-of op)
    using set-map
    by force
  also have ... = (λv. cnf (¬(Atom (Operator t (index (strips-problem.operators-of II) op))))
        ∨ Atom (State (Suc t) (index (strips-problem.variables-of II) v))))
    set (add-effects-of op)
    using image-comp
    by blast
  finally have cnf ' set ?fs2 = cnf ' (λv. ¬(Atom (Operator t (index (strips-problem.operators-of II) op))))
        ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of II) v))))
    set (add-effects-of op)
    by auto
} note nb1 = this
{  
  have cnf ' set ?fs2 = cnf ' (λv. ¬(Atom (Operator t (index (strips-problem.operators-of II) op))))
        ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of II) v))))
    set (add-effects-of op)
    by force
  also have ... = (λv. cnf (¬(Atom (Operator t (index (strips-problem.operators-of II) op))))
        ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of II) v))))
    set (add-effects-of op)
    using image-comp
    by blast
  finally have cnf ' set ?fs2 = (λv. {{ (Operator t (index (strips-problem.operators-of II) op))^{-1}}
        , (State (Suc t) (index (strips-problem.variables-of II) v))^{+} }})
    set (add-effects-of op)
    by auto
} note nb2 = this
{  
  have cnf (encode-operator-effect II t op) = \( \bigcup \) (cnf ' set (?fs1 @ ?fs2))
    unfolding encode-operator-effect-def
    using cnf-BigAnd[of ?fs1 @ ?fs2]
    by meson
  also have ... = \( \bigcup \) (cnf ' set ?fs1 ∪ cnf ' set ?fs2)
    by argo
  also have ... = \( \bigcup \) (cnf ' set ?fs1) ∪ \( \bigcup \) (cnf ' set ?fs2)

201
using \texttt{Union-Un-distrib[of cnf ' set ?fs_1 cnf ' set ?fs_2]}
by argo

finally have \texttt{cnf (encode-operator-effect II t op)}
= (\bigcup v \in \text{set (add-effects-of op)}.
   \{\{ (Operator t (index (strips-problem.operators-of II) op))^{-1}
   , (State (Suc t) (index (strips-problem.variables-of II) v))^+ \}\})
\bigcup (\bigcup v \in \text{set (delete-effects-of op)}.
   \{\{ (Operator t (index (strips-problem.operators-of II) op))^{-1}
   , (State (Suc t) (index (strips-problem.variables-of II) v))^{-1} \}\})
using \texttt{nb_1 nb_2}
by argo
\}
thus \texttt{?thesis}
by blast
qed

\begin{lemma}
\texttt{cnf-of-encode-all-operator-effects-structure}:
\begin{align*}
\texttt{cnf (encode-all-operator-effects II (strips-problem.operators-of II) t)}
= & (\bigcup (k, op) \in (\{0..<t\} \times \text{set (II)}))\times
(\bigcup v \in \text{set (add-effects-of op)}.
   \{\{ (Operator k (index (strips-problem.operators-of II) op))^{-1}
   , (State (Suc k) (index (strips-problem.variables-of II) v))^+ \}\})
\bigcup (\bigcup v \in \text{set (delete-effects-of op)}.
   \{\{ (Operator k (index (strips-problem.operators-of II) op))^{-1}
   , (State (Suc k) (index (strips-problem.variables-of II) v))^{-1} \}\})
\end{align*}
\end{lemma}

\textbf{proof} –
let \texttt{?ops = strips-problem.operators-of II}
and \texttt{?vs = strips-problem.variables-of II}
let \texttt{?\Phi_E = encode-all-operator-effects II ?ops t}
and \texttt{?l = List.product [0..<t] ?ops}
let \texttt{?fs = map (?l t, op). encode-operator-effect II t op) \?l}
\texttt{have nb: set (List.product [0..<t] ?ops) = \{0..<t\} \times set ?ops}
by simp
\{
  have \texttt{cnf ' set ?fs = cnf ' (\lambda(k, op). encode-operator-effect II k op) ' \{0..<t\}
   \times set ?ops)}
  by force
  also have \ldots = (\lambda(k, op). cnf (encode-operator-effect II k op)) ' \{0..<t\} \times
  set ?ops
\}
using \texttt{image-comp}
by fast

finally have \texttt{cnf ' set \?fs = (\lambda(k, op).}
(\bigcup v \in \text{set (add-effects-of op)}.
   \{\{ (Operator k (index (strips-problem.operators-of II) op))^{-1}
   , (State (Suc k) (index (strips-problem.variables-of II) v))^+ \}\})
\bigcup (\bigcup v \in \text{set (delete-effects-of op)}.

\[
\{ \text{(Operator } k \text{ (index (strips-problem.operators-of } \Pi) \text{ op)})^{-1} \\
\text{, (State (Suc } k \text{ (index (strips-problem.variables-of } \Pi) \text{ v)})^{-1} } \}
\}
\]

\text{using cnf-of-encode-operator-effect-structure}
\text{by auto}

\text{thus} \ ?\text{thesis}
\text{unfolding encode-all-operator-effects-def}
\text{using cnf-BigAnd[of ?fs]}
\text{by auto}

\text{qed}

\text{corollary} cnf-of-operator-effect-encoding-contains-add-effect-clause-if:
\text{fixes} \ \Pi::'a strips-problem
\text{assumes} \ \text{is-valid-problem-strips} \ \Pi
\text{and} \ k < t
\text{and} \ op \in \text{set} ((\Pi)_O)
\text{and} \ v \in \text{set} \ (\text{add-effects-of op})
\text{shows} \ \{ \text{(Operator } k \text{ (index (strips-problem.operators-of } \Pi) \text{ op)})^{-1} \\
\text{, (State (Suc } k \text{ (index (strips-problem.variables-of } \Pi) \text{ v)})^+} \}
\in \text{cnf} \ (\text{encode-all-operator-effects } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t)

\text{proof} –
\text{let} \ ?\Phi_E = \text{encode-all-operator-effects} \ \Pi \ (\text{strips-problem.operators-of } \Pi) \ t
\text{and} \ ?\text{ops} = \text{strips-problem.operators-of } \Pi
\text{and} \ ?\text{vs} = \text{strips-problem.variables-of } \Pi
\text{let} \ ?\text{Add} = \bigcup \{(k, op)\in\{0..<t\} \times \text{set} \ ((\Pi)_O)\}
\bigcup v\in\text{set} \ (\text{add-effects-of op}). \{(\text{Operator } k \text{ (index } ?\text{ops op)})^{-1}, \text{(State (Suc } k \text{ (index } ?\text{vs v})^+)} \}
\text{let} \ ?\text{C} = \{(\text{Operator } k \text{ (index } ?\text{ops op)})^{-1}, \text{ (State (Suc } k \text{ (index } ?\text{vs v})^+} \}
\text{have} \ ?\text{Add} \subseteq \text{cnf} \ ?\Phi_E
\text{using cnf-of-encode-all-operator-effects-structure[of } \Pi \ \text{t]} \ \text{Un-upper1[of } ?\text{Add]}\n\text{by presburger}
\text{moreover} \ {\}
\text{have} \ ?\text{C} \in \{(\text{Operator } k \text{ (index } ?\text{ops op)})^{-1}, \text{ (State (Suc } k \text{ (index } ?\text{vs v})^+} \}
\text{using} \ \text{assms(4)}
\text{by blast}
\text{then have} \ ?\text{C} \in (\bigcup v\in\text{set} \ (\text{add-effects-of op})
\{(\text{Operator } k \text{ (index } ?\text{ops op})^{-1}, \text{ (State (Suc } k \text{ (index } ?\text{vs v})^+} \})
\text{using Complete-Lattices.UN-iff[of } ?\text{C} λv. \{(\text{Operator } k \text{ (index } ?\text{ops op)})^{-1} \\
\text{, (State (Suc } k \text{ (index } ?\text{vs v})^+} \} \ \text{set} \ (\text{add-effects-of op})
\text{using} \ \text{assms(4)}
\text{by blast}
\text{moreover have} \ (k, op) \in (\{0..<t\} \times \text{set} \ ((\Pi)_O))
\text{using} \ \text{assms(2, 3)}
\text{by fastforce}
\text{ultimately have} \ ?\text{C} \in ?\text{Add}

203
ultimately show \( \text{thesis} \)
\[
\text{using subset-eq[of Add cnf } \Phi_E] \\
\text{by meson}
\]
\text{qed}

\text{corollary cnf-of-operator-effect-encoding-contains-delete-effect-clause-if:}
\text{fixes II: ’a strips-problem}
\text{assumes is-valid-problem-strips II}
\text{and } k < t \\
\text{and } op \in \text{set } ((\Pi)_\land) \\
\text{and } v \in \text{set } \text{(delete-effects-of } op)
\text{shows } \{ (\text{Operator } k \text{ (index (strips-problem.operators-of } II) } op))^{-1} \\
\text{, (State (Suc } k \text{ (index (strips-problem.variables-of } II) } v))^{-1} \} \\
\in \text{cnf } (\text{encode-all-operator-effects } II \text{ (strips-problem.operators-of } II) t)
\text{proof –}
\text{let } \Phi_E = \text{encode-all-operator-effects } II \text{ (strips-problem.operators-of } II) t \\
\text{and } ?ops = \text{strips-problem.operators-of } II \\
\text{and } ?vs = \text{strips-problem.variables-of } II
\text{let } \text{?Delete } = (\bigcup (k, op) \in \{0..<t\} \times \text{set } ((\Pi)_\land)). \\
\text{\bigcup v \in \text{set } \text{(delete-effects-of } op).} \\
\{\{ (\text{Operator } k \text{ (index } ?ops \text{ op))}^{-1}, (\text{State (Suc } k \text{ (index } ?vs \text{ v}))^{-1} \} \})
\text{let } ?C = \{ (\text{Operator } k \text{ (index } ?ops \text{ op}))^{-1}, (\text{State (Suc } k \text{ (index } ?vs \text{ v}))^{-1} \} \\
\text{have } \text{?Delete } \subseteq \text{cnf } \Phi_E \\
\text{using cnf-of-encode-all-operator-effects-structure[of } II t] \text{ Un-upper2[of } \text{?Delete]}
\text{by presburger}
\text{moreover } \{ \\
\text{have } ?C \in (\bigcup \text{v } \in \text{set } \text{(delete-effects-of } op). \\
\{\{ (\text{Operator } k \text{ (index } ?ops \text{ op}))^{-1}, (\text{State (Suc } k \text{ (index } ?vs \text{ v}))^{-1} \} \}) \\
\text{using assms(4) }
\text{by blast}
\text{moreover have } (k, op) \in \{0..<t\} \times \text{set } ?ops \\
\text{using assms(2, 3) }
\text{by force} \\
\text{ultimately have } ?C \in ?\text{Delete}
\text{by fastforce}
\}
\text{ultimately show } \text{thesis}
\text{using subset-eq[of } \text{?Delete cnf } \Phi_E] \\
\text{by meson}
\text{qed}

\text{private lemma cnf-of-big-or-of-literal-formulas-is[simp]:}
\text{assumes } \forall f \in \text{set } fs. \text{ is-literal-formula } f \\
\text{shows } \text{cnf } (\bigvee fs) = \{ \text{literal-formula-to-literal } f | f. f \in \text{set } fs \}
\]
using assms

proof (induction \( fs \))

  case (Cons \( f \) \( fs \))
  {
    have is-literal-formula-f: is-literal-formula \( f \)
      using Cons.prems(1)
      by simp
    then have \( \text{cnf } f = \{ \{ \text{literal-formula-to-literal } f \} \} \)
      using cnf-of-literal-formula
      by blast
  } note \( nb_1 = \text{this} \)
  {
    have \( \forall f' \in \text{set } fs. \text{is-literal-formula } f' \)
      using Cons.prems
      by fastforce
    hence \( \text{cnf } (\bigvee fs) = \{ \{ \text{literal-formula-to-literal } f | f \in \text{set } fs \} \} \)
      using Cons.IH
      by argo
  } note \( nb_2 = \text{this} \)
  {
    have \( \text{cnf } (\bigvee (f \# fs)) = (\lambda (g, h). g \cup h)
    \times \{ \{ \text{literal-formula-to-literal } f' | f' \in \text{set } fs \} \}) \)
      using \( nb_1 \) \( nb_2 \)
      by simp
    also have \( \ldots = \{ \{ \text{literal-formula-to-literal } f \} \)
      \( \cup \{ \text{literal-formula-to-literal } f' | f' \in \text{set } fs \} \)
      by fast
    finally have \( \text{cnf } (\bigvee (f \# fs)) = \{ \{ \text{literal-formula-to-literal } f' | f' \in \text{set } (f \# fs) \} \} \)
      by fastforce
  }
  thus \( \text{case } \).

qed simp

private lemma set-filter-op-list-mem-vs[simp]:
  \( \text{set (filter } (\lambda \text{op. ListMem } v \text{ vs } \text{ops}) = \{ \text{op. op } \in \text{set } \text{ops } \land v \in \text{set } \text{vs } \} \)
using set-filter[of \( \lambda \text{op. ListMem } v \text{ vs } \text{ops} \) ListMem-iff]
by force

private lemma cnf-of-positive-transition-frame-axiom:
  \( \text{cnf } (\text{encode-positive-transition-frame-axiom II } k \text{ v}) \)
  \( = \{ \{ \text{(State } k \text{ (index } (\text{strips-problem.variables-of II } v))^{+} \}
    \cup (\text{State } (\text{Suc } k) \text{ (index } (\text{strips-problem.variables-of II } v))^{-1} \} \}
  \cup \{ \text{(Operator } k \text{ (index } (\text{strips-problem.operators-of II } \text{op}))^{+} \}
    \text{ | op. op } \in \text{set } (\text{strips-problem.operators-of II } \land v \in \text{set } (\text{add-effects-of op}) \}
  \}
\)
proof
  let \( \?vs = \text{strips-problem.variables-of II} \)
and \( \text{ops} = \text{strips-problem.operators-of } \Pi \)

let \( \text{adding-operators} = \text{filter}(\lambda \text{op. ListMem } v \text{ (add-effects-of op)}) \) \( \text{ops} \)

let \( \text{fs} = \text{map}(\lambda \text{op. Atom } (\text{Operator } k \text{ (index } \text{ops } \text{op}))) \) \( \text{adding-operators} \)

\[
\begin{align*}
\text{have set } \text{fs} & = (\lambda \text{op. Atom } (\text{Operator } k \text{ (index } \text{ops } \text{op}))) \text{' set } \text{adding-operators} \\
\text{using set-map[of } \lambda \text{op. Atom } (\text{Operator } k \text{ (index } \text{ops } \text{op})) \text{ adding-operators]}
\end{align*}
\]

by blast

then have literal-formula-to-literal ' set \( \text{fs} \) 
\[
(\lambda \text{op. (Operator } k \text{ (index } \text{ops } \text{op}))^+) \text{' set } \text{adding-operators}
\]

using image-comp[of literal-formula-to-literal \( \lambda \text{op. Atom } (\text{Operator } k \text{ (index } \text{ops } \text{op})) \) \( \text{adding-operators} \)]

by simp

also have \( \ldots = (\lambda \text{op. (Operator } k \text{ (index } \text{ops } \text{op}))^+) \text{' set } \text{adding-operators} \)

\[
\begin{align*}
\text{using set-filter-op-list-mem-vs[of } v - \text{ops]}
\end{align*}
\]

by auto

finally have literal-formula-to-literal ' set \( \text{fs} \) 
\[
(\lambda \text{op. (Operator } k \text{ (index } \text{ops } \text{op}))^+) \text{' set } \text{adding-operators} \]

using setcompr-eq-image[of \( \lambda \text{op. (Operator } k \text{ (index } \text{ops } \text{op}))^+ \) \( \text{adding-operators} \)]

by blast

hence \( \text{cnf} (\bigwedge \text{fs}) = \{ (\text{Operator } k \text{ (index } \text{ops } \text{op}))^+ \ |
\text{op. op } \in \text{set } \text{ops } \land \text{v } \in \text{set (add-effects-of op)} \} \)

using cnf-of-big-or-of-literal-formulas-is[of \( \text{fs} \)]

setcompr-eq-image[of literal-formula-to-literal \( \lambda f. f \in \text{set } \text{fs} \)]

by force

\}

then have \( \text{cnf} (\neg (\text{Atom } (\text{State } (\text{Suc } k) \text{ (index } \text{vs } v))) \lor (\bigvee \text{fs}) ) = \{ (\text{State } (\text{Suc } k) \text{ (index } \text{vs } v))^{-1} \} \cup \{ (\text{Operator } k \text{ (index } \text{ops } \text{op}))^+ \ |
\text{op. op } \in \text{set } \text{ops } \land \text{v } \in \text{set (add-effects-of op)} \} \}

by force

then have \( \text{cnf} ((\text{Atom } (\text{State } k \text{ (index } \text{vs } v))) \lor (\neg (\text{Atom } (\text{State } (\text{Suc } k) \text{ (index } \text{vs } v)))) \lor (\bigvee \text{fs}))
\]
\[
= \{ (\text{State } k \text{ (index } \text{vs } v))^{-1} \}
\cup \{ (\text{State } (\text{Suc } k) \text{ (index } \text{vs } v))^{-1} \}
\cup \{ (\text{Operator } k \text{ (index } \text{ops } \text{op}))^+ \ |
\text{op. op } \in \text{set } \text{ops } \land \text{v } \in \text{set (add-effects-of op)} \} \}
\]

by simp

moreover have \( \text{cnf} (\text{encode-positive-transition-frame-axiom } \Pi k v)
\]
\[
= \text{cnf} ((\text{Atom } (\text{State } k \text{ (index } \text{vs } v))) \lor (\neg (\text{Atom } (\text{State } (\text{Suc } k) \text{ (index } \text{vs } v)))) \lor (\bigvee \text{fs}))
\]

206
unfolding encode-positive-transition-frame-axiom-def
by metis

ultimately show ?thesis
by blast
qed

private lemma cnf-of-negative-transition-frame-axiom:
cnf (encode-negative-transition-frame-axiom II k v)
= {{ (State k (index (strips-problem.variables-of II) v))⁻¹
   , (State (Suc k) (index (strips-problem.variables-of II) v))₆⁺ } } ∪
  {{ (Operator k (index (strips-problem.operators-of II) op))⁺
   | op. op ∈ set (strips-problem.operators-of II) ∧ v ∈ set (delete-effects-of op) } }
proof
let ?vs = strips-problem.variables-of II
and ?ops = strips-problem.operators-of II
let ?deleting-operators = filter (λop. ListMem v (delete-effects-of op)) ?ops
let ?fs = map (λop. Atom (Operator k (index ?ops op))) ?deleting-operators
{ have set ?fs = (λop. Atom (Operator k (index ?ops op))) ' set ?deleting-operators
using set-map[of λop. Atom (Operator k (index ?ops op))] ?deleting-operators
by blast
then have literal-formula-to-literal ' set ?fs
= (λop. (Operator k (index ?ops op))₆⁺) ' set ?deleting-operators
using image-comp[of literal-formula-to-literal λop. Atom (Operator k (index ?ops op))]
  set ?deleting-operators]
by simp
also have … = (λop. (Operator k (index ?ops op))₆⁺)
  ' { op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) }
using set-filter-op-list-mem-vs[of v - ?ops]
by auto
finally have literal-formula-to-literal ' set ?fs
= { (Operator k (index ?ops op))₆⁺ | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) }
using setcomp-eq-image[of λop. (Operator k (index ?ops op))₆⁺
  λop. op ∈ set ?deleting-operators]
by blast

hence cnf (∨?fs) = {{ (Operator k (index ?ops op))₆⁺
  | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) } }
using cnf-of-big-or-of-literal-formulas-is[of ?fs]
  setcomp-eq-image[of literal-formula-to-literal λf. f ∈ set ?fs]
by force
}
then have cnf \((\text{Atom (State (Suc } k \text{ (index } \text{?vs } v)) \lor \forall \text{?fs})\)
\[ \begin{align*}
&= \{\{ \text{State (Suc } k \text{ (index } \text{?vs } v)) \}^+ \} \cup \{ \text{Operator } k \text{ (index } \text{?ops op}) \}^+ \\
& \quad \mid \text{op. op } \in \text{set } \text{?ops } \land \text{v } \in \text{set } \{\text{delete-effects-of op}\}\}\} \\
\text{by force} &
\end{align*} \]

then have cnf \(((\neg(\text{Atom (State } k \text{ (index } \text{?vs } v)))) \lor (\text{Atom (State (Suc } k \text{ (index } \text{?vs } v)) \lor \forall \text{?fs})\)
\[ \begin{align*}
&= \{\{ \text{State } k \text{ (index } \text{?vs } v))^{-1} \} \\
& \quad \cup \{ \text{State (Suc } k \text{ (index } \text{?vs } v)) \}^+ \} \\
& \quad \cup \{ \text{Operator } k \text{ (index } \text{?ops op}) \}^+ \mid \text{op. op } \in \text{set } \text{?ops } \land \text{v } \in \text{set } \{\text{delete-effects-of op}\}\}\} \\
\text{by simp} &
\end{align*} \]

moreover have cnf \((\text{encode-negative-transition-frame-axiom } \Pi k v)\)
\[ \begin{align*}
&= \text{cnf } ((\neg(\text{Atom (State } k \text{ (index } \text{?vs } v)))) \lor (\text{Atom (State (Suc } k \text{ (index } \text{?vs } v)) \lor \forall \text{?fs})\) \\
\text{unfolding } &\text{encode-negative-transition-frame-axiom-def} \\
\text{by } &\text{metis} \\
\text{ultimately show } &\text{?thesis} \\
\text{by } &\text{blast} \\
\text{qed} &
\end{align*} \]

lemma cnf-of-encode-all-frame-axioms-structure:
\[ \text{cnf (encode-all-frame-axioms } \Pi t\) \]
\[ \begin{align*}
&= \bigcup \{(k, v) \in \{0..<t\} \times \text{set } \{\Pi v\}\}. \\
&\quad \{\{ \text{State } k \text{ (index } \text{strips-problem.variables-of } \Pi v) \}^+ \\
&\quad \quad \mid \text{State (Suc } k \text{ (index } \text{strips-problem.variables-of } \Pi v))^{-1} \} \\
&\quad \quad \cup \{ \text{Operator } k \text{ (index } \text{strips-problem.operators-of } \Pi v) \}^+ \\
&\quad \quad \mid \text{op. op } \in \text{set } \{\Pi v\} \land \text{v } \in \text{set } \{\text{add-effects-of op}\}\}\} \\
&\quad \cup \bigcup \{(k, v) \in \{0..<t\} \times \text{set } \{\Pi v\}\}. \\
&\quad \{\{ \text{State } k \text{ (index } \text{strips-problem.variables-of } \Pi v))^{-1} \\
&\quad \quad \mid \text{State (Suc } k \text{ (index } \text{strips-problem.variables-of } \Pi v)) \}^+ \\
&\quad \quad \cup \{ \text{Operator } k \text{ (index } \text{strips-problem.operators-of } \Pi v) \}^+ \\
&\quad \quad \mid \text{op. op } \in \text{set } \{\Pi v\} \land \text{v } \in \text{set } \{\text{delete-effects-of op}\}\}\}\} \\
\text{proof } &
\end{align*} \]

let \text{?vs } = \text{strips-problem.variables-of } \Pi \\
and \text{?ops } = \text{strips-problem.operators-of } \Pi \\
and \text{?}\Phi_F = \text{encode-all-frame-axioms } \Pi t \\
\text{let } \text{?l } = \text{List.product } \{0..<t\} \text{?vs} \\
\text{let } \text{?fs } = \text{map } (\lambda(k, v). \text{encode-negative-transition-frame-axiom } \Pi k v) \text{?l} \\
\text{at } \text{map } \{ \lambda(k, v). \text{encode-positive-transition-frame-axiom } \Pi k v \text{?l}\} \\
\text{have set-l: } \text{set } \text{?l } = \{0..<t\} \times \text{set } \{\Pi v\} \\
\text{using } \text{set-product
by force

have set ?fs = ?A ∪ ?B
unfolding set-append set-map
using encode-all-frame-axioms-set
by force
then have cnf ' set ?fs = cnf ' ?A ∪ cnf ' ?B
using image-Un[of cnf ?A ?B]
by argo
moreover {
  have ?A = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  { encode-negative-transition-frame-axiom \Pi k v }
  by blast
then have cnf ' ?A = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  { cnf (encode-negative-transition-frame-axiom \Pi k v) }
  by blast
hence cnf ' ?A = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  {{{ (State k (index ?vs v))^{-1} \rightarrow (State (Suc k) (index ?vs v))} \cup
    ((Operator k (index ?ops op)) \rightarrow (op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op)))}}
  using cnf-of-negative-transition-frame-axiom[of \Pi]
  by presburger
}
moreover {
  have ?B = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  { encode-positive-transition-frame-axiom \Pi k v }
  by blast
then have cnf ' ?B = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  { cnf (encode-positive-transition-frame-axiom \Pi k v) }
  by blast
hence cnf ' ?B = (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  {{{ (State k (index ?vs v)) \rightarrow (State (Suc k) (index ?vs v))^{-1} \cup
    ((Operator k (index ?ops op)) \rightarrow (op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op)))}}
  using cnf-of-positive-transition-frame-axiom[of \Pi]
  by presburger
}
ultimately have cnf ' set ?fs
= (∪ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V))).
  {{{ (State k (index ?vs v)) \rightarrow (State (Suc k) (index ?vs v))^{-1} \cup
    ((Operator k (index ?ops op)) \rightarrow (op. op ∈ set ((\Pi)_O) ∧ v ∈ set (add-effects-of op)) )}}
  \cup (\{ (k, v) ∈ (\{0..<t\} × set ((\Pi)_V)).
    {{{ (State k (index ?vs v))^{-1} \rightarrow (State (Suc k) (index ?vs v)))}}}
proof
\[
\{\text{Lemma: } A \text{ technical lemma used in .}
\]

\[ \begin{array}{c}
\text{private lemma cnf-of-encode-goal-state-set-i:} \\
\text{cnf } (\Phi_G \Pi \ t) = \bigcup \{ \text{cnf } (\text{encode-state-variable } t) \\
\text{(index } \text{strips-problem.variables-of } \Pi \text{ ) } v \} (\text{((II)G) } v) \\
| v, v \in \text{set } ((\Pi)v) \wedge (\Pi)G v \neq \text{None } \}
\end{array} \]

\[ \begin{array}{c}
\text{proof} \\
\text{let } ?vs = \text{strips-problem.variables-of } \Pi \\
\text{and } ?G = (\Pi)G \\
\text{and } ?\Phi_G = (\Phi_G \Pi) \ t \\
\text{let } ?fs = \text{map } (\lambda v. \text{encode-state-variable } t \text{ (index } ?vs v) \ (\?G v) \vee \bot) \\
\text{(filter } (\lambda v. ?G v \neq \text{None } \) ?vs) \\
\text{\{ have cnf \{ set } ?fs = \text{cnf } (\lambda v. \text{encode-state-variable } t \text{ (index } ?vs v) \ (\?G v) \vee \bot) \\
\text{\{ v \text{ v } \in \text{set } ?vs } \wedge \?G v \neq \text{None } \}
\end{array} \]

qed

— A technical lemma used in .
unfolding set-map
by force
also have . . . = (λv. cnf (encode-state-variable t (index ?vs v) (?G v) \lor ⊥))
' { v | v. v ∈ set ?vs \land ?G v \neq None }
using image-comp[of cnf (λv. encode-state-variable t (index ?vs v) (?G v) \lor ⊥)]

by fast
finally have cnf ' set ?fs = { cnf (encode-state-variable t (index ?vs v) (?G v))
| v. v ∈ set ?vs \land ?G v \neq None }
unfolding setcompr-eq-image[of λv. cnf (encode-state-variable t (index ?vs v) (?G v) \lor ⊥)]
by auto

moreover have cnf ((ΦG II) t) = \bigcup (cnf ' set ?fs)
unfolding encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def
using cnf-BigAnd[of ?fs]
by force
ultimately show ?thesis
by simp

qed

— A simplification lemma for the above one.

corollary cnf-of-encode-goal-state-set-ii:
assumes is-valid-problem-strips II
shows cnf ((ΦG II) t) = \bigcup (\{\{ literal-formula-to-literal
(encode-state-variable t (index (strips-problem.variables-of II) v) (((II)G) v))
\}|
| v. v ∈ set ((II)V) \land ((II)G) v \neq None })))
proof
let ?vs = strips-problem.variables-of II
and ?G = (II)G
and ?ΦG = (ΦG II) t

{ fix v
assume v ∈ \{ v | v. v ∈ set ((II)V) \land ?G v \neq None }.
then have v ∈ set ((II)V) and G-of-v-is-not-None: ?G v \neq None
by fast+
then consider (A) ?G v = Some True
| (B) ?G v = Some False
by fastforce
hence cnf (encode-state-variable t (index ?vs v) (?G v))
= \{ literal-formula-to-literal (encode-state-variable t (index ?vs v) (?G v))
\}
unfolding encode-state-variable-def
by (cases, force+)
} note nb = this
have \( \text{cnf } \Phi_G = \bigcup \{ \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v) \ (\text{?G } v)) \mid v. v \in \text{set } ((\Pi)_V) \land \text{?G } v \neq \text{None} \} \)

unfolding cnf-of-encode-goal-state-set-i
by blast
also have ... = \( \bigcup \{ (\lambda v. \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v) (((\Pi)_G) v))) \mid v. v \in \text{set } ((\Pi)_V) \land ((\Pi)_G) v \neq \text{None} \} \)
using setcompr-eq-image[of 
  \( \lambda v. \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v) (((\Pi)_G) v)) \)
  \( \lambda v. v \in \text{set } ((\Pi)_V) \land ((\Pi)_G) v \neq \text{None} \]
by presburger
also have ... = \( \bigcup \{ (\lambda v. \{ \text{literal-formula-to-literal } 
  (\text{encode-state-variable } t \ (\text{index } ?v \ v) \ (\text{?G } v)) \}) \mid v. v \in \text{set } ((\Pi)_V) \land ((\Pi)_G) v \neq \text{None} \} \)
using nb
by simp
finally show \(?\text{thesis}\)
  unfolding nb
by auto
qed

— This lemma essentially states that the cnf for the cnf formula for the encoding has a clause for each variable whose state is defined in the goal state with the corresponding literal.

lemma cnf-of-encode-goal-state-set:
  fixes II:: 'a strips-problem
  assumes is-valid-problem-strips II
  and v \in \text{dom } ((\Pi)_G)
  shows ((\Pi)_G) v = \text{Some True} \to (\exists ! C. C \in \text{cnf } ((\Phi_G II) t) 
  \land C = \{ (\text{State } t \ (\text{index } (\text{strips-problem.variables-of } II) v))^{+} \})
  \land (\Pi)_G v = \text{Some False} \to (\exists ! C. C \in \text{cnf } ((\Phi_G II) t)
  \land C = \{ (\text{State } t \ (\text{index } (\text{strips-problem.variables-of } II) v)^{-1} \})
proof —
let ?vs = \text{strips-problem.variables-of } II
and ?G = (\Pi)_G
and ?PhiG = (\Phi_G II) t
have nb1: cnf ?PhiG = \( \bigcup \{ \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v)
  \mid v. v \in \text{set } ((\Pi)_V) \land \text{?G } v \neq \text{None} \} \)
  unfolding cnf-of-encode-goal-state-set-i
  by auto
  have nb2: v \in \{ v. v \in \text{set } ((\Pi)_V) \land \text{?G } v \neq \text{None} \}
    using is-valid-problem-dom-of-goal-state-is assms(1, 2)
  by auto
  have nb3: cnf (\text{encode-state-variable } t \ (\text{index } \text{strips-problem.variables-of } II) v)
  ((\Pi)_G v)) \subseteq \( \bigcup \{ \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v)
  \mid v. v \in \text{set } ((\Pi)_V) \land \text{?G } v \neq \text{None} \} \)
  using \( \text{UN-upper[OP nb2, of } \lambda v. \text{cnf } (\text{encode-state-variable } t \ (\text{index } ?v \ v) \ (\text{?G } v)) \) nb2

212
by blast
show ((Π)G) v = Some True \rightarrow (\exists ! C. C \in cnf ((Φ_G Π) t) \\
\land C = \{ (State t (index (strips-problem:variables-of Π) v))^{+} \})
and ((Π)G) v = Some False \rightarrow (\exists ! C. C \in cnf ((Φ_G Π) t) \\
\land C = \{ (State t (index (strips-problem:variables-of Π) v))^{-1} \})
using nb3
unfolding nb1 encode-state-variable-def
by auto+
qed
end

We omit the proofs that the partial encoding functions produce formulas
in CNF form due to their more technical nature. The following sublocale
proof confirms that definition ?? encodes a valid problem Π into a formula
that can be transformed to CNF (is-cnf (Φ Π t)) and that its CNF has the
required form.

7.3 Soundness of the Basic SATPlan Algorithm

lemma valuation-models-encoding-cnf-formula-equals:
assumes is-valid-problem-strips Π
shows A \models Φ Π t = cnf-semantics A (cnf (Φ Π t))
proof –
let ?Φ = Φ Π t
{ 
  have is-cnf ?Φ 
  using is-cnf-encode-problem[OF assms].
  hence is-nnf ?Φ 
  using is-nnf-cnf
  by blast
}
thus ?thesis
  using cnf-semantics[of ?Φ A]
  by blast
qed

corollary valuation-models-encoding-cnf-formula-equals-corollary:
assumes is-valid-problem-strips Π
shows A \models (Φ Π t) 
  = (\forall C \in cnf (Φ Π t). \exists L \in C. lit-semantics A L)
using valuation-models-encoding-cnf-formula-equals[OF assms]
unfolding cnf-semantics-def clause-semantics-def encode-problem-def
by presburger

— A couple of technical lemmas about decode-plan.

lemma decode-plan-length:
assumes π = Φ^{-1} Π ν t
shows length π = t
using assms
unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
by simp

lemma decode-plan’-set-is[simp]:
set (decode-plan’ Π A k)
= { (strips-problem.operators-of Π) ! (index (strips-problem.operators-of Π) op)
| op, op ∈ set (strips-problem.operators-of Π)
∧ A (Operator k (index (strips-problem.operators-of Π) op)) }

proof –
let ?ops = strips-problem.operators-of Π
let ?f = λop. Operator k (index ?ops op)
let ?us = map ?f ?ops

{ have set (filter A ?us) = set (map ?f (filter (A o ?f) ?ops))
  unfolding filter-map[of A λop. Operator k (index ?ops op) ?ops],.. hence set (filter A ?us) = (λop. Operator k (index ?ops op)) •
  { op ∈ set ?ops. A (Operator k (index ?ops op)) }
  unfolding set-map set-filter
  by simp
  }
have set (decode-plan’ Π A k) = (λv. case k of Operator k i ⇒ ?ops ! i)
  • (λop. Operator k (index ?ops op)) •
  { op ∈ set ?ops. A (Operator k (index ?ops op)) }
  unfolding decode-plan’-def set-map Let-def
  by auto
also have . . . = (λop. case Operator k (index ?ops op) of Operator k i ⇒ ?ops ! i)
  • { op ∈ set ?ops. A (Operator k (index ?ops op)) }
  unfolding image-comp comp-apply
  by argo
also have . . . = (λop. ?ops ! (index ?ops op))
  • { op ∈ set ?ops. A (Operator k (index ?ops op)) }
  by force
finally show ?thesis
  by blast
qed

lemma decode-plan-set-is[simp]:
set (Φ⁻¹ Π A t) = (∪ k ∈ {..<t}. { decode-plan’ Π A k })
unfolding decode-plan-def SAT-Plan-Base.decode-plan-def set-map
using atLeast-upt
by blast

lemma decode-plan-step-element-then-i:
assumes k < t
shows set ((Φ⁻¹ Π A t) ! k)
\[ \{ (\text{strips-problem.operators-of } \Pi)! (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op}) \mid \text{op. op} \in \text{set } ((\Pi)_{O}) \land A (\text{Operator } k \text{ (index } (\text{strips-problem.operators-of } \Pi) \text{ op})) \} \]

**proof**

- have \((\Phi^{-1} \Pi A t)! k = \text{decode-plan'} \Pi A k\)
  - unfolding \text{decode-plan-def SAT-Plan-Base.decode-plan-def}
  - using \text{assms}
  - by \text{simp}
  - thus \?thesis
  - by \text{force}

**qed**

— Show that each operator \text{op} in the \(k\)-th parallel operator in a decoded parallel plan is contained within the problem’s operator set and the valuation is true for the corresponding SATPlan variable.

**lemma decode-plan-step-element-then:**

- fixes \(\Pi::'a \text{ strips-problem}\)
- assumes \(k < t\) and \(\text{op} \in \text{set } ((\Phi^{-1} \Pi A t)! k)\)
- shows \(\text{op} \in \text{set } ((\Pi)_{O})\)
  - and \(A (\text{Operator } k \text{ (index } \text{strips-problem.operators-of } \Pi \text{ op}))\)

**proof**

- let \(?ops = \text{strips-problem.operators-of } \Pi\)
- let \(?Ops = \{ ?ops! (\text{index } ?ops \text{ op}) \mid \text{op. op} \in \text{set } ((\Pi)_{O}) \land A (\text{Operator } k \text{ (index } ?ops \text{ op})) \} \}
- have \(\text{op} \in ?Ops\)
  - using \text{assms}(2)
  - unfolding \text{decode-plan-step-element-then-i[OF assms(1)] assms}
  - by \text{blast}

- moreover have \(\text{op} \in \text{set } ((\Pi)_{O})\)
  - and \(A (\text{Operator } k \text{ (index } ?ops \text{ op}))\)
    - using \text{calculation}
    - by \text{fastforce}+
  - ultimately show \(\text{op} \in \text{set } ((\Pi)_{O})\)
    - and \(A (\text{Operator } k \text{ (index } ?ops \text{ op}))\)
    - by \text{blast}+

**qed**

— Show that the \(k\)-th parallel operators of the decoded plan are distinct lists (i.e. do not contain duplicates).

**lemma decode-plan-step-distinct:**

- assumes \(k < t\)
- shows \(\text{distinct } ((\Phi^{-1} \Pi A t)! k)\)

**proof**

- let \(?ops = \text{strips-problem.operators-of } \Pi\)
- let \(?p_k = (\Phi^{-1} \Pi A t)! k\)
- let \(?f = \lambda \text{op}. \text{Operator } k \text{ (index } ?ops \text{ op})\)
  - and \(?g = \lambda v. \text{case } v \text{ of } \text{Operator } - k \Rightarrow ?ops! k\)
let ?vs = map ?f (remdups ?ops)
have nb1 : ?πk = decode-plan' II .A k
  unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
  using assms
  by fastforce
{
  have distinct (remdups ?ops)
    by blast
  moreover have inj-on ?f (set (remdups ?ops))
    unfolding inj-on-def
    by fastforce
  ultimately have distinct ?vs
    using distinct-map
    by blast
}
note nb2 = this
{
  have inj-on ?g (set ?vs)
    unfolding inj-on-def
    by fastforce
  hence distinct (map ?g ?vs)
    using distinct-map nb2
    by blast
}
thus ?thesis
  using distinct-map-filter[of ?g ?vs .A]
  unfolding nb1 decode-plan'-def Let-def
  by argo
qed

lemma decode-state-at-valid-variable:
  fixes II :: 'a strips-problem
  assumes (ΦS⁻¹ II .A k) v ≠ None
  shows v ∈ set ((II)v)
proof –
  let ?vs = strips-problem.variables-of II
  let ?f = λv. (v,.A (State k (index ?vs v)))
  {
    have fst ' set (map ?f ?vs) = fst ' (λv. (v,.A (State k (index ?vs v)))) ' set ?vs
      by force
    also have ... = (λv. fst (v,.A (State k (index ?vs v)))) ' set ?vs
      by blast
    finally have fst ' set (map ?f ?vs) = set ?vs
      by auto
  }
moreover have ¬v ∉ fst ' set (map ?f ?vs)
  using map-of-eq-None-iff[of map ?f ?vs v] assms
  unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
  by meson
ultimately show ?thesis
— Show that there exists an equivalence between a model $A$ of the (CNF of the) encoded problem and the state at step $k$ decoded from the encoded problem.

**Lemma decode-state-at-encoding-variables-equals-some-of-valuation-if:**

**Fixes** $\Pi$:: $\mathcal{A}$ strips-problem

**Assumes** $\text{is-valid-problem-strips} \; \Pi$

and $A \models \Phi \Pi \; t$

and $k \leq t$

and $v \in \text{set } ((\Pi)_V)$

**Shows** $(\Phi_{S^{-1}} \Pi, A) \; k \; v$

= Some $(A (\text{State } k (\text{strips-problem.variables-of } \Pi) \; v)))$

**Proof**

- let $?vs = \text{strips-problem.variables-of } \Pi$
- let $?l = \text{map } (\lambda x. (x,A (\text{State } k (\text{index } ?vs \; x)))) \; ?vs$

have $\text{set } ?vs \neq \{\}$

using $\text{assms}(4)$

by fastforce

then have $\text{map-of } ?l \; v = \text{Some } (A (\text{State } k (\text{index } ?vs \; v)))$

using $\text{map-of-from-function-graph-is-some-if}[\; \text{of } \; ?vs \; v$

$\lambda v. \; A (\text{State } k (\text{index } ?vs \; v))] \; \text{assms}(4)$

by fastforce

thus $?\text{thesis}$

unfolding $\text{decode-state-at-def SAT-Plan-Base.decode-state-at-def}$

by meson

**QED**

**Lemma decode-state-at-dom:**

**Assumes** $\text{is-valid-problem-strips} \; \Pi$

**Shows** $\text{dom } (\Phi_{S^{-1}} \Pi, A) \; k = \text{set } ((\Pi)_V)$

**Proof**

- let $?s = \Phi_{S^{-1}} \Pi, A \; k$

and $?vs = \text{strips-problem.variables-of } \Pi$

have $\text{dom } ?s = \text{fst } \text{of set } (\text{map } (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v)))) \; ?vs)$

unfolding $\text{decode-state-at-def SAT-Plan-Base.decode-state-at-def}$

using $\text{dom-map-of-conv-image-fst}[\; \text{of } \text{map } (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v)))) \; ?ts)]$

by meson

also have $\ldots = \text{fst } (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v)))) \; \text{of set } ((\Pi)_V)$

using $\text{set-map}[\; \text{of } (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v)))) \; ?vs]$ by simp

also have $\ldots = (\text{fst } \circ (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v)))) \; \text{of set } ((\Pi)_V)$

using $\text{image-comp}[\; \text{of } \text{fst } (\lambda v. (v,A (\text{State } k (\text{index } ?vs \; v))))]$ by presburger

finally show $?\text{thesis}$

by force

**QED**
lemma decode-state-at-initial-state:
assumes is-valid-problem-strips II
and \( \Delta \models \Phi \Pi \)
shows \((\Phi S^{-1} \Pi \Delta 0) = (\Pi)_I\)

proof -
let \( ?I = (\Pi)_I \)
let \( ?s = \Phi S^{-1} \Pi \Delta 0 \)
let \( ?vs = \text{strips-problem.variables-of} \Pi \)
let \( ?\Phi = \Phi \Pi \)
let \( ?\Phi_I = \Phi_i \Pi \)

\{
  have is-cnf ?\Phi_I and cnf ?\Phi_I \subseteq cnf ?\Phi
  
  subgoal
  using is-cnf-encode-initial-state[OF assms(1)]
  by simp
  
  subgoal
  using cnf-of-encode-problem-structure(1)
  unfolding encode-initial-state-def encode-problem-def
  by blast
  
  done

  then have cnf-semantics \( \Delta \) (cnf ?\Phi_I)
  using cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF assms(1)]
  unfolding cnf-semantics-def encode-initial-state-def
  by blast

  } note nb_1 = this

\{

  fix \( v \)
  assume v-in-dom-i: \( v \in \text{dom} ?I \)
  moreover \{
    have v-in-variable-set: \( v \in \text{set} ((\Pi)_V) \)
    using is-valid-problem-strips-initial-of-dom assms(1) v-in-dom-i
    by auto
    hence \((\Phi S^{-1} \Pi \Delta 0) v = \text{Some} \ (\Delta \ (\text{State} \ 0 \ \text{index} ?vy v))\)
    using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms(1, 2) - v-in-variable-set]
    by fast
    } note nb_2 = this

  consider (v-initially-true) \( ?I v = \text{Some} \ True \)
  | (v-initially-false) \( ?I v = \text{Some} \ False \)
  using v-in-dom-i
  by fastforce
  hence \( ?I v = ?s v \)
proof (cases)
case v-initially-true
then obtain C
  where C ∈ cnf ?Φ₁
  and c-is: C = { (State 0 (index ?vs v))\+ }
  using cnf-of-encode-initial-state-set v-in-dom-i assms(1)
  by fastforce
hence A (State 0 (index ?vs v)) = True
  using nb₁
  unfolding clause-semantics-def
  by fastforce
thus ?thesis
  using nb₂ v-initially-true
  by presburger

next
case v-initially-false
then obtain C
  where C ∈ cnf ?Φ₁
  and c-is: C = { (State 0 (index ?vs v))⁻¹ }
  using cnf-of-encode-initial-state-set assms(1) v-in-dom-i
  by fastforce
hence A (State 0 (index ?vs v)) = False
  using nb₁
  unfolding clause-semantics-def
  by fastforce
thus ?thesis
  using nb₂ v-initially-false
  by presburger
qed

hence ?I ⊆ₘ ?s
  using map-le-def
  by blast

moreover {
  \{ 
  fix v
  assume v-in-dom-s: v ∈ dom ?s
  then have v-in-set-vs: v ∈ set ?vs
    using decode-state-at-dom[OF assms(1)]
    by simp
  have v-in-dom-I: v ∈ dom ?I
    using is-valid-problem-strips-initial-of-dom assms(1) v-in-set-vs
    by auto
  have s-v-is: (Φₛ⁻¹ Π A 0) v = Some (A (State 0 (index ?vs v)))
    using decode-state-at-encoding-variables-equals-some-of-valuation-if assms(1, 2)
    v-in-set-vs
    by (metis le0)
  \}
  \}
}
consider \((s\text{-}v\text{-}is\text{-}some\text{-}true) \, ?s \, v = \text{Some True}\)
\| \,(s\text{-}v\text{-}is\text{-}some\text{-}false) \, ?s \, v = \text{Some False}\)
using \(v\text{-}in\text{-}dom\text{-}s\)
by fastforce
hence \(?s \, v = ?I \, v\)

proof (cases)
\begin{itemize}
\item case \(s\text{-}v\text{-}is\text{-}some\text{-}true\)
\begin{itemize}
\item then have \(A\text{-}of\text{-}s\text{-}v: \text{lit\text{-}semantics} \, A \, ((\text{State 0} \, (\text{index} \, ?vs \, v))^+)\)
\item using \(s\text{-}v\text{-}is\)
\item by fastforce
\end{itemize}
\item consider \((I\text{-}v\text{-}is\text{-}some\text{-}true) \, ?I \, v = \text{Some True}\)
\| \,(I\text{-}v\text{-}is\text{-}some\text{-}false) \, ?I \, v = \text{Some False}\)
\item using \(v\text{-}in\text{-}dom\text{-}I\)
\item by fastforce
\end{itemize}
thus \(?thesis\)

proof (cases)
\begin{itemize}
\item case \(I\text{-}v\text{-}is\text{-}some\text{-}true\)
\item then show \(?thesis\)
\item using \(s\text{-}v\text{-}is\text{-}some\text{-}true\)
\item by argo
\item hence \(lit\text{-}semantics \, A \, ((\text{State 0} \, (\text{index} \, ?vs \, v))^{-1})\)
\item using nb1
\item unfolding clause\text{-}semantics\text{-}def
\item by fast
\item thus \(?thesis\)
\item using \(A\text{-}of\text{-}s\text{-}v\)
\item by fastforce
\item qed
\end{itemize}

next
\begin{itemize}
\item case \(s\text{-}v\text{-}is\text{-}some\text{-}false\)
\item then have \(A\text{-}of\text{-}s\text{-}v: \text{lit\text{-}semantics} \, A \, ((\text{State 0} \, (\text{index} \, ?vs \, v))^{-1})\)
\item using \(s\text{-}v\text{-}is\)
\item by fastforce
\item consider \((I\text{-}v\text{-}is\text{-}some\text{-}true) \, ?I \, v = \text{Some True}\)
\| \,(I\text{-}v\text{-}is\text{-}some\text{-}false) \, ?I \, v = \text{Some False}\)
\item using \(v\text{-}in\text{-}dom\text{-}I\)
\item by fastforce
\item thus \(?thesis\)
\item proof (cases)
\item case \(I\text{-}v\text{-}is\text{-}some\text{-}true\)
\item then obtain \(C\)
\item where \(C\text{-}in\text{-}encode\text{-}initial\text{-}state: \, C \in \text{cnf} \, ?\Phi_1\)
\item and \(C\text{-}is: \, C = \{(\text{State 0} \, (\text{index} \, ?vs \, v))^{-1} \}\)
\item using cnf\text{-}of\text{-}encode\text{-}initial\text{-}state\text{-}set\text{\text{-}set} \, \text{assms}(I) \, v\text{-}in\text{-}dom\text{-}I\)
\item by fastforce
\item hence \(lit\text{-}semantics \, A \, ((\text{State 0} \, (\text{index} \, ?vs \, v))^{-1})\)
\item using nb1
\item unfolding clause\text{-}semantics\text{-}def
\item by fast
\item thus \(?thesis\)
\item using \(A\text{-}of\text{-}s\text{-}v\)
\item by fastforce
\item qed
\end{itemize}
next
\begin{itemize}
\item case \(s\text{-}v\text{-}is\text{-}some\text{-}false\)
\item then have \(A\text{-}of\text{-}s\text{-}v: \text{lit\text{-}semantics} \, A \, ((\text{State 0} \, (\text{index} \, ?vs \, v))^{-1})\)
\item using \(s\text{-}v\text{-}is\)
\item by fastforce
\item consider \((I\text{-}v\text{-}is\text{-}some\text{-}true) \, ?I \, v = \text{Some True}\)
\| \,(I\text{-}v\text{-}is\text{-}some\text{-}false) \, ?I \, v = \text{Some False}\)
\item using \(v\text{-}in\text{-}dom\text{-}I\)
\item by fastforce
\item thus \(?thesis\)
\item proof (cases)
\item case \(I\text{-}v\text{-}is\text{-}some\text{-}true\)
\item then obtain \(C\)
where $C$-in-encode-initial-state: $C \in \text{cnf} \ ?\Phi$, 
and $C$-is: $C = \{ \text{State 0 (index ?vs v)}^+ \}$ 
using cnf-of-encode-initial-state-set assms(1) v-in-dom-I
by fastforce

hence lit-semantics $A ((\text{State 0 (index ?vs v)})^+)$ 
using nb$_1$
unfolding clause-semantics-def
by fast
thus ?thesis 
using A-of-s-v
by fastforce

next 
case I-v-is-some-false 
thus ?thesis 
using s-v-is-some-false 
by presburger
qed

next 
case I-v-is-some-false 
thus ?thesis 
using s-v-is-some-false 
by presburger
qed

hence ?s $\subseteq_m$ ?I 
using map-le-def
by blast

ultimately show ?thesis 
using map-le-antisym 
by blast

qed

lemma decode-state-at-goal-state: 
assumes is-valid-problem-strips II 
and $A |\iff \Phi$ II $t$
shows $(\Pi)_G \subseteq_m \Phi_{S^{-1}}$ II $A$ $t$
proof 

let ?vs = strips-problem.variables-of II 
and $?G = (\Pi)_G$
and $?G' = \Phi_{S^{-1}}$ II $A$ $t$
and $?\Phi = \Phi$ II $t$
and $?\Phi_G = (\Phi_G$ II) $t$

{ 
  have is-cnf $?\Phi_G$ and cnf $?\Phi_G \subseteq$ cnf $?\Phi$
  subgoal 
    using encode-goal-state-is-cnf[OF assms(1)]
    by simp
  subgoal 
    using cnf-of-encode-problem-structure(2)
    unfolding encode-goal-state-def encode-problem-def
    by blast
  done 
then have cnf-semantics $A (\text{cnf} \ ?\Phi_G)$ 
  using cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF
\[
\text{assms(1)}
\]
\[
\text{assms(2)}
\]
by blast
hence \( \forall C \in \text{cnf } ?\Phi_G. \text{ clause-semantics } \mathcal{A} C \)
unfolding cnf-semantics-def encode-initial-state-def
by blast
\}
note nb_1 = this

\{
\ 
fix \ v
assume \ v \in \text{ set } ((\Pi)_V) 
major have \ ?v \neq \{\}
using calculation(1)
by fastforce

moreover have \( (\Phi_{S^{-1}} \Pi \mathcal{A} t) \)
= map-of \( (\lambda v. (\mathcal{A} (\text{State} t (\text{index } ?v v)))) ?vs \)
unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
by metis

ultimately have \( (\Phi_{S^{-1}} \Pi \mathcal{A} t) v = \text{ Some } (\mathcal{A} (\text{State} t (\text{index } ?v v))) \)
using map-of-from-function-graph-is-some-if
by fastforce
\}
note nb_2 = this
\{
\ 
fix \ v
assume \ v \in \text{dom } G: v \in \text{dom } ?G 
then have \ v-in-vs: v \in \text{set } ?vs 
using is-valid-problem-dom-of-goal-state-is assms(1)
by auto 
then have decode-state-at-is: \( (\Phi_{S^{-1}} \Pi \mathcal{A} t) v = \text{ Some } (\mathcal{A} (\text{State} t (\text{index } ?v v))) \)
using nb_2
by fastforce

consider \( (A) ?G v = \text{ Some True} \ |
(B) ?G v = \text{ Some False} 
using v-in-dom-G
by fastforce 
hence \( ?G v = ?G' v \)
proof (cases)
case A
\{
\ 
obtain C \text{ where } C \subseteq \text{cnf } ?\Phi_G \text{ and } C = \{ (\text{State} t (\text{index } ?v v))^+ \} \}
using cnf-of-encode-goal-state-set(1)[OF assms(1) v-in-dom-G] A
by auto 
then have \( \{ (\text{State} t (\text{index } ?v v))^+ \} \in \text{cnf } ?\Phi_G \)
by blast 
then have clause-semantics \( \mathcal{A} \{ (\text{State} t (\text{index } ?v v))^+ \} \)
using nb_1 
by blast

222
then have \( \text{lit-semantics } A (\text{(State } t \text{ (index } ?v \text{ ?s) +}) \) \\
unfolding \text{clause-semantics-def} \\
by \text{blast} \\
hence A (\text{(State } t \text{ (index } ?v \text{ ?s)}) = \text{True} \\
by \text{force} \\
} \\
\text{thus } \text{?thesis} \\
using \text{decode-state-at-is } A \\
by \text{presburger} \\
\text{next} \\
\text{case } B \\
\{ \\
\text{obtain } C \text{ where } C \subseteq \text{cnf } \Phi_G \text{ and } C = \{ (\text{(State } t \text{ (index } ?v \text{ ?s))}^{-1} ) \} \\
using \text{cnf-of-encode-goal-state-set}(2)(\text{OF assms(1) } \text{v-in-dom-G}) \text{ B} \\
by \text{auto} \\
then have \{ (\text{(State } t \text{ (index } ?v \text{ ?s)))}^{-1} \} \in \text{cnf } \Phi_G \\
by \text{blast} \\
then have \text{clause-semantics } A \{ (\text{(State } t \text{ (index } ?v \text{ ?s)))}^{-1} \} \\
using \text{nb}_1 \\
by \text{blast} \\
then have \text{lit-semantics } A ((\text{(State } t \text{ (index } ?v \text{ ?s)))}^{-1} \) \\
unfolding \text{clause-semantics-def} \\
by \text{blast} \\
hence A (\text{(State } t \text{ (index } ?v \text{ ?s)}) = \text{False} \\
by \text{simp} \\
\} \\
\text{thus } \text{?thesis} \\
using \text{decode-state-at-is } B \\
by \text{presburger} \\
\text{qed} \\
\} \\
\text{thus } \text{?thesis} \\
using \text{map-le-def} \\
by \text{blast} \\
\text{qed}

— Show that the operator activation implies precondition constraints hold at every time step of the decoded plan.

\textbf{lemma } \text{decode-state-at-preconditions:} \\
\textbf{assumes } \text{is-valid-problem-strips } II \\
\text{and } A \models \Phi II t \\
\text{and } k < t \\
\text{and } op \in \text{set } ((\Phi^{-1} II A) t) ! k \\
\text{and } v \in \text{set } (\text{precondition-of } op) \\
\textbf{shows } A (\text{(State } k \text{ (index } \text{strips-problem.variables-of } II \text{ ) } v)) \\
\textbf{proof} – \\
\text{let } ?ops = \text{strips-problem.operators-of } II \\
\text{and } ?vs = \text{strips-problem.variables-of } II \\
\text{let } ?\Phi = \Phi II t
and \( \Phi_O = \text{encode-operators} \ II \ t \)
and \( \Phi_P = \text{encode-all-operator-preconditions} \ II \ ?ops \ t \)

\[
\begin{align*}
&\{ \\
&\text{have } A \ (\text{Operator } k \ (\text{index } ?ops \ op)) \\
&\quad \text{and } op \in \text{set} \ ((\II)_O) \\
&\quad \text{using } \text{decode-plan-step-element-then}[\text{OF} \ \text{assms}(3, 4)] \\
&\quad \text{by } \text{blast} + \\
&\text{moreover obtain } C \\
&\quad \text{where } \text{clause-is-in-operator-encoding}: C \in \text{cnf} \ ?\Phi_P \\
&\quad \text{and } C = \{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \\
&\quad \text{(State } k \ (\text{index } ?vs \ v))^{+}\} \\
&\quad \text{using } \text{cnf-of-encode-all-operator-preconditions-contains-clause-if}[\text{OF} \ \text{assms}(1, 3)] \\
&\quad \text{calculation}(2) \ \text{assms}(5)] \\
&\quad \text{by } \text{blast} \\
&\text{moreover have } \text{clause-semantics-}A\cdot\Phi_P: \forall C \in \text{cnf} \ ?\Phi_P. \ \text{clause-semantics } A \\
&\quad \text{using } \text{cnf-semantics-monotonous-in-cnfsubsets-if}[\text{OF} \ \text{assms}(2)] \\
&\quad \text{is-cnf-encode-problem}[\text{OF} \ \text{assms}(1)] \\
&\quad \text{cnf-of-operator-precondition-encoding-subset-encoding}] \\
&\text{unfolding } \text{cnf-semantics-def} \\
&\quad \text{by } \text{blast} \\
&\text{ultimately have } \text{lit-semantics } A \ (\text{Pos } (\text{State } k \ (\text{index } ?vs \ v))) \\
&\quad \text{unfolding } \text{clause-semantics-def} \\
&\quad \text{by } \text{fastforce} \\
&\text{thus } \text{thesis} \\
&\quad \text{unfolding } \text{lit-semantics-def} \\
&\quad \text{by } \text{fastforce} \\
\end{align*}
\]

qed

— This lemma shows that for a problem encoding with makespan zero for which a model exists, the goal state encoding must be subset of the initial state encoding. In this case, the state variable encodings for the goal state are included in the initial state encoding.

**lemma** \( \text{encode-problem-parallel-correct-i} \):
**assumes** \( \text{is-valid-problem-strips} \ II \)
and \( A \models \Phi \ II \ 0 \)
**shows** \( \text{cnf } ((\Phi_G \ II) \ 0) \subseteq \text{cnf } (\Phi_I \ II) \)
**proof —**
let \( ?vs = \text{strips-problem.variables-of} \ II \)
and \( ?I = (\II)_I \)
and \( ?G = (\II)_G \)
and \( ?\Phi_I = \Phi_I \ II \)
and \( ?\Phi_G = (\Phi_G \ II) \ 0 \)
and \( ?\Phi = \Phi \ II \ 0 \)
— Show that the model of the encoding is also a model of the partial encodings.

**Proof:**

1. **A-models-\(\Phi_I\):** \(A \models \Phi_I\)
2. **A-models-\(\Phi_G\):** \(A \models \Phi_G\)

Using `assms(2)` and `encode-problem-has-model-then-also-partial-encodings(1, 2)`

By **blast**+

— Show that every clause in the CNF of the goal state encoding \(\Phi_G\) is also in the CNF of the initial state encoding \(\Phi_I\) thus making it a subset. We can conclude this from the fact that both \(\Phi_I\) and \(\Phi_G\) contain singleton clauses—which must all be evaluated to true by the given model \(A\)—and the similar structure of the clauses in both partial encodings.

By extension, if we decode the goal state \(G\) and the initial state \(I\) from a model of the encoding, \(G v = I v\) must hold for variable \(v\) in the domain of the goal state.

```plaintext
{ fix C' assume C'-in-cnf-\(\Phi_G\): C' \(\in\) cnf \(\Phi_G\) then obtain v where v-in-vs: v \(\in\) set ?vs 
  and G-of-v-is-not-None: ?G v \(\neq\) None 
  and C'-is: C' = \{ literal-formula-to-literal (encode-state-variable 0 (index ?vs v) \(\{?G\ v\}\) \} using cnf-of-encode-goal-state-set-ii[OF assms(1)] by auto obtain C where C-in-cnf-\(\Phi_I\): C \(\in\) cnf \(\Phi_I\) 
  and C-is: C = \{ literal-formula-to-literal (encode-state-variable 0 (index ?vs v) \(\{?I\ v\}\) \} using cnf-of-encode-initial-state-set-ii[OF assms(1)] v-in-vs by auto 
  { let ?L = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) \(\{?I\ v\}\) have \{ ?L \} \(\in\) cnf \(\Phi_I\) using C-in-cnf-\(\Phi_I\) C-is by blast hence lit-semantics \(\mathcal{A}\) ?L using model-then-all-singleton-clauses-modelled[OF `is-cnf-encode-initial-state[OF assms(1)]-A-models-\(\Phi_I\)`] by blast } note lit-semantics-\(\mathcal{A}\) ?L = this 
  { let ?L' = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) ?G have \{ ?L' \} \(\in\) cnf \(\Phi_G\) using C'-in-cnf-\(\Phi_G\) C'-is by blast hence lit-semantics \(\mathcal{A}\) ?L' using model-then-all-singleton-clauses-modelled[OF }
```
encode-goal-state-is-cnf [OF assms (1)] A-models-\Phi_G

by blast
}

note lit-semantics-A-L' = this
{

have \( \exists I v = ?G v \)

proof (rule ccontr)

assume contradiction: \( \exists I v \neq ?G v \)

moreover have \( \exists I v \neq None \)

using v-in-vs is-valid-problem-strips-initial-of-dom assms (1)

by auto

ultimately consider (A) \( \exists I v = \text{Some True} \land ?G v = \text{Some False} \)

| (B) \( \exists I v = \text{Some False} \land ?G v = \text{Some True} \)

using G-of-v-is-not-None

by force

thus False


unfolding encode-state-variable-def

by (cases, fastforce+)

qed
}

hence \( C' \in \text{cnf } ?\Phi_I \)

using C-is C-in-cnf-\Phi_I C'-is C'-in-cnf-\Phi_G

by argo
}

thus ?thesis

by blast

qed

--- Show that the encoding secures that for every parallel operator \( ops \) decoded from the plan at every time step \( t < \text{length } pi \) the following hold:

1. \( ops \) is applicable, and
2. the effects of \( ops \) are consistent.

lemma encode-problem-parallel-correct-ii:

assumes is-valid-problem-strips \( \Pi \)

and \( \mathcal{A} \models \Phi \Pi t \)

and \( k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t) \)

shows are-all-operators-applicable (\( \Phi_S^{-1} \Pi \mathcal{A} k )

( (\Phi^{-1} \Pi \mathcal{A} t) ! k )

and are-all-operator-effects-consistent ( (\Phi^{-1} \Pi \mathcal{A} t) ! k )

proof —

let \( \exists vs = \text{strips-problem.variables-of } \Pi \)

and \( \exists ops = \text{strips-problem.operaters-of } \Pi \)

and \( \exists I = \Phi^{-1} \Pi \mathcal{A} t \)

and \( \exists S = \Phi_S^{-1} \Pi \mathcal{A} k \)

let \( \exists \Phi = \Phi \Pi t \)

and \( \exists \Phi_E = \text{encode-all-operator-effects } \Pi \exists ops t \)

have k-ll-t: \( k < t \)
using decode-plan-length assms(3)
by metis

{ fix op v
assume op-in-kth-of-decoded-plan-set: op ∈ set (?π ! k)
and v-in-precondition-set: v ∈ set (precondition-of op)

have A (Operator k (index ?ops op))
using decode-plan-step-element-then[OF k-lt-t op-in-kth-of-decoded-plan-set]
by blast
hence A (State k (index ?vs v))
using decode-state-at-preconditions
[OF assms(1, 2) - op-in-kth-of-decoded-plan-set v-in-precondition-set]
k-lt-t
by blast
}
moreover have k ≤ t
using k-lt-t
by auto
moreover
{ have op ∈ set ((Π)O)
using decode-plan-step-element-then[OF k-lt-t op-in-kth-of-decoded-plan-set]
by simp
then have v ∈ set ((Π)V)
using is-valid-problem-strips-operator-variable-sets(1) assms(1)
v-in-precondition-set
by auto
}
ultimately have (Φ⁻¹S Π A k) v = Some True
using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
assms(1, 2)]
by presburger
}
hence are-all-operators-applicable ?s (?π ! k)
using are-all-operators-applicable-set[of ?s ?π ! k]
by blast
}
moreover { fix op₁ op₂
assume op₁-in-k-th-of-decoded-plan: op₁ ∈ set ((Φ⁻¹ Π A t) ! k)
and op₂-in-k-th-of-decoded-plan: op₂ ∈ set ((Φ⁻¹ Π A t) ! k)
have op₁-in-set-ops: op₁ ∈ set ((Π)O)
and op₂-in-set-ops: op₂ ∈ set ((Π)O)
and op₁-active-at-k: ¬lit-semantics A ((Operator k (index ?ops op₁))⁻¹)
and op₂-active-at-k: ¬lit-semantics A ((Operator k (index ?ops op₂))⁻¹)
subgoal
using decode-plan-step-element-then[OF k-lt-t op₁-in-k-th-of-decoded-plan]
by simp
}
subgoal using decode-plan-step-element-then[OF k-lt-t op₂-in-k-th-of-decoded-plan]
by force
subgoal using decode-plan-step-element-then[OF k-lt-t op₁-in-k-th-of-decoded-plan]
by simp
subgoal using decode-plan-step-element-then[OF k-lt-t op₂-in-k-th-of-decoded-plan]
by simp
done

\{
  fix \( v \)
  assume v-in-add-effects-set-of-op₁: \( v \in \text{set (add-effects-of op₁)} \)
  and v-in-delete-effects-set-of-op₂: \( v \in \text{set (delete-effects-of op₂)} \)
  let \( ?C₁ = \{ \text{(Operator k (index \( ?ops \) op₁))}^{-1}, \)
     (State (Suc k) (index \( ?vs \) v)) \}\)
  and \( ?C₂ = \{ \text{(Operator k (index \( ?ops \) op₂))}^{-1}, \)
     (State (Suc k) (index \( ?vs \) v)) \}^{-1} \}
  have \( ?C₁ \in \text{cnf ?Φ} \) and \( ?C₂ \in \text{cnf ?Φ} \)
subgoal using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[OF
assms(1) k-lt-t op₁-in-set-ops v-in-add-effects-set-of-op₁]
by blast
subgoal using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[OF
assms(1) k-lt-t op₂-in-set-ops v-in-delete-effects-set-of-op₂]
by blast
done
then have \( ?C₁ \in \text{cnf ?Φ} \) and \( ?C₂ \in \text{cnf ?Φ} \)
  using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
  by blast+
then have \( C₁\text{-true}: \text{clause-semantics } \mathcal{A} ?C₁ \) and \( C₂\text{-true}: \text{clause-semantics } \mathcal{A} ?C₂ \)
using valuation-models-encoding-cnf-formula-equals[OF assms(1)] assms(2)
  unfolding cnf-semantics-def
by blast+
have lit-semantics \( \mathcal{A} ((\text{State (Suc k) (index ?vs v)})^{+}) \)
  and lit-semantics \( \mathcal{A} ((\text{State (k + 1) (index ?vs v)})^{-1}) \)
subgoal using op₁-active-at-k C₁\text{-true}
unfolding clause-semantics-def
by blast
subgoal using op₂-active-at-k C₂\text{-true}
unfolding clause-semantics-def
by fastforce
done
hence False

228
by auto
}

} moreover {
fix v
assume v-in-delete-effects-set-of-op: v ∈ set (delete-effects of op1)
and v-in-add-effects-set-of-op2: v ∈ set (add-effects of op2)

let ?C_1 = \{(Operator k (index ?ops op_1))^{-1}, (State (Suc k) (index ?vs v))\}^{-1}
and ?C_2 = \{(Operator k (index ?ops op_2))^{-1}, (State (Suc k) (index ?vs v))\}^{+}

have ?C_1 ∈ cnf ?Φ_E and ?C_2 ∈ cnf ?Φ_E
subgoal
using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[OF
assms(1) k-lt-t op_1-in-set-ops v-in-delete-effects-set-of-op_1]
by fastforce
subgoal
using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[OF
assms(1) k-lt-t op_2-in-set-ops v-in-add-effects-set-of-op_2]
by simp
done
then have ?C_1 ∈ cnf ?Φ and ?C_2 ∈ cnf ?Φ
using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
by blast+
then have C_1-true: clause-semantics A ?C_1 and C_2-true: clause-semantics A ?C_2
using valuation-models-encoding-cnf-formula-equals[OF assms(1)] assms(2)
unfolding cnf-semantics-def
by blast+

have lit-semantics A ((State (Suc k) (index ?vs v))^{-1})
and lit-semantics A ((State (k + 1) (index ?vs v))^{+})
subgoal
using op_1-active-at-k C_1-true
unfolding clause-semantics-def
by blast
subgoal
using op_2-active-at-k C_2-true
unfolding clause-semantics-def
by fastforce
done
hence False
by simp
}

ultimately have set (add-effects of op_1) ∩ set (delete-effects of op_2) = {}
and set (delete-effects of op_1) ∩ set (add-effects of op_2) = {}
by blast+

hence are-all-operator-effects-consistent (?π ! k)
using are-all-operator-effects-consistent-set[of ?π ! k]
by blast
}
ultimately show are-all-operators-applicable \( ?s (?\pi ! k) \)
and are-all-operator-effects-consistent \( (?\pi ! k) \)
by blast+

qed

— Show that for all operators \( op \) at timestep \( k \) of the plan \( \Phi^{-1} \Pi \A t \) decoded from the model \( A \), both add effects as well as delete effects will hold in the next timestep \( S u c k \).

**lemma** encode-problem-parallel-correct-iii:

**assumes** is-valid-problem-strips \( \Pi \)
and \( \A \models \Phi \Pi t \)
and \( k < \text{length} (\Phi^{-1} \Pi \A t) \)
shows \( v \in \text{set} (\text{add-effects-of} \ op) \)
\[ \rightarrow (\Phi S^{-1} \Pi \A (S u c k)) v = \text{Some True} \]
and \( v \in \text{set} (\text{delete-effects-of} \ op) \)
\[ \rightarrow (\Phi S^{-1} \Pi \A (S u c k)) v = \text{Some False} \]

**proof** —

let \( ?\ops = \text{strips-problem.operators-of} \Pi \)
and \( ?vs = \text{strips-problem.variables-of} \Pi \)
let \( ?\Phi_F = \text{encode-all-operator-effects} \Pi \?\ops t \)
and \( ?\A = \left( \bigcup (t, \ op) \in \{\theta..<t\} \times \text{set} (\Pi) \right) \)
\[ \begin{cases} (\text{Operator} \ t (\text{index} \ ?\ops \ op))^{-1}, \text{set}(\text{State} \ (S u c k) \ (\text{index} \ ?vs \ v))^{+} \end{cases} \]
| \( v, v \in \text{set} (\text{add-effects-of} \ op) \))

and \( ?\B = \left( \bigcup (t, \ op) \in \{\theta..<t\} \times \text{set} (\Pi) \right) \)
\[ \begin{cases} (\text{Operator} \ t (\text{index} \ ?\ops \ op))^{-1}, \text{set}(\text{State} \ (S u c k) \ (\text{index} \ ?vs \ v))^{-1} \end{cases} \]
| \( v, v \in \text{set} (\text{delete-effects-of} \ op) \))

have \( k\lt-t; k < t \)
using decode-plan-length assms(3)
by metis

have \( op\lt-\text{valid}; op \in \text{set} (\Pi) \)
using decode-plan-step-element-then\[OF k\lt-t \text{ \text{assms}(4)} \]
by blast

have \( k\lt-t; k < t \)
using \( k\lt-t \ \text{op\lt-\text{valid}} \)
by fastforce

thus \( v \in \text{set} (\text{add-effects-of} \ op) \)
\[ \rightarrow (\Phi S^{-1} \Pi \A (S u c k)) v = \text{Some True} \]
and \( v \in \text{set} (\text{delete-effects-of} \ op) \)
\[ \rightarrow (\Phi S^{-1} \Pi \A (S u c k)) v = \text{Some False} \]

**proof** (auto)

assume \( v\lt-\text{add-effect}; v \in \text{set} (\text{add-effects-of} \ op) \)
have \( \A (\text{Operator} \ k (\text{index} \ ?\ops \ op)) \)
using decode-plan-step-element-then\[OF k\lt-t \text{ \text{assms}(4)} \]
by blast

moreover {
have \( \{(\text{Operator} \ k (\text{index} \ ?\ops \ op))^{-1}, (\text{State} \ (S u c k) \ (\text{index} \ ?vs \ v))^{+}\} \)

\( \in \{(\text{Operator} \ k (\text{index} \ ?\ops \ op))^{-1}, (\text{State} \ (S u c k) \ (\text{index} \ ?vs \ v))^{-1}\} \)
\[ v, v \in \text{set } \text{(add-effects-of op)} \]
using \text{v-is-add-effect}
by blast

then have \{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}\}\in \text{?A}
using \text{k-op-included cnf-of-operator-encoding-structure}
\text{UN-iff[of \{\{\text{Operator t (index ?ops op)}^{-1}, \text{State (Suc t) (index ?vs v)}\}\} - \{0..<t\} \times \text{set ((Π)O)}\]}
by blast

then have \{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}\}\in \bigcup \text{?A}
using \text{Union-iff[of \{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}\}]}
by blast

moreover have \bigcup \text{?A} \subseteq \text{cnf ?ΦF}
using \text{cnf-of-encode-all-operator-effects-structure}
by blast
ultimately have \{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}\}\in \text{cnf ?ΦF}
using \text{in-mono[of \bigcup \text{?A cnf ?ΦF}]}
by \text{presburger}

ultimately have \text{A (State (Suc k) (index ?vs v))}
using \text{cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem}
assms(2)[unfolded valuation-models-encoding-cnf-formula-equals-corollary[\text{OF assms(1)}]]
unfolding \text{Bex-def}
by \text{fastforce}
thus (ΦS^{-1} II A (Suc k)) v = Some True
using assms(1) assms(2)
decode-state-at-encoding-variables-equals-some-of-valuation-if
is-valid-problem-strips-operator-variable-sets(2) k-lt-t op-is-valid subsetD
v-is-add-effect
by \text{fastforce}

next
assume \text{v-is-delete-effect; v \in set (delete-effects-of op)}

have \text{A (Operator k (index ?ops op))}
using decode-plan-step-element-then[\text{OF k-lt-t assms(4)}]
by blast
moreover {
have \{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}^{-1}\}\in \text{\{\{\text{Operator k (index ?ops op)}^{-1}, \text{State (Suc k) (index ?vs v)}\}^{-1}\} | v, v \in \text{set (delete-effects-of op)}\]
using \text{v-is-delete-effect}
by blast

then have \{\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))^{-1}\}\} \in ?B
  using k-op-included cnf-of-encode-all-operator-effects-structure
  UN-iff[of \{\{(\text{Operator } t \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } (\text{Suc } t) \ (\text{index } ?vs \ v))^{-1}\}\}]
  - \{\theta..<t\} \times \text{set } ((\Pi)_{\mathcal{O}})
  by blast

then have \{\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))^{-1}\}\} \in \bigcup \ ?B
  using Union-iff[of \{\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))^{-1}\}\]}
  by blast

moreover have \bigcup \ ?B \subseteq \text{cnf } ?\Phi_F
  using cnf-of-encode-all-operator-effects-structure Un-upper2[of \bigcup \ ?B \bigcup \ ?A]
  by fast
  ultimately have \{\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))^{-1}\}\} \in \text{cnf } ?\Phi_F
  using in_mono[of \bigcup \ ?B \text{cnf } ?\Phi_F]
  by presburger

ultimately have \neg A (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))
  using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
  valuation-models-encoding-cnf-formula-equals-corollary[OF assms(1)]
assms(2)
  by fastforce
moreover have Suc k \leq t
  using k-lt-t
  by fastforce
moreover have v \in \text{set } ((\Pi)_{\mathcal{V}})
  using v-is-delete-effect is-valid-problem-strips-operator-variable-sets(3) assms(1)
  op-is-valid
  by auto
ultimately show (\Phi_S^{-1} \Pi \ A (\text{Suc } k)) v = \text{Some False}
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms(1, 2)]
  by auto
qed

qed

— In broad strokes, this lemma shows that the operator frame axioms ensure that state is propagated—i.e. the valuation of a variable does not change inbetween time steps—, if there is no operator active which has an effect on a given variable a: i.e.
Now, if the disjunctions are empty—i.e. if no operator which is activated at time step \( k \) has either a positive or negative effect—, we have by simplification

\[
A \models \neg(a_i \land a_{i+1}) \lor \exists \neg a_i \lor \neg a_{i+1}
\]

\[
A \models (a_i \lor \neg a_{i+1}) \lor (a_i \lor \neg a_{i+1})
\]

hence

\[
A \models \neg(a_i \lor a_{i+1}) \land (a_i \lor \neg a_{i+1})
\]

\[
\neg A \models \{\neg a_i, a_{i+1}\}, \{a_i, \neg a_{i+1}\}
\]

The lemma characterizes this simplification.  

**lemma encode-problem-parallel-correct-iv:**

**fixes** \( \Pi \): a strips-problem

**assumes** is-valid-problem-strips \( \Pi \)

and \( A \models \Pi \ t \)

and \( k < t \)

and \( \forall \exists \ a \in \text{set} \ ((\Pi)^t \ A \ t) \ k \).

\( \exists \exists \ a \in \text{set} \ (\text{delete-effects-of op}) \lor \exists \exists \ a \in \text{set} \ (\text{add-effects-of op})) \)

**shows** cnf-semantics \( A \ {{\{ \ (\text{State k (index strips-problem.variables-of \( \Pi \)) \ v})^1 \}} \)

and \( A \ {{\{ \ (\text{State (Suc k) (index strips-problem.variables-of \( \Pi \)) \ v})^+ \}} \)

**proof**

let \( \Phi = \text{strips-problem.variables-of \( \Pi \)) \)

and \( \Phi_F = \text{strips-problem.operators-of \( \Pi \)) \)

let \( \Phi = \Phi \Pi \ t \)

and \( \Phi_F = \text{encode-all-frame-axioms \( \Pi \)) \)

and \( \Phi_k = (\Phi^{-1} \ A \ t) \ k \)

and \( A = \bigcup (k, v) \in (\{0...<t\} \times \text{set} \ ?v). \)

\{ \{ \ (\text{State k (index ?v v)})^+, (\text{State (Suc k) (index ?v v)})^1 \} \}

\{ \ (\text{Operator k (index ?ops op)})^+ \ | \ op. \ op \in \text{set} \ ?ops \lor \in \text{set} \ (\text{add-effects-of op}) \} \}

and \( B = \bigcup (k, v) \in (\{0...<t\}) \times \text{set} \ ?v. \)

\{ \{ (\text{State k (index ?v v)})^1, (\text{State (Suc k) (index ?v v)})^+ \} \}

\{ \ (\text{Operator k (index ?ops op)})^+ \ | \ op. \ op \in \text{set} \ ?ops \lor \in \text{set} \ (\text{add-effects-of op}) \} \}

and \( C = \bigcup (k, v) \in (\{0...<t\}) \times \text{set} \ ?v. \)

\{ \ (\text{State k (index ?v v)})^+, (\text{State (Suc k) (index ?v v)})^1 \}

\{ \ (\text{Operator k (index ?ops op)})^+ \ | \ op. \ op \in \text{set} \ ?ops \lor \in \text{set} \ (\text{add-effects-of op}) \} \}

---

*This part of the soundness proof is only treated very briefly in [3, theorem 3.1, p.1044]*
and $C' = \{ (\text{State } k \text{ (index } ?v \text{ v)})^{-1}, (\text{State } (\text{Suc } k) \text{ (index } ?v \text{ v}))^+ \} \\
\cup \{ (\text{Operator } k \text{ (index } ?ops \text{ op}))^+ | \text{ op. op } \in \text{ set } ?ops \wedge v \in \text{ set (delete-effects-of op)} \} \\

have $k$-$v$-included: $(k, v) \in (\{..<t\} \times \text{ set } ((\Pi) v))$
by blast 
have operator-encoding-subset-encoding: $\text{cnf } ?\Phi_F \subseteq \text{cnf } ?\Phi$
using $\text{cnf-of-encode-problem-structure}$
by fast 
— Given the premise that no operator in $\pi_k$ exists with add-effect respectively delete effect $v$, we have the following situation for the EPC (effect precondition) sets:

• assuming $\text{op}$ is in set $\text{ops}$, either $\text{op}$ is in $\pi_k$ (then it doesn’t have effect on $v$ and therefore is not in either of the sets), or if is not, then $\mathcal{A} (\text{Operator } k \text{ (index } ?ops \text{ op})) = \bot$ by definition of $\text{decode-plan}$; moreover,
• assuming $\text{op}$ is not in set $\text{ops}$—this is implicitly encoded as $\text{Operator } k \text{ (length } ?ops)$ and $\mathcal{A} (\text{Operator } k \text{ (length } ?ops))$ may or may not be true—, then it’s not in either of the sets.

. Altogether, we have the situation that the sets only have members $\text{Operator } k \text{ (index } ?ops \text{ op})$ with $\mathcal{A} (\text{Operator } k \text{ (index } ?ops \text{ op})) = \bot$, hence the clause can be reduced to the state variable literals.

More concretely, the following proof block shows that the following two conditions hold for the operators:

$$\forall \text{ op. op } \in \{ ((\text{Operator } k \text{ (index } ?ops \text{ op}))^+ \text{ | op. op } \in \text{ set } ?ops \wedge v \in \text{ set (add-effects-of op)}} \} \\
\rightarrow \neg \text{lit-semantics } \mathcal{A} \text{ op}$$

and

$$\forall \text{ op. op } \in \{ ((\text{Operator } k \text{ (index } ?ops \text{ op}))^+ \text{ | op. op } \in \text{ set } ?ops \wedge v \in \text{ set (delete-effects-of op)}} \} \\
\rightarrow \neg \text{lit-semantics } \mathcal{A} \text{ op}$$

Hence, the operators are irrelevant for $\text{cnf-semantics } \mathcal{A} \{ C \}$ where $C$ is a clause encoding a positive or negative transition frame axiom for a given variable $v$ of the problem.

{$$
\text{let } ?\text{add} = \{ ((\text{Operator } k \text{ (index } ?ops \text{ op}))^+ \text{ | op. op } \in \text{ set } ?ops \wedge v \in \text{ set (add-effects-of op)}} \} \\
\text{and } ?\text{delete} = \{ ((\text{Operator } k \text{ (index } ?ops \text{ op}))^+ \text{ | op. op } \in \text{ set } ?ops \wedge v \in \text{ set (delete-effects-of op)}} \} \\
$$

{ fix $op$ 
assume operator-encoding-in-add: $(\text{Operator } k \text{ (index } ?ops \text{ op}))^+ \in ?\text{add}$}
hence \(~\text{lit-semantics} \ A \ ((\text{Operator } k \ (\text{index } \ ?ops \ op))^+)\)

proof (cases \(op \in \text{set } ?\pi_k\))

\(\text{case True}\)
then have \(v \notin \text{set } \text{(add-effects-of } op)\)
using assms(\(5\))
by simp
then have \((\text{Operator } k \ (\text{index } \ ?ops \ op))^+ \notin \ ?add\)
by fastforce
thus \(?\text{thesis}\)
using operator-encoding-in-add
by blast

next
\(\text{case False}\)
then show \(?\text{thesis}\)
proof (cases \(op \in \text{set } ?\text{ops}\))

\(\text{case True}\)

\{
let \(?A = \{ \ ?ops ! \ \text{index } \ ?ops \ op \mid \op \in \text{set } ((\Pi)_{\sigma}) \land \ A \ (\text{Operator } k \ (\text{index } \ ?ops \ op))\}\)
assume \text{lit-semantics} \ A \ ((\text{Operator } k \ (\text{index } \ ?ops \ op))^+)
moreover have \(\text{operator-active-at-}: \ A \ (\text{Operator } k \ (\text{index } \ ?ops \ op))\)
using calculation
by auto
moreover have \(op \in \text{set } ((\Pi)_{\sigma})\)
by force
moreover have \(\text{(/?ops ! \ \text{index } \ ?ops \ op) } \in ?A\)
using calculation(\(2, 3\))
by blast
ultimately have \(op \in \text{set } ?\pi_k\)
using decode-plan-step-element-then-\(\sigma\)[OF assms(\(3\))]
by auto
hence \(\text{False}\)
using \(\text{False}\)
by blast
\}
thus \(?\text{thesis}\)
by blast

next
\(\text{case False}\)
then have \(op \notin \{ \op \in \text{set } \text{ops} \cdot \ v \in \text{set } \text{(add-effects-of } \op)\}\)
by blast
moreover have \(?\text{add} =\)
\(\lambda \op. \ (\text{Operator } k \ (\text{index } \ ?ops \ op))^+\)
\(\{ \ \op \in \text{set } \text{ops} \cdot \ v \in \text{set } \text{(add-effects-of } \op) \}\)
using setcompr-eq-image[of \(\lambda \op. \ (\text{Operator } k \ (\text{index } \ ?ops \ op))^+\]
\(\lambda \op. \ \op \in \text{set } \text{ops} \land \ v \in \text{set } \text{(add-effects-of } \op)\]
by blast

235
ultimately have $(\text{Operator } k (\text{index } ?ops \ op))^+ \notin ?add$

by force

thus $?thesis$ using operator-encoding-in-add

by blast

qed

} moreover {

fix $op$

assume operator-encoding-in-delete: $((\text{Operator } k (\text{index } ?ops \ op))^+) \in ?delete$

hence $\lnot$lit-semantics $A ((\text{Operator } k (\text{index } ?ops \ op))^+)$

proof (cases $op \in \text{set } ?\pi_k$)

- case $True$
  then have $v \notin \text{set } (\text{delete-effects-of } op)$
    using assms(5)
    by simp
  then have $(\text{Operator } k (\text{index } ?ops \ op))^+ \notin ?delete$
    by fastforce
  thus $?thesis$
    using operator-encoding-in-delete
    by blast

next

- case $False$
  then show $?thesis$
  proof (cases $op \in \text{set } ?ops$)
    - case $True$
      { let $?A = \{ ?ops | index ?ops \ op \}.$
        assume $op \in \text{set } ((\Pi)\circ) \land A (\text{Operator } k (\text{index } ?ops \ op))$}
        moreover have operator-active-at-k: $A (\text{Operator } k (\text{index } ?ops \ op))$
          using calculation
          by auto
        moreover have $op \in \text{set } ((\Pi)\circ)$
          using $True$
          by force
        moreover have $(?ops | index ?ops \ op) \in ?A$
          using calculation($2$, $3$)
          by blast
        ultimately have $op \in \text{set } ?\pi_k$
          using decode-plan-step-element-then-i[OF assms($3$)]
          by auto
        hence $False$
          using $False$
          by blast
      }
    thus $?thesis$
      by blast

next

- case $False$
then have \( \text{op} \not\in \{ \text{op} \in \text{set} \ ?\text{ops} \ . \ v \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \} \)
by blast

moreover have \(?\text{delete} = (\lambda \text{op}. (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+)\)\(^+\)
| \text{op} \in \text{set} \ ?\text{ops} \ \land \ v \in \text{set} \ (\text{add-effects-of} \ \text{op}) \}
using setcomp-eq-image[of \ (\lambda \text{op}. (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+)\]
by blast

ultimately have \((\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+) \not\in \ ?\text{delete} \)
by force
thus \(?\text{thesis} \ using \ operator-encoding-in-delete \)
by blast
qed

|\}
ultimately have \(\forall \text{op} \ . \ \text{op} \in \ ?\text{add} \rightarrow \neg \text{lit-semantics} \ A \ \text{op} \)
and \(\forall \text{op} \ . \ \text{op} \in \ ?\text{delete} \rightarrow \neg \text{lit-semantics} \ A \ \text{op} \)
by blast

|\}
note \(nb = this\)

let \(?\text{Ops} = \{ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \)
| \text{op} \ . \ \text{op} \in \text{set} \ ?\text{ops} \ \land \ v \in \text{set} \ (\text{add-effects-of} \ \text{op}) \} \)
have \(?\text{Ops} \subseteq \ ?\text{C} \)
by blast
moreover have \(?\text{C} - \ ?\text{Ops} = \{ (\text{State} \ k \ (\text{index} \ ?\text{vs} \ v))^+ , \ (\text{State} \ (Suc \ k) \ (\text{index} \ ?\text{vs} \ v))^{-1} \} \)
by fast
moreover have \(\forall \text{L} \in \ ?\text{Ops} . \neg \text{lit-semantics} \ A \ \text{L} \)
using \(nb(1)\)
by blast

ultimately have \text{clause-semantics} \ A \ ?\text{C} = \text{clause-semantics} \ A \ \{ (\text{State} \ k \ (\text{index} \ ?\text{vs} \ v))^+ , \ (\text{State} \ (Suc \ k) \ (\text{index} \ ?\text{vs} \ v))^{-1} \} \)
using \text{lit-semantics-reducible-to-subset-if}[of \ ?\text{Ops} \ ?\text{C}]
by presburger

moreover \{ \)
let \(?\text{Ops}' = \{ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \)
| \text{op} \ . \ \text{op} \in \text{set} \ ?\text{ops} \ \land \ v \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \} \)
have \(?\text{Ops}' \subseteq \ ?\text{C}' \)
by blast
moreover have \(?\text{C}' - \ ?\text{Ops}' = \{ (\text{State} \ k \ (\text{index} \ ?\text{vs} \ v))^{-1} , \ (\text{State} \ (Suc \ k) \ (\text{index} \ ?\text{vs} \ v))^{+} \} \)
by fast
moreover have \(\forall \text{L} \in \ ?\text{Ops}' . \neg \text{lit-semantics} \ A \ \text{L} \)
using \(nb(2)\)
by blast

237
ultimately have clause-semantics \( A \ ?C' \)
\[ = \text{clause-semantics} \ A \ [ (\text{State } k \ (\text{index } ?v s \ v))^{-1}, (\text{State } (\text{Suc } k) \ (\text{index } ?v s \ v))^{+} ] \]
using lit-semantics-reducible-to-subset-if\[ \text{of } \text{Ops} \ ?C' \]
by presburger

} moreover {
have \( \text{cnf-semantics-}A\Phi: \text{cnf-semantics} \ A \ (\text{cnf } ?\Phi) \)
using valuation-models-encoding-cnf-formula-equals\[ of \ \text{assms}(1) \ \text{assms}(2) \]
by blast
have \( k-v\text{-included}: (k, v) \in \{..<t\} \times (\Pi)_{v} \)
using \( \text{assms}(3, 4) \)
by blast

have \( c\text{-in-un-a: } ?C \in \bigcup ?A \text{ and } c'\text{-in-un-b: } ?C' \in \bigcup ?B \)
using \( \text{k-v\text{-included}} \)
by force+
then have \( ?C \in \text{cnf } ?\Phi_{F} \text{ and } ?C' \in \text{cnf } ?\Phi_{F} \)
subgoal
using \( \text{cnf-of-encode-all-frame-axioms-structure} \ UnI1[ of \ ?C \bigcup \ ?A \bigcup \ ?B \]
c-in-un-a
by metis
subgoal
using \( \text{cnf-of-encode-all-frame-axioms-structure} \ UnI2[ of \ ?C' \bigcup \ ?B \bigcup \ ?A \]
c'-in-un-b
by metis
done
then have \( \{ ?C \} \subseteq \text{cnf } ?\Phi_{F} \text{ and } c'\text{-subset-frame-axiom-encoding}: \{ ?C' \} \subseteq \text{cnf } ?\Phi_{F} \)
by blast+
then have \( \{ ?C \} \subseteq \text{cnf } ?\Phi \text{ and } \{ ?C' \} \subseteq \text{cnf } ?\Phi \)
subgoal
using \( \text{operator-encoding-subset-encoding} \)
by fast
subgoal
using \( c'\text{-subset-frame-axiom-encoding} \text{operator-encoding-subset-encoding} \)
by fast
done

hence \( \text{cnf-semantics} \ A \ { \ ?C } \text{ and } \text{cnf-semantics} \ A \ { \ ?C' } \)
using \( \text{cnf-semantics-}A\Phi \ \text{model-for-cnf-is-model-of-all-subsets} \)
by fastforce+
}
ultimately show \( \text{cnf-semantics} \ A \ { \{ (\text{State } k \ (\text{index } ?v s \ v))^{-1}, (\text{State } (\text{Suc } k) \ (\text{index } ?v s \ v))^{+} \} \}
and \( \text{cnf-semantics} \ A \ { \{ (\text{State } k \ (\text{index } ?v s \ v))^{+}, (\text{State } (\text{Suc } k) \ (\text{index } ?v s \ v))^{-1} \} \}
unfolding \( \text{cnf-semantics-def} \)
by blast+
lemma encode-problem-parallel-correct-v:
assumes is-valid-problem-strips Π
and A ⊨ Φ Π t
and k < length (Φ⁻¹ Π A t)
shows (Φ⁵⁻¹ Π A (Suc k)) = execute-parallel-operator (Φ⁻¹ Π A k) ((Φ⁻¹ Π A t) ! k)
proof
let ?vs = strips-problem.variables-of Π
and ?ops = strips-problem.operators-of Π
and ?π = Φ⁻¹ Π A t
and ?sk = Φ⁵⁻¹ Π A k
and ?sk' = Φ⁻¹ Π A (Suc k)
let ?tk' = execute-parallel-operator ?sk (?π ! k)
and ?πk = ?π ! k
have k-lt-t: k < t and k-lte-t: k ≤ t and suc-k-lte-t: Suc k ≤ t
using decode-plan-length[of ?π Π A t] assms(3)
by (argo, fastforce+)
then have operator-preconditions-hold:
are-all-operators-applicable ?sk ?πk ∧ are-all-operator-effects-consistent ?πk
using encode-problem-parallel-correct-ii[OF assms(1, 2, 3)]
by blast
— We show the goal in classical fashion by proving that
Φ⁵⁻¹ Π A (Suc k) v
= execute-parallel-operator (Φ⁻¹ Π A k)
((Φ⁻¹ Π A t) ! k) v
—i.e. the state decoded at time k + 1 is equivalent to the state obtained by
executing the parallel operator (Φ⁻¹ Π A t) ! k on the previous state Φ⁻¹ Π A k—for all variables v given k < t, a model A, and makespan t.
moreover {
{ 
fix v
assume v-in-dom-sk': v ∈ dom ?sk'
then have sk'-not-none: ?sk' v ≠ None
by blast
hence ?sk' v = ?tk' v
proof (cases ∃ op ∈ set ?πk. v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op))
case True
then obtain op
where op-in-πk: op ∈ set ?πk
and v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)
by blast
then consider (v-is-add-effect) v ∈ set (add-effects-of op)
| (v-is-delete-effect) v ∈ set (delete-effects-of op)
by blast
then show ?thesis
proof (cases)
  case v-is-add-effect
  then have \(?s_k' v = \text{Some True}\)
    using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)]
  op-in-\(\pi_k\]
    v-is-add-effect
    by blast
  moreover have are-all-operators-applicable \((\Phi_S^{-1} \Pi \mathcal{A} k) ((\Phi^{-1} \Pi \mathcal{A} t) ! k)\)
    and are-all-operator-effects-consistent \((\Phi^{-1} \Pi \mathcal{A} t) ! k\)
    using operator-preconditions-hold v-is-add-effect
    by blast
  moreover have \(?t_k' v = \text{Some True}\)
    using execute-parallel-operator-positive-effect-if [of
    \(\Phi_S^{-1} \Pi \mathcal{A} k (\Phi^{-1} \Pi \mathcal{A} t) ! k\) op-in-\(\pi_k\]
    v-is-add-effect calculation(2, 3)
    by blast
  ultimately show \(?\text{thesis}\)
    by argo
next
  case v-is-delete-effect
  then have \(?s_k' v = \text{Some False}\)
    using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)]
  op-in-\(\pi_k\]
    v-is-delete-effect
    by blast
  moreover have are-all-operators-applicable \((\Phi_S^{-1} \Pi \mathcal{A} k) ((\Phi^{-1} \Pi \mathcal{A} t) ! k)\)
    and are-all-operator-effects-consistent \((\Phi^{-1} \Pi \mathcal{A} t) ! k\)
    using operator-preconditions-hold
    by blast
  moreover have \(?t_k' v = \text{Some False}\)
    using execute-parallel-operator-effect(2) op-in-\(\pi_k\]
    v-is-delete-effect calculation(2, 3)
    by fast
  moreover have \(?t_k' v = \text{Some False}\)
    by (meson execute-parallel-operator-negative-effect-if op-in-\(\pi_k\]
    operator-preconditions-hold v-is-delete-effect)
  ultimately show \(?\text{thesis}\)
    by argo
qed

next
case False

then have \(?t_k' v = ?s_k v\)
  using execute-parallel-operator-no-effect-if
  by fastforce
moreover { have v-in-set-vs: \(v \in \text{set } ((\Pi)_{\mathcal{V}})\) }
using decode-state-at-valid-variable[OF $s_k'$-not-none].

then have state-propagation-positive:
  \(\text{cnf-semantics } \mathcal{A} \{ ((\text{State } k (\text{index } ?v\text{ } v))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?v\text{ } v))^{+}) \}\)

and state-propagation-negative:
  \(\text{cnf-semantics } \mathcal{A} \{ ((\text{State } k (\text{index } ?v\text{ } v))^+, (\text{State } (\text{Suc } k) (\text{index } ?v\text{ } v))^{-1}) \}\)

using encode-problem-parallel-correct-iv[OF assms (1, 2), k-lt-t - False]
  by fastforce

consider ($s_k'$-v-positive) $s_k'$ $v$ = Some True
  | ($s_k'$-v-negative) $s_k'$ $v$ = Some False

using $s_k'$-not-none
  by fastforce

hence $s_k'$ $v$ = $s_k$ $v$

proof (cases)
  case $s_k'$-v-positive
  then have \(\text{lit-semantics } \mathcal{A} \{ (\text{State } (\text{Suc } k) (\text{index } ?v\text{ } v))^+ \}\)
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) suc-k-lte-t v-in-set-vs]
  by fastforce

then have \(\text{lit-semantics } \mathcal{A} \{ (\text{State } k (\text{index } ?v\text{ } v))^+ \}\)
  using state-propagation-negative
  unfolding cnf-semantics-def clause-semantics-def
  by fastforce

then show ?thesis
using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) k-lte-t v-in-set-vs] $s_k'$-v-positive
  by fastforce

next
  case $s_k'$-v-negative
  then have \(\neg\text{lit-semantics } \mathcal{A} \{ (\text{State } (\text{Suc } k) (\text{index } ?v\text{ } v))^+ \}\)
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) suc-k-lte-t v-in-set-vs]
  by fastforce

then have \(\neg\text{lit-semantics } \mathcal{A} \{ (\text{State } k (\text{index } ?v\text{ } v))^+ \}\)
  using state-propagation-positive
  unfolding cnf-semantics-def clause-semantics-def
  by fastforce

then show ?thesis
using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) k-lte-t v-in-set-vs] $s_k'$-v-negative
  by fastforce

qed

ultimately show ?thesis
  by argo

qed

241
hence $s_k' \subseteq m \ ?t_k'$
using map-le-def
by blast
}
moreover {
{
fix $v$
assume $v \in \text{dom } ?t_k'$
then have $t_k'$-not-none: $?t_k' \ v \neq \text{None}$
by blast
{
{
assume contradiction: $v \notin (\{\Pi\})_v$
then have $(\Phi S^{-1} \ I I \ A \ k) \ v = \text{None}$
using decode-state-at-valid-variable
by fastforce
then obtain $op$
where $op$-in: $op \in (\{(\Phi^{-1} I I A) \! k\})$
and $v$-is-or: $v \in (\text{add-effects-of } op)$
$\forall v \in (\text{delete-effects-of } op)$
using execute-parallel-operators-strips-none-if-contraposition[OF $t_k'$-not-none]
by blast
have $op$-in: $op \in (\{\Pi\})_O$
using $op$-in decode-plan-step-element-then(1) k-lt-t
by blast
consider $(A) \ v \in (\text{add-effects-of } op)$
$| (B) \ v \in (\text{delete-effects-of } op)$
using $v$-is-or
by blast
hence False
proof (cases)
  case $A$
  then have $v \in (\{\Pi\})_V$
  using is-valid-problem-strips-operator-variable-sets(2)[OF
  assms(1)] $op$-in $A$
  by blast
  thus False
  using contradiction
  by blast
next
  case $B$
  then have $v \in (\{\Pi\})_V$
  using is-valid-problem-strips-operator-variable-sets(3)[OF
  assms(1)] $op$-in $B$
  by blast
  thus False
  using contradiction
by blast
qed
}
)
hence v-in-set-us: v ∈ set ((II)\(v\))
by blast
hence \(?t_{k'} v = ?s_{k'} v\)
proof (cases (\(\exists op\in set \?\pi_k. v \in set (add-effects-of op) \vee v \in set (delete-effects-of op)\)))
case True
then obtain op
where op-in-set-\pi_k: op ∈ set \(?\pi_k\)
and v-options: v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)
by blast
then have op ∈ set ((II)\(\sigma\))
using decode-plan-step-element-then[OF k-lt-t]
by blast
then consider (v-is-add-effect) v ∈ set (add-effects-of op)
| (v-is-delete-effect) v ∈ set (delete-effects-of op)
using v-options
by blast
thus \(?thesis\)
proof (cases)
case v-is-add-effect
then have \(?t_{k'} v = Some True\)
using execute-parallel-operator-positive-effect-if[OF - - op-in-set-\pi_k]
operator-preconditions-hold
by blast
moreover have \(?s_{k'} v = Some True\)
using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)]
op-in-set-\pi_k
v-is-add-effect
by blast
ultimately show \(?thesis\)
by argo
next
case v-is-delete-effect
then have \(?t_{k'} v = Some False\)
using execute-parallel-operator-negative-effect-if[OF - - op-in-set-\pi_k]
operator-preconditions-hold
by blast
moreover have \(?s_{k'} v = Some False\)
using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)]
op-in-set-\pi_k
v-is-delete-effect
by blast
ultimately show \(?thesis\)
by argo
qed
next

  case False
  have state-propagation-positive:
    cnf-semantics A {{{State k (index \?vs v))^{-1}, (State (Suc k) (index \?vs v))}^+}}
    and state-propagation-negative:
    cnf-semantics A {{{State k (index \?vs v))^+, (State (Suc k) (index \?vs v))}^{-1}}
      using encode-problem-parallel-correct-iv[OF assms (1, 2) k-lt-t v-in-set-vs False]
      by blast+
    
    { have all-op-in-set-πk-have-no-effect:
      \forall op \in set \?π_k, v \notin set (add-effects-of op) \land v \notin set (delete-effects-of op)
      
      using False
      by blast
      then have \?t_k' v = \?s_k v
      using execute-parallel-operator-no-effect-if[OF all-op-in-set-πk-have-no-effect]
      by blast.
    } note t_k' equals s_k = this
    { have \?s_k v \neq None
      using t_k' not none t_k' equals s_k
      by argo
      then consider (s_k-v-is-some-true) \?s_k v = Some True
      | (s_k-v-is-some-false) \?s_k v = Some False
      by fastforce
    }
  then show \?thesis
  proof (cases)
    case s_k-v-is-some-true
    moreover {
      have lit-semantics A ((State k (index \?vs v))^+)
      using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) k-lte-t v-in-set-vs] s_k-v-is-some-true
      by simp
      then have lit-semantics A ((State (Suc k) (index \?vs v))^+)
      using state-propagation-positive
      unfolding cnf-semantics-def clause-semantics-def
      by fastforce
      then have \?s_k' v = Some True
      using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms (1, 2) suc-k-lte-t v-in-set-vs]
      by fastforce
    }
    ultimately show \?thesis
      using t_k' equals s_k
      by simp

244
next
  case $s_k \cdot v$ is some false
moreover {
  have lit-semantics $A ((\text{State } k \ (\text{index } ?vs \ v))^{-1})$
  using decode-state-at-encoding-variables-equals-some-of-valuation-if $[OF$
    assms(1, 2) $k \cdot \text{lte-t } v \in \text{-set-vs}]$ $s_k \cdot v$ is some false
  by simp
  then have lit-semantics $A ((\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))^{-1})$
    using state-propagation-negative
    unfolding cnf-semantics-def clause-semantics-def
  by fastforce
  then have $?s_k'$ $v$ = Some False
  using decode-state-at-encoding-variables-equals-some-of-valuation-if $[OF$
    assms(1, 2) suc-k-lte-t $v \in \text{-set-vs}]$
  by fastforce
}
ultimately show $?\text{thesis}$
  using $t_k'$-equals-$s_k$
  by simp
qed
qed

hence $?t_k'$ $\subseteq_m$ $?s_k'$
using map-le-def
by blast
}
ultimately show $?\text{thesis}$
  using map-le-antisym
by blast
qed

lemma encode-problem-parallel-correct-vi:
assesms is-valid-problem-strips $\Pi$
  and $A \models \Phi \ \Pi \ t$
  and $k < \text{length (trace-parallel-plan-strips } ((\Pi)_I) \ (\Phi^{-1} \ \Pi \ A \ t))$
shows trace-parallel-plan-strips $((\Pi)_I) \ (\Phi^{-1} \ \Pi \ A \ t) \ ! \ k$
  = $\Phi_S^{-1} \ \Pi \ A \ k$
using assms
proof −
  let $?I = (\Pi)_I$
  and $?\pi = \Phi^{-1} \ \Pi \ A \ t$
  let $?\tau = \text{trace-parallel-plan-strips } ?I \ ?\pi$
  show $?\text{thesis}$
    using assms
  proof (induction $k$)
    case 0
    hence $?\tau ! \ 0 = ?I$
      using trace-parallel-plan-strips-head-is-initial-state
      by blast
    qed
  qed
  qed

245
moreover have $\Phi_{S^{-1}} A 0 = ?I$
- using \textit{decode-state-at-initial-state}[OF assms(1, 2)]
  by simp
ultimately show ?case
  by simp

next
- case (Suc $k$)
  let $?\tau_k = \text{trace-parallel-plan-strips} \ ?I \ ?\pi \ ! k$
  and $?s_k = \Phi_{S^{-1}} A k$

  have $k$-lt-length-$\tau$-minus-one: $k < \text{length } ?\tau - 1$ and $k$-lt-length-$\tau$: $k < \text{length } ?\tau$
    using Suc.prems(3)
    by linarith

  — Use the induction hypothesis to obtain the proposition for the previous step $k$. Then, show that applying the $k$-th parallel operator in the plan $\pi$ on either the state obtained from the trace or decoded from the model yields the same successor state.

    { 
      have $?\tau ! k = \text{execute-parallel-plan} \ ?I \ (\text{take } k \ ?\pi)$
        using \textit{trace-parallel-plan-plan-prefix} $k$-lt-length-$\tau$
        by blast
      hence $?\tau_k = ?s_k$
        using Suc.IH[OF assms(1, 2) $k$-lt-length-$\tau$]
        by blast
    }

moreover have \text{trace-parallel-plan-strips} $\ ?I \ ?\pi \ \text{Suc} \ k$
  = \text{execute-parallel-operator} $?\tau_k (??\pi \ ! k)$
  using \textit{trace-parallel-plan-step-effect-is}[OF $k$-lt-length-$\tau$-minus-one]
  by blast
moreover {
  thm Suc.prems(3)
  have \text{length} (\text{trace-parallel-plan-strips} $\ ?I \ ?\pi$) $\leq$ \text{length } $?\pi + 1$
    using \textit{length-trace-parallel-plan-strips-lte-length-plan-plus-one}
    by blast
  then have $k < \text{length } ?\pi$
    using Suc.prems(3)
    unfolding Suc-eq-plus1
    by linarith
  hence $\Phi_{S^{-1}} A \ (\text{Suc} \ k)$
    = \text{execute-parallel-operator} $?s_k (??\pi \ ! k)$
    using \textit{encode-problem-parallel-correct-v}[OF assms(1, 2)]
    by simp
}
ultimately show ?case
  by argo
qed

lemma \textit{encode-problem-parallel-correct-vii}:
assumes \( \text{is-valid-problem-strips} \, \Pi \)
and \( \mathcal{A} \models \Phi \, \Pi \, t \)
shows \( \text{length} \left( \text{map} \left( \text{decode-state-at} \, \Pi \, \mathcal{A} \right) \left[ 0..<\text{Suc} \left( \text{length} \left( \Phi^{-1} \, \Pi \, \mathcal{A} \, t \right) \right) \right] \right) = \text{length} \left( \text{trace-parallel-plan-strips} \, ((\Pi)_I) \left( \Phi^{-1} \, \Pi \, \mathcal{A} \, t \right) \right) \)

proof –

let \( \mathcal{I} = ((\Pi)_I) \)
and \( \mathcal{F} = \Phi^{-1} \, \Pi \, \mathcal{A} \, t \)
let \( \mathcal{\sigma} = \text{map} \left( \text{decode-state-at} \, \Pi \, \mathcal{A} \right) \left[ 0..<\text{Suc} \left( \text{length} \left( \mathcal{F} \right) \right) \right] \)
and \( \mathcal{\tau} = \text{trace-parallel-plan-strips} \, \mathcal{I} \, \mathcal{\sigma} \)
let \( \mathcal{\ell} = \text{length} \, \mathcal{\tau} \)
let \( \mathcal{k} = \mathcal{\ell} - 1 \)
show \( \mathcal{\text{thesis}} \)

proof (rule ccontr)
assumes \( \text{length-}\mathcal{\sigma}-\text{neg-length-}\mathcal{\tau}: \text{length} \, \mathcal{\sigma} \neq \text{length} \, \mathcal{\tau} \)

{ have \( \text{length} \, \mathcal{\sigma} = \text{length} \, \mathcal{\tau} + 1 \)
  by fastforce
  moreover have \( \text{length} \, \mathcal{\tau} \leq \text{length} \, \mathcal{\tau} + 1 \)
    using \( \text{length-trace-parallel-plan-strips-lte-length-plan-plus-one} \)
    by blast
  moreover have \( \text{length} \, \mathcal{\tau} < \text{length} \, \mathcal{\tau} + 1 \)
    using \( \text{length-}\mathcal{\sigma}-\text{neg-length-}\mathcal{\tau} \) calculation
    by linarith
}

note \( \text{nb}_1 = \text{this} \)

{ have \( 0 < \text{length} \, \mathcal{\tau} \)
  using \( \text{trace-parallel-plan-strips-not-nil} \)
  then have \( \text{length} \, \mathcal{\tau} - 1 < \text{length} \, \mathcal{\tau} \)
    using \( \text{nb}_1 \)
    by linarith
}

note \( \text{nb}_2 = \text{this} \)

{ obtain \( \mathcal{k} \) where \( \text{length} \, \mathcal{\tau} = \text{Suc} \, \mathcal{k} \)
  using \( \text{less-imp-Suc-add}[\text{OF length-trace-parallel-plan-gt-0}] \)
  by blast
  hence \( \mathcal{k} < \text{length} \, \mathcal{\tau} \)
    using \( \text{nb}_2 \)
    by blast
}

note \( \text{nb}_3 = \text{this} \)

{ have \( \mathcal{\tau} \) \( \mathcal{k} = \text{execute-parallel-plan} \, \mathcal{I} \) \( \text{take} \) \( \mathcal{k} \) \( \mathcal{\tau} \)
  using \( \text{trace-parallel-plan-plan-prefix}[\text{OF length-trace-minus-one-lt-length-trace}] \)
  by blast
  thm \( \text{encode-problem-parallel-correct-vi}[\text{OF assms (1, 2)}] \) \( \text{nb}_3 \)
  moreover have \( \left( \Phi_{\mathcal{S}}^{-1} \, \Pi \, \mathcal{A} \right) \mathcal{k} = \mathcal{\tau} \) \( \mathcal{k} \)
    using \( \text{encode-problem-parallel-correct-vi}[\text{OF assms (1, 2)}] \)
    \( \text{length-trace-minus-one-lt-length-trace} \)
ultimately have \((\Phi S^{-1} \Pi A \ ?k) = \text{execute-parallel-plan} \ ?I \ (\text{take} \ ?k \ ?\pi)\)

by argo

} note \(nb_4 = \text{this}\)

{ have are-all-operators-applicable \((\Phi S^{-1} \Pi A \ ?k) (\ ?\pi ! \ ?k)\)
and are-all-operator-effects-consistent \((\ ?\pi ! \ ?k)\)
using encode-problem-parallel-correct-vi(1, 2)[OF assms(1, 2)] nb_3
by blast+
— Unsure why calculation(1, 2) is needed for this proof step. Should just require the default proof.

moreover have \(\neg\text{are-all-operators-applicable} \ ((\Phi S^{-1} \Pi A \ ?k) (\ ?\pi ! \ ?k)\)
and \(\neg\text{are-all-operator-effects-consistent} \ ((\ ?\pi ! \ ?k)\)
using length-trace-parallel-plan-strips-lt-length-plan-plus-one-then[OF nb_1]
calculation(1, 2)
unfolding nb_3 nb_4
by blast+
ultimately have False
by blast

} thus False.
qed

qed

lemma encode-problem-parallel-correct-x:
assumes is-valid-problem-strips II
and \(A \models \Phi II t\)
shows map (decode-state-at II A)
\([0..<\text{Suc} (\text{length} (\Phi^{-1} II A t))]\) = trace-parallel-plan-strips ((II)t) (\(\Phi^{-1} II A t\))

proof –
let \(?I = (II)_{t}\)
and \(?\pi = \Phi^{-1} II A t\)
let \(?\sigma = \text{map} (\text{decode-state-at} II A) [0..<\text{Suc} (\text{length} ?\pi)]\)
and \(?\tau = \text{trace-parallel-plan-strips} ?I ?\pi\)

{ have length ?\tau = length ?\sigma
using encode-problem-parallel-correct-vii[OF assms]..
moreover {
fix \(k\)
assume k-lt-length-\(\tau; k < \text{length} ?\tau\)
then have trace-parallel-plan-strips ((II)t) (\(\Phi^{-1} II A t\)) ! \(k\)
= \(\Phi S^{-1} II A k\)
using encode-problem-parallel-correct-vi[OF assms]
by blast
moreover {
have length ?\tau \leq length ?\pi + 1
using length-trace-parallel-plan-strips-lte-length-plan-plus-one
by blast
then have \(k < \text{length} ?\pi + 1\)
using $k$-lt-length-$\tau$
    by linarith
then have $k < \text{Suc} (\text{length } ?\pi) - 0$
    by simp
hence $?\sigma ! k = \Phi_{S^{-1}} \mathcal{A} k$
    using nth-map-upt[of $k \text{Suc} (\text{length } ?\pi) 0]$
    by auto
}
ultimately have $?\tau ! k = ?\sigma ! k$
    by argo
}
ultimately have $?\tau = ?\sigma$
    using list-eq-iff-nth-conv[of $?\tau ?\sigma]$
    by blast
}
thus $\text{thesis}$
    by argo
qed

lemma encode-problem-parallel-correct-xi:
    fixes $\Pi::'a\ \text{strips-problem}$
    assumes $\text{is-valid-problem-strips } \Pi$
        and $\mathcal{A} \models \Phi \Pi t$
        and $\text{ops} \in \text{set} (\Phi_{S^{-1}} \mathcal{A} t)$
        and $\text{op} \in \text{set} \text{ops}$
    shows $\text{op} \in \text{set} ((\Pi)\mathcal{O})$
proof -
    let $?\pi = \Phi_{S^{-1}} \mathcal{A} t$
    have $\text{length } ?\pi = t$
    using decode-plan-length
    by force
moreover obtain $k$ where $k < \text{length } ?\pi$ and $\text{ops} = ?\pi! k$
    using in-set-conv-nth[of $?\pi$] assms(3)
    unfolding calculation
    by blast
ultimately show $\text{thesis}$
    using assms(4) decode-plan-step-element-then(1)
    by force
qed

To show soundness, we have to prove the following: given the existence of
a model $\mathcal{A}$ of the basic SATPlan encoding $\Phi \Pi t$ for a given valid problem $\Pi$
and hypothesized plan length $t$, the decoded plan $\pi = \Phi_{S^{-1}} \mathcal{A} t$ is a
parallel solution for $\Pi$.
We show this theorem by showing equivalence between the execution trace
of the decoded plan and the sequence of states

$$\sigma = \text{map} \ (\lambda k. \Phi_{S^{-1}} \mathcal{A} k) [\emptyset..<\text{Suc} (\text{length } ?\pi)]$$
decoded from the model $A$. Let

$$\tau \equiv \text{trace-parallel-plan-strips } I \pi$$

be the trace of $\pi$. Theorem ?? first establishes the equality $\sigma = \tau$ of the decoded state sequence and the trace of $\pi$. We can then derive that $G \subseteq_m \text{last } \sigma$ by lemma ??, i.e. the last state reached by plan execution (and moreover the last state decoded from the model), satisfies the goal state $G$ defined by the problem. By lemma ??, we can conclude that $\pi$ is a solution for $I$ and $G$.

Moreover, we show that all operators $op$ in all parallel operators $ops \in \text{set } \pi$ are also contained in $O$. This is the case because the plan decoding function reverses the encoding function (which only encodes operators in $O$).

By definition ?? this means that $\pi$ is a parallel solution for $\Pi$. Moreover $\pi$ has length $t$ as confirmed by lemma .

**Theorem** encode-problem-parallel-sound:

**Assumes** is-valid-problem-strips $\Pi$ and $\mathcal{A} \models \Phi \Pi t$

**Shows** is-parallel-solution-for-problem $\Pi$ $(\Phi^{-1} \Pi \mathcal{A} t)$

**Proof** –

let $\theta_{ops} = \text{strips-problem-operators-of } \Pi$

and $\theta_{I} = (\Pi)_{I}$

and $\theta_{G} = (\Pi)_{G}$

and $\theta_{\pi} = \Phi^{-1} \Pi \mathcal{A} t$

let $\theta_{\sigma} = \text{map } (\lambda k. \Phi_{S}^{-1} \Pi \mathcal{A} k) [\theta..\text{Suc } (\text{length } \theta_{\pi})]$

and $\theta_{\tau} = \text{trace-parallel-plan-strips } \theta_{I} \theta_{\pi}$

{ have $\theta_{\sigma} = \theta_{\tau}$

  using encode-problem-parallel-correct-x[OF assms].

  moreover { have $\text{length } \theta_{\pi} = t$

    using decode-plan-length

    by auto

    then have $\theta_{G} \subseteq_m \text{last } \theta_{\sigma}$

    using decode-state-at-goal-state[OF assms]

    by simp

  }

  ultimately have $(\Pi)_{G} \subseteq_m \text{execute-parallel-plan } ((\Pi)_{I}) (\Phi^{-1} \Pi \mathcal{A} t)$

  using execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace

  by auto

  } moreover have $\forall ops \in \text{set } \theta_{\pi}, \forall op \in \text{set } ops. \ op \in \text{set } ((\Pi)_{O})$

  using encode-problem-parallel-correct-xi[OF assms(1, 2)]

  by auto

ultimately show $\theta_{\text{thesis}}$

\textsuperscript{10}This lemma is used in the proof but not shown.
unfolding is-parallel-solution-for-problem-def
unfolding list-all-iff ListMem-iff operators-of-def STRIPS-Representation.operators-of-def
by fastforce
qed

value stop

7.4 Completeness

definition empty-valuation :: sat-plan-variable valuation (A₀)
where empty-valuation ≡ (λ_. False)

abbreviation valuation-for-state :: 'variable list
⇒ 'variable strips-state
⇒ nat
⇒ 'variable
⇒ sat-plan-variable valuation
⇒ sat-plan-variable valuation
where valuation-for-state vs s k v A
≡ A(State k (index vs v) := (s v = Some True))

— Since the trace may be shorter than the plan length even though the last trace element subsumes the goal state—namely in case plan execution is impossible due to violation of the execution condition but the reached state serendipitously subsumes the goal state—, we also have to repeat the valuation for all time steps \( k' \in \{\text{length } \tau..\text{length } \pi + 1\} \) for all \( v \in V \) (see \( A₂ \)).

definition valuation-for-state-variables :: 'variable strips-problem
⇒ 'variable strips-operator list list
⇒ 'variable strips-state list
⇒ sat-plan-variable valuation
where valuation-for-state-variables \( \Pi \pi \tau \equiv \) let
\( t' = \text{length } \tau \)
\( \tau_\Omega = \tau ! (t' - 1) \)
\( \text{vs} = \text{variables-of } \Pi \)
\( V₁ = \{ \text{State } k \text{ (index vs v) } | \ k v k \in \{0..<t'\} \land v \in \text{set } vs \ \} \)
\( V₂ = \{ \text{State } k \text{ (index vs v) } | \ k v k \in \{t'..(\text{length } \pi + 1)\} \land v \in \text{set } vs \ \} \)
\( A₁ = \text{foldr} \)
\( (\lambda(k, v) \ A \text{ valuation-for-state (variables-of } \Pi) (\tau ! k v A) \)
\( (\text{List.product } [0..<t'] \text{ vs}) \)
\( A₀ \)
\( A₂ = \text{foldr} \)
\( (\lambda(k, v) \ A \text{ valuation-for-state (variables-of } \Pi) \tau_\Omega k v A) \)
\( (\text{List.product } [t'..<\text{length } \pi + 2] \text{ vs}) \)
\( A₀ \)
in override-on (override-on \( A₀ \ A₁ \ V₁ \) \( A₂ \ V₂ \)

— The valuation is left to yield false for the potentially remaining \( k' \in \{\text{length } \tau..\text{length } \pi + 1\} \) for all \( v \in V \) (see \( A₂ \)).
\[ \tau_{\text{length}} \pi + 1 \} \] since no more operators are executed after the trace ends anyway. The definition of \( \mathcal{A}_0 \) as the valuation that is false for every argument ensures this implicitly.

**Definition** valuation-for-operator-variables

\[ \text{variable strips-problem} \Rightarrow \text{variable strips-operator list list} \Rightarrow \text{variable strips-state list} \Rightarrow \text{sat-plan-variable valuation} \]

**Where** valuation-for-operator-variables \( \Pi \pi \tau \equiv \text{let} \)

\[ \text{ops} = \text{operators-of} \ \Pi ; \text{Op} = \{ \text{Operator } k (\text{index ops } op) \mid k \text{ op. } k \in \{0..<\text{length } \tau - 1\} \land op \in \text{set ops} \} \]

\[ \text{in override-on} \]

\[ \mathcal{A}_0 \]

\[ (\text{foldr} \ (\lambda(k, op) \ \mathcal{A}(\text{Operator } k (\text{index ops } op) := \text{True})) \]

\[ (\text{concat} \ (\text{map} (\lambda k. \ \text{map} (\text{Pair } k) (\pi ! k)) [0..<\text{length } \tau - 1])) \]

\[ \mathcal{A}_0 \]

\[ \text{Op} \]

The completeness proof requires that we show that the SATPlan encoding \( \Phi \Pi \ t \) of a problem \( \Pi \) has a model \( \mathcal{A} \) in case a solution \( \pi \) with length \( t \) exists. Since a plan corresponds to a state trace \( \tau \equiv \text{trace-parallel-plan-strips I } \pi \)

\[ \tau ! k = \text{execute-parallel-plan I (take } k \pi) \]

for all \( k < \text{length } \tau \) we can construct a valuation \( \mathcal{A}_V \) modeling the state sequence in \( \tau \) by letting

\[ \mathcal{A}(\text{State } k (\text{index vs } v) := (s \ v = \text{Some True})) \]

or all \( v \in V \) where \( s \equiv \tau ! k \). \(^{11}\)

Similarly to \( \mathcal{A}_V \), we obtain an operator valuation \( \mathcal{A}_O \) by defining

\[ \mathcal{A}(\text{Operator } k (\text{index ops } op) := \text{True}) \]

for all operators \( op \in O \) s.t. \( op \in \text{set } (\pi ! k) \) for all \( k < \text{length } \tau - 1 \).

The overall valuation for the plan execution \( \mathcal{A} \) can now be constructed by combining the state variable valuation \( \mathcal{A}_V \) and operator valuation \( \mathcal{A}_O \).

**Definition** valuation-for-plan

\[ \text{variable strips-problem} \Rightarrow \text{variable strips-operator list list} \Rightarrow \text{sat-plan-variable valuation} \]

**Where** valuation-for-plan \( \Pi \pi \tau \equiv \text{let} \)

\(^{11}\)It is helpful to remember at this point, that the trace elements of a solution contain the states reached by plan prefix execution (lemma ??).
\( vs = \text{variables-of } \Pi \)
\( \text{; ops = operators-of } \Pi \)
\( \text{; } \tau = \text{trace-parallel-plan-strips (initial-of } \Pi ) \pi \)
\( \text{; } t = \text{length } \pi \)
\( \text{; } t' = \text{length } \tau \)
\( \text{; } A_V = \text{valuation-for-state-variables } \Pi \pi \tau \)
\( \text{; } A_O = \text{valuation-for-operator-variables } \Pi \pi \tau \)
\( \text{; } V = \{ \text{State } k \text{ (index } vs v) \}
\text{; } k \in \{0..<t+1\} \land v \in \text{set } vs \}\)
\( \text{; } Op = \{ \text{Operator } k \text{ (index } ops op) \}
\text{; } k \in \{0..<t\} \land op \in \text{set } ops \}\)

in override-on (override-on \( A_0 A_V V \)) A_O Op

— Show that in case of an encoding with makespan zero, it suffices to show that a
given model satisfies the initial state and goal state encodings.

\text{lemma } \text{model-of-encode-problem-makespan-zero-iff:}
\( A \models \Phi \Pi 0 \iff A \models \Phi_I \Pi \land (\Phi_G \Pi) 0 \)
\text{proof –}
\text{have encode-operators } \Pi \Theta = \neg \bot \land \neg \bot
\text{unfolding encode-operators-def encode-all-operator-effects-def}
\text{encode-all-operator-preconditions-def}
\text{by simp}

moreover have encode-all-frame-axioms } \Pi \Theta = \neg \bot
\text{unfolding encode-all-frame-axioms-def}
\text{by simp}

ultimately show ?thesis
\text{unfolding encode-problem-def SAT-Plan-Base.encode-problem-def encode-initial-state-def}
\text{encode-goal-state-def}
\text{by simp}

\text{qed}

\text{lemma } \text{empty-valuation-is-False[simp]: } A_0 v = \text{False}
\text{unfolding empty-valuation-def \ldots}

\text{lemma } \text{model-initial-state-set-valuations:}
\text{assumes is-valid-problem-strips } \Pi
\text{shows set (map } (\lambda v. \text{ case } ((\Pi)_I) v \text{ of Some } b
\Rightarrow A_0(\text{State } 0 \text{ (index } (\text{strips-problem.variables-of } \Pi) v) := b)
| - \Rightarrow A_0)
\text{) (strips-problem.variables-of } \Pi)\)
= \{ A_0(\text{State } 0 \text{ (index } (\text{strips-problem.variables-of } \Pi) v) := \text{the } ((\Pi)_I) v))
| v. v \in \text{set } ((\Pi)_I)\}
\text{proof –}
\text{let } \forall I = (\Pi)_I
\text{and } \forall vs = \text{strips-problem.variables-of } \Pi
\text{let } \forall f = \lambda v. \text{ case } ((\Pi)_I) v \text{ of Some } b

253
\[ A_0(\text{State } 0 \text{ (index } \text{?vs v}) := b) \mid - \Rightarrow A_0 \]

and \( ?g = \lambda v. A_0(\text{State } 0 \text{ (index } \text{?vs v}) := \text{the } (?I v)) \)

let \( ?A_s = \text{map } ?f \text{ ?vs} \)

have \( \text{nb}_1: \text{dom } ?I = \text{set } ((II)_v) \)

using is-valid-problem-strips-initial-of-dom assms

by fastforce

\{ \text{fix } v \text{ assume } v \in \text{dom } ?I \\
\text{hence } ?f v = ?g v \\
\text{using } \text{nb}_1 \\
\text{by fastforce} \}

hence \( ?f ' \text{ set } ((II)_v) = ?g ' \text{ set } ((II)_v) \)

using \( \text{nb}_1 \)

by force

then have \( \text{set } ?A_s = ?g ' \text{ set } ((II)_v) \)

unfolding set-map

by simp

thus \( ?\text{thesis} \)

by blast

qed

lemma valuation-of-state-variable-implies-lit-semantics-if:

assumes \( v \in \text{dom } S \)

and \( A (\text{State } k \text{ (index } \text{vs v})) = \text{the } (S v) \)

shows \( \text{lit-semantics } A (\text{literal-formula-to-literal } (\text{encode-state-variable } k \text{ (index } \text{vs v}) (S v))) \)

proof –

let \( ?L = \text{literal-formula-to-literal } (\text{encode-state-variable } k \text{ (index } \text{vs v}) (S v)) \)

consider \( (\text{True} ) S v = \text{Some True} \)

| (\text{False} ) S v = \text{Some False} \\
\text{using assms(1) } \\
\text{by fastforce} \)

thus \( ?\text{thesis} \)

unfolding encode-state-variable-def

using assms(2)

by (cases, force+)

qed

lemma foldr-fun-upd:

assumes inj-on \( f \) (set \( \text{xs} \))

and \( x \in \text{set } \text{xs} \)

shows \( \text{foldr } (\lambda x. A(f x := g x)) \text{ xs } A (f x) = g x \)

using assms
proof (induction xs)
  case (Cons a xs) then show ?case
    proof (cases xs = [])
      case True then have x = a using Cons.prems(2)
        by simp
      thus ?thesis by simp
    next
      case False thus ?thesis proof (cases a = x)
        next
case False
  { from False have x ∈ set xs using Cons.prems(2)
    by simp
    moreover have inj-on f (set xs)
      using Cons.prems(1)
      by fastforce
    ultimately have (foldr (λx A. A(f x := g x)) xs A) (f x) = g x
      using Cons.IH
      by blast
    }
      moreover { — Follows from modus tollens on the definition of inj-on.
    have f a ≠ f x
      using Cons.prems False
      by force
    moreover have foldr (λx A. A(f x := g x)) (a ≠ xs) A
      = (foldr (λx A. A(f x := g x)) xs A)(f a := g a)
      by simp
    ultimately have foldr (λx A. A(f x := g x)) (a ≠ xs) A (f x)
      = (foldr (λx A. A(f x := g x)) xs A) (f x)
      unfolding fun-upd-def
      by presburger
    } ultimately show ?thesis
      by argo
    qed simp
  qed simp
qed fastforce

lemma foldr-fun-no-upd:
  assumes inj-on f (set xs)
  and y ∉ f : set xs
  shows foldr (λx A. A(f x := g x)) xs A y = A y
using assms

proof (induction xs)
  case (Cons a xs)
  { have inj-on f (set xs) and y /\ f ` set xs
     using Cons.prems
     by (fastforce, simp)
     hence foldr (\x. A(f x := g x)) xs A y = A y
     using Cons.IH
     by blast }
  moreover { have f a \# y
     using Cons.prems(2)
     by auto
     moreover have foldr (\x. A(f x := g x)) (a \# xs) A
     = (foldr (\x. A(f x := g x)) xs A)(f a := g a)
     by simp
     ultimately have foldr (\x. A(f x := g x)) (a \# xs) A y
     = (foldr (\x. A(f x := g x)) xs A) y
     unfolding fun-upd-def
     by presburger }
  ultimately show ?case
  by argo
qed simp

— We only use the part of the characterization of A which pertains to the state variables here.

lemma encode-problem-parallel-complete-i:
  fixes I::'a strips-problem
  assumes is-valid-problem-strips II
  and (II) \subseteq_m execute-parallel-plan ((II)_I) \pi
  \forall v k. k < length (trace-parallel-plan-strips ((II)_I) \pi)
  \rightarrow (A(State k (index (strips-problem.variables-of II) v))
  \leftrightarrow (trace-parallel-plan-strips ((II)_I) \pi ! k) v = Some True)
  \wedge (\neg A(State k (index (strips-problem.variables-of II) v))
  \leftrightarrow (trace-parallel-plan-strips ((II)_I) \pi ! k) v \neq Some True))
  shows A |\= \Phi_I II

proof –
  let ?as = strips-problem.variables-of II
  and ?I = (II)_I
  and ?G = (II)_G
  and \Phi_I = \Phi_I II
  let ?\tau = trace-parallel-plan-strips ?I \pi
  { fix C
    assume C \in cnf \Phi_I
    then obtain v

  256
where \( v \in \text{set vs} \)
and \( C \text{-is} \): \( C = \{ \text{literal-formula-to-literal} (\text{encode-state-variable} 0 (\text{index} ?vs v)) \} \)
using \( \text{cnf-of-encode-initial-state-set-ii[OF assms(1)]} \)
by auto

\[
\begin{align*}
&\text{have } 0 < \text{length } ?\tau \\
&\quad \text{using } \text{trace-parallel-plan-strips-not-nil} \\
&\quad \text{by blast} \\
&\text{then have } \mathcal{A} (\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v)) \\
&\quad \iff (\text{trace-parallel-plan-strips } ((\Pi)_1) \pi ! 0) v = \text{Some True} \\
&\quad \text{and } \neg \mathcal{A} (\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v)) \\
&\quad \iff ((\text{trace-parallel-plan-strips } ((\Pi)_1) \pi ! 0) v \neq \text{Some True}) \\
&\quad \text{using assms(3)} \\
&\quad \text{by (presburger+)}
\end{align*}
\]

note \( nb = \text{this} \)

\[
\begin{align*}
&\text{let } ?L = \text{literal-formula-to-literal} (\text{encode-state-variable} 0 (\text{index} ?vs v)) (\pi I v)) \\
&\text{have } \tau \text{-0-is: } \pi ! 0 = ?I \\
&\quad \text{using } \text{trace-parallel-plan-strips-head-is-initial-state} \\
&\quad \text{by blast} \\
&\text{have } v \text{-in-dom-I: } v \in \text{dom } ?I \\
&\quad \text{using } \text{is-valid-problem-strips-initial-of-dom assms(1) v-in-set-vs} \\
&\quad \text{by fastforce} \\
&\text{then consider } (I-v-is-Some-True) ?I v = \text{Some True} \\
&\quad | (I-v-is-Some-False) ?I v = \text{Some False} \\
&\quad \text{by fastforce} \\
&\text{hence lit-semantics } \mathcal{A} ?L \\
&\quad \text{unfolding } \text{encode-state-variable-def} \\
&\quad \text{using assms(3) } \tau \text{-0-is nb} \\
&\quad \text{by (cases, forcc+)}
\end{align*}
\]

hence \( \text{clause-semantics } \mathcal{A} C \)
unfolding \( \text{clause-semantics-def } C \text{-is} \)
by blast

\[
\begin{align*}
&\text{thus } ?\text{thesis} \\
&\quad \text{using is-cnf-encode-initial-state[OF assms(1)] is-nnf-cnf cnf-semantics} \\
&\quad \text{unfolding } \text{cnf-semantics-def} \\
&\quad \text{by blast}
\end{align*}
\]

qed

— Plans may terminate early (i.e. by reaching a state satisfying the goal state before reaching the time point corresponding to the plan length). We therefore have to show the goal by splitting cases on whether the plan successfully terminated early. If not, we can just derive the goal from the assumptions pertaining to \( \mathcal{A} \). Otherwise, we have to first show that the goal was reached (albeit early) and that our valuation \( \mathcal{A} \) reflects the termination of plan execution after the time point at which the goal
was reached.

**Lemma** `encode-problem-parallel-complete-ii`:

**Fixes** `Π::'a strips-problem`

**Assumes** `is-valid-problem-strips Π` and `(Π) G ⊆ m execute-parallel-plan ((Π) I) π`

and `∀ v k. k < length (trace-parallel-plan-strips ((Π) I) π) −→ (A (State k (index (strips-problem.variables-of Π) v)) (trace-parallel-plan-strips ((Π) I) π) ! k) v = Some True)`

and `∀ v l. l ≥ length (trace-parallel-plan-strips ((Π) I) π) ∧ l < length π + 1 −→ A (State l (index (strips-problem.variables-of Π) v)) = A (State (length (trace-parallel-plan-strips ((Π) I) π) − 1) (index (strips-problem.variables-of Π) v))`

**Shows** `A |= (Φ G Π)(length π)`

**Proof**


\{ fix `v` assume `G-of-v-is-not-None: ?G v ≠ None`

have `?G ⊆ m last ?τ`

using `execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace assms(2)`

by `blast`

also have `… = ?τ ! (?t' − 1)` using `last-conv-nth[OF trace-parallel-plan-strips-not-nil]`.

finally have `?G ⊆ m ?τ ! (?t' − 1)` by `argo`

hence `(?τ ! (?t' − 1)) v = ?G v`

using `G-of-v-is-not-None`

unfolding `map-le-def`

by `force`
\} note `nb1 = this`

— Discriminate on whether the trace has full length or not and show that the model valuation of the state variables always correspond to the (defined) goal state values.

\{ fix `v` assume `G-of-v-is-not-None: ?G v ≠ None`

hence `A (State ?t (index ?vs v)) −→ ?G v = Some True`

proof (cases `?t' = ?t + 1`)

<table>
<thead>
<tr>
<th>case <code>True</code></th>
</tr>
</thead>
</table>

moreover have `?t < ?t'` using `calculation`
by fastforce
moreover have $A$ (State $?t$ (index $?vs v$)) $\iff$ ($\forall ?! \ ?t$ $v = \text{Some True}$
using assms(3) calculation(2)
by blast
ultimately show $?thesis$
using $nb_1[OF \ G-of-v-is-not-None]$
by force
next
case $False$
{ have $?t' < $?t + 1
using length-trace-parallel-plan-strips-lte-length-plan-plus-one $False$
le-neq-implies-less
by blast
moreover have $A$ (State $?t$ (index $?vs v$)) $\iff$ $A$ (State $?t' - 1$ (index $?vs v$))
using assms(4) calculation
by simp
moreover have $?t' - 1 < $?t'
using trace-parallel-plan-strips-not-nil length-greater-0-conv[of $?\tau$]
less-diff-conv2[of 1 $?t' ?t$]
by force
moreover have $A$ (State $?t' - 1$ (index $?vs v$)) $\iff$ ($\forall ?! \ (?t' - 1)$) $v = \text{Some True}$
using assms(3) calculation(3)
by blast
ultimately have $A$ (State $?t$ (index $?vs v$)) $\iff$ ($\forall ?! \ (?t' - 1)$) $v = \text{Some True}$
by blast
}
thus $?thesis$
using $nb_1[OF \ G-of-v-is-not-None]$
by presburger
qed
} note $nb_2 = this$
} fix $C$
assume $C-in-cnfo\Phi_G$: $C \in \text{cnf } \Phi_G$

moreover obtain $v$
where $v \in \text{set } ?vs$
and $G-of-v-is-not-None$: $?G v \neq None$
and $C-is$: $C = \{ \text{literal-formula-to-literal (encode-state-variable } ?t \ (\text{index } ?vs v) \}$

using $cnf-of-encode-goal-state-set-ii[OF \ assms(1)]$ calculation
by auto
consider ($G-of-v-is-Some-True$) $?G v = \text{Some True}$
| ($G-of-v-is-Some-False$) $?G v = \text{Some False}$
using $G$-of-$v$-is-None
by fastforce
then have clause-semantics $A$ $C$
  using $nb$ $C$-is
  unfolding clause-semantics-def $encode$-state-variable-def
by (cases, force+)
}
thus $\negthesis$
  using $cnf$-semantics $[OF$ is-nnf-cnf $[OF$ encode-goal-state-is-cnf $[OF$ $assms(t)]]]$ $C$
  unfolding $cnf$-semantics-def
by blast
qed

-- We are not using the full characterization of $A$ here since it's not needed.

lemma encode-problem-parallel-complete-iii-a:
fixes $\Pi$:: ′ a strips-problem
assumes $\Pi$ is-valid-problem-strips
and $(\Pi)$ $G$ $\subseteq_m$ $m$
execute-parallel-plan $((\Pi)\ I) \pi$
and $C \in cnf$ $(encode$-all-operator-preconditions $\Pi$ $(strips$-problem.$operators-of \Pi) \pi)$
and $\forall k \ op$. $k < length (trace$-parallel-plan-strips $((\Pi)\ I) \pi) - 1$
$\longrightarrow A \ (Operator \ k \ (index \ (strips$-problem.$operators-of \Pi) \ op)) = (op \in set \ (\pi \ k))$
and $\forall l \ op$. $l \geq length (trace$-parallel-plan-strips $((\Pi)\ I) \pi) - 1 \land l < length \pi$
$\longrightarrow \neg A \ (Operator \ l \ (index \ (strips$-problem.$operators-of \Pi) \ op))$
and $\forall v \ k$. $k < length (trace$-parallel-plan-strips $((\Pi)\ I) \pi)$
$\longrightarrow (A \ (State \ k \ (index \ (strips$-problem.$variables-of \Pi) \ v))$
$\longleftrightarrow (trace$-parallel-plan-strips $((\Pi)\ I) \pi \ k) \ v = Some \ True)$
shows clause-semantics $A$ $C$
proof
let $?ops$ = strips-problem.$operators-of \Pi$
and $?vs$ = strips-problem.$variables-of \Pi$
and $?t$ = length $\pi$
let $?\tau$ = trace-parallel-plan-strips $((\Pi)\ I) \pi$

obtain $k \ op$
  where $k$-and-op-are: $(k, \ op) \in ((0..<?t) \times set ((\Pi)_0))$
  and $C \in ((\bigcup \ v \in set \ (precondition-of \ op). \ \{(Operator \ k \ (index \ ?ops \ op))\}^{-1}$
  , $(State \ k \ (index \ ?vs \ v))^+ \}))$
using cnf-of-encode-all-operator-preconditions-structure $assms(3)$
$UN-E[\of \ C ]$
by auto
then obtain $v$
  where $v$-in-preconditions-of-op: $v \in set \ (precondition-of \ op)$
  and $C$-is: $C = \{ (Operator \ k \ (index \ ?ops \ op))^{-1}, (State \ k \ (index \ ?vs \ v))^+ \}$
by blast
thus $\negthesis$
proof (cases $k < length ?\tau - 1$)
case $k \lt \text{length}\tau - \text{minus\-one}$: True

thus $\neg \text{thesis}$

proof (cases $op \in \text{set}(\pi \cdot k))$

  case True

  
  have $\text{are\-all\-operators\-applicable}(\pi \cdot k)$
  using $\text{trace\-parallel\-plan\-strips\-operator\-preconditions\ k\\lt\text{length}\tau\\text{\-minus\-one}}$
  by blast

  then have $(\pi \cdot k) v = \text{Some True}$
  using $\text{are\-all\-operators\-applicable\-set} v\\text{-in\-preconditions\-of\-op True}$
  by fast

  hence $\text{A}(\text{State} k (\text{index} ?vs v))$
  using assms(6) $k \lt \text{length}\tau \\text{\-minus\-one}$
  by force

  }

  thus $\neg \text{thesis}$

  using $C\text{-is}$

  unfolding clause\-semantics\-def

  by fastforce

next

  case False

  then have $\neg \text{A}(\text{Operator} k (\text{index} ?ops op))$
  using assms(4) $k \lt \text{length}\tau \\text{\-minus\-one}$
  by blast

  thus $\neg \text{thesis}$

  using $C\text{-is}$

  unfolding clause\-semantics\-def

  by fastforce

qed

next

  case False

  then have $k \geq \text{length} ?\tau - 1 k < ?t$

  using $k\text{-and-op\-are}$

  by(force, simp)

  then have $\neg \text{A}(\text{Operator} k (\text{index} ?ops op))$
  using assms(5)
  by blast

  thus $\neg \text{thesis}$

  unfolding clause\-semantics\-def

  using $C\text{-is}$

  by fastforce

qed

qed

— We are not using the full characterization of $\text{A}$ here since it’s not needed.

lemma encode\-problem\-parallel\-complete\-iii\:-b:

  fixes $\Pi \cdot \text{\'a strips\-problem}$

  assumes $\text{is\-valid\-problem\-strips} \Pi$

261
and $(\Pi)_G \subseteq_m \text{execute-parallel-plan} ((\Pi)_I) \pi$
and $C \in \text{cnf} (\text{encode-all-operator-effects} \ \Pi \ \text{(strips-problem.operators-of \ II)})$

(length $\pi$)

and $\forall k. a. k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_I) \pi) - 1$
\[ \longrightarrow \mathcal{A} (\text{Operator } k (\text{index} (\text{strips-problem.operators-of \ II} \ a))) = (a \in \text{set} (\pi \ k)) \]

and $\forall l. a. l \geq \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_I) \pi) - 1 \land l < \text{length} \pi$
\[ \longrightarrow \neg \mathcal{A} (\text{Operator } l (\text{index} (\text{strips-problem.operators-of \ II} \ a))) \]

and $\forall v,k. a. k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_I) \pi)$
\[ \longrightarrow (\mathcal{A} (\text{State } k (\text{index} (\text{strips-problem:variables-of \ II} \ a))) \land \text{trace-parallel-plan-strips} ((\Pi)_I) \pi \ \pi = \text{Some True}) \]

shows clause-semantics $\mathcal{A} C$

proof –

let $?\text{ops} = \text{strips-problem.operators-of \ II}$

and $?\text{vs} = \text{strips-problem:variables-of \ II}$

and $?\text{t} = \text{length} \pi$

let $?\text{A} = \bigcup k. (k, \text{op}) \in \{0..<?\text{t}\} \times \text{set} ((\Pi)_I).$

$\bigcup v. \text{set} (\text{add-effects-of } \text{op}).$

{ { (\text{Operator } k (\text{index} (?\text{ops} \ \text{op})))^{-1}, (\text{State } (\text{Suc } k) (\text{index} (?\text{vs} \ v)))^+ } }$

and $?\text{B} = \bigcup k. (k, \text{op}) \in \{0..<?\text{t}\} \times \text{set} ((\Pi)_I).$

$\bigcup v. \text{set} (\text{delete-effects-of } \text{op}).$

{ { (\text{Operator } k (\text{index} (?\text{ops} \ \text{op})))^{-1}, (\text{State } (\text{Suc } k) (\text{index} (?\text{vs} \ v)))^{-1 } } }$

consider (C-in-A) $C \in $?A

| (C-in-B) $C \in ?B$


by (metis C-in-A C-in-B)

thus $?\text{thesis}$

proof (cases)

case C-in-A

then obtain $k \ \text{op}$

where $k-and-op: (k, \text{op}) \in \{0..<?\text{t}\} \times \text{set}((\Pi)_I)$

and $C \in (\bigcup v \in \text{set} (\text{add-effects-of } \text{op}).$

{ { (\text{Operator } k (\text{index} (?\text{ops} \ \text{op})))^{-1}, (\text{State } (\text{Suc } k) (\text{index} (?\text{vs} \ v)))^+ } }$

by blast

then obtain $v$ where $v-in-add-effects-of-op: v \in \text{set} (\text{add-effects-of } \text{op})$

and C-is: $C = \{ (\text{Operator } k (\text{index} (?\text{ops} \ \text{op})))^{-1}, (\text{State } (\text{Suc } k) (\text{index} (?\text{vs} \ v)))^+ \}$

by blast

thus $?\text{thesis}$

proof (cases)

case k-\lt-length-\tau-minus-one: True

thus $?\text{thesis}$

proof (cases $\text{op} \in \text{set} (\pi \ k))$

  case True

  { then have are-all-operators-applicable ($?\text{t} \ k$) (π ! k)

    and are-all-operator-effects-consistent (π ! k)

    using trace-parallel-plan-strips-operator-preconditions k-\lt-length-\tau-minus-one

  }

  case $\text{op} \in \text{set} (\pi \ k))$

  { then have are-all-operators-applicable ($?\text{t} \ k$) (π ! k)

    and are-all-operator-effects-consistent (π ! k)

    using trace-parallel-plan-strips-operator-preconditions k-\lt-length-\tau-minus-one

  }

  case $\text{op} \in \text{set} (\pi \ k))$

  { then have are-all-operators-applicable ($?\text{t} \ k$) (π ! k)

    and are-all-operator-effects-consistent (π ! k)

    using trace-parallel-plan-strips-operator-preconditions k-\lt-length-\tau-minus-one

  }

262
by blast+

hence execute-parallel-operator (?τ ! k) (π ! k) v = Some True
using execute-parallel-operator-positive-effect-if[
    OF - - True v-in-add-effects-of-op, of ?τ ! k]
by blast

then have τ-Suc-k-is-Some-True: (?τ ! Suc k) v = Some True
using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
by argo

have A (State (Suc k) (index ?vs v))
using assms(6) k-lt-length-τ-minus-one τ-Suc-k-is-Some-True
by fastforce
thus ?thesis
using C-is
unfolding clause-semantics-def
by fastforce

next

case False
then have ¬A (Operator k (index ?ops op))
using assms(4) k-lt-length-τ-minus-one
by blast
thus ?thesis
using C-is
unfolding clause-semantics-def
by force

qed

next

— This case is completely symmetrical to the one above.

next

— C-in-B
then obtain k op
where k-and-op-are: (k, op) ∈ {θ..<?t} × set ((Π)O)
and C ∈ (∪ v ∈ set (delete-effects-of op).
{ ((Operator k (index ?ops op))−1, (State (Suc k) (index ?vs v))−1 })
by blast
then obtain v where v-in-delete-effects-of-op: v ∈ set (delete-effects-of op)
and C-is: C = { (Operator k (index ?ops op))−1, (State (Suc k) (index ?vs
v)\^{-1} \}
\text{by blast}
\text{thus } \text{?thesis}
\text{proof (cases } k < \text{ length } \tau - 1 \text{)}
\text{case } k\text{-lt-length-}\tau\text{-minus-one: True}
\text{thus } \text{?thesis}
\text{proof (cases } \text{op } \in \text{ set } (\pi ! k))
\text{case True}
{ \text{then have are-all-operators-applicable } (\forall \tau ! k) (\pi ! k)
\text{and are-all-operator-effects-consistent } (\pi ! k)
\text{using trace-parallel-plan-strips-operator-preconditions } k\text{-lt-length-}\tau\text{-minus-one}
\text{by blast+}
\text{hence execute-parallel-operator } (\forall \tau ! k) (\pi ! k) v = \text{Some False}
\text{using execute-parallel-operator-negative-effect-if [}
\text{OF - - True v-in-delete-effects-of-op, of } (\forall \tau ! k)]
\text{by blast}
\}
\text{then have } \tau\text{-Suc-k-is-Some-True: } (\forall \tau ! \text{ Suc } k) v = \text{Some False}
\text{using trace-parallel-plan-step-effect-is[OF } k\text{-lt-length-}\tau\text{-minus-one]}
\text{by argo}
\text{have } \neg A (\text{State } (\text{Suc } k) (\text{index } ?v s v))
\text{using assms(6) } k\text{-lt-length-}\tau\text{-minus-one } \tau\text{-Suc-k-is-Some-True}
\text{by fastforce}
\text{thus } \text{?thesis}
\text{using C-is}
\text{unfolding clause-semantics-def}
\text{by fastforce}
\text{next}
\text{case False}
\text{then have } \neg A (\text{Operator } k (\text{index } ?ops op))
\text{using assms(4) } k\text{-lt-length-}\tau\text{-minus-one}
\text{by blast}
\text{thus } \text{?thesis}
\text{using C-is}
\text{unfolding clause-semantics-def}
\text{by force}
\text{qed}
\text{next}
\text{case False}
\text{then have } k \geq \text{ length } \tau - 1 \text{ and } k < ?t
\text{using k-and-op-are}
\text{by auto}
\text{then have } \neg A (\text{Operator } k (\text{index } ?ops op))
\text{using assms(5)}
\text{by blast}
\text{thus } \text{?thesis}
\text{using C-is}
\text{unfolding clause-semantics-def}
by fastforce
qed
qed

lemma encode-problem-parallel-complete-iii:
fixes Π::′a strips-problem
assumes is-valid-problem-strips Π and (Π) G ⊆ m execute-parallel-plan ((Π) I) π and ∀ k op. k < length (trace-parallel-plan-strips ((Π) I) π) − 1 −→ A (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set (π ! k))
and ∀ l op. l ≥ length (trace-parallel-plan-strips ((Π) I) π) − 1 ∧ l < length π −→ ¬A (Operator l (index (strips-problem.operators-of Π) op))
and ∀ v k. k < length (trace-parallel-plan-strips ((Π) I) π) −→ (A (State k (index (strips-problem.variables-of Π) v))) (trace-parallel-plan-strips ((Π) I) π ! k) v = Some True
shows A |= encode-operators Π (length π)
proof –
let ?t = length π
and ?ops = strips-problem.operators-of Π
let ?Φ O = encode-operators Π ?t
and ?Φ P = encode-all-operator-preconditions Π ?ops ?t
and ?Φ E = encode-all-operator-effects Π ?ops ?t
{ fix C
assume C ∈ cnf ?Φ O
then consider (C-in-precondition-encoding) C ∈ cnf ?Φ P
| (C-in-effect-encoding) C ∈ cnf ?Φ E
using cnf-of-operator-encoding-structure
by blast
hence clause-semantics A C
proof (cases)
case C-in-precondition-encoding
thus ?thesis
using encode-problem-parallel-complete-iii-a[OF assms(1, 2) - assms(3, 4, 5)]
by blast
next
case C-in-effect-encoding
thus ?thesis
using encode-problem-parallel-complete-iii-b[OF assms(1, 2) - assms(3, 4, 5)]
by blast
qed
}
thus ?thesis
using encode-operators-is-cnf[OF assms(1)] is-nnf-cnf cnf-semantics
unfolding \texttt{cnf-semantics-def} \\
by \texttt{blast} \\
\texttt{qed}

\textbf{lemma} \texttt{encode-problem-parallel-complete-iv-a:} \\
\textbf{fixes} \Pi :: a strips-problem \\
\textbf{assumes} \texttt{STRIPS-Semantics.is-parallel-solution-for-problem II \pi} \\
\textbf{and} \forall k \text{ op}. k < \text{length} (\text{trace-parallel-plan-strips ((II)_{I}) \pi}) - 1 \\
\rightarrow \text{A} (\text{Operator k (index (strips-problem.operators-of II) \text{op}))} = (\text{op} \in \text{set (\pi ! k)}) \\
\textbf{and} \forall v k. k < \text{length} (\text{trace-parallel-plan-strips ((II)_{I}) \pi}) \\
\rightarrow (\text{A (State k (index (strips-problem.variables-of II) v))}) \\
\leftrightarrow (\text{trace-parallel-plan-strips ((II)_{I}) \pi ! k} \ v = \text{Some True}) \\
\textbf{and} \forall v l. l \geq \text{length} (\text{trace-parallel-plan-strips ((II)_{I}) \pi}) \land l < \text{length} \pi + 1 \\
\rightarrow \text{A (State l (index (strips-problem.variables-of II) v))} \\
= \text{A (State (length (trace-parallel-plan-strips ((II)_{I}) \pi) - 1)} \\
\text{index (strips-problem.variables-of II) v))} \\
\textbf{and} \ C \in \bigcup (\bigcup (k, v) \in \{0..<\text{length} \pi\} \times \text{set ((II)_{Y})}. \\
\{\{ (\text{State k (index (strips-problem.variables-of II) v)})^{+} \\
\land (\text{State (Suc k) (index (strips-problem.variables-of II) v)})^{-1} \} \\
\cup \{ (\text{Operator k (index (strips-problem.operators-of II) \text{op}))}^{+}\ |	ext{op. op} \in \text{set ((II)_{O}) \land v} \in \text{set (add-effects-of op) } \})\}) \\
\textbf{shows} \text{clause-semantics A C} \\
\textbf{proof} – \\
\textbf{let} ?vs = \text{strips-problem.variables-of II} \\
\textbf{and} ?ops = \text{strips-problem.operators-of II} \\
\textbf{and} ?t = \text{length} \pi \\
\textbf{let} ?\tau = \text{trace-parallel-plan-strips ((II)_{I}) \pi} \\
\textbf{let} ?A = \bigcup (k, v) \in \{0..<?t\} \times \text{set ?es}. \\
\{\{ (\text{State k (index ?vs v)})^{+}, (\text{State (Suc k) (index ?vs v)})^{-1} \} \\
\cup \{ (\text{Operator k (index ?ops op)})^{+}\ |	ext{op. op} \in \text{set ?ops \land v} \in \text{set (add-effects-of op) } \})\}) \\
\{ \\
\text{obtain C' where} \ C' \in {?A \text{ and C-in-C'}: C \in C'} \\
\text{using Union-iff assms(5)} \\
\text{by auto} \\
\text{then obtain} k v \\
\text{where} (k, v) \in \{0..<?t\} \times \text{set ?es} \\
\text{and} C' \in \{\{ (\text{State k (index ?vs v)})^{+}, (\text{State (Suc k) (index ?vs v)})^{-1} \} \\
\cup \{ (\text{Operator k (index ?ops op)})^{+}\ |	ext{op. op} \in \text{set ?ops \land v} \in \text{set (add-effects-of op) } \})\}} \\
\text{using UN-E} \\
\text{by blast} \\
\text{hence} \exists k v. \\
k \in \{0..<?t\}
∧ v ∈ set ?vs
∧ C = { (State k (index ?vs v))\^+, (State (Suc k) (index ?vs v))^{-1} } 
∪ { (Operator k (index ?ops op))\^+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }

using C-in-C' by blast
then obtain k v 
where k-in: k ∈ {0..<?t}
and v-in-vs: v ∈ set ?vs
and C-is: C = { (State k (index ?vs v))\^+, (State (Suc k) (index ?vs v))^{-1} }
∪ { (Operator k (index ?ops op))\^+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }
by blast
show ?thesis

proof (cases k < length ?τ - 1)
case k-lt-length-τ-minus-one: True
then have k-lt-t: k < ?t
using k-in by force
have all-operators-applicable: are-all-operators-applicable (?τ ! k) (?τ ! k)
and all-operator-effects-consistent: are-all-operator-effects-consistent (?τ ! k)
using trace-parallel-plan-strips-operator-preconditions[OF k-lt-length-τ-minus-one]
by simp+
then consider (A) ∃ op ∈ set (?τ ! k). v ∈ set (add-effects-of op) 
| (B) ∃ op ∈ set (?τ ! k). v ∈ set (delete-effects-of op) 
| (C) ∀ op ∈ set (?τ ! k). v /∈ set (add-effects-of op) ∧ v /∈ set (delete-effects-of op)
by blast
thus ?thesis

proof (cases)
case A
moreover obtain op
where op-in-π_k: op ∈ set (?τ ! k)
and v-is-add-effect: v ∈ set (add-effects-of op)
using A
by blast
moreover {
have (?τ ! k) ∈ set π
using k-lt-t
by simp
hence op ∈ set ?ops
using is-parallel-solution-for-problem-operator-set[OF assms(1) - op-in-π_k]
by blast
}
ultimately have (Operator k (index ?ops op))\^+ 
∈ { (Operator k (index ?ops op))\^+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }

267
using \textit{v-is-add-effect} \\
by blast \\
then have \((\text{Operator } k (\text{index } \textit{ops} \textit{op}))^+ \in C\) \\
using \textit{C-is} \\
by auto \\
moreover have \(\textit{A} (\text{Operator } k (\text{index } \textit{ops} \textit{op}))\) \\
using \textit{assms}(2) \(k\text{-lt-length-}\tau\text{-minus-one} \textit{op-in-}\pi_k\) \\
by blast \\
ultimately show \(?\textit{thesis}\) \\
unfolding \textit{clause-semantics-def} \\
by force \\
next \\
case \(B\) \\
then obtain \textit{op} \\
where \(\textit{op-in-}\pi_k\): \(\textit{op} \in \text{set } (\pi ! k)\) \\
and \(\textit{v-is-delete-effect}: \textit{v} \in \text{set } (\textit{delete-effects-of } \textit{op})..\) \\
then have \(\neg(\exists \textit{op} \in \text{set } (\pi ! k). \textit{v} \in \text{set } (\textit{add-effects-of } \textit{op}))\) \\
using \textit{all-operator-effects-consistent are-all-operator-effects-consistent-set} \\
by fast \\
then have \(\text{execute-parallel-operator } (\?\tau ! k) (\pi ! k) \textit{v} = \text{Some False}\) \\
using \textit{execute-parallel-operator-negative-effect-if}[\text{OF all-operators-applicable} \\
\textit{all-operator-effects-consistent op-in-}\pi_k \textit{v-is-delete-effect}] \\
by blast \\
moreover have \((?\tau ! \text{Suc } k) \textit{v} = \text{execute-parallel-operator } (\?\tau ! k) (\pi ! k) \textit{v}\) \\
using \textit{trace-parallel-plan-step-effect-is}[\text{OF } k\text{-lt-length-}\tau\text{-minus-one}] \\
by simp \\
ultimately have \(\neg \textit{A} (\text{State } (\text{Suc } k) (\text{index } \textit{vs} \textit{v}))\) \\
using \textit{assms}(3) \(k\text{-lt-length-}\tau\text{-minus-one}\) \\
by simp \\
thus \(?\textit{thesis}\) \\
using \textit{C-is} \\
unfolding \textit{clause-semantics-def} \\
by simp \\
next \\
case \(C\) \\
show \(?\textit{thesis}\) \\
proof (cases \((?\tau ! k) \textit{v} = \text{Some True}) \\
case \text{True} \\
then have \(\textit{A} (\text{State } k (\text{index } \textit{vs} \textit{v}))\) \\
using \textit{assms}(3) \(k\text{-lt-length-}\tau\text{-minus-one}\) \\
by force \\
thus \(?\textit{thesis}\) \\
using \textit{C-is} \\
unfolding \textit{clause-semantics-def} \\
by fastforce \\
next \\
case \text{False}
have $(\tau!\text{Succ}) = \text{execute-parallel-operator}(\tau!k)(\pi!k)$
using \text{trace-parallel-plan-step-effect-is}[\text{OF}k-\text{ll-length}-\tau\text{-minus-one}].
then have $(\tau!\text{Succ})v = (\tau!k)v$
using \text{execute-parallel-operator-no-effect-if}C
by \text{fastforce}
hence $(\tau!\text{Succ})v \neq \text{Some True}$
using $\text{False}$
by \text{argo}

then have $\neg\mathcal{A}(\text{State}(\text{Suc}k)(\text{index}?vv))$
using \text{assms}(3) k-\text{ll-length}-\tau\text{-minus-one}
by \text{auto}
thus $?\text{thesis}$
using $\text{C-is}$
unfolding \text{clause-semantics-def}
by \text{fastforce}
qed

next
\text{case} k-\text{gte-length}-\tau\text{-minus-one}: \text{False}
\text{show} $?\text{thesis}$
\text{proof} (\text{cases} \mathcal{A}(\text{State}(\text{length}??\tau-1)(\text{index}?vv)))
\text{case} True
{ have $\mathcal{A}(\text{State}k(\text{index}?vv)) = \mathcal{A}(\text{State}(\text{length}??\tau-1)(\text{index}?vv))$
proof (\text{cases} k = \text{length}??\tau-1)
case $\text{False}$
then have $\text{length}??\tau \leq k$ and $k < ??t+1$
using $k-\text{gte-length}-\tau\text{-minus-one}k\text{-in}$
by \text{fastforce+}
thus $?\text{thesis}$
using \text{assms}(4)
by \text{blast}
qed \text{blast}
hence $\mathcal{A}(\text{State}k(\text{index}?vv))$
using $\text{True}$
by \text{blast}
}
thus $?\text{thesis}$
using $\text{C-is}$
unfolding \text{clause-semantics-def}
by \text{simp}
next
\text{case} False
{ have $\text{length}??\tau \leq \text{Suc}k$ and $\text{Suc}k < ??t+1$
using $k-\text{gte-length}-\tau\text{-minus-one}k\text{-in}$

269
by fastforce
then have \( A (\text{State} (\text{Suc} k) (\text{index} ?vs v)) = A (\text{State} (\text{length} ?\tau - 1)) \) (index ?vs v)
  using assms(4)
  by blast
hence \( \neg A (\text{State} (\text{Suc} k) (\text{index} ?vs v)) \)
  using False
  by blast
\}
thus \(?thesis\)
  using C-is unfolding clause-semantics-def
  by fastforce
qed
qed
qed

lemma encode-problem-parallel-complete-iv-b:
  fixes \( \Pi :: \tau a \text{ strips-problem} \)
  assumes is-parallel-solution-for-problem \( \Pi \pi \)
  and \( \forall k \text{ op}. \ k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_{\pi}) \pi) - 1 \)
  \( \rightarrow A (\text{Operator} k (\text{index} (\text{strips-problem.operators-of} \Pi) \text{ op})) = (\text{op} \in \text{set} (\pi ! k)) \)
  and \( \forall v \text{ k}. \ k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_{\pi}) \pi) \)
  \( \rightarrow (A (\text{State} k (\text{index} (\text{strips-problem.variables-of} \Pi) \text{ v}))) \)
  \( \leftarrow (\text{trace-parallel-plan-strips} ((\Pi)_{\pi}) \pi ! k) \ \text{v} = \text{Some True} \)
  and \( \forall v \text{ l}. \ l \geq \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_{\pi}) \pi) \land l < \text{length} \pi + 1 \)
  \( \rightarrow A (\text{State} l (\text{index} (\text{strips-problem.variables-of} \Pi) \text{ v}))) \)
  \( = A (\text{State} (\text{length} (\text{trace-parallel-plan-strips} ((\Pi)_{\pi}) \pi) - 1) \)
  \( (\text{index} (\text{strips-problem.variables-of} \Pi) \text{ v}))) \)
  and \( C \in \bigcup \{ (k, v) \in \{0..<\text{length} \pi\} \times \text{set} ((\Pi)_{\pi})\} \)
  \( \{\{ (\text{State} k (\text{index} (\text{strips-problem.variables-of} \Pi) \text{ v})))^{-1} \)
  \( , (\text{State} (\text{Suc} k) (\text{index} (\text{strips-problem.variables-of} \Pi) \text{ v})))^+ \} \)
  \( \bigcup \{ (\text{Operator} k (\text{index} (\text{strips-problem.operators-of} \Pi) \text{ op}))^+ \)
  \( | \text{op. op} \in \text{set} ((\Pi)_{\Pi}) \land \text{v} \in \text{set} (\text{delete-effects-of op}) \} \} \} \}
shows clause-semantics \( A \ C \)
proof –
let \(?vs = \text{strips-problem.variables-of} \Pi \)
and \(?ops = \text{strips-problem.operators-of} \Pi \)
and \(?t = \text{length} \pi \)
let \(?A = (\bigcup k, v \in \{0..<?t\} \times \text{set} \?vs\) \)
  \( \{\{ (\text{State} k (\text{index} \?vs v))^{-1} , (\text{State} (\text{Suc} k) (\text{index} \?vs v)))^+ \} \)
  \( \bigcup \{ (\text{Operator} k (\text{index} \?ops \text{ op}))^+ \)
  \( | \text{op. op} \in \text{set} ((\Pi)_{\Pi}) \land \text{v} \in \text{set} (\text{delete-effects-of op}) \} \} \}
}
obtain $C'$ where $C' \in A$ and $C\in C'$
using Union-iff assms(5)
by auto

then obtain $k v$
where $(k, v) \in \{0 < t\} \times \text{set } vs$
and $C' \in \{\{(\text{State } k (\text{index } vs v))^{-1}, (\text{State } (\text{Suc } k) (\text{index } vs v))^+ \}
\cup \{(\text{Operator } k (\text{index } ops op))^+ \mid \text{op. op } \in \text{set } \text{ops} \land v \in \text{set } \text{delete-effects-of op}\}\}$
using $\text{UN-E}$
by fastforce

hence $\exists k v. k \in \{0 < t\}$
$\land v \in \text{set } vs$
$\land C = \{\{(\text{State } k (\text{index } vs v))^{-1}, (\text{State } (\text{Suc } k) (\text{index } vs v))^+ \}
\cup \{(\text{Operator } k (\text{index } ops op))^+ \mid \text{op. op } \in \text{set } \text{ops} \land v \in \text{set } \text{delete-effects-of op} \}\}$
using $\text{C\in-C'}$
by auto

then obtain $k v$
where $k\text{-in: } k \in \{0 < t\}$
and $v\text{-in-vs: } v \in \text{set } \text{pl}\text{ene}$
and $C\text{-is: } C = \{\{(\text{State } k (\text{index } vs v))^{-1}, (\text{State } (\text{Suc } k) (\text{index } vs v))^+ \}
\cup \{(\text{Operator } k (\text{index } ops op))^+ \mid \text{op. op } \in \text{set } \text{ops} \land v \in \text{set } \text{delete-effects-of op} \}\}$
by auto

show $\text{thesis}$

proof (cases $k < \text{length } ?t - 1$)
case $k\text{-lt-length}\text{-}\tau\text{-minus-one: } \text{True}$
then have $k\text{-lt-t: } k < \text{t}$
using $k\text{-in}$
by force
have all-operators-applicable: are-all-operators-applicable $(\pi k (k) (\pi k))$
and all-operator-effects-consistent: are-all-operator-effects-consistent $(\pi k)$
using trace-parallel-plan-strips-operator-preconditions[Of $k\text{-lt-length}\text{-}\tau\text{-minus-one}$]
by simp
then consider $(A) \exists \text{op } \in \text{set } (\pi k), v \in \text{set } \text{delete-effects-of op}$
$\mid (B) \exists \text{op } \in \text{set } (\pi k), v \in \text{set } \text{add-effects-of op}$
$\mid (C) \forall \text{op } \in \text{set } (\pi k), v \notin \text{set } \text{add-effects-of op} \land v \notin \text{set } \text{delete-effects-of op}$
by blast
thus $\text{thesis}$

proof (cases)
case $A$
moreover obtain $\text{op}$
where $\text{op-in-}\pi k: \text{op } \in \text{set } (\pi k)$
and $\text{v-is-delete-effect: } v \in \text{set } \text{delete-effects-of op}$

271
using $A$
by blast
moreover {
 have $(\pi ! k) \in \text{set } \pi$
  using $k$-lt-t
  by simp
hence $op \in \text{set } \text{ops}$
  using $\text{is-parallel-solution-for-problem-operator-set}(\text{OF } \text{assms}(1) \cdot op-in-\pi_k)$
  by auto
}
ultimately have $(\text{Operator } k \text{ (index } \text{ops } op) )^+$
  $\in \{ (\text{Operator } k \text{ (index } \text{ops } op) )^+ \mid op \cdot op \in \text{set } \text{ops} \land v \in \text{set } (\text{delete-effects-of } op) \}$
using $\text{v-is-delete-effect}$
by blast
then have $(\text{Operator } k \text{ (index } \text{ops } op) )^+ \in C$
using $C$-is
by auto
moreover have $A \text{ (Operator } k \text{ (index } \text{ops } op) )$
  using $\text{assms}(2) \cdot k$-lt-length-$\tau$-minus-one op-in-\pi_k
by blast
ultimately show $?\text{thesis}$
unfolding $\text{clause-semantics-def}$
by force
next
case $B$
them obtain $op$
where $op-in-\pi_k$: $op \in \text{set } (\pi ! k)$
  and $\text{v-is-add-effect}$: $v \in \text{set } (\text{add-effects-of } op)$.,
then have $\neg(\exists op \in \text{set } (\pi ! k). v \in \text{set } (\text{delete-effects-of } op))$
using $\text{all-operator-effects-consistent}$
are-$\text{all-operator-effects-consistent-set}$
by fast
then have $\text{execute-parallel-operator } (?\tau ! k) \ (\pi ! k) \ v = \text{Some } \text{True}$
using $\text{execute-parallel-operator-positive-effect-if}[\text{OF } \text{all-operators-applicable}$
  all-operator-effects-consistent op-in-\pi_k $v$-is-add-effect]
by blast
moreover have $(?\tau ! \text{Suc } k) \ v = \text{execute-parallel-operator } (?\tau ! k) \ (\pi ! k) \ v$
using $\text{trace-parallel-plan-step-effect-is}[\text{OF } k$-lt-length-$\tau$-minus-one]
by simp
ultimately have $A \text{ (State } \text{Suc } k \text{ (index } \text{vs } v) )$
using $\text{assms}(3) \cdot k$-lt-length-$\tau$-minus-one
by simp
thus $?\text{thesis}$
using $C$-is
unfolding $\text{clause-semantics-def}$
by simp
next
case C
show ?thesis
— We split on cases for (?τ ! k) v = Some True here to avoid having to
proof (?τ ! k) v ≠ None.
proof (cases (?τ ! k) v = Some True)
  case True
  { have (?τ ! Suc k) = execute-parallel-operator (?τ ! k) (π ! k)
      using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one].
      then have (?τ ! Suc k) v = (?τ ! k) v
        using execute-parallel-operator-no-effect-if C
        by fastforce
      then have (?τ ! Suc k) v = Some True
        using True
        by argo
      hence A (State (Suc k) (index ?vs v))
        using assms(3) k-lt-length-τ-minus-one
        by fastforce
  }
  thus ?thesis
  using C-is
  unfolding clause-semantics-def
  by fastforce
next
  case False
  then have ¬A (State k (index ?vs v))
    using assms(3) k-lt-length-τ-minus-one
    by simp
  thus ?thesis
  using C-is
  unfolding clause-semantics-def
  by fastforce
qed
qed
next
case k-gte-length-τ-minus-one: False
show ?thesis
proof (cases A (State (length ?τ − 1) (index ?vs v)))
  case True
  { have length ?τ ≤ Suc k and Suc k < ?t + 1
      using k-gte-length-τ-minus-one k-in
      by fastforce+
      then have A (State (Suc k) (index ?vs v)) = A (State (length ?τ − 1)
        (index ?vs v))
        using assms(4)
        by blast
      hence A (State (Suc k) (index ?vs v))
        using True
  }
by blast
}
thus \( \neg \text{thesis} \)
using C-is
unfolding clause-semantics-def
by fastforce
next
case False
{
  have \( A \left( \text{State } k \left( \text{index } \Pi \text{ } v \right) \right) = A \left( \text{State } \left( \text{length } \Pi \right) - 1 \left( \text{index } \Pi \text{ } v \right) \right) \)
  proof
  (cases \( k = \text{length } \Pi - 1 \))
  case False
  then have \( \text{length } \Pi \leq k \) and \( k < ?t + 1 \)
  using k-gte-length-\( \Pi \)-minus-one k-in
  by fastforce+
  thus \( \neg \text{thesis} \)
  using assms(4)
  by blast
qed blast

hence \( \neg A \left( \text{State } k \left( \text{index } \Pi \text{ } v \right) \right) \)
using False
by blast
}
thus \( \neg \text{thesis} \)
using C-is
unfolding clause-semantics-def
by simp
qed
qed

lemma encode-problem-parallel-complete-iv:
fixes II::'a strips-problem
assumes is-valid-problem-strips II
and is-parallel-solution-for-problem II \( \pi \)
and \( \forall k \text{ op. } k < \text{length } \left( \text{trace-parallel-plan-strips } (II) \pi \right) - 1 \)
\( \rightarrow A \left( \text{Operator } k \left( \text{index } \left( \text{strips-problem.operators-of } II \right) \text{ op} \right) \left( \text{op } \in \text{set } (\pi ! k) \right) \right) \)
and \( \forall v \text{ k. } k < \text{length } \left( \text{trace-parallel-plan-strips } (II) \pi \right) \)
\( \rightarrow A \left( \text{State } k \left( \text{index } \left( \text{strips-problem.variables-of } II \right) \text{ v} \right) \right) \)
\( \left( \text{op } \in \text{set } \left( \text{trace-parallel-plan-strips } (II) \pi ! k \right) v = \text{Some } \text{True} \right) \)
and \( \forall v l. l \geq \text{length } \left( \text{trace-parallel-plan-strips } (II) \pi \right) \wedge l < \text{length } \pi + 1 \)
\( \rightarrow A \left( \text{State } l \left( \text{index } \left( \text{strips-problem.variables-of } II \right) \text{ v} \right) \right) \)
\( = A \left( \text{State } \left( \text{length } \left( \text{trace-parallel-plan-strips } (II) \pi \right) - 1 \right) \left( \text{index } \left( \text{strips-problem.variables-of } II \right) \text{ v} \right) \right) \)
shows \( A \models \text{encode-all-frame-axioms } II \left( \text{length } \pi \right) \)

274
proof

let \(?\Phi_F = encode-all-frame-axioms II (length \(\pi)\)
let \(?\upsilon = strips-problem.variables-of II\)
and \(?\upsigma = strips-problem.operators-of II\)
and \(?t = length \(\pi)\)
let \(?A = \bigcup (\bigcup (k, v) \in \{0..<?t\} \times \text{set ((II)_v)\}.
\{\{(\text{State } k (\text{index } ?\upsilon v)^+) \cup (\text{State } (\text{Suc } k) (\text{index } ?\upsilon v)^{-1})\}
\cup \{(\text{Operator } k (\text{index } ?\upsigma op)^+)
| op. op \in \text{set (add-effects-of op) }\})))\}
and \(?B = \bigcup (\bigcup (k, v) \in \{0..<?t\} \times \text{set ((II)_v).}
\{\{(\text{State } k (\text{index } ?\upsilon v)^{-1}) \cup (\text{State } (\text{Suc } k) (\text{index } ?\upsilon v)^{+})\}
\cup \{(\text{Operator } k (\text{index } ?\upsigma op)^{+})
| op. op \in \text{set (add-effects-of op) }\})))\}

have cnf-\(?\Phi_F\)-is-A-union-B: cnf \(?\Phi_F = ?A \cup ?B\)
using cnf-of-encode-all-frame-axioms-structure
by (simp add: cnf-of-encode-all-frame-axioms-structure)

\{ fix C
assume C \in cnf \(?\Phi_F\)
then consider (C-in-A) C \in ?A
| (C-in-B) C \in ?B
by argo
hence clause-semantics A C
proof (cases)
  case C-in-A
  then show \(?\text{thesis}\)
  using encode-problem-parallel-complete-iv-a[OF assms(2, 3, 4, 5) C-in-A]
  by blast
  next
  case C-in-B
  then show \(?\text{thesis}\)
  using encode-problem-parallel-complete-iv-b[OF assms(2, 3, 4, 5) C-in-B]
  by blast
  qed
\}
thus \(?\text{thesis}\)
using encode-frame-axioms-is-cnf is-nnf-cnf clause-semantics
unfolding cnf-semantics-def
by blast
qed

lemma valuation-for-operator-variables-is:
fixes II :: 'a strips-problem
assumes is-parallel-solution-for-problem II \(\pi)\)
and \(k < length (trace-parallel-plan-strips ((II)_I) \(\pi)) - 1\)
and \(op \in \text{set (add-effects-of op)}\)

shows valuation-for-operator-variables II π (trace-parallel-plan-strips ((Π)₁) π)
(Operator k (index (strips-problem.operators-of II) op))
= (op ∈ set (π ! k))

proof —
let ?ops = strips-problem.operators-of II
and ?τ = trace-parallel-plan-strips ((Π)₁) π
let ?o = Operator k (index ?ops op)
and ?Op = { Operator k (index ?ops op)
| k op. k ∈ {0..<length ?τ − 1} ∧ op ∈ set ((Π)₁) }
and ?f = λx. Operator (fst x) (index ?ops (snd x))
— show that our operator construction function is injective on set (concat (map
(lk. map (Pair k) (π ! k)) [0..<length ?τ − 1])).
have k-in: k ∈ {0..<length ?τ − 1}
using assms(2)
by fastforce

{k

{ fix k k' op op'
assume k-op-in: (k, op) ∈ set ?l and k'-op'-in: (k', op') ∈ set ?l
have Operator k (index ?ops op) = Operator k' (index ?ops op') ←→ (k, op)
= (k', op')
proof (rule iffI)
assume index-op-is-index-op': Operator k (index ?ops op) = Operator k'
(index ?ops op')
than have k-is-k': k = k'
by fast
moreover {
have k'-lt: k' < length ?τ − 1
using k'-op'-in
by fastforce

have op-in: op ∈ set (π ! k)
using k-op-in
by force
then have op'-in: op' ∈ set (π ! k)
using k'-op'-in k-is-k'
by auto

have length-τ-gt-1: length ?τ > 1
using assms(2)
by linarith
have length ?τ − Suc 0 ≤ length π + 1 = Suc 0
using length-trace-parallel-plan-strips-lte-length-plan-plus-one
using diff-le-mono
by blast
then have length ?τ − 1 ≤ length π

276
by fastforce
then have \( k' < \text{length } \pi \)
  using length-t-gl-1 k'-lt
by linarith
hence \( \pi ! k' \in \text{set } \pi \)
by simp

moreover have \( \text{op} \in \text{set } ?\text{ops} \) and \( \text{op}' \in \text{set } ?\text{ops} \)
using is-parallel-solution-for-problem-operator-set[OF assms(1)] op-in
op'-in k-is-k'
calculation
by auto
ultimately have \( \text{op} = \text{op}' \)
using index-op-is-index-op'
by force

ultimately show \((k, \text{op}) = (k', \text{op}')\)
by blast
qed fast

hence inj-on ?f (set ?l)
unfolding inj-on-def fst-def snd-def
by fast

note inj-on-f-set-l = this

{ have set ?l = \( \bigcup \{ \text{set } \{ \text{map } (\lambda k. \text{map } (\text{Pair } k) (\pi ! k)) [0..<\text{length } \tau - 1]\}\} \)
  using set-concat
by metis
also have \( \ldots = \bigcup \{ \text{set } \{ (\lambda k. \text{map } (\text{Pair } k) (\pi ! k)) \} \cdot \{0..<\text{length } \tau - 1\}\} \)
by force
also have \( \ldots = \bigcup \{ (\lambda k. \text{Pair } k) \cdot \text{set } (\pi ! k) \} \cdot \{0..<\text{length } \tau - 1\}\}
by force
also have \( \ldots = \bigcup \{ (\lambda k. \{ (k, \text{op}) \mid \text{op} \in \text{set } (\pi ! k) \}) \}
by blast
also have \( \ldots = \bigcup \{ \{ (k, \text{op}) \} \mid k \text{ op} \in \{0..<\text{length } \tau - 1\} \land \text{op} \in \text{set } (\pi ! k) \} \}
by blast

finally have set ?l = \( \bigcup (\lambda (k, \text{op}). \{ (k, \text{op}) \}) \cdot \{ (k, \text{op}) \} \)
  \{ \{ k, \text{op} \} \mid k \in \{0..<\text{length } \tau - 1\} \land \text{op} \in \text{set } (\pi ! k) \} \}
using setcompr-eq-image[of \lambda (k, \text{op}). \{ (k, \text{op}) \}] -]
by auto
} note set-l-is = this

{ have Operator k (index ?ops op) \in ?Op
using assms(3) k-in
by blast

hence valuation-for-operator-variables II π ?τ ?v
= foldr (λ(k, op) A. A(Operator k (index ?ops op) := True)) ?l A₀ ?v

unfolding valuation-for-operator-variables-def override-on-def Let-def
by auto

} note nb = this

show ?thesis
proof (cases op ∈ set (π ! k))

moreover have k-op-in: (k, op) ∈ set ?l

using set-l-is k-in calculation
by blast

— There is some problem with the pattern match in the lambda in fact, so we have to do some extra work to convince Isabelle of the truth of the statement.

moreover {
let ?g = λ_. True
thm foldr-fun-upd[OF inj-on-f-set-l k-op-in]

have ?v = Operator (fst (k, op)) (index ?ops (snd (k, op)))

by simp

moreover have (λ(k, op) A. A(Operator k (index ?ops op) := True))
= (λx A. A(Operator (fst x) (index ?ops (snd x)) := True))

by fastforce

moreover have foldr (λx A. A(Operator (fst x) (index ?ops (snd x)) := ?g x))

?l A₀ (Operator (fst (k, op)) (index ?ops (snd (k, op)))) = True

unfolding foldr-fun-upd[OF inj-on-f-set-l k-op-in].

ultimately have valuation-for-operator-variables II π ?τ ?v = True

using nb
by argo

}

thus ?thesis

using True
by blast

next

case False

{

have (k, op) /∈ set ?l

using False set-l-is

by fast

moreover {

fix k’ op’

assume (k’, op’) ∈ set ?l

and (k’ op’) = (k, op)

hence (k’ op’) = (k, op)

using inj-on-f-set-l assms(3)

by simp

}
ultimately have Operator k (index ?ops op) ∉ 𝔟f set 𝔟l
using image-iff
by force

note operator-not-in-f-image-set-l = this

have 𝒜₀ (Operator k (index ?ops op)) = False
by simp
moreover have (λ(k, op) 𝒜. 𝒜(Operator k (index ?ops op) := True))
= (lx 𝒜. 𝒜(Operator (fst x) (index ?ops (snd x)) := True))
by fastforce
ultimately have foldr (λ(k, op) 𝒜. 𝒜(Operator k (index ?ops op) := True))
𝟏 𝒜₀ 𝒉 = False
using foldr-fun-no-upd[OF inj-on-f-set-l operator-not-in-f-image-set-l, of
λ-. True 𝒜₀]
by presburger

thus 𝜆thesis
using nb False
by blast
qed
qed

lemma encode-problem-parallel-complete-vi-a:
fixes II :: ′a strips-problem
assumes is-parallel-solution-for-problem II π
and k < length (trace-parallel-plan-strips ((II)I) π) − 1
shows valuation-for-plan II π (Operator k (index (strips-problem.operators-of II)
op))
= (op ∈ set (π ! k))

proof –
let 𝒜v = strips-problem.variables-of II
and 𝒜ops = strips-problem.operators-of II
and 𝒜t = length π
and 𝒜τ = trace-parallel-plan-strips ((II)I) π
let 𝒜v = valuation-for-plan II π
and 𝒜₀ = valuation-for-operator-variables II π 𝒜τ
and 𝒜op = {Operator k (index 𝒜ops op) | k op. k ∈ {0..<?t} ∧ op ∈ set 𝒜ops }
and 𝒜v = {State k (index 𝒜v v) | k v. k ∈ {0..<?t + 1} ∧ v ∈ set 𝒜v }

have length 𝒜τ ≤ length π + 1
then have length 𝒜τ − 1 ≤ length π
by simp
then have k < 𝒜t

279
using assms
by fastforce
} note k-lt-length-\(\pi\) = this

show ?thesis
proof (cases \(op \in \text{set } ((\Pi)_O))\)
  case True
  { have \(?v \in ?Op\)
    using k-lt-length-\(\pi\) True
    by auto
    hence \(?A_\pi ?v = ?A_O ?v\)
    unfolding valuation-for-plan-def override-on-def Let-def
    by force
  }
  then show ?thesis
  using valuation-for-operator-variables-is[OF assms (1, 2) True]
  by blast

next

  case False
  { 
    -- We have \(\neg \text{index }?ops op < \text{length }?ops\) due to the assumption that \(\neg \text{op \in set } ?ops\). Hence \(\neg k \in \{0..<?t\} \text{ and therefore } ?v \notin ?Op\).
    have \(?Op = (\lambda (k, \text{op}). \text{Operator } k \text{ (index }?ops op) \times \{0..<?t\} \times \text{set } ?ops)\)
      by fast
    moreover have \(\neg \text{index }?ops op < \text{length }?ops\)
      using False
      by simp
    ultimately have \(?v \notin ?Op\)
      by fastforce
  }
  moreover have \(?v \notin ?V\)
  by force

  ultimately have \(?A_\pi ?v = A_O ?v\)
  unfolding valuation-for-plan-def override-on-def
  by metis
  hence \(\neg ?A_\pi ?v\)
  unfolding empty-valuation-def
  by blast
}
moreover have \((\pi ! k) \in \text{set } \pi\)
using k-lt-length-\(\pi\)
by simp
moreover have \(op \notin \text{set } (\pi ! k)\)
using is-parallel-solution-for-problem-operator-set[OF assms (1) calcula-
\[\text{by blast}\]

ultimately show \(?\text{thesis}\)

\[\text{by blast}\]

\(\text{qed}\)

\(\text{qed}\)

\(\text{lemma \ encode-problem-parallel-complete-vi-b:}\)

\(\text{fixes } \Pi \mapsto \text{’a strips-problem}\)

\(\text{assumes } \text{is-parallel-solution-for-problem } \Pi \ \pi\)

\(\text{and } l \geq \text{length } (\text{trace-parallel-plan-strips } (\Pi|) \pi) - 1\)

\(\text{and } l < \text{length } \pi\)

\(\text{shows } \neg \text{valuation-for-plan } \Pi \pi (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op}))\)

\(\text{proof --}\)

\(\text{let } ?vs = \text{strips-problem.variables-of } \Pi\)

\(\text{and } ?ops = \text{strips-problem.operators-of } \Pi\)

\(\text{and } ?t = \text{length } \pi\)

\(\text{and } ?\tau = \text{trace-parallel-plan-strips } (\Pi|) \pi\)

\(\text{let } ?A = \text{valuation-for-plan } \Pi \pi\)

\(\text{and } ?A_O = \text{valuation-for-operator-variables } \Pi \pi \ ?\tau\)

\(\text{and } ?Op = \{ \text{Operator } k (\text{index } ?ops \text{ op}) | k op. k \in \{0..<?t\} \land op \in \text{set } ?ops \}\)

\(\text{and } ?Op' = \{ \text{Operator } k (\text{index } ?ops \text{ op}) | k op. k \in \{0..<\text{length } ?\tau - 1\} \land op \in \text{set } ?ops \}\)

\(\text{and } ?V = \{ \text{State } k (\text{index } ?vs \text{ v}) | k v. k \in \{0..<?t + 1\} \land v \in \text{set } ?vs \}\)

\(\text{and } ?v = \text{Operator } l (\text{index } ?ops \text{ op})\)

\(\text{show } ?\text{thesis}\)

\(\text{proof (cases } op \in \text{set } ((\Pi)_C))\)

\(\text{case } \text{True}\)

\(\{\)

\(\text{have } ?v \in ?Op\)

\(\text{using } \text{assms(3) True}\)

\(\text{by auto}\)

\(\text{hence } ?A_{\tau} ?v = ?A_O ?v\)

\(\text{unfolding } \text{valuation-for-plan-def override-on-def Let-def}\)

\(\text{by simp}\)

\(\}\)

\(\text{moreover }\{\)

\(\text{have } l \notin \{0..<\text{length } ?\tau - 1\}\)

\(\text{using } \text{assms(2)}\)

\(\text{by simp}\)

\(\text{then have } ?v \notin ?Op'\)

\(\text{by blast}\)

\(\text{hence } ?A_O ?v = A_0 ?v\)
unfolding valuation-for-operator-variables-def override-on-def
by meson
}
ultimately have ¬?Aπ ?v
unfolding empty-valuation-def
by blast
}
then show ?thesis
by blast
next

case False
{
{
— We have ¬index ?ops op < length ?ops due to the assumption that ¬op ∈ set ?ops. Hence ¬k ∈ {0..<?t} and therefore ?v \not\in ?Op.

have ?Op = (λ(k, op). Operator k (index ?ops op)) ' {(0..<?t) × set ?ops}

by fast
moreover have ¬index ?ops op < length ?ops
using False
by simp
ultimately have ?v \not\in ?Op
by fastforce
}
moreover have ?v \not\in ?V
by force

ultimately have ?Aπ ?v = A0 ?v
unfolding valuation-for-plan-def override-on-def
by metis
hence ¬?Aπ ?v
unfolding empty-valuation-def
by blast
}
thus ?thesis
by blast
qed
qed

— As a corollary from lemmas and we obtain the result that the constructed valuation A ≡ valuation-for-plan II π valuates SATPlan operator variables as false if they are not contained in any operator set π ! k for any time point k < length π.

corollary encode-problem-parallel-complete-vi-d:

fixes II :: 'variable strips-problem
assumes is-parallel-solution-for-problem II π
and k < length π
and op \not\in set (π ! k)
shows ¬valuation-for-plan \(\Pi\) \(\pi\) (Operator \(k\) (index (strips-problem.operators-of \(\Pi\)) \(op\)))

using encode-problem-parallel-complete-vi-a[OF assms(1)] assms(3)
encode-problem-parallel-complete-vi-b[OF assms(1) - assms(2)] assms(3)
by (cases \(k < \text{length (trace-parallel-plan-strips ((\(\Pi\)) \(\pi\)) − 1}\); fastforce)

lemma list-product-is-nil-iff: \(\text{List.product } xs \ ys = [] \iff xs = [] \lor ys = []\)
proof (rule iffI)
assume product-xs-ys-is-Nil: \(\text{List.product } xs \ ys = []\)
show \(xs = [] \lor ys = []\)
proof (rule ccontr)
assume \(¬(xs = [] \lor ys = [])\)
then have \(xs \neq [] \land ys \neq []\)
by simp+
then obtain \(x xs' \ y ys'\) where \(xs = x \# xs' \land ys = y \# ys'\)
using list.exhaust
by metis
then have \(\text{List.product } xs \ ys = (x, y) \# \text{map (Pair } x\) ys' @ \text{List.product } xs'\)
\((y \# ys')\)
by simp
thus \(\text{False}\)
using product-xs-ys-is-Nil
by simp
qed

next
assume \(xs = [] \lor ys = []\)
thus \(\text{List.product } xs \ ys = []\)
— First cases in the next two proof blocks follow from definition of List.product.
proof (rule disjE)
assume ys-is-Nil: \(ys = []\)
show \(\text{List.product } xs \ ys = []\)
proof (induction \(xs\))
case (Cons \(x xxs\))
have \(\text{List.product } (x \# xxs) \ ys = \text{map (Pair } x\) ys @ \text{List.product } xxs \ ys\)
by simp
also have \(\ldots = [] @ \text{List.product } xxs \ ys\)
using Nil-is-map-conv ys-is-Nil
by blast
finally show ?case
using Cons.IH
by force
qed auto
qed simp
qed

— We keep the state abstract by requiring a function \(s\) which takes the index \(k\)
and returns state. This makes the lemma cover both cases, i.e. dynamic (e.g. the
\(k\)-th trace state) as well as static state (e.g. final trace state).
lemma valuation-for-state-variables-is:
assumes \( k \in \text{set } ks \) and \( v \in \text{set vs} \)
shows \( \text{foldr} (\lambda(k, v) . A. \text{valuation-for-state vs } (s k) k v A) (\text{List.product } ks vs) A_0 \)
\( \leftarrow\rightarrow (s k) v = \text{Some True} \)

proof –
let \(?v = \text{State } k \text{ (index vs } v\text{)}\) and \(?ps = \text{List.product } ks vs\)
let \(?A = \text{foldr} (\lambda(k, v) . A. \text{valuation-for-state vs } (s k) k v A) ?ps A_0\)
and \(?f = \lambda x . \text{State } (\text{fst } x) (\text{index vs } (\text{snd } x))\)
and \(?g = \lambda x . (s (\text{fst } x)) (\text{snd } x) = \text{Some True}\)
have \(nb_1: (k, v) \in \text{set } ?ps\)
using \(\text{assms(1, 2)} \text{ set-product}\)
by \(\text{simp}\)

moreover \{
\{ fix x y
assume \(x\text{-in-ps}: x \in \text{set } ?ps \text{ and } y\text{-in-ps}: y \in \text{set } ?ps\)
and \(\neg(\text{if } x = \text{if } y \rightarrow x = y)\)
then have \(f-x-is-f-y: \text{if } x = \text{if } y \text{ and } x\text{-is-not-y: } x \neq y\)
by \(\text{blast+}\)
then obtain \(k' k'' v' v''\)
where \(x\text{-is: } x = (k', v')\)
and \(y\text{-is: } y = (k'', v'')\)
by \(\text{fastforce}\)
then consider \((A) k' \neq k''\)
| \((B) v' \neq v''\)
using \(x\text{-is-not-y}\)
by \(\text{blast}\)
hence \(\text{False}\)
proof (cases)
case \(A\)
then have \(\text{if } x \neq \text{if } y\)
using \(x\text{-is y-is}\)
by \(\text{simp}\)
thus \(\text{thesis}\)
using \(f-x-is-f-y\)
by \(\text{argo}\)
next
case \(B\)
have \(v' \in \text{set vs} \text{ and } v'' \in \text{set vs}\)
using \(x\text{-in-ps } x\text{-is } y\text{-in-ps } y\text{-is set-product}\)
by \(\text{blast+}\)
then have \(\text{index vs } v' \neq \text{index vs } v''\)
using \(B\)
by \(\text{force}\)
then have \( ?f x \neq ?f y \)
  using \( \text{x-is y-is} \)
  by simp
thus \( \text{False} \)
  using \( \text{f-x-is-f-y} \)
  by blast
qed

} hence \( \text{inj-on} \ ?f \)
  \( (\text{set} \ ?ps) \)
using \( \text{inj-on-def} \)
by blast

\{ note \( \text{nb}_2 = \text{this} \)

\{ have \( \text{foldr} (\lambda x. \text{valuation-for-state} vs (s (\text{fst} x)) (\text{fst} x) (\text{snd} x)) \)
(\( \text{List.product} ks vs \)) \( A_0 (\text{State} (\text{fst} (k, v)) (\text{index} vs (\text{snd} (k, v)))) = \)
(\( s (\text{fst} (k, v)) (\text{snd} (k, v)) = \text{Some True} \))
  using \( \text{foldr-fun-upd} [OF \text{nb}_2 \text{nb}_1, \text{of} ?g A_0] \)
  by blast
moreover have \( \text{((λx. \text{valuation-for-state} vs (s (\text{fst} x)) (\text{fst} x) (\text{snd} x))} \)
\( = (\lambda (k, v). \text{valuation-for-state} vs (s k) k v) \)
  by fastforce
ultimately have \( \text{??} (?? (k, v)) = ?g (k, v) \)
  by simp
\}
thus \( \text{??} \)
  by simp
qed

lemma \( \text{encode-problem-parallel-complete-vi-c} \):
fixes \( \Pi :: 'a \text{strips-problem} \)
assumes \( \text{is-valid-problem-strips} \ \Pi \)
and \( \text{is-parallel-solution-for-problem} \ \Pi \pi \)
and \( k < \text{length} \ (\text{trace-parallel-plan-strips} ((\Pi)_I) \pi) \)
shows \( \text{valuation-for-plan} \ \Pi \pi \text{State} k (\text{index} \ (\text{strips-problem.variables-of} \ \Pi) v) \)
\( \leftarrow \rightarrow (\text{trace-parallel-plan-strips} ((\Pi)_I) \pi ! k) v = \text{Some True} \)

proof –

let \( ?vs = \text{strips-problem.variables-of} \ \Pi \)
and \( ?ops = \text{strips-problem.operators-of} \ \Pi \)
and \( ?\tau = \text{trace-parallel-plan-strips} ((\Pi)_I) \pi \)
let \( ?t = \text{length} \ \pi \)
and \( ?t' = \text{length} ?\tau \)
let \( ?A_v = \text{valuation-for-plan} \ \Pi \pi \)
and \( ?A_V = \text{valuation-for-state-variables} \ \Pi \pi ?\tau \)
and \( ?A_O = \text{valuation-for-state-variables} \ \Pi \pi ?\tau \)
and \( ?A_1 = \text{foldr} \)
(\( \lambda (k, v). \text{valuation-for-state} ?vs (?\tau ! k) k v \ A) \)
(\( \text{List.product} [0..<?t'] ?vs) \ A_0 \)

285
\( \text{and } ?Op = \{ \text{Operator } k (\text{index } ?ops op) \mid k \text{ op. } k \in \{0..<?t\} \land \text{op } \in \text{set } ((\Pi)_\circ) \} \)

\( \text{and } ?Op' = \{ \text{Operator } k (\text{index } ?ops op) \mid k \text{ op. } k \in \{0..<?t' - 1\} \land \text{op } \in \text{set } ((\Pi)_\circ) \} \)

\( \text{and } ?V = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{0..<?t\} \land v \in \text{set } ((\Pi)_V) \} \)

\( \text{and } ?V_1 = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{0..<?t'\} \land v \in \text{set } ((\Pi)_V) \} \)

\( \text{and } ?V_2 = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{?t'...(?t + 1)\} \land v \in \text{set } ((\Pi)_V) \} \)

\( \text{and } ?v = \text{State } k (\text{index } ?vs v) \)

\( \text{have } v - \text{notin-Op: } ?v \notin ?Op \)

\( \text{by blast} \)

\( \text{have } k \text{-lte-length-\(\pi\)-plus-one: } k < \text{length } \pi + 1 \)

\( \text{using less-le-trans length-trace-parallel-plan-lte-length-plan-plus-one assms(3)} \)

\( \text{by blast} \)

\( \text{show } ?\text{thesis} \)

\( \text{proof } (\text{cases } v \in \text{set } ((\Pi)_V)) \)

\( \text{case True} \)

\( \{ \)

\( \text{have } ?v \in ?V ?v \notin ?Op \)

\( \text{using k-lte-length-\(\pi\)-plus-one True} \)

\( \text{by force} + \)

\( \text{hence } ?A_{\pi} ?v = ?A_V ?v \)

\( \text{unfolding valuation-for-plan-def override-on-def Let-def} \)

\( \text{by simp} \)

\} 

\( \text{moreover } \{ \)

\( \text{have } ?v \in ?V_1 ?v \notin ?V_2 \)

\( \text{using assms(3) True} \)

\( \text{by fastforce} + \)

\( \text{hence } ?A_V ?v = ?A_1 ?v \)

\( \text{unfolding valuation-for-state-variables-def override-on-def Let-def} \)

\( \text{by force} \)

\} 

\( \text{ultimately have } ?A_{\pi} ?v = ?A_1 ?v \)

\( \text{by blast} \)

\} 

\( \text{moreover have } k \in \text{set } \{0..<?t'\} \)

\( \text{using assms(3)} \)

\( \text{by simp} \)

\( \text{moreover have } v \in \text{set } (\text{strips-problem.variables-of } \Pi) \)

\( \text{using True} \)

\( \text{by simp} \)

\( \text{ultimately show } ?\text{thesis} \)

\( \text{using valuation-for-state-variables-is[of } k \{0..<?t'\}] \)

\( \text{by fastforce} \)

next
case False
{

  have ¬ index ?vs v < length ?vs
    using False index-less-size-conv
    by simp
  hence ?v ∉ ?V
    by fastforce
}
then have ¬ ?A π ?v
  using v-notin-Op
  unfolding valuation-for-plan-def override-on-def empty-valuation-def Let-def
  variables-of-def operators-of-def
  by presburger

moreover have ¬(?τ ! k) v = Some True
  using trace-parallel-plan-strips-none-if[of π k v] assms(1, 2, 3) False
  unfolding initial-of-def
  by force
ultimately show ?thesis
  by blast
qed


definition encode-problem-parallel-complete-vi-f:
  fixes π :: 'a strips-problem
  assumes is-valid-problem-strips II
  and is-parallel-solution-for-problem II π
  and l ≥ length (trace-parallel-plan-strips ((II)I) π)
  and l < length π + 1
  shows valuation-for-plan II π (State l (index (strips-problem.variables-of II) v)) = valuation-for-plan II π (State (length (trace-parallel-plan-strips ((II)I) π) - 1) (index (strips-problem.variables-of II) v))
proof

  let ?vs = strips-problem.variables-of II
  and ?ops = strips-problem.operators-of II
  and ?τ = trace-parallel-plan-strips ((II)I) π
  let ?l = length π
  and ?l′ = length ?τ

  let ?τΩ = ?τ ! (?l′ - 1)
  and ?Aπ = valuation-for-plan II π
  and ?AV = valuation-for-state-variables II π ?τ
  and ?AO = valuation-for-state-variables II π ?τ
  let ?A2 = foldr
    (λ(k, v) A. valuation-for-state (strips-problem.variables-of II) ?τΩ k v A)
    (List.product [?l′..<length π + 2] ?vs)
\[
\begin{align*}
\mathcal{A}_0 \\
\text{and } \mathcal{Op} &= \{ \text{Operator } k (\text{index } \mathcal{ops} \ op) \mid k \text{ op. } k \in \{0..<\mathcal{t}\} \land op \in \text{set } ((\Pi)^\circ) \} \\
\text{and } \mathcal{Op}' &= \{ \text{Operator } k (\text{index } \mathcal{ops} \ op) \mid k \text{ op. } k \in \{0..<\mathcal{t}' - 1\} \land op \in \text{set } ((\Pi)^\circ) \} \\
\text{and } \mathcal{V} &= \{ \text{State } k (\text{index } \mathcal{vs} \ v) \mid k \text{ v. } k \in \{0..<\mathcal{t} + 1\} \land v \in \text{set } ((\Pi)^V) \} \\
\text{and } \mathcal{V}_1 &= \{ \text{State } k (\text{index } \mathcal{vs} \ v) \mid k \text{ v. } k \in \{0..<\mathcal{t}'\} \land v \in \text{set } ((\Pi)^V) \} \\
\text{and } \mathcal{V}_2 &= \{ \text{State } k (\text{index } \mathcal{vs} \ v) \mid k \text{ v. } k \in \{\mathcal{t}'..(\mathcal{t} + 1)\} \land v \in \text{set } ((\Pi)^V) \} \\
\text{and } \mathcal{v} &= \text{State } l (\text{index } \mathcal{vs} \ v) \\
\text{have } \mathcal{v} \notin \mathcal{Op} \\
\text{by } \text{blast} \\
\text{show } \mathcal{thesis} \\
\text{proof } (\text{cases } v \in \text{set } ((\Pi)^V)) \\
\text{case } \text{True} \\
\{ \\
\quad \text{have } \mathcal{v} \in \mathcal{V} \mathcal{v} \notin \mathcal{Op} \\
\quad \text{using } \text{assms}(4) \ \text{True} \\
\quad \text{by } \text{force+} \\
\quad \text{hence } \mathcal{A}_\pi \mathcal{v} = \mathcal{A}_V \mathcal{v} \\
\quad \text{unfolding } \text{valuation-for-plan-def override-on-def Let-def} \\
\quad \text{by } \text{simp} \\
\} \\
\text{moreover } \{ \\
\quad \text{have } \mathcal{v} \notin \mathcal{V}_1 \mathcal{v} \in \mathcal{V}_2 \\
\quad \text{using } \text{assms}(3, 4) \ \text{True} \\
\quad \text{by } \text{force+} \\
\quad \text{hence } \mathcal{A}_V \mathcal{v} = \mathcal{A}_2 \mathcal{v} \\
\quad \text{unfolding } \text{valuation-for-state-variables-def override-on-def Let-def} \\
\quad \text{by } \text{auto} \\
\} \\
\text{ultimately have } \mathcal{A}_\pi \mathcal{v} = \mathcal{A}_2 \mathcal{v} \\
\text{by } \text{blast} \\
\} \text{ note } nb = \text{this} \\
\text{moreover } \{ \\
\quad \text{have } l \in \text{set } [\mathcal{t}'..<\mathcal{t} + 2] \\
\quad \text{using } \text{assms}(3, 4) \\
\quad \text{by } \text{auto} \\
\quad \text{hence } \mathcal{A}_2 \mathcal{v} \leftarrow \mathcal{\tau_\Omega} v = \text{Some True} \\
\quad \text{using } \text{valuation-for-state-variables-is}[\text{of } l [\mathcal{t}'..<\mathcal{t} + 2]] \ \text{True nb} \\
\quad \text{by } \text{fastforce} \\
\} \\
\text{ultimately have } \mathcal{A}_\pi \mathcal{v} \leftarrow \mathcal{\tau_\Omega} v = \text{Some True}
\end{align*}
\]
by fast
moreover { 
  have 0 < ?t' 
    using trace-parallel-plan-strips-not-nil 
    by blast 
  then have ?t' − 1 < ?t' 
    using diff-less 
    by presburger 
}
ultimately show ?thesis 
  using encode-problem-parallel-complete-vi-c[of - - ?t' − 1, OF assms(1, 2)] 
  by blast 
next
  case False 
  |
  |
  |
  |
  |

moreover have ¬(∀ v. v ∈ ?V) 
  using v-notin-Op 
  unfolding valuation-for-plan-def override-on-def empty-valuation-def Let-def 
  variables-of-def operators-of-def
  by presburger 
}
moreover { 
  have 0 < ?t' 
    using trace-parallel-plan-strips-not-nil 
    by blast 
  then have ?t' − 1 < ?t' 
    by simp 
}
moreover have ¬((?τ ! (?t' − 1)) v = Some True) 
  using trace-parallel-plan-strips-none-if[of - - ?t' − 1 v, OF - assms(2) calculation(2)] 
  assms(1) False 
  by simp 
ultimately show ?thesis 
  using encode-problem-parallel-complete-vi-c[of - - ?t' − 1, OF assms(1, 2)] 
  by blast 
qed 

Let now τ ≡ trace-parallel-plan-strips I π be the trace of the plan π, t ≡ length π, and t' ≡ length τ.

Any model of the SATPlan encoding A must satisfy the following properties:
1. for all $k$ and for all $op$ with $k < t' - (1::'a)$

$$A (Operator \ k \ (\text{index} \ (\text{operators-of} \ \Pi) \ op)) = op \in \text{set} \ (\pi \ k)$$

2. for all $l$ and for all $op$ with $t' - (1::'a) \leq l$ and $l < \text{length} \ \pi$ we require

$$A (Operator \ l \ (\text{index} \ (\text{operators-of} \ \Pi) \ op))$$

3. for all $v$ and for all $k$ with $k < t'$ we require

$$A (State \ k \ (\text{index} \ (\text{variables-of} \ \Pi) \ v)) \rightarrow ((\tau \ ! \ k) \ v = \text{Some True})$$

4. and finally for all $v$ and for all $l$ with $t' \leq l$ and $l < t + (1::'a)$ we require

$$A (State \ l \ (\text{index} \ (\text{variables-of} \ \Pi) \ v)) = A (State \ (t' - 1) \ (\text{index} \ (\text{variables-of} \ \Pi) \ v))$$

Condition “1.” states that the model must reflect operator activation for all operators in the parallel operator lists $\pi ! k$ of the plan $\pi$ for each time step $k < t' - (1::'a)$ s.t. there is a successor state in the trace. Moreover “3.” requires that the model is consistent with the states reached during plan execution (i.e. the elements $\tau ! k$ for $k < t'$ of the trace $\tau$). Meaning that $A (State \ k \ (\text{index} \ (\Pi_V) \ v))$ for the SAT plan variable of every state variable $v$ at time point $k$ if and only if $(\tau ! k) \ v = \text{Some True}$ for the corresponding state $\tau ! k$ at time $k$ (and $\neg A (State \ k \ (\text{index} \ (\Pi_V) \ v))$ otherwise).

The second respectively fourth condition cover early plan termination by negating operator activation and propagating the last reached state. Note that in the state propagation constraint, the index is incremented by one compared to the similar constraint for operators, since operator activations are always followed by at least one successor state. Hence the last state in the trace has index $\text{length} \ (\text{trace-parallel-plan-strips} (\Pi_I) \ (\pi)) - 1$ and the remaining states take up the indexes to $\text{length} \ \pi + 1$.

**value** stop

— To show completeness—i.e. every valid parallel plan $\pi$ corresponds to a model for the SATPlan encoding $\Phi \ \Pi \ (\text{length} \ \pi)$—, we simply split the conjunction defined by the encoding into partial encodings and show that the model satisfies each of them.

**theorem**

\[ \text{Cf.} \ [3, \text{Theorem 3.1, p. 1044}] \text{for the construction of} \ A. \]
encode-problem-parallel-complete:
assumes is-valid-problem-strips \( \Pi \)
and is-parallel-solution-for-problem \( \Pi \) \( \pi \)
shows valuation-for-plan \( \Pi \) \( \pi \) \( = \Phi (\text{length } \pi) \)

proof –
let \( ?I = \text{length } \pi \)
and \( ?I = (\Pi)_I \)
and \( ?G = (\Pi)_G \)
and \( ?\mathcal{A} = \text{valuation-for-plan } \Pi \pi \)
have \( nb: ?G \subseteq_m \text{execute-parallel-plan } ?I \pi \)
using assms(2)
unfolding is-parallel-solution-for-problem-def
by force
have \( ?\mathcal{A} \models \Phi \Pi \)
using encode-problem-parallel-complete-i[OF assms(1) nb]
encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
by presburger
moreover have \( ?\mathcal{A} \models (\Phi_G \Pi) ?t \)
using encode-problem-parallel-complete-ii[OF assms(1) nb]
encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
encode-problem-parallel-complete-vi-f[OF assms(1, 2)]
by presburger
moreover have \( ?\mathcal{A} \models \text{encode-operators } \Pi \) ?t
using encode-problem-parallel-complete-iii[OF assms(1) nb]
encode-problem-parallel-complete-vi-a[OF assms(2)]
encode-problem-parallel-complete-vi-b[OF assms(2)]
encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
by presburger
moreover have \( ?\mathcal{A} \models \text{encode-all-frame-axioms } \Pi \) ?t
using encode-problem-parallel-complete-iv[OF assms(1, 2)]
encode-problem-parallel-complete-vi-a[OF assms(2)]
encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
encode-problem-parallel-complete-vi-f[OF assms(1, 2)]
by presburger
ultimately show \( ?\text{thesis} \)
unfolding encode-problem-def SAT-Plan-Base.encode-problem-def
encode-initial-state-def encode-goal-state-def
by auto
qed
end

theory SAT-Plan-Extensions
  imports SAT-Plan-Base
begin
8 Serializable SATPlan Encodings

A SATPlan encoding with exclusion of operator interference (see definition ??) can be defined by extending the basic SATPlan encoding with clauses

\[
\neg (\text{Atom} (\text{Operator } k (\text{index ops } op_1))) \\
\lor \neg (\text{Atom} (\text{Operator } k (\text{index ops } op_2)))
\]

for all pairs of distinct interfering operators \(op_1, op_2\) for all time points \(k < t\) for a given estimated plan length \(t\). Definitions ?? and ?? implement the encoding for operator pairs resp. for all interfering operator pairs and all time points.

**definition** encode-interfering-operator-pair-exclusion

:: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula

where encode-interfering-operator-pair-exclusion \(\Pi k op_1 op_2\)

\[\equiv (\text{let ops } = \text{operators-of } \Pi \text{ in } \neg (\text{Atom} (\text{Operator } k (\text{index ops } op_1)))) \lor \neg (\text{Atom} (\text{Operator } k (\text{index ops } op_2))))\]

**definition** encode-interfering-operator-exclusion

:: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula

where encode-interfering-operator-exclusion \(\Pi t\)

\[\equiv (\text{let ops } = \text{operators-of } \Pi \text{ in } \neg (\text{Atom} (\text{Operator } k (\text{index ops } op_1)))) \lor \neg (\text{Atom} (\text{Operator } k (\text{index ops } op_2))))\]

A SATPlan encoding with interfering operator pair exclusion can now be defined by simplying adding the conjunct `encode-interfering-operator-pair-exclusion` \(\Pi t\) to the basic SATPlan encoding.

— NOTE This is the quadratic size encoding for the \(\forall\)-step semantics as defined in [3, 3.2.1, p.1045]. This encoding ensures that decoded plans are sequentializable by simply excluding the simultaneous execution of operators with potential interference at any timepoint. Note that this yields a \(\forall\)-step plan for which parallel operator execution at any time step may be sequentialised in any order (due to non-interference).

**definition** encode-problem-with-operator-interference-exclusion

:: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula

\((\Phi_\forall \cdot 52)\)

where encode-problem-with-operator-interference-exclusion \(\Pi t\)

292
\[\equiv \text{encode-initial-state} \Pi \land (\text{encode-operators} \Pi t) \land (\text{encode-all-frame-axioms} \Pi t) \land (\text{encode-interfering-operator-exclusion} \Pi t) \land (\text{encode-goal-state} \Pi t))\]

— Immediately proof the sublocale proposition for strips in order to gain access to definitions and lemmas.

**Lemma cnf-of-encode-interfering-operator-pair-exclusion-is-i [simp]:**

\[\text{cnf} (\text{encode-interfering-operator-pair-exclusion} \Pi k \ op_1 \ op_2) = \{\{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of} \Pi) \ op_1))^{-1}, (\text{Operator } k (\text{index } (\text{strips-problem.operators-of} \Pi) \ op_2))^{-1} \}\}\]

**Proof**

let \(?ops = \text{strips-problem.operators-of} \Pi\)

\[\text{have \ cnf \ (encode-interfering-operator-pair-exclusion} \Pi k \ op_1 \ op_2) \equiv \text{cnf} (\neg (\text{Atom} (\text{Operator } k (\text{index } ?ops \ op_1))) \lor \neg (\text{Atom} (\text{Operator } k (\text{index } ?ops \ op_2))))\]

unfolding encode-interfering-operator-pair-exclusion-def

by metis

also have \(\ldots \equiv \{C \cup D \mid C \cdot D. \ \ C \in \text{cnf} (\neg (\text{Atom} (\text{Operator } k (\text{index } ?ops \ op_1)))) \land D \in \text{cnf} (\neg (\text{Atom} (\text{Operator } k (\text{index } ?ops \ op_2))))\}\)

by simp

finally show \(?thesis\)

by auto

qed

**Lemma cnf-of-encode-interfering-operator-pair-exclusion-is-ii [simp]:**

\[\text{set} (\text{encode-interfering-operator-pair-exclusion} \Pi k \ op_1 \ op_2. (op_1, op_2) \leftarrow \text{filter} (\lambda (op_1, op_2). \text{index } (\text{strips-problem.operators-of} \Pi) \ op_1 \neq \text{index } (\text{strips-problem.operators-of} \Pi) \ op_2) \land \text{are-operators-interfering} \ op_1 \ op_2) \ \\
\ (\text{List.product } (\text{strips-problem.operators-of} \Pi) (\text{strips-problem.operators-of} \Pi)) \ \\
, k \leftarrow [0..<t])\]

\[= (\bigcup (op_1, op_2) \in \text{set } (\text{operators-of} \Pi) \times \text{set } (\text{operators-of} \Pi). \text{index } (\text{strips-problem.operators-of} \Pi) \ op_1 \neq \text{index } (\text{strips-problem.operators-of} \Pi) \ op_2) \land \text{are-operators-interfering} \ op_1 \ op_2) \ \\
(\lambda k. \text{encode-interfering-operator-pair-exclusion} \Pi k \ op_1 \ op_2) \cdot \{0..<t\})\]

**Proof**

let \(?ops = \text{strips-problem.operators-of} \ Pi\)

let \(?interfering = \text{filter} (\lambda (op_1, op_2). \text{index } ?ops \ op_1 \neq \text{index } ?ops \ op_2) \land \text{are-operators-interfering} \ op_1 \ op_2) \ (\text{List.product} ?ops ?ops)\]
let ?fs = [encode-interfering-operator-pair-exclusion \Pi k op_1 op_2.
(op_1, op_2) ← \?interfering, k ← [0..<t]]

have set ?fs = \bigcup\{set
  ' (\lambda(op_1, op_2). map (\lambda. encode-interfering-operator-pair-exclusion \Pi k op_1 op_2) [0..<t])
  ' (set (filter (\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2 \land are-operators-interfering
        op_1 op_2)
        (List.product ?ops ?ops))))

unfolding set-concat set-map
by blast
— TODO slow.
also have \ldots = \bigcup\{\lambda(op_1, op_2).
  set (map (\lambda. encode-interfering-operator-pair-exclusion \Pi k op_1 op_2) [0..<t])
  ' (set (filter (\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2 \land are-operators-interfering
        op_1 op_2)
        (List.product ?ops ?ops))))

unfolding image-comp[of
  set \lambda(op_1, op_2). map (\lambda. encode-interfering-operator-pair-exclusion \Pi k op_1 op_2) [0..<t]
  comp-apply
by fast
also have \ldots = \bigcup\{\lambda(op_1, op_2).
  (\lambda. encode-interfering-operator-pair-exclusion \Pi k op_1 op_2) \{0..<t\}
  ' (set (filter (\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2 \land are-operators-interfering
        op_1 op_2)
        (List.product ?ops ?ops))))

unfolding set-map[of - [0..<t]] atLeastLess-than-up[t of 0 t]
by blast
also have \ldots = \bigcup\{\lambda(op_1, op_2).
  (\lambda. encode-interfering-operator-pair-exclusion \Pi k op_1 op_2) \{0..<t\}
  ' (set (filter (\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2 \land are-operators-interfering
        op_1 op_2)
        (List.product ?ops ?ops))))

unfolding set-filter[of \lambda(op_1, op_2). are-operators-interfering op_1 op_2 List.product
?ops ?ops]
by force
— TODO slow.
finally show \?thesis
unfolding operators-of-def set-product[of ?ops ?ops]
by fastforce
qed

lemma cnf-of-encode-interfering-operator-exclusion-is-iii[simp]:

fixes II :: 'variable strips-problem
shows cnf ' set [encode-interfering-operator-pair-exclusion \Pi k op_1 op_2.
(op_1, op_2) ← filter (\lambda(op_1, op_2). index (strips-problem.operators-of II) op_1 \neq index (strips-problem.operators-of II) op_2)
II) $op_2$
   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 \]
   
   \[(\text{List.product } (\text{strips-problem.operators-of } \Pi) \text{ } (\text{strips-problem.operators-of } \Pi)) \]
   \[ k \leftarrow [0..<t]\]
   \[ = (\bigcup (op_1, op_2)) \]
   \[ \in \{ (op_1, op_2) \in \text{set } (\text{strips-problem.operators-of } \Pi) \times \text{set } (\text{strips-problem.operators-of } \Pi) \}. \]

   \[ \text{index } \text{(strips-problem.operators-of } \Pi) \text{ } op_1 \neq \text{index } \text{(strips-problem.operators-of } \Pi) \text{ } op_2 \]

   \[ \text{proof} - \]
   
   \[ \text{let } ?ops = \text{strips-problem.operators-of } \Pi \]
   
   \[ \text{let } ?\text{interfering} = \text{filter } (\lambda (op_1, op_2). \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]
   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 ) \] (\text{List.product } ?ops \text{ } ?ops)
   
   \[ \text{let } ?fs = \text{encode-interfering-operator-pair-exclusion } \Pi \text{ } op_1 \text{ } op_2. \]
   
   \[ \text{let } ?\text{fs} = \text{filter } (\lambda (op_1, op_2). \in \{ (op_1, op_2) \}. \]
   
   \[ \text{cnf } \text{'} \text{ } ?\text{fs} = \text{cnf } \text{'} (\bigcup (op_1, op_2)) \in \{ (op_1, op_2) \}. \]
   
   \[ \text{cnf } \text{'} \text{ } (op_1, op_2) \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]
   
   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \text{unfolding } \text{cnf-of-encode-interfering-operator-pair-exclusion-is-ii} \]
   \[ \text{by blast} \]

   \[ \text{also have } \ldots = (\bigcup (op_1, op_2)) \in \{ (op_1, op_2) \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \text{unfolding } \text{image-Un image-comp comp-apply} \]
   \[ \text{by blast} \]

   \[ \text{also have } \ldots = (\bigcup (op_1, op_2)) \in \{ (op_1, op_2) \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \text{by simp} \]

   \[ \text{also have } \ldots = (\bigcup (op_1, op_2)) \in \{ (op_1, op_2) \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \land \text{are-operators-interfering } op_1 \text{ } op_2 \}. \]
   
   \[ \text{(op_1, op_2) } \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \land \text{index } ?ops \text{ } op_1 \neq \text{index } ?ops \text{ } op_2 \]

   \[ \text{by blast} \]

   \[ \text{— TODO slow.} \]
finally show \[ ? \text{thesis} \]
unfolding operators-of-def setcompr-ev-image[of - \lambda k. k \in \{0..<t\}]
by force

qed

lemma cnf-of-encode-interfering-operator-exclusion-is:
cnf (encode-interfering-operator-exclusion II t) = \bigcup \{ (op_1, op_2) \in (\text{operators-of II}) \times (\text{operators-of II}).
index (\text{strips-problem.operators-of II}) op_1 \neq \text{index (\text{strips-problem.operators-of II}) op_2} \}
\land \text{are-operators-interfering op_1 op_2} \}.
\bigcup \{ (\text{Operator k (index (\text{strips-problem.operators-of II}) op_1)})^{-1}
, (\text{Operator k (index (\text{strips-problem.operators-of II}) op_2)})^{-1} \} | k. k \in \{0..<t\}\}

proof --
let ?ops = \text{strips-problem.operators-of II}
let ?interfering = filter (\lambda (op_1, op_2). \text{index ?ops op_1} \neq \text{index ?ops op_2}
\land \text{are-operators-interfering op_1 op_2}) (\text{List.product ?ops ?ops})
let ?fs = encode-interfering-operator-pair-exclusion II k op_1 op_2.
(op_1, op_2) \rightarrow ?interfering, k \rightarrow [0..<t]
have cnf (encode-interfering-operator-exclusion II t) = cnf (foldr (\land) ?fs (\bot))
unfolding encode-interfering-operator-exclusion-def
by metis
also have ... = \bigcup (\text{cnf ' set ?fs})
unfolding cnf-foldr-and[of ?fs]..
finally show \[ ? \text{thesis} \]
unfolding cnf-of-encode-interfering-operator-exclusion-is-iii[of II t]
by blast

qed

lemma cnf-of-encode-interfering-operator-exclusion-contains-clause-if:

fixes II :: \text{'variable strips-problem}
assumes k < t
and op_1 \in \text{(strips-problem.operators-of II)} and op_2 \in \text{(strips-problem.operators-of II)}
and index (\text{strips-problem.operators-of II}) op_1 \neq \text{index (\text{strips-problem.operators-of II}) op_2}
and \text{are-operators-interfering op_1 op_2}
shows \{ (\text{Operator k (index (\text{strips-problem.operators-of II}) op_1)})^{-1}
, (\text{Operator k (index (\text{strips-problem.operators-of II}) op_2)})^{-1} \}
\in cnf (\text{encode-interfering-operator-exclusion II t})

proof --
let \Phi X = encode-interfering-operator-exclusion II t
let ?ops = \{ (op_1, op_2) \in \text{(operators-of II)} \times \text{(operators-of II)}.
index ?ops op_1 \neq \text{index ?ops op_2} \land \text{are-operators-interfering op_1 op_2} \}
and \phi = \lambda (op_1, op_2). \{ (\text{Operator k (index ?ops op_1)})^{-1}, (\text{Operator k (index ?ops op_2)})^{-1} \}\}

296
\[ k \in \{0..<t\} \]

let \( ?A = \bigcup (op_1, op_2) \in ?Ops. ?f (op_1, op_2) \)

let \( ?B = \bigcup ?A \)

and \( ?C = \{ (Operator k (index ?ops op_1))^{-1}, (Operator k (index ?ops op_2))^{-1} \} \)

\[
\begin{cases}
\text{have } (op_1, op_2) \in ?Ops \\
\text{using } \text{assms}(2, 3, 4, 5) \\
\text{unfolding } \text{operators-of-def} \\
\text{by } \text{force}
\end{cases}
\]

moreover have \( \{ ?C \} \in ?f (op_1, op_2) \)

using \( \text{assms}(1) \)

by \( \text{auto} \)

moreover have \( \{ ?C \} \in ?A \)

using \( \text{UN-iff}[of ?C - ?Ops] \text{calculation}(1, 2) \)

by \( \text{blast} \)

ultimately have \( \exists X \in ?A. ?C \in X \)

by \( \text{auto} \)

thus \( \text{thesis} \)

unfolding \( \text{cnf-of-encode-interfering-operator-exclusion-is} \)

using \( \text{UN-iff}[of ?C ?A] \)

by \( \text{auto} \)

qed

lemma \( \text{is-cnfs-encode-interfering-operator-exclusion} \):

fixes \( \Pi :: \text{variable strips-problem} \)

shows \( \text{is-cnfs} (\text{encode-interfering-operator-exclusion} \Pi t) \)

proof –

let \( ?ops = \text{strips-problem.operators-of} \Pi \)

let \( ?interfering = \text{filter} (\lambda (op_1, op_2). \text{index} ?ops op_1 \neq \text{index} ?ops op_2 \) \)

\( \wedge \text{are-operators-interfering} \) \( op_1 \) \( op_2 \)

Let \( ?Fs = [\text{encode-interfering-operator-pair-exclusion} \Pi k \) \( op_1 \) \( op_2. \) \( (op_1, op_2) \) \( \rightarrow ?interfering, k \rightarrow [0..<t] \]] \)

let \( ?Fs = (\bigcup (op_1, op_2) \) \)

\( \in \{ (op_1, op_2) \in \text{set} (\text{operators-of} \Pi) \times \text{set} (\text{operators-of} \Pi). \text{are-operators-interfering} \) \( op_1 \) \( op_2 \}) \).

(\( \lambda k. \text{encode-interfering-operator-pair-exclusion} \Pi k \) \( op_1 \) \( op_2 \) \( \cdot \) \( \{0..<t\} \))

\[
\begin{cases}
\text{fix } f \\
\text{assume } f \in \text{set } ?Fs \\
\text{then have } f \in ?Fs \\
\text{unfolding } \text{cnf-of-encode-interfering-operator-exclusion-is-ii} \\
\text{by } \text{blast}
\end{cases}
\]

then obtain \( op_1 \) \( op_2 \)

where \( (op_1, op_2) \in \text{set} (\text{operators-of} \Pi) \times \text{set} (\text{operators-of} \Pi) \)

297
and are-operators-interfering \( \text{op}_1 \text{ op}_2 \)
and \( f \in (\lambda k. \text{encode-interfering-operator-pair-exclusion} \Pi k \text{ op}_1 \text{ op}_2) \cdot \{0..<t\} \)
by fast
then obtain \( k \) where \( f = \text{encode-interfering-operator-pair-exclusion} \Pi k \text{ op}_1 \text{ op}_2 \)
by blast
then have \( f = \neg (\text{Atom} (\text{Operator} k (\text{index} \ ?\text{ops} \text{ op}_1))) \lor \neg (\text{Atom} (\text{Operator} k (\text{index} \ ?\text{ops} \text{ op}_2))) \)
unfolding \text{encode-interfering-operator-pair-exclusion-def} 
by metis
hence \( \text{is-cnf} f \)
by force 
}
thus \( \text{thesis} \)
unfolding \text{encode-interfering-operator-exclusion-def} 
using \( \text{is-cnf-foldr-and-if[of \ ?fs]} \)
by meson
qed

lemma \text{is-cnf-encode-problem-with-operator-interference-exclusion}: 
assumes \( \text{is-valid-problem-strips} \Pi \)
shows \( \text{is-cnf} (\Phi_I \Pi) \)
using \( \text{is-cnf-encode-problem} \text{ is-cnf-encode-interfering-operator-exclusion} \text{ assms} \)
unfolding \text{encode-problem-with-operator-interference-exclusion-def} \text{SAT-Plan-Base. encode-problem-def} 
\text{is-cnf.simps}(t) 
by blast

lemma \text{cnf-of-encode-problem-with-operator-interference-exclusion-structure}: 
shows \( \text{cnf} (\Phi_I \Pi) \subseteq \text{cnf} (\Phi_I \Pi) \)
and \( \text{cnf} ((\Phi_G \Pi) \Pi) \subseteq \text{cnf} (\Phi_I \Pi) \)
and \( \text{cnf} (\text{encode-operators} \Pi) \Pi) \subseteq \text{cnf} (\Phi_I \Pi) \)
and \( \text{cnf} (\text{encode-all-frame-axioms} \Pi) \Pi) \subseteq \text{cnf} (\Phi_I \Pi) \)
and \( \text{cnf} (\text{encode-interfering-operator-exclusion} \Pi) \Pi) \subseteq \text{cnf} (\Phi_I \Pi) \)
unfolding \text{encode-problem-with-operator-interference-exclusion-def} \text{encode-problem-def} \text{SAT-Plan-Base. encode-problem-def} 
\text{encode-initial-state-def} 
\text{encode-goal-state-def} 
by auto+

lemma \text{encode-problem-with-operator-interference-exclusion-has-model-then-also-partial-encodings}: 
assumes \( \mathcal{A} \models \Phi_I \Pi \)
shows \( \mathcal{A} \models \text{SAT-Plan-Base. encode-initial-state} \Pi \)
and \( \mathcal{A} \models \text{SAT-Plan-Base. encode-operators} \Pi \)
and \( \mathcal{A} \models \text{SAT-Plan-Base. encode-all-frame-axioms} \Pi \)
and \( \mathcal{A} \models \text{encode-interfering-operator-exclusion} \Pi \)
and \( \mathcal{A} \models \text{SAT-Plan-Base. encode-goal-state} \Pi \)
using \text{assms} 
unfolding \text{encode-problem-with-operator-interference-exclusion-def} \text{encode-problem-def} 

298
Just as for the basic SATPlan encoding we defined local context for the SATPlan encoding with interfering operator exclusion. We omit this here since it is basically identical to the one shown in the basic SATPlan theory replacing only the definitions of and . The sublocale proof is shown below. It confirms that the new encoding again a CNF as required by locale .

8.1 Soundness

The Proof of soundness for the SATPlan encoding with interfering operator exclusion follows directly from the proof of soundness of the basic SATPlan encoding. By looking at the structure of the new encoding which simply extends the basic SATPlan encoding with a conjunct, any model for encoding with exclusion of operator interference also models the basic SATPlan encoding and the soundness of the new encoding therefore follows from theorem ??.

Moreover, since we additionally added interfering operator exclusion clauses at every timestep, the decoded parallel plan cannot contain any interfering operators in any parallel operator (making it serializable).

— NOTE We use the subseq formulation in the fourth assumption to be able to instantiate the induction hypothesis on the subseq ops given the induction premise op ∉ ops ∈ set (subseqs (Φ⁻¹ Π A t ! k)). We do not use subsets in the assumption since we would otherwise lose the distinctness property which can be inferred from ops ∈ set (subseqs (Φ⁻¹ Π A t ! k)) using lemma subseqs-distinctD.

lemma encode-problem-serializable-sound-i:
  assumes is-valid-problem-strips II
  and A |= Φᵥ II t
  and k < t
  and ops ∈ set (subseqs ((Φ⁻¹ Π A t) ! k))
  shows are-all-operators-non-interfering ops
  proof
    let ?ops = strips-problem/operators-of II
    and ?π = Φ⁻¹ Π A t
    and ?Φₓ = encode-interfering-operator-exclusion-II t
    let ?πk = (Φ⁻¹ Π A t) ! k

    |
    fix C
    assume C-in: C ∈ cnf ?Φₓ
    have cnf-semantics A (cnf ?Φₓ)
      using cnf-semantics-monotonous-in-cnf-subsets-if[OF assms(2)]
      is-cnf-encode-problem-with-operator-interference-exclusion[OF assms(1)]
      cnf-of-encode-problem-with-operator-interference-exclusion-structure(5)].
    hence clause-semantics A C
    unfolding cnf-semantics-def
using C-in
by fast
}

note nb₁ = this
{
  fix op₁ op₂
  assume op₁ ∈ set ?πₖ and op₂ ∈ set ?πₖ
  and index-op₁-is-not-index-op₂: index ?ops op₁ ≠ index ?ops op₂
  moreover have op₁-in: op₁ ∈ set ?ops and A-models-op₁:A (Operator k (index ?ops op₁))
  and op₂-in: op₂ ∈ set ?ops and A-models-op₂:A (Operator k (index ?ops op₂))
  using decode-plan-step-element-then[OF assms(3)] calculation
  unfolding decode-plan-def
  by blast+
  moreover
  let ?C = { (Operator k (index ?ops op₁))⁻¹, (Operator k (index ?ops op₂))⁻¹
  }
  assume are-operators-interfering op₁ op₂
  moreover have ?C ∈ cnf ?Φ_X
    using cnf-of-encode-interfering-operator-exclusion-contains-clause-if[OF assms(3)]
      op₁-in op₂-in index-op₁-is-not-index-op₂ calculation
    by blast
  moreover have ¬clause-semantics A ?C
    using A-models-op₁ A-models-op₂
    unfolding clause-semantics-def
    by auto
  ultimately have False
    using nb₁
    by blast
  }
  ultimately have ¬are-operators-interfering op₁ op₂
    by blast
}

note nb₃ = this

show ?thesis
using assms

proof (induction ops)
  case (Cons op₁ ops)
  have are-all-operators-non-interfering ops
    using Cons.IH[OF Cons.prems(1, 2, 3) Cons-in-subseqsD[OF Cons.prems(4)]]
    by blast
  moreover
  {  
    fix op₂
    assume op₂-in-ops: op₂ ∈ set ops
    moreover have op₁-in-πₖ: op₁ ∈ set ?πₖ and op₂-in-πₖ: op₂ ∈ set ?πₖ
      using element-of-subseqs-then-subset[OF Cons.prems(4)] calculation(1)
      by auto+
    moreover
    {  
      have distinct (op₁ ≠ ops)
    }
  }
using subseqs-distinctD[OF Cons.prems(4)]
decode-plan-step-distinct[OF Cons.prems(3)]
unfolding decode-plan-def
by blast
moreover have op1 ∈ set ?ops and op2 ∈ set ?ops
using decode-plan-step-element-then(1)[OF Cons.prems(3)] op1-in-π_k

unfolding decode-plan-def
by force+
moreover have op1 ≠ op2
using op2-in-ops calculation(1)
by fastforce
ultimately have index ?ops op1 ≠ index ?ops op2
using index-eq-index-conv
by auto

ultimately have ~ are-operators-interfering op1 op2
using nb3
by blast

ultimately show ?case
using list-all-iff
by auto
qed simp

theorem encode-problem-serializable-sound:
assumes is-valid-problem-strips II
and A |= Φ, II t
shows is-parallel-solution-for-problem II (Φ⁻¹, II A t)
and ∀ k < length (Φ⁻¹, II A t). are-all-operators-non-interfering ((Φ⁻¹, II A t) ! k)
proof −
{
have A |= SAT-Plan-Base.encode-initial-state II
and A |= SAT-Plan-Base.encode-operators II t
and A |= SAT-Plan-Base.encode-all-frame-axioms II t
and A |= SAT-Plan-Base.encode-goal-state II t
using assms(2)
unfolding encode-problem-with-operator-interference-exclusion-def
by simp+
then have A |= SAT-Plan-Base.encode-problem II t
unfolding SAT-Plan-Base.encode-problem-def
by simp
}
thus is-parallel-solution-for-problem II (Φ⁻¹, II A t)
using encode-problem-parallel-sound assms(1, 2)
unfolding decode-plan-def
by blast

301
let \( ?\pi = \Phi^{-1} \Pi A t \)

\[
\begin{align*}
\text{fix } k \\
\text{assume } k < t \\
\text{moreover have } ?\pi^! k \in \text{set } (\text{subseqs } (?\pi^! k)) \\
\text{using } \text{subseqs-refl} \\
\text{by } \text{blast}
\end{align*}
\]

ultimately have \( \text{are-all-operators-non-interfering } (?\pi^! k) \)

\[
\begin{align*}
\text{using } \text{encode-problem-serializable-sound-i[OF assms]} \\
\text{unfolding } \text{SAT-Plan-Base.decode-plan-def decode-plan-def by } \text{blast}
\end{align*}
\]

moreover have \( \text{length } ?\pi = t \)

\[
\begin{align*}
\text{unfolding } \text{SAT-Plan-Base.decode-plan-def decode-plan-def by } \text{simp}
\end{align*}
\]

ultimately show \( \forall k < \text{length } ?\pi. \text{are-all-operators-non-interfering } (?\pi^! k) \)

\[
\begin{align*}
\text{by } \text{simp}
\end{align*}
\]

qed

8.2 Completeness

Lemma \( \text{encode-problem-with-operator-interference-exclusion-complete-i}: \)

\[
\begin{align*}
\text{assumes } & \text{is-valid-problem-strips } II \\
& \text{and } \text{is-parallel-solution-for-problem } II \pi \\
& \text{and } \forall k < \text{length } \pi. \text{are-all-operators-non-interfering } (?\pi^! k) \end{align*}
\]

shows \( \text{valuation-for-plan } II \pi \models \text{encode-interfering-operator-exclusion } II (\text{length } \pi) \)

Proof

let \( ?A = \text{valuation-for-plan } II \pi \)

\[
\begin{align*}
\text{and } ?\Phi_X = \text{encode-interfering-operator-exclusion } II (\text{length } \pi) \\
\text{and } ?\text{ops} = \text{strips-problem/operators-of } II \\
\text{and } ?t = \text{length } \pi
\end{align*}
\]

let \( ?\tau = \text{trace-parallel-plan-strips } ((I)I) \pi \)

let \( ?\text{Ops} = \{ (op_1, op_2). (op_1, op_2) \in \text{set } \text{operators-of } II \times \text{set } \text{operators-of } II \}
\]

\[
\begin{align*}
& \text{\& index } ?\text{ops} op_1 \neq \text{index } ?\text{ops} op_2 \\
& \text{\& are-operators-interfering } op_1 \text{ op}_2 \\
& \text{\& } \lambda = \text{\lambda}(op_1, op_2). \{ \{ (\text{Operator } k \text{ (index } ?\text{ops } op_1))^{-1}, (\text{Operator } k \text{ (index } ?\text{ops } op_2))^{-1} \} \}
\end{align*}
\]

| \( k, k \in \{0..<\text{length } \pi \} \}

let \( ?A = \bigcup (?f \text{ ?Ops}) \)

let \( ?B = \bigcup ?A \)

have nb:\( \forall \text{ops } \in \text{set } \pi. \forall \text{op } \in \text{set } \text{ops. op } \in \text{set } \text{operators-of } II \)

\[
\begin{align*}
\text{using } \text{is-parallel-solution-for-problem/operator-set[OF assms(2)]}
\end{align*}
\]

unfolding \text{operators-of-def by } \text{blast}

\[
\begin{align*}
\end{align*}
\]

302
fix \( k \) \( op \)

given \( k < \text{length } \pi \) and \( op \in \text{set } (\pi ! k) \)
hence \( \text{lit-semantics } \mathcal{A} ((\text{Operator } k (\text{index } ?\text{ops } op))^{+}) = (k < \text{length } ?\tau - 1) \)

using \( \text{encode-problem-parallel-complete-vi-a}(\text{OF assms}(2)) \)
\( \text{encode-problem-parallel-complete-vi-b}(\text{OF assms}(2)) \) initial-of-def
by (cases \( k < \text{length } ?\tau - 1 \); simp)

} note \( nb_2 = \text{this} \)

{ fix \( k \) \( op_1 \) \( op_2 \)
given \( k < \text{length } \pi \)
\( \text{and } op_1 \in \text{set } (\pi ! k) \)
\( \text{and } \text{index } ?\text{ops } op_1 \neq \text{index } ?\text{ops } op_2 \)
\( \text{and are-operators-interfering } op_1 \) \( op_2 \)
moderately have \( \text{are-all-operators-non-interfering } (\pi ! k) \)
using \( \text{assms}(3) \) calculation(1)
by blast

moderately have \( op_1 \neq op_2 \)
using calculation(3)
by blast

ultimately have \( op_2 \notin \text{set } (\pi ! k) \)
using \( \text{are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs} \)
using \( \text{assms}(3) \) by blast

} note \( nb_3 = \text{this} \)

{ fix \( C \)
given \( C \in \text{cnf } ?\Phi_X \)
thén have \( C \in ?B \)
using \( \text{cnf-of-encode-interfering-operator-exclusion-is}[\text{of } \Pi \text{ length } \pi] \)
by argo
then obtain \( C' \) where \( C' \in ?A \) and \( C \in C' \)
using \( \text{Union-iff}[\text{of } C ?A] \)
by meson
then obtain \( \{ op_1, op_2 \} \) where \( (op_1, op_2) \in \text{set } (\text{operators-of } \Pi) \times \text{set } (\text{operators-of } \Pi) \)
\( \text{and } \text{index-op}_1\neq\text{index-op}_2: \text{index } ?\text{ops } op_1 \neq \text{index } ?\text{ops } op_2 \)
\( \text{and are-operators-interfering-op}_1\text{-op}_2: \text{are-operators-interfering } op_1 \) \( op_2 \)
\( \text{and } C'\text{-in: } C' \in \{ \{ (\text{Operator } k (\text{index } ?\text{ops } op_1))^{-1}, (\text{Operator } k (\text{index } ?\text{ops } op_2))^{-1} \} \mid k, k \in \{0..<\text{length } \pi \} \} \)
using \( \text{UN-iff}[\text{of } C' ?f ?\text{Ops}] \)
by blast
then obtain \( k \) where \( k \in \{0..<\text{length } \pi \} \)
\( \text{and } C\text{-in: } C \equiv \{ (\text{Operator } k (\text{index } ?\text{ops } op_1))^{-1}, (\text{Operator } k (\text{index } ?\text{ops } op_2))^{-1} \} \)
using \( C\text{-in } C'\text{-in} \)
by blast
then have \( k\text{-lt-length-}\pi: k < \text{length } \pi \)

303
by simp
cconsider $(A) \ op_1 \in set (\pi \ k)$
$| (B) \ op_2 \in set (\pi \ k)$
$| (C) \neg op_1 \in set (\pi \ k) \lor \neg op_2 \in set (\pi \ k)$
by linarith
hence clause-semantics $\forall A \ C$
proof (cases)
case $A$
moreover have $op_2 \notin set (\pi \ k)$
using nb3 $k$-lt-length-$\pi$ calculation index-$op_1$-is-not-index-$op_2$ are-operators-interfering-$op_1$-$op_2$
by blast
moreover have $\neg A (Operator k (index ?ops op_2))$
using encode-problem-parallel-complete-vi-d [$OF \ assms (2)$ $k$-lt-length-$\pi$]
calculation (2)
by blast
ultimately show $?thesis$
using $C$-is
unfolding clause-semantics-def
by force
next
case $B$
moreover have $op_1 \notin set (\pi \ k)$
using nb3 $k$-lt-length-$\pi$ calculation index-$op_1$-is-not-index-$op_2$ are-operators-interfering-$op_1$-$op_2$
by blast
moreover have $\neg A (Operator k (index ?ops op_1))$
using encode-problem-parallel-complete-vi-d [$OF \ assms (2)$ $k$-lt-length-$\pi$]
calculation (2)
by blast
ultimately show $?thesis$
using $C$-is
unfolding clause-semantics-def
by force
next
case $C$
then show $?thesis$
proof (rule disjE)
assume $op_1 \notin set (\pi \ k)$
then have $\neg A (Operator k (index ?ops op_1))$
using encode-problem-parallel-complete-vi-d [$OF \ assms (2)$ $k$-lt-length-$\pi$]
by blast
thus clause-semantics (valuation-for-plan $\Pi \ \pi$) $C$
using $C$-is
unfolding clause-semantics-def
by force
next
assume $op_2 \notin set (\pi \ k)$
then have $\neg A (Operator k (index ?ops op_2))$
using encode-problem-parallel-complete-vi-d [$OF \ assms (2)$ $k$-lt-length-$\pi$]
by blast

304
thus clause-semantics \((valuation-for-plan \Pi \pi) C\)
using \(C\)-is unfolding clause-semantics-def by force
\text{qed}

\text{qed}

\{
then have cnf-semantics \(A (cnf \Phi_X)\)
unfolding cnf-semantics-def.. thus \(?thesis\)
using cnf-semantics[OF is-nnf-cnf[OF is-cnf-encode-interfering-operator-exclusion]]
by fast
\text{qed}

Similar to the soundness proof, we may reuse the previously established facts about the valuation for the completeness proof of the basic SATPlan encoding (?). To make it clearer why this is true we have a look at the form of the clauses for interfering operator pairs \(op_1\) and \(op_2\) at the same time index \(k\) which have the form shown below:

\[
\{ (\text{Operator } k (\text{index } ops \ op_1))^{-1}, (\text{Operator } k (\text{index } ops \ op_2))^{-1} \}
\]

where \(ops \equiv \Pi_\Omega\). Now, consider an operator \(op_1\) that is contained in the \(k\)-th plan step \(\pi ! k\) (symmetrically for \(op_2\)). Since \(\pi\) is a serializable solution, there can be no interference between \(op_1\) and \(op_2\) at time \(k\). Hence \(op_2\) cannot be in \(\pi ! k\) This entails that for \(A \equiv valuation-for-plan \Pi \pi\) it holds that

\[A \models \neg \text{Atom (Operator } k (\text{index } ops \ op_2))\]

and \(A\) therefore models the clause.
Furthermore, if neither is present, than \(A\) will evaluate both atoms to false and the clause therefore evaluates to true as well.
It follows from this that each clause in the extension of the SATPlan encoding evaluates to true for \(A\). The other parts of the encoding evaluate to true as per the completeness of the basic SATPlan encoding (theorem ??).

\text{theorem encode-problem-serializable-complete:}
\text{assumes is-valid-problem-strips } \Pi
\text{and is-parallel-solution-for-problem } \Pi \pi
\text{and } \forall k < \text{length } \pi. \text{ are-all-operators-non-interfering } (\pi ! k)
\text{shows valuation-for-plan } \Pi \pi \models \Phi_{\pi} (\text{length } \pi)
\text{proof –}
let \(?A = valuation-for-plan \Pi \pi\)
and \(?\Phi_X = encode-interfering-operator-exclusion \Pi (\text{length } \pi)\)
have \(?A \models SAT-Plan-Base.\text{encode-problem } \Pi (\text{length } \pi)\)
using assms(1, 2) encode-problem-parallel-complete
by auto

305
moreover have $\mathcal{A} \models \Phi_X$
using $\text{encode-problem-with-operator-interference-exclusion-complete-i}[\text{OF } \text{assms}]$.

ultimately show $\text{thesis}$

unfolding $\text{encode-problem-with-operator-interference-exclusion-def}$
$\text{encode-problem-def}$
$\text{SAT-Plan-Base}$. $\text{encode-problem-def}$
by force

qed

value $\text{stop}$

lemma $\text{encode-problem-forall-step-decoded-plan-is-serializable-i}$:
assumes $\text{is-valid-problem-strips } \Pi$
and $\mathcal{A} \models \Phi \Pi t$
shows $(\Pi)G \subseteq_m \text{execute-serial-plan } ((\Pi)_I) (\text{concat } (\Phi^{-1} \Pi \mathcal{A} t))$
proof −
let $?G = (\Pi)_G$
and $?I = (\Pi)_I$
and $?\pi = \Phi^{-1} \Pi \mathcal{A} t$
let $?\pi' = \text{concat } (\Phi^{-1} \Pi \mathcal{A} t)$
and $?\tau = \text{trace-parallel-plan-strips } ?I ?\pi$
and $?\sigma = \text{map } (\text{decode-state-at } \Pi \mathcal{A}) [0..<\text{Suc } (\text{length } ?\pi)]$

{ fix $k$
assume $k$-lt-length-$\pi$: $k < \text{length } ?\pi$
moreover have $\mathcal{A} \models \text{SAT-Plan-Base}. \text{encode-problem } \Pi t$
using $\text{assms}(2)$
unfolding $\text{encode-problem-with-operator-interference-exclusion-def}$
$\text{encode-problem-def}$
$\text{SAT-Plan-Base}. \text{encode-problem-def}$
by simp
moreover have $\text{length } ?\sigma = \text{length } ?\tau$
using $\text{encode-problem-parallel-correct-vii}$ $\text{assms}(1)$ $\text{calculation}$
unfolding $\text{decode-state-at-def}$
$\text{decode-plan-def}$
$\text{initial-of-def}$
by fast
ultimately have $k < \text{length } ?\tau - 1$ and $k < t$
unfolding $\text{decode-plan-def}$
$\text{SAT-Plan-Base}. \text{decode-plan-def}$
by force
}

{ note $\text{nb} = \text{this}$
{ have $?G \subseteq_m \text{execute-parallel-plan } ?I ?\pi$
using $\text{encode-problem-serializable-sound}$ $\text{assms}$
unfolding $\text{is-parallel-solution-for-problem-def}$
$\text{decode-plan-def}$
$\text{goal-of-def}$
$\text{initial-of-def}$
by blast
hence $?G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } ?I ?\pi)$
using $\text{execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace}$
by fast
}
moreover {
fix $k$
assume $k$-lt-length-$\pi$: $k < \text{length } ?\pi$
moreover have $k < \text{length } ?\tau - 1$ and $k < t$
  using nb calculation
  by blast+
moreover have are-all-operators-applicable (?$\tau$ ! $k$) (?$\pi$ ! $k$)
  and are-all-operator-effects-consistent (?$\pi$ ! $k$)
  using trace-parallel-plan-strips-operator-preconditions calculation(2)
  by blast+
moreover have are-all-operators-non-interfering (?$\pi$ ! $k$)
  using encode-problem-serializable-sound(2)[OF assms(1, 2)] k-lt-length-$\pi$
  by blast
ultimately have are-all-operators-applicable (?$\tau$ ! $k$) (?$\pi$ ! $k$)
  and are-all-operator-effects-consistent (?$\pi$ ! $k$)
  and are-all-operators-non-interfering (?$\pi$ ! $k$)
  by blast+
}
ultimately show ?thesis
  using execute-parallel-plan-is-execute-sequential-plan-if assms(1)
  by metis
qed

lemma encode-problem-forall-step-decoded-plan-is-serializable-ii:

fixes $\Pi :: 'a variable strips-problem$
shows list-all ($\lambda$op. ListMem op (strips-problem.operators-of $\Pi$))
  (concat ($\Phi^{-1} \Pi . \mathcal{A} . t$))
proof (case list-all)
  let $?\pi = \Phi^{-1} \Pi . \mathcal{A} . t$
  let $?\pi' = \text{concat } ?\pi$

  { have set $?\pi' = \bigcup(\{ k < t. \{ \text{decode-plan}' \Pi . \mathcal{A} . k \} \})$
    unfolding decode-plan-def decode-plan-set-is set-concat
    by auto
  also have ... = $\bigcup(\{ k < t. \{ \text{set} \text{(decode-plan)}' \Pi . \mathcal{A} . k \} \})$
    by blast
  finally have set $?\pi' = (\bigcup k < t. \{ \text{set} \text{(decode-plan)}' \Pi . \mathcal{A} . k \})$
    by blast
  } note nb = this

  { fix op
    assume op $\in$ set $?\pi'$
    then obtain $& \text{where } k < t$ and op $\in$ set (decode-plan' $\Pi . \mathcal{A} . k$)
      using nb
      by blast
    moreover have op $\in$ set (decode-plan $\Pi . \mathcal{A} . t ! k$)
      using calculation
  }
unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
by simp
ultimately have op ∈ set (operators-of Π)
  using decode-plan-step-element-then(1)
  unfolding operators-of-def decode-plan-def
  by blast
}
thus thesis
unfolding list-all-iff ListMem-iff operators-of-def
by blast
qed

Given the soundness and completeness of the SATPlan encoding with interfering operator exclusion \( \Phi_\forall \Pi \ t \), we can now conclude this part with showing that for a parallel plan \( \pi \equiv \Phi^{-1} \Pi \ A \ t \) that was decoded from a model \( A \) of \( \Phi_\forall \Pi \ t \) the serialized plan \( \pi' \equiv \text{concat} \ \pi \) is a serial solution for \( \Pi \). To this end, we have to show that

- the state reached by serial execution of \( \pi' \) subsumes \( G \), and
- all operators in \( \pi' \) are operators contained in \( O \).

While the proof of the latter step is rather straightforward, the proof for the former requires a bit more work. We use the previously established theorem on serial and parallel STRIPS equivalence (theorem ??) to show the serializability of \( \pi \) and therefore have to show that \( G \) is subsumed by the last state of the trace of \( \pi' \)

\[ G \subseteq_m \text{last (trace-sequential-plan-strips I } \pi') \]

and moreover that at every step of the parallel plan execution, the parallel operator execution condition as well as non interference are met

\[ \forall k < \text{length } \pi. \ \text{are-all-operators-non-interfering} \ (\pi ! k) \]

Note that the parallel operator execution condition is implicit in the existence of the parallel trace for \( \pi \) with

\[ G \subseteq_m \text{last (trace-parallel-plan-strips I } \pi) \]

warranted by the soundness of \( \Phi_\forall \Pi \ t \).

**theorem** serializable-encoding-decoded-plan-is-serializable:
assumes is-valid-problem-strips \( \Pi \)
and \( A \models \Phi_\forall \Pi \ t \)
sows is-serial-solution-for-problem \( \Pi \) (concate (\( \Phi^{-1} \Pi \ A \ t \))

---

13 These propositions are shown in lemmas encode_problem_forall_step_decoded_plan_is_serializable_i and encode_problem_forall_step_decoded_plan_is_serializable_i which have been omitted for brevity.
using \textit{encode-problem-forall-step-decoded-plan-is-serializable-i}\{OF \textit{assms}\}
\textit{encode-problem-forall-step-decoded-plan-is-serializable-ii}

unfolding \textit{is-serial-solution-for-problem-def} goal-of-def
initial-of-def \textit{decode-plan-def}
by blast

end

theory \textit{SAT-Solve-SAS-Plus}
imports \textit{SAS-Plus-STRIPS}
\textit{SAT-Plan-Extensions}
begin

9 SAT-Solving of SAS+ Problems

lemma \textit{sas-plus-problem-has-serial-solution-iff-i}:
assumes \textit{is-valid-problem-sas-plus} \(\Psi\)
and \(A \models \Phi_{\varphi} (\varphi \Psi) \, t\)
shows \textit{is-serial-solution-for-problem} \(\Psi\) \([\varphi_{\textbf{op}} \leftarrow \text{concat} \, (\Phi_{\textbf{op}} \,\, t)\, A]\)

proof –
let \(\?\Pi = \varphi \Psi\)
and \(\?\pi' = \text{concat} \, (\Phi_{\textbf{op}} \,\, t)\)
let \(\?\psi = [\varphi_{\textbf{op}} \leftarrow \text{concat} \, (\Phi_{\textbf{op}} \,\, t)]\)

\{ have \textit{is-valid-problem-strips} \?\Pi
using \textit{is-valid-problem-sas-plus-then-strips-transformation-too}\{OF \textit{assms}(1)\},
moreover have \textit{STRIPS-Semantics.is-serial-solution-for-problem} \?\Pi \?\pi'
using calculation \textit{serializable-encoding-decoded-plan-is-serializable}\{OF
- \textit{assms}(2)\}
unfolding \textit{decode-plan-def}
by simp
ultimately have \textit{SAS-Plus-Semantics.is-serial-solution-for-problem} \(\Psi\) \?\psi
using \textit{assms}(1) \textit{serial-strips-equivalent-to-serial-sas-plus}
by blast
\}
thus \?\textit{thesis}
using \textit{serial-strips-equivalent-to-serial-sas-plus}\{OF \textit{assms}(1)\}
by blast
qed

lemma \textit{sas-plus-problem-has-serial-solution-iff-ii}:
assumes \textit{is-valid-problem-sas-plus} \(\Psi\)
and \textit{is-serial-solution-for-problem} \(\Psi\) \?\psi
and \(h = \text{length} \, \?\psi\)
shows \(\exists \, A. \, (A \models \Phi_{\varphi} (\varphi \Psi) \, h)\)

proof –
let \(\?\Pi = \varphi \Psi\)
and \(?\pi = \varphi_P \Psi (\text{embed } \psi)\)
let \(?A = \text{valuation-for-plan } ?\Pi ?\pi\)
let \(?t = \text{length } \psi\)

have nb: length \(\psi = \text{length } ?\pi\)
  unfolding SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
  sasp-op-to-strips-def
  sas-plus-parallel-plan-to-strips-parallel-plan-def
  by (induction \(\psi\); auto)
have is-valid-problem-strips \(?\Pi\)
  using assms(1) is-valid-problem-sas-plus-then-strips-transformation-too
  by blast
moreover have STRIPS-Semantics.is-parallel-solution-for-problem \(?\Pi ?\pi\)
  using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus[OF assms(1,2)]
    strips-equivalent-to-sas-plus[OF assms(1)]
  by blast
moreover {
  fix \(k\)
  assume \(k < \text{length } ?\pi\)
  moreover obtain \(\text{ops}'\) where \(\text{ops}' = ?\pi ! k\)
    by simp
  moreover have \(\text{ops}' \in \text{set } ?\pi\)
    using calculation nth-mem
    by blast
  moreover have \(?\pi = [([\varphi_O \Psi \op. \op \leftarrow \text{ops}], \text{ops} \leftarrow \text{embed } \psi)]\)
    unfolding SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    sas-plus-parallel-plan-to-strips-parallel-plan-def
    ..
  moreover obtain \(\text{ops}\)
    where \(\text{ops}' = [\varphi_O \Psi \op. \op \leftarrow \text{ops}]\)
    and \(\text{ops} \in \text{set } (\text{embed } \psi)\)
    using calculation(3, 4)
    by auto
  moreover have \(\text{ops} \in \{ [\op] \mid \op. \op \in \text{set } \psi \}\)
    using calculation(6) set-of-embed-is
    by blast
  moreover obtain \(\op\)
    where \(\text{ops} = [\op]\) and \(\op \in \text{set } \psi\)
    using calculation(7)
    by blast
  ultimately have are-all-operators-non-interfering (?\pi ! k)
    by fastforce
}
ultimately show ?thesis
  using encode-problem-serializable-complete nb
  by (auto simp: assms(3))
To wrap-up our documentation of the Isabelle formalization, we take a look at the central theorem which combines all the previous theorem to show that SAS+ problems $\Psi$ can be solved using the planning as satisfiability framework.

A solution $\psi$ for the SAS+ problem $\Psi$ exists if and only if a model $A$ and a hypothesized plan length $t$ exist s.t.

$$A \models \Phi_{\forall} (\varphi \Psi) t$$

for the serializable SATPlan encoding of the corresponding STRIPS problem $\Phi_{\forall} \varphi \Psi t$ exist.

**Theorem** sas-plus-problem-has-serial-solution-iff:

- **Assumes** is-valid-problem-sas-plus $\Psi$
- **Shows** $(\exists \psi. \, \text{is-serial-solution-for-problem } \Psi \psi) \iff (\exists A t. \, A \models \Phi_{\forall} (\varphi \Psi) t)$

**Using** sas-plus-problem-has-serial-solution-iff-i[OF assms]

**Using** sas-plus-problem-has-serial-solution-iff-ii[OF assms]

**By** blast

## 10 Adding Noop actions to the SAS+ problem

Here we add noop actions to the SAS+ problem to enable the SAT formula to be satisfiable if there are plans that are shorter than the given horizons.

**Definition** empty-sasp-action $\equiv$ $(\text{SAS-Plus-Representation.sas-plus-operator.precondition-of} = []$, $\text{SAS-Plus-Representation.sas-plus-operator.effect-of} = [])$

**Lemma** sasp-exec-noops: execute-serial-plan-sas-plus $s$ (replicate $n$ empty-sasp-action) $= s$

**By** (induction $n$ arbitrary: )

(auto simp: empty-sasp-action-def STRIPS-Representation.is-operator-applicable-in-def execute-operator-def)

**Definition** prob-with-noop $\Pi \equiv$

lemma sasp-noops-in-noop-problem: set (replicate n empty-sasp-action) \subseteq set (SAS-Plus-Representation.sas-plus-problem (prob-with-noop \Pi))
    by (induction n) (auto simp: prob-with-noop-def)

lemma noops-complete:
    SAS-Plus-Semantics.is-serial-solution-for-problem \psi \pi \Rightarrow
    SAS-Plus-Semantics.is-serial-solution-for-problem (prob-with-noop \psi) ((replicate n empty-sasp-action) \oplus \pi)
    by (induction n)
        (auto simp: SAS-Plus-Semantics.is-serial-solution-for-problem-def insert list.pred-set
         sasp-exec-noops prob-with-noop-def Let-def empty-sasp-action-def elem)

definition rem-noops \equiv filter (\lambda op. op \neq empty-sasp-action)

lemma sasp-filter-empty-action:
    execute-serial-plan-sas-plus s (rem-noops \pi s) = execute-serial-plan-sas-plus s \pi s
    by (induction \pi s arbitrary: s)
        (auto simp: empty-sasp-action-def rem-noops-def)

lemma noops-sound:
    SAS-Plus-Semantics.is-serial-solution-for-problem (prob-with-noop \psi) \pi s \Rightarrow
    SAS-Plus-Semantics.is-serial-solution-for-problem \psi (rem-noops \pi s)
    by (induction \pi s)
        (fastforce simp: SAS-Plus-Semantics.is-serial-solution-for-problem-def insert list.pred-set
         prob-with-noop-def ListMem-iff rem-noops-def
         sasp-filter-empty-action[unfolded empty-sasp-action-def rem-noops-def]
         empty-sasp-action-def)+

lemma noops-valid: is-valid-problem-sas-plus \psi \Rightarrow is-valid-problem-sas-plus (prob-with-noop \psi)
    by (auto simp: is-valid-problem-sas-plus-def prob-with-noop-def Let-def
         empty-sasp-action-def is-valid-operator-sas-plus-def list.pred-set)

lemma sas-plus-problem-has-serial-solution-iff-i':
    assumes is-valid-problem-sas-plus \Psi
    and \A \models \Phi_{\psi} (\varphi (prob-with-noop \Psi)) \tau
    shows SAS-Plus-Semantics.is-serial-solution-for-problem \Psi
        (rem-noops
            (map (\lambdaop. \varphi^{-1}_{\tau} (prob-with-noop \Psi) op)
            \concat (\Phi^{-1} (\varphi (prob-with-noop \Psi)) \A \tau))))
    using assms noops-valid
    by (force intro!: noops-sound sas-plus-problem-has-serial-solution-iff-i)

lemma sas-plus-problem-has-serial-solution-iff-ii':
    assumes is-valid-problem-sas-plus \Psi
    and SAS-Plus-Semantics.is-serial-solution-for-problem \psi \tau
    and length \psi \leq h
shows $\exists A \ (A \models \Phi \ (\varphi (\text{prob-with-nop} \ \Psi)) \ h)$

using assms
by (fastforce
  intro!: assms noops-valid noops-complete
  sas-plus-problem-has-serial-solution-iff-ii
  [where $\psi = (\text{replicate} \ (h - \text{length} \ \psi) \ \text{empty-sasp-action}) \ @ \ \psi] )
end

theory AST-SAS-Plus-Equivalence
imports AI-Planning-Languages-Semantics.SASP-Semantics SAS-Plus-Semantics
List-Index
begin

11 Proving Equivalence of SAS+ representation and Fast-Downward’s Multi-Valued Problem Representation

11.1 Translating Fast-Downward’s representation to SAS+

type-synonym nat-sas-plus-problem = (nat, nat) sas-plus-problem
type-synonym nat-sas-plus-operator = (nat, nat) sas-plus-operator
type-synonym nat-sas-plus-plan = (nat, nat) sas-plus-plan
type-synonym nat-sas-plus-state = (nat, nat) state

definition is-standard-effect :: ast-effect $\Rightarrow$ bool
  where is-standard-effect $\equiv$ $\lambda$ (pre, -, -, -).
    pre = []

definition is-standard-operator :: ast-operator $\Rightarrow$ bool
  where is-standard-operator $\equiv$ $\lambda$(-, -, effects, -).
    list-all is-standard-effect effects

fun rem-effect-implicit-pres :: ast-effect $\Rightarrow$ ast-effect
  where rem-effect-implicit-pres (preconds, v, implicit-pre, eff) =
    (preconds, v, None, eff)

fun rem-implicit-pres :: ast-operator
  where rem-implicit-pres (name, preconds, effects, cost) =
    (name, (implicit-pres effects) @ preconds, map rem-effect-implicit-pres effects, cost)

fun rem-implicit-pres-ops :: ast-problem $\Rightarrow$ ast-problem
  where rem-implicit-pres-ops (vars, init, goal, ops) =
    (vars, init, goal, map rem-implicit-pres-ops ops)

definition consistent-map-lists xs1 xs2 $\equiv$ ($\forall (x1, x2) \in$ set xs1. $\forall (y1, y2) \in$ set xs2.
  x1 = y1 $\rightarrow$ x1 = y2)

lemma map-add-comm: ($\forall x. \ x \in$ dom m1 $\land$ x $\in$ dom m2 $\rightarrow$ m1 x = m2 x)
\[ m1 ++ m2 = m2 ++ m1 \]

by (fastforce simp add: map-add-def split: option.splits)

**lemma** first-map-add-submap: \( (\forall x. x \in \text{dom } m1 \land x \in \text{dom } m2 \Rightarrow m1 x = m2 x) \Rightarrow m1 ++ m2 \subseteq m \)

using map-add-le-mapE map-add-comm

by force

**lemma** subsuming-states-map-add:
\( (\forall x. x \in \text{dom } m1 \land x \in \text{dom } m2 \Rightarrow m1 x = m2 x) \Rightarrow m1 ++ m2 \subseteq m \)

by (auto simp: map-add-le-mapI intro: first-map-add-submap map-add-le-mapE)

**lemma** consistent-map-lists:
\[
[\text{distinct } (\text{map fst } (xs1 \odot xs2)); x \in \text{dom } (\text{map-of } xs1) \cap \text{dom } (\text{map-of } xs2)] \Rightarrow \\
(\text{map-of } xs1) x = (\text{map-of } xs2) x
\]

apply (induction xs1)
apply (simp-all add: consistent-map-lists-def image-def)
using map-of-SomeD
by fastforce

**lemma** subsuming-states-append:
\[
(\text{distinct } (\text{map fst } (xs @ ys)); (\text{map-of } xs) \subseteq m s \iff (\text{map-of } ys) \subseteq m s) \
\Rightarrow (\text{map-of } xs @ ys) \subseteq m s
\]

unfolding map-of-append
apply (intro subsuming-states-map-add)
apply (auto simp add: image-def)
by (metis mono-tags lifting IntI empty_iff fst_conv mem_Collect_eq)

**definition** consistent-pres-op where
consistent-pres-op op \( \equiv \) (case op of \( \text{name, pres, effs, cost} \Rightarrow \text{distinct } \) \( \text{map fst } \) \( \text{(implicit-pres effs)} \))

\( \land \text{consistent-map-lists } \text{pres } \) \( \text{(implicit-pres effs)} \)

**definition** consistent-pres-op’ where
consistent-pres-op’ op \( \equiv \) (case op of \( \text{name, pres, effs, cost} \Rightarrow \text{consistent-map-lists } \text{pres } \) \( \text{(implicit-pres effs)} \))

**lemma** consistent-pres-op-then’: consistent-pres-op op \( \Rightarrow \) consistent-pres-op’ op
by (auto simp add: consistent-pres-op’-def consistent-pres-op-def)

**lemma** rem-implicit-pres-ops-valid-states:
\[
\text{ast-problem.valid-states } (\text{rem-implicit-pres-ops } \text{prob}) = \text{ast-problem.valid-states } \text{prob}
\]

apply (cases prob)
by (auto simp add: ast-problem.valid-states-def ast-problem.Dom-def
ast-problem.numVars-def ast-problem.astDom-def)
lemma rem-implicit-pres-ops-lookup-op-None:
  ast-problem.lookup-operator (vars, init, goal, ops) name = None \iff
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name
  = None
by (induction ops) (auto simp: ast-problem.lookup-operator-def ast-problem.ast\delta-def)

lemma rem-implicit-pres-ops-lookup-op-Some-1:
  ast-problem.lookup-operator (vars, init, goal, ops) name = Some (n,p,vp,e) \implies
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name
  = Some (rem-implicit-pres (n,p,vp,e))
by (induction ops) (fastforce simp: ast-problem.lookup-operator-def ast-problem.ast\delta-def+)

lemma rem-implicit-pres-ops-lookup-op-Some-1\':
  ast-problem.lookup-operator prob name = Some (n,p,vp,e) \implies
  \exists op. ast-problem.lookup-operator (rem-implicit-pres-ops prob) name
  = Some (rem-implicit-pres (n,p,vp,e))
apply (cases prob)
using rem-implicit-pres-ops-lookup-op-Some-1
by simp

lemma implicit-pres-empty: implicit-pres (map rem-effect-implicit-pres effs) = []
by (induction effs) (auto simp: implicit-pres-def)

lemma rem-implicit-pres-ops-lookup-op-Some-2:
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name
  = Some op \implies \exists op'. ast-problem.lookup-operator (vars, init, goal, ops) name
  = Some op' \land
  (op = rem-implicit-pres op')
by (induction ops) (auto simp: ast-problem.lookup-operator-def ast-problem.ast\delta-def implicit-pres-empty image-def)

lemma rem-implicit-pres-ops-lookup-op-Some-2\':
  \exists op'. ast-problem.lookup-operator (rem-implicit-pres-ops prob) name
  = Some (n,p,e,c) \implies
  \exists op'. ast-problem.lookup-operator prob name
  = Some op' \land
  ((n,p,e,c) = rem-implicit-pres op')
apply (cases prob)
using rem-implicit-pres-ops-lookup-op-Some-2
by auto

lemma subsuming-states-def\':
  s \in ast-problem.subsuming-states prob ps = (s \in (ast-problem.valid-states prob)
  \land ps \subseteq_m s)
by (auto simp add: ast-problem.subsuming-states-def)

lemma rem-implicit-pres-ops-enabled-1:
\[
\forall op. \ op \in \text{set} (\text{ast-problem.ast}\prob) \implies \text{consistent-pres-op op}; \\
\text{ast-problem.enabled prob name s} \implies \text{ast-problem.enabled (rem-implicit-pres-ops prob) name s}
\]

by (fastforce simp: ast-problem.enabled-def rem-implicit-pres-ops-valid-states subsuming-states-def)

\text{dest: rem-implicit-pres-ops-lookup-op-Some-1'}

\text{split: option.splits}+

\text{context ast-problem}

\text{begin}

\text{lemma lookup-Some-in}\prob: \text{lookup-operator} \pi = \text{Some op} \implies \text{op \in set ast}\prob

by (auto simp: find-Some-iff in-set-conv-nth lookup-operator-def)

\text{end}

\text{lemma rem-implicit-pres-ops-enabled-2:}

\text{assumes} (\forall op. \ op \in \text{set} (\text{ast-problem.ast}\prob) \implies \text{consistent-pres-op op})

\text{shows} \text{ast-problem.enabled (rem-implicit-pres-ops prob) name s = ast-problem.enabled prob name s}

\text{using assms[OF ast-problem.lookup-Some-in}\prob, unfolded consistent-pres-op-def]

\text{apply} (\text{auto simp: subsuming-states-append rem-implicit-pres-ops-valid-states subsuming-states-def})

\text{dest!: rem-implicit-pres-ops-lookup-op-Some-2'}

\text{split: option.splits})

\text{using subsuming-states-map-add consistent-map-lists}

\text{apply (metis Map.map-add-comm dom-map-of-conv-image-fst map-add-le-mapE)}

\text{using map-add-le-mapE by blast}

\text{lemma rem-implicit-pres-ops-enabled:}

(\forall op. \ op \in \text{set} (\text{ast-problem.ast}\prob) \implies \text{consistent-pres-op op}) \implies 

\text{ast-problem.enabled (rem-implicit-pres-ops prob) name s = ast-problem.enabled prob name s}

\text{using rem-implicit-pres-ops-enabled-1 rem-implicit-pres-ops-enabled-2}

\text{by blast}

\text{context ast-problem}

\text{begin}

\text{lemma std-eff-enabled[simp]:}

\text{is-standard-operator (name, pres, effs, layer) \implies s \in valid-states \implies (filter (eff-enabled s) effs) = effs}

\text{by (induction effs) (auto simp: is-standard-operator-def is-standard-effect-def eff-enabled-def subsuming-states-def)}

\text{end}

\text{lemma is-standard-operator-rem-implicit: is-standard-operator (n,p,\text{vp},v) \implies}

316
is-standard-operator (rem-implicit-pres (n, p, vp,v))
by (induction vp) (auto simp: is-standard-operator-def is-standard-effect-def)

lemma is-standard-operator-rem-implicit-pres-ops:
\[ (\forall \text{op} \in \text{set} (\text{ast-problem.ast}\delta (a,b,c,d)) \implies \text{is-standard-operator op}); \]
\[ \text{op} \in \text{set} (\text{ast-problem.ast}\delta (\text{rem-implicit-pres-ops} (a,b,c,d))) \] \[ \implies \text{is-standard-operator op} \]

lemma is-standard-operator-rem-implicit-pres-ops′:
\[ \forall \text{op} \in \text{set} (\text{ast-problem.ast}\delta \text{rem-implicit-pres-ops prob}); \]
\[ (\forall \text{op} \in \text{set} (\text{ast-problem.ast}\delta \text{prob}) \implies \text{is-standard-operator op}) \] \[ \implies \text{is-standard-operator op} \]
apply (cases prob)
using is-standard-operator-rem-implicit-pres-ops
by blast

lemma in-rem-implicit-pres-δ:
\[ \text{op} \in \text{set} (\text{ast-problem.ast}\delta \text{prob}) \implies \text{rem-implicit-pres op} \in \text{set} (\text{ast-problem.ast}\delta (\text{rem-implicit-pres-ops prob})) \]
by (auto simp add: ast-problem.astδ-def)

lemma rem-implicit-pres-ops-execute:
assumes \[ (\forall \text{op} \in \text{set} (\text{ast-problem.ast}\delta \text{prob}) \implies \text{is-standard-operator op}) \] \[ \text{and s} \in \text{ast-problem.valid-states prob} \]
shows ast-problem.execute (rem-implicit-pres-ops prob) name s = ast-problem.execute prob name s
proof –
have \[ (n,p,s,e,c) \in \text{set} (\text{ast-problem.ast}\delta \text{prob}) \implies \]
\[ (\text{filter (ast-problem.eff-enabled prob s) es} = es \text{ for } n \text{ p } s \text{ es } c) \]
using assms(2)
by (auto simp add: ast-problem.std-eff-enabled dest: rem-implicit-pres-ops-valid-states)
moreover have \[ (n,p,s,e,c) \in \text{set} (\text{ast-problem.ast}\delta \text{prob}) \implies \]
\[ (\text{filter (ast-problem.eff-enabled (rem-implicit-pres-ops prob) s) (map rem-effect-implicit-pres es)}) \]
\[ = \text{map rem-effect-implicit-pres es for } n \text{ p } s \text{ es } c \]
using assms
moreover have map-of (map (λ(-x, -, v). (x,v)) o rem-effect-implicit-pres effs)
\[ = \text{map-of (map (λ(-x, -, v). (x,v)) effs) for effs} \]
by (induction effs) auto
ultimately show ?thesis

317
qed

lemma rem-implicit-pres-ops-path-to:
wf-ast-problem prob \implies
(\forall op. op \in \set{ast-problem.ast \ op prob} \implies consistent-pres-op op) \implies
(\forall op. op \in \set{ast-problem.ast \ op prob} \implies is-standard-operator op) \implies
s \in ast-problem.valid-states prob \implies
ast-problem.path-to (rem-implicit-pres-ops prob) s = s \ast-problem.path-to prob s s'

by (induction \pi s arbitrary: s)
(auto simp: rem-implicit-pres-ops-execute rem-implicit-pres-ops-enabled
ast-problem.path-to.simps wf-ast-problem.execute-preserves-valid)

lemma rem-implicit-pres-ops-astG[simp]: ast-problem.astG (rem-implicit-pres-ops prob) =
ast-problem.astG prob

apply(cases prob)
by (auto simp add: ast-problem.astG-def)

lemma rem-implicit-pres-ops-goal[simp]: ast-problem.G (rem-implicit-pres-ops prob)
= ast-problem.G prob

apply(cases prob)
using rem-implicit-pres-ops-valid-states

lemma rem-implicit-pres-ops-astI[simp]:
ast-problem.astI (rem-implicit-pres-ops prob) = ast-problem.astI prob

apply(cases prob)
by (auto simp add: ast-problem.astI-def ast-problem.astG-def subsuming-states-def')

lemma rem-implicit-pres-ops-init[simp]: ast-problem.I (rem-implicit-pres-ops prob)
= ast-problem.I prob

apply(cases prob)
by (auto simp add: ast-problem.I-def ast-problem.astI-def)

lemma rem-implicit-pres-ops-valid-plan:
assumes wf-ast-problem prob
(\forall op. op \in \set{ast-problem.ast \ op prob} \implies consistent-pres-op op)
(\forall op. op \in \set{ast-problem.ast \ op prob} \implies is-standard-operator op)
shows ast-problem.valid-plan (rem-implicit-pres-ops prob) \pi s = ast-problem.valid-plan prob \pi s

using wf-ast-problem.I-valid[OF assms(1) rem-implicit-pres-ops-path-to[OF assms]]
by (simp add: ast-problem.valid-plan-def rem-implicit-pres-ops-goal rem-implicit-pres-ops-init)

lemma rem-implicit-pres-ops-numVars[simp]:
ast-problem.numVars (rem-implicit-pres-ops prob) = ast-problem.numVars prob

by (cases prob) (simp add: ast-problem.numVars-def ast-problem.astDom-def)

lemma rem-implicit-pres-ops-numVals[simp]:
\[ \text{ast-problem.numVals} (\text{rem-implicit-pres-ops} \ \text{prob}) \ x = \text{ast-problem.numVals} \ \text{prob} \ x \]
\[ \text{by} \ (\text{cases} \ \text{prob}) \ (\text{simp add: ast-problem.numVals-def ast-problem.astDom-def}) \]

\textbf{lemma} \textit{in-implicit-pres}:
\[(x, a) \in \text{set} \ (\text{implicit-pres} \ \text{effs}) \implies (\exists \text{epres} \ v \ \text{vp}. \ (\text{epres},x,v,\text{vp}) \in \text{set} \ \text{effs} \ \wedge \ \text{vp} = \text{Some} \ a) \]
\[ \text{by} \ (\text{induction} \ \text{effs}) \ (\text{fastforce simp: implicit-pres-def image-def split: if-splits})+ \]

\textbf{lemma} \textit{pair4-eqD}:
\[(a_1,a_2,a_3,a_4) = (b_1,b_2,b_3,b_4) \implies a_3 = b_3 \]
\[ \text{by} \ \text{simp} \]

\textbf{lemma} \textit{rem-implicit-pres-ops-wf-partial-state}:
\[ \text{ast-problem.wf-partial-state} (\text{rem-implicit-pres-ops} \ \text{prob}) \ s = \]
\[ \text{ast-problem.wf-partial-state} \ \text{prob} \ s \]
\[ \text{by} \ (\text{auto simp: ast-problem.wf-partial-state-def}) \]

\textbf{lemma} \textit{rem-implicit-pres-wf-operator}:
\[ \text{assumes consistent-pres-op} \ \text{op} \]
\[ \text{shows} \]
\[ \text{ast-problem.wf-operator} (\text{rem-implicit-pres-ops} \ \text{prob}) (\text{rem-implicit-pres op}) \]
\[ \text{proof} \]
\[ \text{obtain name pres effs cost where} \ \text{op} = (\text{name}, \text{pres}, \text{effs}, \text{cost}) \]
\[ \text{by} \ (\text{cases \ op}) \]
\[ \text{hence \ asses: consistent-pres-op} (\text{name}, \text{pres}, \text{effs}, \text{cost}) \]
\[ \text{ast-problem.wf-operator} \ \text{prob} (\text{name}, \text{pres}, \text{effs}, \text{cost}) \]
\[ \text{using} \ \text{assms} \]
\[ \text{by} \ \text{auto} \]
\[ \text{hence \ distinct} (\text{map \ fst} \ ((\text{implicit-pres \ effs}) @ \text{pres})) \]
\[ \text{by} \ (\text{simp only: consistent-pres-op-def}) \ \text{auto} \]
\[ \text{moreover \ have} \ x < \text{ast-problem.numVars} (\text{rem-implicit-pres-ops} \ \text{prob}) \]
\[ v < \text{ast-problem.numVars} (\text{rem-implicit-pres-ops} \ \text{prob}) \ x \]
\[ \text{if} \ (x,v) \in \text{set} \ ((\text{implicit-pres \ effs}) @ \text{pres}) \ \text{for} \ x \ v \]
\[ \text{using} \ \text{that} \ \text{asses} \]
\[ \text{by} \ (\text{auto dest!: in-implicit-pres simp: ast-problem.wf-partial-state-def ast-problem.wf-operator-def}) \]
\[ \text{ultimately \ have} \]
\[ \text{ast-problem.wf-partial-state} (\text{rem-implicit-pres-ops} \ \text{prob}) ((\text{implicit-pres \ effs}) \ @ \text{pres}) \]
\[ \text{by} \ (\text{auto simp only: ast-problem.wf-partial-state-def}) \]
\[ \text{moreover \ have} \ (\text{map} (\lambda(\cdot, v, \cdot, \cdot). v) \ \text{effs} = \]
\[ (\text{map} (\lambda(\cdot, v, \cdot, \cdot). v) \ (\text{map \ rem-effect-implicit-pres \ effs})) \]
\[ \text{by} \ \text{auto} \]
\[ \text{hence \ distinct} (\text{map} (\lambda(\cdot, v, \cdot, \cdot). v) \ (\text{map \ rem-effect-implicit-pres \ effs})) \]
\[ \text{using} \ \text{assms(2)} \]
\[ \text{by} \ (\text{auto simp only: op \ ast-problem.wf-operator-def \ rem-implicit-pres.simps dest!: pair4-eqD}) \]
\[ \text{moreover \ have} \ (\exists \text{vp}. \ (\text{epres},x,\text{vp},v) \in \text{set} \ \text{effs} \ \iff \ (\text{epres},x,\text{None},v) \in \text{set} \ (\text{map \ rem-effect-implicit-pres \ effs})) \]
\[ \text{for} \ \text{epres} \ x \ v \]

319
by force
ultimately show thesis
using assms(2)
by (auto simp: op ast-problem.wf-operator-def rem-implicit-pres-ops-wf-partial-state
         split: prod.splits)
qed

lemma rem-implicit-pres-ops-inD: op ∈ set (ast-problem.astδ (rem-implicit-pres-ops prob))
         ⟹ (∃ op'. op' ∈ set (ast-problem.astδ prob) ∧ op = rem-implicit-pres op')
by (cases prob) (force simp: ast-problem.astδ-def)

lemma rem-implicit-pres-ops-well-formed:
assumes (∀ op. op ∈ set (ast-problem.astδ prob) ⟹ consistent-pres-op op)
shows ast-problem.well-formed (rem-implicit-pres-ops prob)
proof --
have map fst (ast-problem.astδ (rem-implicit-pres-ops prob)) = map fst (ast-problem.astδ prob)
by (cases prob) (auto simp: ast-problem.astδ-def)
thus thesis using assms by (auto simp add: ast-problem.well-formed-def rem-implicit-pres-ops-wf-partial-state
                                 simp del: rem-implicit-pres-simps
dest!: rem-implicit-pres-ops-inD
intro!: rem-implicit-pres-wf-operator)
qed

definition is-standard-effect' :: ast-effect ⇒ bool
where is-standard-effect' ≡ λ(pre, -, vpre, -). pre = [] ∧ vpre = None

definition is-standard-operator' :: ast-operator ⇒ bool
where is-standard-operator' ≡ λ( -, -, effects, -). list-all is-standard-effect' effects

lemma rem-implicit-pres-is-standard-operator':
is-standard-operator (n,p,es,c) ⟹ is-standard-operator' (rem-implicit-pres (n,p,es,c))
by (induction es) (auto simp: is-standard-operator'-def is-standard-operator-def
       is-standard-effect-def
       is-standard-effect'-def)

lemma rem-implicit-pres-ops-is-standard-operator':
(∀ op. op ∈ set (ast-problem.astδ (vs, I, G, ops))) ⟹ is-standard-operator op
⟹
π ∈ set (ast-problem.astδ (rem-implicit-pres-ops (vs, I, G, ops))) ⟹ is-standard-operator'
π
by (cases ops) (auto simp: ast-problem.astδ-def dest!: rem-implicit-pres-is-standard-operator')
locale abs-ast-prob = wf-ast-problem +
assumes no-cond-effs: ∀π ∈ set ast. is-standard-operator’ π

context ast-problem
begin

definition abs-ast-variable-section = [0..<(length astDom)]

definition abs-range-map :: (nat ⇀ nat list)
where abs-range-map ≡
  map-of (zip abs-ast-variable-section
           (map ((λvals. [0..<length vals]) o snd o snd)
                astDom))

end

context abs-ast-prob
begin

lemma is-valid-vars-1: astDom ≠ [] ⇒ abs-ast-variable-section ≠ []
  by (simp add: abs-ast-variable-section-def)

end

lemma upt-eq-Nil-conv: ([] = [i..<j]) = (j = 0 ∨ j ≤ i)
  by (induct j) simp-all

lemma map-of-zip-map-Some:
  v < length xs
  ⇒ (map-of (zip [0..<length xs] (map f xs)) v) = Some (f (xs ! v))
  by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)

lemma map-of-zip-Some:
  v < length xs
  ⇒ (map-of (zip [0..<length xs] xs) v) = Some (xs ! v)
  by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)

lemma in-set-zip-lengthE:
  (x,y) ∈ set (zip [0..<length xs] xs) ⇒ ([] < length xs; xs ! x = y) ⇒ R ⇒ R
  by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)

context abs-ast-prob
begin

lemma is-valid-vars-2:
  shows list-all (λv. abs-range-map v ≠ None) abs-ast-variable-section

321
by (auto simp add: abs-range-map-def abs-ast-variable-section-def list.pred-set)
end

context ast-problem
begin
definition abs-ast-initial-state :: nat-sas-plus-state where
abs-ast-initial-state ≡ map-of (zip [0..<length astI] astI)
end

context abs-ast-prob
begin

lemma valid-abs-init-1: abs-ast-initial-state v ≠ None ↔ v ∈ set abs-ast-variable-section
by (simp add: abs-ast-variable-section-def numVars-def wf-initial (1) abs-ast-initial-state-def)

lemma abs-range-map-Some:
shows v ∈ set abs-ast-variable-section →
(abs-range-map v) = Some [0..<length (snd (snd (astDom ! v)))]
by (simp add: numVars-def abs-range-map-def o-def abs-ast-variable-section-def map-of-zip-map-Some)

lemma in-abs-v-sec-length: v ∈ set abs-ast-variable-section ←→ v < length astDom
by (simp add: abs-ast-variable-section-def)

lemma [simp]: v < length astDom → (abs-ast-initial-state v) = Some (astI ! v)
using wf-initial (1) [simplified numVars-def, symmetric]
by (auto simp add: map-of-Some abs-ast-initial-state-def split: prod.splits)

lemma [simp]: v < length astDom → astI ! v < length (snd (snd (astDom ! v)))
using wf-initial (1) [simplified numVars-def, symmetric] uf-initial
by (auto simp add: numVals-def abs-ast-initial-state-def split: prod.splits)

lemma [intro!]: v ∈ set abs-ast-variable-section → x < length (snd (snd (astDom ! v))) →
               x ∈ set (the (abs-range-map v))
using abs-range-map-Some
by (auto simp add: )

lemma [intro!]: x<length astDom → astI ! x < length (snd (snd (astDom ! x)))
using wf-initial [unfolded numVars-def numVals-def]
by auto

lemma [simp]: abs-ast-initial-state v = Some a → a < length (snd (snd (astDom ! v)))
by (auto simp add: abs-ast-initial-state-def)
lemma valid-abs-init-2:
abs-ast-initial-state v \neq \text{None} \implies (\text{the (abs-ast-initial-state v)}) \in \text{set (the (abs-range-map v)})
using valid-abs-init-1
by auto
end

context ast-problem
begin

definition abs-ast-goal :: nat-sas-plus-state
where abs-ast-goal \equiv \text{map-of astG}
end

context abs-ast-prob
begin

lemma [simp]: wf-partial-state s \implies (v, a) \in \text{set s} \implies v \in \text{set abs-ast-variable-section}
by (auto simp add: wf-partial-state-def abs-ast-variable-section-def numVars-def split: prod.splits)

lemma valid-abs-goal-1: abs-ast-goal v \neq \text{None} \implies v \in \text{set abs-ast-variable-section}
using wf-goal
by (auto simp add: abs-ast-goal-def dest!: map-of-SomeD)

lemma in-abs-rangeI: wf-partial-state s \implies (v, a) \in \text{set s} \implies (a \in \text{set (the (abs-range-map v)))}
by (auto simp add: abs-range-map-Some wf-partial-state-def numVals-def split: prod.splits)

lemma valid-abs-goal-2:
abs-ast-goal v \neq \text{None} \implies (\text{the (abs-ast-goal v)}) \in \text{set (the (abs-range-map v))}
using wf-goal
by (auto simp add: map-of-SomeD weak-map-of-SomeI abs-ast-goal-def intro!: in-abs-rangeI)
end

context ast-problem
begin

definition abs-ast-operator :: ast-operator \Rightarrow nat-sas-plus-operator
where abs-ast-operator  ≡ λ(name, preconditions, effects, cost).

\( () \ preconditions-of \ = \ preconditions, \)
\( effect-of = [(v, x). (-, v, -) ← effects] () \)

end

context abs-ast-prob
begin

lemma abs-rangeI: wf-partial-state s ⇒ (v, a) ∈ set s ⇒ (abs-range-map v ≠ None)
by (auto simp add: wf-partial-state-def abs-range-map-def abs-ast-variable-section-def list.pred-set
  numVars-def
  split: prod.splits)

lemma abs-valid-operator-1[intro!]:
wf-operator op ⇒ list-all (λ(v, a). ListMem v abs-ast-variable-section)
(precondition-of (abs-ast-operator op))
by (cases op; auto simp add: abs-ast-operator-def wf-operator-def list)
  pred-set ListMem-iff)

lemma wf-operator-preD: wf-operator (name, pres, effs, cost) ⇒ wf-partial-state pres
by (simp add: wf-operator-def)

lemma abs-valid-operator-2[intro!]:
wf-operator op ⇒
list-all (λ(v, a), (∃ y. abs-range-map v = Some y) ∧ ListMem a (the (abs-range-map v)))
(precondition-of (abs-ast-operator op))
by (cases op, auto dest!: wf-operator-preD simp: list.pred-set ListMem-iff abs-ast-operator-def
intro!: abs-rangeI[simplified not-None-eq in-abs-rangeI]

lemma wf-operator-effE: wf-operator (name, pres, effs, cost) ⇒
(∀ distinct (map (λ(-, v, -). v) effs):
  \( \land \) epres x vp v. (epres,x,vp,v)∈ set effs ⇒ wf-partial-state epres;
  \( \land \) epres x vp v. (epres,x,vp,v)∈ set effs ⇒ x < numVars;
  \( \land \) epres x vp v. (epres,x,vp,v)∈ set effs ⇒ v < numVals x;
  \( \land \) epres x vp v. (epres,x,vp,v)∈ set effs ⇒
    case vp of None ⇒ True | Some v ⇒ v < numVals x]
  ⇒ P)
⇒ P

unfolding wf-operator-def
by (auto split: prod.splits)

lemma abs-valid-operator-3' :
wf-operator (name, pre, eff, cost) ⇒
\[\text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{map } (\lambda(\cdot, \cdot, \cdot, a). (v, a)) \text{ eff})\]
by (fastforce simp add: list.pred-set ListMem-iff abs-ast-variable-section-def image-def numVars-def elim!: wf-operator-effE split: prod.split)

**lemma abs-valid-operator-3[intro!]:**

\[\text{wf-operator } \rightarrow\]
\[\text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{effect-of } (\text{abs-ast-operator } op))\]
by (cases op, simp add: abs-ast-operator-def abs-valid-operator-3)

**lemma abs-valid-operator-4':**

\[\text{wf-operator } (\lambda(v, a). (\text{abs-range-map } v \neq \text{None}) \land \text{ListMem } a \text{ (the } (\text{abs-range-map } v))) (\text{map } (\lambda(\cdot, \cdot, \cdot, a). (v, a)) \text{ eff})\]
apply (subst list.pred-set ListMem-iff)+
apply (drule wf-eff)
by (metis (mono-tags, lifting) abs-rangeI case-prodI2 in-abs-rangeI)

**lemma abs-valid-operator-5':**

\[\text{wf-operator } \rightarrow\]
\[\text{list-all } (\lambda(v, a). \exists y. (\text{abs-range-map } v = \text{Some } y) \land \text{ListMem } a \text{ (the } (\text{abs-range-map } v)))\]
using abs-valid-operator-4'
by (cases op, simp add: abs-ast-operator-def)

**lemma consistent-list-set: **

\[\text{wf-partial-state } s \rightarrow\]
\[\text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \lor a = a') s) s\]
by (auto simp add: list.pred-set wf-partial-state-def eq-key-imp-eq-value split: prod.splits)

**lemma abs-valid-operator-5':**

\[\text{wf-operator } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \lor a = a') \text{ pre}) \text{ pre}\]
apply (drule wf-operator-preD)
by (intro consistent-list-set)

**lemma abs-valid-operator-5[intro!]:**

\[\text{wf-operator } \rightarrow\]
\[\text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \lor a = a') (\text{precondition-of } (\text{abs-ast-operator } op)))\]
(by intro consistent-list-set)
using abs-valid-operator-5′
by (cases op, simp add: abs-ast-operator-def)

lemma consistent-list-set-2; distinct (map fst s) \implies
list-all (λ(v, a). list-all (λ(v’, a’). v ≠ v’ ∨ a = a’) s) s
by (auto simp add: list.pred-set wf-partial-state-def eq-key-imp-eq-value split: prod.splits)

lemma abs-valid-operator-6′:
assumes wf-operator (name, pre, eff, cost)
shows list-all (λ(v, a). list-all (λ(v’, a’). v ≠ v’ ∨ a = a’) (map (λ(·, v, ·, a). (v, a)) eff))
(map (λ(·, v, ·, a). (v, a)) eff)
proof−
have *: map fst (map (λ(·, v, ·, a). (v, a)) eff) = (map (λ(·, v, ·). v) eff)
by (induction eff) auto
show ?thesis
using assms
apply (elim wf-operator-effE)
apply (intro consistent-list-set-2)
by (subst *)
qed

lemma abs-valid-operator-6[intro!]:
wf-operator op \implies
 list-all (λ(v, a). list-all (λ(v’, a’). v ≠ v’ ∨ a = a’) (effect-of (abs-ast-operator op))))
(effect-of (abs-ast-operator op))
using abs-valid-operator-6′
by (cases op, simp add: abs-ast-operator-def)

end

context ast-problem
begin

definition abs-ast-operator-section :: nat-sas-plus-operator list
where abs-ast-operator-section ≡ [abs-ast-operator op. op ← astδ]

definition abs-prob :: nat-sas-plus-problem
where abs-prob = { variables-of = abs-ast-variable-section,
operators-of = abs-ast-operator-section,
initial-of = abs-ast-initial-state,
goal-of = abs-ast-goal,
range-of = abs-range-map }

lemma \[\text{simp}]: \text{op} \in \text{set ast} \Rightarrow (\text{is-valid-operator-sas-plus abs-prob}) (\text{abs-ast-operator op})
\begin{align*}
\text{apply}(\text{cases op}) \\
\text{apply}(\text{subst is-valid-operator-sas-plus-def Let-def})+ \\
\text{using uf-operators(2)} \\
\text{by(fastforce simp add: abs-prob-def})+
\end{align*}

lemma \text{abs-ast-operator-section-valid}:
\text{list-all} (\text{is-valid-operator-sas-plus abs-prob}) \text{abs-ast-operator-section}
\begin{align*}
\text{by (auto simp: abs-ast-operator-section-def list.pred-set)}
\end{align*}

lemma \text{abs-prob-valid}:
\text{is-valid-problem-sas-plus abs-prob}
\begin{align*}
\text{using valid-abs-goal-1 valid-abs-goal-2 valid-abs-init-1 is-valid-vars-2} \\
\text{abs-ast-operator-section-valid[unfolded abs-prob-def]}
\end{align*}
\begin{align*}
\text{by (auto simp add: is-valid-problem-sas-plus-def Let-def ListMem-iff abs-prob-def)}
\end{align*}

definition \text{abs-ast-plan}:: \text{SASP-Semantics.plan} \Rightarrow \text{nat-sas-plus-plan}
\begin{align*}
\text{where abs-ast-plan } \pi s \\
\equiv \text{map (abs-ast-operator o the o lookup-operator) } \pi s
\end{align*}

lemma \text{std-then-implici-effs}[simp]: \text{is-standard-operator'}(\text{name, pres, effs, layer})
\Rightarrow \text{implicit-pres effs = []}
\begin{align*}
\text{apply(induction effs)} \\
\text{by (auto simp add: is-standard-operator'}-def implicit-pres-def is-standard-effect'-def)
\end{align*}

lemma \text{effs-eq-abs-effs}: (\text{effect-of (abs-ast-operator (name, pres, effs, layer)})) =
\begin{align*}
\text{map (x,x,v, v) effs}
\end{align*}
\begin{align*}
\text{by (auto simp add: abs-ast-operator-def} \\
\text{split: option.splits prod.splits)}
\end{align*}

lemma \text{exect-eq-abs-execute}:
\begin{align*}
[\text{enabled } \pi s; \text{lookup-operator } \pi = \text{Some (name, preconds, effs, layer)}; \\
\text{is-standard-operator'}(\text{name, preconds, effs, layer})] \Rightarrow \\
\text{execute } \pi s = (\text{execute-operator-sas-plus s ((abs-ast-operator o the o lookup-operator) } \\
\pi))
\end{align*}
\begin{align*}
\text{using effs-eq-abs-effs}
\end{align*}
by (auto simp add: execute-def execute-operator-sas-plus-def)

lemma enabled-then-sas-applicable:
  enabled π s \implies\ SAS-Plus-Representation.is-operator-applicable-in s ((abs-ast-operator o the o lookup-operator) π)
  by (auto simp add: subsuming-states-def enabled-def lookup-operator-def
       SAS-Plus-Representation.is-operator-applicable-in-def abs-ast-operator-def
       split: option.splits prod.splits)

lemma path-to-then-exec-serial: \(\forall \pi \in \sett \pi s.\ lookup-operator \pi \neq \text{None} \implies\)
  path-to s π s s' \implies\ s' \subseteq_m \text{execute-serial-plan-sas-plus s (abs-ast-plan π s)}
proof (induction π s arbitrary: s s')
  case (Cons a π s)
  then show \?case
    by (force simp add: exect-eq-abs-execute abs-ast-plan-def lookup-Some-in δ no-cond-effs
        dest: enabled-then-sas-applicable)
qed (auto simp add: execute-serial-plan-sas-plus-def abs-ast-plan-def)

lemma map-of-eq-None-iff:
  \(\text{None} = \text{map-of xys x} \) = \(x \notin \text{fst '(set xys)}\)
by (induct xys) simp-all

lemma [simp]: I = abs-ast-initial-state
  apply (intro HOL.ext)
  by (auto simp add: map-of-eq-None-iff set-map[ symmetric] I-def abs-ast-initial-state-def
       map-of-zip-Some
       dest: map-of-SomeD)

lemma [simp]: \(\forall \pi \in \sett \pi s.\ lookup-operator \pi \neq \text{None} \implies\)
  op\in \sett (abs-ast-plan π s) \implies\ op \in \sett abs-ast-operator-section
by (induction π s) (auto simp: abs-ast-plan-def abs-ast-operator-section-def lookup-Some-in δ)
end

context ast-problem
begin

lemma path-to-then-lookup-Some: \(\exists s' \in \text{G. path-to s π s s'} \) \implies\ \(\forall \pi \in \sett \pi s.\ lookup-operator \pi \neq \text{None}\)
by (induction π s arbitrary: s) (force simp add: enabled-def split: option.splits)+

lemma valid-plan-then-lookup-Some: valid-plan π s \implies\ \(\forall \pi \in \sett \pi s.\ lookup-operator \pi \neq \text{None}\)
  using path-to-then-lookup-Some
  by (simp add: valid-plan-def)
end
context abs-ast-prob
begin

theorem valid-plan-then-is-serial-sol:
  assumes valid-plan π
  shows is-serial-solution-for-problem abs-prob (abs-ast-plan π)
using valid-plan-then-lookup-Some[OF assms] assms
end

11.2 Translating SAS+ representation to Fast-Downward’s
context ast-problem
begin

definition lookup-action:: nat-sas-plus-operator ⇒ ast-operator option where
  lookup-action op ≡
    find (λ(-, pres, effs, -). precondition-of op = pres ∧
      map (λ(v,a). ([], v, None, a)) (effect-of op) = effs)
      astδ
end

context abs-ast-prob
begin

lemma find-Some: find P xs = Some x ⇒ x ∈ set xs ∧ P x
  by (auto simp add: find-Some-iff)

lemma distinct-find: distinct (map f xs) ⇒ x ∈ set xs ⇒ find (λx′. f x′ = f x) xs = Some x
  by (induction xs) (auto simp: image-def)

lemma lookup-operator-find: lookup-operator nme = find (λop. fst op = nme) astδ
  by (auto simp: lookup-operator-def intro!: arg-cong[where f = (λx. find x astδ)])

lemma lookup-operator-works-1: lookup-action op = Some π' ⇒ lookup-operator (fst π') = Some π'
  by (auto simp: wf-operators(1) lookup-operator-find lookup-action-def dest: find-Some intro: distinct-find)

lemma lookup-operator-works-2:
  lookup-action (abs-ast-operator (name, pres, effs, layer)) = Some (name', pres', effs', layer')

329
lemma [simp]: is-standard-operator' (name, pres, effs, layer) \implies
map (λ(v,a). ([], v, None, a)) (effect-of (abs-ast-operator (name, pres, effs, layer))) = effs
by (induction effs) (auto simp: is-standard-operator'-def abs-ast-operator-def is-standard-effect'-def)

lemma lookup-operator-works-3:
  is-standard-operator' (name, pres, effs, layer) \implies (name, pres, effs, layer) ∈ set astδ \implies
  lookup-action (abs-ast-operator (name, pres, effs, layer)) = Some (name', pres', effs', layer')
  \implies effs = effs'
by (auto simp: is-standard-operator'-def lookup-action-def dest!: find-Some)

lemma mem-find-Some: x ∈ set xs \implies P x \implies ∃ x'. find P xs = Some x'
by (induction xs) auto

lemma [simp]: precondition-of (abs-ast-operator (x1, a, aa, b)) = a
by (simp add: abs-ast-operator-def)

lemma std-lookup-action: is-standard-operator' ast-op \implies ast-op ∈ set astδ \implies
  ∃ ast-op'. lookup-action (abs-ast-operator ast-op) = Some ast-op'
unfolding lookup-action-def
apply (intro mem-find-Some)
by (auto split: prod.splits simp: o-def)

lemma is-applicable-then-enabled-1:
  ast-op ∈ set astδ \implies
  ∃ ast-op'. lookup-operator ((fst o the o lookup-action o abs-ast-operator) ast-op)
= Some ast-op'
using lookup-operator-works-1 std-lookup-action no-cond-effs
by auto

lemma lookup-action-Some-in-δ: lookup-action op = Some ast-op \implies ast-op ∈ set astδ
using lookup-operator-works-1 lookup-Some-inδ by fastforce

lemma lookup-operator-eq-name: lookup-operator name = Some (name', pres, effs, layer) \implies name = name'
using lookup-operator-af (2)
by fastforce

lemma eq-name-eq-pres: (name, pres, effs, layer) ∈ set astδ \implies (name, pres', effs', layer') ∈ set astδ
⇒ pres = pres'

⇒ pres = pres'
by (auto simp: lookup-action-def abs-ast-operator-def dest!: find-Some)
using eq-key-imp-eq-value[OF wf-operators(1)]
by auto

lemma eq-name-eq-effs:
  name = name' \implies (name, pres, effs, layer) \in \text{set ast} \implies (name', pres', effs', layer') \in \text{set ast} \implies effs = effs'
using eq-key-imp-eq-value[OF wf-operators(1)]
by auto

lemma is-applicable-then-subsumes:
  s \in \text{valid-states} \implies \text{SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator (name, pres, effs, layer))} \implies s \in \text{subsuming-states (map-of pres)}
by (simp add: subsuming-states-def SAS-Plus-Representation.is-operator-applicable-in-def abs-ast-operator-def)

lemma eq-name-eq-pres':
  \[ s \in \text{valid-states}; \text{is-standard-operator'} (name, pres, effs, layer); (name, pres, effs, layer) \in \text{set ast} \delta ; \text{lookup-operator} ((\text{fst o the o lookup-action o abs-ast-operator}) (name, pres, effs, layer)) = \text{Some} (name', pres', effs', layer') \]
  \implies pres = pres'
using lookup-operator-eq-name lookup-operator-works-2
by (fastforce dest!: std-lookup-action
  simp: eq-name-eq-pres[OF lookup-action-Some-in-\delta lookup-Some-in-\delta])

lemma is-applicable-then-enabled-2:
  \[ s \in \text{valid-states}; \text{ast-op} \in \text{set ast} \delta ; \text{SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator ast-op)}; \text{lookup-operator} ((\text{fst o the o lookup-action o abs-ast-operator}) \text{ast-op}) = \text{Some} (name, pres, effs, layer) \]
  \implies s \in \text{subsuming-states (map-of pres)}
apply(cases ast-op)
using eq-name-eq-pres' is-applicable-then-subsumes no-cond-effs
by fastforce

lemma is-applicable-then-enabled-3:
  \[ s \in \text{valid-states}; \text{lookup-operator} ((\text{fst o the o lookup-action o abs-ast-operator}) \text{ast-op}) = \text{Some} (name, pres, effs, layer) \]
  \implies s \in \text{subsuming-states (map-of (implicit-pres effs))}
apply(cases ast-op)
using no-cond-effs
by (auto dest!: std-then-implici-effs std-lookup-action lookup-Some-in-\delta
  simp: subsuming-states-def)

lemma is-applicable-then-enabled:
\[ s \in \text{valid-states}; \text{ast-op} \in \text{set ast}\delta; \]
\[
\text{SAS-Plus-Representation.is-operator-applicable-in } s \; (\text{abs-ast-operator ast-op}) \]
\[
\implies \text{enabled } ((\text{fst o the o lookup-action o abs-ast-operator}) \; \text{ast-op}) \; s
\]
using is-applicable-then-enabled-1 is-applicable-then-enabled-2 is-applicable-then-enabled-3

by (simp add: enabled-def split: option.splits)

**lemma** eq-name-eq-effs':
assumes lookup-operator ((fst o the o lookup-action o abs-ast-operator) (name, pres, effs, layer)) =
Some (name', pres', effs', layer')
is-standard-operator' (name, pres, effs, layer) (name, pres, effs, layer) \in set ast\delta
s \in \text{valid-states}
shows effs = effs'
using std-lookup-action[OF assms (2,3)] assms
by (auto simp: lookup-operator-works-3[OF assms (2,3)]
eq-name-eq-effs[OF lookup-operator-eq-name lookup-Some-in-\delta lookup-Some-in-\delta])

**lemma** std-eff-enabled'[simp]:
is-standard-operator' (name, pres, effs, layer) \implies s \in \text{valid-states} \implies (filter (eff-enabled s) effs) = effs
by (induction effs) (auto simp: is-standard-operator'-def is-standard-effect'-def eff-enabled-def subsuming-states-def)

**lemma** execute-abs:
\[ s \in \text{valid-states}; \text{ast-op} \in \text{set ast}\delta; \]
\[
\text{SAS-Plus-Representation.is-operator-applicable-in } s \; (\text{abs-ast-operator ast-op}) \]
\[
\implies \text{execute } ((\text{fst o the o lookup-action o abs-ast-operator}) \; \text{ast-op}) \; s =
\text{execute-operator-sas-plus s} \; (\text{abs-ast-operator ast-op})
\]
using no-cond-effs
by (cases ast-op)
(fastforce simp add: execute-def execute-operator-sas-plus-def effs-eq-abs-effs
dest: is-applicable-then-enabled-1 eq-name-eq-effs'[unfolded o-def]
split: option.splits)+

**fun** sat-preconds-as where
sat-preconds-as s [] = True
| sat-preconds-as s (op#ops) =
(SAS-Plus-Representation.is-operator-applicable-in s op \land
sat-preconds-as (execute-operator-sas-plus s op) ops)

**lemma** exec-serial-then-path-to':
\[ s \in \text{valid-states}; \]
\[
\forall \text{op} \in \text{set ops}. \exists \text{ast-op} \in \text{set ast}\delta. \text{op} = \text{abs-ast-operator ast-op};
(sat-preconds-as s ops) \implies
\text{path-to s} \; (\text{map } (\text{fst o the o lookup-action}) \; \text{ops}) \; (\text{execute-serial-plan-sas-plus s ops})
\]

332
proof (induction ops arbitrary: s)

case (Cons a ops)

then show ?case
  using execute-abs is-applicable-then-enabled execute-preserves-valid
apply simp
by metis
qed auto
end

fun rem-condless-ops where
rem-condless-ops s [] = []
| rem-condless-ops s (op#ops) =
  (if SAS-Plus-Representation.is-operator-applicable-in s op then
   op #(rem-condless-ops (execute-operator-sas-plus s op) ops)
  else [])

context abs-ast-prob
begin

lemma exec-rem-consdless: execute-serial-plan-sas-plus s (rem-condless-ops s ops) =
execute-serial-plan-sas-plus s ops
by (induction ops arbitrary: s) auto

lemma rem-conless-sat: sat-preconds-as s (rem-condless-ops s ops)
by (induction ops arbitrary: s) auto

lemma set-rem-condlessD: x ∈ set (rem-condless-ops s ops) ⇒ x ∈ set ops
by (induction ops arbitrary: s) auto

lemma exec-serial-then-path-to:
[s ∈ valid-states;
∀ op∈set ops. ∃ ast-op∈ set astδ. op = abs-ast-operator ast-op] ⇒
path-to s ((map (fst o the o lookup-action)) o rem-condless-ops s ops) (execute-serial-plan-sas-plus s ops)
using rem-conless-sat
by (fastforce dest!: set-rem-condlessD
intro!: exec-serial-then-path-to'
[where s = s and ops = rem-condless-ops s ops,
unfolded exec-rem-consdless])

lemma is-serial-solution-then-abstracted:
is-serial-solution-for-problem abs-prob ops
⇒ ∀ op∈set ops. ∃ ast-op∈ set astδ. op = abs-ast-operator ast-op
by (auto simp: is-serial-solution-for-problem-def abs-prob-def Let-def list.pred-set
ListMem-iff abs-ast-operator-section-def
split: if-splits)

lemmalookup-operator-works-1': lookup-action op = Some π' ⇒ ∃ op. lookup-operator

333
\[(\text{fst } \pi') = \text{op}\]
\[\text{using lookup-operator-works-1 by auto}\]

\text{lemma is-serial-sol-then-valid-plan-1:}
\[\text{is-serial-solution-for-problem abs-prob ops;}\]
\[\pi \in \text{set } ((\text{map } (\text{fst } o \text{ the } o \text{ lookup-action}) o \text{ rem-condless-ops I } \text{ ops})) \implies \text{lookup-operator } \pi \neq \text{None}\]
\[\text{using std-lookup-action lookup-operator-works-1 no-cond-effs}\]
\[\text{by } (\text{fastforce dest!: set-rem-condlessD is-serial-solution-then-abstracted}\]
\[\text{simp: valid-plan-def list.pred-set ListMem-iff })\]

\text{lemma is-serial-sol-then-valid-plan-2:}
\[\text{is-serial-solution-for-problem abs-prob ops} \implies ((\exists s' \in G. \text{ path-to I } ((\text{map } (\text{fst } o \text{ the } o \text{ lookup-action}) o \text{ rem-condless-ops I } \text{ ops}) s'))\]
\[\text{using I-valid}\]
\[\text{by } (\text{fastforce intro: path-to-pres-valid exec-serial-then-path-to}\]
\[\text{intro!: bexI[where } z = \text{execute-serial-plan-sas-plus I ops]}\]
\[\text{dest: is-serial-solution-then-abstracted}\]
\[\text{simp: list.pred-set ListMem-iff abs-ast-operator-section-def}\]
\[\text{G-def subsuming-states-def is-serial-solution-for-problem-def}\]
\[\text{abs-prob-def abs-ast-goal-def})+\]

\text{end}\n
\text{context ast-problem}\n\text{begin}\n\text{definition decode-abs-plan } \equiv (\text{map } (\text{fst } o \text{ the } o \text{ lookup-action}) o \text{ rem-condless-ops I})\n\text{end}\n
\text{context abs-ast-prob}\n\text{begin}\n\text{theorem is-serial-sol-then-valid-plan:}\n\[\text{is-serial-solution-for-problem abs-prob ops} \implies \text{valid-plan } (\text{decode-abs-plan ops})\]
\[\text{using is-serial-sol-then-valid-plan-1 is-serial-sol-then-valid-plan-2}\]
\[\text{by (simp add: valid-plan-def decode-abs-plan-def})\]
\text{end}\n
\text{end}\n
334
theory Solve-SASP

begin

11.3 SAT encoding works for Fast-Downward’s representation

context abs-ast-prob

begin

theorem is-serial-sol-then-valid-plan-encoded:
\[ \mathcal{A} \models \varphi (\text{prob-with-noop abs-prob}) t \implies \text{valid-plan} \]
\[
\begin{align*}
\text{decode-abs-plan} \\
\text{rem-noops} \\
\text{map} (\Lambda \text{op}. \varphi^{-1} (\text{prob-with-noop abs-prob}) \text{op}) \\
\text{concat} (\Phi^{-1} (\varphi (\text{prob-with-noop abs-prob}) \mathcal{A} t)))
\end{align*}
\]

by (fastforce intro!: is-serial-sol-then-valid-plan abs-prob-valid sas-plus-problem-has-serial-solution-iff-i')

lemma length-abs-ast-plan: length \pi s = length (abs-ast-plan \pi s)

by (auto simp: abs-ast-plan-def)

theorem valid-plan-then-is-serial-sol-encoded:
valid-plan \pi s \implies length \pi s \leq h \implies \exists \mathcal{A}. \mathcal{A} \models \varphi (\text{prob-with-noop abs-prob})

h

apply(subst (asm) length-abs-ast-plan)

by (fastforce intro!: sas-plus-problem-has-serial-solution-iff-it' abs-prob-valid valid-plan-then-is-serial-sol)

end

12 DIMACS-like semantics for CNF formulae

We now push the SAT encoding towards a lower-level representation by replacing the atoms which have variable IDs and time steps into natural numbers.

lemma gTD: \((l::nat) < n) \implies (\exists m. n = Suc m \land l \leq m)

by (induction n) auto

locale cnf-to-dimacs =

fixes h :: nat and n-ops :: nat

begin

fun var-to-dimacs where

var-to-dimacs (Operator t k) = 1 + t + k * h
\[ \text{var-to-dimacs} \ (\text{State} \ t \ k) = 1 + n-\text{ops} \ast h + t + k \ast (h) \]

**definition dimacs-to-var**

\[
\begin{align*}
\text{dimacs-to-var} \ v & \equiv \\
& \text{if } v < 1 + n-\text{ops} \ast h \text{ then} \\
& \quad \text{Operator} \ ((v - 1) \text{ mod } (h)) \ ((v - 1) \text{ div } (h)) \\
& \text{else} \\
& \quad \text{let } k = ((v - 1) - n-\text{ops} \ast h) \text{ in} \\
& \quad \text{State} \ (k \text{ mod } (h)) \ (k \text{ div } (h))
\end{align*}
\]

**fun valid-state-var**

\[
\begin{align*}
\text{valid-state-var} \ (\text{Operator} \ t \ k) & \iff t < h \land k < n-\text{ops} \\
\text{valid-state-var} \ (\text{State} \ t \ k) & \iff t < h
\end{align*}
\]

**lemma State-works:**

\[
\begin{align*}
\text{valid-state-var} \ (\text{State} \ t \ k) & \implies \\
\text{dimacs-to-var} \ \left(\text{var-to-dimacs} \ (\text{State} \ t \ k)\right) & = \\
& \text{(State} \ t \ k) \\
& \text{by (induction } k \text{) (auto simp add: dimacs-to-var-def add.left-commute Let-def)}
\end{align*}
\]

**lemma Operator-works:**

\[
\begin{align*}
\text{valid-state-var} \ (\text{Operator} \ t \ k) & \implies \\
\text{dimacs-to-var} \ \left(\text{var-to-dimacs} \ (\text{Operator} \ t \ k)\right) & = \\
& \text{(Operator} \ t \ k) \\
& \text{by (induction } k \text{) (auto simp add: algebra-simps dimacs-to-var-def gr0-conv-Suc nat-le-iff-add dest: gtD)}
\end{align*}
\]

**lemma sat-plan-to-dimacs-works:**

\[
\begin{align*}
\text{valid-state-var} \ sv & \implies \\
\text{dimacs-to-var} \ \left(\text{var-to-dimacs} \ sv\right) & = sv \\
& \text{apply(cases } sv\text{)} \\
& \text{using State-works Operator-works} \\
& \text{by auto}
\end{align*}
\]

**end**

**lemma changing-atoms-works:**

\[
\begin{align*}
(\forall x. \ P \ x & \implies (f \circ g) \ x = x) \implies (\forall x \in \text{atoms } phi. \ P \ x) \implies M \models phi \iff M \circ f \models g \phi \\
& \text{by (induction } phi\text{) auto}
\end{align*}
\]

**lemma changing-atoms-works':**

\[
\begin{align*}
M \circ g & \models phi \iff M \models \text{map-formula } g \phi \\
& \text{by (induction } phi\text{) auto}
\end{align*}
\]

**context cnf-to-dimacs**

**begin**

**lemma sat-plan-to-dimacs:**

336
(\(\Lambda sv. sv \in \text{atoms sats-plan-formula} \implies \text{valid-state-var} \ sv\)) \implies 
\(M \models \text{sats-plan-formula}\)
\(\iff M \circ \text{dimacs-to-var} \models \text{map-formula var-to-dimacs sats-plan-formula}\)
\(\text{by(}\text{auto intro! changing-atoms-works where } P = \text{valid-state-var} \text{ simp: sats-plan-to-dimacs-works})\)

**Lemma dimacs-to-sat-plan:**
\(M \circ \text{var-to-dimacs} \models \text{sats-plan-formula}\)
\(\iff M \models \text{map-formula var-to-dimacs sats-plan-formula}\)
\(\text{using changing-atoms-works'}\).

**Locale sat-solve-sasp = abs-ast-prob \(\Pi + \text{cnf-to-dimacs} \ Suc \ h \ Suc \ (\text{length ast}\delta)\) for \(\Pi \ h\)**

**Lemma encode-initial-state-valid:**
\(sv \in \text{atoms (encode-initial-state Prob)} \implies \text{valid-state-var} \ sv\)
\(\text{by (}\text{auto simp add: encode-state-variable-def Let-def encode-initial-state-def split: sat-plan-variable.splits bool.splits})\)

**Lemma length-operators:**
\(\text{length (operators-of (}\varphi (\text{prob-with-noop abs-prob})) = Suc (\text{length ast}\delta)\)

**Lemma encode-operator-effect-valid-1:**
\(t < h \implies op \in \text{set (operators-of (}\varphi (\text{prob-with-noop abs-prob}))\) \implies \)
\(sv \in \text{atoms (}\bigwedge (map (\lambda v. \neg (\text{Atom (Operator t (index (operators-of (}\varphi (\text{prob-with-noop abs-prob})) op)))) \lor \text{Atom (State (Suc t) (index vs v))) \lor \text{valid-state-var sv}}\)
\(\text{using length-operators}\)
\(\text{by (}\text{induction asse})\text{ (}\text{auto simp: simp add: cnf-to-dimacs.valid-state-var.simps})\)

**Lemma encode-operator-effect-valid-2:**
\(t < h \implies op \in \text{set (operators-of (}\varphi (\text{prob-with-noop abs-prob}))\) \implies \)
\(sv \in \text{atoms (}\bigwedge (map (\lambda v. \neg (\text{Atom (Operator t (index (operators-of (}\varphi (\text{prob-with-noop abs-prob})) op)))) \lor \neg (\text{Atom (State (Suc t) (index vs v))) \lor \text{valid-state-var sv}}\)
\(\text{using length-operators}\)
by (induction asses) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)

end

lemma atoms-And-append: atoms (\(\bigwedge (as1 @ as2)\)) = atoms (\(\bigwedge as1\) \(\cup\) atoms (\(\bigwedge as2\))
by (induction as1) auto

context sat-solve-sasp

begin

lemma encode-operator-effect-valid:
sv \(\in\) atoms (encode-operator-effect (\(\varphi\) (prob-with-noop abs-prob)) t op) \(\implies\)
t < h \(\implies\) op \(\in\) set (operators-of (\(\varphi\) (prob-with-noop abs-prob))) \(\implies\)
valid-state-var sv
by (force simp: encode-operator-effect-def Let-def atoms-And-append
introl: encode-operator-effect-valid-1 encode-operator-effect-valid-2)

end

lemma foldr-And: foldr (\(\land\)) as (\(\neg\) \(\bot\)) = (\(\bigwedge\) as)
by (induction as) auto

context sat-solve-sasp

begin

lemma encode-all-operator-effects-valid:
t < Suc h \(\implies\)
sv \(\in\) atoms (encode-all-operator-effects (\(\varphi\) (prob-with-noop abs-prob)) (operators-of
(\(\varphi\) (prob-with-noop abs-prob))) t) \(\implies\)
valid-state-var sv
unfolding encode-all-operator-effects-def foldr-And
by (force simp add: encode-operator-effect-valid)

end

lemma encode-operator-precondition-valid-1:
t < h \(\implies\) op \(\in\) set (operators-of (\(\varphi\) (prob-with-noop abs-prob))) \(\implies\)
sv \(\in\) atoms
(\(\bigwedge\)(map (\(\lambda\)v. 
\(\neg\) (Atom (Operator t (index (operators-of (\(\varphi\) (prob-with-noop abs-prob))) op)))) \(\lor\) Atom (State t (f v))))
asses)) \(\implies\)
valid-state-var sv
using length-operators
by (induction asses) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)

lemma encode-operator-precondition-valid:
sv \(\in\) atoms (encode-operator-precondition (\(\varphi\) (prob-with-noop abs-prob)) t op)
\(\implies\)
t < h \(\implies\) op \(\in\) set (operators-of (\(\varphi\) (prob-with-noop abs-prob))) \(\implies\)
valid-state-var sv
by (force simp: encode-operator-precondition-def Let-def
intro: encode-operator-precondition-valid-1)

lemma encode-all-operator-preconditions-valid:
t < Suc h ⟹
  sv ∈ atoms (encode-all-operator-preconditions (φ (prob-with-noop abs-prob))
operators-of (φ (prob-with-noop abs-prob))) t ⟹
valid-state-var sv
unfolding encode-all-operator-preconditions-def foldr-And
by (force simp add: encode-operator-precondition-valid)

lemma encode-operators-valid:
sv ∈ atoms (encode-operators (φ (prob-with-noop abs-prob)) t) ⟹ t < Suc h
⟹
valid-state-var sv
unfolding encode-operators-def Let-def
by (force simp add: encode-all-operator-preconditions-valid encode-all-operator-effects-valid)

lemma encode-negative-transition-frame-axiom':
t < h ⟹
set deleting-operators ⊆ set (operators-of (φ (prob-with-noop abs-prob))) ⟹
sv ∈ atoms
  (¬Atom (State t v-idx))
  ∨ (Atom (State (Suc t) v-idx))
  ∨ (map (λop. Atom (Operator t (index (operators-of (φ (prob-with-noop
  abs-prob))) op))))) deleting-operators)) ⟹
valid-state-var sv
by (induction deleting-operators) (auto simp: length-operators[symmetric] cnf-to-dimacs.valid-state-var.simps)

lemma encode-negative-transition-frame-axiom-valid:
sv ∈ atoms (encode-negative-transition-frame-axiom (φ (prob-with-noop abs-prob)))
t v) ⟹ t < h ⟹
valid-state-var sv
unfolding encode-negative-transition-frame-axiom-def Let-def
apply(intro encode-negative-transition-frame-axiom'[of t])
by auto

lemma encode-positive-transition-frame-axiom-valid:
sv ∈ atoms (encode-positive-transition-frame-axiom (φ (prob-with-noop abs-prob))
t v) ⟹ t < h ⟹
valid-state-var sv
unfolding encode-positive-transition-frame-axiom-def Let-def
apply(intro encode-negative-transition-frame-axiom'[of t])
by auto

lemma encode-all-frame-axioms-valid:
sv ∈ atoms (encode-all-frame-axioms (φ (prob-with-noop abs-prob)) t) ⟹ t <
Suc h $\rightarrow$
valid-state-var sv

unfolding encode-all-frame-axioms-def Let-def atoms-And-append
by (force simp add; encode-negative-transition-frame-axiom-valid encode-positive-transition-frame-axiom-valid)

lemma encode-goal-state-valid:
sv $\in$ atoms (encode-goal-state Prob t) $\rightarrow$ t $<$ Suc h $\rightarrow$ valid-state-var sv
by (auto simp add; encode-state-variable-def Let-def encode-goal-state-def split: sat-plan-variable. splits bool.splits)

lemma encode-problem-valid:
sv $\in$ atoms (encode-goal-state (ϕ (prob-with-noop abs-prob)) h) $\rightarrow$ valid-state-var sv
unfolding encode-problem-def
using encode-initial-state-valid encode-operators-valid encode-all-frame-axioms-valid encode-goal-state-valid
by fastforce

lemma encode-interfering-operator-pair-exclusion-valid:
sv $\in$ atoms (encode-interfering-operator-pair-exclusion (ϕ (prob-with-noop abs-prob)))
t op_1 op_2 $\rightarrow$ t $<$ Suc h $\rightarrow$
operators-of (ϕ (prob-with-noop abs-prob)) $\rightarrow$ op_2 $\in$ set
(operators-of (ϕ (prob-with-noop abs-prob))) $\rightarrow$
valid-state-var sv
by (auto simp: encode-interfering-operator-pair-exclusion-def Let-def foldr-And)

lemma encode-interfering-operator-exclusion-valid:
sv $\in$ atoms (encode-interfering-operator-exclusion (ϕ (prob-with-noop abs-prob)))
t $\rightarrow$ t $<$ Suc h $\rightarrow$
valid-state-var sv
unfolding encode-interfering-operator-exclusion-def Let-def foldr-And
by (force simp add: encode-interfering-operator-pair-exclusion-valid)

lemma encode-problem-with-operator-interference-exclusion-valid:
sv $\in$ atoms (encode-problem-with-operator-interference-exclusion (ϕ (prob-with-noop abs-prob)))
h $\rightarrow$ valid-state-var sv
unfolding encode-problem-with-operator-interference-exclusion-def
using encode-initial-state-valid encode-operators-valid encode-all-frame-axioms-valid encode-goal-state-valid
encode-interfering-operator-exclusion-valid
by fastforce

lemma planning-by-cnf-dimacs-complete:
valid-plan π s $\rightarrow$ length π s $\leq$ h $\rightarrow$
$\exists M. M \models map-formula var-to-dimacs (ϕ (prob-with-noop abs-prob)) h$
using valid-plan-then-is-serial-sol-encoded
by meson
lemma planning-by-cnf-dimacs-sound:
\[
\mathcal{A} \models \text{map-formula var-to-dimacs} (\Phi \forall (\varphi (\text{prob-with-noop abs-prob})) t) \implies \\
\text{valid-plan} \quad \\
\quad \text{(decode-abs-plan)} \\
\quad \text{(rem-noops)} \\
\quad (\text{map} (\lambda \varphi. \varphi^{-1} (\text{prob-with-noop abs-prob}) \text{op}) \\
\quad (\text{concat} (\Phi^{-1} (\varphi (\text{prob-with-noop abs-prob})) (\mathcal{A} \circ \text{var-to-dimacs}) t))))
\]
using changing-atoms-works
by (fastforce intro: is-serial-sol-then-valid-plan-encoded)
end

12.1 Going from Formualae to DIMACS-like CNF

We now represent the CNF formulae into a very low-level representation that is reminiscent to the DIMACS representation, where a CNF formula is a list of list of integers.

fun disj-to-dimacs :: nat formula => int list where
\[
\text{disj-to-dimacs} (\varphi_1 \lor \varphi_2) = \text{disj-to-dimacs} \varphi_1 @ \text{disj-to-dimacs} \varphi_2
\]
| \text{disj-to-dimacs} \bot = []
| \text{disj-to-dimacs} (\text{Not} \bot) = [-1::int,1::int]
| \text{disj-to-dimacs} (\text{Atom} v) = [int v]
| \text{disj-to-dimacs} (\text{Not} (\text{Atom} v)) = [-(int v)]

fun cnf-to-dimacs :: nat formula => int list list where
\[
\text{cnf-to-dimacs} (\varphi_1 \land \varphi_2) = \text{cnf-to-dimacs} \varphi_1 @ \text{cnf-to-dimacs} \varphi_2
\]
| \text{cnf-to-dimacs} \bot = [\text{disj-to-dimacs} \bot]

definition dimacs-lit-to-var l \equiv \text{nat} (\text{abs} l)

definition find-max (xs::nat list) \equiv (\text{fold max xs 1})

lemma find-max-works:
\[
x \in \text{set} \, xs \implies x \leq \text{find-max} \, xs \quad (\text{is} \ ?P \implies \ ?Q)
\]
proof
have \(x \in \text{set} \, xs \implies (x::\text{nat}) \leq (\text{fold max} \, xs \, m)\) for \(m\)
unfolding max-def
apply (induction \(xs\) arbitrary; \(m\) rule: rev-induct)
using nat-le-linear
by (auto dest: le-trans simp add:)
thus \(?P \implies \ ?Q\)
by(auto simp add: find-max-def max-def)
qed

fun formula-vars where
\[
\text{formula-vars} (\bot) = [] | \\
\text{formula-vars} (\text{Atom} k) = [k] | \\
\text{formula-vars} (\text{Not} F) = \text{formula-vars} F | \\
\]
341
\[ (\text{And } F \ G) = F \circ G \]
\[ (\text{Imp } F \ G) = F \circ G \]
\[ (\text{Or } F \ G) = F \circ G \]

\textbf{lemma} atoms-formula-vars: \( f = \text{set } (\text{formula-vars } f) \)
\hspace{1em} \text{by (induction } f\text{) auto}

\textbf{lemma} max-var: \( v \in \text{atoms } (f::\text{nat formula}) \implies v \leq \text{find-max } (\text{formula-vars } f) \)
\hspace{1em} \text{using find-max-works}
\hspace{1em} \text{by (simp add: atoms-formula-vars)}

\textbf{definition} dimacs-max-var \( cs \equiv \text{find-max } (\text{map } (\text{find-max } o (\text{map } \text{nat } o \text{abs}))) cs \)

\textbf{lemma} fold-max-ge: \( b \leq a \implies (b::\text{nat}) \leq \text{fold } (\lambda x m. \text{if } m \leq x \text{ then } x \text{ else } m) \ ys a \)
\hspace{1em} \text{by (induction } ys\text{ arbitrary: } a b\text{) auto}

\textbf{lemma} find-max-append: \( \text{find-max } (xs @ ys) = \text{max } (\text{find-max } xs) (\text{find-max } ys) \)
\hspace{1em} \text{apply (simp only: Max.set-eq-fold[symmetric] append-Cons[symmetric] set-append find-max-def)}
\hspace{1em} \text{by (metis List.finite-set Max.union Un-absorb Un-insert-left Un-insert-right list.distinct(1) list.simps(15) set-empty)}

\textbf{definition} dimacs-model::\text{int list } \Rightarrow \text{int list list } \Rightarrow \text{bool where}
\hspace{1em} \text{dimacs-model } ls cs \equiv (\forall c \in \text{set } cs. (\exists l \in \text{set } ls. l \in \text{set } c)) \land \text{distinct } (\text{map } \text{dimacs-lit-to-var } ls)

\textbf{fun} model-to-dimacs-model \text{ where}
\hspace{1em} \text{model-to-dimacs-model } M (v#vs) = (\text{if } M v \text{ then int } v \text{ else } -(\text{int } v)) # (\text{model-to-dimacs-model } M vs)
\hspace{1em} \text{| model-to-dimacs-model } -[] = []

\textbf{lemma} model-to-dimacs-model-append:
\hspace{1em} \text{set } (\text{model-to-dimacs-model } M (vs @ vs')) = \text{set } (\text{model-to-dimacs-model } M vs) \cup \text{set } (\text{model-to-dimacs-model } M vs')
\hspace{1em} \text{by (induction } vs\text{) auto}

\textbf{lemma} upt-append-sing: \( xs @ [x] = [a..<\text{n-vars}] \implies a < \text{n-vars} \implies (xs = [a..<\text{n-vars} - 1] \land x = \text{n-vars} - 1 \land \text{n-vars} > 0) \)
\hspace{1em} \text{by (induction } \text{n-vars}\text{) auto}

\textbf{lemma} upt-eqD: \( \text{upt } a \ b = \text{upt } a \ b' \implies (b = b' \lor b' \leq a \lor b \leq a) \)
\hspace{1em} \text{by (induction } b\text{) (auto dest: upt-append-sing split: if-splits)}

\textbf{lemma} pos-in-model: \( M n \Rightarrow 0 < n \Rightarrow n < \text{n-vars} \Rightarrow \text{int } n \in \text{set } (\text{model-to-dimacs-model } M [1..<\text{n-vars}]) \)
\hspace{1em} \text{by (induction } \text{n-vars}\text{) (auto simp add: less-Suc-eq model-to-dimacs-model-append}
lemma neg-in-model: \( \neg M \ n \implies 0 < n \implies n < n\text{-vars} \implies -(\text{int} \ n) \in \text{set} M[1..<n\text{-vars}] \)
   by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma in-model: \( 0 < n \implies n < n\text{-vars} \implies \text{int} \ n \in \text{set} M[1..<n\text{-vars}] \)
   by (metis)

lemma model-to-dimacs-model-all-vars:
   \((\forall v \in \text{atoms} \ f. \ 0 < v \land v < n\text{-vars}) \implies \text{is-cnf} \ f \implies M \models f \implies \)
   \((\forall n < n\text{-vars}. \ 0 < n \implies (\text{int} \ n) \in \text{set} M[(1::\text{nat})..<n\text{-vars}]) \)
   \lor \(- (\text{int} \ n) \in \text{set} M[(1::\text{nat})..<n\text{-vars}]))) \)
   by (auto simp add: le-less model-to-dimacs-model-append split: if-splits)

lemma cnf-And: set (cnf-to-dimacs (f1 \& f2)) = set (cnf-to-dimacs f1) \cup set (cnf-to-dimacs f2)
   by auto

lemma one-always-in:
   \(1 < n\text{-vars} \implies 1 \in \text{set} M[1..<n\text{-vars}] \lor -1 \in \text{set} M[1..<n\text{-vars}] \)
   by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma [simp]: (disj-to-dimacs (f1 \lor f2)) = (disj-to-dimacs f1) \oplus (disj-to-dimacs f2)
   by auto

lemma [simp]: (atoms (f1 \lor f2)) = atoms f1 \cup atoms f2
   by auto

lemma isdisj-disjD: (is-disj (f1 \lor f2)) \implies is-disj f1 \land is-disj f2
   by (cases f1; auto)

lemma disj-to-dimacs-sound:
   \(1 < n\text{-vars} \implies (\forall v \in \text{atoms} \ f. \ 0 < v \land v < n\text{-vars}) \implies \text{is-disj} \ f \implies M \models f \implies \)
   \(\exists l \in \text{set} M[(1::\text{nat})..<n\text{-vars}]. \ l \in \text{set} \text{disj-to-dimacs} f) \)
   apply (induction f)
   using neg-in-model pos-in-model one-always-in
   by (fastforce elim!: is-lit-plus.elims dest!: isdisj-disjD)+

lemma is-cnf-disj: is-cnf (f1 \lor f2) \implies (\forall f. \ f1 \lor f2 = f \implies is-disj f \implies P)
   \implies P
   by auto
lemma cnf-to-dimacs-disj: is-disj f \implies cnf-to-dimacs f = [disj-to-dimacs f]
by (induction f) auto

lemma model-to-dimacs-model-all-clauses:
I < n-vars \implies (\forall v \in \text{atoms } f. 0 < v \land v < n-vars) \implies is-cnf f \implies M \models f \implies 
\exists l \in \text{set } (\text{cnf-to-dimacs-model } M [[1::nat)..<n-vars]]).
l \in \text{set } c
proof (induction f arbitrary: )
then show ?case
  using in-model neg-in-model
  by (fastforce elim!: is-lit-plus.elims+)
next
case (Or f1 f2)
then show ?case
  using cnf-to-dimacs-disj disj-to-dimacs-sound
  by (elim is-cnf-disj, simp)
qed (insert in-model neg-in-model pos-in-model, auto)

lemma upt-eq-Cons-conv:
(x \# xs = [i..<j]) = (i < j \land i = x \land [i+1..<j] = xs)
using upt-eq-Cons-conv
by metis

lemma model-to-dimacs-model-append':
(model-to-dimacs-model M (vs @ vs')) = (model-to-dimacs-model M vs) @ (model-to-dimacs-model M vs')
by (induction vs) auto

lemma model-to-dimacs-neg-nin:
n-vars \leq x \implies \text{int } x \notin \text{set } (\text{model-to-dimacs-model } M [a..<n-vars])
by (induction n-vars arbitrary: a) (auto simp: model-to-dimacs-model-append')

lemma model-to-dimacs-pos-nin:
n-vars \leq x \implies - \text{int } x \notin \text{set } (\text{model-to-dimacs-model } M [a..<n-vars])
by (induction n-vars arbitrary: a) (auto simp: model-to-dimacs-model-append')

lemma int-cases2' :
z \neq 0 \implies (\land n. 0 \neq (\text{int } n) \implies z = \text{int } n \implies P) \implies (\land n. 0 \neq - (\text{int } n) \implies z = - (\text{int } n) \implies P)
by (metis (full-types) int-cases2)

lemma model-to-dimacs-model-distinct:
I < n-vars \implies \text{distinct } (\text{map } \text{dimacs-lit-to-var } (\text{model-to-dimacs-model } M [1..<n-vars]))
byp (induction n-vars)
  (fastforce elim!: int-cases2'
    simp add: dimacs-lit-to-var-def model-to-dimacs-model-append' model-to-dimacs-neg-nin model-to-dimacs-pos-nin)
lemma model-to-dimacs-model-sound:
\[ 1 < n \text{-vars} \implies (\forall v \in \text{atoms } f. \ 0 < v \land v < n \text{-vars}) \implies \text{is-cnf } f \implies M \models f \implies \text{dimacs-model (model-to-dimacs-model } M [((\text{nat})..<n \text{-vars})] \text{ (cnf-to-dimacs } f) \]

unfolding dimacs-model-def
using model-to-dimacs-model-all-vars model-to-dimacs-model-all-clauses model-to-dimacs-model-distinct
by auto

lemma model-to-dimacs-model-sound-exists:
\[ 1 < n \text{-vars} \implies (\forall v \in \text{atoms } f. \ 0 < v \land v < n \text{-vars}) \implies \text{is-cnf } f \implies M \models f \implies \exists M \text{-dimacs. dimacs-model } M \text{-dimacs (cnf-to-dimacs } f) \]
using model-to-dimacs-model-sound
by metis

definition dimacs-to-atom :: \text{int} \Rightarrow \text{nat formula}
where
\[ \text{dimacs-to-atom } l \equiv \begin{cases} \text{Not} \left( \text{Atom} \left( \text{nat} \left( \text{abs } l \right) \right) \right) & \text{if } (l < 0) \\ \text{Atom} \left( \text{nat} \left( \text{abs } l \right) \right) & \text{else} \end{cases} \]

definition dimacs-to-disj :: \text{int list} \Rightarrow \text{nat formula}
where
\[ \text{dimacs-to-disj } f \equiv \bigvee \left( \text{map dimacs-to-atom } f \right) \]

definition dimacs-to-cnf :: \text{int list list} \Rightarrow \text{nat formula}
where
\[ \text{dimacs-to-cnf } f \equiv \bigwedge \left( \text{map dimacs-to-disj } f \right) \]

definition dimacs-model-to-abs \text{dimacs-M } M 
\equiv 
\begin{cases} \text{fold} \left( \lambda l M. \left( \begin{array}{l} \text{if } (l > 0) \text{ then } M \left( \text{nat} \left( \text{abs } l \right) \right) := \text{True} \text{ else } M \left( \text{nat} \left( \text{abs } l \right) \right) := \text{False} \end{array} \right) \text{ dimacs-M } M \right) 
\equiv \text{map dimacs-model-to-abs dimacs-M } M \text{ x} \end{cases}

proof (induction dimacs-M arbitrary: M rule: rev-induct)
case (suc a dimacs-M)
thus ?case
by (auto simp add: dimacs-model-to-abs-def dimacs-lit-to-var-def image-def)

qed auto

lemma dimacs-model-to-abs-atom':
\[ 0 < x \implies \neg \left( \begin{array}{c} \text{int } x \in \text{set dimacs-M} \implies \text{distinct } \text{map dimacs-lit-to-var dimacs-M} \\
\implies \text{dimacs-model-to-abs dimacs-M } M \text{ x} \end{array} \right) \]

proof (induction dimacs-M arbitrary: M rule: rev-induct)
case (suc a dimacs-M)
thus ?case
by (auto simp add: dimacs-model-to-abs-def dimacs-lit-to-var-def image-def)

qed auto

lemma model-to-dimacs-model-complete-disj:
\[ (\forall v \in \text{atoms } f. \ 0 < v \land v < n \text{-vars}) \implies \text{is-disj } f \implies \text{distinct } \text{map dimacs-lit-to-var} \]
\[
\text{dimacs-M} \quad \Rightarrow \quad \text{dimacs-model dimacs-M} \quad \text{cnf-to-dimacs f} \quad \Rightarrow \quad \text{dimacs-model-to-abs dimacs-M} \\
(\lambda\text{-} \text{False}) \models f \\
\text{by (induction f)} \\
\quad (\text{fastforce elim!: is-lit-plus.elims dest!: isdisj-disjD}) \\
\quad \text{simp: cnf-to-dimacs-disj dimacs-model-def dimacs-model-to-abs-atom'} \\
\quad \text{dimacs-model-to-abs-atom} \quad +
\]

lemma model-to-dimacs-model-complete:
(\forall v \in \text{atoms } f . 0 < v \land v < n-vars) \Rightarrow \text{is-cnf f} \Rightarrow \text{distinct (map dimacs-lit-to-var dimacs-M)} \\
\Rightarrow \text{dimacs-model dimacs-M} \quad (\text{cnf-to-dimacs f}) \Rightarrow \text{dimacs-model-to-abs dimacs-M} \\
(\lambda\text{-} \text{False}) \models f \\
\text{proof (induction f)} \\
\quad \text{case (Not f)} \\
\quad \quad \text{then show ?case} \\
\quad \quad \quad \text{by (auto elim!: is-lit-plus.elims simp add: dimacs-model-to-abs-atom'} \text{dimacs-model-def)} \\
\quad \text{next} \\
\quad \quad \text{case (Or f1 f2)} \\
\quad \quad \quad \text{then show ?case} \\
\quad \quad \quad \quad \text{using cnf-to-dimacs-disj model-to-dimacs-model-complete-disj} \\
\quad \quad \quad \quad \quad \text{by (elim is-cnf-disj, simp add: dimacs-model-def)} \\
\quad \text{qed (insert dimacs-model-to-abs-atom, auto simp: dimacs-model-def)}

lemma model-to-dimacs-model-complete-max-var:
(\forall v \in \text{atoms } f . 0 < v) \Rightarrow \text{is-cnf f} \Rightarrow \\
\text{dimacs-model dimacs-M} \quad (\text{cnf-to-dimacs f}) \Rightarrow \\
\text{dimacs-model-to-abs dimacs-M} \quad (\lambda\text{-} \text{False}) \models f \\
\text{using le-imp-less-Suc [OF max-var]} \\
\text{by (auto intro!: model-to-dimacs-model-complete simp: dimacs-model-def)}

lemma model-to-dimacs-model-sound-max-var:
(\forall v \in \text{atoms } f . 0 < v) \Rightarrow \text{is-cnf f} \Rightarrow \\
\text{dimacs-model (model-to-dimacs-model M [(1::nat)..<(find-max (formula-vars f)) + 2])} \\
\text{(cnf-to-dimacs f)} \\
\text{using le-imp-less-Suc[unfolded Suc-eq-plus1, OF max-var]} \\
\text{by (fastforce intro!: model-to-dimacs-model-sound)}

context sat-solve-sasp 
begin

lemma [simp]: var-to-dimacs sv > 0 \\
by (cases sv) auto

lemma var-to-dimacs-pos: \\
v \in \text{atoms} \quad (\map{\text{formula}} \text{var-to-dimacs f}) \Rightarrow 0 < v \\
by (induction f) auto
lemma map-is-disj: is-disj \( f \) \( \Rightarrow \) is-disj (map-formula \( F f \))
by (induction \( f \)) (auto elim: is-lit-plus.elims)

lemma map-is-cnf: is-cnf \( f \) \( \Rightarrow \) is-cnf (map-formula \( F f \))
by (induction \( f \)) (auto elim: is-lit-plus.elims simp: map-is-disj)

lemma planning-dimacs-complete:
valid-plan \( \pi s \) \( \Rightarrow \) length \( \pi s \) \( \leq \) \( h \) \( \Rightarrow \)
let cnf-formula = (map-formula var-to-dimacs
(\( \Phi \) (\( \forall \) (\( \varphi \) (prob-with-noop abs-prob)) \( h \))
)
in
\( \exists \) dimacs-M. dimacs-model dimacs-M (cnf-to-dimacs cnf-formula)

unfolding Let-def
by (fastforce simp: var-to-dimacs-pos
dest!: planning-by-cnf-dimacs-complete
intro: model-to-dimacs-model-sound-max-var map-is-cnf
is-cnf-encode-problem-with-operator-interference-exclusion
is-valid-problem-sas-plus-then-strips-transformation-too
noops-valid abs-prob-valid)

lemma planning-dimacs-sound:
let cnf-formula =
(map-formula var-to-dimacs
(\( \Phi \) (\( \forall \) (\( \varphi \) (prob-with-noop abs-prob)) \( h \))
)
in
dimacs-model dimacs-M (cnf-to-dimacs cnf-formula) \( \Rightarrow \)
valid-plan
(decode-abs-plan
(rem-noops
(map (\( \lambda \)op. \( \varphi \)\(^{-1}\) (prob-with-noop abs-prob) op)
(concat
(\( \Phi \)\(^{-1}\) (\( \varphi \) (prob-with-noop abs-prob)) ((dimacs-model-to-abs dimacs-M
(\( \lambda \). False)) o var-to-dimacs) \( h \))))))
by (fastforce simp: var-to-dimacs-pos Let-def
intro: planning-by-cnf-dimacs-sound model-to-dimacs-model-complete-max-var
map-is-cnf is-cnf-encode-problem-with-operator-interference-exclusion
is-valid-problem-sas-plus-then-strips-transformation-too abs-prob-valid
noops-valid)

end

13 Code Generation

We now generate SML code equivalent to the functions that encode a problem as a CNF formula and that decode the model of the given encodings into a plan.

lemma [code]:
\[
\text{dimacs-model } \text{ls } \text{cs} \equiv \text{list-all } (\lambda c. \text{list-ex } (\lambda l. \text{ListMem } l \ c) \ \text{ls}) \ \text{cs} \land \\
\text{distinct } (\text{map dimacs-lit-to-var } \text{ls}) \\
\text{unfolding dimacs-model-def} \text{ by } (\text{auto simp: list.pred-set ListMem-iff list-ex-iff}) \\
\text{definition} \text{ SASP-to-DIMACS } h \ \text{prob} \equiv \\
\text{cnf-to-dimacs} \\
(\text{map-formula} \\
(\text{cnf-to-dimacs.var-to-dimacs } (\text{Suc } h) (\text{Suc } (\text{length } (\text{ast-problem.astδ } \text{prob})))) \\
(\Phi \forall (\varphi \ (\text{prob-with-noop } (\text{ast-problem.abs-prob } \text{prob}))) \ h)) \\
\text{lemma planning-dimacs-complete-code:} \\
\text{[ast-problem.well-formed } \text{prob}; \\
\forall \pi \in \text{set } (\text{ast-problem.astδ } \text{prob}). \ \text{is-standard-operator'} \ \pi; \\
\text{ast-problem.valid-plan } \text{prob } \pi s; \\
\text{length } \pi s \leq h] \implies \\
\text{let cnf-formula } = (\text{SASP-to-DIMACS } h \ \text{prob}) \ \text{in} \\
\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula} \\
\text{unfolding SASP-to-DIMACS-def Let-def} \\
\text{apply}(\text{rule sat-solve-sasp.planning-dimacs-complete}[\text{unfolded Let-def}]) \\
\text{apply unfold-locales} \\
\text{by auto} \\
\text{definition} \text{ SASP-to-DIMACS'} h \ \text{prob} \equiv \text{SASP-to-DIMACS } h (\text{rem-implicit-pres-ops } \text{prob}) \\
\text{lemma planning-dimacs-complete-code':} \\
\text{[ast-problem.well-formed } \text{prob}; \\
(\forall \text{op. op } \in \text{set } (\text{ast-problem.astδ } \text{prob}) \implies \text{consistent-pres-op op}); \\
(\forall \text{op. op } \in \text{set } (\text{ast-problem.astδ } \text{prob}) \implies \text{is-standard-operator op}); \\
\text{ast-problem.valid-plan } \text{prob } \pi s; \\
\text{length } \pi s \leq h] \implies \\
\text{let cnf-formula } = (\text{SASP-to-DIMACS'} h \ \text{prob}) \ \text{in} \\
\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula} \\
\text{unfolding Let-def SASP-to-DIMACS'-def} \\
\text{by} (\text{auto simp add: rem-implicit-pres-ops-valid-plan}[\text{symmetric}] \ \text{wf-ast-problem-def} \\
\text{simp del: rem-implicit-pres-simps} \\
\text{intro!: rem-implicit-pres-is-standard-operator'} \\
\text{planning-dimacs-complete-code}[\text{unfolded Let-def}] \\
\text{rem-implicit-pres-ops-well-formed} \\
\text{dest!: rem-implicit-pres-ops-inδD}) \\
\text{A function that does the checks required by the completeness theorem above,} \\
\text{and returns appropriate error messages if any of the checks fail.} \\
\text{definition} \text{ encode } h \ \text{prob} \equiv \\
\text{if ast-problem.well-formed } \text{prob then} \\
\text{if } (\forall \text{op. op } \in \text{set } (\text{ast-problem.astδ } \text{prob}). \ \text{consistent-pres-op op}) \text{ then}
if (∀ op ∈ set (ast-problem.astδ prob). is-standard-operator op) then
  Inl (SASP-to-DIMACS' h prob)
else
  Inl (STR "Error: Conditional effects!")
else
  Inr (STR "Error: Preconditions inconsistent")
else
  Inr (STR "Error: Problem malformed!")

lemma encode-sound:
\[
\begin{align*}
&\text{[ast-problem.valid-plan prob πs; length } πs \leq h; \\
&\quad \text{encode } h \text{ prob } = \text{Inl cnf-formula]} \\
&\quad \exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}\end{align*}
\]

unfolding encode-def
by (auto split: if-splits simp: list.pred-set
    intro: planning-dimacs-complete-code [unfolded Let-def])

lemma encode-complete:
\[
\begin{align*}
&\text{encode } h \text{ prob } = \text{Inr err } \implies \\
&\neg (\text{ast-problem.well-formed prob } \land (\forall \text{op } \in \text{set (ast-problem.astδ prob). consistent-pres-op op}) \\
\land \\
&\quad (\forall \text{op } \in \text{set (ast-problem.astδ prob). is-standard-operator op}))\end{align*}
\]

unfolding encode-def
by (auto split: if-splits simp: list.pred-set
    intro: planning-dimacs-complete-code [unfolded Let-def])

definition match-pre where
\[
\text{match-pre } \equiv \lambda (x, v) \text{ s. s x } = \text{Some v}
\]

definition match-pres where
\[
\text{match-pres } \equiv \forall \text{pre } \in \text{set pres. match-pre pre s}
\]

lemma match-pres-distinct:
\[
\begin{align*}
&\text{distinct } (\text{map } \text{fst } \text{pres}) \implies \text{match-pres } \text{pres } s \iff \text{Map.m-of } \text{pres } \subseteq m \text{ s}
\end{align*}
\]

unfolding match-pres-def match-pre-def
using map-le-def map-of-SomeD
apply (auto split: prod.splits)
apply fastforce
using domI map-of-is-SomeI
by smt

fun tree-map-of where
\[
\begin{align*}
&\text{tree-map-of updatea } T [] = T \\
&\quad | \text{tree-map-of updatea } T ((v, a) \# m) = \text{updatea } v \text{ a } (\text{tree-map-of updatea } T m)\end{align*}
\]

category Map
begin

abbreviation tree-map-of' ≡ tree-map-of update

349
lemma tree-map-of-invar: invar $T \Rightarrow$ invar (tree-map-of' $T$ pres)
    by (induction pres) (auto simp add: invar-update)

lemma tree-map-of-works: lookup (tree-map-of' empty pres) $x = \text{map-of pres } x$
    by (induction pres) (auto simp: map-empty map-update[OF tree-map-of-invar[OF invar-empty]])

lemma tree-map-of-dom: dom (lookup (tree-map-of' empty pres)) = dom (map-of pres)
    by (induction pres) (auto simp: map-empty map-update[OF tree-map-of-invar[OF invar-empty]] tree-map-of-works)
end

lemma distinct-if-sorted: sorted $xs \Rightarrow$ distinct $xs$
    by (induction $xs$ rule: induct-list012) auto

context Map-by-Ordered begin

lemma tree-map-of-distinct:
    distinct (map fst (inorder (tree-map-of' empty pres)))
    apply (induction pres)
    apply (clarsimp simp: inorder-empty)
    using distinct-if-sorted invar-def invar-empty invar-update tree-map-of-invar by blast
end

lemma set-tree-intorder: set-tree $t = \text{set } (\text{inorder } t)$
    by (induction $t$) auto

lemma map-of-eq:
    map-of $xs = \text{Map.map-of } xs$
    by (induction $xs$) (auto simp: map-of-simps split: option.split)

lemma lookup-someD: lookup $T x = \text{Some } y \Rightarrow \exists p. p \in \text{set } (\text{inorder } T) \land p = (x, y)$
    by (induction $T$) (auto split: if-splits)

lemma map-of-lookup: sorted1 (inorder $T$) $\Rightarrow \text{Map.map-of } (\text{inorder } T) = \text{lookup } T$
    apply (induction $T$)
    apply (auto split: prod.splits intro!: map-le-antisym simp: lookup-map-of map-add-Some-iff map-of-None2 sorted-wrt-append)
    using lookup-someD
    by (force simp: map-of-eq map-add-def map-le-def split: option.splits)+

lemma map-le-cong: ($\forall x. m1 x = m2 x) \Rightarrow m1 \subseteq_m s \iff m2 \subseteq_m s$

end
by presburger

lemma match-pres-submap:
  match-pres (inorder (M.tree-map-of' empty pres)) s ←→ Map.map-of pres ⊆ m s
using match-pres-distinct[OF M.tree-map-of-distinct]
by (smt M.invar-def M.invar-empty M.tree-map-of-invar M.tree-map-of-works
    map-le-cong map-of-eq map-of-lookup)

lemma [code]:
  SAS-Plus-Representation.is-operator-applicable-in s op ←→
  match-pres (inorder (M.tree-map-of' empty
    (SAS-Plus-Representation.precondition-of op))) s
by (simp add: match-pres-submap SAS-Plus-Representation.is-operator-applicable-in-def)

definition decode-DIMACS-model dimacs-M h prob ≡
  (ast-problem.decode-abs-plan prob
    (rem-noops
      (map (λop. φO⁻¹ (prob-with-noop (ast-problem.abs-prob prob)) op)
        (concat
          (Φ⁻¹ (φ (prob-with-noop (ast-problem.abs-prob prob))))
            ((dimacs-model-to-abs dimacs-M (λ. False)) o
              (cnf-to-dimacs.var-to-dimacs (Suc h))
                (Suc (length (ast-problem.astδ prob))))))
        h))))

lemma planning-dimacs-sound-code:
[ast-problem.well-formed prob;
 ∀ π∈set (ast-problem.astδ prob). is-standard-operator' π] =⇒
let
  cnf-formula = (SASP-to-DIMACS h prob);
  decoded-plan = decode-DIMACS-model dimacs-M h prob
in
  (dimacs-model dimacs-M cnf-formula =⇒ ast-problem.valid-plan prob decoded-plan)
unfolding SASP-to-DIMACS-def decode-DIMACS-model-def Let-def
apply(rule impl sat-solve-sasp.planning-dimacs-sound[unfolded Let-def])+
apply unfold-locales
by auto

definition decode-DIMACS-model' dimacs-M h prob ≡
  decode-DIMACS-model dimacs-M h (rem-implicit-pres-ops prob)

lemma planning-dimacs-sound-code' :
[ast-problem.well-formed prob;
  (∀ op∈set (ast-problem.astδ prob) =⇒ consistent-pres-op op);
  ∀ π∈set (ast-problem.astδ prob). is-standard-operator π] =⇒
let
  cnf-formula = (SASP-to-DIMACS' h prob);
\[
\text{decoded-plan} = \text{decode-DIMACS-model'} \text{ dimacs-M h prob}
\]
in
\((\text{dimacs-model dimacs-M cnf-formula} \rightarrow \text{ast-problem.valid-plan prob decoded-plan})\)

\textbf{unfolding} \text{SASP-to-DIMACS'-def decode-DIMACS-model'-def}

\textbf{apply}(\text{subst rem-implicit-pres-ops-valid-plan}[\text{symmetric}])

\textbf{by}(\text{fastforce simp only: rem-implicit-pres-ops-valid-plan wf-ast-problem-def}

\text{intro!: rem-implicit-pres-is-standard-operator'*}

\text{rev-ifD2[OF - rem-implicit-pres-ops-valid-plan]}

\text{planning-dimacs-sound-code wf-ast-problem.intro}

\text{dest!: rem-implicit-pres-ops-in}\delta D)+

Checking if the model satisfies the formula takes the longest time in the decoding function. We reimplement that part using red black trees, which makes it 10 times faster, on average!

\textbf{fun list-to-rbt :: int list \Rightarrow int rbt where}

\text{list-to-rbt} [] = Leaf
| \text{list-to-rbt} (x#xs) = insert-rbt x (list-to-rbt xs)

\textbf{lemma inv-list-to-rbt: invc (list-to-rbt xs) \land invh (list-to-rbt xs)}
\textbf{by (induction xs)} (auto simp: \text{rbt-def RBT}.inv-insert)

\textbf{lemma Tree2-list-to-rbt: Tree2.bst (list-to-rbt xs)}
\textbf{by (induction xs)} (auto simp: \text{RBT}.bst-insert)

\textbf{lemma set-list-to-rbt: Tree2.set-tree (list-to-rbt xs) = set xs}
\textbf{by (induction xs)} (simp add: \text{RBT}.set-tree-insert Tree2-list-to-rbt)+

The following

\textbf{lemma dimacs-model-code[\text{code}]:}
\text{dimacs-model ls cs \leftarrow}
| (let tls = list-to-rbt ls in
| (\forall c \in \text{set cs}. size (inter-rbt (tls) (list-to-rbt c)) \neq 0) \land
| \text{distinct (map dimacs-lit-to-var ls))}

\textbf{using RBT.set-tree-inter[OF Tree2-list-to-rbt Tree2-list-to-rbt]}
\textbf{apply (auto simp: dimacs-model-def Let-def set-list-to-rbt inter-rbt-def)}
\textbf{apply (metis IntI RBT.set-empty empty-iff)}
\textbf{by (metis Tree2.eq-set-tree-empty disjoint-iff-not-equal)}

\textbf{definition decode M h prog \equiv}
| if ast-problem.well-formed prob then
| if (\forall op\in\text{set (ast-problem.ast op prob)}. consistent-pres-op op) then
| if (\forall op\in\text{set (ast-problem.ast op prob). is-standard-operator op}) then
| if (\text{dimacs-model M (SASP-to-DIMACS' h prog)}) then
| \text{Inl (decode-DIMACS-model' M h prob)}
| else \text{Inr (STR "Error: Model does not solve the problem!")}
| else
| \text{Inr (STR "Error: Conditional effects!")}

352
else
  Inr (STR "Error: Preconditions inconsistent")
else
  Inr (STR "Error: Problem malformed!")

lemma decode-sound:
decode M h prob = Inl plan ⟷
  ast-problem.valid-plan prob plan
unfolding decode-def
apply (auto split: if-splits simp: list.pred-set)
using planning-dimacs-sound-code'
by auto

lemma decode-complete:
decode M h prob = Inr err ⟷
  ¬ (ast-problem.well-formed prob ∧
    (∀ op ∈ set (ast-problem.ast δ prob). consistent-pres-op op) ∧
    (∀ π ∈ set (ast-problem.ast δ prob). is-standard-operator π) ∧
    dimacs-model M (SASP-to-DIMACS' h prob))
unfolding decode-def
by (auto split: if-splits simp: list.pred-set)

lemma [code]:
ListMem x' [] = False
ListMem x' (x#xs) = (x' = x ∨ ListMem x' xs)
by (simp add: ListMem-iff)+

lemmas [code] = SASP-to-DIMACS-def ast-problem.abs-prob-def
  ast-problem.abs-ast-initial-state-def ast-problem.abs-range-map-def
  ast-problem.abs-ast-goal-def cnf-to-dimacs.var-to-dimacs.simps

definition nat-opt-of-integer :: integer ⇒ nat option where
  nat-opt-of-integer i = (if (i ≥ 0) then Some (nat-of-integer i) else None)

definition max-var :: int list ⇒ int where
  max-var xs ≡ fold (λ(x::int) (y::int). if abs x ≥ abs y then (abs x) else y) xs
  (0::int)

export-code encode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode
  String.implode max-var concat char-of-nat Int.nat integer-of-int length
int-of-integer
  in SML module-name exported file code/generated/SASP-to-DIMACS.sml

export-code decode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode
  String.implode max-var concat char-of-nat Int.nat integer-of-int length
References

