Verified SAT-Based AI Planning

Mohammad Abdulaziz and Friedrich Kurz

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.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>The Basic SATPlan Encoding</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>7.1 Encoding Function Definitions</td>
<td>177</td>
</tr>
<tr>
<td></td>
<td>7.2 Decoding Function Definitions</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>7.3 Soundness of the Basic SATPlan Algorithm</td>
<td>213</td>
</tr>
<tr>
<td></td>
<td>7.4 Completeness</td>
<td>250</td>
</tr>
<tr>
<td>8</td>
<td>Serializable SATPlan Encodings</td>
<td>291</td>
</tr>
<tr>
<td></td>
<td>8.1 Soundness</td>
<td>298</td>
</tr>
<tr>
<td></td>
<td>8.2 Completeness</td>
<td>301</td>
</tr>
<tr>
<td>9</td>
<td>SAT-Solving of SAS+ Problems</td>
<td>308</td>
</tr>
<tr>
<td>10</td>
<td>Adding Noop actions to the SAS+ problem</td>
<td>310</td>
</tr>
<tr>
<td>11</td>
<td>Proving Equivalence of SAS+ representation and Fast-Downward’s</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>Multi-Valued Problem Representation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11.1 Translating Fast-Downward’s representation to SAS+</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>11.2 Translating SAS+ representation to Fast-Downward’s</td>
<td>328</td>
</tr>
<tr>
<td></td>
<td>11.3 SAT encoding works for Fast-Downward’s representation</td>
<td>334</td>
</tr>
<tr>
<td>12</td>
<td>DIMACS-like semantics for CNF formulae</td>
<td>334</td>
</tr>
<tr>
<td></td>
<td>12.1 Going from Formulae to DIMACS-like CNF</td>
<td>340</td>
</tr>
<tr>
<td>13</td>
<td>Code Generation</td>
<td>347</td>
</tr>
</tbody>
</table>
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.

```plaintext
type-synonym (\text{variable}, \text{domain}) state = \text{variable} \mapsto \text{domain}
```

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

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

end

theory STRIPS-Representation
  imports State-Variable-Representation
begin

2 STRIPS Representation

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

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.

```plaintext
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.

\footnote{For 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 pre add delete ≡ 
  precondition-of = pre
  , add-effects-of = add
  , delete-effects-of = delete []

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

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

2

definition execute-operator
:: 'variable strips-state
g⇒ 'variable strips-operator
g⇒ '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

2Function 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 $s \Rightarrow \text{op}$ of the given state and operator.

\begin{verbatim}
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 ≡ ((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 π :: 'a strips-problem
assumes is-valid-problem-strips π
shows dom (π) = set (π)
proof –
{ 
  let \( \pi I \) = strips-problem.initial-of \( \Pi \) 
  let \( \pi vs \) = strips-problem.variables-of \( \Pi \) 
  fix \( v \) 
  have \( \pi I \ v \neq \text{None} \longleftrightarrow \text{ListMem} \ v \pi vs \)
  using assms(1)
  unfolding is-valid-problem-strips-def
  by meson
  hence \( v \in \text{dom} \ (\pi I) \longleftrightarrow v \in \pi vs \)
  using ListMem-iff
  by fast
}
thus \( \pi \text{thesis} \)
by auto 
qed

lemma is-valid-problem-dom-of-goal-state-is:
fixes \( \Pi \):: 'a strips-problem 
assumes is-valid-problem-strips \( \Pi \)
shows \( \text{dom} \ ((\Pi)_G) \subseteq \text{set} \ ((\Pi)_V) \)
proof –
  let \( \pi vs = \text{strips-problem.variables-of} \ (\Pi) \)
  let \( \pi G = \text{strips-problem.goal-of} \ (\Pi) \)
  have \( \pi nb: \forall v. \pi G \ v \neq \text{None} \longrightarrow \text{ListMem} \ v \pi vs \)
  using assms(1)
  unfolding is-valid-problem-strips-def
  by meson
  { 
  fix \( v \)
  assume \( v \in \text{dom} \ (\pi G) \)
  then have \( (\pi G) \ v \neq \text{None} \)
  by blast
  hence \( v \in \text{set} \ (\pi vs) \)
  using \( \pi nb \)
  unfolding ListMem-iff
  by blast
  }
  thus \( \pi \text{thesis} \)
  by auto
qed

lemma is-valid-problem-strips-operator-variable-sets:
fixes \( \Pi \):: 'a strips-problem 
assumes is-valid-problem-strips \( \Pi \) 
and \( \pi op \in \text{set} \ ((\Pi)_O) \)
shows \( \text{set} \ ((\pi \text{precondition-of} \ op)) \subseteq \text{set} \ ((\Pi)_V) \)
and \( \text{set} \ ((\pi \text{add-effects-of} \ op)) \subseteq \text{set} \ ((\Pi)_V) \)
and \( \text{set} \ ((\pi \text{delete-effects-of} \ op)) \subseteq \text{set} \ ((\Pi)_V) \)
and \( \text{disjnt} \ (\text{set} \ ((\pi \text{add-effects-of} \ op)) \ (\text{set} \ ((\pi \text{delete-effects-of} \ op))) \)
proof

let ?ops = strips-problem.operators-of II
and ?vs = strips-problem.variables-of II
have list-all (is-valid-operator-strips II) ?ops
  using assms(1)
unfolding is-valid-problem-strips-def
by meson
moreover have ∀ v ∈ set (precondition-of op). v ∈ set ((II)V)
  and ∀ v ∈ set (add-effects-of op). v ∈ set ((II)V)
  and ∀ v ∈ set (delete-effects-of op). v ∈ set ((II)V)
  and ∀ v ∈ set (delete-effects-of op). v ∉ set (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 set (precondition-of op) ⊆ set ((II)V)
  and set (add-effects-of op) ⊆ set ((III)V)
  and set (delete-effects-of op) ⊆ set ((III)V)
  and disjoint (set (add-effects-of op)) (set (delete-effects-of op))
unfolding disjoint-def
by fast+
qed

lemma effect-to-assignments-i:
assumes as = effect-to-assignments op
shows as = (map (λ v. (v, True)) (add-effects-of op))
@ map (λ v. (v, False)) (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 @ as2
  and as1 = map (λ v. (v, True)) (add-effects-of op)
  and as2 = map (λ v. (v, False)) (delete-effects-of op)
by (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 $v \in \text{set} \ (\text{add-effects-of } \text{op})$
and $\text{as} = \text{effect-to-assignments } \text{op}$
obtains $a$ where $a \in \text{set} \ a = (v, \text{True})$

proof

let $\ ?\text{add-assignments} = (\lambda v. (v, \text{True})) \ ' \ \text{set} \ (\text{add-effects-of } \text{op})$
let $\ ?\text{delete-assignments} = (\lambda v. (v, \text{False})) \ ' \ \text{set} \ (\text{delete-effects-of } \text{op})$

obtain $a_1, a_2$
where $a_1$: $a = a_1 \ominus a_2$
and $a_2$: $a_1 = \text{map} \ (\lambda v. (v, \text{True})) \ (\text{add-effects-of } \text{op})$
and $a_3$: $a_2 = \text{map} \ (\lambda v. (v, \text{False})) \ (\text{delete-effects-of } \text{op})$
using $\text{assms}(2)$ $\text{effect-to-assignments-ii}$
by $\text{blast}$

then have $b$: $\text{set} \ as$

= $\ ?\text{add-assignments} \cup \ ?\text{delete-assignments}$
by $\text{auto}$

— 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 $\text{blast}$
moreover have $\{(v, \text{True})\} \subseteq \ ?\text{add-assignments}$
using $\text{assms}(1)$ $a_2$
by $\text{blast}$
ultimately have $\exists a. \ a \in \text{set} \ as \land a = (v, \text{True})$
by $\text{blast}$
}

then show $\ ?\text{thesis}$
using that
by $\text{blast}$

qed

lemma $\text{effect-to-assignments-iii-b}$:

— NOTE This proof is symmetrical to the one above.

fixes $v$
assumes $v \in \text{set} \ (\text{delete-effects-of } \text{op})$
and $\text{as} = \text{effect-to-assignments } \text{op}$
obtains $a$ where $a \in \text{set} \ a = (v, \text{False})$

proof

let $\ ?\text{add-assignments} = (\lambda v. (v, \text{True})) \ ' \ \text{set} \ (\text{add-effects-of } \text{op})$
let $\ ?\text{delete-assignments} = (\lambda v. (v, \text{False})) \ ' \ \text{set} \ (\text{delete-effects-of } \text{op})$

obtain $a_1, a_2$
where $a_1$: $a = a_1 \ominus a_2$
and $a_2$: $a_1 = \text{map} \ (\lambda v. (v, \text{True})) \ (\text{add-effects-of } \text{op})$
and $a_3$: $a_2 = \text{map} \ (\lambda v. (v, \text{False})) \ (\text{delete-effects-of } \text{op})$
using $\text{assms}(2)$ $\text{effect-to-assignments-ii}$
by $\text{blast}$

then have $b$: $\text{set} \ as$

= $\ ?\text{add-assignments} \cup \ ?\text{delete-assignments}$

8
by auto
— NOTE The existence of an assignment as proposed can be shown by the following sequence of set inclusions.

\[
\{\text{from } b \text{ have } ?\text{delete-assignments} \subseteq \text{set as} \\
\quad \text{by blast} \\
\quad \text{moreover have } \{(v, False)\} \subseteq ?\text{delete-assignments} \\
\quad \text{using } \text{assms}(t) \ a2 \\
\quad \text{by blast} \\
\quad \text{ultimately have } \exists a. \ a \in \text{set as} \land a = (v, False) \\
\quad \text{by blast} \}
\]

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

lemma effect--strips-i:
fixes op 
assumes e = effect--strips op 
obtains es1 es2 
where \( e = (es_1 @ es_2) \) 
and es1 = map (\( \lambda v. (v, True) \)) (add-effects-of op) 
and es2 = map (\( \lambda v. (v, False) \)) (delete-effects-of op) 
proof 
- obtain es1 es2 where a: e = (es1 @ es2) 
and b: es1 = map (\( \lambda v. (v, True) \)) (add-effects-of op) 
and c: es2 = map (\( \lambda v. (v, False) \)) (delete-effects-of op) 
using \text{assms}(t) 
unfolding effect--strips-def 
by blast 
then show ?thesis 
using that 
by force 
qed

lemma effect--strips-ii:
fixes op 
assumes e = ConjunctiveEffect (es1 @ es2) 
and es1 = map (\( \lambda v. (v, True) \)) (add-effects-of op) 
and es2 = map (\( \lambda v. (v, False) \)) (delete-effects-of op) 
sows \( \forall v \in \text{set (add-effects-of op)}. \ (\exists e' \in \text{set es1}. \ e' = (v, True)) \) 
and \( \forall v \in \text{set (delete-effects-of op)}. \ (\exists e' \in \text{set es2}. \ e' = (v, False)) \) 
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: \( v \in \text{set (add-effects-of op)} \)
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)
sshows dom m = set vs
proof —
  let ?vs' = map (λv. (v, d)) vs
  have dom m = fst ′ set ?vs'
    using assms(1) dom-map-of-conve-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 >>= op)
    shows \( \forall v. v \in \text{set} (\text{add-effects-of op}) \implies s' v = \text{Some True} \)
    proof
      fix v
      assume a: v \in \text{set} (\text{add-effects-of op})
      let ?as = \text{effect-to-assignments op}
      obtain as1 as2 where b: ?as = as1 @ as2
        and c: as1 = \text{map} (\lambda v. (v, True)) (\text{add-effects-of op})
        and as2 = \text{map} (\lambda v. (v, False)) (\text{delete-effects-of op})
        using \text{effect-to-assignments-ii}
        by blast
      have d: \text{map-of} ?as = \text{map-of as2} ++ \text{map-of as1}
        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 dom (\text{map-of as1}) = \text{set} (\text{add-effects-of op})
        using c \text{map-of-constant-assignments-dom}
        by metis
      then have v \in dom (\text{map-of as1})
        using a
        by blast
      then have \text{map-of} ?as v = \text{map-of as1} v
        using d
        by force
      \} moreover \{
      let ?f = \lambda _. \text{True}
      from c have \text{map-of as1} = (\text{Some o ?f}) \circ (\text{set} (\text{add-effects-of op}))
        using \text{map-of-map-restrict}
        by fast
      then have \text{map-of as1} v = \text{Some True}
        using a
        by auto
      \}
      moreover have s' = s ++ \text{map-of as2} ++ \text{map-of as1}
        using assms(1)
        unfolding \text{execute-operator-def}
        using b
        by simp
      ultimately show s' v = \text{Some True}
        by simp
    qed
lemma effect--strips-iii-b:
assumes $s' = (s \triangleright\triangleright op)$
shows $\forall v. \forall v \in \text{set (delete-effects-of op)} \land v \notin \text{set (add-effects-of op)} \implies s' v = \text{Some False}$
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 $as = \text{effect-to-assignments op}$
obtain $as_1 as_2$ where $b: ?as = as_1 \odot as_2$
and $c: as_1 = \text{map } (\lambda v. (v, \text{True}))$ (add-effects-of op)
and $d: as_2 = \text{map } (\lambda v. (v, \text{False}))$ (delete-effects-of op)
using effect-to-assignments-ii
by blast
have $c: \text{map-of } ?as = \text{map-of } as_2 ++ \text{map-of } as_1$
using $b$ Map.map-of-append
by auto
{
  have $\text{dom } (\text{map-of } as_1) = \text{set (add-effects-of op)}$
    using $c$ map-of-constant-assignments-dom
    by metis
  then have $v \notin \text{dom } (\text{map-of } as_1)$
    using $a1$
    by blast
}
{
  let $?vs = \text{delete-effects-of op}$
  have $?vs \neq []$
    using $a2$
    by force
  then have $\text{dom } (\text{map-of } as_2) = \text{set } ?vs$
    using $d$ map-of-constant-assignments-dom
    by metis
}
{
  have $s' = s ++ \text{map-of } as_2 ++ \text{map-of } as_1$
    using assms(1)
    unfolding execute-operator-def
    using $b$
    by simp
  thm $f$ map-add-dom-app-simps(3)[OF $f$, of $s ++ \text{map-of } as_2$]
  moreover have $s' v = (s ++ \text{map-of } as_2) v$
    using calculation map-add-dom-app-simps(3)[OF $f$, of $s ++ \text{map-of } as_2$]
    by blast
  moreover have $v \in \text{dom } (\text{map-of } as_2)$
    using $a2$ $g$
    by argo
  ultimately have $s' v = \text{map-of } as_2 v$
    by fastforce
}
moreover 

\{ 
  \text{let } \delta f = \lambda \cdot \text{False} \\
  \text{from } d \text{ have } \text{map-of as}_2 = (\text{Some } \delta f) \circ (\text{set (delete-effects-of op)}) \\
  \quad \text{using } \text{map-of-map-restrict} \\
  \quad \text{by } \text{fast} \\
  \text{then have } \text{map-of as}_2 v = \text{Some False} \\
  \quad \text{using } a_2 \\
  \quad \text{by } \text{force} 
\} \\
\text{ultimately show } s' v = \text{Some False} \\
\quad \text{by } \text{argo} \\
\text{qed}

\textbf{lemma } \text{effect-strips-iii-c:} \\
\textbf{assumes } s' = (s \gg op) \\
\textbf{shows } \bigwedge v. v \notin \text{set (add-effects-of op)} \land v \notin \text{set (delete-effects-of op)} \rightarrow s' v = s v \\
\textbf{proof (auto)} \\
\textbf{fix } v \\
\textbf{assume } a_1: v \notin \text{set (add-effects-of op)} \textbf{ and } a_2: v \notin \text{set (delete-effects-of op)} \\
\textbf{let } \delta as = \text{effect-to-assignments op} \\
\textbf{obtain } \text{as}_1 \text{ as}_2 \textbf{ where } b: \delta as = \text{as}_1 \oplus \text{as}_2 \\
\quad \text{and } c: \text{as}_1 = \text{map } (\lambda v. (v, \text{True})) (\text{add-effects-of op}) \\
\quad \text{and } d: \text{as}_2 = \text{map } (\lambda v. (v, \text{False})) (\text{delete-effects-of op}) \\
\quad \text{using } \text{effect-to-assignments-ii} \\
\quad \text{by } \text{blast} \\
\textbf{have } c: \text{map-of } \delta as = \text{map-of as}_2 ++ \text{map-of as}_1 \\
\quad \text{using } b \text{ } \text{Map.map-of-append} \\
\quad \text{by } \text{auto} \\
\} \\
\text{moreover } \{ \\
\textbf{have } \text{dom (map-of as}_1) = \text{set (add-effects-of op)} \\
\quad \text{using } c \text{ map-of-constant-assignments-dom} \\
\quad \text{by } \text{metis} \\
\textbf{then have } v \notin \text{dom (map-of as}_1) \\
\quad \text{using } a_1 \\
\quad \text{by } \text{blast} 
\} \\
\text{ultimately show } s' v = s v \\
\quad \text{using } \text{assms(1)}
unfold \textit{execute-operator-def}

by (simp add: b map-add-dom-app-simps(3))

\textbf{qed}

The following theorem combines three preceding sublemmas which show that the following properties hold for the successor state \( s' = \text{execute-operator op s} \) obtained by executing an operator \( op \) in a state \( s \) :\footnote{Lemmas effect__strips_iii_a, effect__strips_iii_b, and effect__strips_iii_c (not shown).}

- 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.

\textbf{theorem} \ operator-effect--strips:
\textbf{assumes} \( s' = (s \triangleright op) \)
\textbf{shows} \(
\forall v. \ \ v \in \text{set (add-effects-of op)} \\
\quad \quad \Rightarrow \ s' \ v = \text{Some True}
\)
and \(
\forall v. \ \ v \not\in \text{set (add-effects-of op)} \land v \in \text{set (delete-effects-of op)} \\
\quad \quad \Rightarrow \ s' \ v = \text{Some False}
\)
and \(
\forall v. \ \ v \not\in \text{set (add-effects-of op)} \land v \not\in \text{set (delete-effects-of op)} \\
\quad \quad \Rightarrow \ s' \ v = s \ v
\)
\textbf{proof} (auto)

\textbf{show} \(\forall v. \ \ v \in \text{set (add-effects-of op)} \Rightarrow \ s' \ v = \text{Some True} \)
\textbf{using} \text{assms} \text{effect--strips-iii-a}
\textbf{by} \text{fast}

\textbf{next}

\textbf{show} \(\forall v. \ \ v \not\in \text{set (add-effects-of op)} \Rightarrow \ v \in \text{set (delete-effects-of op)} \Rightarrow \ s' \ v = \text{Some False} \)
\textbf{using} \text{assms} \text{effect--strips-iii-b}
\textbf{by} \text{fast}

\textbf{next}

\textbf{show} \(\forall v. \ \ v \not\in \text{set (add-effects-of op)} \Rightarrow \ s' \ v = s \ v \)
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 op
1 op
2 ≡ let
add
1 = add-effects-of op
1
; add
2 = add-effects-of op
2
; del
1 = delete-effects-of op
1
; del
2 = delete-effects-of op
2
in ¬list-ex (λv. list-ex ((=) v) del
2) add
1 ∧ ¬list-ex (λv. list-ex ((=) v) add
2) del
1

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-plan
⇒ 'variable strips-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 op
1 op
2
≡ list-ex (λv. list-ex ((=) v) (delete-effects-of op
1)) (precondition-of op
2)
∨ list-ex (λv. list-ex ((=) v) (precondition-of op
1)) (delete-effects-of op
2)

primrec are-all-operators-non-interfering
:: 'variable strips-operator list ⇒ bool
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 op₁ ∈ set ops
and op₂ ∈ set ops
shows are-operator-effects-consistent op₁ op₂
= (set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {})
∧ set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {})
proof −
have (∀v. list-ex ((=) v) (delete-effects-of op₂) (add-effects-of op₁))
= (set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {})
  using list-ex-intersection[of delete-effects-of op₂ add-effects-of op₁]
by meson
moreover have (∀v. list-ex ((=) v) (add-effects-of op₂) (delete-effects-of op₁))
  using list-ex-intersection[of add-effects-of op₂ delete-effects-of op₁]
by meson
ultimately show are-operator-effects-consistent op₁ op₂
= (set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {})
∧ (set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {}))
unfolding are-operator-effects-consistent-def
by presburger
qed

lemma are-all-operator-effects-consistent-set:
are-all-operator-effects-consistent ops
←→ (∀op₁ ∈ set ops, ∀op₂ ∈ set ops.
  (set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {})
∧ (set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {}))
proof −
{ fix op₁ op₂
  assume op₁ ∈ set ops and op₂ ∈ set ops
  hence are-operator-effects-consistent op₁ op₂
  = (set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {})
∧ (set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {}))
  using are-operator-effects-consistent-set[of op₁ ops op₂]
  by fast
  qed
thus \( \text{thesis} \)

unfolding \text{are-all-operator-effects-consistent-def} \ list-all-iff

by force

qed


lemma \text{are-all-effects-consistent-tail}:

assumes \text{are-all-operator-effects-consistent} \ (\text{op} \# \text{ops})

shows \text{are-all-operator-effects-consistent} \ \text{ops}

proof

from \text{assms}

have \( a \): \ list-all \ (\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}')\) \n\ (\text{Cons} \ \text{op} \ \text{ops}) \ (\text{Cons} \ \text{op} \ \text{ops})

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

by blast

then have b-1: \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}) \ (\text{op} \# \ \text{ops})

and b-2: \ list-all \ (\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}') \ (\text{op} \# \ \text{ops})) \ \text{ops}

by force+

then have \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}) \ \text{ops}

by simp

moreover

\{ 

\{ fix \ z 

assume \ z \in \ \text{set} \ (\text{Cons} \ \text{op} \ \text{ops})

and \ list-all \ (\text{are-operator-effects-consistent} \ \text{z}) \ (\text{op} \# \ \text{ops})

then have \ list-all \ (\text{are-operator-effects-consistent} \ \text{z}) \ \text{ops}

by auto 

\}

then have \ list-all \ (\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}') \ \text{ops}) \ \text{ops}

using \ list.\text{pred-mono-strong}\ of

(\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}') \ (\text{op} \# \ \text{ops}))

\ \text{Cons} \ \text{op} \ \text{ops} \ (\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}') \ \text{ops})

] \ a

by fastforce

\}

ultimately have \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}) \ \text{ops}

\ \land \ list-all \ (\lambda \text{op}'. \ list-all \ (\text{are-operator-effects-consistent} \ \text{op}') \ \text{ops}) \ \text{ops}

by blast

then show \ \text{thesis}

using \ text{are-all-operator-effects-consistent-def}

by fast

qed


lemma \text{are-all-operators-non-interfering-tail}:

assumes \text{are-all-operators-non-interfering} \ (\text{op} \# \ \text{ops})

shows \text{are-all-operators-non-interfering} \ \text{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 op1 op2
and op1 \neq op2
shows op1 \notin set ops \vee op2 \notin set ops
using assms
proof (induction ops)
case (Cons op ops)
thm Cons.IH[OF - Cons.prems(2, 3)]
have nb1: \forall op' \in set ops. \neg 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) op \neq op1 \land op \neq op2
by blast
thus \exists case
proof (cases)
case A
{|
assume op2 \in set (op \# ops)
then have op2 \in set ops
using Cons.prems(3) A
by force
then have \neg are-operators-interfering op1 op2
using nb1 A
by fastforce
hence False
using Cons.prems(2)..
}
thus \exists thesis
by blast
next
case B
assume \( op_1 \in \text{set} \ (op \ # \ ops) \)
then have \( op_1 \in \text{set} \ ops \)
using \( \text{Cons}.\text{prems}(3) \ B \)
by force
then have \( \neg \text{are-operators-interfering} \ op_1 \ op_2 \)
using \( nb_1 \ B \ \text{are-operators-interfering-symmetric} \)
by blast
hence \( \text{False} \)
using \( \text{Cons}.\text{prems}(2) \).

thus \( \text{thesis} \)
by blast
next
case \( C \)
thus \( \text{thesis} \)
using \( \text{Cons}.\text{IH}[\text{OF} \ nb_2 \ \text{Cons}.\text{prems}(2, 3)] \)
by force
qed

lemma \( \text{execute-parallel-plan-precondition-cons-i} \):
fixes \( s :: (\text{variable, bool}) \ \text{state} \)
assumes \( \neg \text{are-operators-interfering} \ op \ op' \)
and \( \text{is-operator-applicable-in} \ s \ op \)
and \( \text{is-operator-applicable-in} \ s \ op' \)
shows \( \text{is-operator-applicable-in} \ (s ++ \text{map-of} \ (\text{effect-to-assignments} \ op)) \ op' \)
proof
let \( ?s' = s ++ \text{map-of} \ (\text{effect-to-assignments} \ op) \)
— TODO slightly hackish to exploit the definition of \( \text{execute-operator} \), but we otherwise have to rewrite theorem \( \text{operator-effect--strips} \) (which is a todo as of now).

\[
\begin{align*}
\{ \\
\text{have a: } ?s' = s \gg op \\
\text{by (simp add: execute-operator-def)} \\
\text{then have } \forall v. v \in \text{set} \ (\text{add-effects-of} \ op) \Longrightarrow ?s' \ v = \text{Some True} \\
\text{and } \forall v. v \notin \text{set} \ (\text{add-effects-of} \ op) \land v \in \text{set} \ (\text{delete-effects-of} \ op) \Longrightarrow ?s' \ v = \text{Some False} \\
\text{and } \forall v. v \notin \text{set} \ (\text{add-effects-of} \ op) \land v \notin \text{set} \ (\text{delete-effects-of} \ op) \Longrightarrow ?s' \ v = s \ v \\
\text{using operator-effect--strips} \\
\text{by metis+} \\
\} \\
\text{note a = this} \\
— \text{TODO refactor lemma not-have-interference-set.}
\{ \\
\text{fix } v \\
\text{assume } \alpha : v \in \text{set} \ (\text{precondition-of} \ op')
\}
\]
\[ \{ \\
\text{fix } v \\
\text{have } \neg\text{list-ex } ((=) v) \text{ (delete-effects-of op)} \\
= \text{list-all } (\lambda v'. \neg v = v') \text{ (delete-effects-of op)} \\
\text{using not-list-ex-equals-list-all-not[}
where P=(=) v \text{ and } xs=\text{delete-effects-of op]}
\text{by blast}
\} \] morever \{ \\
\text{from assms(1)} \\
\text{have } \neg\text{list-ex } (\lambda v. \text{list-ex } ((=) v) \text{ (delete-effects-of op)}) \text{ (precondition-of op')}
\text{unfolding are-operators-interfering-def}
\text{by blast}
\text{then have } \text{list-all } (\lambda v. \neg\text{list-ex } ((=) v) \text{ (delete-effects-of op)}) \text{ (precondition-of op')}
\text{using not-list-ex-equals-list-all-not[}
where P=\lambda v. \text{list-ex } ((=) v) \text{ (delete-effects-of op) and } xs=\text{precondition-of op']}
\text{by blast}
\} \\
\text{ultimately have } \beta:
\text{list-all } (\lambda v. \text{list-all } (\lambda v'. \neg v = v') \text{ (delete-effects-of op)}) \text{ (precondition-of op')}
\text{by presburger}
\} morever \{ \\
\text{fix } v \\
\text{have } \text{list-all } (\lambda v'. \neg v = v') \text{ (delete-effects-of op)}
= (\forall v' \in \text{set } (\text{delete-effects-of op}). \neg v = v')
\text{using list-all-iff [where P=\lambda v'. \neg v = v' and } x=\text{delete-effects-of op]}
\} \\
\text{ultimately have } \forall v \in \text{set } (\text{precondition-of op'}). \forall v' \in \text{set } (\text{delete-effects-of op}). \neg v = v'
\text{using } \beta \text{ list-all-iff[}
\text{where P=\lambda v. \text{list-all } (\lambda v'. \neg v = v') \text{ (delete-effects-of op) and } x=\text{precondition-of op']}
\text{by presburger}
\text{then have } v \notin \text{set } (\text{delete-effects-of op})
\text{using } \alpha
\text{by fast}
\} \\
\text{note } b = \text{this}
\} morever \{ \\
\text{fix } v \\
\text{assume } a: v \in \text{set } (\text{precondition-of op'})
\text{have } \text{list-all } (\lambda v. s v = \text{Some True}) \text{ (precondition-of op')}
\text{using assms(3)}
\text{unfolding is-operator-applicable-in-def}
\text{STRIPS-Representation.is-operator-applicable-in-def}
\text{by presburger}
\text{then have } \forall v \in \text{set } (\text{precondition-of op'}). s v = \text{Some True}
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 \( \exists s' v = \text{Some True} \)
proof (cases \( v \in \text{set } (\text{add-effects-of op}) \))
case True
then show \(?thesis\)
using \( a \)
by blast
next
case False
then show \(?thesis\)
proof (cases \( v \in \text{set } (\text{delete-effects-of op}) \))
case True
then show \(?thesis\)
using \( \text{assms(1)} \ b \ d \)
by fast
next
case False
then have \( \exists s' v = s v \)
using \( a \ c \)
by blast
then show \(?thesis\)
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 \(?thesis\)
unfolding \( \text{is-operator-applicable-in-def} \)
\( \text{STRIPS-Representation.is-operator-applicable-in-def} \)
by auto
qed

— The third assumption \( \text{are-all-operators-non-interfering (a \# ops)} \)" 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)  

23
= 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 ◦ effect-to-assignments) (Cons a ops))
    unfolding execute-parallel-operator-def
    by presburger
    also have ... = foldl (++) (?s’)
      (map (map-of ◦ effect-to-assignments) ops)
    by auto
    finally have execute-parallel-operator s (Cons a ops)
      = foldl (++) (?s’)
        (map (map-of ◦ 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 (map-of ◦ effect-to-assignments) a)
    unfolding execute-operator-def
    by simp

  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 \(\lambda v. (v, \text{True})\) (add-effects-of op) @ map \(\lambda v. (v, \text{False})\) (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) | v a. (v, a) \in set (add-effects-of op) \} 
∪ \{ (v, a), \text{False} | v a. (v, a) \in set (delete-effects-of op) \}

proof
- obtain as where effect--strips op = as
  and as = map \(\lambda v. (v, \text{True})\) (add-effects-of op)
  @ map \(\lambda v. (v, \text{False})\) (delete-effects-of op)
  unfolding effect--strips-def
  by blast
  moreover have as = map \(\lambda v. (v, \text{True})\) (add-effects-of op) @ map \(\lambda v. (v, \text{False})\) (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) \cap set (delete-effects-of op) = {}
shows effect-to-assignments op = map \(\lambda v. (v, \text{if ListMem v (add-effects-of op) then True else False})\)
(add-effects-of op @ delete-effects-of op)
and effect-to-assignments op = map \(\lambda v. (v, \text{if ListMem v (delete-effects-of op) then False else True})\)
(add-effects-of op @ delete-effects-of op)

proof
- let \(?f = \lambda v. (v, \text{if ListMem v (add-effects-of op) then True else False})\)
  and \(?g = \lambda v. (v, \text{if ListMem v (add-effects-of op) then False else True})\)
  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

25
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

(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)

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 \( v \notin \text{fst} \cdot \text{set} \ ?l \)
using \( \text{assms}(1, 2) \)
by blast

\}
thus \(?\text{thesis}\
using \?l \ v \\map-of-eq-None iff[of \ ?l \ v]\\by \text{blast}\\\\\\\text{qed}

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

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

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

let \( \text{?m} = (\text{map-of } \circ \text{ effect-to-assignments)} \text{ op} \)

have \( \text{?m} \in \text{ set } \text{?l} \)
    using assms(3) set-l-is
    by blast

moreover have \( \text{?m } v = \text{ Some True} \)
    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 \( \text{op'} \)
        where \( \text{op'}-\text{in-set-ops}: \text{op'} \in \text{ set ops} \)
            and \( \text{m'-is}: m' = (\text{map-of } \circ \text{ effect-to-assignments)} \text{ op'} \)
        by auto
    then have \( \text{set (add-effects-of op) } \cap \text{ set (delete-effects-of op')} = \{\} \)
        using assms(2, 3) are-all-operator-effects-consistent-set[of ops]
        by blast
    then have \( v \notin \text{ set (delete-effects-of op')} \)
        using assms(4)
        by blast
    then consider \( (\text{v-in-set-add-effects}) \text{ v } \in \text{ set (add-effects-of op')} \)
        | (\text{otherwise}) \( \neg v \in \text{ set (add-effects-of op')} \land \neg v \in \text{ set (delete-effects-of op')} \)
        by blast
    hence \( m' \text{ v } = \text{ Some True} \lor m' \text{ v } = \text{ None} \)
    proof (cases)
        case \( \text{v-in-set-add-effects} \)
            — 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 \( \text{otherwise} \)
            then have \( \neg v \in \text{ set (add-effects-of op')} \)
                and \( \neg v \in \text{ set (delete-effects-of op')} \)
                by blast+
            thus \( \text{?thesis} \)
                using map-of-effect-to-assignments-is-none-if[of v op'] m'-is
                by fastforce
    qed
}
ultimately show \( \texttt{thesis} \)

unfolding \texttt{execute-parallel-operator-def} 

using \texttt{foldl-map-append-is-some-if[of s v True ?l]}

by meson

qed

lemma \texttt{execute-parallel-operator-negative-effect-if-i}:

assumes are-all-operators-applicable s ops

and are-all-operator-effects-consistent ops 

and op \( \in \) set ops

and v \( \in \) set (delete-effects-of op)

shows map-of (effect-to-assignments op) v = Some False

proof

let \( \forall f = \lambda x. \text{if ListMem } x \text{ (delete-effects-of op) then False else True} \)

and \( \forall !' = \text{map (} \lambda v. \text{if ListMem } v \text{ (delete-effects-of op) then False else True} \)

True) 

(\text{add-effects-of op} @ delete-effects-of op)

have set (\text{delete-effects-of op} @ add-effects-of op) \( \neq \) \{\}

using assms.(4)

by fastforce

moreover have v \( \in \) set (\text{delete-effects-of op} @ add-effects-of op)

using assms.(4)

by simp

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 = \( !' \)

using effect-to-assignments-construction-from-function-graph(2) calculation

by blast

ultimately have map-of (effect-to-assignments op) = map-of \( !' \)

by argo

}

ultimately have map-of (effect-to-assignments op) v = Some (\( \forall f v \)

using Map-Supplement.map-of-from-function-graph-is-some-if[

of add-effects-of op @ delete-effects-of op v \( \forall f \)]

by force

thus \( \texttt{thesis} \)

using assms(4)

unfolding ListMem-iff

by presburger

qed

lemma \texttt{execute-parallel-operator-negative-effect-if}: 

assumes are-all-operators-applicable s ops 

and are-all-operator-effects-consistent ops 

and op \( \in \) set ops

and v \( \in \) set (delete-effects-of op)
shows \( \text{execute-parallel-operator } s \text{ ops } v = \text{Some False} \)

proof -

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

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

\{
  let \(?m = (\text{map-of o effect-to-assignments) op}\
  \text{have } ?m \in \text{set } ?l\
  \text{using assms(3) set-l-is}
  \text{by blast}

  moreover have \(?m v = \text{Some False}\
    \text{using execute-parallel-operator-negative-effect-if-i[OF assms]}\
    \text{by fastforce}
  \}

  ultimately have \( \exists m \in \text{set } ?l. m v = \text{Some False} \)
  \text{by blast}
\}

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

proof (cases)
  case v\in-set-delete-effects
  \text{— TODO slow.}
  \text{thus } ?thesis
    \text{using execute-parallel-operator-negative-effect-if-i[}
    \text{OF assms(1, 2) op'\in-set-ops, of v] m'\text{-is}}\
    \text{by simp}

next
  case otherwise
  \text{then have } \neg v \in \text{set (add-effects-of op')}
  \text{and } \neg v \in \text{set (delete-effects-of op')}
  \text{by blast+}
  \text{thus } ?thesis
    \text{using map-of-effect-to-assignments-is-none-if[of v op'] m'\text{-is}}\
    \text{by fastforce}
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
by argo
**corollary** execute-parallel-operators-strips-none-if:

**assumes** \( \forall \text{op} \in \text{set ops}. \neg v \in \text{set (add-effects-of op)} \land \neg v \in \text{set (delete-effects-of op)} \)

\[ \text{and } s \ v = \text{None} \]

**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** \( \neg \text{execute-parallel-operator s ops v = None} \)

**shows** \( (\exists \text{op} \in \text{set ops}. v \in \text{set (add-effects-of op)} \lor v \in \text{set (delete-effects-of op)}) \land s \ v \neq \text{None} \)

**proof**

- **let** ?P = \((\forall \text{op} \in \text{set ops}. \neg v \in \text{set (add-effects-of op)} \land \neg v \in \text{set (delete-effects-of op)}) \land s \ v = \text{None} \)
  - **and** ?Q = execute-parallel-operator s ops v = None
  - **have** ?P \(\Rightarrow\) ?Q
    - **using** execute-parallel-operators-strips-none-if[of ops v s]
  - **by blast**
- **then have** \(\neg ?P \)
  - **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 \( \text{op} \in \text{set ops} \) and with \( v \) being an add effect of op, then the successor state

\( s' \equiv \text{execute-parallel-operator s ops} \)

will evaluate \( v \) to true, that is

\( \text{execute-parallel-operator s ops v} = \text{Some True} \)

Symmetrically, if \( v \) is a delete effect, we have

\( \text{execute-parallel-operator s ops v} = \text{Some False} \)

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 ops, then the state after parallel operator execution remains unchanged, i.e.
execute-parallel-operator $s$ $ops$ $v = s$ $v$

**Theorem** execute-parallel-operator-effect:
**Assumes** are-all-operators-applicable $s$ $ops$
**And** are-all-operator-effects-consistent $ops$
**Shows** $op \in set$ $ops \land v \in set$ (add-effects-of $op$)
$\rightarrow$ execute-parallel-operator $s$ $ops$ $v = Some$ True
**And** $op \in set$ $ops \land v \in set$ (delete-effects-of $op$)
$\rightarrow$ execute-parallel-operator $s$ $ops$ $v = Some$ False
**Using** execute-parallel-operator-positive-effect-if[OF assms]
execute-parallel-operator-negative-effect-if[OF assms]
execute-parallel-operator-no-effect-if[of $ops$ $v$ $s$]
**By** blast+

**Lemma** is-parallel-solution-for-problem-operator-set:
**Fixes** $II::`a$ strips-problem
**Assumes** is-parallel-solution-for-problem $\Pi$ $\pi$
**And** $ops \in set$ $\pi$
**And** $op \in set$ $ops$
**Shows** $op \in set$ ($(\Pi \cap )$
**Proof** –
**Have** $\forall$ $ops \in set$ $\pi$. $\forall$ $op \in set$ (strips-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 $I$ (hd $\pi$) $\land$ are-all-operator-effects-consistent (hd $\pi$), simp+)
**Qed** simp

**Corollary** length-trace-parallel-plan-gt-0(simp): $0 < length$ (trace-parallel-plan-strips $I$ $\pi$)
**Using** trace-parallel-plan-strips-not-nil..

**Corollary** length-trace-minus-one-ll-length-trace[simp]:
length (trace-parallel-plan-strips $I$ $\pi$) $- 1 < length$ (trace-parallel-plan-strips $I$ $\pi$)
**Using** diff-less[OF - length-trace-parallel-plan-gt-0]
by \textit{auto}

\textbf{lemma} \textit{trace-parallel-plan-strips-head-is-initial-state}: \textit{trace-parallel-plan-strips} $I\pi!0 = I$

\textbf{proof} \ (\textit{cases} $\pi$)

\textbf{case} ($\textit{Cons} \ a \ \textit{list}$)

\textbf{then show} \ ?thesis

\textbf{by} \ (\textit{cases} \ \textit{are-all-operators-applicable} \ $I\ a\ \wedge \ \textit{are-all-operator-effects-consistent} \ a, \ \textit{simp+})

\textbf{qed} \ \textit{simp}

\textbf{lemma} \textit{trace-parallel-plan-strips-length-gt-one-if}:

\textbf{assumes} \ $k < \textit{length} \ (\textit{trace-parallel-plan-strips} \ I\pi) - 1$

\textbf{shows} \ $1 < \textit{length} \ (\textit{trace-parallel-plan-strips} \ I\pi)$

\textbf{using} \ \textit{assms}

\textbf{by} \ \textit{linarith}

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

\textbf{lemma} \textit{trace-parallel-plan-strips-last-cons-then}:

\textbf{last} \ ($s \# \textit{trace-parallel-plan-strips} s' \pi) = \textit{last} \ (\textit{trace-parallel-plan-strips} s' \pi)$

\textbf{by} \ (\textit{cases} $\pi, \ \textit{simp, force}$)

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

First, all parallel operators $\textit{ops} = \pi!k$ for any index $k$ with $k < \textit{length} \ \tau - 1$ (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 $\textit{ops}$ would have been the last element. This would violate the assumption that $k < \textit{length} \ \tau - 1$ is not the last index since the index of the last element is \textit{length} $\tau - 1$. \footnote{More precisely, the index of the last element is \textit{length} $\tau - 1$ if $\tau$ is not empty which is however always true since the trace contains at least the initial state.}

\textbf{lemma} \textit{trace-parallel-plan-strips-operator-preconditions}:

\textbf{assumes} \ $k < \textit{length} \ (\textit{trace-parallel-plan-strips} \ I\pi) - 1$

\textbf{shows} \ \textit{are-all-operators-applicable} \ (\textit{trace-parallel-plan-strips} \ I\pi!) \ (\pi!k) \ 

$\wedge \ \textit{are-all-operator-effects-consistent} \ (\pi!k)$

\textbf{using} \ \textit{assms}

\textbf{proof} \ (\textit{induction} $\pi$ arbitrary; \ $I \ k$)

— NOTE Base case yields contradiction with assumption and can be left to automation.

\textbf{case} \ ($\textit{Cons} \ a \ \pi$)

\textbf{then show} \ ?case

\textbf{proof} \ (\textit{cases} \ \textit{are-all-operators-applicable} \ $I \ a\ \wedge \ \textit{are-all-operator-effects-consistent} \ a, \ \textit{simp+})$
case True
have trace-parallel-plan-strips-cons: trace-parallel-plan-strips I (a # π)
  = I # trace-parallel-plan-strips (execute-parallel-operator I a) π
  using True
  by simp
then show ?thesis
proof (cases k)
case 0
have trace-parallel-plan-strips I (a # π) ! 0 = I
  using trace-parallel-plan-strips-cons
  by simp
moreover have (a # π) ! 0 = a
  by simp
ultimately show ?thesis
  using True 0
  by presburger
next
case (Suc k')
let I' = execute-parallel-operator I a
have trace-parallel-plan-strips I (a # π) ! Suc k' = trace-parallel-plan-strips I' π ! k'
  using trace-parallel-plan-strips-cons
  by simp
moreover have (a # π) ! Suc k' = π ! k'
  by simp
moreover {
  have length (trace-parallel-plan-strips I (a # π))
    = 1 + length (trace-parallel-plan-strips ?I' π)
    unfolding trace-parallel-plan-strips-cons
    by simp
  then have k' < length (trace-parallel-plan-strips ?I' π) - 1
    using Suc.Cons.prems
    by fastforce
  hence are-all-operators-applicable (trace-parallel-plan-strips ?I' π ! k')
    (π ! k')
    ∧ are-all-operator-effects-consistent (π ! k')
    using Cons.IH[of k']
    by blast
}
ultimately show ?thesis
  using Suc
  by argo
qed
next
case False
then have trace-parallel-plan-strips I (a # π) = [I]
  by force
then have length (trace-parallel-plan-strips I (a # π)) - 1 = 0
by simp
— NOTE Thesis follows from contradiction with assumption.
then show ?thesis
  using Cons.prems
  by force
qed
qed auto

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** 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 assms
  proof (induction $\pi$ arbitrary: $I k$
    case (Cons $a \pi$
      then show ?case
        proof (cases are-all-operators-applicable $I a \land$ 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 ?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 $\text{execute-parallel-plan} I (\text{take} k (a \# \pi))$
                    $= \text{execute-parallel-plan} ?I' (\text{take} k' \pi)$
                    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 $k' < \text{length} (\text{trace-parallel-plan-strips} (\text{execute-parallel-operator} I$
                      $a) \pi)$
                      using Cons.prems $\sigma$-equals $k$-is-suc-of-$k'$
                      by force
                    hence $\text{trace-parallel-plan-strips} ?I' \pi ! k'$
execute-parallel-plan ?I' (take k' \pi)
    using Cons.IH[of k' ?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-strips I (a \# \pi) = [I]
      using operator-precondition-violated
      by force
    moreover have execute-parallel-plan I (take k (a \# \pi)) = 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 length-trace-parallel-plan-strips-lte-length-plan-plus-one:
  shows length (trace-parallel-plan-strips I \pi) \leq length \pi + 1
  proof (induction \pi arbitrary: I)
    case (Cons a \pi)
    then show ?case
    proof (cases are-all-operators-applicable I a \land are-all-operator-effects-consistent a)
      case True
      let ?I' = execute-parallel-operator I a
      { have trace-parallel-plan-strips I (a \# \pi) = I \# trace-parallel-plan-strips ?I' \pi
        using True
        by auto
        then have length (trace-parallel-plan-strips I (a \# \pi))
          = length (trace-parallel-plan-strips ?I' \pi) + 1
          by simp
        moreover have length (trace-parallel-plan-strips ?I' \pi) \leq length \pi + 1
          using Cons.IH[of ?I']
          by blast
        ultimately have length (trace-parallel-plan-strips I (a \# \pi)) \leq 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 \(\text{ops } \pi'\) where \(\pi = \text{ops } \# \pi'\)

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 \(\neg \text{are-all-operators-applicable}\)

\((\text{execute-parallel-plan } I (\text{take } (\text{length } (\text{trace-parallel-plan-strips } I \pi) - 1) \pi))\)

\((\pi ! (\text{length } (\text{trace-parallel-plan-strips } I \pi) - 1))\)

\(\lor \neg \text{are-all-operator-effects-consistent}\)

proof (induction \(\pi\) arbitrary: \(I\))
  case (Cons \(\text{ops } \pi\))
  let \(?\tau = \text{trace-parallel-plan-strips } I (\text{ops } \# \pi)\)
  and \(?I' = \text{execute-parallel-operator } I \text{ ops}\)
  show \(?case\)
    proof (cases \(\text{are-all-operators-applicable } I \text{ ops } \land \text{are-all-operator-effects-consistent ops}\))
      case True
      then have \(\tau\)-is: \(?\tau = I \# \text{trace-parallel-plan-strips } ?I' \pi\)
        by fastforce
      show \(?thesis\)
        proof (cases \(\text{length } (\text{trace-parallel-plan-strips } ?I' \pi) < \text{length } \pi + 1\))
case True
then have ¬ are-all-operators-applicable
  (execute-parallel-plan ?I’
   (take (length (trace-parallel-plan-strips ?I’ π) − 1) π)
   (π ! (length (trace-parallel-plan-strips ?I’ π) − 1))
   ∨ ¬ are-all-operator-effects-consistent
   (π ! (length (trace-parallel-plan-strips ?I’ π) − 1))
  using Cons.IH[of ?I’]
  by blast
moreover have trace-parallel-plan-strips ?I’ π ≠ []
  using trace-parallel-plan-strips-not-nil
  by blast
ultimately show ?thesis
  unfolding take-Cons’
  by simp
next
case False
then have length (trace-parallel-plan-strips ?I’ π) ≥ length π + 1
  by fastforce
  thm Cons.prems
moreover have length (trace-parallel-plan-strips I (ops # π))
  = 1 + length (trace-parallel-plan-strips ?I’ π)
  using True
  by force
moreover have length (trace-parallel-plan-strips ?I’ π)
  < length (ops # π)
  using Cons.prems calculation(2)
  by force
ultimately have False
  by fastforce
  thus ?thesis..
qed
next
case False
then have τ-is-singleton: ?τ = [I]
  using False
  by auto
then have ops = (ops # π) ! (length ?τ − 1)
  by fastforce
moreover have execute-parallel-plan I (take (length ?τ − 1) π) = I
  using τ-is-singleton
  by auto
  — TODO slow.
ultimately show ?thesis
  using False
  by auto
qed
qed simp
**Lemma** \( \text{trace-parallel-plan-step-effect-is} \):

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

**Shows** \( \text{trace-parallel-plan-strips I } \pi \! \text{ Suc k} \)

\[ = \text{execute-parallel-operator (trace-parallel-plan-strips I } \pi \! k) \text{(}\pi \! k) \]

**Proof**

— NOTE Rewrite the proposition using lemma \( \text{trace-parallel-plan-strips-subplan} \).

\[
\begin{align*}
\text{let } ?\tau &= \text{trace-parallel-plan-strips I } \pi \\
\text{have } \text{Suc } k &< \text{length } ?\tau \\
\text{by } \text{linarith} \\
\text{hence } \text{trace-parallel-plan-strips I } \pi \! \text{ Suc k} \\
&= \text{execute-parallel-plan I (take (Suc k) } \pi) \\
\text{using } \text{trace-parallel-plan-plan-prefix[of Suc k I } \pi] \\
\text{by } \text{blast}
\end{align*}
\]

**Moreover have** \( \text{execute-parallel-plan I (take (Suc k) } \pi) \\
= \text{execute-parallel-operator (trace-parallel-plan-strips I } \pi \! k) \text{(}\pi \! k) \)

**Proof** ( induction \( k \) arbitrary: \( I \pi \))

**Case** \( 0 \)

\[
\begin{align*}
\text{then have } &\text{execute-parallel-operator (trace-parallel-plan-strips I } \pi \! 0) \text{(}\pi \! 0) \\
&= \text{execute-parallel-operator I (}\pi \! 0) \\
\text{using } \text{trace-parallel-plan-strips-head-is-initial-state[of I } \pi] \\
\text{by } \text{argo} \\
\text{moreover } \{ \\
\text{obtain } &\text{ops } \pi' \text{ where } \pi = \text{ops } \# \pi' \\
\text{using } &\text{plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF 0.prems]} \\
\text{by } &\text{blast} \\
\text{then have } &\text{take (Suc 0) } \pi = [\pi \! 0] \\
\text{by } &\text{simp} \\
\text{hence } &\text{execute-parallel-plan I (take (Suc 0) } \pi) \\
&= \text{execute-parallel-plan I [}\pi \! 0] \\
\text{by } &\text{argo}
\}
\]

**Moreover** \( \{ \\
\text{have } &0 < \text{length (trace-parallel-plan-strips I } \pi) - 1 \\
\text{using } &\text{trace-parallel-plan-strips-length-gt-one-if 0.prems} \\
\text{by } &\text{fastforce} \\
\text{hence } &\text{are-all-operators-applicable I (}\pi \! 0) \\
\text{\& } &\text{are-all-operator-effects-consistent (}\pi \! 0) \\
\text{using } &\text{trace-parallel-plan-strips-operator-preconditions[of 0 I } \pi] \\
\text{trace-parallel-plan-strips-head-is-initial-state[of I } \pi] \\
\text{by } &\text{argo}
\}
\]

**Ultimately show** \( ?\text{case} \)

**by** \text{auto}
next
  case (Suc k)
  obtain ops π' where π-split: π = ops # π'
      using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF Suc.prems]
      by blast
  let ?I' = execute-parallel-operator I ops
  {
    have length (trace-parallel-plan-strips I π) =
        I + length (trace-parallel-plan-strips ?I' π')
      using Suc.prems π-split
      by fastforce
    then have k < length (trace-parallel-plan-strips ?I' π')
      using Suc.prems
      by fastforce
    moreover have trace-parallel-plan-strips I π ! Suc k
        = trace-parallel-plan-strips ?I' π' ! k
      using Suc.prems π-split
      by force
    ultimately have trace-parallel-plan-strips I π ! Suc k
        = execute-parallel-plan ?I' (take k π')
      using trace-parallel-plan-plan-prefix[of k ?I' π']
      by argo
  }
  moreover have execute-parallel-plan I (take (Suc (Suc k)) π)
        = execute-parallel-plan ?I' (take (Suc k) π')
      using Suc.prems π-split
      by fastforce
  moreover {
    have 0 < length (trace-parallel-plan-strips I π) − 1
      using Suc.prems
      by linarith
    hence are-all-operators-applicable I (π ! 0)
      ∧ are-all-operator-effects-consistent (π ! 0)
      using trace-parallel-plan-strips-operator-preconditions[of 0 I π]
      trace-parallel-plan-strips-head-is-initial-state[of 1 π]
      by argo
  }
  ultimately show ?case
      using Suc.IH Suc.prems π-split
      by auto
  qed
  ultimately show ?thesis
      using assms
      by argo
  qed

— 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
lemma trace-parallel-plan-strips-none-if:
fixes II:: 'a strips-problem
assumes is-valid-problem-strips II
and is-parallel-solution-for-problem II π
and k < length (trace-parallel-plan-strips ((Π)_I) π)
shows (trace-parallel-plan-strips ((Π)_I) π ! k) v = None ⟷ v /∈ set ((Π)_V)
proof
  let ?vs = strips-problem.variables-of II
  and ?ops = strips-problem.operators-of II
  and ?τ = trace-parallel-plan-strips ((Π)_I) π
  and ?I = strips-problem.initial-of II
  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)
      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 /∈ set ((Π)_V)
    proof (rule ccontr)
      assume ¬v /∈ set ((Π)_V)
      then have v-in-set-v: v ∈ set((Π)_V)
      by blast
    show False
proof (cases ∃ op ∈ set (π ! k).
  v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op))
  case True
  then obtain op
    where op-in-πₖ: op ∈ set (π ! k)
    and v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)
  thus consider (A) v ∈ set (add-effects-of op)
    by blast
    thus False
    using execute-parallel-operator-positive-effect-if[OF
      all-operators-applicable all-effects-consistent op-in-πₖ]
    execute-parallel-operator-negative-effect-if[OF
      all-operators-applicable all-effects-consistent op-in-πₖ]
    τ-Suc-k-of-v-is-None τ-Suc-k-is
    by (cases, fastforce+)
  next
  case False
  then have ∀ op ∈ set (π ! k).
    v /∈ set (add-effects-of op) ∧ v /∈ set (delete-effects-of op)
    by blast
  then have (?τ ! Suc k) v = (?τ ! k) v
    using execute-parallel-operator-no-effect-if τ-Suc-k-is
    by fastforce
  then have v /∈ set ((Π)ᵥ)
    using IH τ-Suc-k-of-v-is-None
    by simp
  thus False
  using v-in-set-vs
  by blast
qed
next
assume v-notin-vs: v /∈ set ((Π)ᵥ)
{
  fix op
  assume op-in-πₖ: op ∈ set (π ! k)
  {
    have 1 < length ?τ
      using trace-parallel-plan-strips-length-gt-one-if[OF
        k-lt-length-τ-minus-one],
    then have 0 < length ?τ − 1
      using k-lt-length-τ-minus-one
      by linarith
    moreover have length ?τ − 1 ≤ length π
      using length-trace-parallel-plan-strips-lte-length-plan-plus-one
      le-diff-conv
      by blast
    then have k < length π
      using k-lt-length-τ-minus-one
  }
}
by force

hence $\pi ! k \in \text{set } \pi$

by simp

} then have $\text{op-in-ops: op } \in \text{set } ?\text{ops}$

using $\text{is-parallel-solution-for-problem-operator-set} [OF \text{assms}(2) - \text{op-in-}\pi_k]$

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 $\text{is-valid-problem-strips-operator-variable-sets}(2) \text{ assms}(1)$

$\text{op-in-ops}$

$v\text{-notin-vs}$

by auto

subgoal

using $\text{is-valid-problem-strips-operator-variable-sets}(3) \text{ assms}(1)$

$\text{op-in-ops}$

$v\text{-notin-vs}$

by auto

done

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

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

**proof**

let $?\text{LHS} = G \subseteq_m \text{execute-parallel-plan } I \pi$

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

**show** $?\text{thesis}$

**proof** (rule iffI)

**assume** $?\text{LHS}$

**thus** $?\text{RHS}$

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

— NOTE Nil case follows from simplification.

**case** $(\text{Cons } a \pi)$

**thus** $?\text{case}$

44
using Cons.prems

proof (cases are-all-operators-applicable I a ∧ are-all-operator-effects-consistent)

  a)
  
  case True
  let ?I' = execute-parallel-operator I a
  {
    have execute-parallel-plan I (a ≠ π) = execute-parallel-plan ?I' π
      using True
      by auto
    then have G ⊆m execute-parallel-plan ?I' π
      using Cons.prems
      by presburger
    hence G ⊆m last (trace-parallel-plan-strips ?I' π)
      using Cons.H[I \of ?I]
      by blast
  }
  moreover {
    have trace-parallel-plan-strips I (a ≠ π)
      = I ≠ trace-parallel-plan-strips ?I' π
      using True
      by simp
    then have last (trace-parallel-plan-strips I (a ≠ π))
      = last (I ≠ trace-parallel-plan-strips ?I' π)
      by argo
    hence last (trace-parallel-plan-strips I (a ≠ π))
      = last (trace-parallel-plan-strips ?I' π)
      using trace-parallel-plan-strips-last-cons-then[I \of ?I]
      by argo
  }
  ultimately show ?thesis
  by argo
  qed simp
  qed force

next
assume ?RHS
thus ?LHS

proof (induction π arbitrary: I)
— NOTE Nil case follows from simplification.
  case (Cons a π)
  thus ?case

proof (cases are-all-operators-applicable I a ∧ are-all-operator-effects-consistent)

  a)
  
  case True
  let ?I' = execute-parallel-operator I a
  {
    have trace-parallel-plan-strips I (a ≠ π) = I ≠ (trace-parallel-plan-strips

?I' π)
      using True
      by simp

45
then have last (trace-parallel-plan-strips I (a ≠ π))
= last (trace-parallel-plan-strips ?I' π)
using trace-parallel-plan-strips-last-cons-then[of I ?I' π]
by argo
hence G ⊆ m last (trace-parallel-plan-strips ?I' π)
using Cons.prems
by argo
} thus ?thesis
using True Cons
by simp
next
case False
then have last (trace-parallel-plan-strips I (a ≠ π)) = I
and execute-parallel-plan I (a ≠ π) = I
by (fastforce, force)
thus ?thesis
using Cons.prems
by argo
qed
qed fastforce
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

lemma execute-parallel-operator-equals-execute-sequential-strips-if:
  fixes s :: (variable, bool) state
  assumes are-all-operators-applicable s ops

5In the source literature it is required that app_Σ(s) is defined which requires that app_Σ(s) is defined for every o ∈ S. This again means that the preconditions hold in s and the set of effects is consistent which translates to the execution condition in execute-parallel-operator. [3, Lemma 2.11., p.1037]

Also, the proposition [3, Lemma 2.11., p.1037] is in fact proposed to be true for any total 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.
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 \circ effect-to-assignments) Nil)
using Nil.prems(1,2)
unfolding execute-parallel-operator-def
by presburger
also have \ldots = 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
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

47
is-operator-applicable-in-def
by fastforce
ultimately show
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
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
using are-all-operators-applicable-cons(2)
unfolding is-operator-applicable-in-def
STRIPS-Representation.is-operator-applicable-in-def
are-all-operators-applicable-def list-all-iff
by simp
qed

48
Show that plans \( \pi \) can be split into separate executions of partial plans \( \pi_1 \) and \( \pi_2 \) with \( \pi = \pi_1 \oplus \pi_2 \), 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 \( s : (\text{'variable, bool}) \text{state} \)

Assumes are-all-operators-applicable \( s \pi_1 \)

And are-all-operators-non-interfering \( \pi_1 \)

Shows \( \text{execute-serial-plan} \ s \ (\pi_1 \oplus \pi_2) = \text{execute-serial-plan} \ (\text{execute-serial-plan} \ s \pi_1) \pi_2 \)

Using assms

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

Case \((\text{Cons op} \pi_1)\)

Let \( ?t = s \gg op \) \n
\{  

have are-all-operators-applicable \( \{s \gg op\} \pi_1 \)  

using \( \text{execute-serial-plan-split-i[OF Cons.prems(1, 2)]} \).

moreover have are-all-operators-non-interfering \( \pi_1 \)  

using \( \text{are-all-operators-non-interfering-tail[OF Cons.prems(2)]} \).

ultimately have \( \text{execute-serial-plan} \ ?t \ (\pi_1 \oplus \pi_2) = \text{execute-serial-plan} \ (\text{execute-serial-plan} \ ?t \pi_1) \pi_2 \)  

using \( \text{Cons.IH[of ?t]} \)  

by blast

\}

moreover have \( \text{STRIPS-Representation.is-operator-applicable-in} \ s \ op \)  

using \( \text{Cons.prems(1)} \)

unfolding are-all-operators-applicable-def list-all-iff  

by fastforce

ultimately show \( ?case \)  

unfolding \( \text{execute-serial-plan-def} \)  

by simp

qed simp

**Lemma embedding-lemma-i:**

Fixes \( I : (\text{'variable, bool}) \text{state} \)

Assumes \( \text{is-operator-applicable-in} \ I \ op \)  

And \( \text{are-operator-effects-consistent} \ op \ op \)

Shows \( I \gg op = \text{execute-parallel-operator} \ I \ [op] \)

Proof =

have \( \text{are-all-operators-applicable} \ I \ [op] \)  

using assms(I)

unfolding are-all-operators-applicable-def list-all-iff is-operator-applicable-in-def  

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

lemma \(\text{execute-serial-plan-is-execute-parallel-plan-ii}\):
fixes \(I :: \text{variable strips-state}\)
assumes \(\forall \text{op} \in \text{set } \pi. \text{are-operator-effects-consistent} \ \text{op} \ \text{op}\)
and \(G \subseteq_m \text{execute-serial-plan} \ I \ \pi\)
shows \(G \subseteq_m \text{execute-parallel-plan} \ I \ (\text{embed } \pi)\)
proof -
show \(?\text{thesis}\)
using assms
proof (induction \(\pi\) arbitrary: \(I\))
case (Cons \(\text{op}\) \(\pi\))
then show \(?\text{case}\)
proof (cases is-operator-applicable-in \(I \ \text{op}\))
  case True
  let \(?J = I \gg \text{op}\)
  and \(?J' = \text{execute-parallel-operator} \ I \ [\text{op}]\)
  {
  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 \(I \ [\text{op}]\)
    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 \([\text{op}]\)
    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 ¬ all-operators-applicable I [op]
    using False
    unfolding 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

lemma embedding-lemma-iii:
fixes II:: 'a strips-problem
assumes ∀ op ∈ set π. op ∈ set ((II) op)
shows ∀ ops ∈ set (embed π). ∀ op ∈ set ops. op ∈ set ((II) op)
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]
      using nb calculation
      by blast
    ultimately have ∀ op ∈ set ops. op ∈ set ((II) op)
      using assms(1)
      by simp
  }

51
thus \textit{thesis}.
qed

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 List-Supplement.\textit{embed}). The proof shows this by first confirming that

$$G \subseteq_m \text{execute-serial-plan} \left( (\Pi)_I \right) \pi$$
$$\Rightarrow G \subseteq_m \text{execute-serial-plan} \left( (\Pi)_I \right) \left( \text{embed} \pi \right)$$

using lemma ; and moreover by showing that

$$\forall \text{ops} \in \text{set} \left( \text{embed} \pi \right), \forall \text{op} \in \text{set} \text{ops}. \text{op} \in (\Pi)_{\text{O}}$$

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:}
assumes \textit{is-valid-problem-strips} $\Pi$
and \textit{is-serial-solution-for-problem} $\Pi \ \pi$
shows \textit{is-parallel-solution-for-problem} $\Pi \ (\text{embed} \ \pi)$
proof —
have nb1: $\forall \text{op} \in \text{set} \ \pi. \ \text{op} \in (\Pi)_{\text{O}}$
using assms\textit{(2)}
unfolding \textit{is-serial-solution-for-problem-def} list-all-iff ListMem-iff operators-of-def
by blast
\{  
fix \textit{op}  
assume \textit{op} \in \text{set} \ \pi  
moreover have \textit{op} \in (\Pi)_{\text{O}}
using nb1 calculation
by fast
moreover have \textit{is-valid-operator-strips} $\Pi \ \textit{op}$
using assms\textit{(1)} calculation\textit{(2)}
unfolding \textit{is-valid-problem-strips-def} is-valid-problem-strips-def list-all-iff operators-of-def
by meson
moreover have list-all (\lambda v. \neg ListMem v (\text{delete-effects-of op})) (\text{add-effects-of op})
and list-all (\lambda v. \neg ListMem v (\text{add-effects-of op})) (\text{delete-effects-of op})
using calculation\textit{(3)}
unfolding \textit{is-valid-operator-strips-def}
by meson+
moreover have \neg list-ex (\lambda v. ListMem v (\text{delete-effects-of op})) (\text{add-effects-of op})
and \neg list-ex (\lambda v. ListMem v (\text{add-effects-of op})) (\text{delete-effects-of op})
using calculation\textit{(4, 5)} not-list-ex-equals-list-all-not

52
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 \( \neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) \text{(add-effects-of op)}) \text{(delete-effects-of op)} \)
using calculation (6, 7)
unfolding list-ex-iff ListMem-iff
by blast+
ultimately have are-operator-effects-consistent op op
unfolding are-operator-effects-consistent-def Let-def
by blast
\}
\text{note } nb_2 = \text{this}
moreover \{
\begin{align*}
\text{have } & (\Pi)_G \subseteq_m \text{execute-serial-plan } ((\Pi)_I) \pi \\
\text{using } & \text{assms} (2) \\
\text{unfolding } & \text{is-serial-solution-for-problem-def} \\
\text{by } & \text{simp} \\
\text{hence } & (\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) (\text{embed } \pi) \\
\text{using } & \text{execute-serial-plan-is-execute-parallel-plan-II } nb_2 \\
\text{by } & \text{blast} \\
\end{align*}
\}
moreover have \( \forall \text{ops } \in \text{set (embed } \pi). \forall \text{op } \in \text{set ops. op } \in \text{set } ((\Pi)_O) \)
using embedding-lemma-iii [OF nb_1],
ultimately show \( ?\text{thesis} \)
unfolding is-parallel-solution-for-problem-def goal-of-def
initial-of-def operators-of-def list-all-iff ListMem-iff
by blast
qed

\text{lemma flattening-lemma-i:}
\text{fixes } II:: \text{‘a strips-problem}
\text{assumes } \forall \text{ops } \in \text{set } \pi. \forall \text{op } \in \text{set ops. op } \in \text{set } ((\Pi)_O) 
\text{shows } \forall \text{op } \in \text{set } (\text{concat } \pi). \text{op } \in \text{set } ((\Pi)_O) 
\text{proof} –
\begin{align*}
\text{fix } & \text{op} \\
\text{assume } & \text{op } \in \text{set } (\text{concat } \pi) \\
\text{moreover have } & \text{op } \in \bigcup \text{ops } \in \text{set } \pi. \text{set ops} \\
\text{using } & \text{calculation} \\
\text{unfolding } & \text{set-concat.} \\
\text{then obtain } & \text{ops where } \text{ops } \in \text{set } \pi \text{ and } \text{op } \in \text{set ops} \\
\text{using } & \text{UN-iff} \\
\text{by } & \text{blast} \\
\text{ultimately have } & \text{op } \in \text{set } ((\Pi)_O) \\
\text{using } & \text{assms} \\
\text{by } & \text{blast} \\
\end{align*}
\text{thus } ?\text{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' = concat \pi

  \{ 
  fix \text{op}
  assume is-valid-operator-strips \Pi \text{ op}
  moreover have list-all (\( \lambda v. \neg \text{ListMem } v (\text{delete-effects-of } \text{op}) \)) (\text{add-effects-of op})
    and list-all (\( \lambda v. \neg \text{ListMem } v (\text{add-effects-of } \text{op}) \)) (\text{delete-effects-of op})
    using calculation(1)
    unfolding is-valid-operator-strips-def
    by meson+
  moreover have \( \neg \text{list-ex } (\lambda v. \text{ListMem } v (\text{delete-effects-of } \text{op})) (\text{add-effects-of op}) \)
    and \( \neg \text{list-ex } (\lambda v. \text{ListMem } v (\text{add-effects-of op})) (\text{delete-effects-of op}) \)
    using calculation(2, 3) not-list-ex-equals-list-all-not
    by blast+
  moreover have \( \neg \text{list-ex } (\lambda v. \text{list-ex } ((=) v) (\text{delete-effects-of } \text{op})) (\text{add-effects-of op}) \)
    and \( \neg \text{list-ex } (\lambda v. \text{list-ex } ((=) v) (\text{add-effects-of } \text{op})) (\text{delete-effects-of op}) \)
    using calculation(4, 5) list-ex-iff ListMem-iff
    by blast+
  ultimately have are-operator-effects-consistent op \Pi \text{ op}
  unfolding are-operator-effects-consistent-def Let-def
  by blast
  } note nb\_1 = this
show ?\text{thesis}
using asms
proof (induction \pi arbitrary: I)
case (Cons \text{ops } \pi )
obtain \text{op} where \text{ops-is: ops} = [\text{op}] and \text{is-valid-op: is-valid-operator-strips }
\Pi \text{ op}
  using Cons.prems(1)
  by fastforce
show ?case
proof (cases are-all-operators-applicable I \text{ops})
case True
let ?J = execute-parallel-operator I [\text{op}]
  and ?J' = I \Rightarrow \text{op}
have nb\_2: is-operator-applicable-in I \text{op}
  using True \text{ops-is}
unfolding are-all-operators-applicable-def list-all-iff
  is-operator-applicable-in-def
by simp
have nb:
  are-operator-effects-consistent op op
using nb[OF is-valid-op],
{
  then have are-all-operator-effects-consistent ops
    unfolding are-all-operator-effects-consistent-def list-all-iff
    using ops-is
    by fastforce
    hence G ⊆ m execute-parallel-plan ?J π
    using Cons.prems(2) ops-is True
    by fastforce
}
moreover have execute-serial-plan I (concat (ops ≠ π))
  = execute-serial-plan ?J' (concat π)
using ops-is nb2
unfolding is-operator-applicable-in-def
by (simp add: execute-operator-def nb2)
moreover have ?J = ?J'
unfolding execute-parallel-operator-def execute-operator-def comp-apply
by fastforce
ultimately show ?thesis
  using Cons.IH Cons.prems
  by force
next
case False
moreover have G ⊆ m I
using Cons.prems(2) calculation
by force
moreover {
  have ¬ is-operator-applicable-in I op
    using ops-is False
    unfolding are-all-operators-applicable-def list-all-iff
    is-operator-applicable-in-def
    by fastforce
    hence execute-serial-plan I (concat (ops ≠ π)) = I
    using ops-is is-operator-applicable-in-def
    by simp
}
ultimately show ?thesis
  by argo
qed
qed force
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 ∀ ops ∈ set π. ∃ op. ops = [op]
and is-parallel-solution-for-problem $\Pi \pi$

shows is-serial-solution-for-problem $\Pi (\text{concat } \pi)$

proof

let $\pi' = \text{concat } \pi$

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

moreover {
  
  fix ops
  assume ops $\in$ set $\pi$
  moreover obtain op where ops = [op]
    using assms(2) calculation
    by blast
  moreover have op $\in$ set (($\Pi\phi$))
    using assms(3) calculation
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  moreover have 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 $\exists \text{op. ops }= [\text{op}] \land \text{is-valid-operator-strips }\Pi \text{ op}$
    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 $\Pi \ ?\pi'$
  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 $k < \text{length } \pi$, the flattening of the parallel plan 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** execute-parallel-plan-is-execute-sequential-plan-if:

**fixes** \(I : (\text{variable, bool})\) state

**assumes** is-valid-problem II

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

**and** \(\forall k < \text{length } \pi.\)

\[\text{are-all-operators-applicable } (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k)\]
\[\wedge \text{are-all-operator-effects-consistent } (\pi ! k)\]
\[\wedge \text{are-all-operators-non-interfering } (\pi ! k)\]

**shows** \(G \subseteq_m \text{execute-serial-plan } I (\text{concat } \pi)\)

**using** assms

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

**case** \((\text{Cons } \text{ops } \pi)\)

let \(?ops' = \text{take } (\text{length } \text{ops}) (\text{concat } (\text{ops } \# \pi))\)

let \(?J = \text{execute-parallel-operator } I \text{ops}\)

and \(?J' = \text{execute-serial-plan } I ?ops'\)

\{ have \(\text{trace-parallel-plan-strips } I \pi ! 0 = I \text{ and } (\text{ops } \# \pi) ! 0 = \text{ops}\)

unfolding \(\text{trace-parallel-plan-strips-head-is-initial-state}\)

by 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 Cons.prems(3)

by auto+

then have \(\text{trace-parallel-plan-strips } I (\text{ops } \# \pi)\)

\[= I \# \text{trace-parallel-plan-strips } ?J \pi\]

by fastforce

\}

**note** \(nb = \text{this}\)

\{ have \(\text{last } (\text{trace-parallel-plan-strips } I (\text{ops } \# \pi))\)

= \(\text{last } (\text{trace-parallel-plan-strips } ?J \pi)\)

using \(\text{trace-parallel-plan-strips-last-cons-then } nb\)

by metis

**hence** \(G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } ?J \pi)\)

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 \( G \subseteq \text{m} \ execute-serial-plan \ ?J \ (\text{concat} \ \pi) \)
    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 \( ?\text{case} \)
    by argo
qed force

3.4 Auxiliary lemmas about STRIPS

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

definition is-valid-operator-sas-plus :: ('variable, 'domain) sas-plus-problem ⇒ ('variable, 'domain) sas-plus-operator ⇒ bool
  where is-valid-operator-sas-plus Ψ op ≡ let
    pre = precondition-of op
```
\[ \text{eff} = \text{effect-of op} \]
\[ \text{vs} = \text{variables-of } \Psi \]
\[ D = \text{range-of } \Psi \]

\[ \text{in list-all } (\lambda(v, a). \text{ListMem } v \text{ vs } \text{pre}) \]
\[ \land \text{list-all } (\lambda(v, a). (D v \neq \text{None}) \land \text{ListMem } a \text{ (the } (D v))) \text{ pre} \]
\[ \land \text{list-all } (\lambda(v, a). \text{ListMem } v \text{ vs } \text{eff}) \]
\[ \land \text{list-all } (\lambda(v, a). (D v \neq \text{None}) \land \text{ListMem } a \text{ (the } (D v))) \text{ eff} \]
\[ \land \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \lor a = a') \text{ pre} \]
\[ \land \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \lor a = a') \text{ eff}) \]

**definition** \( \text{is-valid-problem-sas-plus } \Psi \)
\[ \equiv \text{let } \text{ops} = \text{operators-of } \Psi \]
\[ \text{vs} = \text{variables-of } \Psi \]
\[ I = \text{initial-of } \Psi \]
\[ G = \text{goal-of } \Psi \]
\[ D = \text{range-of } \Psi \]

\[ \text{in list-all } (\lambda(v, D v \neq \text{None}) \text{ vs} \]
\[ \land \text{list-all } \text{is-valid-operator-sas-plus } \Psi \text{ ops} \]
\[ \land (\forall v. I v \neq \text{None } \leftrightarrow \text{ListMem } v \text{ vs}) \]
\[ \land (\forall v. G v \neq \text{None } \rightarrow \text{ListMem } (\text{the } (I v)) \text{ (the } (D v))) \]
\[ \land (\forall v. G v \neq \text{None } \rightarrow \text{ListMem } (\text{the } (G v)) \text{ (the } (D v))) \]

**definition** \( \text{is-operator-applicable-in} \)
\[ :: (\text{variable, 'domain}) \text{ state} \]
\[ \Rightarrow (\text{variable, 'domain}) \text{ sas-plus-operator} \]
\[ \Rightarrow \text{bool} \]

**where** \( \text{is-operator-applicable-in } s \text{ op} \)
\[ \equiv \text{map-of } \text{(precondition-of } \text{op}) \subseteq_m s \]

**definition** \( \text{execute-operator-sas-plus} \)
\[ :: (\text{variable, 'domain}) \text{ state} \]
\[ \Rightarrow (\text{variable, 'domain}) \text{ sas-plus-operator} \]
\[ \Rightarrow (\text{variable, 'domain}) \text{ state} \text{ (infixl } > \! > \text{ 52}) \]

**where** \( \text{execute-operator-sas-plus } s \text{ op} \equiv s ++ \text{map-of } \text{(effect-of } \text{op}) \)

— Set up simp rules to keep use of local parameters transparent within proofs (i.e. automatically substitute definitions).

**lemma**[simp]:
\[ \text{is-operator-applicable-in } s \text{ op} = (\text{map-of } \text{(precondition-of } \text{op}) \subseteq_m s) \]
\[ s > \! > \text{ op} = s ++ \text{map-of } \text{(effect-of } \text{op}) \]

**unfolding** initial-of-def goal-of-def variables-of-def range-of-def operators-of-def

\[ \text{SAS-Plus-Representation.is-operator-applicable-in-def} \]
\[ \text{SAS-Plus-Representation.execute-operator-sas-plus-def} \]

**by** simp+

**lemma** range-of-not-empty:
\textbf{proof}

\[ \langle \mathcal{R}_+ \rightleftharpoons \Psi \rangle v \neq \text{None} \land \mathcal{R}_+ \rightleftharpoons \Psi v \neq \text{Some} \]

\[ \langle \mathcal{R}_+ \rightleftharpoons \Psi \rangle v \neq \{ \}
\]

\textbf{apply} (cases \mathcal{R}_+ \rightleftharpoons \Psi v)

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

\textbf{lemma} \textbf{is-valid-operator-sas-plus-then}

\textbf{fixes} \Psi::\langle \text{v',d} \rangle \text{ sas-plus-problem}

\textbf{assumes} \textbf{is-valid-operator-sas-plus} \textbf{ op}

\textbf{shows} \forall (v, a) \in \text{set} \left( \text{precondition-of} \text{ op} \right). \ v \in \text{set} \left( (\Psi)_{\mathcal{V}_+} \right)

\text{and} \ 
\forall (v, a) \in \text{set} \left( \text{precondition-of} \text{ op} \right). \ (\mathcal{R}_+ \rightleftharpoons \Psi v) \neq \{ \} \land a \in \mathcal{R}_+ \Psi v

\text{and} \ 
\forall (v, a) \in \text{set} \left( \text{effect-of} \text{ op} \right). \ v \in \text{set} \left( (\Psi)_{\mathcal{V}_+} \right)

\text{and} \ 
\forall (v, a) \in \text{set} \left( \text{effect-of} \text{ op} \right). \ (\mathcal{R}_+ \rightleftharpoons \Psi v) \neq \{ \} \land a \in \mathcal{R}_+ \Psi v

\text{and} \ 
\forall (v, a) \in \text{set} \left( \text{precondition-of} \text{ op} \right). \ \forall (v', a') \in \text{set} \left( \text{precondition-of} \text{ op} \right). \ v \neq v' \lor a = a'

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

\textbf{proof --}

\textbf{let} \ ?v := \mathcal{R}_+ \rightleftharpoons \Psi \text{ variables-of} \Psi

\text{and} \ 
\?v := \mathcal{R}_+ \rightleftharpoons \Psi \text{ variables-of} \Psi

\text{and} \ 
\?D := \mathcal{R}_+ \rightleftharpoons \Psi \text{ range-of} \Psi

\textbf{have} \forall (v, a) \in \text{set} \ ?v \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v

\text{using} \text{assms}

\textbf{unfolding} \textbf{ is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff}

\textbf{by} \textbf{meson+}

\textbf{moreover have} \forall (v, a) \in \text{set} \ ?v \in \text{set} \ ((\Psi)_{\mathcal{V}_+})

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v \in \text{set} \ ((\Psi)_{\mathcal{V}_+})

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v \in \text{set} \ ?v \lor a = a'

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v \in \text{set} \ ?v \lor a = a'

\text{using} \textbf{calculation}

\textbf{unfolding} \textbf{variables-of-def}

\textbf{by} \textbf{blast+}

\textbf{moreover} \{ \email{61} \}

\textbf{have} \forall (v, a) \in \text{set} \ ?v \in \text{set} \ ((\Psi)_{\mathcal{V}_+})

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v \in \text{set} \ ?v \lor a = a'

\text{and} \ 
\forall (v, a) \in \text{set} \ ?v \in \text{set} \ ?v \lor a = a'

\text{using} \textbf{calculation}

\textbf{unfolding} \textbf{is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff}

\textbf{by} \textbf{argo}
lemma is-valid-problem-sas-plus-then:
fixes ⦃′v,′d⦄ sas-plus-problem
assumes is-valid-problem-sas-plus ⦃Ψ⦄
shows ∀ ⦃v⦄ ⦃Ψ⦄ ≠ {} 
  and ∀ ⦃op⦄ ⦃Ψ⦄ ≠ {} 
  and dom ⦃op⦄ = set ⦃op⦄ 
  and dom ⦃Ψ⦄ ⊆ set ⦃Ψ⦄ 
  and ∀ ⦃v⦄ ⦃Ψ⦄ ≠ {} 

proof –
let ⦃vs⦄ = 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 [] ←→ ((⦃R+⦄ ⦃Ψ⦄ ⦃v⦄) ≠ {}) 
  
  by (cases ⦃D⦄ ⦃v⦄; (auto simp: range-of′-def)) 
} note nb = this 

have nb1: ∀ ⦃v⦄ ⦃Ψ⦄ ≠ None 
  and ∀ ⦃op⦄ ⦃Ψ⦄ ≠ {} 
  and ⦃I⦄ ⦃v⦄ ≠ None = (⦃v⦄ ⦃Ψ⦄ ⦃v⦄) 
  and nb2: ∀ ⦃v⦄ ⦃I⦄ ⦃v⦄ ≠ None → the (⦃I⦄ ⦃v⦄ ⦃v⦄)
and \( \forall v. \ ?G v \neq None \rightarrow v \in \text{set } ?v \)

and \( \text{nb3: } \forall v. \ ?G v \neq None \rightarrow \text{the } (?G v) \in \text{set } (\text{the } (?D v)) \)

using assms

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

by argo+

then have \( G3: \forall op \in \text{set } ((\Psi)_{\sigma^+}). \ \text{is-valid-operator-sas-plus } \Psi \ op \)

and \( G4: \ \text{dom } ((\Psi)_{I^+}) = \text{set } ((\Psi)_{V^+}) \)

and \( G5: \ \text{dom } ((\Psi)_{G^+}) \subseteq \text{set } ((\Psi)_{V^+}) \)

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

by auto+

moreover {
  fix \( v \)
  assume \( v \in \text{set } ((\Psi)_{V^+}) \)
  then have \( ?D v \neq None \)
    using \( \text{nb}_1 \)
    by force+
} note \( G6 = \text{this} \)

moreover {
  fix \( v \)
  assume \( v \in \text{dom } ((\Psi)_{I^+}) \)
  moreover have \( ((\Psi)_{I^+}) v \neq None \)
    using calculation
    by blast+
  moreover {
    have \( v \in \text{set } ((\Psi)_{V^+}) \)
      using \( G4 \) calculation(1)
      by argo
    then have \( \text{sas-plus-problem.range-of } \Psi v \neq None \)
      using \( \text{range-of-not-empty} \)
      unfolding \( \text{range-of'-def} \)
      using \( G6 \)
      by fast+
    hence \( \text{set } (\text{the } (?D v)) = R^+ \Psi v \)
      by (simp add: \( \text{sas-plus-problem.range-of } \Psi v \neq None \) option.case-eq-if \( \text{range-of'-def} \))
  }

  ultimately have \( (\text{the } ((\Psi)_{I^+}) v) \in R^+ \Psi v \)
    using \( \text{nb}_2 \)
    by force
}

moreover {
  fix \( v \)
  assume \( v \in \text{dom } ((\Psi)_{G^+}) \)
  then have \( ((\Psi)_{G^+}) v \neq None \)
    by blast
  moreover {
    have \( v \in \text{set } ((\Psi)_{V^+}) \)
      using \( G5 \) calculation(1)

63
by fast
then have sas-plus-problem.range-of Ψ v ≠ None
using range-of-not-empty
using G6
by fast+
hence set (the (?D v)) = R_+ Ψ v
by (simp add: sas-plus-problem.range-of Ψ v ≠ None \ option.case-eq-if
range-of'-'def)
}
ultimately have the (((Ψ)_{G+}) v) ∈ R_+ Ψ v
using \ nb_{3}
by auto
}
ultimately show ∀ v ∈ set ((Ψ)_{V+}). (R_+ Ψ v) ≠ {}
and ∀ op ∈ set((Ψ)_{O+}). is-valid-operator-sas-plus Ψ op
and dom ((Ψ)_{I+}) = set ((Ψ)_{V+})
and ∀ v ∈ dom ((Ψ)_{I+}). the (((Ψ)_{I+}) v) ∈ R_+ Ψ v
and dom ((Ψ)_{G+}) ⊆ set ((Ψ)_{V+})
and ∀ v ∈ dom ((Ψ)_{G+}). the (((Ψ)_{G+}) v) ∈ R_+ Ψ 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
  | execute-serial-plan-sas-plus s (op # ops)
  = (if is-operator-applicable-in s op
   then execute-serial-plan-sas-plus (execute-operator-sas-plus s op) ops
   else s)

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

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

64
where is-serial-solution-for-problem $\Psi$ $\psi$

$\equiv$ let

$I = \text{sas-plus-problem}.\text{initial-of} \; \Psi$
$; \; G = \text{sas-plus-problem}.\text{goal-of} \; \Psi$
$; \; \text{ops} = \text{sas-plus-problem}.\text{operators-of} \; \Psi$
in $G \subseteq_m \text{execute-serial-plan-sas-plus} \; I \; \psi$
$\land \; \text{list-all} (\lambda op. \text{ListMem op ops} \; \psi$

context
begin

private lemma execute-operator-sas-plus-effect-i:
assumes is-operator-applicable-in $s \; \text{op}$
and $\forall (v, a) \in \text{set (effect-of op)}. \forall (v', a') \in \text{set (effect-of op)}.$
$\quad v \neq v' \lor a = a'$
and $(v, a) \in \text{set (effect-of op)}$
shows $\langle s \triangleright_+, \text{op} \rangle \; v = \text{Some a}$
proof –
let $\; ?\text{effect} = \text{effect-of op}$
have $\; \text{map-of} \; ?\text{effect} \; v = \text{Some a}$
using $\; \text{map-of-constant-assignments-defined-if} \; [\text{OF assms(2, 3)}]$ try0
by blast
thus $\text{?thesis}$
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 $s \; \text{op}$
and $\forall (v', a') \in \text{set (effect-of op)}. \; v' \neq v$
shows $\langle s \triangleright_+, \text{op} \rangle \; v = s \; v$
proof –
let $\; ?\text{effect} = \text{effect-of op}$

{ have $\; v \notin \text{fst \; set \; ?\text{effect}}$
using assms(2)
by fastforce
then have $\; v \notin \text{dom (map-of \; ?\text{effect})}$
using $\; \text{dom-map-of-conv-image-fst}[\text{of \; ?\text{effect}]$ by argo
hence $\; (s \; ++ \; \text{map-of \; ?\text{effect}}) \; v = s \; v$
using $\; \text{map-add-dom-app-simps(3)}[\text{of \; v \; map-of \; ?\text{effect \; s}]$
by blast
}
thus $\text{?thesis}$
by fastforce
qed

Given an operator $\text{op}$ 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:

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

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 \( s \ \text{op} \)

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

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

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

\[ \rightarrow (s \gg_+ \text{op}) \cdot v = \text{Some } a \]

\[ \rightarrow (s \gg_+ \text{op}) \cdot v = s \cdot v \]

**Proof**

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

\[ \rightarrow (s \gg_+ \text{op}) \cdot v = \text{Some } a \]

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

by blast

**Next**

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

\[ \rightarrow (s \gg_+ \text{op}) \cdot v = s \cdot v \]

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

by blast

**Qed**

**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** ('variable, 'domain) sas-plus-parallel-plan

\[ = ('variable, 'domain) sas-plus-operator list list \]

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

\[ :: ('variable, 'domain) state \]

\[ \Rightarrow ('variable, 'domain) sas-plus-operator list \]

\[ \Rightarrow \text{bool} \]

where are-all-operators-applicable-in \( s \ \text{ops} \)

\[ \equiv \text{list-all } (\text{is-operator-applicable-in } s) \ \text{ops} \]

**Definition** are-operator-effects-consistent

\[ :: ('variable, 'domain) sas-plus-operator \]

\[ \Rightarrow ('variable, 'domain) sas-plus-operator \]
\[ \text{bool} \]

\textbf{where} are-operator-effects-consistent op op'
\[ \equiv \text{let} \]
\[ \text{effect} = \text{effect-of op} \]
\[ ; \text{effect'} = \text{effect-of op'} \]
\[ \text{in} \text{list-all} (\lambda (v, a). \text{list-all} (\lambda (v', a'). v \neq v' \lor a = a') \text{effect}) \text{effect} \]

\textbf{definition} are-all-operator-effects-consistent
\[ :: (\text{variable, 'domain}) \text{sas-plus-operator list} \Rightarrow \text{bool} \]
\[ \text{where} \text{are-all-operator-effects-consistent} \text{ops} \]
\[ \equiv \text{list-all} (\lambda (\text{op}). \text{list-all} (\lambda (\text{op'}). \text{are-operator-effects-consistent op} \text{ops}) \text{ops}) \text{ops} \]

\textbf{definition} execute-parallel-operator-sas-plus
\[ :: (\text{variable, 'domain}) \text{state} \Rightarrow (\text{variable, 'domain}) \text{sas-plus-operator list} \Rightarrow (\text{variable, 'domain}) \text{state} \]
\[ \text{where} \text{execute-parallel-operator-sas-plus} \text{s} \text{ops} \]
\[ \equiv \text{foldl} (++) \text{s} (\text{map} (\text{map-of} \circ \text{effect-of}) \text{ops}) \]

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 ??).

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

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

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 $ops$ 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 $\Psi \psi$

\[
\equiv let \\
G = sas-plus-problem.goal-of $\Psi$ \\
I = sas-plus-problem.initial-of $\Psi$ \\
Ops = sas-plus-problem.operators-of $\Psi$ \\
in G \subseteq m execute-parallel-plan-sas-plus $I \psi$ \\
\land list-all ($\lambda ops.\ list-all ($\lambda op.\ listMem op Ops$) ops$) $\psi$
\]

**context**

**begin**

**lemma** execute-parallel-operator-sas-plus-cons[simp]:

execute-parallel-operator-sas-plus $s$ ($op \neq ops$)

\[= execute-parallel-operator-sas-plus (s ++ map-of (effect-of op)) ops\]

**unfolding** execute-parallel-operator-sas-plus-def

**by** simp

The following lemmas show the properties of SAS+ parallel plan execution traces. The results are analogous to those for STRIPS. So, let $\tau \equiv trace-parallel-plan-sas-plus I \psi$ 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 $k < length \tau - 1$—the parallel operator $\pi ! k$ is executable (lemma ??); and finally,
- for all $k < length \tau$, the parallel execution of the plan prefix take $k \psi$ with initial state $I$ equals the $k$-th element of the trace $\tau ! k$ (lemma ??).

**lemma** trace-parallel-plan-sas-plus-head-is-initial-state:

trace-parallel-plan-sas-plus $I \psi ! 0 = I$

**proof** (cases $\psi$)

**case** (Cons a list)

**then** show $?thesis
by (cases are-all-operators-applicable-in I a \land are-all-operator-effects-consistent a;
  simp+)
qed simp

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

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

lemma plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements:
assumes k < length (trace-parallel-plan-sas-plus I \psi) - 1
obtains ops \psi' where \psi = ops \# \psi'
proof
  let ?\tau = trace-parallel-plan-sas-plus I \psi
  have length ?\tau \leq length \psi + 1
    using length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one
    by fast
  then have 0 < length \psi
using trace-parallel-plan-sas-plus-length-gt-one-if[OF assms]
by fastforce
then obtain k' where length ψ = Suc k'
  using gr0-implies-Suc
by meson
thus ?thesis using that
  using length-Suc-conv[of ψ k']
by blast
qed

lemma trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds:
  assumes k < length (trace-parallel-plan-sas-plus π) - 1
  shows are-all-operators-applicable-in (trace-parallel-plan-sas-plus π ! k) (π ! k)
    ∧ are-all-operator-effects-consistent (π ! k)
using assms
proof (induction π arbitrary: I k)
  — NOTE Base case yields contradiction with assumption and can be left to automation.
case (Cons a π)
then show ?case
  proof (cases are-all-operators-applicable-in I a ∧ are-all-operator-effects-consistent a)
case True
  have trace-parallel-plan-sas-plus-cons: trace-parallel-plan-sas-plus I (a # π)
    = I # trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) π
  using True
  by simp
then show ?thesis
  proof (cases k)
case 0
  have trace-parallel-plan-sas-plus I (a # π) ! 0 = I
  using trace-parallel-plan-sas-plus-cons
  by simp
moreover have (a # π) ! 0 = a
  by simp
ultimately show ?thesis
  using True 0
  by presburger
next
case (Suc k')
  have trace-parallel-plan-sas-plus I (a # π) ! Suc k'
    = trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) π ! k'
  using trace-parallel-plan-sas-plus-cons
  by simp
moreover have (a # π) ! Suc k' = π ! k'
  by simp
moreover {
  let ?I' = execute-parallel-operator-sas-plus I a
  have length (trace-parallel-plan-sas-plus I (a # π))
= 1 + \text{length (trace-parallel-plan-sas-plus } ?I' \pi) \\
\text{using trace-parallel-plan-sas-plus-cons} \\
\text{by auto} \\
\text{then have } k' < \text{length (trace-parallel-plan-sas-plus } ?I' \pi) - 1 \\
\text{using Cons.prems Suc} \\
\text{unfolding Suc-eq-plus1} \\
\text{by fastforce} \\
\text{hence are-all-operators-applicable-in} \\
\text{(trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) } \pi ! k') \\
\text{(} \pi ! k'\text{)} \\
\text{\wedge are-all-operator-effects-consistent (} \pi ! k') \\
\text{using Cons.IH [of k' execute-parallel-operator-sas-plus I a] Cons.prems} \\
\text{Suc trace-parallel-plan-sas-plus-cons} \\
\text{by simp} \\
\text{)} \\
\text{ultimately show } \text{thesis} \\
\text{using Suc} \\
\text{by argo} \\
\text{qed} \\
\text{next} \\
\text{case False} \\
\text{then have } \text{trace-parallel-plan-sas-plus I (a } \# \pi) = [I] \\
\text{by force} \\
\text{then have } \text{length (trace-parallel-plan-sas-plus I (a } \# \pi)) - 1 = 0 \\
\text{by simp} \\
\text{— NOTE Thesis follows from contradiction with assumption.} \\
\text{then show } \text{thesis} \\
\text{using Cons.prems} \\
\text{by force} \\
\text{qed} \\
\text{lemma trace-parallel-plan-sas-plus-prefix:} \\
\text{assumes } k < \text{length (trace-parallel-plan-sas-plus I } \psi) \\
\text{shows } \text{trace-parallel-plan-sas-plus I } \psi ! k = \text{execute-parallel-plan-sas-plus I (take k } \psi) \\
\text{using assms} \\
\text{proof (induction } \psi \text{ arbitrary: I k)} \\
\text{case (Cons a } \psi) \\
\text{then show } \text{thesis} \\
\text{proof (cases are-all-operators-applicable-in I a } \wedge \text{ are-all-operator-effects-consistent a)} \\
\text{case True} \\
\text{let } ?\sigma = \text{trace-parallel-plan-sas-plus I (a } \# \psi) \\
\text{and } ?I' = \text{execute-parallel-operator-sas-plus I a} \\
\text{have } \sigma\text{-equals: } (?\sigma = I \# \text{trace-parallel-plan-sas-plus } ?I' \psi) \\
\text{using True} \\
\text{by auto} \\
\text{then show } \text{thesis} \\
\text{qed}
proof (cases \( k = 0 \))

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

lemma \text{trace-parallel-plan-sas-plus-step-effect-is}: 
assumes \( k < \text{length} \ (\text{trace-parallel-plan-sas-plus} \ I \ \psi) - 1 \)
shows \( \text{trace-parallel-plan-sas-plus} \ I \ \psi \ ! \text{Suc} \ k \)
  \( = \text{execute-parallel-operator-sas-plus} \ (\text{trace-parallel-plan-sas-plus} \ I \ \psi \ ! k) \ (\psi \ ! k) \)
proof –
let \( ?\tau = \text{trace-parallel-plan-sas-plus} \ I \ \psi \)
let \( ?\tau_k = ?\tau \ k \)
and \( ?\tau_k' = ?\tau \ \text{Suc} \ k \)

— 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 asms
    by linarith
  hence \( ?\tau_k' = \text{execute-parallel-plan-sas-plus} \ I \ (\text{take} \ (\text{Suc} \ k) \ \psi) \)
    using trace-parallel-plan-sas-plus-prefix[of Suc \( k \)]
    by blast
  } note rewrite-goal = this
have execute-parallel-plan-sas-plus \ I \ (\text{take} \ (\text{Suc} \ k) \ \psi)
  = \text{execute-parallel-operator-sas-plus} \ (\text{trace-parallel-plan-sas-plus} \ I \ ?\tau \ k) \ (\psi \ k)
  using asms
proof (induction \( k \) arbitrary: \( I \ \psi \))
case 0
  obtain ops \( \psi' \) where \( \psi\)-is: \( \psi = \text{ops} \# \psi' \)
    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 \ (\text{take} \ (\text{Suc} \ 0) \ \psi)
      = \text{execute-parallel-plan-sas-plus} \ I \ [\psi \ 0]
      by argo
  }
moreover { 
  have trace-parallel-plan-sas-plus \ I \ ?\tau \ 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]
      = \text{execute-parallel-operator-sas-plus} \ I \ (\psi \ 0)
      by simp
  }
}
ultimately have execute-parallel-operator-sas-plus \ (\text{trace-parallel-plan-sas-plus} \ I \ ?\tau \ 0)
  (\psi \ 0)
  = \text{execute-parallel-plan-sas-plus} \ I \ [\psi \ 0]
  by argo
ultimately show \( \text{?case} \)
by argo

next

\( \text{case } (\text{Suc } k) \)

obtain \( \text{ops } \psi' \) where \( \psi': \text{ops } \# \psi' \)

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

by blast

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

have \( \text{execute-parallel-plan-sas-plus } \text{I } \) (\( \text{take } (\text{Suc } (\text{Suc } k)) \) \( \psi' \))

= \( \text{execute-parallel-plan-sas-plus } \text{?I'} \) (\( \text{take } (\text{Suc } k) \) \( \psi' \))

using Suc.prems \( \psi'-\text{is} \)
by fastforce

moreover {

thm Suc.IH[of ]

have length (\( \text{trace-parallel-plan-sas-plus } \text{I } \psi' \))

\( = 1 + \) length (\( \text{trace-parallel-plan-sas-plus } \text{?I'} \) \( \psi' \))

using \( \psi'-\text{is} \) Suc.prems
by fastforce

moreover have \( \psi' = \text{Suc} . \text{prems calculation} \)
by fastforce

ultimately have execute-parallel-plan-sas-plus \( \text{?I'} \) (\( \text{take } (\text{Suc } k) \) \( \psi' \))

\( = \text{execute-parallel-operator-sas-plus } \text{(trace-parallel-plan-sas-plus } \text{?I'} \) \( \psi' \) ! \( k \))

using Suc.IH[of \( \text{?I'} \) \( \psi' \)]

by blast

}

moreover have execute-parallel-operator-sas-plus (\( \text{trace-parallel-plan-sas-plus } \text{?I'} \) \( \psi' \) ! \( k \))

\( \leq \text{execute-parallel-operator-sas-plus } \text{(trace-parallel-plan-sas-plus } \text{I } \psi' \) ! \( \text{Suc } k \))

\( \leq \) \( \text{Suc} . \text{prems } \psi'-\text{is} \)
by auto

ultimately show \( \text{?case} \)
by argo

qed

thus \( \text{?thesis} \)

using rewrite-goal
by argo

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.

\textbf{lemma} execute-parallel-plan-sas-plus-reaches-goal-iff-goal-is-last-element-of-trace:
\[ G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi \]
\[ \iff G \subseteq_m \text{last (trace-parallel-plan-sas-plus } I \psi) \]

**proof**

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

**proof (rule iffI)**

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

**proof (induction } \psi \text{ arbitrary; } I)\)**

- NOTE Base case follows from simplification.

**case (Cons } ops \psi)\)**

- show \( ?\text{case} \)

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

  **case True**

  - let \( ?s = \text{execute-parallel-operator-sas-plus } I \text{ ops} \right)\)
  
  ```
  \{ 
  \begin{align*}
  & \text{have } G \subseteq_m \text{execute-parallel-operator-sas-plus } ?s \psi \\
  & \text{using } \text{True Cons.prems} \\
  & \text{by simp} \\
  & \text{hence } G \subseteq_m \text{last (trace-parallel-plan-sas-plus } ?s \psi) \\
  & \text{using } \text{Cons.IH} \\
  & \text{by auto}
  \end{align*}
  \}
  ```

  **moreover {**

  ```
  \begin{align*}
  & \text{have } \text{trace-parallel-plan-sas-plus } I \text{ (ops } \# \psi) \\
  & \right \text{=} I \right \text{# trace-parallel-plan-sas-plus } ?s \psi \\
  & \text{using } \text{True} \\
  & \text{by simp} \\
  & \text{moreover have } \text{trace-parallel-plan-sas-plus } ?s \psi \right \text{=} [] \\
  & \text{using } \text{trace-parallel-plan-sas-plus.elims} \\
  & \text{by blast} \\
  & \text{ultimately have } \text{last (trace-parallel-plan-sas-plus } I \text{ (ops } \# \psi)) \\
  & \right \text{=} \text{last (trace-parallel-plan-sas-plus } ?s \psi) \\
  & \text{using } \text{last-ConsR} \\
  & \text{by simp}
  \end{align*}
  ```

  **}**

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

  ```
  \text{by argo}
  ```

  **next**

  **case False**

  then have \( G \subseteq_m I \)

  ```
  \text{using Cons.prems} \\
  \text{by force} \\
  ```

  thus \( ?\text{thesis} \)

  ```
  \text{using False} \\
  \text{by force}
  ```

  **qed**

  **qed force**

75
next
assume $G \subseteq_m \text{last } \tau$
thus $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi$

proof (induction $\psi$ arbitrary; $I$)
case $(\text{Cons ops } \psi)$
thus $?\text{case}$
proof (cases are-all-operators-applicable-in $I$ ops
& are-all-operator-effects-consistent ops)
case $\text{True}$
let $?s = \text{execute-parallel-operator-sas-plus } I \text{ ops}$
{

have $\text{trace-parallel-plan-sas-plus } I (\text{ops } # \psi)$
$= I \neq \text{trace-parallel-plan-sas-plus } ?s \psi$
using $\text{True}$
by simp

moreover have $\text{trace-parallel-plan-sas-plus } ?s \psi \neq []$
using $\text{trace-parallel-plan-sas-plus.elsms}$
by blast

ultimately have $\text{last } (\text{trace-parallel-plan-sas-plus } I (\text{ops } # \psi))$
$= \text{last } (\text{trace-parallel-plan-sas-plus } ?s \psi)$
using $\text{last-ConsR}$
by simp

hence $G \subseteq_m \text{execute-parallel-plan-sas-plus } ?s \psi$
using $\text{Cons.IH[of } ?s] Cons.prems}$
by argo
}
moreover have $\text{execute-parallel-plan-sas-plus } I (\text{ops } # \psi)$
$= \text{execute-parallel-plan-sas-plus } ?s \psi$
using $\text{True}$
by force
ultimately show $?\text{thesis}$
by argo
next
case $\text{False}$

have $G \subseteq_m I$
using $\text{Cons.prems False}$
by simp
thus $?\text{thesis}$
using $\text{False}$
by force

qed
qed simp
qed
qed

lemma is-parallel-solution-for-problem-plan-operator-set:

fixes $\Psi :: (\langle v, d \rangle \text{ sas-plus-problem})$
assumes $\text{is-parallel-solution-for-problem } \Psi \psi$
5.3 Serializable Parallel Plans

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

1. the embedding $\text{List-Supplement}.\text{embed} \psi$ of $\psi$ is a parallel solution for $\Psi$ (lemma ??); and conversely that,

2. a parallel solution to $\Psi$ that has the form of an embedded serial plan can be concatenated to obtain a serial solution (lemma ??).
shows $G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\text{embed } \psi)$
using assms
proof (induction $\psi$ arbitrary: $I$)
case (Cons op $\psi$)
show $?case$
proof (cases are-all-operators-applicable-in $I [\text{op}]$)
case True
let $?J = \text{execute-operator-sas-plus } I \text{ op}$
let $?J' = \text{execute-parallel-operator-sas-plus } I [\text{op}]$
have $\text{SAS-Plus-Representation. is-operator-applicable-in } I \text{ op}$
using True
unfolding are-all-operators-applicable-in-def list-all-iff
by force
moreover have $G \subseteq_m \text{execute-serial-plan-sas-plus } ?J \psi$
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 $\text{execute-parallel-plan-sas-plus } I ([\text{op]} \# \text{ embed } \psi)$
$= \text{execute-parallel-plan-sas-plus } ?J' (\text{embed } \psi)$
using True calculation(3)
by simp
moreover {
have is-operator-applicable-in $I \text{ op}$
are-operator-effects-consistent $op \text{ op}$
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 $\neg$is-operator-applicable-in $I \text{ op}$
using calculation
unfolding are-all-operators-applicable-in-def
SAS-Plus-Representation. is-operator-applicable-in-def list-all-iff
by fastforce
moreover have $G \subseteq_m I$
using Cons.prems(2) calculation(2)
unfolding is-operator-applicable-in-def
by simp
moreover have \( \text{execute-parallel-plan-sas-plus} \ I \ ([\text{op}] \ # \ \text{embed} \ \psi) = I \)
using calculation(1)
by fastforce
ultimately show \( \text{thesis} \)
by force
qed
qed simp

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

lemma \( \text{execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iv} \):
fixes \( \Psi :: (\vdash \nu, \text{id}) \ \text{sas-plus-problem} \)
assumes \( \forall \ \text{op} \in \text{set} \ \psi. \ \text{op} \in \text{set} \ ((\Psi)_{\bigoplus}^{+}) \)
shows \( \forall \ \text{ops} \in \text{set} \ (\text{embed} \ \psi). \ \forall \ \text{op} \in \text{set} \ \text{ops}. \ \text{op} \in \text{set} \ ((\Psi)_{\bigoplus}^{+}) \)
proof –
let \( \Psi' = \text{embed} \ \psi \)
have \( \text{nb: set} \ \Psi' = \{ \{\text{op}\} | \ \text{op} \in \text{set} \ \psi \} \)
by (induction \( \psi \); force)
{
fix \( \text{ops} \)
assume \( \text{ops} \in \text{set} \ \Psi' \)
moreover obtain \( \text{op} \ \text{where} \ \text{ops} = \{\text{op}\} \ \text{and} \ \text{op} \in \text{set} \ ((\Psi)_{\bigoplus}^{+}) \)
using assms(1) nb calculation
by blast
ultimately have \( \forall \ \text{op} \in \text{set} \ \text{ops}. \ \text{op} \in \text{set} \ ((\Psi)_{\bigoplus}^{+}) \)
by fastforce
}
thus \( \text{thesis} \)
qed

theorem \( \text{execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus} \):
assumes is-valid-problem-sas-plus $\Psi$
and is-serial-solution-for-problem $\Psi$ $\psi$
shows is-parallel-solution-for-problem $\Psi$ ($\text{embed } \psi$)

proof

let $?ops = \text{sas-plus-problem.operators-of } \Psi$
and $?\psi' = \text{embed } \psi$

\begin{verbatim}
\{ 
    thm execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii[OF]
    have $(\Psi)_{G+} \subseteq_m \text{execute-serial-plan-sas-plus }((\Psi)_{I+}) \psi$
        using assms(2)
        unfolding is-serial-solution-for-problem-def Let-def
        by simp
    moreover have $\forall op \in \text{set } \psi. \text{are-operator-effects-consistent op op}$
        using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iii[OF assms]..
    ultimately have $(\Psi)_{G+} \subseteq_m \text{execute-parallel-plan-sas-plus }((\Psi)_{I+}) \ ?\psi'$
        using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii
        by blast
\} 
moreover \{ 
    have $\forall op \in \text{set } \psi. \ op \in \text{set } ((\Psi)_{O+})$
        using assms(2)
        unfolding is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
        by fastforce
    hence $\forall ops \in \ ?\psi'. \forall op \in \text{set ops. op \in set } ((\Psi)_{O+})$
        using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iv
        by blast
\} 
ultimately show $\text{thesis}$
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff Let-def
    goal-of-def
    initial-of-def
    by fastforce
qed

lemma flattening-lemma-i:
fixes $\Psi :: (\ 'v, 'd) \text{sas-plus-problem}$
assumes $\forall ops \in \text{set } \pi. \forall op \in \text{set ops. op \in set } ((\Psi)_{O+})$
shows $\forall op \in \text{set } (\text{concat } \pi). \ op \in \text{set } ((\Psi)_{O+})$

proof

\{ 
    fix op
    assume $op \in \text{set } (\text{concat } \pi)$
    moreover have $op \in (\bigcup \text{ops} \in \text{set } \pi. \ \text{set ops})$
        using calculation
        unfolding set-concat.
    then obtain ops where $ops \in \text{set } \pi$ and $op \in \text{set ops}$
        using UN-iff
        by blast
    ultimately have $op \in \text{set } ((\Psi)_{O+})$
\}
lemma flattening-lemma-ii:
  fixes I :: ('variable, 'domain) state
  assumes \( \forall \text{ops} \in \text{set } \psi. \exists \text{op. } \text{ops} = [\text{op}] \land \text{is-valid-operator-sas-plus } \Psi \text{ op} \)
  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 \(?thesis\)
  using assms
proof (induction \( \psi \) arbitrary: \( I \))
  case (Cons ops \( \psi \))
  obtain op where \( \text{ops-is} \): \( \text{ops} = [\text{op}] \land \text{is-valid-op } \Psi \text{ op} \)
  using Cons.prems(1)
  by 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 True \( \text{ops-is} \)
    unfolding are-all-operators-applicable-in-def is-operator-applicable-in-def
    list-all-iff
    by force
    have \( \text{nb}_2 : \text{are-operator-effects-consistent } \text{op} \text{ op} \)
    unfolding are-operator-effects-consistent-def list-all-iff Let-def
    using is-valid-operator-sas-plus-then(6)\( |\text{OF } \text{is-valid-op} \)
    by blast
    have \( \text{are-all-operator-effects-consistent } \text{ops} \)
    using \( \text{ops-is} \)
    unfolding are-all-operator-effects-consistent-def list-all-iff
    using \( \text{nb}_2 \)
    by force
  moreover have \( G \subseteq m \text{ execute-parallel-plan-sas-plus } ?J \psi \)
  using Cons.prems(2) True calculation \( \text{ops-is} \)
  by 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 \( \text{ops-is } \text{nb}_1 \text{ is-operator-applicable-in-def} \)
  by simp
  moreover have \( ?J = ?J’ \)
  using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i\( |\text{OF} \)
by simp
ultimately show ?thesis
  using Cons.IH[of ?I] Cons.prems(1)
by force
next
case False
moreover have $G \subseteq m$ $I$
  using Cons.prems(2) calculation
by fastforce
moreover {
  have ¬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
qed

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. $op \in set ((\psi)_{(op)+})$
    using assms(3)
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
  hence $\forall$ $op \in set ?\psi'$. $op \in set ((\psi)_{(op)+})$
    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)G+)
  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)G+ ⊆m execute-parallel-plan-sas-plus ((Ψ)I+) ψ
  using assms(3)
  unfolding is-parallel-solution-for-problem-def
  by fastforce
ultimately have (Ψ)G+ ⊆m execute-serial-plan-sas-plus ((Ψ)I+) ?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

5.4 Auxiliary lemmata on SAS+

class 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

83
then have \((\mathcal{R}_+ \Psi v) \neq \{\}\) and \(a \in \mathcal{R}_+ \Psi v\)
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 \((\mathcal{R}_+ \Psi v) \neq \{\}\) and \(a \in \mathcal{R}_+ \Psi v\)
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 \(\Psi :: (\'v, \'d)\) sas-plus-problem
assumes is-valid-problem-sas-plus \(\Psi\)
\(v \in \text{set} ((\Psi)_V^+ )\)
shows set (the (sas-plus-problem.range-of \(\Psi\) \(v\))) = \(\mathcal{R}_+ \Psi v\)
proof−
have \(v \in \text{set} ((\Psi)_V^+ )\)
using assms(2)
unfolding variables-of-def.
moreover have \((\mathcal{R}_+ \Psi v) \neq \{\}\)
using assms(1) calculation is-valid-problem-sas-plus-then(1)
by blast
moreover have sas-plus-problem.range-of \(\Psi\) \(v\) \(\neq\) None
and sas-plus-problem.range-of \(\Psi\) \(v\) \(\neq\) 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 \(\Psi :: (\'v, \'d)\) sas-plus-problem
assumes is-valid-problem-sas-plus \(\Psi\)
shows \(\forall v \in \text{set} ((\Psi)_V^+ ). (\mathcal{R}_+ \Psi v) \neq \{\}\)
and \(\forall op \in \text{set} ((\Psi)_O^+ ). \text{is-valid-operator-sas-plus} \Psi op\)
and dom ((\Psi)_I^+ ) = set ((\Psi)_V^+ )
and $\forall v \in \text{dom} \ ((\Psi)_I \ v)$. the $(((\Psi)_I \ v)) \in \mathcal{R}_+ \ \Psi \ v$

and $\text{dom} \ ((\Psi)_{G_+}) \subseteq \text{set} \ ((\Psi)_{V_+})$

and $\forall v \in \text{dom} \ ((\Psi)_{G_+})$, the $(((\Psi)_{G_+}) \ v) \in \mathcal{R}_+ \ \Psi \ v$

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 (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 \text{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_\mathcal{S} - - 99)

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

definition sasp-op-to-strips
:: (\variable, \domain) sas-plus-problem
\Rightarrow (\variable, \domain) sas-plus-operator
\Rightarrow (\variable, \domain) assignment strips-operator

(\varphi_\mathcal{O} - - 99)

where sasp-op-to-strips \Psi op \equiv \text{let}
\text{pre} = \text{precondition-of} \ op

85
add = effect-of op
delete = \{(v, a'). (v, a) \leftarrow effect-of op, a' \leftarrow filter ((\neq) \ a) \ (the \ (range-of \ \Psi \ v))\}

in STRIPS-Representation.

\textbf{definition} \ \textit{sas-plus-problem-to-strips-problem}
:: \ ('variable,' domain) sas-plus-problem \Rightarrow \ ('variable,' domain) assignment strips-problem

(\varphi - 99)
\textbf{where} \ \textit{sas-plus-problem-to-strips-problem} \ \Psi \equiv let
\hspace{1em} vs = [as. v \leftarrow variables-of \ \Psi, as \leftarrow (possible-assignments-for \ \Psi) \ v]
\hspace{1em} ; \ \textit{ops} = \textit{map} (sasp-op-to-strips \ \Psi) (\textit{operators-of} \ \Psi)
\hspace{1em} ; \ I = \textit{state-to-strips-state} \ \Psi (\textit{initial-of} \ \Psi)
\hspace{1em} ; \ G = \textit{state-to-strips-state} \ \Psi (\textit{goal-of} \ \Psi)
\hspace{1em} in STRIPS-Representation.

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

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

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

\textbf{definition} \ \textit{strips-parallel-plan-to-sas-plus-parallel-plan}
:: \ ('variable,' domain) sas-plus-problem
⇒ (‘variable × ‘domain) strips-parallel-plan
⇒ (‘variable, ‘domain) sas-plus-parallel-plan

(ϕ_p⁻¹ - 99)

where strips-parallel-plan-to-sas-plus-parallel-plan Π π
≡ [[strips-op-to-sasp Π. op ← ops. ops ← π]

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.

definition range-of-strips Π x ≡ { True, False }

colorbox[gray]{0.1}{begin
— Set-up simp rules.

lemma[simp]:

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

and (ϕ_Σ s)
= (let defined = \text{filter (λv. s v ≠ None) (variables-of } \Psi) in
  \text{map-of (map (λ(v, a). (v, a), the (s v = a))
                  \text{concat [possible-assignments-for } \Psi. v ← \text{defined}])})

and (ϕ_O Ψ op)
= (let
  pre = \text{precondition-of op}
  ; add = \text{effect-of op}
  ; delete = [\{(v, a'), (v, a) ← \text{effect-of op, a' ← filter ((≠ a) (the (range-of } \Ψ
              \text{v)})
  in STRIPS-Representation.operator-for pre add delete)

and (ϕ_O Ψ ψ) = [ϕ_O Ψ op. op ← ops. ops ← ψ]
and (ϕ_S⁻¹ Ψ s') = \text{map-of (filter (λ(v, a). s' (v, a) = Some True)
  (all-possible-assignments-for } \Psi))
and (ϕ_O⁻¹ Ψ op') = (let
  \text{precondition = strips-operator.precondition-of op'}
  ; \text{effect = strips-operator.add-effects-of op'}
  in (\{ \text{precondition-of = precondition, effect-of = effect }\})

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

unfolding

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

by blast+

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

\begin{align*}
\text{lemma } \text{is-valid-problem-sas-plus-dom-sas-plus-problem-range-of':} \\
\text{assumes } \text{is-valid-problem-sas-plus } \Psi \\
\text{shows } \forall v \in \text{set } ((\Psi)_V). \ v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi) \\
\text{using } \text{assms}(I) \text{ is-valid-problem-sas-plus-then}(I) \\
\text{unfolding } \text{is-valid-problem-sas-plus-def} \\
\text{by } (\text{meson domIff list.pred-set}) \\
\end{align*}

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

\begin{align*}
\text{lemma } \text{all-possible-assignments-for-set-is:} \\
\text{assumes } \forall v \in \text{set } ((\Psi)_V). \ \text{range-of } \Psi v \neq \text{None} \\
\text{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 \}) \\
\text{proof} – \\
\text{let } \text{?vs} = \text{variables-of } \Psi \\
\text{have } \text{set } (\text{all-possible-assignments-for } \Psi) = \\
(\bigcup (\text{set } \cdot (\lambda v. \text{ map } (\lambda(v, a). (v, a)) (\text{possible-assignments-for } \Psi v)) \cdot \text{ set } \text{?vs})) \\
\text{unfolding } \text{all-possible-assignments-for-def set-concat} \\
\end{align*}
using set-map
by auto
also have \ldots = (\bigcup ((\lambda v. \text{set } (\text{possible-assignments-for } \Psi v)) \cdot \text{set } ?\text{vs}))
using image-comp set-map
by simp
also have \ldots = (\bigcup ((\lambda v. \{ (v, a) \mid a \in \mathcal{R}_+ \Psi v \}) \cdot \text{set } ?\text{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 \(\forall v \in \text{set } ((\Psi)_{\mathcal{V}+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)\)
shows \(\text{set } (\text{concat } \{ \text{possible-assignments-for } \Psi v. v \leftarrow \text{filter } (\lambda v. s v \neq \text{None }) (\text{variables-of } \Psi)\}) = (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{\mathcal{V}+}) \land s v \neq \text{None } \}. \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \}))\)
proof -
let ?\text{vs} = \text{variables-of } \Psi
let ?\text{defined} = \text{filter } (\lambda v. s v \neq \text{None }) ?\text{vs}
let ?l = concat \{ \text{possible-assignments-for } \Psi v. v \leftarrow ?\text{defined}\}
have nb: \text{set } ?\text{defined} = \{ v \mid v. v \in \text{set } ((\Psi)_{\mathcal{V}+}) \land s v \neq \text{None } \}\n  unfolding set-filter
by force
have \text{set } ?l = \bigcup (\text{set } \text{concat } \text{map } \{ \text{possible-assignments-for } \Psi \} ?\text{defined })\)
  unfolding set-concat image-Union
by blast
also have \ldots = \bigcup (\text{set } \text{concat } \text{map } \{ \text{possible-assignments-for } \Psi \} ?\text{defined })\)
  unfolding set-map
by blast
also have \ldots = (\bigcup v \in \text{set } ?\text{defined}. \text{set } (\text{possible-assignments-for } \Psi v))\)
by blast
also have \ldots = (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{\mathcal{V}+}) \land s v \neq \text{None } \}. \text{set } (\text{possible-assignments-for } \Psi 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

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 dom (\(\varphi_S \Psi s\))
\begin{align*}
= \left( \bigcup v \in \{ v \mid v, v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \} \right). \\
\{ (v, a) \mid a \in \mathcal{R}_+ \Psi v \})
\end{align*}

\textbf{proof} –

let \( \forall v \in \{ \text{variables-of } \Psi \} \)

let \( \forall l = \text{concat } \{ \text{possible-assignments-for } \Psi v, v \leftarrow \text{filter } (\lambda v, s v \neq \text{None}) \} \)

have \( \forall v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \}

using \isvalidproblemSasPlusDomSasPlusProblemRangeOfAssms(1)

by fastforce

have \( \text{dom } (\varphi_S \Psi s) = \text{fst } \{ \text{map } (\lambda (v, a), ((v, a), \text{the } (s v) = a)) \} \)

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

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

using \text{dom-map-of-cone-image-fst}[\text{of map } (\lambda (v, a), ((v, a), \text{the } (s v) = a)) \}

by presburger

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

unfolding \text{set-map}

by blast

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

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

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

by blast

finally show \( \forall v \in \text{set } \{ \text{variables-of } \Psi \} \) \in \text{dom } \( \varphi_S \Psi s \)

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

by force

\textbf{corollary} \text{state-to-strips-state-dom-element-iff:}

\textbf{assumes } \isvalidproblemSasPlus\Psi

\textbf{shows } \( (v, a) \in \text{dom } (\varphi_S \Psi s) \leftarrow v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \)

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

\textbf{proof} –

let \( \forall v \in \{ v \mid v, v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \}

\textbf{and } \( \forall v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v

\textbf{show } \forall v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v

\textbf{proof (rule iffI)}

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

\textbf{then have } v \in \{ v \mid v, v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \}

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

\textbf{unfolding } \text{state-to-strips-state-dom-is}[\text{OF assms}(1)]

by force+

moreover have \( v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v

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

by force

\textbf{next}

\textbf{assume } v \in \text{set } ((\Psi)_{v^+}) \land s v \neq \text{None} \land a \in \mathcal{R}_+ \Psi v

\textbf{then have } v \in \text{set } ((\Psi)_{v^+})

90
and $s v \neq \text{None}$
and a-in-range-of-v: $a \in R_+ \Psi v$
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[OF assms(1)]
using a-in-range-of-v
by blast
qed

lemma state-to-strips-state-range-is:
assumes 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 $?vs = \text{variables-of } \Psi$
let $?s' = \varphi_S \Psi s$
and $?\text{defined} = \text{filter } (\lambda v. s v \neq \text{None}) ?vs$
let $?l = \text{concat } \{ \text{possible-assignments-for } \Psi v. v \leftarrow \text{?defined} \}$
have $v-\text{in-set-vs}: v \in \text{set } ?vs$
and $s-\text{of-v-is-not-None}: s v \neq \text{None}$
and a-in-range-of-v: $a \in R_+ \Psi v$
using assms(2)
unfolding state-to-strips-state-dom-is[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 $(v, a) \in \text{set } ?l$
unfolding state-to-strips-state-dom-is-\text{of } \text{OF calculation(1)}
using s-of-v-is-not-None a-in-range-of-v v-\text{in-set-vs}
by fastforce
moreover have set $?l \neq \{\}$
using calculation
— 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[of $\\ ?l \ (v, a) \lambda (v, a). \text{the } (s v) = a$]
unfolding SAS-Plus-STRIPS.state-to-strips-state-def
state-to-strips-state-def Let-def case-prod-beta'
by fastforce
}
thus $?\text{thesis}$.
qed
— 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 \( \Psi \)
- \((v, a) \in \text{dom} (\varphi_S \Psi s)\)
- \((v, a') \in \text{dom} (\varphi_S \Psi s)\)
- \((\varphi_S \Psi s) (v, a) = \text{Some True}\)
- \((\varphi_S \Psi s) (v, a') = \text{Some True}\)

**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**:

\(\text{set} (\text{strips-operator.delete-effects-of} (\varphi_O \Psi \op)) = \bigcup (v, a) \in \text{set} (\text{effect-of}\ \op). \{ (v, a') \mid a'. a' \in (R_+ \Psi v) \land a' \neq a \})\)

**Proof**—

- let \(?D = \text{range-of} \Psi\)
- ?effect = effect-of \op
- let \(?\text{delete} = [(v, a'). (v, a) \leftarrow ?\text{effect, a'} \leftarrow \text{filter} ((\neq) a) (\text{the} (?D\ v))]\)
  
- fix \(v a\)
- assume \((v, a) \in \text{set} ?\text{effect}\)
- then have \((R_+ \Psi v) = \text{set} (\text{the} (?D\ v))\)
- using assms
- using is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op
- by fastforce
- hence \(\text{set} (\text{filter} ((\neq) a) (\text{the} (?D\ v))) = \{ a' \in R_+ \Psi v. a' \neq a \}\)
- unfolding set-filter
- by blast

**Note**: nb = this

{ — TODO slow.
- have set ?\text{delete} = \bigcup (\text{set} ' (\lambda(v, a). \text{map} (\text{Pair}\ v) (\text{filter} ((\neq) a) (\text{the} (?D\ v)))))
  
- (set ?\text{effect}))
using set-concat
by simp
also have \ldots = \bigcup ((\lambda (v, a). Pair v \cdot set (\text{filter} \ ((\neq) a) (\text{the} \ (\equiv) v))))
\quad \text{('set } ?\text{effect}')
unfolding image-comp[of set] set-map
by auto
— TODO slow.
also have \ldots = (\bigcup (v, a) \in set \ ?\text{effect}). Pair v \cdot \{ a' \in R_+ \Psi v. a' \neq a \})
using nb
by fast
finally have set \ ?\text{delete} = (\bigcup (v, a) \in set \ ?\text{effect}.
\quad \{ (v, a') \mid a', a' \in (R_+ \Psi v) \land a' \neq 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 \Pi 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 \Psi
shows set ((\varphi \Psi)_V) = (\bigcup v \in set ((\Psi)_V). \{ (v, a) \mid a. a \in R_+ \Psi v \})
proof –
let \Pi = \varphi \Psi
and \ ?\text{vs} = \text{variables-of} \ \Psi
{
  have set (strips-problem.variables-of \ ?\Pi)
  = set [as. v \leftarrow \ ?\text{vs}, as \leftarrow \text{possible-assignments-for} \ \Psi v]
  unfolding sas-plus-problem-to-strips-problem-def
  SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
  by force
  also have \ldots = (\bigcup (set \cdot (\lambda v. \text{possible-assignments-for} \ \Psi v) \cdot set \ ?\text{vs}))
  using set-concat
  by auto
  also have \ldots = (\bigcup ((set \circ \text{possible-assignments-for} \ \Psi) \cdot set \ ?\text{vs}))
  using image-comp[of set \lambda v. \text{possible-assignments-for} \ \Psi v set \ ?\text{vs}]
  by argo
  finally have set (strips-problem.variables-of \ ?\Pi)
  = (\bigcup v \in set \ ?\text{vs}. set (\text{possible-assignments-for} \ \Psi v))
  unfolding o-apply
  by blast
}
moreover have \forall v \in set \ ?\text{vs}. v \in dom (sas-plus-problem.range-of \ \Psi)
using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms
by force
ultimately show ?thesis
corollary sas-plus-problem-to-strips-problem-variable-set-element-iff:
assumes is-valid-problem-sas-plus Ψ
shows (v, a) ∈ set ((ψΨ)ψ) ←→ v ∈ set ((ψΨ)ψ) ∧ a ∈ Rψ ψ v
unfolding sas-plus-problem-to-strips-problem-variable-set-is[OF assms]
by fast

lemma sasp-op-to-strips-effect-consistent:
assumes op = φΨ op'
and op' ∈ set ((ψΨ)ψ)
and is-valid-operator-sas-plus Ψ op'
shows (v, a) ∈ set (add-effects-of op) → (v, a) /∈ set (delete-effects-of op)
and (v, a) ∈ set (delete-effects-of op) → (v, a) /∈ set (add-effects-of op)
proof
have nb: (∀(v, a) ∈ set (effect-of op'). (∀(v', a') ∈ set (effect-of op'). v ≠ v' ∨ a = a'))
using assms(3)
unfolding is-valid-operator-sas-plus-def
SAS-Plus-Representation.is-valid-operator-sas-plus-def list-all-iff ListMem-iff
Let-def
by argo
{
fix v a
assume v-a-in-add-effects-of-op: (v, a) ∈ set (add-effects-of op)
have (v, a) /∈ set (delete-effects-of op)
proof (rule ccontr)
assume ¬(v, a) /∈ set (delete-effects-of op)
moreover have (v, a) ∈
(⋃(v, a') ∈ set (effect-of op'). { (v, a'') |
 a'' ∈ (Rψ v) ∧ a'' ≠ a' })
using calculation sasp-op-to-strips-set-delete-effects-is
assms
by blast
moreover obtain a' where (v, a') ∈ set (effect-of op') and a ≠ a'
using calculation
by blast
moreover have (v, a') ∈ set (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) ∈ set (effect-of op') and (v, a') ∈ set (effect-of 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
}
morerover {
fix v a
assume v-a-in-delete-effects-of-op: (v, a) ∈ set (delete-effects-of op)
have (v, a) ∉ set (add-effects-of op)
proof (rule ccontr)
assume ¬(v, a) ∉ set (add-effects-of op)
morerover have (v, a) ∈ set (add-effects-of op)
using calculation
by blast
morerover have (v, a) ∈ 
(∪ (v, a′) ∈ set (effect-of op′). { (v, a′′) | a′′, a′′ ∈ (R+ Ψ v) ∧ a′′ ≠ a′ })
using sasp-op-to-strips-set-delete-effects-is
nb assms(1, 3) v-a-in-delete-effects-of-op
by force
morerover obtain a′ where (v, a′) ∈ set (effect-of op′) and a ≠ a′
using calculation
by blast
morerover have (v, a′) ∈ set (add-effects-of op)
using assms(1) calculation(4)
unfolding sasp-op-to-strips-def
SAS-Plus-STRIPS,sasp-op-to-strips-def
Let-def
by fastforce
morerover have (v, a) ∈ set (effect-of op′) and (v, a′) ∈ set (effect-of 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) ∈ set (add-effects-of op)
→ (v, a) ∉ set (delete-effects-of op)
and (v, a) ∈ set (delete-effects-of op)
→ (v, a) ∉ 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 (is-valid-operator-strips (\( \varphi \) \( \Psi \)))`

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

proof —
let \( \Pi = \varphi \) \( \Psi \)
let \(?\)vs = \text{strips-problem.variables-of} \( \Pi \)
\{
fix \( op \)
assume \( op \in \text{set} \) \( \text{strips-problem.operators-of} \) \( \Pi \)
— TODO slow.
then obtain \( op' \)
where \( op\text{-is: } op = \varphi_\bigcirc \) \( \Psi \) \( op' \)
and \( op'\text{-in-operators: } op' \in \text{set} \) \( (\Psi)_{\bigcirc +} \)
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 \( \text{is-valid-op'}: \text{is-valid-operator-sas-plus} \) \( \Psi \) \( op' \)
using \( \text{sublocale-sas-plus-finite-domain-representation-ii(2)[OF assms]} \)
by blast
moreover \{
fix \( v \) \( a \)
assume \( (v, a) \in \text{set} \) \( \text{strips-operator.precondition-of} \) \( op \)
— TODO slow.
then have \( (v, a) \in \text{set} \) \( \text{sas-plus-operator.precondition-of} \) \( op' \)
using \( op\text{-is} \)
unfolding \( \text{SAS-Plus-STRIPS.sasp-op-to-strips-def} \)
\( \text{sasp-op-to-strips-def} \)
by force
moreover have \( v \in \text{set} \) \( (\Psi)_{\bigcirc +} \)
using \( \text{is-valid-op'} \) \( \text{calculation} \)
using \( \text{is-valid-operator-sas-plus-then(1)} \)
by fastforce
moreover have \( a \in \mathbb{R}_+ \) \( \Psi \) \( v \)
using \( \text{is-valid-op'} \) \( \text{calculation(1)} \)
using \( \text{is-valid-operator-sas-plus-then(2)} \)
by fast
ultimately have \( (v, a) \in \text{set} \) \(?\)vs
using \( \text{sas-plus-problem-to-strips-problem-variable-set-element-iff}[\text{OF assms}(1)] \)
by force
\}
moreover \{
fix \( v \) \( a \)
assume \( (v, a) \in \text{set} \) \( \text{strips-operator.add-effects-of} \) \( op \)
then have \( (v, a) \in \text{set} \) \( \text{effect-of} \) \( op' \)
using \( op\text{-is} \)
unfolding \( \text{SAS-Plus-STRIPS.sasp-op-to-strips-def} \)
\( \text{sasp-op-to-strips-def} \)
by force
\}

96
then have $v \in \text{set } ((\Psi)_{\forall v}^+) \text{ and } a \in \mathbb{R}_+ \psi v$

using is-valid-operator-sas-plus-then is-valid-op'

by fastforce+

hence $(v, a) \in \text{set } ?vs$

using sas-plus-problem-to-strips-problem-variable-set-element-iff[FO assms(1)]

by force

} moreover {

fix $v, a'$

assume $v, a'$-in-delete-effects: $(v, a') \in \text{set } (\text{strips-operator.delete-effects-of } \text{op})$

moreover have set $(\text{strips-operator.delete-effects-of } \text{op})$

be $(\bigcup (v, a) \in \text{set } (\text{effect-of } op')).$

{ $(v, a') \mid a', a' \in (\mathbb{R}_+ \psi v) \land a' \neq a$}

using sasp-op-to-strips-set-delete-effects-is[OF is-valid-op']

op-is

by simp

— TODO slow.

ultimately obtain $a$

where $(v, a) \in \text{set } (\text{effect-of } op')$

and $a'$-in: $a' \in \{a' \in \mathbb{R}_+ \psi v. a' \neq a\}$

by blast

moreover have is-valid-operator-sas-plus $\psi \text{ op'}$

using op'-in-operators assms(1)

is-valid-problem-sas-plus-then(2)

by blast

moreover have $v \in \text{set } ((\Psi)_{\forall v}^+)$

using is-valid-operator-sas-plus-then calculation(1, 3)

by fast

moreover have $a' \in \mathbb{R}_+ \psi v$

using a'-in

by blast

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

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

by force

}

ultimately have set $(\text{strips-operator.precondition-of } \text{op}) \subseteq \text{set } ?vs$

\wedge set $(\text{strips-operator.add-effects-of } \text{op}) \subseteq \text{set } ?vs$

\wedge set $(\text{strips-operator.delete-effects-of } \text{op}) \subseteq \text{set } ?vs$

\wedge (\forall v \in \text{set } (\text{add-effects-of } \text{op}). v \notin \text{set } (\text{delete-effects-of } \text{op}))

\wedge (\forall v \in \text{set } (\text{delete-effects-of } \text{op}). v \notin \text{set } (\text{add-effects-of } \text{op}))

using sasp-op-to-strips-effect-consistent[OF op-is op'-in-operators is-valid-op']

by fast+

} thus $\lnot\text{thesis}$

unfolding is-valid-operator-strips-def STRIPS-Representation.is-valid-operator-strips-def

list-all-iff ListMem-iff Let-def
lemma is-valid-problem-sas-plus-then-strips-transformation-too-iv:
assumes is-valid-problem-sas-plus \( \Psi \)
shows \( \forall \, x. \, \left( (\varphi \ \Psi)_{t} \right) \, x \neq \text{None} \hspace{1cm} \iff \rightleftharpoons \text{ListMem} \, x \, (\text{strips-problem.variables-of} \, (\varphi \ \Psi)) \)

proof –
let \( \Psi \) = variables-of \( \Psi \)
and \( I = \text{initial-of} \ \Psi \)
and \( \Pi = \varphi \ \Psi \)
let \( \Psi' = \text{strips-problem.variables-of} \, \Pi \)
and \( I' = \text{strips-problem.initial-of} \, \Pi \)
let \( x = \text{variable-set-element-iff} \,[\text{OF assms}(1), \, \text{of v a} \, \Pi] \)

{ fix \( x \)

have \( I' \, x \neq \text{None} \iff \text{ListMem} \, x \, \Psi' \) proof (rule iffI)
assume \( I' \)-of-x-is-not-None: \( I' \, x \neq \text{None} \)
then have \( x \in \text{dom} \, I' \)
by blast
moreover obtain \( v \, a \) where \( x-is: x = (v, \, a) \)
by fastforce
ultimately have \( (v, \, a) \in \text{dom} \, I' \)
by blast
then have \( v \in \text{set} \, \Psi' \)
and \( I' \, v \neq \text{None} \)
and \( a \in \mathcal{R}_{+} \, \Psi \, v \)

using state-to-strips-state-dom-element-iff[OF assms(I), \, \text{of v a} \, \Pi]

unfolding sas-plus-problem-to-strips-problem-def
SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
state-to-strips-state-def
SAS-Plus-STRIPS.state-to-strips-state-def
by simp-
thus ListMem \( x \, \Psi' \)
unfolding ListMem-iff
using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(I)]

x-is
by auto
next
assume list-mem-x-vs': ListMem \( x \, \Psi' \)
then obtain \( v \, a \) where \( x-is: x = (v, \, a) \)
by fastforce
then have \( (v, \, a) \in \text{set} \, \Psi' \)
using list-mem-x-vs'
unfolding ListMem-iff
by blast
then have \( v \in \text{set} \, \Psi' \) and \( a \in \mathcal{R}_{+} \, \Psi \, v \)
using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF
assms(1)

by force+

moreover have \( ?I \) \( v \neq \) 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), \( \text{of } v \ a ?I \)]

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 \( ?I' \) \( x \neq \) None

using x-is

by fastforce

qed

}

thus \( ?\text{thesis} \)

by simp

qed

private lemma is-valid-problem-sas-plus-then-strips-transformation-too-v:

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

shows \( \forall x. \ ((\varphi \ \Psi) \ G) \ x \neq \text{None} \)

\[\rightarrow\ 
\text{ListMem } x \ \text{(strips-problem.variables-of } (\varphi \ \Psi))\]  

proof –

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

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

and \( ?\text{G} = \text{goal-of } \Psi \)

let \( ?\text{I} = \varphi \ \Psi \)

let \( ?\text{vs}' = \text{strips-problem.variables-of } ?\text{I} \)

and \( ?\text{G}' = \text{strips-problem.goal-of } ?\text{I} \)

have nb: \( ?\text{G}' = \varphi_S \ \Psi \ ?\text{G} \)

by simp

{

fix \( x \)

assume \( ?\text{G}' \ x \neq \text{None} \)

moreover obtain \( v \ a \text{ where } x = (v, a) \)

by fastforce

moreover have \((v, a) \in \text{dom } ?\text{G}'\)

using domIff calculation(1, 2)

by blast

moreover have \( v \in \text{set } ?\text{vs} \text{ and } a \in R_+ \ \Psi \ v \)

using state-to-strips-state-dom-is[of assms(1), \( \text{of } ?\text{G} \) nb calculation(3)

by auto+

ultimately have \( x \in \text{set } ?\text{vs}' \)

using sas-plus-problem-to-strips-problem-variable-set-element-iff[of assms(1)]

by auto

}

99
thus \( \text{thesis} \)

unfolding \( \text{ListMem-iff} \)

by \( \text{simp} \)

qed

We now show that given \( \Psi \) is a valid SASPlus problem, then \( \Pi \equiv \varphi \Psi \) 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 \( \Pi_V \).

**theorem**

\( \text{is-valid-problem-sas-plus-then-strips-transformation-too} \):

**assumes** \( \text{is-valid-problem-sas-plus} \ \Psi \)

**shows** \( \text{is-valid-problem-strips} \ (\varphi \ \Psi) \)

**proof**

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

have \( \text{list-all} \ (\text{is-valid-operator-strips} \ (\varphi \ \Psi)) \)

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

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

moreover have \( \forall \ x. \ ((\varphi \ \Psi)_I) x \neq \text{None} \rightarrow \text{ListMem} \ x \ (\text{strips-problem.variables-of} \ (\varphi \ \Psi)) \)

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

moreover have \( \forall \ x. \ ((\varphi \ \Psi)_G) x \neq \text{None} \rightarrow \text{ListMem} \ x \ (\text{strips-problem.variables-of} \ (\varphi \ \Psi)) \)

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

ultimately show \( \text{thesis} \)

unfolding \( \text{STRIPS-Representation} \).\( \text{is-valid-problem-strips-def} \)

by \( \text{fastforce} \)

qed

**lemma** \( \text{set-filter-all-possible-assignments-true-is} \):

**assumes** \( \text{is-valid-problem-sas-plus} \ \Psi \)

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

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

\( = \ (\bigcup v \in \text{set} \ ((\Psi)_V^+). \ \text{Pair} \ v \ \{ \ a \in \mathcal{R}_+ \ \Psi \ v \ s (v, a) = \text{Some True} \} \))

**proof**

let \( \aleph = \text{set} \text{-filter} \ \text{set} \text{-filter} \text{-problem-variables-of} \ \Psi \)

and \( \text{?P = (\lambda (v, a). s (v, a) = \text{Some True})} \)

let \( \aleph = \text{filter} \ ?P \ (\text{all-possible-assignments-for} \ \Psi) \)
proof

have set ?l = set (concat (map (?P) (map (possible-assignments-for ?Ψ) ?vs)))
  unfolding all-possible-assignments-for-def
  by simp
also have ... = set (concat (map (λv. ?P (possible-assignments-for ?Ψ v)) ?vs))
  unfolding map-map comp-apply
  by blast
also have ... = set (concat (map (λv. map (Pair v) (filter (?P o Pair v) (the (range-of ?Ψ v)))) ?vs))
  unfolding possible-assignments-for-def filter-map
  by blast
also have ... = set (concat (map (λv. map (Pair v) (filter (λa. s (v, a) = Some True) (the (range-of ?Ψ v)))) ?vs))
  unfolding comp-apply
  by fast
also have ... = (∪v ∈ set ?vs. Pair v ∗ set (filter (λa. s (v, a) = Some True) (the (range-of ?Ψ v))))
  unfolding set-concat set-map..
also have ... = (∪v ∈ set ?vs. Pair v ∗ set (the (range-of ?Ψ v)). s (v, a) = Some True)
  unfolding image-comp[of set] comp-apply set-map..
also have ... = (∪v ∈ set ?vs. Pair v ∗ 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(f)]
  by auto
qed

lemma strips-state-to-state-dom-is:
  assumes is-valid-problem-sas-plus ?Ψ
  shows dom (ψ−1 S ?Ψ s)
    = (∪v ∈ set ((?Ψ)v+).
      { v | a ∈ (R+ ?Ψ v) ∧ s (v, a) = Some True })
proof
  let ?vs = variables-of ?Ψ
  and ?s' = ψ−1 S ?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 ∗
      { a ∈ (R+ ?Ψ v) s (v, a) = Some True })
    unfolding set-filter-all-possible-assignments-true-is[OF assms]
    by auto
    also have ... = (∪v ∈ set ?vs. fst ∗ Pair v

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

by blast

also have \( \ldots = (\bigcup v \in \text{set ?vs}. (\lambda a. \text{fst (Pair v a)}) \cdot \)
\[ \{ a \in \mathbb{R}_+ \mid \Psi v. s(v, a) = \text{Some True} \} \]

unfolding image-comp[of fst] comp-apply

by blast

finally have \( \text{fst} \cdot \text{set ?l} = (\bigcup v \in (\bigcup v \in (\Psi^{v_+})). \)
\[ \{ v \mid a. a \in (\mathbb{R}_+ \mid \Psi v) \wedge s(v, a) = \text{Some True} \} \]

unfolding setcompr-eq-image fst-conv

by simp

thus \(?thesis\)

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

strips-state-to-state-def dom-map-of-conv-image-fst

by blast

qed

lemma strips-state-to-state-range-is:

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

and \( v \in (\bigcup v \in (\Psi^{v_+})) \)

and \( a \in \mathbb{R}_+ \mid \Psi v \)

and \( (v, a) \in \text{dom s'} \)

and \( \forall (v, a) \in \text{dom s'} \cdot \forall (v, a') \in \text{dom s'}. s'(v, a) = \text{Some True} \wedge s'(v, a') = \text{Some True} \rightarrow (v, a) = (v, a') \)

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

proof –

let \( ?\text{us} = \text{variables-of} \Psi \)

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

and \( ?\text{s} = \varphi^{-1}_{S} \Psi s' \)

let \( ?\text{as} = \text{all-possible-assignments-for} \Psi \)

let \( ?\text{l} = \text{filter} (\lambda(v, a). s'(v, a) = \text{Some True}) ?\text{as} \)

show \(?thesis\)

proof (rule iffI)

assume \( s\text{-of-v-is-Some-a: } ?\text{s v} = \text{Some a} \)

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

using s-of-v-is-Some-a

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

strips-state-to-state-def

using map-of-SomeD

by fast

hence \( s'(v, a) = \text{Some True} \)

unfolding all-possible-assignments-for-set-is set-filter

by blast

} thus \( \text{the (s' (v, a))} \)

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) = Some True
  using assms(4) domHf
  by force
— TODO slow.
moreover {  
  fix v v' a a'
  assume (v, a) ∈ set ?l and (v', a') ∈ set ?l
  then have v ≠ v' ∨ a = a'
  using assms(5)
  by fastforce
}
moreover {
  have ∀ v ∈ set ((Ψ)_V⁺). sas-plus.problem.range-of Ψ v ≠ None
    using is-valid-problem-sas-plus-then(1) assms(1)
    range-of-not-empty
    by force
  moreover have set ?l = Set.filter (λ(v, a). s' (v, a) = Some True)
    (⋃ v ∈ set ((Ψ)_V⁺). { (v, a) | a ∈ R⁺ Ψ v })
    using all-possible-assignments-for-set-is calculation
    by force
  ultimately have (v, a) ∈ set ?l
    using assms(2, 3) s'-of-v-a-is-true
    by simp
}
ultimately show ?s v = Some a
  using map-of-constant-assignments-defined-if[of ?l v a]
  unfolding SAS-Plus-STRIPS.strip-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 Ψ
  and v ∈ set ((Ψ)_V⁺)
  and s v ≠ None
  and a ∈ R⁺ Ψ v
shows (ϕ_S Ψ s) (v, a) = Some (the (s v) = a)
proof —
  let ?vs = sas-plus.problem.variables-of Ψ
  let ?s' = ϕ_S Ψ s
  and ?l = λ(v, a). the (s v) = a
  and ?l = concat (map (possible-assignments-for Ψ) (filter (λv. s v ≠ None) ?vs))
have \((v, a) \in \text{dom } ?s'\)
  using state-to-strips-state-dom-element-iff
  \(OF \text{ assms(1)}\) assms(2, 3, 4)
  by presburger

\{
  have \(v \in \{ \ v \mid v. v \in \text{set } ((\Psi)_{\mathbb{V}^+}) \land s v \neq \text{None} \ \}\)
     using assms(2, 3)
     by blast
  moreover have \(\forall v \in \text{set } ((\Psi)_{\mathbb{V}^+}). v \in \text{dom } (\text{sas-plus-problem}.\text{range-of } \Psi)\)
     using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of[OF assms(1)].

  moreover have \(\text{set } ?l = (\bigcup\{ \ (v, a) \mid a. a \in \mathbb{R}_+ \Psi v \ \}\)\)
    unfolding state-to-strips-state-dom-is-i[OF calculation(2)]
    by blast
  ultimately have \((v, a) \in \text{set } ?l\)
    using assms(4)
    by blast
  }

moreover have \(\text{set } ?l \neq \{\}\)
  using calculation
  by force
— TODO slow.

ultimately show \(?thesis\)
  unfolding SAS-Plus-STRIPS.state-to-strips-state-def
  state-to-strips-state-def
  using map-of-from-function-graph-is-some-if[of ?l (v, a) ]
  unfolding 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 strips-state-to-state-inverse-is-ii:
assumes is-valid-problem-sas-plus \(\Psi\)
  and \(v \in \text{set } ((\Psi)_{\mathbb{V}^+})\)
  and \(s v = \text{Some } a\)
  and \(a \in \mathbb{R}_+ \Psi v\)
  and \(a' \in \mathbb{R}_+ \Psi v\)
  and \(a' \neq a\)
shows \((\varphi_S \Psi s) (v, a') = \text{Some False}\)
proof —
  have \(s v \neq \text{None}\)
    using assms(3)
    by simp
  moreover have \(\text{the } (s v) \neq a'\)
    using assms(3, 6)
    by simp
ultimately show \( \text{thesis} \)

using \( \text{strips-state-to-state-inverse-is-i}[\text{OF assms}(1, 2) - \text{assms}(5)] \)

by force

qed

— NOTE Follows from the corollary above by contraposition.

**corollary** strips-state-to-state-inverse-is-iii:

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

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

and \( s\ v = \text{Some } a \)

and \( a' \in R_+ \Psi v \)

and \( (\varphi_S \Psi s) (v, a) = \text{Some True} \)

and \( (\varphi_S \Psi s) (v, a') = \text{Some True} \)

shows \( a = a' \)

proof –

have \( s\ v \neq \text{None} \)

using \( \text{assms}(3) \)

by blast

thus \( \text{thesis} \)

using \( \text{strips-state-to-state-inverse-is-i}[\text{OF assms}(1, 2)] \text{ assms}(4, 5, 6, 7) \)

by auto

qed

**lemma** strips-state-to-state-inverse-is-iv:

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

and \( \text{dom } s \subseteq \text{set } ((\Psi)V^+) \)

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

and \( s\ v = \text{Some } a \)

and \( a \in R_+ \Psi v \)

shows \( (\varphi_S^{-1} \Psi (\varphi_S \Psi s)) v = \text{Some } a \)

proof –

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

and \( ?s' = \varphi_S \Psi s \)

let \( ?s'' = \varphi_S^{-1} \Psi ?s' \)

let \( ?P = \lambda(v, a). \ ?s' (v, a) = \text{Some True} \)

let \( ?as = \text{filter } ?P \ (\text{all-possible-assignments-for } \Psi) \)

and \( ?As = \text{Set.filter } ?P \ (\bigcup v \in \text{set } ((\Psi)V^+). \ \{ (v, a) | a. a \in R_+ \Psi v \}) \)

\{ \)

have \( \forall v \in \text{set } ((\Psi)V^+). \ \text{range-of } \Psi v \neq \text{None} \)

using \( \text{sublocale-sas-plus-finite-domain-representation-ii}[\text{OF assms}(1)] \)

range-of-not-empty

by force

hence \( \text{set } ?as = ?As \)

unfolding \( \text{set-filter} \)
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 ∈ \( R_+ \) Ψ v
and a'-in-range-of-v: a' ∈ \( R_+ \) Ψ 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-Some-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-Some-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]
— Show that that $\varphi_S^{-1} \Psi$ is the inverse of $\varphi_S \Psi$. The additional constraints
dom $s = \text{set } (\Psi_{\forall})$ and $\forall v \in \text{dom } s. \text{ the } (s v) \in \mathcal{R}_+ \Psi v$ 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 $s \subseteq \text{set } ((\Psi_{\forall})_{\forall})$

and $\forall v \in \text{dom } s. \text{ the } (s v) \in \mathcal{R}_+ \Psi v$

shows $s = (\varphi_S^{-1} \Psi (\varphi_S \Psi s))$

**proof**

let ?vs = variables-of $\Psi$

and ?D = 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

\} note nb = this

moreover \{

fix $v$

assume $v \in \text{dom } ?s''$

then obtain $a$

by blast

\}

— TODO slow.

thus $\textbf{thesis}$ unfolding SAS-Plus-STRIPS.strips-state-to-state-def

strips-state-to-state-def all-possible-assignments-for-def

by simp

qed

{107}
where \( a \in \mathbb{R}_+ \psi \)

and \( \rho'(v, a) = \text{Some True} \)

using \( \text{strips-state-to-state-dom-is[OF assms(1)]} \)

by blast

then have \((v, a) \in \text{dom } \rho'\)

by blast

then have \( s v \neq \text{None} \)

using \( \text{state-to-strips-state-dom-is[OF assms(1)]} \)

by simp

then obtain \( a \) where \( s v = \text{Some } a \)

by blast

hence \( \rho'' v = s v \)

using \( \text{nb} \)

by \( \text{fastforce} \)

\}

— TODO slow.

ultimately show \( \text{thesis} \)

using \( \text{map-le-antisym[of } s \rho'' \text{]} \)

unfolding \( \text{strips-state-to-state-def} \)

\( \text{state-to-strips-state-def} \)

by blast

qed

— An important lemma which shows that the submap relation does not change if we transform the states on either side from \( \text{SAS}^+ \) to \text{STIRPS}.

\textbf{lemma} \( \text{state-to-strips-state-map-le-iff} \):

\begin{align*}
\text{assumes } & \text{is-valid-problem-sas-plus } \psi \\
& \text{and } \text{dom } s \subseteq \set{\psi}_{\psi^+} \\
& \text{and } \forall v \in \text{dom } s. \ \text{the } (s v) \in \mathbb{R}_+ \psi v \\
\text{shows } & s \subseteq_m t \leftrightarrow (\varphi_S \psi s) \subseteq_m (\varphi_S \psi t) \\
\end{align*}

\textbf{proof} –

let \( \rho s = \text{variables-of } \psi \)

and \( \rho D = \text{range-of } \psi \)

and \( \rho s' = \varphi_S \psi s \)

and \( \rho t' = \varphi_S \psi t \)

show \( \text{thesis} \)

\textbf{proof} (rule iffI)

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

\{

fix \( v a \)

assume \((v, a) \in \text{dom } \rho'\)

moreover have \( v \in \set{\psi}_{\psi^+} \) and \( s v \neq \text{None} \) and \( a \in \mathbb{R}_+ \psi v \)

using \( \text{state-to-strips-state-dom-is[OF assms(1)]} \)

by blast+ 

moreover have \( \rho'(v, a) = \text{Some } (\text{the } (s v) = a) \)

using \( \text{state-to-strips-state-range-is[OF assms(1)]} \)

by \( \text{meson} \)

moreover have \( v \in \text{dom } s \)

using \( \text{calculation(3)} \)

\}
by auto
moreover have \( s \cdot v = t \cdot v \)
  using \( s \cdot \text{map-le-t} \) calculation(6)
unfolding \( \text{map-le-def} \)
by blast
moreover have \( t \cdot v \neq \text{None} \)
  using calculation(3, 7)
by argo
moreover have \((v, a) \in \text{dom } \not\exists t'\)
  using \( \text{state-to-strips-state-dom-is} \) calculation(2, 4, 8)
by blast
moreover have \( \not\exists t' (v, a) = \text{Some} (\text{the } (t \cdot v) = a) \)
  using \( \text{state-to-strips-state-range-is} \) calculation(9)
by simp
ultimately have \( \not\exists s' (v, a) = \not\exists t' (v, a) \)
by presburger
}
thus \( \not\exists s' \subseteq_m \not\exists t' \)
unfolding \( \text{map-le-def} \)
by fast

next
assume \( s' \cdot \text{-map-le-t'} \); \( \not\exists s' \subseteq_m \not\exists t' \)
{
  fix \( v \)
assume \( v \in \text{-dom-s}: v \in \text{dom } s \)
moreover obtain \( a \) where \( \text{the-of-s-of-v-is-a}: \text{the } (s \cdot v) = a \)
by blast
moreover have \( v \cdot \text{-in-vs}: v \in \text{set } ((\Psi)_{\forall v}) \)
  and \( s \cdot \text{-of-v-is-not-None}: s \cdot v \neq \text{None} \)
  and \( a \cdot \text{-in-range-of-v}: a \in \mathcal{R}_+ \Psi v \)
using \( \text{assms}(2, 3) \) \( v \cdot \text{-in-dom-s} \) calculation
by blast+
moreover have \((v, a) \in \text{dom } \not\exists s'\)
  using \( \text{state-to-strips-state-dom-is} \) calculation(3, 4, 5)
by simp
moreover have \( \not\exists s' (v, a) = \not\exists t' (v, a) \)
  using \( s' \cdot \text{-map-le-t'} \) calculation
unfolding \( \text{map-le-def} \)
by blast
moreover have \((v, a) \in \text{dom } \not\exists t'\)
  using calculation
unfolding \( \text{domIff} \)
by argo
moreover have \( \not\exists s' (v, a) = \text{Some} (\text{the } (s \cdot v) = a) \)
  and \( \not\exists t' (v, a) = \text{Some} (\text{the } (t \cdot v) = a) \)
using \( \text{state-to-strips-state-range-is} \) calculation
by fast+
moreover have \( s \cdot v = \text{Some } a \)
using calculation(2, 4)
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 \( (t v) = a \)
using calculation(10, 13) try0
by force
moreover { 
  have \( v \in \text{dom } t \)
  using state-to-strips-state-dom-element-iff[\text{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 \( \varphi^{-1}_O \Pi \) 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 \( \text{op} \in \text{set } (\Psi)_O+ \)
shows \( (\varphi_O^{-1} \Psi (\varphi_O \Psi \text{op})) = \text{op} \)
proof –
let \( \text{op} = \varphi_O^{-1} \Psi (\varphi_O \Psi \text{op}) \)
have precondition-of \( \text{op} = \text{precondition-of } \text{op} \)
unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
by fastforce
moreover have effect-of \( \text{op} = \text{effect-of } \text{op} \)
unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def

110
strips-op-to-sasp-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
by force
ultimately show \( \text{thesis} \)
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 \( \langle \emptyset \text{ precondition-of} = [], \text{add-effects-of} = [], \text{delete-effects-of} = [(v, a), ...] \rangle \) correspond to one SAS+ operator (since the delete effects are being discarded in the transformation function).

**Lemma** strips-operator-inverse-is:
assumes is-valid-problem-sas-plus \( \Psi \)
and \( op' \in \text{set} ((\varphi \Psi)_O) \)
shows \( (\varphi_O \Psi (\varphi_O^{-1} \Psi op')) = op' \)
proof
let \( \Pi = \varphi \Psi \)
obtain \( op \) where \( op \in \text{set} ((\Psi)_O^+) \) and \( op' = \varphi_O \Psi op \)
using assms
by auto
moreover have \( \varphi_O^{-1} \Psi op' = op \)
using sas-plus-operator-inverse-is[OF assms(1) calculation(1)] calculation(2)
by blast
ultimately show \( \text{thesis} \)
by argo
qed

**Lemma** sas-plus-equivalent-to-strips-i-a-I:
assumes is-valid-problem-sas-plus \( \Psi \)
and set \( ops' \subseteq \text{set} ((\varphi \Psi)_O) \)
and STRIPS-Semantics.are-all-operators-applicable \( (\varphi_S \Psi s) \) \( ops' \)
and \( op \in \text{set} [\varphi_O^{-1} \Psi op'. \; op' \leftarrow \; ops'] \)
shows map-of \( \text{precondition-of} \) \( \subseteq \) \( m (\varphi_S^{-1} \Psi (\varphi_S \Psi s)) \)
proof
let \( \Pi = \varphi \Psi \)
and \( ?s' = \varphi_S \Psi s \)
let \( ?s = \varphi_S^{-1} \Psi ?s' \)
and \( ?D = \text{range-of} \Psi \)
and \( ?ops = [\varphi_O^{-1} \Psi op'. \; op' \leftarrow \; ops'] \)
and \( ?pre = \text{precondition-of} \)
have \( \forall (v, a) \in \text{dom} \; ?s'. \)
\( \forall (v, a') \in \text{dom} \; ?s'. \)
\( ?s' (v, a) = \text{Some True} \wedge ?s' (v, a') = \text{Some True} \)
\( \rightarrow (v, a) = (v, a') \)
using state-to-strips-state-effect-consistent[OF assms(1)]
by blast
{
  fix op'
  assume op' ∈ set ops'
  moreover have op' ∈ set ((Π)O)
    using assms(2) calculation
  by blast
  ultimately have ∃ op ∈ set ((Ψ)O+). op' = (φO Ψ op)
  by auto
} note nb2 = this
{
  fix op
  assume op ∈ set ops
  then obtain op' where op' ∈ set ops' and op = φO⁻¹ Ψ op'
    using assms(4)
  by auto
  moreover obtain op'' where op'' ∈ set ((Ψ)O+) and op' = φO Ψ op''
    using nb2 calculation(1)
  by blast
  moreover have op = op''
    using sas-plus-operator-inverse-is[OF assms(1) calculation(3)] calculation(2, 4)
  by blast
  ultimately have op ∈ set ((Ψ)O+)
  by blast
} note nb3 = this
{
  fix op v a
  assume op ∈ set ops
    and v-a-in-precondition-of-op': (v, a) ∈ set (precondition-of op)
  moreover obtain op' where op' ∈ set ops' and op = φO⁻¹ Ψ op'
    using calculation(1)
  by auto
  moreover have strips-operator.precondition-of op' = precondition-of op
    using calculation(4)
  unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
    strips-op-to-sasp-def
  by simp
  ultimately have ∃ op' ∈ set ops'. op = (φO⁻¹ Ψ op')
    ∧ (v, a) ∈ set (strips-operator.precondition-of op')
  by metis
} note nb4 = this
{
  fix op' v a
  assume op' ∈ set ops'
    and v-a-in-precondition-of-op': (v, a) ∈ set (strips-operator.precondition-of op')
  moreover have s'-of-v-a-is-Some-True: ?s' (v, a) = Some True
    using assms(3) calculation(1, 2)
unfolding are-all-operators-applicable-set
by blast
moreover {
  obtain op where op ∈ set ((Ψ)O+) and op' = ϕO Ψ op
  using nb2 calculation(1)
  by blast
moreover have strips-operator.precondition-of op' = precondition-of op
  using calculation(2)
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
  by simp
moreover have (v, a) ∈ set (precondition-of op)
  using v-a-in-precondition-of-op' calculation(3)
  by argo
moreover have is-valid-operator-sas-plus Ψ op
  using is-valid-problem-sas-plus-then(2) assms(1) calculation(1)
  unfolding is-valid-operator-sas-plus-def
  by auto
moreover have v ∈ set ((Ψ)V+) and a ∈ R_+ Ψ v
  using is-valid-operator-sas-plus-then(1,2) calculation(4, 5)
  unfolding is-valid-operator-sas-plus-def
  by fastforce+
moreover have v ∈ dom ?s
  using strips-state-to-state-dom-is[OF assms(1), of ?s']
  s'-of-v-a-is-True calculation(6, 7)
  by blast
moreover have (v, a) ∈ dom ?s'
  using s'-of-v-a-is-True domIff
  by blast
ultimately have ?s v = Some a
  using strips-state-to-state-range-is[OF assms(1) - - nb1]
  s'-of-v-a-is-True
  by simp
}
} hence ?s v = Some a.
} note nb5 = this
{
  fix v
  assume v ∈ dom (map-of ?pre)
  then obtain a where map-of ?pre v = Some a
  by fast
moreover have (v, a) ∈ set ?pre
  using map-of-SomeD calculation
  by fast
moreover {
  have op ∈ set ((Ψ)O+)
    using assms(4) nb3
    by blast
  then have is-valid-operator-sas-plus Ψ op
}
using \textit{is-valid-problem-sas-plus-then}(2) \textit{assms}(1)

unfolding \textit{is-valid-operator-sas-plus-def}

by \text{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 \textit{is-valid-operator-sas-plus-then}(5)

unfolding \textit{is-valid-operator-sas-plus-def}

by \text{fast}

\}

\text{moreover have } \text{map-of } ?\text{pre } v = \text{Some } a

using \textit{map-of-constant-assignments-defined-if}[\text{of } ?\text{pre}] \text{ calculation}(2, 3)

by \text{blast}

\text{moreover obtain } op' \text{ where } op' \in \text{set } ops'

and (v, a) \in \text{set } (\text{strips-operator.precondition-of } op')

using \textit{nb4}[\text{OF } \textit{assms}(4) \text{ calculation}(2)]

by \text{blast}

\text{moreover have } ?s \text{ v } = \text{Some } a

using \textit{nb5} \text{ calculation}(5, 6)

by \text{fast}

ultimately have \text{map-of } ?\text{pre } v = ?s \text{ v}

by \text{argo}

\}

\text{thus } ?\text{thesis}

unfolding \textit{map-le-def}

by \text{blast}

\text{qed}

\text{lemma } \textit{to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure}:

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

and \text{set } ops' \subseteq \text{set } ((\varphi \Psi)_O)

and \text{op } \in \text{set } [\varphi_O^{-1} \Psi \text{ op'. op' } \leftarrow \text{ ops'}]

\text{shows } \text{op } \in \text{set } ((\Psi)_O +) \land (\exists \text{ op' } \in \text{set } ops'. \text{ op' } = \varphi_O \Psi \text{ op})

\text{proof –}

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

obtain \text{op'} \text{ where } \text{op' } \in \text{set } ops' \text{ and } \text{op } = \varphi_O^{-1} \Psi \text{ op'}

using \textit{assms}(3)

by \text{auto}

\text{moreover have } \text{op' } \in \text{set } ((\Pi)_O)

using \textit{assms}(2) \text{ calculation}(1)

by \text{blast}

\text{moreover obtain } \text{op'' } \text{ where } \text{op'' } \in \text{set } ((\Psi)_O +) \text{ and } \text{op' } = \varphi_O \Psi \text{ op''}

using \textit{calculation}(3)

by \text{auto}

\text{moreover have } \text{op } = \text{op''}

using \textit{sas-plus-operator-inverse-is}[\text{OF } \textit{assms}(1) \text{ calculation}(4)] \text{ calculation}(2, 5)

by \text{presburger}

ultimately show ?\text{thesis}

by \text{blast}

\text{qed}
lemma sas-plus-equivalent-to-strips-i-a-II:
fixes $\Psi :: (\text{variable}, \text{domain})$ sas-plus-problem
fixes $s :: (\text{variable}, \text{domain})$ state
assumes is-valid-problem-sas-plus $\Psi$
and set $\text{ops'} \subseteq \text{set } ((\varphi \Psi) \cup O)$
and STRIPS-Semantics.are-all-operators-applicable ($\varphi_s \Psi s$) $\text{ops'}$
and STRIPS-Semantics.are-all-operator-effects-consistent $\text{ops'}$
shows are-all-operator-effects-consistent $[\varphi_O^{-1} \Psi \text{ op'}, \text{ op'} \leftarrow \text{ops'}]$
proof
let $?s' = \varphi_S \Psi s$
let $?s = \varphi_S^{-1} \Psi ?s'$
and $?\Pi = \varphi \Psi$
have nb: $\forall (v, a) \in \text{dom } ?s'$.
$v, a' \in \text{dom } ?s'$.
$v = \text{Some True} \land ?s' (v, a') = \text{Some True}$
$v, a = (v, a')$
using state-to-strips-state-effect-consistent[OF assms(1)]
by blast
{ fix $\text{op}_1', \text{op}_2'$
assume $\text{op}_1' \in \text{set } \text{ops'}$ and $\text{op}_2' \in \text{set } \text{ops'}$
hence STRIPS-Semantics.are-operator-effects-consistent $\text{op}_1' \text{ op}_2'$
using assms(3)
unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
by blast }
{ note nb_1 = this
fix $\text{op}_1 \text{ op}_1', \text{op}_2 \text{ op}_2'$
assume $\text{op}_1 \text{-in-ops} : \text{op}_1 \in \text{set } ?\text{ops}$
and $\text{op}_1 \text{-in-ops'} : \text{op}_1' \in \text{set } \text{ops'}$
and $\text{op}_1 \text{-is} : \text{op}_1' = \varphi_O \Psi \text{ op}_1$
and $\text{is-valid-ops} : \text{is-valid-operator-sas-plus } \Psi \text{ op}_1$
and $\text{op}_2 \text{-in-ops} : \text{op}_2 \in \text{set } ?\text{ops}$
and $\text{op}_2 \text{-in-ops'} : \text{op}_2' \in \text{set } \text{ops'}$
and $\text{op}_2 \text{-is} : \text{op}_2' = \varphi_O \Psi \text{ op}_2$
and $\text{is-valid-ops} : \text{is-valid-operator-sas-plus } \Psi \text{ op}_2$
have $\forall (v, a) \in \text{set } (\text{add-effects-of } \text{op}_1') \forall (v', a') \in \text{set } (\text{add-effects-of } \text{op}_2')$.
$v \neq v' \lor a = a'$
proof (rule ccontr)
assume $\lnot (\forall (v, a) \in \text{set } (\text{add-effects-of } \text{op}_1'). \forall (v', a') \in \text{set } (\text{add-effects-of } \text{op}_2')$.
$v \neq v' \lor a = a'$
then obtain $v v' a a'$ where $(v, a) \in \text{set } (\text{add-effects-of } \text{op}_1')$
and $(v', a') \in \text{set } (\text{add-effects-of } \text{op}_2')$
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 \(\text{SAS-Plus-STRIPS.sasp-op-to-strips-def}\)
\(\text{sasp-op-to-strips-def}\)
by force
moreover \{ \)
have \((v', a') \in \text{set (effect-of } op_2)\)
using \(op_2\text{-is calculation}(2)\)
unfolding \(\text{SAS-Plus-STRIPS.sasp-op-to-strips-def}\)
\(\text{sasp-op-to-strips-def}\)
by force
hence \(a' \in \mathbb{R}_+ \Psi v\)
using \(\text{is-valid-operator-sas-plus-then is-valid-op calculation}(3)\)
by fastforce
\}
moreover have \((v, a') \in \text{set (delete-effects-of } op_1')\)
using \(\text{sasp-op-to-strips-set-delete-effects-is}\)
\(op_1\text{-is is-valid-op}_1\) calculation\((3, 4, 5, 6)\)
by blast
moreover have \(\neg \text{STRIPS-Semantics.are-operator-effects-consistent } op_1'\)
\(op_2'\)
unfolding \(\text{STRIPS-Semantics.are-operator-effects-consistent-def list-ex-iff}\)
using \(\text{calculation}(2, 3, 7)\)
by meson
ultimately show \(\text{False}\)
using \(\text{assms}(3)\) \(op_1\text{-in-ops'} op_2\text{-in-ops'}\)
unfolding \(\text{STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff}\)
by blast
qed
\}
note \(nb_3 = \text{this}\)
\{
fix \(op_1, op_2\)
assume \(op_1\text{-in-ops: } op_1 \in \text{set } ?ops\) and \(op_2\text{-in-ops: } op_2 \in \text{set } ?ops\)
moreover have \(op_1\text{-in-operators-of-}\Psi: \ op_1 \in \text{set } ((\Psi)\Omega_{+})\)
and \(op_2\text{-in-operators-of-}\Psi: \ op_2 \in \text{set } ((\Psi)\Omega_{+})\)
using \(\text{to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure}[OF}\)
\(\text{assms}(1, 2)\) \) 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 \(\text{is-valid-problem-sas-plus-then}(2)\) \(op_1\text{-in-operators-of-}\Psi op_2\text{-in-operators-of-}\Psi\)
\(\text{assms}(1)\)
unfolding \(\text{is-valid-operator-sas-plus-def}\)
by auto+
moreover obtain \(op_1' op_2'\)
where \(op_1\text{-in-ops': } op_1' \in \text{set } ops'\)
and \(op_1\text{-is: } op_1' = \varphi_O \Psi op_1\)

116
and \( \text{op}_2 \)-in-ops'; \( \text{op}_2' \) ∈ set \( \text{ops}' \)
and \( \text{op}_2 \)-is: \( \text{op}_2' = \varphi_O \Psi \text{op}_2 \)

**using** to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure(OF
\text{assms}(1, 2)) \( \text{op}_1 \)-in-ops \( \text{op}_2 \)-in-ops

by blast

— TODO slow.

ultimately have \( \forall (v, a) \in \text{set} \ (\text{add-effects-of} \ \text{op}_1'). \forall (v', a') \in \text{set} \ (\text{add-effects-of} \ \text{op}_2'). \)

\( v \neq v' \lor a = a' \)

**using** \( \text{nb}_3 \)

by auto

hence are-operator-effects-consistent \( \text{op}_1 \ \text{op}_2 \)

**using** \( \text{op}_1 \)-is \( \text{op}_2 \)-is

unfolding are-operator-effects-consistent-def

sasp-op-to-strips-def

list-all-iff Let-def

by simp

}\)

thus \( \text{thesis} \)

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 \( \text{is-valid-problem-sas-plus} \ \Psi \)

and set \( \text{ops}' \subseteq \text{set} \ ((\varphi \Psi)_O) \)

and STRIPS-Semantics.are-all-operators-applicable \( (\varphi_S \Psi \ s) \ \text{ops}' \)

\land \text{STRIPS-Semantics.are-all-operator-effects-consistent} \ (\text{ops}' \)

shows are-all-operators-applicable-in \( (\varphi_S^{-1} \Psi (\varphi_S \Psi \ s)) \ [\varphi_O^{-1} \Psi \ op'. \ op' \leftarrow \ \text{ops}'] \land \)

are-all-operator-effects-consistent \( [\varphi_O^{-1} \Psi \ op'. \ op' \leftarrow \ \text{ops}' \]

**proof** —

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

and \( ?s' = \varphi_S \Psi \ s \)

let \( ?s' = \text{strips-problem.variables-of} \ \Pi \)

and \( ?\text{ops}' = \text{strips-problem.operators-of} \ \Pi \)

and \( ?s = \text{variables-of} \ \Psi \)

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

and \( ?s = \varphi_S^{-1} \Psi \ ?s' \)

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

have \( \text{nb} \ \forall (v, a) \in \text{dom} \ ?s'. \)

\( \forall (v, a') \in \text{dom} \ ?s'. \)

\( ?s' (v, a) = \text{Some True} \land ?s' (v, a') = \text{Some True} \)
\[\rightarrow (v, a) = (v, a')\]

**using** state-to-strips-state-effect-consistent[OF assms(1)]

**by** blast

\{
  **have** STRIPS-Semantics.are-all-operators-applicable ?s' ops'
    **using** assms(3)
    **by** simp

  **moreover have** list-all \((\text{\lambda op. map-of (precondition-of op)} \subseteq_m ?s) \?ops\)
    **using** sas-plus-equivalent-to-strips-i-a-I[OF assms(1) assms(2)]
    **calculation**
    **unfolding** list-all-iff
    **by** blast

  **moreover have** list-all \((\text{\lambda op. list-all (are-operator-effects-consistent op)} ?ops) \?ops\)
    **using** sas-plus-equivalent-to-strips-i-a-II assms nb
    **unfolding** are-all-operator-effects-consistent-def is-valid-operator-sas-plus-def
    **calculation**
    **unfolding** list-all-iff
    **by** blast

  **ultimately have** are-all-operators-applicable-in ?s ?ops
    **using** sas-plus-equivalent-to-strips-i-a-II assms nb
    **by** simp

  **ultimately show** ?thesis
    **by** simp

**qed**

**lemma** sas-plus-equivalent-to-strips-i-a-VI:

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

**and** dom s \(\subseteq\) set \((\psi)_{\psi+}\)

**and** \(\forall v \in\) dom s, \(\text{the} (s v) \in R_+ \psi v\)

**and** set ops' \(\subseteq\) set \((\varphi \psi)_{\varphi}\)

**and** are-all-operators-applicable-in s \([\varphi^{-1}\psi \text{ op', op' \leftarrow ops}]\) \wedge are-all-operator-effects-consistent \([\varphi^{-1}\psi \text{ op', op' \leftarrow ops}]\)

**shows** STRIPS-Semantics.are-all-operators-applicable \((\varphi_S \psi s) \?ops'\)

**proof** –

**let** ?vs = variables-of \(\Psi\)

**and** ?D = range-of \(\Psi\)

**and** ?\!I = \(\varphi \psi\)

**and** ?ops = \([\varphi^{-1}\psi \text{ op', op' \leftarrow ops}]\)

**and** ?s' = \(\varphi_S \psi s\)

\(\text{-- TODO refactor.}\)

\{
  **fix** op'
  **assume** op' \(\in\) set ops'

  **moreover obtain** op where op \(\in\) set ?ops and op = \(\varphi^{-1}\psi op'\)
    **using** calculation

**qed**

118
moreover obtain $op''$ where $op'' \in \text{set } ((\Psi)\circ+) \text{ and } op' = \varphi_O \Psi op''$

by auto

moreover have $\text{is-valid-operator-sas-plus } \Psi op''$

using $\text{is-valid-problem-sas-plus-then(2) } \text{assms(1) calculation(4)}$

unfolding $\text{is-valid-operator-sas-plus-def}$

by auto

moreover have $op = op''$

using $\text{sas-plus-operator-inverse-is[OF assms(1)] calculation(3, 4, 5)}$

by blast

ultimately have $\exists op \in \text{set } ?\text{ops}. \ op \in \text{set } ?\text{ops} \land \ op = (\varphi_O^{-1} \Psi \ op')$

$\land \text{is-valid-operator-sas-plus } \Psi \ op$

by blast

} note nb1 = this

have nb2: $\forall (v, a) \in \text{dom } ?s'$.

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

$\exists s' (v, a) = \text{Some True } \land \ ?s' (v, a') = \text{Some True}$

$\rightarrow (v, a) = (v, a')$

using $\text{state-to-strips-state-effect-consistent[OF assms(1), of - - s]}$

by blast

} note nb3 = this

} fix op

assume $op \in \text{set } ?\text{ops}$

hence $\text{map-of (precondition-of } \ op) \subseteq_m s$

using $\text{assms(5)}$

unfolding $\text{are-all-operators-applicable-in-def}$

$\text{is-operator-applicable-in-def list-all-iff}$

by blast

} note nb4 = this

} fix $op'$

assume $op' \in \text{set } ops'$

then obtain $op$ where $\text{op-in-ops: } op \in \text{set } ?\text{ops}$

and $\text{op-is: } op = (\varphi_O^{-1} \Psi \ op')$

and $\text{is-valid-operator-op: } \text{is-valid-operator-sas-plus } \Psi \ op$

using nb1

by force

moreover have $\text{preconditions-are-consistent:}$

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

$\forall a = a'$

using $\text{is-valid-operator-sas-plus-then(5) calculation(3)}$

unfolding $\text{is-valid-operator-sas-plus-def}$

by fast

moreover {

fix $v \ a$

assume $(v, a) \in \text{set } (\text{strips-operator.precondition-of } op')$

moreover have $\text{v-a-in-precondition-of-op: } (v, a) \in \text{set } (\text{precondition-of } \ op)$

using $\text{op-is calculation}$

119
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-le-def
by force
moreover have v ∈ set ((Ψ)υ+) and a ∈ R+ 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−1 Ψ ?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-Semantics.are-all-operator-effects-consistent-ops
by simp
qed

lemma sas-plus-equivalent-to-strips-i-a-VII:
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)O)
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-operator-effects-consistent ops′
proof 
let $s' = \varphi_S \Psi s$
and $\text{ops}' = [\varphi_O^{-1} \Psi \text{op'}, \text{op'} \leftarrow \text{ops}']$
and $\text{ops} = \text{range-of } \Psi$
and $\Pi = \varphi \Psi$
— TODO refactor.

{ 
  fix $\text{op}'$
  assume $\text{op}' \in \text{set } \text{ops}'$
  moreover obtain $\text{op}$ where $\text{op} \in \text{set } \text{ops}$ and $\text{op} = \varphi_O^{-1} \Psi \text{op}'$
  using calculation
  by force
  moreover obtain $\text{op}''$ where $\text{op}'' \in \text{set } (\Psi)_{O+}$ and $\text{op}' = \varphi_O \Psi \text{op}''$
  using $\text{assms(4)}$ calculation(1)
  by auto
  moreover have $\text{is-valid-operator-sas-plus } \Psi \text{op}''$
  using $\text{is-valid-problem-sas-plus-then(2) assms(1) calculation(4)}$
  unfolding $\text{is-valid-operator-sas-plus-def}$
  by auto
  moreover have $\text{op} = \text{op}''$
  using $\text{sas-plus-operator-inverse-is[OF assms(1)] calculation(3, 4, 5)}$
  by blast
  ultimately have $\exists \text{op} \in \text{set } \text{ops, op} \in \text{set } \text{ops} \land \text{op}' = (\varphi_O \Psi \text{op})$
  and $\text{is-valid-operator-sas-plus } \Psi \text{op}$
  by blast
}

} note $nb_1 = \text{this}$

{ 
  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 $\exists (v, a) \in \text{set } (\text{add-effects-of } \text{op}_1')$. $\exists (v', a') \in \text{set } (\text{delete-effects-of } \text{op}_2')$.
  $(v, a) = (v', a')$
  moreover obtain $\text{op}_1 \text{op}_2$
  where $\text{op}_1 \in \text{set } \text{ops}$
  and $\text{op}_1' = \varphi_O \Psi \text{op}_1$
  and $\text{is-valid-operator-sas-plus } \Psi \text{op}_1$
  and $\text{op}_2 \in \text{set } \text{ops}$
  and $\text{op}_2' = \varphi_O \Psi \text{op}_2$
  and $\text{is-valid-op}_2$: $\text{is-valid-operator-sas-plus } \Psi \text{op}_2$
  using $\text{nb}_1$ calculation(1, 2)
  by meson
  moreover obtain $v v' a a'$
  where $(v, a) \in \text{set } (\text{add-effects-of } \text{op}_1')$
  and $(v', a') \in \text{set } (\text{delete-effects-of } \text{op}_2')$
  and $(v, a) = (v', a')$
  using calculation
  by blast
  moreover have $(v, a) \in \text{set } (\text{effect-of } \text{op}_1)$
  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 \((v', a') \in (\bigcup (v'', a'')) \in \text{set (effect-of op2)}\).

\{ \( (v'', a''') \mid a'''' \in (\mathbb{R}_+ \psi v''') \land a'''' \neq a'' \) \}

using sasp-op-to-strips-set-delete-effects-is

calculation(8, 11) is-valid-op

by blast

then obtain \( v'' a'' \) where \((v'', a'') \in \text{set (effect-of op2)}\)

and \( (v', a') \in \{ (v'', a''') \mid a'''' \in (\mathbb{R}_+ \psi v'') \land a'''' \neq a'' \} \)

by blast

moreover have \((v', a'') \in \text{set (effect-of op2)}\)

using calculation

by blast

moreover have \( a' \in \mathbb{R}_+ \psi v'' \) and \( a' \neq a'' \)

using calculation(1, 2)

by fast+

ultimately have \( \exists a''. \ (v', a'') \in \text{set (effect-of op2)} \land a' \in (\mathbb{R}_+ \psi v') \land a'' \neq a'' \\

by blast

\}

moreover obtain \( a'' \) where \((v', a'') \in \text{set (effect-of op2)}\)

and \( a' \in \mathbb{R}_+ \psi v' \) and \( a' \neq a'' \)

using calculation(16)

by blast

moreover have \( \exists (v, a) \in \text{set (effect-of op1)}. (\exists (v', a') \in \text{set (effect-of op2)}. \ v = v' \land a \neq a') \)

using calculation(13, 14, 15, 17, 19)

by blast

moreover have \( \neg\text{are-operator-effects-consistent op1 op2} \)

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' \in \text{set ops'} and op2' in-ops: op2' \in \text{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').
∃(v', a') ∈ set (delete-effects-of op₂'). (v, a) = (v', a')
| (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)ν+)
and ∀ v ∈ dom s. the (s v) ∈ Rₚ Ψ v
and set ops' ⊆ set ((φ Ψ)〇)
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 (φ₂ 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
unfolding execute-parallel-operator-sas-plus-def
by simp
next
case (Cons op ops)
let ?s' = s ++ map-of (effect-of op)
— TODO Wrap IH instantiation in block.


\{
\begin{align*}
\text{have } & \forall (v, a) \in \text{set (effect-of op)}. \ v \in V \\
\text{using } & \text{Cons.prems(2)} \\
\text{by fastforce}
\end{align*}
\}

\begin{align*}
\text{moreover have } & \text{fst ' set (effect-of op) } \subseteq V \\
\text{using } & \text{calculation} \\
\text{by fastforce}
\end{align*}

\begin{align*}
\text{ultimately have } & \text{dom } ?s' \subseteq V \\
\text{unfolding } & \text{dom-map-add dom-map-of-conv-image-fst} \\
\text{using } & \text{Cons.prems(1)} \\
\text{by blast}
\end{align*}

\begin{align*}
\text{moreover have } & \forall op \in \text{set ops}. \forall (v, a) \in \text{set (effect-of op)}. \ v \in V \\
\text{using } & \text{Cons.prems(2)} \\
\text{by fastforce}
\end{align*}

\begin{align*}
\text{ultimately have } & \text{dom } (\text{execute-parallel-operator-sas-plus } ?s' \text{ ops}) \subseteq V \\
\text{using } & \text{Cons.IH[of } ?s'] \\
\text{by fast}
\end{align*}

\text{thus } ?\text{case}

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

\text{qed}

\text{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.

\begin{lemma}
\text{sas-plus-equivalent-to-strips-i-a-X:}
\end{lemma}

\begin{assumes}
\text{dom } s \subseteq V \\
\text{and } V \subseteq \text{dom } D \\
\text{and } \forall v \in \text{dom } s. \text{the } (s v) \in \text{set (the } (D v)) \\
\text{and } \forall op \in \text{set ops}. \forall (v, a) \in \text{set (effect-of op)}. \ v \in V \land a \in \text{set (the } (D v))
\end{assumes}

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

\text{proof –}

\text{show } ?\text{thesis}

\text{using } \text{assms}

\text{proof (induction ops arbitrary: } s) \\
\text{case Nil}
\text{ then show } ?\text{case}
\text{ unfolding } \text{execute-parallel-operator-sas-plus-def} \\
\text{ by simp}

\text{next}
\text{ case } (\text{Cons } op \text{ ops})
\text{ let } ?s' = s ++ \text{ map-of (effect-of op)}
\{
\begin{align*}
\text{have } & \forall (v, a) \in \text{set (effect-of op)}. \ v \in V
\end{align*}
\}

\text{124}
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

}\n
moreover { 

fix \( v \)
assume \( v \in \text{dom-s':} \ v \in \text{dom \ ?s'} \)

hence \( \langle \text{(?s' \ v)} \rangle \in \text{set \ (the \ (D \ v))} \)

proof (cases \( v \in \text{dom \ (map-of \ (effect-of \ op))} \))

\begin{itemize}
  \item \textbf{case} \( \text{True} \)
    \begin{itemize}
      \item more have \( \text{?s' \ v} = (\text{map-of \ (effect-of \ op)}) \ v \)
      \textbf{unfolding} map-add-dom-app-simps(1)[OF \ True]
      \textbf{by} blast
    \end{itemize}
  \item \textbf{moreover obtain} \( a \) \textbf{where} \( (\text{map-of \ (effect-of \ op)}) \ v = \text{Some} \ a \)
    \textbf{using} calculation(1)
    \textbf{by} fast
  \item \textbf{moreover have} \( (v, a) \in \text{set \ (effect-of \ op)} \)
    \textbf{using} map-of-SomeD calculation(3)
    \textbf{by} fast
  \item \textbf{moreover have} \( a \in \text{set \ (the \ (D \ v))} \)
    \textbf{using} Cons.prems(4) calculation(4)
    \textbf{by} fastforce
  \item ultimately show \( \text{?thesis} \)
    \textbf{by} force
\end{itemize}

\textbf{next}
\begin{itemize}
  \item \textbf{case} \( \text{False} \)
    \begin{itemize}
      \item then show \( \text{?thesis} \)
        \textbf{unfolding} map-add-dom-app-simps(3)[OF \ False]
        \textbf{using} Cons.prems(3) v-in-dom-s'
        \textbf{by} fast
    \end{itemize}
\end{itemize}

\textbf{qed}

}\n
moreover have \( \forall \text{op} \in \text{set ops}. \forall (v, a) \in \text{set \ (effect-of \ op)}, \ v \in V \land a \in \text{set \ (the \ (D \ v))} \)
using Cons.prems(4)
by auto

ultimately have \( \forall v \in \text{dom \ (execute-parallel-operator-sas-plus \ ?s' \ ops)}. \ the \ (execute-parallel-operator-sas-plus \ ?s' \ ops \ v) \in \text{set \ (the \ (D \ v))} \)
using Cons.IH[of s ++ map-of \ (effect-of \ op), OF - Cons.prems(2)]
by meson

}\n
thus \( \text{?case} \)

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

125
by blast
qed

lemma transform-sas-plus-problem-to-strips-problem-operators-valid:
assumes is-valid-problem-sas-plus \( \Psi \)
and \( op' \in \text{set} ((\varphi \ \Psi)_O) \)
obtains \( op \)
where \( op \in \text{set} ((\Psi)_O+) \)
and \( op' = (\varphi_O \ \Psi \ op) \) is-valid-operator-sas-plus \( \Psi \ op \)
proof

\{
  obtain \( op \) where \( op \in \text{set} ((\Psi)_O+) \) and \( op' = \varphi_O \ \Psi \ op \)
  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 \( \exists \ op \in \text{set} ((\Psi)_O+) \).
  \( op' = (\varphi_O \ \Psi \ op) \)
  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 \( op' \in \text{set} ((\varphi \ \Psi)_O) \)
shows \( (\varphi_S \ \Psi \ s) ++ \text{map-of} \ (\text{effect-to-assignments} \ op') \)
\( = \varphi_S \ \Psi \ (s ++ \text{map-of} \ (\text{effect-of} \ (\varphi_O^{-1} \ \Psi \ op'))) \)
proof

let \( ?\Pi = \varphi \ \Psi \)
let \( ?\vars = \text{variables-of} \ \Psi \)
and \( ?\ops = \text{operators-of} \ \Psi \)
and \( ?\ops' = \text{strips-problem.operators-of} \ ?\Pi \)
let \( ?s' = \varphi_S \ \Psi \ s \)
let \( ?t = ?s' ++ \text{map-of} \ (\text{effect-to-assignments} \ op') \)
and \( ?t' = \varphi_S \ \Psi \ (s ++ \text{map-of} \ (\text{effect-of} \ (\varphi_O^{-1} \ \Psi \ op'))) \)
obtain \( op \) where \( op'-is: \ op' = (\varphi_O \ \Psi \ op) \)
and \( op-in-ops: \ op \in \text{set} ((\Psi)_O+) \)
and is-valid-operator-op: is-valid-operator-sas-plus \( \Psi \ op \)
using transform-sas-plus-problem-to-strips-problem-operators-valid[OF assms]
by auto
have nb1: \( (\varphi_O^{-1} \ \Psi \ op') = \ op \)
using sas-plus-operator-inverse-is[OF assms(1)] op'-is op-in-ops
by blast
— TODO refactor.
\{ 
  have dom (map-of (effect-to-assignments op'))
  = set (strips-operator.add-effects-of op') ∪ set (strips-operator.delete-effects-of op')
  unfolding dom-map-of-conv-image-fst
  by force
  — TODO slow.
  also have ... = set (effect-of op) ∪ set (strips-operator.delete-effects-of op')
  using op'-is
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
  by auto
  — TODO slow.
  finally have dom (map-of (effect-to-assignments op')) = set (effect-of op)
    ∪ (\bigcup (v, a) ∈ set (effect-of op). \{ (v, a') | a'. a' ∈ (R⁺ ♦ v) ∧ a' ≠ a \})
  using sasp-op-to-strips-set-delete-effects-is[OF
    is-valid-operator-op] op'-is
  by argo
  — TODO refactor.
\} note nb₂ = this

have nb₃: dom ?t = dom ?s' ∪ set (effect-of op)
  ∪ (\bigcup (v, a) ∈ set (effect-of op). \{ (v, a') | a'. a' ∈ (R⁺ ♦ v) ∧ a' ≠ a \})
  unfolding nb₂ dom-map-add
  by blast
  — TODO refactor.

have nb₄: dom (s ++ map-of (ϕO⁻¹ ♦ op'))
  = dom s ∪ fst ' set (effect-of op)
  unfolding dom-map-add dom-map-of-conv-image-fst nb₁
  by fast
  \{ 
    let ?u = s ++ map-of (ϕO⁻¹ ♦ op'))
    have dom ?t = (\bigcup v ∈ \{ v | v ∈ set ((Ψ)⁺) ∧ ?u v ≠ None \}.
      \{ (v, a) | a. a ∈ R⁺ ♦ v \})
      using state-to-strips-state-dom-is[OF assms(1)]
      by presburger
  \} note nb₅ = this
  — TODO refactor.

have nb₆: set (add-effects-of op') = set (effect-of op)
  using op'-is
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
  sasp-op-to-strips-def
  by auto
  — TODO refactor.

have nb₇: set (delete-effects-of op') = (\bigcup (v, a) ∈ set (effect-of op).
  \{ (v, a') | a'. a' ∈ (R⁺ ♦ v) ∧ a' ≠ a \})
  using sasp-op-to-strips-set-delete-effects-is[OF
    is-valid-operator-op] op'-is
  by argo
  — TODO refactor.
\}
\{  
  let \( ?\text{Add} = \text{set } (\text{effect-of op}) \) 
  let \( ?\text{Delete} = (\bigcup (v, a) \in \text{set } (\text{effect-of op}). \{ (v, a') | a' \in (\mathbb{R}^+ \Psi v) \land a' \neq a \}) \) 
  have dom-add: dom (map-of (\lambda v. (v, True)) (add-effects-of op')) = ?\text{Add} 
    unfolding dom-map-of-conv-image-fst set-map image-comp comp-apply 
    using nb_6 
    by simp 
  have dom-delete: dom (map-of (\lambda v. (v, False)) (delete-effects-of op')) = ?\text{Delete} 
    unfolding dom-map-of-conv-image-fst set-map image-comp comp-apply 
    using nb_7 
    by auto 
\} 

\{  
  fix v a 
  assume v-a-in-dom-add: (v, a) \in dom (map-of (\lambda v. (v, True)) (add-effects-of op')) 
  have (v, a) \notin dom (map-of (\lambda v. (v, False)) (delete-effects-of op')) 
    proof (rule contr) 
      assume \( \neg ((v, a) \notin \text{dom } (\lambda v. (v, False)) (\text{delete-effects-of op'})) \) 
    then have (v, a) \in ?\text{Delete} and (v, a) \in ?\text{Add} 
      using dom-add dom-delete v-a-in-dom-add 
      by argo+ 
    moreover have \( \forall (v', a') \in ?\text{Add}. v \neq v' \lor a = a' \) 
      using is-valid-operator-sas-plus-then(6) is-valid-operator-op 
      calculation(2) 
    unfolding is-valid-operator-sas-plus-def 
    by fast 
    ultimately show False 
    by fast 
  qed 
\} 

hence disjoint (dom (map-of (\lambda v. (v, True)) (add-effects-of op'))))) 
  (dom (map-of (\lambda v. (v, False)) (delete-effects-of op')))) 
    unfolding disjoint-def Int-def 
    using nb_7 
    by simp 

hence dom (map-of (\lambda v. (v, True)) (add-effects-of op')) = ?\text{Add} 
  and dom (map-of (\lambda v. (v, False)) (delete-effects-of op')) = ?\text{Delete} 
  and disjoint (dom (map-of (\lambda v. (v, True)) (add-effects-of op')))) 
  (dom (map-of (\lambda v. (v, False)) (delete-effects-of op')))) 
  using dom-add dom-delete 
  by blast+ 
\} 

note nb_8 = this  
— TODO refactor.
{ 
  let ?Add = set (effect-of op) 
  let ?Delete = (\{\{v, a\} \in set (effect-of op). 
    \{ (v, a') | a', a' \in (R_+ \Psi v) \land a' \neq a \}\}) 
  — TODO slow.
  have \forall (v, a) \in ?Add. map-of (effect-to-assignments op') (v, a) = Some True 
  and \forall (v, a) \in ?Delete. map-of (effect-to-assignments op') (v, a) = Some False 

  proof — 
  { 
    fix v a 
    assume (v, a) \in ?Add 
    hence map-of (effect-to-assignments op') (v, a) = Some True 
    unfolding effect-to-assignments-simp 
    using nb_6 map-of-defined-if-constructed-from-list-of-constant-assignments[of 
      map (λv. (v, True)) (add-effects-of op') True add-effects-of op'] 
    by force 
  } 
  moreover 
  { 
    fix v a 
    assume (v, a) \in ?Delete 
    moreover have (v, a) \in dom (map-of (map (λv. (v, False)) (delete-effects-of op'))) 
      using nb_8(2) calculation(1) 
      by argo 
    moreover have (v, a) /\in dom (map-of (map (λv. (v, True)) (add-effects-of op'))) 
      using nb_8 
      unfolding disjoint-def 
      using calculation(1) 
      by blast 
    moreover have map-of (effect-to-assignments op') (v, a) 
      = map-of (map (λv. (v, False)) (delete-effects-of op')) (v, a) 
    unfolding effect-to-assignments-simp map-of-append 
    using map-add-dom-app-simps(3)[OF calculation(3)] 
    by presburger 
    — TODO slow. 
    ultimately have map-of (effect-to-assignments op') (v, a) = Some False 
    using map-of-defined-if-constructed-from-list-of-constant-assignments[ 
      of map (λv. (v, False)) (delete-effects-of op') False delete-effects-of op'] 
    nb_7 
    by auto 
  } 
  ultimately show \forall (v, a) \in ?Add. map-of (effect-to-assignments op') (v, a) = Some True 
  and \forall (v, a) \in ?Delete. map-of (effect-to-assignments op') (v, a) = Some False 
  by blast+}
qed
}

{ fix v a
  assume (v, a) ∈ set (effect-of op)
moreover have ∀(v, a) ∈ set (effect-of op). ∀(v', a') ∈ set (effect-of op). v ≠ v' ∨ a = a'
    using is-valid-operator-sas-plus-then is-valid-operator-op
    unfolding is-valid-operator-sas-plus-def
    by fast
ultimately have map-of (effect-of op) v = Some a
    using map-of-constant-assignments-defined-if[of effect-of op]
    by presburger
}

{ fix v a
  assume v-a-in-effect-of-op: (v, a) ∈ set (effect-of op)
  and (s ++ map-of (effect-of (φ⁻¹ S Ψ op'))) v ≠ None
moreover have v ∈ set ?vs
    using is-valid-operator-op is-valid-operator-sas-plus-then(3) calculation(1)
    by fastforce
moreover have is-valid-problem-strips ?Π
    using is-valid-problem-sas-plus-then-strips-transformation-too
    assms(1)
    by blast
thm calculation(1) nb₉ assms(2)
moreover have set (add-effects-of op') ⊆ set ((?Π)v)
    using assms(2) is-valid-problem-strips-operator-variable-sets(2)
    calculation
    by blast
moreover have (v, a) ∈ set ((?Π)v)
    using v-a-in-effect-of-op nb₉ calculation(2)
    by blast
ultimately have a ∈ ℝ⁺ Ψ v
    using sas-plus-problem-to-strips-problem-variable-set-element-iff[of
    assms[1]]
    by fast
}

— TODO slow.

ultimately have (v, a) ∈ dom (φS Ψ (s ++ map-of (effect-of (φ⁻¹ S Ψ op'))))

    using state-to-strips-state-dom-is[of assms(1), of
    s ++ map-of (effect-of (φ⁻¹ S Ψ op'))]
    by simp
}

{ note nb₁₀ = this
{ fix v a
  assume (v, a) ∈ set (effect-of op)
moreover have \( v \in \text{dom} \left( \text{map-of (effect-of } \text{op}) \right) \)

unfolding \( \text{dom-map-of-conv-image-fst} \)
using calculation
by force
moreover have \( (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \ v = \text{Some a} \)
unfolding \( \text{map-add-dom-app-simps}(1)[OF \text{ calculation}(2)] \ nb_{10} \)
using \( nb_{10} \) calculation(1)
by blast
moreover have \( (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \ v \neq \text{None} \)
using calculation(3)
by auto
moreover have \( (v, a) \in \text{dom} \left( \varphi_S \Psi \ (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \right) \)
using \( nb_{11} \) calculation(1, 4)
by presburger
ultimately have \( \varphi_S \Psi \ (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \ (v, a) = \text{Some True} \)
using \( \text{state-to-strips-state-range-is}[OF \text{ assms}(1)] \)
by simp
}

note \( nb_{12} = \text{this} \)

\{
fix \( v \ a' \)
assume \( (v, a') \in \text{dom} \left( \text{map-of (effect-to-assignments } \text{op'}) \right) \)
and \( (v, a') \in (\bigcup (v, a) \in \text{set (effect-of } \text{op}),
\{ (v, a') \mid a', a' \in (\mathbb{R}_+ \Psi v) \land a' \neq a \}) \)
moreover have \( v \in \text{dom} \left( \text{map-of (effect-of } \text{op}) \right) \)
unfolding \( \text{dom-map-of-conv-image-fst} \)
using calculation(2)
by force
moreover have \( v \in \text{set } \mathbb{R} \)
using calculation(3) is-valid-operator-sas-plus-then(3) is-valid-operator-op
unfolding \( \text{dom-map-of-conv-image-fst} \) is-valid-operator-sas-plus-def
by fastforce
moreover obtain \( a \) where \( (v, a) \in \text{set (effect-of } \text{op}) \)
and \( a' \in \mathbb{R}_+ \Psi v \)
and \( a' \neq a \)
using calculation(2)
by blast
moreover have \( (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \ v = \text{Some a} \)
unfolding \( \text{map-add-dom-app-simps}(1)[OF \text{ calculation}(3)] \ nb_{10} \)
using \( nb_{10} \) calculation(5)
by blast
moreover have \( (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \ v \neq \text{None} \)
using calculation(8)
by auto
— TODO slow.
moreover have \( (v, a') \in \text{dom} \left( \varphi_S \Psi \ (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))) \right) \)
using \( \text{state-to-strips-state-dom-is}[OF \text{ assms}(1), \text{of } (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi \text{ op'}))] \) calculation(4, 6, 9)
by simp
ultimately have $(\varphi \Psi (s ++ \text{map-of} (\varphi O^{-1} \Psi o''))) (v, a') = Some \text{False}$

using state-to-strips-state-range-is[of assms(1), of v a' s ++ \text{map-of} (\varphi O^{-1} \Psi o')]]
by simp

} note nb13 = this

{ fix v a
assume (v, a) ∈ dom ?t
and (v, a) ∉ dom (map-of (effect-to-assignments o'))
moreover have (v, a) ∈ 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(1, 2)...]
ultimately have ?t (v, a) = Some (the (s v) = a)
using state-to-strips-state-range-is[of assms(1)]
by presburger
} note nb14 = this

{ fix v a
assume (v, a) ∈ dom ?t
and (v, a) ∉ dom (map-of (effect-to-assignments o'))
moreover have (v, a) ∈ dom ?s'
using calculation(1, 2)
unfolding dom-map-add
by blast
moreover have (v, a) ∈ \{ (v, a) ∈ set ((\Psi)V_+) \land s v \neq \text{None} \}. { (v, a) | a. a ∈ \mathcal{R}_+ \Psi v \}
using state-to-strips-state-dom-is[of assms(1)] calculation(3)
by presburger
moreover have v ∈ set ((\Psi)V_+) and s v \neq \text{None} and a ∈ \mathcal{R}_+ \Psi v
using calculation(4)
by blast+
— NOTE Hasn’t this been proved before?
moreover {
have dom (map-of (effect-to-assignments o')) = (\bigcup (v, a) ∈ set (effect-of op)). { (v, a) }
\cup (\bigcup (v, a) ∈ set (effect-of op).
{ (v, a') | a'. a' ∈ (\mathcal{R}_+ \Psi v) \land a' \neq a }
}
unfolding nb2
by blast
also have ... = (\bigcup (v, a) ∈ set (effect-of op). { (v, a) }
\cup \{ (v, a') | a'. a' ∈ (\mathcal{R}_+ \Psi v) \land a' \neq a \})
by blast
finally have dom (map-of (effect-to-assignments o'))
= (\bigcup (v, a) ∈ set (effect-of op). { (v, a) }
∪ \{ (v, a) \mid a \cdot a \in \mathbb{R}_+ \Psi v \})

by auto
then have \( (v, a) \notin (\bigcup (v, a) \in \text{set (effect-of op)}) \).
\{ (v, a) \mid a \cdot a \in \mathbb{R}_+ \Psi v \})
using v-a-not-in
by blast

— TODO slow.

moreover have \( v \notin \text{dom (map-of (effect-of op))} \)
using dom-map-of-cert-image-fst calculation
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)[OF calculation(9)]
by simp
— TODO slow.

moreover have \( (v, a) \in \text{dom } ?t' \)
using state-to-strips-state-dom-is[OF assms(1), of s]
by auto

— TODO refactor.

have \( nb_{16} : \text{dom } ?t = (\bigcup v \in \{ \ v \mid v \cdot v \in \text{set } (\varphi_S \Psi s) \ \land \ s v \neq \text{None} \}. \)
\{ (v, a) \mid a \cdot a \in (\mathbb{R}_+ \Psi v ) \})
∪ \{ (v, a') \mid a', a' \in (\mathbb{R}_+ \Psi v ) \ \land \ a' \neq a \})
unfolding dom-map-add nb_{2}
using state-to-strips-state-dom-is[OF assms(1), of s]
by auto

{ \ 
fix \( v \ a \)
assume \( (v, a) \in \text{dom } ?t \)
then consider \( (A) \ (v, a) \in \text{dom } (\varphi_S \Psi s) \)
\mid (B) \ (v, a) \in \text{dom } (\text{map-of (effect-to-assignments op')})
by fast
hence \( (v, a) \in \text{dom } ?t' \)

proof (cases)
case \( A \)
then have \( v \in \text{set } (\varphi_S \Psi v) \text{ and } s v \neq \text{None } \text{and } a \in \mathbb{R}_+ \Psi v \)
unfolding state-to-strips-state-dom-element-iff[OF assms(1)]
by blast+
thm map-add-None state-to-strips-state-dom-element-iff[OF assms(1)]
moreover have \( (s ++ \text{map-of (effect-of } (\varphi_O^{-1} \Psi op'))) \ v \neq \text{None} \)
using calculation(2)
by simp
ultimately show \( \text{thesis} \)

unfolding state-to-strips-state-dom-element-iff[OF assms(1)]
by blast

next
case B
then have \( (v, a) \in \)
set (effect-of op)
\( \cup (\bigcup (v, a) \in \text{set (effect-of op)}. \{ (v, a') | a' \in \mathcal{R}^+ \Psi v \land a' \neq a \}) \)
unfolding nb\(_2\)
by blast
then consider \( (B_1) (v, a) \in \text{set (effect-of op)} \)
| \( (B_2) (v, a) \in (\bigcup (v, a) \in \text{set (effect-of op)}. \{ (v, a') | a' \in \mathcal{R}^+ \Psi v \land a' \neq a \}) \)
by blast
thm nb\(_{12}\) nb\(_{13}\) nb\(_2\)
thus \( \text{thesis} \)
proof (cases)
case B\(_1\)
then show \( \text{thesis} \)
using nb\(_{12}\)
by fast
next
case B\(_2\)
then show \( \text{thesis} \)
using nb\(_{13}\) B
by blast
qed
qed

moreover {
let \( ?u = s + + \text{map-of (effect-of } (\varphi^{-1}_O \Psi \text{ op}^\prime)} \)
fix \( v a \)
assume v-a-in-dom-t' : \( (v, a) \in \text{dom } ?t' \)
thm nb\(_5\)
then have v-in-vs : \( v \in \text{set } ((\Psi)_{\mathcal{Y}^+}) \)
and u-of-v-is-not-None : \( ?u v \neq \text{None} \)
and a-in-range-of-v : \( a \in \mathcal{R}^+ \Psi v \)
using state-to-strips-state-dom-element-iff[OF assms(1)]
v-a-in-dom-t'
by meson+
}

assume \( (v, a) \notin \text{dom } ?t \)
then have contradiction: \( (v, a) \notin \)
\( (\bigcup v \in \{ v | v. v \in \text{set } ((\Psi)_{\mathcal{Y}^+}) \land s v \neq \text{None}, \{ (v, a) | a. a \in \mathcal{R}^+ \Psi v \} ) \)
\( \cup \text{ set (effect-of op)} \)
\( \cup (\bigcup (v, a) \in \text{set (effect-of op)}. \{ (v, a') | a' \in \mathcal{R}^+ \Psi v \land a' \neq a \}) \)
unfolding nb\(_{16}\)
by fast
hence False

134
proof \((\text{cases map-of (effect-of } (\varphi O^{-1} \Psi) op')) v = \text{None}\) 

\text{case True}

then have \(s v \neq \text{None}\)

\text{using u-of-v-is-not-None}

by simp

then have \((v, a) \in (\bigcup v \in \{v \mid v. v \in \text{set } ((\Psi)_{\mathbb{V}^+} \land s v \neq \text{None})\}, \\
\{ (v, a) \mid a. a \in \mathbb{R}_+ \Psi v \})\)

\text{using v-in-vs a-in-range-of-v}

by blast

thus \(\text{thesis}\)

\text{using contradiction}

by blast

next

\text{case False}

then have \(v \in \text{dom (map-of (effect-of op))}\)

\text{using u-of-v-is-not-None nb1}

by blast

then obtain \(a'\) where \(\text{map-of-effect-of-op-v-is: map-of (effect-of op) v = Some a'}\)

by blast

then have \(v-a'-\text{in: } (v, a') \in \text{set (effect-of op)}\)

\text{using map-of-SomeD}

by fast

then show \(\text{thesis}\)

\text{proof (cases } a = a'\)

\text{case True}

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

\text{using v-a'-in}

by blast

then show \(\text{thesis}\)

\text{using contradiction}

by blast

next

\text{case False}

then have \((v, a) \in (\bigcup (v, a) \in \text{set (effect-of op)}), \\
\{ (v, a') \mid a'. a' \in \mathbb{R}_+ \Psi v \land a' \neq a \}\})

\text{using v-a'-in calculation a-in-range-of-v}

by blast

thus \(\text{thesis}\)

\text{using contradiction}

by fast

qed

qed

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 dom t = dom t′
by force

note nb17 = this

{ fix v a
assume v-a-in-dom-t: (v, a) ∈ dom t
hence ?t (v, a) = ?t′ (v, a)
proof (cases (v, a) ∈ dom (map-of (effect-to-assignments op′)))
  case True
    — TODO slow.
  case False
    moreover have ?t (v, a) = Some True
    unfolding map-add-dom-app-simps(1)[OF True]
    using nb9(1)
    by fast
    moreover have ?t′ (v, a) = Some True
    using nb12[OF True A].
    ultimately show ?thesis.
next
  case A1
    then have ?t (v, a) = Some True
    unfolding map-add-dom-app-simps(1)[OF True]
    using nb9(2)
    by blast
    moreover have ?t′ (v, a) = Some True
    using nb13[OF True A2].
    ultimately show ?thesis.
next
  case A2
    then have ?t (v, a) = Some False
    unfolding map-add-dom-app-simps(1)[OF True]
    using nb9(2)
    by blast
    moreover have ?t′ (v, a) = Some False
    using nb13[OF True A].
    ultimately show ?thesis.
next
  case False
    moreover have ?t (v, a) = Some (the (s v) = a)
    using nb14[OF v-a-in-dom-t False].
    moreover have ?t′ (v, a) = Some (the (s v) = a)
    using nb15[OF v-a-in-dom-t False].
ultimately show \( \exists \)thesis
by argo
qed
\}

note \( nb_{18} = this \)
moreover { 
  fix \( v \ a \)
  assume \( (v, a) \in \text{dom } \exists \)t\’
  hence \( \exists \)t \( (v, a) = \exists \)t\’ \( (v, a) \)
  using \( nb_{17} nb_{18} \)
  by presburger
}

— TODO slow.
ultimately have \( \exists \)t \( \subseteq m \exists \)t\’ and \( \exists \)t\’ \( \subseteq m \exists \)t
unfolding map-le-def
by fastforce+
thus \( \exists \)thesis
using map-le-antisym[of \( \exists \)t \( \exists \)t\’]
by fast
qed

— NOTE This is the essential step in the SAS+/STRIPS equivalence theorem. We show that executing a given parallel STRIPS operator \( ops^\prime \) on the corresponding STRIPS state \( s^\prime = \varphi_S \Psi s \) yields the same state as executing the transformed SAS+ parallel operator \( ops = [\varphi_O^{-1} (\varphi \Psi) \ op^\prime \ op^\prime \leftarrow ops^\prime] \) on the original SAS+ state \( s \) and the transforming the resultant SAS+ state to its corresponding STRIPS state.

\begin{lemma}
\textbf{sas-plus-equivalent-to-strips-i-a-XII:}
\begin{align*}
\text{assumes} \ & \text{is-valid-problem-sas-plus } \Psi \\
& \forall \ op^\prime \in \text{set } ops^\prime. \ op^\prime \in \text{set } ((\varphi \Psi)_{\bigcirc}) \\
\text{shows} \ & \text{execute-parallel-operator } ((\varphi_S \Psi s) \ ops^\prime) \\
& = \varphi_S \Psi (\text{execute-parallel-operator-sas-plus } s [\varphi_O^{-1} \Psi \ op^\prime \ op^\prime \leftarrow ops^\prime]) \\
\text{using} \ & \text{assms}
\end{align*}
\begin{proof}
(induction \( ops^\prime \) arbitrary: \( s \))
\begin{case}
\text{Nil}
\begin{proof}
\text{unfolding } \text{execute-parallel-operator-def execute-parallel-operator-sas-plus-def}
by simp
\end{proof}
\end{case}
\begin{case}
(\text{Cons } op^\prime \ ops^\prime)
\begin{proof}
let \( \exists \Pi = \varphi \Psi \\
let \( \exists \)t\’ = (\varphi_S \Psi s) ++ \text{map-of } (\text{effect-to-assignments } op^\prime) \\
\text{and } \exists \)t = \( s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi op^\prime)) \\

\begin{proof}
limit w { op^\prime \in \text{set } ops^\prime. \ op^\prime \in \text{set } (\text{strips-problem/operators-of } \exists \Pi) \\
\text{using } \text{Cons/prems}(2) \\
by simp
\}
then have \( \text{execute-parallel-operator} \ (\varphi_S \Psi ?t) \ ops' \)

\[ = \varphi_S \Psi \ (\text{execute-parallel-operator-sas-plus} \ ?t \ [\varphi_{O^{-1}} \Psi \ x. \ x \leftarrow \ ops']) \]

using Cons.IH[OF Cons.prems(1), of ?t]
by fastforce

hence \( \text{execute-parallel-operator} \ ?t' \ ops' \)

\[ = \varphi_S \Psi \ (\text{execute-parallel-operator-sas-plus} \ ?t \ [\varphi_{O^{-1}} \Psi \ x. \ x \leftarrow \ ops']) \]

using nb1
by argo
}
thus ?case
by simp
qed

lemma sas-plus-equivalent-to-strips-i-a-XIII:
assumes is-valid-problem-sas-plus \( \Psi \)
and \( \forall \ op' \in \text{set} \ ops', \ op' \in \text{set} \ ((\varphi \Psi)_O) \)
and \( (\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan} \)
(\( \text{execute-parallel-operator} \ (\varphi_S \Psi I) \ ops' \)) \( \pi \)
shows \( (\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan} \)
(\( \text{execute-parallel-operator-sas-plus} I \ [\varphi_{O^{-1}} \Psi \ op', \ op' \leftarrow \ ops'] \)) \( \pi \)

proof –

let \( ?I' = (\varphi_S \Psi I) \)

and \( ?G' = \varphi_S \Psi G \)
and \( ?ops = [\varphi_{O^{-1}} \Psi \ op', \ op' \leftarrow \ ops'] \)
and \( ?I = \varphi \Psi \)

let \( ?J = \text{execute-parallel-operator-sas-plus} I \ ?ops \)
{| fix \ v \ a 
assume \( (v, a) \in \text{dom} \ ?G' \)
then have \( ?G' \ (v, a) = \text{execute-parallel-plan} \)
(\( \text{execute-parallel-operator} \ ?I' \ ops' \)) \( \pi \ (v, a) \)
using assms(3)
unfolding map-le-def
by auto
hence \( ?G' \ (v, a) = \text{execute-parallel-plan} \ (\varphi_S \Psi ?J) \ \pi \ (v, a) \)
using sas-plus-equivalent-to-strips-i-a-XII[OF assms(1, 2)]
by simp
}
thus ?thesis
unfolding map-le-def
by fast
qed

— 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 \( \varphi_{P^{-1}} (\varphi \Psi) \) 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 \( \text{dom } I \subseteq \text{set } ((\Psi)_V) \)

and \( \forall v \in \text{dom } I. \text{the } (I v) \in \mathcal{R}_+ \Psi v \)

and \( \text{dom } G \subseteq \text{set } ((\Psi)_V) \)

and \( \forall v \in \text{dom } G. \text{the } (G v) \in \mathcal{R}_+ \Psi v \)

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

\( \text{op}' \in \text{set } ((\varphi \Psi)_{\mathcal{O}}) \)

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

**Shows** \( G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\varphi^{-1} \Psi \pi) \)

**Proof** —

let \( \Psi' = \text{variables-of } \Psi \)

and \( \psi = \varphi^{-1} \Psi \pi \)

show \( \text{thesis} \)

using \( \text{assms} \)

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

**Case** Nil

then have \( (\varphi \Psi G) \subseteq_m (\varphi \Psi I) \)

by fastforce

then have \( G \subseteq_m I \)

using state-to-strips-state-map-le-iff[\( \text{OF } \text{assms}(1, 4, 5) \)]

by blast

thus \( \text{?case} \)


strips-parallel-plan-to-sas-plus-parallel-plan-def

by fastforce

**Next**

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

let \( \Psi' = \text{range-of } \Psi \)

and \( \Pi' = \varphi \Psi \)

and \( I' = \varphi \Psi I \)

and \( G' = \varphi \Psi G \)

let \( \text{ops} = [\varphi^{-1} \Psi \text{op}', \text{op}' \leftarrow \text{ops}'] \)

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

and \( J' = \text{execute-parallel-operator } I' \text{ops}' \)

have nb\(_1\): set \( \text{ops}' \subseteq \text{set } ((\Pi)_{\mathcal{O}}) \)

using Cons.prems(6)

unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def list-all-iff

ListMem-iff

by fastforce

\{ 

fix \( \text{op} \)

moreover obtain \( \text{op}' \) where \( \text{op}' \in \text{set } \text{ops}' \) and \( \text{op} = \varphi^{-1} \Psi \text{op}' \)

using calculation

by auto

moreover have \( \text{op}' \in \text{set } ((\Pi)_{\mathcal{O}}) \)

using nb\(_1\) calculation(2)

by blast

139
moreover obtain \( \text{op''} \) where \( \text{op''} \in \text{set} \ (\Psi)_{O^+} \) and \( \text{op'} = \varphi_O \Psi \text{op''} \),
using calculation(4)
by auto
moreover have \( \text{op} = \text{op''} \)
using sas-plus-operator-inverse-is[OF assms(1)] calculation(5)
calculation(3, 6)
by presburger
ultimately have \( \text{op} \in \text{set} \ (\Psi)_{O^+} \) \( \land \) \( (\exists \text{op'} \in \text{set ops'. op'} = \varphi_O \Psi \text{op}) \)
by blast
}\ note nb2 = this
{ fix \( \text{op} \ \text{v} \ \text{a} \)
assume \( \text{op} \in \text{set} \ (\Psi)_{O^+} \) and \( (\text{v, a}) \in \text{set} \ \text{effect-of op} \)
moreover have \( \text{op} \in \text{set} \ (\Psi)_{O^+} \)
using nb2 calculation(1)
by blast
moreover have \( \text{is-valid-operator-sas-plus} \ \Psi \ \text{op} \)
using is-valid-problem-sas-plus-then(2) Cons.prems(1) calculation(3)
by blast
ultimately have \( \text{v} \in \text{set} \ (\Psi)_{V^+} \)
using is-valid-operator-sas-plus-then(3)
by fastforce
}\ note nb3 = this
{ fix \( \text{op} \)
assume \( \text{op} \in \text{set} \ ?\text{ops} \)
then have \( \text{op} \in \text{set} \ (\Psi)_{O^+} \)
using nb2
by blast
then have \( \text{is-valid-operator-sas-plus} \ \Psi \ \text{op} \)
using is-valid-problem-sas-plus-then(2) Cons.prems(1)
by blast
hence \( \forall (\text{v, a}) \in \text{set} \ \text{effect-of op}. \ \text{v} \in \text{set} \ (\Psi)_{V^+} \)
\( \land \ \text{a} \in \mathbb{R}_+ \ \Psi \ \text{v} \)
using is-valid-operator-sas-plus-then(3,4)
by fast
}\ note nb4 = this
show \ ?\text{case}
proof \ (\text{cases} \ STRIPS-Semantics.\text{are-all-operators-applicable} \ ?I' \ \text{ops'} \\
\land \ STRIPS-Semantics.\text{are-all-operator-effects-consistent} \ \text{ops'})
case True
{ have \( \text{dom I} \subseteq \text{set} \ (\Psi)_{V^+} \)
using Cons.prems(2)
by blast
hence \( \varphi S^{-1} \Psi \ ?I' = I \)
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(1) nb₁, of I] True
by simp
moreover have \( (\varphi_P^{-1} \Psi (\text{ops'} \# \pi)) = ?\text{ops} # (\varphi_P^{-1} \Psi \pi) \)
strips-parallel-plan-to-sas-plus-parallel-plan-def
SAS-Plus-STRIPS.strips-op-to-sasp-def
by simp
ultimately have execute-parallel-plan-sas-plus I \( (\varphi_P^{-1} \Psi (\text{ops'} \# \pi)) \)
= execute-parallel-plan-sas-plus ?J \( (\varphi_P^{-1} \Psi \pi) \)
by force

} note nb₅ = this
— Show the goal using the IH.

{ have dom-J-subset-eq-vs: dom ?J \subseteq \text{set } ((\Psi)_{\forall})
using sas-plus-equivalent-to-strips-i-a-IX[of assms(1)] nb₂ nb₄
by blast
moreover {
  have set \( ((\Psi)_{\forall}) \subseteq \text{dom } (\text{range-of } \Psi) \)
  using is-valid-problem-sas-plus-then(1)[of assms(1)]
  by fastforce
  moreover have \( \forall v \in \text{dom } I, \text{the } (I v) \in \text{set } (\text{the } (\text{range-of } \Psi v)) \)
  using Cons.prems(2, 3) assms(1) set-the-range-of-is-range-of-sas-plus-if
  by force
  moreover have \( \forall \text{op } \in \text{set } ?\text{ops}, \forall (v, a) \in \text{set } (\text{effect-of } \text{op}). \)
  \( v \in \text{set } ((\Psi)_{\forall}) \wedge a \in \text{set } (\text{the } (\text{?D } v)) \)
  using set-the-range-of-is-range-of-sas-plus-if assms(1) nb₄
  by fastforce
  moreover have v-in-dom-J-range: \( \forall v \in \text{dom } ?J, \text{the } (?J v) \in \text{set } (\text{the } (\text{?D } v)) \)
  using sas-plus-equivalent-to-strips-i-a-X[of 
  I set \( ((\Psi)_{\forall}) \text{?D } ?\text{ops}, \text{of } \text{Cons.prems(2)} \) calculation(1, 2, 3)
  by fastforce
  { fix v
    assume v \in \text{dom } ?J
    moreover have v \in \text{set } ((\Psi)_{\forall})
    using nb₂ calculation dom-J-subset-eq-vs
    by blast
    moreover have set (\text{the } (\text{range-of } \Psi v)) = \mathcal{R}_+ \Psi v
    using set-the-range-of-is-range-of-sas-plus-if[of assms(1)]
    calculation(2)
    by presburger
    ultimately have the \( (?J v) \in \mathcal{R}_+ \Psi v \)

141
ultimately have \( \forall v \in \text{dom } ?J \text{ the } (?J v) \in \mathcal{R}_+ \Psi v \)
by fast

moreover have \( \forall \text{ops}' \in \text{set } \pi. \forall \text{op}' \in \text{set } \text{ops}'. \text{op}' \in \text{set } ((\varphi \Psi)_\Delta) \)
using \( \text{Cons.prems}(6) \)
by simp

moreover {
  have \( ?G' \subseteq_m \text{execute-parallel-plan } ?J' \pi \)
  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 sas-plus-equivalent-to-strips-i-a-XIII[OF Cons.prems(1)] nb₁
  by blast
}

ultimately have \( G \subseteq_m \text{execute-parallel-plan-sas-plus } I \ (\varphi_p^{-1} \Psi (\text{ops}'
# \pi)) \)
using \( \text{Cons.IH}[\text{of } ?J, \text{ OF } \text{Cons.prems}(1) - - \text{Cons.prems}(4, 5)] \)
\( \text{Cons.prems}(6) \) nb₆
by presburger

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

next
  case False
  then have \( ?G' \subseteq_m ?I' \)
  using \( \text{Cons.prems}(7) \)
  by force

moreover {
  have \( \text{dom } I \subseteq \text{set } ?u \)
  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 sas-plus-equivalent-to-strips-i-a-VIII[OF Cons.prems(1) - - Cons.prems(4)] nb₁
  False
  by force
}

moreover {
  have \( (\varphi_p^{-1} \Psi (\text{ops}' \# \pi)) = \text{ops}' \# (\varphi_p^{-1} \Psi \pi) \)
strips-parallel-plan-to-sas-plus-parallel-plan-def
SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
by simp
hence \( G \subseteq_m \text{execute-parallel-plan-sas-plus } I \ (\text{ops}' \# (\varphi_p^{-1} \Psi \pi)) \)
\( \leftrightarrow G \subseteq_m I \)
ultimately show the thesis

strips-parallel-plan-to-sas-plus-parallel-plan-def
SAS-Plus-STRIPS.strips-op-to-sas-plus-def
strips-op-to-sas-plus-def

qed

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+ \( \Psi \) problem to the corresponding STRIPS problem \( \Pi \) (this is implicitly also valid by lemma is-valid-problem-sas-plus-then-strips-transformation-too); then,
2. get a solution \( \pi \)—if it exists—for the induced STRIPS problem by executing SATPlan; and finally,
3. convert \( \pi \) back to a solution \( \psi \) for the SAS+ problem.

\[\text{lemma sas-plus-equivalent-to-strips-i:}\]
\[\text{assumes is-valid-problem-sas-plus } \Psi \]
\[\text{and STRIPS-Semantics.is-parallel-solution-for-problem}\]
\[\langle \varphi \Psi \rangle \pi\]
\[\text{shows goal-of } \Psi \subseteq_m \text{ execute-parallel-plan-sas-plus}\]
\[\langle \text{sas-plus-problem.initial-of } \Psi \rangle \langle \varphi P^{-1} \Psi \pi \rangle\]
\[\text{proof —}\]
\[\text{let } \text{?vs = variables-of } \Psi\]
\[\text{and } \text{?I = initial-of } \Psi\]
\[\text{and } \text{?G = goal-of } \Psi\]
\[\text{let } \text{?I = } \varphi \Psi\]
\[\text{let } \text{?G' = strips-problem.goal-of } \Pi\]
\[\text{and } \text{?I' = strips-problem.initial-of } \Pi\]
\[\text{let } \text{?psi = } \varphi P^{-1} \Psi \pi\]
\[\text{have dom } ?I \subseteq set ?vs\]
\[\text{using is-valid-problem-sas-plus-then(3) assms(1)}\]
\[\text{by auto}\]
\[\text{moreover have } \forall v \in \text{ dom } ?I. \text{ the } \langle ?I v \rangle \in \mathcal{R}_+ \Psi v\]
\[\text{using is-valid-problem-sas-plus-then(4) assms(1) calculation }\]
\[\text{by auto}\]
\[\text{moreover have } \text{ dom } ?G \subseteq set ?vs \text{ and } \forall v \in \text{ dom } ?G. \text{ the } \langle ?G v \rangle \in \mathcal{R}_+ \Psi v\]
\[\text{using is-valid-problem-sas-plus-then(5, 6) assms(1)}\]
\[\text{by blast+}\]
\[\text{moreover have } \forall ops' \subseteq set \pi. \forall op' \subseteq set ops'. \text{ op' } \in \text{ set } (\langle \Pi \rangle_\sigma)\]
using is-parallel-solution-for-problem-operator-set[OF 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 $?G' = \varphi_S \Psi ?G$ 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[OF 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 (list-all (\lambda op. ListMem op (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[OF assms(1)]
  by simp
  have nb1: $\forall op' \in \text{set} (\langle \langle \Pi \rangle \rangle_{\mathcal{O}}). \ (\exists op \in \text{set} ?\text{ops}. \ \text{op'} = (\varphi_{O} \Psi \text{op}))$
  by auto
  {
    fix $\text{ops'} \ \text{op'} \ \text{op}$
    assume $\text{ops'} \in \text{set} \ \pi$ 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[OF assms(2)]
    by simp
    then obtain $\text{op}$ where $\text{op} \in \text{set} (\langle \langle \Psi \rangle \rangle_{\mathcal{O}^{-}}) \ \text{and} \ \text{op'} = (\varphi_{O} \Psi \text{op})$
    by auto
    then have $(\varphi_{O^{-1}} \Psi \text{op'}) \in \text{set} (\langle \Psi \rangle_{\mathcal{O}^{-}})$
    using sas-plus-operator-inverse-is[OF assms(1)]
    by presburger
  }
  thus $?\text{thesis}$
  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
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 $?I = \text{initial-of} \ \Psi$

and $?G = \text{goal-of} \ \Psi$

and $?ops = \text{operators-of} \ \Psi$

and $?\psi = \varphi P^{-1} \Psi \pi$

show ?thesis

unfolding is-parallel-solution-for-problem-def Let-def

proof (rule conjI)

show $?G \subseteq_m \text{execute-parallel-plan-sas-plus}\ I ?\psi$

using sas-plus-equivalent-to-strips-i[OF assms].

next

show list-all (list-all ($\lambda o. \ \text{ListMem} \ o \ \text{ops}$)) $?\psi$

using sas-plus-equivalent-to-strips-ii[OF assms].

qed

qed

**Private Lemma**

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

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

and $\forall \ o \in\ \text{set} \ \text{ops} \ \ o \in\ \text{set} \ \text{((}\Psi\text{)}_{O+})$

and $\ o' \in\ \text{set} \ [\varphi O \ \Psi \ \ o \ \leftrightarrow \ \text{ops}]$

obtains $o$ where $o \in\ \text{set} \ \text{ops}$

and $o' = \varphi O \ \Psi \ \ o$

**Proof** –

let $?\Pi = \varphi \ \Psi$

let $?\text{ops} = \text{operators-of} \ \Psi$

obtain $o$ where $o \in\ \text{set} \ \text{ops}$ and $o' = \varphi O \ \Psi \ \ o$

using assms(3)

by auto

thus ?thesis

using that

by blast
qed

private corollary strips-equivalent-to-sas-plus-i-a-II:
assumes \( \text{is-valid-problem-sas-plus } \Psi \)
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 \( \text{is-valid-operator-strips } (\varphi \ \Psi) \ \text{op}' \)
proof –
let \( \Pi = \varphi \ \Psi \)
let \( \text{?ops} = \text{operators-of } \Psi \)
and \( \text{?ops}' = \text{strips-problem.operators-of } \Pi \)
obtain \( \text{op where \ op-in: \ op} \in \text{set \ ops} \ \text{and \ op}'-\text{is}: \ \text{op}' = \varphi_O \ \Psi \ \text{op} \)
using strips-equivalent-to-sas-plus-i-a-I [OF assms].
then have \( \text{nb: \ op}' \in \text{set } ((\varphi \ \Psi)_O) \)
using assms(2) \( \text{op-in \ op}'-\text{is} \)
by fastforce
thus \( \text{op}' \in \text{set } ((\varphi \ \Psi)_O) \)
and \( \text{is-valid-operator-strips } \Pi \ \text{op}' \)
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 assms(1)]
unfolding list-all-iff.
thus \( \text{is-valid-operator-strips } \Pi \ \text{op}' \)
using \( \text{nb} \)
by fastforce
qed fastforce
qed

lemma strips-equivalent-to-sas-plus-i-a-III:
assumes \( \text{is-valid-problem-sas-plus } \Psi \)
and \( \forall \, \text{op} \in \text{set \ ops}, \ \text{op} \in \text{set } ((\Psi)_O^+) \)
shows \( \text{execute-parallel-operator } (\varphi_S \ \Psi \ s) \ [\varphi_O \ \Psi \ \text{op}, \ \text{op} \leftarrow \text{ops}] \)
= \( (\varphi_S \ \Psi \ (\text{execute-parallel-operator-sas-plus } s \ \text{ops})) \)
proof –
\{ 
\begin{align*}
\text{fix } \text{op} \ s \\
\text{assume } \text{op} \in \text{set } ((\Psi)_O^+) \\
\text{moreover have } (\varphi_O \ \Psi \ \text{op}) \in \text{set } ((\varphi \ \Psi)_O) \\
\text{using calculation} \\
\text{by simp} \\
\text{moreover have } (\varphi_S \ s) \ + \ \text{map-of } (\text{effect-to-assignments } (\varphi_O \ \Psi \ \text{op})) \\
= (\varphi_S \ s \ + \ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \ \Psi \ (\varphi_O \ \Psi \ \text{op})))) \\
\text{using sas-plus-equivalent-to-strips-i-a-XI [OF assms(1)] calculation(2)} \\
\text{by blast} \\
\text{moreover have } (\varphi_O^{-1} \ \Psi \ (\varphi_O \ \Psi \ \text{op})) = \text{op} \\
\text{using sas-plus-operator-inverse-is [OF assms(1) calculation(1)]}. \\
\text{ultimately have } (\varphi_S \ s) \gg (\varphi_O \ \Psi \ \text{op})
\end{align*}
\}
\[ (\varphi_S \Psi (s \gg_+ op)) \]

unfolding \( \text{execute-operator-def execute-operator-sas-plus-def} \)
by simp

}\ note \( nb_1 = \) this

show \( ?\)thesis

using \( \text{assms} \)

proof (induction \( \text{ops arbitrary} \): \( s \))

case \( \text{Nil} \)

then show \( ?\)case

unfolding \( \text{execute-parallel-operator-def execute-parallel-operator-sas-plus-def} \)
by simp

next

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

let \( ?t = s \gg_+ \) op

let \( ?s' = \varphi_S \Psi s \)

and \( \text{ops}' = [\varphi_O \Psi \text{ op} \leftarrow \text{ op} \neq \text{ ops}] \)

let \( ?t' = ?s' \gg_+ (?\varphi_O \Psi \text{ op}) \)

have \( \text{execute-parallel-operator} \ ?s' \ ?\text{ops}' \)

= \( \text{execute-parallel-operator} \ ?t' [\varphi_O \Psi x. x \leftarrow \text{ops}] \)

unfolding \( \text{execute-operator-def} \)
by simp

moreover have \( (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ s \ (\text{op} \neq \text{ ops}))) \)

= \( (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ ?t \ \text{ops}))) \)

unfolding \( \text{execute-operator-sas-plus-def} \)
by simp

moreover \{ 
have \( ?t' = (\varphi_S \Psi \ ?t) \)

using \( nb_1 \ Cons.prems(2) \)

by simp

hence \( \text{execute-parallel-operator} \ ?t' [\varphi_O \Psi x. x \leftarrow \text{ops}] \)

= \( (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus} \ ?t \ \text{ops})) \)

using \( Cons.IH[\text{of} \ ?t] \ Cons.prems \)

by simp

\}

ultimately show \( ?\)case

by argo

qed

qed

private lemma \( \text{strips-equivalent-to-sas-plus-i-a-IV} \): 

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

and \( \forall \text{ op} \in \text{ set operations}. \text{ op} \in \text{ set } ((\Psi)_\text{O}^+ \) 

and \( \text{are-all-operators-applicable-in I} \ \text{ops} \)

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

shows \( \text{STRIPS-Semantics. are-all-operators-applicable} \ (\varphi_S \Psi \ I) [\varphi_O \Psi \text{ op} \leftarrow \text{ ops}] \)

\( \land \text{STRIPS-Semantics. are-all-operator-effects-consistent} [\varphi_O \Psi \text{ op} \leftarrow \text{ ops}] \)

proof –
let \( \Psi \) vs = variables-of \( \Psi \)
and \( \Psi \) ops = operators-of \( \Psi \)

let \( \varphi \) \( S \) \( \Psi \) I
and \( \varphi \) \( O \) \( \Psi \) op ← \( \Psi \) ops

have \( nb_1 : \forall op \in \text{set ops}. \, \text{is-operator-applicable-in} \ I \ op \)
using \( \text{assms}(3) \)
unfolding \( \text{are-all-operators-applicable-in-def} \) list-all-iff
by blast

have \( nb_2 : \forall op \in \text{set ops}. \, \text{is-valid-operator-sas-plus} \ \Psi \ op \)
using \( \text{is-valid-problem-sas-plus-then}(2) \) \( \text{assms}(1, 2) \)
unfolding \( \text{is-valid-operator-sas-plus-def} \)
by auto

have \( nb_3 : \forall op \in \text{set ops}. \, \text{map-of}(\text{precondition-of} \ op) \subseteq_m I \)
using \( nb_1 \)
unfolding \( \text{is-operator-applicable-in-def} \) list-all-iff
by blast

\{ fix \( op_1 \) \( op_2 \)
assume \( op_1 \in \text{set ops} \) and \( op_2 \in \text{set ops} \)
hence \( \text{are-operator-effects-consistent} \ op_1 \ op_2 \)
using \( \text{assms}(3) \)
unfolding \( \text{are-all-operator-effects-consistent-def} \) list-all-iff
by blast \}

note \( nb_4 = \text{this} \)

\{ fix \( op_1 \) \( op_2 \)
assume \( op_1 \in \text{set ops} \) and \( op_2 \in \text{set ops} \)
hence \( \forall (v, a) \in \text{set (effect-of} \ op_1). \, \forall (v', a') \in \text{set (effect-of} \ op_2). \)
\( v \neq v' \lor a = a' \)
using \( nb_4 \)
unfolding \( \text{are-operator-effects-consistent-def} \) \( \text{Let-def} \) list-all-iff
by presburger \}

note \( nb_5 = \text{this} \)

\{ fix \( op_1' \) \( op_2' \) \( I \)
assume \( op_1' \in \text{set \?ops' } \)
and \( op_2' \in \text{set \?ops' } \)
and \( \exists (v, a) \in \text{set (add-effects-of} \ op_1'). \, \exists (v', a') \in \text{set (delete-effects-of} \ op_2'). \)
\( (v, a) = (v', a') \)
morerover obtain \( op_1 \) \( op_2 \)
where \( op_1 \in \text{set ops} \)
and \( op_1' = \varphi \) \( O \) \( \Psi \) \( op_1 \)
and \( op_2 \in \text{set ops} \)
and \( op_2' = \varphi \) \( O \) \( \Psi \) \( op_2 \)
using \( \text{strips-equivalent-to-sas-plus-i-a-I[OF} \ \text{assms}(1, 2)] \) \( \text{calculation}(1, 2) \)
by auto

morerover have \( \text{is-valid-operator-sas-plus} \ \Psi \ op_1 \)
and \( \text{is-valid-operator-op_2: is-valid-operator-sas-plus} \ \Psi \ op_2 \)
using \( \text{calculation}(4, 6) \) \( nb_2 \)
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 \((v', a') \in \bigcup \{(v, a) \in \text{set (effect-of } op_2)\}
\{ (v, a') | a'. a' \in (\mathcal{R}_+ \Psi v') \land a' \neq a' \}\)
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 \((v', a') \in \{ (v'', a'') | a''. a''' \in (\mathcal{R}_+ \Psi v'') \land a''' \neq a'' \}\)
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 \(\exists a''. (v', a'') \in \text{set (effect-of } op_2) \land a' \in (\mathcal{R}_+ \Psi v')
\land a' \neq a''\)
by blast
}\)
moreover obtain \(a''\) where \(a' \in \mathcal{R}_+ \Psi v'\)
and \((v', a'') \in \text{set (effect-of } op_2)\)
and \(a' \neq a''\)
using calculation(16)
by blast
moreover have \(\exists (v, a) \in \text{set (effect-of } op_1). (\exists (v', a') \in \text{set (effect-of } op_2).
v = v' \land a \neq a')\)
using calculation(13, 14, 15, 17, 18, 19)
by blast
— TODO slow.
ultimately have \(\exists op_1 \in \text{set ops}. \exists op_2 \in \text{set ops}. \neg\text{are-operator-effects-consistent}
\)
\(op_1 op_2\)
unfolding are-operator-effects-consistent-def list-all-iff
by fastforce
\}

\text{note nb}_0 = \text{this}
show thesis
proof (rule conjI)
{  
  fix op'
  assume op' ∈ set ?ops'
  moreover obtain op where op-in: op ∈ set ops
      and op'-is: op' = φo Ψ op
      and op'-in: op' ∈ set ((φ Ψ) o)
      and is-valid-op': is-valid-operator-strips (φ Ψ) op'
      using strips-equivalent-to-sas-plus-i-a-I[OF assms(1, 2)]
      strips-equivalent-to-sas-plus-i-a-II[OF assms(1, 2)]
      calculation
      by metis
  moreover have is-valid-op: is-valid-operator-sas-plus Ψ op
      using nb2 calculation(2)..
  }
  fix v a
  assume v-a-in-preconditions': (v, a) ∈ set (strips-operator.precondition-of op')
  have v-a-in-preconditions: (v, a) ∈ set (precondition-of op)
      using op'-is
      unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def Let-def
      using v-a-in-preconditions'
      by force
  moreover have v ∈ set ?ves and a ∈ R_+ Ψ v
      using is-valid-operator-sas-plus-then(1,2) is-valid-op
calculation(1)
      by fastforce+
  moreover {
    have ∀ (v, a) ∈ set (precondition-of op). ∀ (v', a') ∈ set (precondition-of op).
      v ≠ v' ∨ a = a'
      using is-valid-operator-sas-plus-then(5) is-valid-op
      by fast
      hence map-of (precondition-of op) v = Some a
      using map-of-constant-assignments-defined-if[OF - v-a-in-preconditions]
      by blast
  }
  moreover have v ∈ dom (map-of (precondition-of op))
      using calculation(4)
      by blast
  moreover have I v = Some a
      using nb3
      unfolding map-le-def
      using op-in calculation(4, 5)
      by metis
  moreover have (v, a) ∈ dom ?I'
      using state-to-strips-state-dom-element-iff[OF assms(1)]
calculation(2, 3, 6)
      by simp
  }
ultimately have \( ?I' (v, a) = \text{Some True} \)

using state-to-strips-state-range-is[OF assms(1)]

by simp

}\)

hence STRIPS-Representation.is-operator-applicable-in \( ?I' \) op'

unfolding

STRIPS-Representation.is-operator-applicable-in-def

Let-def list-all-iff

by fast

}\)

thus are-all-operators-applicable \( ?I' \) \( ?ops' \)

unfolding are-all-operators-applicable-def list-all-iff

by blast

next

\{ fix \( op_1' \) \( op_2' \)
assume \( op_1' \)'-in-ops': \( op_1' \) \( \in \) set \( ?ops' \) and \( op_2' \)'-in-ops': \( op_2' \) \( \in \) set \( ?ops' \)

have STRIPS-Semantics.are-operator-effects-consistent \( op_1' \) \( op_2' \)

unfolding 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 \( \neg\neg\text{list-ex} (\lambda x. \text{list-ex} ((=) x) (\text{delete-effects-of} \ op_2' )) \)

\( (\text{add-effects-of} \ op_1' ) \)

then have \( \exists (v, a) \in \text{set} (\text{delete-effects-of} \ op_2' ). (v, a) = (v', a') \)

unfolding list-ex-iff

by fastforce

then obtain \( op_1 \) \( op_2 \) where \( op_1 \) \( \in \) set \( ops \)

and \( op_2 \) \( \in \) set \( ops \)

and \( \neg\text{are-operator-effects-consistent} \ op_1 \ op_2 \)

using \( \text{nb}_6[\text{OF} \ op_1' \) 'in-ops' \( op_2' \) '-in-ops' \]

by blast

thus False

using \( \text{nb}_4 \)

by blast

qed

next

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 \( \neg\neg\text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{add-effects-of} \ op_2' )) \)

\( (\text{delete-effects-of} \ op_1' ) \)

then have \( \exists (v, a) \in \text{set} (\text{delete-effects-of} \ op_1' ). (v, a) = (v', a') \)

unfolding list-ex-iff

by fastforce

151
then obtain $op_1, op_2$ where $op_1 \in \text{set ops}$
  and $op_2 \in \text{set ops}$
  and $\neg$are-operator-effects-consistent $op_1, op_2$
  using $n_b([OF \ op_2\text{-in-ops}' \ op_1\text{-in-ops}'])$
  by blast
thus $False$
  using $n_b$
  by blast
qed

qed

private lemma strips-equivalent-to-sas-plus-i-a-V:
assumes $is\text{-valid\text{-problem\text{-sas\text{-plus}} }\Psi}$
  and $\forall \ op \in \text{set ops}. \ op \in \text{set }((\Psi)_{O+})$
  and $\neg(\text{are\text{-all\text{-operators\text{-applicable\text{-in s ops}}}}}$
    $\land$ are-all-operator-effects-consistent $\text{ops}$)
shows $\neg(\text{STRIPS\text{-Semantics. are\text{-all\text{-operators\text{-applicable}}}(\varphi_S \ \Psi \ s)}$ $\varphi_O \ \Psi \ \text{op} \ \text{op} \leftarrow \text{ops}$
  $\land$ STRIPS-Semantics.are-all-operator-effects-consistent $\varphi_O \ \Psi \ \text{op} \ \text{op} \leftarrow \text{ops})$
proof
  let $\ ?vs = \text{variables\text{-of } }\Psi$
  and $\ ?ops = \text{operators\text{-of } }\Psi$
  let $\ ?s' = \varphi_S \ \Psi \ s$
  and $\ ?ops' = [\varphi_O \ \Psi \ \text{op} \ \text{op} \leftarrow \text{ops}]$
  { fix $\text{op}$
    assume $\text{op} \in \text{set ops}$
    hence $\exists \ \text{op}' \in \text{set }\ ?ops'$. $\ \text{op}' = \varphi_O \ \Psi \ \text{op}$
      by simp
  } note $n_b = \text{this}$
  { fix $\text{op}$
    assume $\text{op} \in \text{set ops}$
    then have $\text{op} \in \text{set }((\Psi)_{O+})$
      using $\text{assms}(2)$
      by blast
    then have $\text{is\text{-valid\text{-operator\text{-sas\text{-plus}} }\Psi \ \text{op}}$
      using $\text{is\text{-valid\text{-problem\text{-sas\text{-plus\text{-then}}}(2)}\ \text{assms}(1)$
      unfolding $\text{is\text{-valid\text{-operator\text{-sas\text{-plus\text{-def}}}$}
        by auto
    hence $\forall (v, a) \in \text{set }\text{(precondition\text{-of op})}. \ \forall (v', a') \in \text{set }\text{(precondition\text{-of op})}$.
      $v \neq v' \lor a = a'$
      using $\text{is\text{-valid\text{-operator\text{-sas\text{-plus\text{-then}}(5)$}

152
unfolding is-valid-operator-sas-plus-def
by fast
}
ote nb2 = this
{
consider (A) ¬are-all-operators-applicable-in s ops
| (B) ¬are-all-operator-effects-consistent ops
using assms(3)
by blast
hence ¬STRIPS-Semantics.are-all-operators-applicable ?s' ?ops'
∨ ¬STRIPS-Semantics.are-all-operator-effects-consistent ?ops'
proof (cases)
case A
then obtain op where op-in: op ∈ set ops
and not-precondition-map-le-s: ¬(map-of (precondition-of op) ⊆m s)
using A
unfolding are-all-operators-applicable-in-def list-all-iff
is-operator-applicable-in-def
by blast
then obtain op' where op'-in: op' ∈ set ?ops' and op'-is: op' = ϕO Ψ op
using nb1
by blast
have ¬are-all-operators-applicable ?s' ?ops'
proof (rule ccontr)
assume ¬¬are-all-operators-applicable ?s' ?ops'
then have all-operators-applicable: are-all-operators-applicable ?s' ?ops'
by simp
moreover {
fix v
assume v ∈ dom (map-of (precondition-of op))
moreover obtain a where map-of (precondition-of op) v = Some a
using calculation
by blast
moreover have (v, a) ∈ set (precondition-of op)
using map-of-SomeD[OF calculation(2)]
moreover have (v, a) ∈ set (strips-operator.precondition-of 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) = Some True
using all-operators-applicable calculation
unfolding are-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) ∈ dom ?s'
using calculation(5)
}
by blast
moreover have \((v, a) \in \text{set (precondition-of op)}\)
  using \(\text{op'}-\text{is calculation(3)}\)
unfolding \sasp-op-to-strips-def \ Let-def
by fastforce
moreover have \(v \in \text{set } ?v s\)
  and \(a \in \mathbb{R}_+ \Psi v\)
  and \(s v \neq \text{None}\)
using \state-to-strips-state-dom-element-iff[\text{OF }\text{assms}(1)]
calculation(6)
by simp+
moreover have \(?s\' (v, a) = \text{Some (the } (s v) = a)\)
  using \state-to-strips-state-range-is[\text{OF }\text{assms}(1) \text{ 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) \text{ option.collapse}[\text{OF }\text{calculation(10)}]
by argo
moreover have \(\text{map-of (precondition-of op) } v = \text{Some } a\)
  using \map-of-constant-assignments-defined-if[\text{OF }\text{nb2[OF }\text{op-in] calculation(?)}].
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 \(\text{False}\)
using \not-precondition-map-le-s
by simp
qed
thus \(?\text{thesis}\)
by simp
next
case \(B\)
{
  obtain \(\text{op}_1 \text{ op}_2 \ v \ v' \ a \ a'\)
    where \(\text{op}_1 \in \text{set }\text{ops}\)
    and \(\text{op}_2-\text{in: } \text{op}_2 \in \text{set }\text{ops}\)
    and \(\text{v-a-in: } (v, a) \in \text{set (effect-of op}_1)\)
    and \(\text{v'-a'-in: } (v', a') \in \text{set (effect-of op}_2)\)
    and \(\text{v-is: } v = v' \text{ and a-is: } a \neq a'\)
using \(B\)
unfolding \are-all-operator-effects-consistent-def
  \are-operator-effects-consistent-def list-all-iff \ Let-def
by blast
moreover obtain \(\text{op}_1' \text{ op}_2' \text{ where } \text{op}_1' \in \text{set } ?\text{ops}' \text{ and } \text{op}_1' = \varphi_O \Psi \text{op}_1\)
and \( \text{op}_1 \)' \(\in\) \(\text{set } ?\text{ops}'\) \text{ and } \(\text{op}_2 \)'-is:: \(\text{op}_2 \)' = \(\varphi_\text{O} \Psi \text{ op}_2\)

using \(\text{nb}_1[\text{OF calculation}(1)] \text{ nb}_1[\text{OF calculation}(2)]\)

by blast

moreover have \((v, a) \in \text{set } (\text{add-effects-of } \text{op}_1')\)
using \(\text{calculation}(3, 8)\)

unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
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 \(a \in \mathbb{R}_+ \Psi v\)
using \(\text{is-valid-operator-sas-plus-then}(4) \text{ calculation } v-a-\text{in}\)
unfolding \(\text{is-valid-operator-sas-plus-def}\)
by fastforce

ultimately have \((v, a) \in \text{set } (\text{delete-effects-of } \text{op}_2')\)
using \(\text{sasp-op-to-strips-set-delete-effects-is}[\text{of } \Psi \text{ op}_2]\)
\(v'-a'-\text{in } v\)-is a-is

using \(\text{op}_2'-\text{is}\)
by blast

\}
— TODO slow.

ultimately have \(\exists \text{ op}_1 \)' \(\in\) \(\text{set } ?\text{ops}'\). \(\exists \text{ op}_2 \)' \(\in\) \(\text{set } ?\text{ops}'\).
\(\exists (v, a) \in \text{set } (\text{delete-effects-of } \text{op}_2')\). \(\exists (v', a') \in \text{set } (\text{add-effects-of } \text{op}_1')\).
\((v, a) = (v', a')\)
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

\}

thus \(?\text{thesis}\)
by blast

qed

lemma strips-equivalent-to-sas-plus-i-a:
assumes \(\text{is-valid-problem-sas-plus } \Psi\)
and dom I \subseteq set ((\Psi)V_+) \\
and \forall v \in dom I. (I v) \in R_+ \Psi v \\
and \forall G \subseteq R_+ \Psi v \\
and \forall v \in dom G. the (G v) \in R_+ \Psi v \\
and \forall \text{ops} \in set \psi. \forall op \in \text{set ops. op} \in set ((\Psi)O_+) \\
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 ?thesis \\
using assms \\
proof (induction \psi arbitrary: I) \\
\text{case Nil} \\
let ?I' = \varphi_S \Psi I \\
have \text{G} \subseteq_m ?I' \\
using Nil \\
by simp \\
moreover have \text{G'} \subseteq_m ?I' \\
using state-to-strips-state-map-le-iff[OF Nil.prems(1,4,5)] \\
calculation.. \\
ultimately show ?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 \\
\text{case } (\text{Cons ops} \psi) \\
let ?vs = \text{variables-of } \Psi \\
and ?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 ?ops' = [\varphi_O \Psi \text{ op. op} \leftarrow \text{ops}] \\
\{ \\
fix \text{op} \text{, v } a \\
assume \text{op} \in \text{set ops and } (v, a) \in \text{set (effect-of op) } \\
moreover have \text{op} \in \text{set } ?\text{ops} \\
using Cons.prems(6) calculation(1) \\
by simp \\
moreover have \text{is-valid-operator-sas-plus } \Psi \text{ 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 R_+ \Psi v \\
using is-valid-operator-sas-plus-then(3,4) \\
by fastforce+ 
\} note nb1 = this
show \( ?\text{case} \)

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

\textbf{case True}

\begin{enumerate}
\item
have \((\varphi_P \Psi (\text{ops} \# \psi)) = ?\text{ops}' \# (\varphi_P \Psi \psi)\)
unfolding \text{SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def}
\text{Sas-op-to-strips-def}
\text{SAS-Plus-STRIPS.sasp-op-to-strips-def}
by \text{simp}

moreover have \( \forall \text{op} \in \text{set \ ops}. \ \text{op} \in \text{set ((}\Psi)_{\lor +}) \)
using \text{Cons.prems(6)}
by \text{simp}

moreover have \text{STRIPS-Semantics.are-all-operators-applicable ?I'} ?\text{ops}'
and \text{STRIPS-Semantics.are-all-operator-effects-consistent ?ops}'
using \text{strips-equivalent-to-sas-plus-i-a-IV[OF Cons.prems(1) \- True]}
\end{enumerate}

\text{calculation}

by \text{blast+}

ultimately have \text{execute-parallel-plan ?I'} ?\pi'
= \text{execute-parallel-plan (execute-parallel-operator ?I' ?ops') (}\varphi_P \Psi \psi\)
by \text{fastforce}

\} — NOTE Instantiate the IH on the next state of the SAS+ execution

\text{execute-parallel-operator-sas-plus I \text{ops}}.

moreover

\begin{enumerate}
\item
have \text{dom \ I} \subseteq \text{set (sas-plus-problem.variables-of \( \Psi \))}
using \text{Cons.prems(2)}
by \text{blast}

moreover have \( \forall \text{op} \in \text{set \ ops}. \ \forall (v, a) \in \text{set (effect-of \ op)}.
\ v \in \text{set ((}\Psi)_{\lor +}) \)
using \text{nb1(1)}
by \text{blast}

ultimately have \text{dom ?J} \subseteq \text{set ((}\Psi)_{\lor +})
using \text{sas-plus-equivalent-to-strips-i-a-IX[of I set ?vs]}
by \text{simp}
\} note \text{nb2 = this}

moreover \{  

have \text{dom \ I} \subseteq \text{set (sas-plus-problem.variables-of \( \Psi \))}
using \text{Cons.prems(2)}
by \text{blast}

moreover have \text{set (sas-plus-problem.variables-of \( \Psi \))}
\( \subseteq \text{dom (range-of \( \Psi \))} \)
using \text{is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)}
by \text{auto}

moreover \{  
fix \text{v}
\}

\end{enumerate}

157
assume \( v \in \text{dom } I \)
moreover have \( v \in \text{set } ((\Psi)_v)_+ \)
  using Cons.prems(2) calculation
  by blast
ultimately have the \((I v) \in \text{set } ((\text{range-of } \Psi v))\)
  using Cons.prems(3)
  using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
  by blast
}
moreover have \( \forall \text{ op } \in \text{set } \text{ops}. \forall (v, a) \in \text{set } ((\text{effect-of op})) \in \text{set } (\text{the } (\text{range-of } \Psi v) \wedge a \in \text{set } ((\text{the } \text{range-of } \Psi v)))\)
using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)] nb1(1) nb1(2)
  by force
moreover have \( \forall v \in \text{dom } ?J. \text{ the } (?J v) \in \text{set } ((\text{the } \text{range-of } \Psi v))\)
using sas-plus-equivalent-to-strips-i-a-X[of I set \(?\text{us range-of } \Psi \text{ ops}]\]
calculation
by fast
moreover { 
  fix \( v \)
  assume \( v \in \text{dom } ?J \)
  moreover have \( v \in \text{set } ((\Psi)_v)_+ \)
    using nb2 calculation
    by blast
  moreover have \( \text{set } ((\text{the } \text{range-of } \Psi v)) = \mathcal{R}_+ \Psi v \)
    using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
    calculation(2)
    by presburger
  ultimately have the \((?J v) \in \mathcal{R}_+ \Psi v \)
    using nb3
    by blast
}
ultimately have \( \forall v \in \text{dom } ?J. \text{ the } (?J v) \in \mathcal{R}_+ \Psi v \)
  by fast
}
moreover have \( \forall \text{ ops } \in \psi. \forall \text{ op } \in \text{set } \text{ops. op } \in \text{set } ?\text{ops}\)
  using 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 \((\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } ?J' (\varphi_F \Psi \psi)\)
  using Cons.IH[of \( ?J, \text{OF Cons.prems(1)} - -\text{ 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 \(nb: G \subseteq_m I\)
using Cons.prems(\?)
by force
moreover {
have \(\pi = ?ops' \# (\varphi_P \Psi \psi)\)
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 Let-def
by auto
moreover have set \(?ops' \subseteq set (\text{strips-problem.operators-of } ?\Pi)\)
using strips-equivalent-to-sas-plus-i-a-II(\[OF assms(1)\]) Cons.prems(\?)
by auto
moreover have \(\neg (\text{STRIPS-Semantics.operators-applicable } ?I' ?ops')\)
\(\land \text{STRIPS-Semantics.operators-effects-consistent } ?ops'\)
using strips-equivalent-to-sas-plus-i-a-V(\[OF assms(1) - False\]) Cons.prems(\?)
by force
ultimately have execute-parallel-plan \(?I' \pi = ?I'\)
by auto
}
moreover have \(?G' \subseteq_m ?I'\)
using state-to-strips-state-map-le-iff(\[OF Cons.prems(1, 4, 5)\]) nb
by blast
ultimately show \(?thesis\)
by presburger
qed
qed

lemma strips-equivalent-to-sas-plus-i:
assumes is-valid-problem-sas-plus \(\Psi\)
and is-parallel-solution-for-problem \(\Psi \psi\)
shows \((\text{strips-problem.goal-of } (\varphi \Psi))) \subseteq_m execute-parallel-plan
\((\text{strips-problem.initial-of } (\varphi \Psi))) (\varphi_P \Psi \psi)\)
proof —
let \(?vs = \text{variables-of } \Psi\)
and \(?ops = \text{operators-of } \Psi\)
and \(?I = \text{initial-of } \Psi\)
and \(?G = \text{goal-of } \Psi\)
let \(\Pi = \varphi \Psi\)
let \(\Pi' = \text{strips-problem.initial-of } ?\Pi\)
and \(?G' = \text{strips-problem.goal-of } ?\Pi\)
have dom \(?I \subseteq set ?vs\)
using is-valid-problem-sas-plus-then(\?) \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) assms(1) calculation
by auto
moreover have \( \text{dom } ?G \subseteq \text{set } ((\Psi)_+) \)
using is-valid-problem-sas-plus-then(3) assms(1)
by auto
moreover have \( \forall v \in \text{dom } ?G \text{. the } (?G v) \in \mathcal{R}_+ \Psi v \)
using is-valid-problem-sas-plus-then(6) assms(1)
by auto
moreover have \( \forall \text{ ops } \in \text{set } \psi \).
\( \forall \text{ op } \in \text{ops. op } \in \text{set } \text{ops} \)
using is-parallel-solution-for-problem-plan-operator-set[OF assms(2)]
by fastforce
moreover have \( ?G \subseteq_m \text{ execute-parallel-plan-sas-plus } ?I \psi \)
using assms(2)
unfolding is-parallel-solution-for-problem-def
by simp
ultimately show \( ?\text{thesis} \)
using strips-equivalent-to-sas-plus-i-a[OF assms(1), of ?I ?G \psi]
unfolding sas-plus-problem-to-strips-problem-def
SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
state-to-strips-state-def
SAS-Plus-STRIPS.state-to-strips-state-def
by force
qed

lemma strips-equivalent-to-sas-plus-ii:
assumes is-valid-problem-sas-plus \( \Psi \)
and is-parallel-solution-for-problem \( \Psi \ \psi \)
shows list-all (list-all (\lambda op. ListMem op (strips-problem.operators-of (\varphi \ \Psi))))
(\varphi_P \ \Psi \ \psi)
proof –
let \( ?\text{ops} = \text{operators-of } \Psi \)
let \( ?\Pi = \varphi \ \Psi \)
let \( ?\text{ops}' = \text{strips-problem.operators-of } ?\Pi \)
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 \( \forall \text{ op } \in \text{set } ?\text{ops. (} \exists \text{ op}' \in \text{set } ?\text{ops}'. \text{op}' = (\varphi_O \ \Psi \ \text{op}) \) \)
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 \( op \in \text{set} \ ((\Psi)_{\ominus}) \)
using \( \text{is-parallel-solution-for-problem-plan-operator-set}(\text{OF assms}(2)) \)
calculation
by blast
moreover obtain \( op' \) where \( op' \in \text{set} \ ?ops' \) and \( op' = (\varphi_O \ \Psi \ \text{op}) \)
using \( \text{nb1} \) calculation(3)
by auto
ultimately have \( (\varphi_O \ \Psi \ \text{op}) \in \text{set} \ ?ops' \)
by blast

thus \( \text{thesis} \)
unfolding \( \text{list-all-iff ListMem-iff Let-def} \)
\( \text{sas-plus-problem-to-strips-problem-def} \)
\( \text{SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def} \)
\( \text{sas-plus-parallel-plan-to-strips-parallel-plan-def} \)
\( \text{SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def} \)
\( \text{sasp-op-to-strips-def} \)
\( \text{SAS-Plus-STRIPS.sasp-op-to-strips-def} \)
\( \text{Let-def} \)
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 \( G' \subseteq \pi \) of the corresponding STRIPS problem, i.e.

\[
G' \subseteq \pi \ \text{execute-parallel-plan} \ I' \ \pi
\]

and that all operators in the transformed plan \( \pi \) are operators of \( \Pi \).

theorem
\( \text{strips-equivalent-to-sas-plus}: \)
assumes \( \text{is-valid-problem-sas-plus} \ \Psi \)
and \( \text{is-parallel-solution-for-problem} \ \Psi \ \psi \)
shows \( \text{STRIPS-Semantics.is-parallel-solution-for-problem} \ (\varphi \ \Psi) \ (\varphi_P \ \Psi \ \psi) \)
proof –
let \( \Pi = \varphi \ \Psi \)
let \( ?I' = \text{strips-problem.initial-of} \ \Pi \)
and \( ?G' = \text{strips-problem.goal-of} \ \Pi \)
and \( ?ops' = \text{strips-problem/operators-of} \ \Pi \)
and \( ?\pi = \varphi_P \ \Psi \ \psi \)
show \( \text{thesis} \)
unfolding \( \text{STRIPS-Semantics.is-parallel-solution-for-problem-def} \)
proof (rule conjI)
show \( ?G' \subseteq \pi \ \text{execute-parallel-plan} \ I' \ ?\pi \)
using \( \text{strips-equivalent-to-sas-plus-i}(\text{OF assms}) \)
by simp
next
  show list-all (list-all (\lambda op. ListMem op ?ops') ?π)
  using strips-equivalent-to-sas-plus-ii[OF assms].
qed

lemma embedded-serial-sas-plus-plan-operator-structure:
  assumes ops \in set \{ embed \ψ \}
  obtains op
  where op \in set \ψ
  and [ϕ O \Ψ op. op \leftarrow ops] = [ϕ O \Ψ op]
proof −
  let ?ψ' = embed \ψ
  {
    have ?ψ' = \{ [op]. op \leftarrow ψ \}
      by (induction ψ; force)
    moreover obtain op where ops = [op] and op \in set ψ
      using assms calculation
      by fastforce
    ultimately have \exists op \in set ψ. [ϕ O \Ψ op. op \leftarrow ops] = [ϕ O \Ψ op]
      by auto
  }
  thus ?thesis
  using that
  by meson
qed

private lemma serial-sas-plus-equivalent-to-serial-strips-i:
  assumes ops \in set \{ ϕ P \Ψ (embed \ψ) \}
  obtains op where op \in set ψ and ops = [ϕ O \Ψ op]
proof −
  let ?ψ' = embed \ψ
  {
    have set \{ ϕ P \Ψ (embed \ψ) \} = \{ [ϕ O \Ψ op. op \leftarrow ops] | ops, ops \in set ?ψ' \}
      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 set-map
      using setcompr-eq-image
      by blast
    moreover obtain ops' where ops' \in set ?ψ' and ops = [ϕ O \Ψ op. op \leftarrow ops']
      using assms(1) calculation
      by blast
    moreover obtain op where op \in set ψ and ops = [ϕ O \Ψ op]
      using embedded-serial-sas-plus-plan-operator-structure calculation(2, 3)
      by blast
    ultimately have \exists op \in set ψ. ops = [ϕ O \Ψ op]
      by meson
  }

162
thus ?thesis using that.

qed

private lemma serial-sas-plus-equivalent-to-serial-strips-ii[simp]:
concat (\varphi_P \Psi (\text{embed } \psi)) = [\varphi_O \Psi \text{ op. op } \leftarrow \psi]

proof
let ?\psi' = List-Supplement.\text{embed } \psi
have concat (\varphi_P \Psi ?\psi') = map (\lambda \text{op. } \varphi_O \Psi \text{ op}) (concat ?\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 Let-def
map-concat
by blast
also have \ldots = map (\lambda \text{op. } \varphi_O \Psi \text{ op}) \psi
unfolding concat-is-inverse-of-embed[of \psi]..
finally show concat (\varphi_P \Psi (\text{embed } \psi)) = [\varphi_O \Psi \text{ op. op } \leftarrow \psi].

qed

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 List-Supplement.\text{embed } \psi is an equivalent parallel solution to \Psi. By parallel plan equivalence, \pi \equiv \varphi_P \Psi List-Supplement.\text{embed } \psi is a parallel solution for the corresponding STRIPS problem \varphi \Psi. Moreover, since 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 concat \pi is a serial solution for \varphi \Psi. Since concat moreover can be shown to be the inverse of List-Supplement.\text{embed}, the term

concat \pi = concat (\varphi_P \Psi (\text{embed } \psi))

can be reduced to the intuitive form

\pi = [\varphi_O \Psi \text{ 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 \text{ op. op } \leftarrow \psi]
proof
  let ?ψ' = embed ψ
  and ?Π = ψ ·
let ?π' = ψ · ?ψ'
let ?π = concat ?π'
  {  
    have SAS-Plus-Semantics.is-parallel-solution-for-problem Ψ ?ψ'
      using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus[OF assms]
      by simp
    hence STRIPS-Semantics.is-parallel-solution-for-problem ?Π ?π'
      using strips-equivalent-to-sas-plus[OF assms 1]
      by simp
  }
moreover have ?π = [ϕ · Π op. op ← ψ]
  by simp
moreover have is-valid-problem-strips ?Π
  using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms 1]
moreover have ∀ ops ∈ set ?π'. ∃ op ∈ set ψ. ops = [ϕ · Π op]
  using serial-sas-plus-equivalent-to-serial-strips-ii[of · Ψ ψ]
  by metis
ultimately show ?thesis
  using STRIPS-Semantics.flattening-lemma[of ?Π]
  by metis
qed

lemma embedded-serial-strips-plan-operator-structure:
  assumes ops' ∈ set (embed π)
  obtains op
    where op ∈ set π and [ϕ · Π op. op ← ops'] = [ϕ · Π op]
proof
  let ?π' = embed π
  {  
    have ?π' = [[[op]. op ← π]
      by (induction π; force)
    moreover obtain op where ops' = [op] and op ∈ set π
      using calculation assms
      by fastforce
    ultimately have ∃ op ∈ set π. [ϕ · Π op. op ← ops'] = [ϕ · Π op]
      by auto
  }
thus ?thesis
  using that
  by meson
qed

private lemma serial-strips-equivalent-to-serial-sas-plus-i:
  assumes ops ∈ set (ϕ · Π (embed π))
  obtains op where op ∈ set π and ops = [ϕ · Π 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 } \in \text{ set } ??\}\}
 } 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’ where } \text{ ops’ } \in \text{ set } ?? \text{ and } \text{ ops } = [\varphi_O^{-1} \Pi \text{ op } \leftarrow \text{ ops’}] 
 using assms(1) calculation
 by blast
 moreover obtain \text{ op where } \text{ op } \in \text{ set } \pi \text{ and } \text{ ops } = [\varphi_O^{-1} \Pi \text{ op}] 
 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 \((\varphi_P^{-1} \Pi \text{ (embed } \pi)) = [\varphi_O^{-1} \Pi \text{ op } \leftarrow \pi]\)
 proof –
 let \(?\pi’ = \text{List-Supplement.embed } \pi\)
 have concat \((\varphi_P^{-1} \Pi ??) = \text{map } (\lambda \text{ op. } \varphi_O^{-1} \Pi \text{ op}) \text{ (concat ??)}\)
 unfolding strips-parallel-plan-to-sas-plus-parallel-plan-def 
 strips-op-to-sasp-def 
 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 ]
 finally show \text{concat } (\varphi_P^{-1} \Pi \text{ (embed } \pi)) = [\varphi_O^{-1} \Pi \text{ op } \leftarrow \pi].

qed

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 \text{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_O^{-1} \Psi \text{ op } \leftarrow
proof
  let ?π' = embed π
  and ?Π = ϕ Ψ
  let ?ψ' = ϕ⁻¹ Ψ ?π'
  let ?ψ = concat ?ψ'
  
  have STRIPS-Semantics.is-parallel-solution-for-problem ?Π ?π'
    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 Ψ ?ψ'
    using sas-plus-equivalent-to-strips[OF assms(1)]
    by simp
  }
  moreover have ?ψ = [ϕ⁻¹ Ψ op. op ← π]
    by simp
  moreover have is-valid-problem-strips ?Π
    using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)].
  moreover have ∀ ops ∈ set ?ψ'. ∃ op ∈ set π. ops = [ϕ⁻¹ Ψ op]
    using serial-strips-equivalent-to-serial-sas-plus-i
    by metis
  ultimately show ?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 ops k ≡ { π. set π ⊆ set ops ∧ length π = 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
  ⇒ ('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'
\[ :: (\text{variable} \times \text{domain}) \text{strips-problem} \Rightarrow \text{nat} \Rightarrow (\text{variable} \times \text{domain}) \text{strips-plan set} \]

\[ \text{where} \quad \text{bounded-solution-set-strips'} \Pi k \equiv \{ \pi. \text{STRIPS-Semantics.is-serial-solution-for-problem} \Pi \pi \wedge \text{length} \pi = k \} \]

**abbreviation** bounded-solution-set-strips
\[ :: (\text{variable} \times \text{domain}) \text{strips-problem} \Rightarrow \text{nat} \Rightarrow (\text{variable} \times \text{domain}) \text{strips-plan set} \]

\[ \text{where} \quad \text{bounded-solution-set-strips} \Pi N \equiv (\bigcup k \in \{0..N\}. \text{bounded-solution-set-strips'} \Pi k) \]

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 \( \lambda \psi. [\varphi_\mathcal{O} \Psi \text{op} \leftarrow \psi] \) on the set of plans \( P_k \equiv \text{bounded-plan-set} (\text{operators-of} \Psi) k \) with length bound \( k \). The injectiveness of \( \text{Sol}_k \equiv \text{bounded-solution-set-sas-plus} \Psi k \)—the set of solutions with length bound \( k \)—then follows from the subset relation \( \text{Sol}_k \subseteq P_k \).

**lemma** sasp-op-to-strips-injective:

**assumes** \( (\varphi_\mathcal{O} \Psi \text{op}_1) = (\varphi_\mathcal{O} \Psi \text{op}_2) \)

**shows** \( \text{op}_1 = \text{op}_2 \)

**proof** —

let \( ?\text{op}_1' = \varphi_\mathcal{O} \Psi \text{op}_1 \)
and \( ?\text{op}_2' = \varphi_\mathcal{O} \Psi \text{op}_2 \)

\{ 
  have \( \text{strips-operator.precondition-of} \ ?\text{op}_1' = \text{strips-operator.precondition-of} \ ?\text{op}_2' \)
  using assms
  by argo
  hence \( \text{sas-plus-operator.precondition-of} \text{op}_1 = \text{sas-plus-operator.precondition-of} \text{op}_2 \)

  unfolding sasp-op-to-strips-def
  SAS-Plus-STRIPS.sasp-op-to-strips-def
  Let-def
  by simp

  moreover {
    have \( \text{strips-operator.add-effects-of} \ ?\text{op}_1' = \text{strips-operator.add-effects-of} \ ?\text{op}_2' \)
    using assms
    unfolding sasp-op-to-strips-def Let-def
    by argo
    hence \( \text{sas-plus-operator.effect-of} \text{op}_1 = \text{sas-plus-operator.effect-of} \text{op}_2 \)

    unfolding sasp-op-to-strips-def Let-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def
    by simp
  } \}

167
ultimately show \( ?\)thesis
  by simp
qed

lemma \textit{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a}:
  assumes is-valid-problem-sas-plus \( \Psi \)
  shows inj-on \((\lambda \psi. [\varphi_O \Psi \ op. \ op \leftarrow \psi])\) (bounded-plan-set (sas-plus-problem.
operators-of \( \Psi \)) \( k \))
  proof
    let \( ?\)ops = sas-plus-problem.
operators-of \( \Psi \)
    and \( \varphi_P = \lambda \psi. [\varphi_O \Psi \ op. \ op \leftarrow \psi] \)
    let \( ?P = \text{bounded-plan-set} \ ?\)ops
    {  
      fix \( \psi_1 \ \psi_2 \)
      assume \( \psi_1\text{-in}: \psi_1 \in ?P \ k \)
      and \( \psi_2\text{-in}: \psi_2 \in ?P \ k \)
      and \( \varphi_P\text{-of-}\psi_1\text{-in-}\varphi_P\text{-of-}\psi_2: \ (\varphi_P \ \psi_1) = (\varphi_P \ \psi_2) \)
      hence \( \psi_1 = \psi_2 \)
      proof (induction \( k \) arbitrary: \( \psi_1 \ \psi_2 \))
        case 0
          then have length \( \psi_1 = 0 \)
          and length \( \psi_2 = 0 \)
          using \( \psi_1\text{-in} \ \psi_2\text{-in} \)
          unfolding bounded-solution-set-sas-plus'-def
          by blast+
          then show \( ?\)case
            by blast
        next
        case (Suc \( k \))
        moreover have length \( \psi_1 = \text{Suc} \ k \) and length \( \psi_2 = \text{Suc} \ k \)
          using length-Suc-conv \( \text{Suc}(2, \ 3) \)
          unfolding bounded-solution-set-sas-plus'-def
          by blast+
        moreover obtain \( op_1 \ \psi_1' \text{ where } \psi_1 = op_1 \ # \ \psi_1' \)
          and set \( (op_1 \ # \ \psi_1') \subseteq \text{set} \ ?\)ops
          and length \( \psi_1' = k \)
          using calculation(5) \( \text{Suc}(2) \)
          unfolding length-Suc-conv
          by blast
        moreover obtain \( op_2 \ \psi_2' \text{ where } \psi_2 = op_2 \ # \ \psi_2' \)
          and set \( (op_2 \ # \ \psi_2') \subseteq \text{set} \ ?\)ops
          and length \( \psi_2' = k \)
          using calculation(6) \( \text{Suc}(3) \)
          unfolding length-Suc-conv
          by blast
        moreover have set \( \psi_1' \subseteq \text{set} \ ?\)ops and set \( \psi_2' \subseteq \text{set} \ ?\)ops
          using calculation(8, \ 11)
          by auto+  

168
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 $?\varphi_P \psi_1 = (?\varphi_O \psi op_1) \not\# ?\varphi_P \psi_1'$
and $?\varphi_P \psi_2 = (?\varphi_O \psi op_2) \not\# ?\varphi_P \psi_2'$
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 $(\lambda \psi \cdot [?\varphi_O \psi op \ op \leftarrow \psi])$ (bounded-solution-set-sas-plus' $\Psi$ k)
proof
let $?ops = sas-plus-problem.operators-of $ $\Psi$
and $?\varphi_P = \lambda \psi \cdot [\varphi_O \psi op \ op \leftarrow \psi]$
{
  fix $\psi$
  assume $\psi \in $ bounded-solution-set-sas-plus' $\Psi$ k
  then have $set \psi \subseteq set ?ops$
  and $length \psi = k$
  unfolding bounded-solution-set-sas-plus'-def is-serial-solution-for-problem-def
Let-def
  list-all-iff ListMem-iff
  by fast+
  hence $\psi \in $ bounded-plan-set $?ops$ k
  by blast
}
  hence $ bounded-solution-set-sas-plus' $ \Psi$ k $\subseteq $ bounded-plan-set $?ops$ k
  by blast
moreover have inj-on $?\varphi_P$ (bounded-plan-set $?ops$ k)
using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a[OF
ultimately show \( \text{thesis} \)

using \( \text{inj-on-subset}\{\varphi_p \ \text{bounded-plan-set} \ ? \text{ops} \ k \ \text{bounded-solution-set-sas-plus}' \Psi \ k \} \)

by fast

qed

— Show that mapping plan transformation \( \lambda \psi. [\varphi_O \ ? \text{ops} \ op \leftarrow \psi] \) over the solution set for a given SAS+ problem yields the solution set for the induced STRIPS problem.

**private lemma** \( \text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c} \):

assumes \( \text{is-valid-problem-sas-plus} \ ? \psi \)

shows \( (\lambda \psi. [\varphi_O \ ? \text{ops} \ op \leftarrow \psi]) \) ' (\( \text{bounded-solution-set-sas-plus}' \ ? \psi \ k \) ) = \( \text{bounded-solution-set-strips}' (\varphi \ ? \psi) \ k \)

**proof** —

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

and \( \varphi_p = \lambda \psi. [\varphi_O \ ? \text{ops} \ op \leftarrow \psi] \)

let \( \text{Sol}_k = \text{bounded-solution-set-sas-plus}' \ ? \psi \ k \)

and \( \text{Sol}_k' = \text{bounded-solution-set-strips}' \ ? \Pi \ k \)

{ assume \( \varphi_p \ ' \ ? \text{Sol}_k \neq \ ? \text{Sol}_k' \)

then consider \( (A) \exists \pi \in \varphi_p \ ' \ ? \text{Sol}_k. \ ? \pi \notin \ ? \text{Sol}_k' \)

| \( (B) \exists \pi \in \ ? \text{Sol}_k'. \ ? \pi \notin \ ? \varphi_p \ ' \ ? \text{Sol}_k \)

by blast

hence False

**proof** (cases)

case A

moreover obtain \( \pi \) where \( \pi \in \varphi_p \ ' \ ? \text{Sol}_k \) and \( \pi \notin \ ? \text{Sol}_k' \)

using calculation

by blast

moreover obtain \( \psi \) where \( \text{length} \ ? \psi = k \)

and \( \text{SAS-Plus-Semantics.is-serial-solution-for-problem} \ ? \psi \)

and \( \pi = \varphi_p \ ? \psi \)

using calculation(2)

unfolding \( \text{bounded-solution-set-sas-plus}'-\text{def} \)

by blast

moreover have \( \text{length} \ ? \pi = k \) and \( \text{STRIPS-Semantics.is-serial-solution-for-problem} \ ? \Pi \ ? \pi \)

?\Pi \ ? \pi

subgoal

using calculation(4, 6) by auto

subgoal

using \( \text{serial-sas-plus-equivalent-to-serial-strips} \)

assms(1) calculation(5) calculation(6)

by blast

done

moreover have \( \pi \in \ ? \text{Sol}_k' \)

unfolding \( \text{bounded-solution-set-strips}'-\text{def} \)
ultimately show \( ?thesis \)

by fast

next

case \( B \)

moreover obtain \( \pi \) where \( \pi \in \text{Sol}_k' \) and \( \pi \notin ?\varphi_P ' ?\text{Sol}_k \)

using calculation by blast

moreover have \( \text{STRIPS-Semantics.is-serial-solution-for-problem } ?\Pi \pi \)

and \( \text{length } \pi = k \)

using calculation(2)

unfolding \( \text{bounded-solution-set-strips'-def} \)

by simp+

moreover have \( \text{length } [\varphi_O^{-1} ?\Pi \ op \ op \leftarrow \pi] = k \)

and \( \text{SAS-Plus-Semantics.is-serial-solution-for-problem } ?\varphi_O^{-1} ?\Psi \ op \ op \leftarrow \pi] \)

by blast

moreover { \n
have \( \forall \ op \in \text{set } \pi. \ op \in \text{set } ((\exists ?\Pi)_O) \)

using calculation(4)

unfolding \( \text{STRIPS-Semantics.is-serial-solution-for-problem-def list-all-iff} \)

ListMem-iff

by simp

hence \( ?\varphi_P [\varphi_O^{-1} ?\Psi \ op \ op \leftarrow \pi] = \pi \)

proof (induction \( \pi \))

case (\( \text{Cons } op \pi \))

moreover have \( ?\varphi_P [\varphi_O^{-1} ?\Psi \ op \ op \leftarrow \pi \ # \ # \pi] \)

= \( (\varphi_O \Psi (\varphi_O^{-1} ?\Psi \ op)) \ # \ # ?\varphi_P [\varphi_O^{-1} ?\Psi \ op \ op \leftarrow \pi] \)

by simp

moreover have \( \ op \in \text{set } ((\exists ?\Pi)_O) \)

using Cons.prems

by simp

moreover have \( (\varphi_O \Psi (\varphi_O^{-1} ?\Psi \ op)) = \) \( op \)

using strips-operator-inverse-is[\( \text{OF assms(1) calculation(4)}. \)]
moreover have \( \varphi_P [\varphi_O^{-1} \Psi \ op. \ op \leftarrow \pi] = \pi \)
using Cons.IH Cons.prems
by auto
ultimately show \( \text{?case} \)
by argo
qed simp

} moreover have \( \pi \in \varphi _{P} \ \ ? \ \text{Sol}_k \)
using calculation(8, 9)
by force
ultimately show \( \text{?thesis} \)
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'
(\( \varphi \ \Psi \)) \( k \))
proof —
let \( \Xi = \varphi \ \Psi \)
and \( \varphi_P = \lambda \psi. [\varphi_O \ \Psi \ op. \ op \leftarrow \psi] \)
let \( \text{Sol}_k = \text{bounded-solution-set-sas-plus}' \( \Psi \ k \)
and \( \text{Sol}_k' = \text{bounded-solution-set-strips}' \( \Xi \ k \)
have card (\( \varphi _P \ \ ? \ \text{Sol}_k \)) = card (\( \text{Sol}_k \))
using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b[OF assms(1)]
card-image
by blast
moreover have \( \varphi_P \ ? \ ? \text{Sol}_k = \text{Sol}_k' \)
using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c[OF assms(1)].
ultimately show \( \text{?thesis} \)
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 \{ \( \pi. \ \text{set} \pi \subseteq \text{set} \ \text{ops} \ \land \ \text{length} \ \pi = k \) \}
proof (induction \( k \))
case (Suc \( k \))
let \( \text{P} = \{ \pi. \ \text{set} \pi \subseteq \text{set} \ \text{ops} \ \land \ \text{length} \ \pi = k \} \)
and \( \text{P'} = \{ \pi. \ \text{set} \pi \subseteq \text{set} \ \text{ops} \ \land \ \text{length} \ \pi = \text{Suc} \ k \} \)
let \( \text{P}'' = (\bigcup \text{op} \in \text{set} \ \text{ops}. \ (\bigcup \pi \in \text{P}. \ \{ \text{op} \neq \pi \})) \)
\{have \( \forall \ \text{op} \pi. \ \text{finite} \{ \text{op} \neq \pi \} \)

172
by simp
then have \( \forall \text{ op. finite } (\bigcup \pi \in ?P. \{ \text{ op \# } \pi \}) \)
  using finite-UN[of \( ?P \) \ Suc
by blast
hence finite \( ?P'' \)
  using finite-UN[of set ops]
by blast
}
moreover {
{
fix \pi
assume \( \pi \in ?P' \)
moreover have set \( \pi \subseteq \) set ops
  and length \( \pi = \) Suc \( k \)
  using calculation
by simp+
moreover obtain op \( \pi' \) where \( \pi = \) op \# \( \pi' \)
  using calculation \((3)\)
unfolding length-Suc-conv
by fast
moreover have set \( \pi' \subseteq \) set ops and op \( \in \) set ops
  using calculation\((2, 4)\)
by simp+
moreover have length \( \pi' = k \)
  using calculation\((3, 4)\)
by auto
moreover have \( \pi' \in ?P \)
  using calculation\((5, 7)\)
by blast
ultimately have \( \pi \in ?P'' \)
by blast
}
hence \( ?P' \subseteq ?P'' \)
by blast
}
ultimately show \( \forall \text{ case} \)
  using rev-finite-subset[of \( ?P'' \) \( ?P' \)]
by blast
qed force

— 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 \((\text{bounded-solution-set-sas-plus}' \Psi \ k)\)
proof
let \( ?\text{Ops} = \text{set } ((\Psi)_O+) \)
let \( ?\text{Sol}_k = \text{bounded-solution-set-sas-plus}' \Psi \ k \)
and \( ?P_k = \{ \pi. \text{ set } \pi \subseteq ?\text{Ops} \land \text{length } \pi = k \} \)
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 Ψ
  shows finite (bounded-solution-set-strips' (ϕ Ψ) k)
proof —
let Π = ϕ Ψ
let ?Ops = set ((Π)₀)
let ?Sol_k = bounded-solution-set-strips' Π k
  and ?P_k = { π. set π ⊆ ?Ops ∧ length π = k }
{ fix π
  assume π ∈ ?Sol_k
  then have length π = k and set π ⊆ ?Ops
      unfolding bounded-solution-set-strips'-def
      STRIPS-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff List-Mem-iff
    by fastforce+
  hence π ∈ ?P_k
      by blast
 }
then have ?Sol_k ⊆ ?P_k
  by force
thus ?thesis
  by auto
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**

- **Assumptions**: $\text{is-valid-problem-sas-plus } \Psi$
- **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** $\text{finite-R} \\ \text{finite } ?R$

    - **By simp**

    - **Moreover**

      * **Have** $\forall \ k \in ?R. \ \text{finite } (\text{bounded-solution-set-sas-plus'} \ ?\Psi \ k)$

      * **Using** $\text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a}[\text{OF assms}(1)]$,

      * **Moreover have** $\forall \ j \in ?R. \ \forall \ k \in ?R. \ j \neq k$

        * $\rightarrow \ (\text{bounded-solution-set-sas-plus'} \ ?\Psi \ j \ \cap \ \text{bounded-solution-set-sas-plus'} \ ?\Psi \ k = \emptyset)$

      * **Unfolding** $\text{bounded-solution-set-sas-plus'}-\text{def}$

      * **By blast**

  * **Ultimately have** $\text{card } (\text{bounded-solution-set-sas-plus } \Psi \ N) = \ (\sum \ k \in ?R. \ \text{card } (\text{bounded-solution-set-sas-plus'} \ ?\Psi \ k))$

    - **Using** $\text{card-UN-disjoint}$

    * **By blast**

   **Moreover**

   * **Have** $\forall \ k \in ?R. \ \text{finite } (\text{bounded-solution-set-strips'} \ ?\Pi \ k)$

   * **Using** $\text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b}[\text{OF assms}(1)]$,
moreover have \( \forall j \in \mathbb{R} \setminus \{k\} \) \( j \neq k \)

\[ \cap \text{bounded-solution-set-strips'} \Pi j \]

unfolding \( \text{bounded-solution-set-strips'} \text{-def} \)
by blast

ultimately have \( \text{card} \ (\text{bounded-solution-set-strips'} \Pi N) \)
\[ = (\sum_{k \in \mathbb{R}} \text{card} \ (\text{bounded-solution-set-strips'} \Pi k)) \]
using \( \text{card-UN-disjoint} \)
by blast

moreover {
fix \( k \)

have \( \text{card} \ (\text{bounded-solution-set-sas-plus'} \Psi k) \)
\[ = \text{card} \ ((\lambda \psi. [\varphi \circ \psi \circ \text{op} \circ \text{op} \leftarrow \psi]) \ ' \text{bounded-solution-set-sas-plus'} \Psi k) \]
using \( \text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b[OF assms]} \)
\( \text{card-image[symmetric]} \)
by blast

hence \( \text{card} \ (\text{bounded-solution-set-sas-plus'} \Psi k) \)
\[ = \text{card} \ (\text{bounded-solution-set-strips'} \Pi k) \]
using \( \text{sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c[OF assms]} \)
by presburger

ultimately show \( \text{thesis} \)
by presburger
qed

end

end

theory SAT-Plan-Base
imports List-Index.List-Index
   Propositional-Proof-Systems.Formulas
   STRIPS-Semantics
   Map-Supplement List-Supplement
   CNF-Semantics-Supplement CNF-Supplement
begin

— Hide constant and notation for \( (\bot) \) to prevent warnings.
hide-const (open) Orderings.bot-class.bot

— Hide constant and notation for \( ((-\bot)) \) to prevent warnings.
hide-const (open) Transitive-Closure.trancl

176
no-notation Transitive-Closure.trancl ((→) [1000] 999)

— Hide constant and notation for ((→)) to prevent warnings.
hide-const (open) Relation.converse
no-notation Relation.converse ((→⁻¹) [1000] 999)

7 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 $\Phi \equiv \text{encode-to-sat} \ II \ t$ 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.

```
datatype sat-plan-variable =
  State nat nat
  | Operator nat nat
```

A SATPlan formula is a regular propositional formula over SATPlan variables. We add a type synonym to improve readability.

```
type-synonym sat-plan-formula = sat-plan-variable 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 $A \equiv \text{encode-state-variable} \ 0 \ n \ b$ with $n \equiv \text{index} \ vs \ v$ and $b \equiv I$
\( v = \text{Some True} \) 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 \( A \lor \perp \) for which it is easy to show that we can normalize to conjunctive normal form (CNF).

**definition** \( \text{encode-initial-state} \)

\[ \text{\( \text{encode-initial-state} : \text{variable strips-problem} \Rightarrow \text{sat-plan-variable formula} \ (\Phi_I - 99) \)} \]

where \( \text{encode-initial-state} \Pi \)

\[ \equiv \text{let } I = \text{initial-of } \Pi \;
\]
\[ ; \ vs = \text{variables-of } \Pi \]
\[ \text{in } \bigwedge \left( \text{map } (\lambda v. \text{encode-state-variable } 0 \ (\text{index vs } v) \ (I v) \lor \perp) \right) \]
\[ (\text{filter } (\lambda v. I v \neq \text{None}) \ vs) \]

**definition** \( \text{encode-goal-state} \)

\[ \text{\( \text{encode-goal-state} : \text{variable strips-problem} \Rightarrow \text{nat} \Rightarrow \text{sat-plan-variable formula} \ (\Phi_G - 99) \)} \]

where \( \text{encode-goal-state} \Pi t \)

\[ \equiv \text{let } \]
\[ \ vs = \text{variables-of } \Pi \;
\]
\[ ; \ G = \text{goal-of } \Pi \]
\[ \text{in } \bigwedge \left( \text{map } (\lambda v. \text{encode-state-variable } t \ (\text{index vs } v) \ (G v) \lor \perp) \right) \]
\[ (\text{filter } (\lambda v. G v \neq \text{None}) \ vs) \]

Operator preconditions are encoded using activation-implies-precondition formulation as mentioned in ??: i.e. for each operator \( op \in O \) and \( p \in \text{set}\ (\text{precondition-of } op) \) we have to encode

\[ \text{Atom } (\text{Operator } k \ (\text{index ops } op)) \rightarrow \text{Atom } (\text{State } k \ (\text{index vs } v)) \]

We use the equivalent disjunction in the formalization to simplify conversion to CNF.

**definition** \( \text{encode-operator-precondition} \)

\[ \text{\( \text{encode-operator-precondition} : \text{variable strips-problem} \Rightarrow \text{nat} \Rightarrow \text{variable strips-operator} \Rightarrow \text{sat-plan-variable formula} \)} \]

where \( \text{encode-operator-precondition} \Pi t \ op \equiv \text{let } \]
\[ \ vs = \text{variables-of } \Pi \;
\]
\[ ; \ ops = \text{operators-of } \Pi \]
\[ \text{in } \bigwedge \left( \text{map } (\lambda v. \neg \ (\text{Atom } (\text{Operator } t \ (\text{index ops } op))) \lor \text{Atom } (\text{State } t \ (\text{index vs } v)) \ (\text{precondition-of } op)) \right) \]

**definition** \( \text{encode-all-operator-preconditions} \)

\[ \text{\( \text{encode-all-operator-preconditions} : \text{variable strips-problem} \Rightarrow \text{variable strips-operator list} \Rightarrow \text{nat} \Rightarrow \text{sat-plan-variable formula} \)} \]

where \( \text{encode-all-operator-preconditions} \Pi ops t \equiv \text{let } \]
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.

\[ Atom \ (\text{Operator} \ k \ m) \rightarrow \ Atom \ (\text{State} \ (\text{Suc} \ k) \ n) \]

for add effects respectively

\[ Atom \ (\text{Operator} \ k \ m) \rightarrow \neg \ Atom \ (\text{State} \ (\text{Suc} \ k) \ n) \]

for delete effects. We again encode the implications as their equivalent disjunctions in definition ??.

**Definition encode-operator-effect**

:: 'variable strips-problem => nat

\[ \Rightarrow \ '\text{variable strips-operator} \]

\[ \Rightarrow \ sat\text{-plan}\text{-variable formula} \]

where

\[ encode\text{-operator-effect} \ \Pi \ t \ op \]

\[ \equiv \ let \]

\[ vs = \text{variables-of} \ \Pi ; \ ops = \text{operators-of} \ \Pi \]

\[ \text{in} \ \wedge \ (\text{map} \ (\lambda v. \neg (Atom \ (\text{Operator} \ t \ (\text{index} \ ops \ op)))) \]

\[ \lor \ Atom \ (\text{State} \ (\text{Suc} \ t) \ (\text{index} \ vs \ v))) \]

\[ \text{(add-effects-of} \ op) \]

\[ @ \text{map} \ (\lambda v. \neg (Atom \ (\text{Operator} \ t \ (\text{index} \ ops \ op)))) \]

\[ \lor \neg (Atom \ (\text{State} \ (\text{Suc} \ t) \ (\text{index} \ vs \ v))) \]

\[ \text{(delete-effects-of} \ op) \)

**Definition encode-all-operator-effects**

:: 'variable strips-problem => 'variable strips-operator list

\[ \Rightarrow \ nat \]

\[ \Rightarrow \ sat\text{-plan}\text{-variable formula} \]

where

\[ encode\text{-all-operator-effects} \ \Pi \ ops \ t \]

\[ \equiv \ let \ l = \text{List.product} \ [0..<t] \ ops \]

\[ \text{in} \ \text{foldr} \ (\wedge) \ (\text{map} \ (\lambda (t, \ op). \ text{encode-operator-precondition} \ \Pi \ t \ op) \ l) \ (\neg \bot) \]

**Definition encode-operators**

:: 'variable strips-problem => nat => sat\text{-plan}\text{-variable formula}

where

\[ encode\text{-operators} \ \Pi \ t \]

\[ \equiv \ let \ ops = \text{operators-of} \ \Pi \]

\[ \text{in} \ encode\text{-all-operator-preconditions} \ \Pi \ ops \ t \ \wedge \ encode\text{-all-operator-effects} \ \Pi \ ops \ t \]
Definitions ?? and ?? similarly encode the negative resp. positive transition frame axioms as disjunctions.

**definition** encode-negative-transition-frame-axiom
:: 'variable strips-problem
⇒ nat
⇒ 'variable
⇒ sat-plan-variable formula

**where** encode-negative-transition-frame-axiom II t v
≡ let vs = variables-of II
; ops = operators-of II
; deleting-operators = filter (λop. ListMem v (delete-effects-of op)) ops
in ¬(Atom (State t (index vs v)))
∨ (Atom (State (Suc t) (index vs v)))
∨ (∨ (map (λop. Atom (Operator t (index ops op))) deleting-operators))

**definition** encode-positive-transition-frame-axiom
:: 'variable strips-problem
⇒ nat
⇒ 'variable
⇒ sat-plan-variable formula

**where** encode-positive-transition-frame-axiom II t v
≡ let vs = variables-of II
; ops = operators-of II
; adding-operators = filter (λop. ListMem v (add-effects-of op)) ops
in (Atom (State t (index vs v)))
∨ (¬(Atom (State (Suc t) (index vs v))))
∨ (∨ (map (λop. Atom (Operator t (index ops op))) adding-operators)))

**definition** encode-all-frame-axioms
:: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula

**where** encode-all-frame-axioms II t
≡ let l = List.product [0..<t] (variables-of II)
in (∧(map (λ(k, v). encode-negative-transition-frame-axiom II k v) l)
∧ map (λ(k, v). encode-positive-transition-frame-axiom II k v) 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 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 II t
≡ encode-initial-state II
∧ (encode-operators II t
∧ (encode-all-frame-axioms II t
∧ (encode-goal-state II t)))

7Not shown.
Decoding plans from a valuation $\mathcal{A}$ 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 ??). 8

--- 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'**

\[
\begin{align*}
\text{decode-plan'} \colon & \quad \text{variable strips-problem} \\
& \quad \Rightarrow \text{sat-plan-variable valuation} \\
& \quad \Rightarrow \text{nat} \\
& \quad \Rightarrow \text{variable strips-operator list} \\
\text{where} & \quad \text{decode-plan'} \Pi \ A \ i \\
& \quad \equiv \text{let ops} = \text{operators-of} \ \Pi \\
& \quad \quad : \ vs = \text{map} (\lambda \text{op}. \ \text{Operator} \ i (\text{index ops op})) (\text{remdups ops}) \\
& \quad \quad \quad \text{in map} (\lambda \text{v}. \ \text{case v of Operator } - k \Rightarrow \text{ops} \ ! k) (\text{filter } \mathcal{A} \ vs)
\end{align*}
\]

--- 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 $t - (t::'a)$.

**definition decode-plan**

\[
\begin{align*}
\text{decode-plan} \colon & \quad \text{variable strips-problem} \\
& \quad \Rightarrow \text{sat-plan-variable valuation} \\
& \quad \Rightarrow \text{nat} \\
& \quad \Rightarrow \text{variable strips-parallel-plan} (\Phi^{-1} - - 99) \\
\text{where} & \quad \text{decode-plan} \Pi \ A \ t \equiv \text{map} (\text{decode-plan'} \Pi \ A) [0..<t]
\end{align*}
\]

Similarly to the operator decoding, we can decode a state at time $k$ from a valuation of of the SATPlan encoding $\mathcal{A}$ by constructing a map from list of assignments $(v, \mathcal{A} (\text{State} \ k (\text{index} vs v)))$ for all $v \in \mathcal{V}$.

**definition decode-state-at**

\[
\begin{align*}
\text{decode-state-at} \colon & \quad \text{variable strips-problem} \\
& \quad \Rightarrow \text{sat-plan-variable valuation} \\
& \quad \Rightarrow \text{nat} \\
& \quad \Rightarrow \text{variable strips-state} (\Phi_S^{-1} - - 99) \\
\text{where} & \quad \text{decode-state-at} \Pi \ A \ k \\
& \quad \equiv \text{let} \\
& \quad \quad vs = \text{variables-of} \ \Pi \\
& \quad \quad : \ \text{state-encoding-to-assignment} = \lambda v. \ (v, \mathcal{A} (\text{State} \ k (\text{index} vs v))) \\
& \quad \quad \text{in map-of} (\text{map state-encoding-to-assignment} vs)
\end{align*}
\]

8This is handled by function $\text{decode_plan'}$ (not shown).
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 (Φ_T - - 99) where**

\[
\begin{align*}
\text{encode-transitions } & \Pi t \\
& \equiv \text{SAT-Plan-Base.encode-operators } \Pi t \land \\
& \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t
\end{align*}
\]

— Immediately proof the sublocale proposition for strips in order to gain access to definitions and lemmas.

— Setup simp rules.

**lemma [simp]:**

\[
\begin{align*}
\text{encode-transitions } & \Pi t \\
& = \text{SAT-Plan-Base.encode-operators } \Pi t \land \\
& \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t
\end{align*}
\]

**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 \Pi

and \( v \in \text{dom } s \)

shows is-lit-plus (encode-state-variable \( k \) (index (strips-problem.variables-of \Pi) \( v \)) \( s v \))

**proof —**

have \( s v \neq \text{None} \)

using is-valid-problem-strips-initial-of-dom assms(2)

by blast

then consider \((s-of-v-is-some-true) s v = \text{Some True} \)

| \((s-of-v-is-some-false) s v = \text{Some False} \)

by fastforce

thus \(?thesis\)

**unfolding** encode-state-variable-def

by (cases, simp+)

**qed**

**lemma is-cnfs-encode-initial-state:**

assumes is-valid-problem-strips \Pi

shows is-cnfs (Φ_I \Pi)

**proof —**

let \(?I = \Pi_I\)

and \(?vs = \text{strips-problem.variables-of }\Pi\)

let \(?I = \text{map } (λv. \text{encode-state-variable } 0 (\text{index } ?vs \ v) \ (?I \ v) \lor \bot) \)

(filter (λv. \(?I \ v \neq \text{None} \) ?vs) \{ 

182
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) ?es)
   using set-map[of λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥
     filter (λv. ?I v ≠ None) ?es]
   by blast
then have set ?l = (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥) ·
   {v ∈ set ?es. ?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] asms(I)
     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
  using is-cnf-BigAnd[of ?l]
  by (smt is-cnf-BigAnd)
qed

lemma encode-goal-state-is-cnf:
  assumes is-valid-problem-strips Π
  shows is-cnf (encode-goal-state Π t)
proof –
  let ?I = (Π)_I
  and ?G = (Π)_G
and \( ?vs = \text{strips-problem.variables-of II} \)

let \( \mathcal{I} = \text{map} (\lambda v. \text{encode-state-variable t (index} \ ?vs \ v) \ (\?G \ v) \lor \bot) \)

(f\text{ilter} (\lambda v. \?G v \neq \text{None} \ ?vs)

\{ 

fix \( C \)

assume \( C \in \text{set} \ ?I \)

moreover \{ 

have set \( \mathcal{I} = (\lambda v. \text{encode-state-variable t (index} \ ?vs \ v) \ (\?G \ v) \lor \bot) \)

' set (f\text{ilter} (\lambda v. \?G v \neq \text{None} \ ?vs)

unfolding set-map

by blast

then have set \( \mathcal{I} = \{ \text{encode-state-variable t (index} \ ?vs \ v) \ (\?G \ v) \lor \bot \)

| v, v \in \text{set} \ ?us \land \?G v \neq \text{None} \}

by auto

moreover obtain \( v \ where \ C-is: C = \text{encode-state-variable t (index} \ ?vs \ v) \ (\?G \ v) \lor \bot \)

and \( v \in \text{set} \ ?vs \)

and \( G-of-v-is-not-None: \?G v \neq \text{None} \)

using calculation(1)

by auto

moreover \{ 

have \( v \in \text{dom} \ ?G \)

using G-of-v-is-not-None

by blast

moreover have \( \text{is-lit-plus} (\text{encode-state-variable t (index} \ ?vs \ v) \ (\?G \ v)) \)

using assms(1) calculation

by (simp add: \text{encode-state-variable-is-lit-plus-if})

moreover have \( \text{is-lit-plus} \bot \)

by simp

ultimately have \( \text{is-disj} C \)

unfolding C-is

by force

\}

ultimately have \( \text{is-cnf} C \)

by simp

\}

thus \( ?\text{thesis} \)

unfolding \text{encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def}

using \( \text{is-cnf-BigAnd[of} \ ?I] \)

by simp

qed

private lemma \( \text{encode-operator-precondition-is-cnf}: \)

\( \text{is-cnf} (\text{encode-operator-precondition II} k \ ap) \)

proof –

let \( ?vs = \text{strips-problem.variables-of II} \)
and ?ops = strips-problem.operators-of II
let ?l = map (λv. ¬(Atom (Operator k (index ?ops op))) ∨ Atom (State k (index ?vs v)))
(precondition-of op)
{
  have set ?l = (λv. ¬(Atomic (Operator k (index ?ops op))) ∨ Atomic (State k (index ?vs v)))
  ' set (precondition-of op)
  using set-map
  by force
  then have set ?l = { ¬(Atomic (Operator k (index ?ops op))) ∨ Atomic (State k (index ?vs v))
    | v. v ∈ set (precondition-of op) }
  using setcompr-eq-image[of
    λv. ¬(Atomic (Operator k (index ?ops op))) ∨ Atomic (State k (index ?vs v))
    λv. v ∈ set (precondition-of op)]
  by simp
} note set-l-is = this
{
  fix C
  assume C ∈ set ?l
  then obtain v
  where v ∈ set (precondition-of op)
  and C = ¬(Atomic (Operator k (index ?ops op))) ∨ Atomic (State k (index ?vs v))
  using set-l-is
  by blast
  hence is-cnf C
  by simp
}
thus ?thesis
unfolding encode-operator-precondition-def
using is-cnf-BigAnd[of ?l]
by meson
qed

private lemma set-map-operator-precondition[simp]:
set (map (λ(k, op). encode-operator-precondition II k op) (List.product [0..<t] ops))
= { encode-operator-precondition II k op | k op. (k, op) ∈ ([0..<t] × set ops) }
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 \ldots = \{ encode-operator-precondition \Pi k \ op \mid k \ op. (k, \ op) \in \{0 < t\} \times set \ ops\}
using setcompr-eq-image
by fast
finally have set ?fs = \{ encode-operator-precondition \Pi k \ op \mid k \ op. (k, \ op) \in ((0 < t) \times set \ ops) \}
by blast
}
thus ?thesis
by blast
qed

private lemma is-cnf-encode-all-operator-preconditions:
is-cnf (encode-all-operator-preconditions \Pi (strips-problem.operators-of \Pi) t)
proof −
let ?l' = List.product [0 ..< t] (strips-problem.operators-of \Pi)
let ?fs = map (λ(k, \ op). encode-operator-precondition \Pi k \ op) ?l'
have \forall f \in 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 \lor B v) vs) = \{ A v \lor B v \mid v. v \in set vs \}
proof −
let ?l = map (λv. A v \lor B v) vs
have set ?l = (λv. A v \lor B v) \ clop vs
using set-map
by force
thus ?thesis
using setcompr-eq-image
by auto
qed

private lemma encode-operator-effects-is-cnf-i:
is-cnf (\A(map (λv. \neg (Atom (Operator t (index (strips-problem.operators-of \Pi) op))) \lor Atom (State (Suc t) (index (strips-problem.variables-of \Pi) v)))) (add-effects-of op))))
proof −
let ?fs = map (λv. \neg (Atom (Operator t (index (strips-problem.operators-of \Pi) op))))
\lor Atom (State (Suc t) (index (strips-problem.variables-of \Pi) v))) (add-effects-of
\[ \text{fix } C \]
\[ \text{assume } C \in \text{set } ?fs \]
\[ \text{then obtain } v \]
\[ \text{where } v \in \text{set } (\text{add-effects-of } op) \]
\[ \text{and } C = \neg(\text{Atom } (\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi ) \text{ op}))) \]
\[ \lor \text{Atom } (\text{State } (\text{Suc } t) (\text{index } (\text{strips-problem.variables-of } \Pi ) v)) \]
\[ \text{by auto} \]
\[ \text{hence } \text{is-cnf } C \]
\[ \text{by fastforce} \]
\]
thus \text{thesis}
\[ \text{using } \text{is-cnf-BigAnd} \]
\[ \text{by blast} \]
qed

private lemma encode-operator-effects-is-cnf-ii:
\[ \text{is-cnf } (\bigwedge (\text{map } (\lambda v. \neg(\text{Atom } (\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi ) \text{ op})))) \lor \neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index } (\text{strips-problem.variables-of } \Pi ) v)))) (\text{delete-effects-of } op))) \]
proof -
\[ \text{let } ?fs = (\text{map } (\lambda v. \neg(\text{Atom } (\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi ) \text{ op})))) \lor \neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index } (\text{strips-problem.variables-of } \Pi ) v)))) (\text{delete-effects-of } op) \]
\[ \{ \]
\[ \text{fix } C \]
\[ \text{assume } C \in \text{set } ?fs \]
\[ \text{then obtain } v \]
\[ \text{where } v \in \text{set } (\text{delete-effects-of } op) \]
\[ \text{and } C = \neg(\text{Atom } (\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi ) \text{ op}))) \]
\[ \lor \neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index } (\text{strips-problem.variables-of } \Pi ) v))) \]
\[ \text{by auto} \]
\[ \text{hence } \text{is-cnf } C \]
\[ \text{by fastforce} \]
\}
thus \text{thesis}
\[ \text{using } \text{is-cnf-BigAnd} \]
\[ \text{by blast} \]
qed

private lemma encode-operator-effect-is-cnf:
\[ \text{shows } \text{is-cnf } (\text{encode-operator-effect } \Pi \text{ t op}) \]
proof -
\[ \text{let } ?ops = \text{strips-problem.operators-of } \Pi \]
\[ \text{and } ?vs = \text{strips-problem.variables-of } \Pi \]
\[ \text{let } ?fs = (\text{map } (\lambda v. \neg(\text{Atom } (\text{Operator } t (\text{index } ?ops \text{ op})))) \]
\[ 187 \]
∀ Atom (State (Suc t) (index ?es v)))
(add-effects-of op)
and ?fs' = map (λv. ¬(Atom (Operator t (index ?ops op)) (∨ ¬(Atom (State (Suc t) (index ?es v)))))
(delete-effects-of op)
have encode-operator-effect Π t op = (∧(?fs @ ?fs'))

unfolding encode-operator-effect-def[of Π t op]
by metis
moreover {
have ∀f ∈ set ?fs. is-cnf f ∀f ∈ set ?fs'. is-cnf f
using encode-operator-effects-is-cnf[of t Π op]
by (simp+)

hence ∀f ∈ set (?fs @ ?fs'). is-cnf f
by auto
}
ultimately show ?thesis
using is-cnf-BigAnd[of ?fs @ ?fs']
by presburger
qed

private lemma set-map-encode-operator-effect[simp]:
set (map (λ(t, op). encode-operator-effect Π t op) (List.product [0..<t] (strips-problem.operators-of Π)))
= { encode-operator-effect Π k op | k op. (k, op) ∈ (0..<t) × set (strips-problem.operators-of Π) }

proof –
let ?ops = strips-problem.operators-of Π
and ?es = strips-problem.variables-of Π
let ?fs = map (λ(t, op). encode-operator-effect Π t op) (List.product [0..<t] ?ops)
have set ?fs = (λ(t, op). encode-operator-effect Π t op) · (0..<t) × set ?ops
unfolding encode-operator-effect-def[of Π t]
by force
thus ?thesis
using setcompr-eq-image[of λ(t, op). encode-operator-effect Π t op
λ(k, op). (k, op) ∈ (0..<t) × set ?ops]
by force
qed

private lemma encode-all-operator-effects-is-cnf:
assumes is-valid-problem-strips Π
shows is-cnf (encode-all-operator-effects Π (strips-problem.operators-of Π) t)

proof –
let ?ops = strips-problem.operators-of Π
let ?l = List.product [0..<t] ?ops
let ?fs = map (λ(t, op). encode-operator-effect Π t op) ?l
have ∀f ∈ set ?fs. is-cnf f
using encode-operator-effect-is-cnf

188
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-cnff:
assumes is-valid-problem-strips II
shows is-cnff (encode-operators II t)
unfolding encode-operators-def
using is-cnff-encode-all-operator-preconditions[of II t]
encode-all-operator-effects-is-cnff[of II t]
is-cnff.simps(1)of encode-all-operator-preconditions II (strips-problem.operators-of II) t
encode-all-operator-effects (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 ∈ 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 (λop. ListMem v vs) ?ops))
= (λop. Atom (Operator t (index ?ops op))) . { op ∈ set ?ops. v ∈ set vs }
using set-map[of λop. Atom (Operator t (index ?ops op))]
by presburger
thus ?thesis
by blast
qed

lemma is-disj-big-or-if:
assumes ∀ f ∈ set fs. is-lit-plus f
shows is-disj ∨ fs
using assms

proof (induction fs)
  case (Cons f fs)
  have is-lit-plus f
    using Cons.prems
    by simp
  moreover have is-disj ?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 Π t v)
proof –
  let ?vs = strips-problem.variables-of Π
  and ?ops = strips-problem.operators-of Π
  let ?deleting = filter (λop. ListMem v (delete-effects-of op)) ?ops
  let ?fs = map (λop. 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 ∈ set ?fs
      then obtain op
        where op ∈ set ?ops
        and v ∈ set (delete-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-negative-transition-frame-axiom-def
  by meson
lemma is-cnf-encode-positive-transition-frame-axiom:
  shows is-cnf (encode-positive-transition-frame-axiom \( \Pi t v \))

proof –

let \(?vs = \text{strips-problem.variables-of} \Pi\)
and \(?ops = \text{strips-problem.operators-of} \Pi\)
let \(?adding = \text{filter} (\lambda op. \text{ListMem} v (\text{add-effects-of} op)) \?ops\)
let \(?fs = \text{map} (\lambda op. \text{Atom (Operator} t (\text{index} \?ops op))) \?adding\)
and \(?A = \text{Atom (State} t (\text{index} \?vs v))\)
and \(?B = \neg(\text{Atom (State (Suc} t) (\text{index} \?vs v)))\)
\{ 
fix \(f\)
assume \(f \in \text{set} \?fs\)
then obtain \(op\)
  where \(op \in \text{set} \?ops\)
  and \(v \in \text{set} \text{add-effects-of} op\)
  and \(f = \text{Atom (Operator} t (\text{index} \?ops op))\)
using \(\text{set-map-to-operator-atom[of} t \Pi v\)]\)
by fastforce
hence \(\text{is-lit-plus} f\)
by simp
\} 
note \(\text{nb} = \text{this}\)
\{ 
  have \(\text{is-disj} \bigvee \?fs\)
  using \(\text{is-disj-big-or-if} \text{nb}\)
  by blast
then have \(\text{is-disj} (\?B \bigvee \bigvee \?fs)\)
  by force
then have \(\text{is-disj} (\?A \bigvee (\?B \bigvee \bigvee \?fs))\)
  by fastforce
hence \(\text{is-cnf} (\?A \bigvee (\?B \bigvee \bigvee \?fs))\)
  by fastforce
\} 
thus \(?\text{thesis}\)
unfolding \(\text{encode-positive-transition-frame-axiom-def}\)
by meson

qed

private lemma \(\text{encode-all-frame-axioms-set[simp]}:\)
set \((\text{map} \lambda (k, v). \text{encode-negative-transition-frame-axiom} \Pi k v)\)
\((\text{List.product} [0..<t] \text{strips-problem.variables-of} \Pi))\)
@ \((\text{map} \lambda (k, v). \text{encode-positive-transition-frame-axiom} \Pi k v)\)
\((\text{List.product} [0..<t] \text{strips-problem.variables-of} \Pi))\))
= \{ \text{encode-negative-transition-frame-axiom} \Pi k v \mid k v. (k, v) \in ([0..<t] \times \text{set} \text{strips-problem.variables-of} \Pi) \}\)
\cup \{ \text{encode-positive-transition-frame-axiom} \Pi k v \mid k v. (k, v) \in ([0..<t] \times \text{set} \text{strips-problem.variables-of} \Pi) \}\)
proof –
  let ?l = List.product [0..<t] (strips-problem.variables-of II)
  let ?A = (λ(k, v). encode-negative-transition-frame-axiom Π k v) ' set ?l
    and ?B = (λ(k, v). encode-positive-transition-frame-axiom Π k v) ' set ?l
  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 II
  have set-l-is: set ?l = {0..<t} × set ?vs
    by simp
  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
    ∪ (0..<t) × set ?vs]
    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
    ∪ (0..<t) × set ?vs]
    by fast
  ultimately show ?thesis
    by argo
qed

lemma encode-frame-axioms-is-cnfl:
  shows is-cnfl (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) }
  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)
  { fix f
    assume f ∈ set ?fs
    then consider (f-encodes-negative-frame-axiom) f ∈ ?A
    | (f-encodes-positive-frame-axiom) f ∈ ?B
      by fastforce
    hence is-cnfl f
      using is-cnfl-encode-negative-transition-frame-axiom
  }

is-cnf-encode-positive-transition-frame-axiom
by (smt mem-Collect-eq)
}
thus ?thesis
  unfolding encode-all-frame-axioms-def
  using is-cnf-BigAnd[of "fs"
  by meson
qed

lemma is-cnf-encode-problem:
assumes is-valid-problem-strips Π
shows is-cnf (Φ Π t)
proof –
  have is-cnf (Φ Π t)
    using is-cnf-encode-initial-state assms
    by auto
  moreover have is-cnf (encode-goal-state Π t)
    using encode-goal-state-is-cnf[OF assms]
    by simp
  moreover have is-cnf (encode-operators Π t ∧ encode-all-frame-axioms Π t)
    using encode-operators-is-cnf[OF assms] encode-frame-axioms-is-cnf
    unfolding encode-transitions-def
    by simp
  ultimately show ?thesis
    unfolding encode-problem-def SAT-Plan-Base.encode-problem-def
    encode-transitions-def encode-initial-state-def[symmetric] encode-goal-state-def[symmetric]
    by simp
qed

lemma encode-problem-has-model-then-also-partial-encodings:
assumes A |= SAT-Plan-Base.encode-problem Π t
shows A |= SAT-Plan-Base.encode-initial-state Π
  and A |= SAT-Plan-Base.encode-goal-state Π t
  and A |= SAT-Plan-Base.encode-operators Π t
  and A |= SAT-Plan-Base.encode-all-frame-axioms Π t
using assms
unfolding SAT-Plan-Base.encode-problem-def
by simp+

lemma cnf-of-encode-problem-structure:
  shows cnf (SAT-Plan-Base.encode-initial-state Π)
    ⊆ cnf (SAT-Plan-Base.encode-problem Π t)
  and cnf (SAT-Plan-Base.encode-goal-state Π t)
    ⊆ cnf (SAT-Plan-Base.encode-problem Π t)
  and cnf (SAT-Plan-Base.encode-operators Π t)
    ⊆ cnf (SAT-Plan-Base.encode-problem Π t)
  and cnf (SAT-Plan-Base.encode-all-frame-axioms Π t)
    ⊆ cnf (SAT-Plan-Base.encode-problem Π t)
unfolding SAT-Plan-Base.encode-problem-def
SAT-Plan-Base.encode-problem-def[of II t] SAT-Plan-Base.encode-initial-state-def[of II]
SAT-Plan-Base.encode-goal-state-def[of II t] SAT-Plan-Base.encode-operators-def
SAT-Plan-Base.encode-all-frame-axioms-def[of II t]

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 (Φ_I Π) = \bigcup \{ cnf (encode-state-variable 0 (index (strips-problem.variables-of Π) v) ((Π_I) v)) |
v, v ∈ set (strips-problem.variables-of Π) ∧ ((Π_I) v) \neq None \}
proof –
let ?vs = strips-problem.variables-of Π
and ?I = strips-problem.initial-of Π
let ?ls = map (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥) (filter (λv. ?I v) \neq None) ?vs
{ have cnf ’ set ?ls = cnf ’ (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥)
  ’ set (filter (λv. ?I v) \neq None) ?vs)
  using set-map[of λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥]
  by presburger
also have ... = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥))
  ’ set (filter (λv. ?I v) \neq None) ?vs)
  using image-comp
  by blast
also have ... = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v)))
  ’ \{ v ∈ set ?vs. ?I v \neq None \}
  using set-filter[of λv. ?I v \neq None ?vs]
  by auto
finally have cnf ’ set ?ls = \{ cnf (encode-state-variable 0 (index ?vs v) (?I v))
  | v, v ∈ set ?vs ∧ (?I v) \neq None \}
  using setcompr-eq-image[of λv. cnf (encode-state-variable 0 (index ?vs v) (?I v))]
  by presburger
}
moreover have cnf (Φ_I Π) = \bigcup (cnf ’ set ?ls)
  unfolding encode-initial-state-def SAT-Plan-Base.encode-initial-state-def
  using cnf-BigAnd[of ?ls]
  by meson
ultimately show ?thesis
  by auto
qed
A simplification lemma for the above one.

corollary cnf-of-encode-initial-state-set-ii:

assumes is-valid-problem-strips II

shows cnf (Φ I Π) = (∪ v ∈ set (strips-problem.variables-of II). {{{ literal-formula-to-literal (encode-state-variable 0 (index (strips-problem.variables-of II) v) (strips-problem.initial-of II v) ) }}})

proof –

let ?vs = strips-problem.variables-of II
and ?I = strips-problem.initial-of II
have nb₁: { v. v ∈ set ?vs ∧ ?I v ≠ None } = set ?vs
using is-valid-problem-strips-initial-of-dom assms(1)
by auto

{ fix v assume v ∈ set ?vs then have ?I v ≠ None using is-valid-problem-strips-initial-of-dom assms(1)
by auto
then consider (I-v-is-Some-True) ?I v = Some True
| (I-v-is-Some-False) ?I v = Some False
by fastforce
hence cnf (encode-state-variable 0 (index ?vs v) (?I v)) = {{{ literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?I v)) }}}
unfolding encode-state-variable-def
by (cases, simp+)
}

note nb₂ = this

{ have { cnf (encode-state-variable 0 (index ?vs v) (?I v)) | v. v ∈ set ?vs ∧ ?I v ≠ None } = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v))) ` set ?vs
using setcompr-eq-image[λv. cnf (encode-state-variable 0 (index ?vs v) (?I v))]
by presburger
hence { cnf (encode-state-variable 0 (index ?vs v) (?I v)) | v. v ∈ set ?vs ∧ ?I v ≠ None } = (λv. {{{ literal-formula-to-literal (encode-state-variable 0 (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 \( \text{cnf-of-encode-initial-state-set} \):

assumes is-valid-problem-strips II and \( v \in \text{dom} (\text{strips-problem}.\text{initial-of} II) \)

shows \( \text{strips-problem}.\text{initial-of} II v = \text{Some True} \rightarrow (\exists! C. C \in \text{cnf} (\Phi_I II) \wedge C = \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^+ \}) \) and \( \text{strips-problem}.\text{initial-of} II v = \text{Some False} \rightarrow (\exists! C. C \in \text{cnf} (\Phi_I II) \wedge C = \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^{-1} \}) \)

proof

let \( ?I = (II)_I \)

let \( ?\text{vs} = \text{strips-problem}.\text{variables-of} II \)

let \( ?\Phi_I = \Phi_I II \)

have \( \text{nb}_1 : \text{cnf} (\Phi_I II) = \bigcup \{ \text{cnf} (\text{encode-state-variable 0} (\text{index} ?\text{vs} v) (\text{strips-problem}.\text{initial-of} II v)) \mid v. v \in \text{set } ?\text{vs} \wedge ?I v \neq \text{None} \} \)

using \( \text{cnf-of-encode-initial-state-set-i} \)

by blast

{ 
  have \( v \in \text{set } ?\text{vs} \)
  using is-valid-problem-strips-initial-of-dom assms(1, 2)
  by blast
  hence \( v \in \{ v. v \in \text{set } ?\text{vs} \wedge ?I v \neq \text{None} \} \)
  using assms(2)
  by auto
}

note \( \text{nb}_2 = \text{this} \)

show \( \text{strips-problem}.\text{initial-of} II v = \text{Some True} \rightarrow (\exists! C. C \in \text{cnf} (\Phi_I II) \wedge C = \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^+ \}) \) and \( \text{strips-problem}.\text{initial-of} II v = \text{Some False} \rightarrow (\exists! C. C \in \text{cnf} (\Phi_I II) \wedge C = \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^{-1} \}) \)

proof (auto)

assume i-v-is-some-true: \( \text{strips-problem}.\text{initial-of} II v = \text{Some True} \)

then have \( \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^+ \} \in \text{cnf} (\text{encode-state-variable 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v) (?I v)) \)

unfolding encode-state-variable-def

using i-v-is-some-true

by auto

thus \( \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^+ \} \in \text{cnf} (\Phi_I II) \)

using \( \text{nb}_1 \) \( \text{nb}_2 \)

by auto

next

assume i-v-is-some-false: \( \text{strips-problem}.\text{initial-of} II v = \text{Some False} \)

then have \( \{ (\text{State 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v))^{-1} \} \in \text{cnf} (\text{encode-state-variable 0} (\text{index} (\text{strips-problem}.\text{variables-of} II) v) (?I v)) \)

unfolding encode-state-variable-def

using i-v-is-some-false

by auto
\[
\text{thus } \{ (\text{State } 0 \ (\text{index } (\text{strips-problem.variables-of } II ) \ v))^{-1} \} \\
\in \text{cnf } (\Phi I I) \\
\text{using } nb_1 \ nb_2 \\
\text{by auto}
\]

\[\text{qed}\]

\[\text{qed}\]

**lemma** cnf-of-operator-encoding-structure:
\[
\text{cnf } (\text{encode-operators } II \ t) = \text{cnf } (\text{encode-all-operator-preconditions } II \\
(\text{strips-problem.operators-of } II) \ t) \\
\cup \text{cnf } (\text{encode-all-operator-effects } II \ (\text{strips-problem.operators-of } II) \ t)
\]

**unfolding** encode-operators-def

**using** cnf.simps(5)

**by** metis

**corollary** cnf-of-operator-precondition-encoding-subset-encoding:
\[
\text{cnf } (\text{encode-all-operator-preconditions } II \ (\text{strips-problem.operators-of } II) \ t) \\
\subseteq \text{cnf } (\Phi II) \ t
\]

**using** cnf-of-operator-encoding-structure cnf-of-encode-problem-structure subset-trans

**unfolding** encode-problem-def

**by** blast

**lemma** cnf-foldr-and[simp]:
\[
\text{cnf } (\text{foldr } (\land) \ (f \ # \ fs) \ (\neg \bot)) = (\bigcup f \in \text{set } \text{fs}. \text{cnf } f)
\]

**proof** (induction fs)

**case** (Cons \text{fs})

**have** ih: \[
\text{cnf } (\text{foldr } (\land) \ (\text{fs } (\neg \bot)) = (\bigcup f \in \text{set } \text{fs}. \text{cnf } f)
\]

**using** Cons.IH

**by** blast

\{
\begin{align*}
\text{have } & \text{cnf } (\text{foldr } (\land) \ (f \ # \ fs) \ (\neg \bot)) = \text{cnf } (f \land \text{foldr } (\land) \ (\text{fs } (\neg \bot)) \\
& \text{by simp}
\end{align*}
\]

**also have** \ldots = \text{cnf } f \cup \text{cnf } (\text{foldr } (\land) \ (f \ # \ fs) \ (\neg \bot))

**by** force

**finally have** \[
\text{cnf } (\text{foldr } (\land) \ (f \ # \ fs) \ (\neg \bot)) = \text{cnf } f \cup (\bigcup f \in \text{set } \text{fs}. \text{cnf } f)
\]

**using** ih

**by** argo

\}

**thus** ?case

**by** auto

**qed** simp

**private lemma** cnf-of-encode-operator-precondition[simp]:
\[
\text{cnf } (\text{encode-operator-precondition } II \ t \ op) = (\bigcup v \in \text{set } \text{(precondition-of } op). \\
\{ (\text{Operator } t \ (\text{index } (\text{strips-problem.operators-of } II) \ op))^{-1} \\
, (\text{State } t \ (\text{index } (\text{strips-problem.variables-of } II) \ v))^{-1} \} \}
\]

**proof** –
let ?vs = strips-problem.variables-of Π
  and ?ops = strips-problem.operators-of Π
  and ?Ψ_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 (Λ ?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[α 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) ∈ (0..<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 ?Ψ_P = 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)₀)
      by auto
    then have ?A = (λ(t, op). encode-operator-precondition Π t op) ' ((0..<t) × set ((Π)₀))
      using set-map
      by force
  }

198
corollary cnf-of-encode-all-operator-preconditions-contains-clause-if:

fixes II::'variable STRIPS-Representation.strips-problem

assumes is-valid-problem-strips (II::'variable STRIPS-Representation.strips-problem)
  and k < t
  and op ∈ set ((II)ο)
  and v ∈ set (precondition-of op)

shows { (Operator k (index (strips-problem.operators-of II) op))⁻¹
  , (State k (index (strips-problem.variables-of II) v))⁺ } ∈ cnf (encode-all-operator-preconditions II (strips-problem.operators-of II) t)

proof –

let ?ops = strips-problem.operators-of II
  and ?vs = strips-problem.variables-of II

let ?ΦP = encode-all-operator-preconditions II ?ops t
  and ?C = { (Operator k (index (strips-problem.operators-of II) op))⁻¹
  , (State k (index (strips-problem.variables-of II) v))⁺ } } { 
  have nb: (k, op) ∈ {..<t} × set ((II)ο)
    using assms(2, 3)
    by blast

moreover { 
  have ?C ∈ (∪ v∈set (precondition-of op).
    {((Operator k (index (strips-problem.operators-of II) op))⁻¹,
      (State k (index (strips-problem.variables-of II) v))⁺) } )

  using UN-iff[where A= set (precondition-of op)
    and B= λv. {((Operator t (index (strips-problem.operators-of II) op))⁻¹,
      (State t (index (strips-problem.variables-of II) v))⁺) } ] assms(4)
    by blast

  hence ∃ x∈{..<t} × set ((II)ο).
  ?C ∈ (case x of (k, op) ⇒ ∪ v∈set (precondition-of op).
    {((Operator k (index (strips-problem.operators-of II) op))⁻¹,
      (State k (index (strips-problem.variables-of II) v))⁺) })

  using nb
by blast

ultimately have \( ?C \in (\bigcup (t, \text{op}) \in \{\ldots t\} \times \text{set } ((\Pi)_{\Theta})). \)

\( (\bigcup v \in \text{set } \text{(precondition-of op)}) \)

\( \{\{ (\text{Operator } t \text{ (index } ?\text{ops op}))^{-1}, (\text{State } t \text{ (index } ?v \text{ v)})^+ \}\}\) 
by blast

thus \( ?\text{thesis} \)
using cnf-of-encode-all-operator-preconditions-structure[of II t] 
by argo

\textit{corollary} cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem:

\( \text{cnf (encode-all-operator-effects II (strips-problem.operators-of II) t)} \)

\( \subseteq \text{cnf (} \Phi \text{ II t)} \)
using cnf-of-encode-problem-structure(3) cnf-of-operator-encoding-structure

\textit{unfolding} encode-problem-def 
by blast

\textit{private lemma} cnf-of-encode-operator-effect-structure[simp]:

\( \text{cnf (encode-operator-effect II t op)} \)

\( = (\bigcup v \in \text{set } \text{(add-effects-of op)}. \{\{ (\text{Operator } t \text{ (index } \text{strips-problem.operators-of II) op})^{-1}, (\text{State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))^+ \}\}\) 
\( \cup (\bigcup v \in \text{set } \text{(delete-effects-of op)} \)

\( \{\{ (\text{Operator } t \text{ (index } \text{strips-problem.operators-of II) op})^{-1}, (\text{State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))^{-1} \}\}\)

\textit{proof} –

let \( ?fs_1 = \text{map } (\lambda v. \neg (\text{Atom (Operator } t \text{ (index } \text{strips-problem.operators-of II) op})) \)

\( \lor \text{Atom (State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))) \)

\( (\text{add-effects-of op}) \)

\textit{and} \( ?fs_2 = \text{map } (\lambda v. \neg (\text{Atom (Operator } t \text{ (index } \text{strips-problem.operators-of II) op})) \)

\( \lor \neg (\text{Atom (State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))) \)

\( (\text{delete-effects-of op}) \)

\{ 

\textit{have} cnf \( ?fs_1 = \text{cnf} \)

\( (\lambda v. \neg (\text{Atom (Operator } t \text{ (index } \text{strips-problem.operators-of II) op})) \)

\( \lor \text{Atom (State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))) \)

\( (\text{add-effects-of op}) \)

\textit{using} set-map 
by force

\textit{also have} \ldots \( = (\lambda v. \text{cnf } \neg (\text{Atom (Operator } t \text{ (index } \text{strips-problem.operators-of II) op})) \)

\( \lor \text{Atom (State } \text{Suc } t \text{ (index } \text{strips-problem.variables-of II} v))) \)

\( (\text{set (add-effects-of op}) \)

\textit{using} image-comp 
by blast
finally have cnf \( \text{set ?fs} = (\lambda v. \{ (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))^{-1} \\
(\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))^{+} \}) \} \) \( \text{set } \) (add-effects-of op)
by auto
}

\text{note } nb_1 = \text{this }
{
\text{have } cnf \ ' \text{set ?fs} = cnf \ ' (\lambda v. \neg(\text{Atom } (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))))
\\\vee \neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))))
\ ' \text{set } \) (delete-effects-of op)
using set-map
by force
also have \ldots = (\lambda v. cnf (\neg(\text{Atom } (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))))
\\\vee \neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))))
\ ' \text{set } \) (delete-effects-of op)
using image-comp
by blast

finally have cnf \ ' \text{set ?fs} = (\lambda v. \{ (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))^{-1} \\
(\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))^{-1} \}) \} \) \( \text{set } \) (delete-effects-of op)
by auto
}

\text{note } nb_2 = \text{this }
{
\text{have } cnf (\text{encode-operator-effect } \Pi t \text{ op}) = \bigcup (cnf \ ' \text{set } \) (?fs_1 @ ?fs_2))
unfolding encode-operator-effect-def
using cnf-BigAnd[of ?fs_1 @ ?fs_2]
by meson
also have \ldots = \bigcup (cnf \ ' \text{set } ?fs_1 \cup cnf \ ' \text{set } ?fs_2)
using set-append[of ?fs_1 @ ?fs_2] image-Un[of cnf set ?fs_1 set ?fs_2]
by argo
also have \ldots = \bigcup (cnf \ ' \text{set } ?fs_1) \cup \bigcup (cnf \ ' \text{set } ?fs_2)
using Union-Un-distrib[of cnf \ ' \text{set } ?fs_1 cnf \ ' \text{set } ?fs_2]
by argo

finally have cnf (encode-operator-effect \Pi t \text{ op})
= (\bigcup v \in \text{set } (\text{add-effects-of op}.
\{ (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))^{-1} \\
(\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))^{+} \}) \} \\
\cup (\bigcup v \in \text{set } (\text{delete-effects-of op}.
\{ (\text{Operator } t (\text{index }(\text{strips-problem.operators-of } \Pi) \text{ op}))^{-1} \\
(\text{State } (\text{Suc } t) (\text{index }(\text{strips-problem.variables-of } \Pi) v))^{-1} \}) \} \\
using nb_1 nb_2
by argo
}
thus \(\text{thesis}\)

by \textit{blast}

\textbf{Qed}

\textbf{Lemma} cnf-of-encode-all-operator-effects-structure:

\(\text{cnf}\ (\text{encode-all-operator-effects } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t)\)

\(= (\bigcup \{ (k, \text{op}) \in \{(0..<t) \times \text{set } ((\Pi)_{\mathcal{O}})\}).\)

\((\bigcup v \in \text{set } (\text{add-effects-of } \text{op})\).

\((\bigcup \{ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ \text{op}))^{-1}
\quad (\text{State } (\text{Suc } k) \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+\})).\)

\(\bigcup (\bigcup (k, \text{op}) \in \{(0..<t) \times \text{set } ((\Pi)_{\mathcal{O}})\}).\)

\(\bigcup v \in \text{set } (\text{delete-effects-of } \text{op})\).

\((\bigcup \{ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ \text{op}))^{-1}
\quad (\text{State } (\text{Suc } k) \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^{-1}\})).\)

\textbf{Proof} –

let \(\text{ops} = \text{strips-problem.operators-of } \Pi\)

and \(\text{vs} = \text{strips-problem.variables-of } \Pi\)

let \(\text{cfE} = \text{encode-all-operator-effects } \Pi \ \text{ops} \ t\)

and \(\text{l} = \text{List.product } [0..<t] \ ?\text{ops}\)

let \(\text{fs} = \text{map } (\lambda (t, \text{op}). \text{encode-operator-effect } \Pi \ t \ \text{op}) \ ?l\)

have \text{nb}: \text{set } (\text{List.product } [0..<t] \ ?\text{ops}) = \{0..<t\} \times \text{set } \?\text{ops}\)

by \textit{simp}

\{\n
\text{have } \text{cnf } \times \text{set } \text{fs} = \text{cnf } \times (\lambda (k, \text{op}). \text{encode-operator-effect } \Pi \ k \ \text{op}) \times \{0..<t\} \times \text{set } \text{ops}\)

by \textit{force}

\text{also have } \ldots = (\lambda (k, \text{op}). \text{cnf } \text{encode-operator-effect } \Pi \ k \ \text{op}) \times \{0..<t\} \times \text{set } \text{ops}\)

using \textit{image-comp}

by \textit{fast}

\textbf{Finally have} \text{cnf } \times \text{set } \text{fs} = (\lambda (k, \text{op}).

\((\bigcup v \in \text{set } (\text{add-effects-of } \text{op})\).

\((\bigcup \{ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ \text{op}))^{-1}
\quad (\text{State } (\text{Suc } k) \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+\})).\)

\(\bigcup (\bigcup v \in \text{set } (\text{delete-effects-of } \text{op})\).

\((\bigcup \{ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ \text{op}))^{-1}
\quad (\text{State } (\text{Suc } k) \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^{-1}\})).\)

\times \{0..<t\} \times \text{set } \text{ops}\)

using \textit{cnf-of-encode-operator-effect-structure}

by \textit{auto}

\}

\textbf{thus }\text{thesis}

\textbf{Unfolding} \textit{encode-all-operator-effects-def}

\textbf{Using} \textit{cnf-BigAnd[of } \text{fs]}

by \textit{auto}

\textbf{Qed}
corollary cnf-of-operator-effect-encoding-contains-add-effect-clause-if:
fixes Π:: ′a strips-problem
assumes is-valid-problem-strips Π
and k < t
and op ∈ set ((Π)\(O_o\))
and v ∈ set (add-effects-of op)
shows \{ (Operator k (index (strips-problem.operators-of Π) op))\(^{-1}\)
, (State (Suc k) (index (strips-problem.variables-of Π) v))\(^+\) \}\
∈ cnf (encode-all-operator-effects Π (strips-problem.operators-of Π) t)

proof –
let ?Φ_E = encode-all-operator-effects Π (strips-problem.operators-of Π) t
and ?ops = strips-problem.operators-of Π
and ?vs = strips-problem.variables-of Π
let ?Add = \(\bigcup (k, op) \in \{0..<t\} \times set ((Π)\(O_o\)). \\bigcup v\in set \(add-effects-of op\). \{\{ (Operator k (index ?ops op))\(^{-1}\), (State (Suc k) (index ?vs v))\(^+\) \}\}\)
let ?C = \{ (Operator k (index ?ops op))\(^{-1}\), (State (Suc k) (index ?vs v))\(^+\) \}\ have ?Add ⊆ cnf ?Φ_E
by presburger
moreover {
have ?C ∈ \{ (Operator k (index ?ops op))\(^{-1}\), (State (Suc k) (index ?vs v))\(^+\) \}\ using assms(4)
by blast
then have ?C ∈ \(\bigcup v\in set \(add-effects-of op\). \{\{ (Operator k (index ?ops op))\(^{-1}\), (State (Suc k) (index ?vs v))\(^+\) \}\}\)
using Complete-Lattices.UN-iff \(\lambda v. \{\{ (Operator k (index ?ops op))\(^{-1}\), (State (Suc k) (index ?vs v))\(^+\) \}\ set \(add-effects-of op\))\ using assms(4)
by blast
moreover have (k, op) ∈ \(\{0..<t\} \times set ((Π)\(O_o\))\)
using assms(2, 3)
by fastforce
ultimately have ?C ∈ ?Add
by blast
}
ultimately show ?thesis
using subset-eq[of ?Add cnf ?Φ_E]
by meson
qed

corollary cnf-of-operator-effect-encoding-contains-delete-effect-clause-if:
fixes Π:: ′a strips-problem
assumes is-valid-problem-strips Π
and k < t
and op ∈ set ((Π)\(O_o\))
and v ∈ set (delete-effects-of op)
shows \{ (\text{Operator } k \text{ (index } \text{strips-problem.operators-of } \Pi \text{ op}))^{-1} \\
, (\text{State } (\text{Suc } k) \text{ (index } \text{strips-problem.variables-of } \Pi \text{ v}))^{-1} \} \\
\in \text{cnf } (\text{encode-all-operator-effects } \Pi \text{ (strips-problem.operators-of } \Pi \text{ ) } t)\\n\}

proof —

let \(\Phi_E = \text{encode-all-operator-effects } \Pi \text{ (strips-problem.operators-of } \Pi \text{ ) } t\)
and \(\?ops = \text{strips-problem.operators-of } \Pi\)
and \(\?vs = \text{strips-problem.variables-of } \Pi\)

let \(\?Delete = (\bigcup \{ (k, op) \in \{0..< t\} \times \text{set } ((\Pi)_{O}).\}
\bigcup \text{v} \in \text{set } (\text{delete-effects-of } op)\).
\{ \{ (\text{Operator } k \text{ (index } \?ops op))^{-1}, (\text{State } (\text{Suc } k) \text{ (index } \?vs v))^{-1} \} \}\)

let \(\?C = \{ (\text{Operator } k \text{ (index } \?ops op))^{-1}, (\text{State } (\text{Suc } k) \text{ (index } \?vs v))^{-1} \}\)

have \(\?Delete \subseteq \text{cnf } \Phi_E\)
using \text{cnf-of-encode-all-operator-effects-structure}[of \Pi \ t] \text{ Un-upper2}[of \?Delete]
by presburger

moreover {\\nhave \(\?C \in (\bigcup \text{v} \in \text{set } (\text{delete-effects-of } op).\)
\{ \{ (\text{Operator } k \text{ (index } \?ops op))^{-1}, (\text{State } (\text{Suc } k) \text{ (index } \?vs v))^{-1} \} \}\)
using assms(4)
by blast

moreover have \((k, op) \in \{0..< t\} \times \text{set } \?ops\)
using assms(2, 3)
by force

ultimately have \(\?C \in \?Delete\)
by fastforce

}

ultimately show \(\?thesis\)
using \text{subset-eq}[of \?Delete \text{cnf } \?\Phi_E] \text{ by meson}

qed

private lemma \text{cnf-of-big-or-of-literal-formulas-is}[simp]:
assumes \(\forall f \in \text{set } fs . \text{is-literal-formula } f\)
shows \(\text{cnf } \{ \text{literal-formula-to-literal } f \mid f . f \in \text{set } fs \}\)
using assms

proof (induction fs)

case \(\text{Cons } fs\)

\{\\nhave \text{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 \text{cnf-of-literal-formula}
by blast
\}

note \(nb_1 = \text{this}\)

\{\\nhave \(\forall f' \in \text{set } fs . \text{is-literal-formula } f'\)
using Cons.prems
by fastforce

hence cnf (∨fs) = \{ \{ literal-formula-to-literal f | f ∈ set fs \} \}
using Cons.IH
by argo

} note nb2 = this

{ have cnf (∨(f ≠ fs)) = (λ(g, h). g ∪ h)
  \times \{ \{ literal-formula-to-literal f' | f'. f' ∈ set fs \} \}
using nb1 nb2
by simp
also have ... = \{ \{ literal-formula-to-literal f \}
  \∪ \{ literal-formula-to-literal f' | f'. f' ∈ set fs \} \}
by fast
finally have cnf (∨(f ≠ fs)) = \{ \{ literal-formula-to-literal f' | f'. f' ∈ set (f ≠ fs) \} \}
by fastforce
}
thus ?case .

qed simp

private lemma set-filter-op-list-mem-vs\[simp\]:
  set (filter (λop. ListMem v vs) ops) = \{ op. op ∈ set ops ∧ v ∈ set vs \}
using set-filter[of λop. ListMem v vs ops] ListMem-iff
by force

private lemma cnf-of-positive-transition-frame-axiom:
cnf (encode-positive-transition-frame-axiom Π k v)
   = \{ \{ State k (index (strips-problem.variables-of Π) v) \}^+
       \∪ \{ (State (Suc k) (index (strips-problem.variables-of Π) v))^-1 \} \}
       \∪ \{ (Operator k (index (strips-problem.operators-of Π) op))^+
       \∪ \{ op. op ∈ set (strips-problem.operators-of Π) ∧ v ∈ set (add-effects-of op) \}
\}

proof –
let ?vs = strips-problem.variables-of Π
and ?ops = strips-problem.operators-of Π
let ?adding-operators = filter (λop. ListMem v (add-effects-of op)) ?ops
let ?fs = map (λop. Atom (Operator k (index ?ops op))) ?adding-operators
{ have set ?fs = (λop. Atom (Operator k (index ?ops op))) \ ' set ?adding-operators
using set-map[of λop. Atom (Operator k (index ?ops op)) ?adding-operators]
by blast

then have literal-formula-to-literal \ ' set ?fs
   = (λop. (Operator k (index ?ops op))^+) \ ' set ?adding-operators
using image-comp[of literal-formula-to-literal λop. Atom (Operator k (index
?ops op))]
set ?adding-operators]

205
by simp
also have \ldots = (\lambda op. (Operator k (index \ops \op))^+ \cdot \{ op. \op \in \set \ops \land v \in \set (add-effects-of op) \})
using set-filter-op-list-mem-vs[of v - \ops]
by auto

finally have literal-formula-to-literal \cdot set \fs
= \{(Operator k (index \ops \op))^+ \mid op. \op \in \set \ops \land v \in \set (add-effects-of op) \}
using setcompr-eq-image[of \lambda op. (Operator k (index \ops \op))^+
\lambda op. \op \in \set \adding-operators]
by blast

hence cnf (\bigvee \fs) = \{(Operator k (index \ops \op))^+ \mid op. \op \in \set \ops \land v \in \set (add-effects-of op) \}
using cnf-of-big-or-of-literal-formulas-is[of \fs]
setcompr-eq-image[of literal-formula-to-literal \lambda f. f \in \set \fs]
by force

then have cnf \((\neg(Atom (State (Suc k) (index \vs v))) \lor \bigvee \fs)\)
= \{(State (Suc k) (index \vs v))^{-1}\} \cup \{(Operator k (index \ops \op))^+ \mid op. \op \in \set \ops \land v \in \set (add-effects-of op) \}
by force

then have cnf \((\neg(Atom (State k (index \vs v))) \lor \neg(Atom (State (Suc k) (index \vs v))) \lor \bigvee \fs)\)
= \{(State k (index \vs v))^{+}\}
\cup \{(State (Suc k) (index \vs v))^{-1}\}
\cup \{(Operator k (index \ops \op))^+ \mid op. \op \in \set \ops \land v \in \set (add-effects-of op) \}
by simp

moreover have cnf (encode-positive-transition-frame-axiom \Pi k v)
= cnf \((\neg(Atom (State k (index \vs v))) \lor \neg(Atom (State (Suc k) (index \vs v))) \lor \bigvee \fs)\)
\lor unfolding encode-positive-transition-frame-axiom-def
by metis

ultimately show \?thesis
by blast
qed

private lemma cnf-of-negative-transition-frame-axiom:
\begin{align*}
\text{cnf (encode-negative-transition-frame-axiom \Pi k v)}
&= \{(State k (index \vstraps-problem.variables-of \Pi) v)^{-1} \\
&\cup (State (Suc k) (index \vstraps-problem.variables-of \Pi) v)^{+}\}
\cup \{(Operator k (index \vstraps-problem.operators-of \Pi) op))^+ \mid op. \op \in \set (\vstraps-problem.operators-of \Pi) \land v \in \set (delete-effects-of op)\}
\end{align*}
proof

let ?vs = strips-problem:variables-of Π
and ?ops = strips-problem:operators-of Π

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[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))') ' set ?deleting-operators
using image-comp[λop. Atom (Operator k (index ?ops op))] ?deleting-operators

by simp

also have . . . = (λop. (Operator k (index ?ops op))') ' \{ 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)) | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) \}
using setcompr-eq-image[λop. (Operator k (index ?ops op))']
λop. op ∈ set ?deleting-operators
by blast

hence cnf (\bigvee ?fs) = \{ (Operator k (index ?ops op))' | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) \}
using cnf-of-big-or-of-literal-formulas-is[of ?fs]
setcompr-eq-image[λf. f ∈ set ?fs]
by force

then have cnf (Atom (State (Suc k) (index ?vs v)) ∨ \bigvee ?fs)
= \{ (State (Suc k) (index ?vs v))' \} ∪ \{ (Operator k (index ?ops op))' | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) \}
by force

then have cnf ((¬ (Atom (State k (index ?vs v)))) ∨ (Atom (State (Suc k) (index ?vs v)) ∨ \bigvee ?fs))
= \{ (State k (index ?vs v))^-1 \}
∪ \{ (State (Suc k) (index ?vs v))' \}
∪ \{ (Operator k (index ?ops op))' | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) \}
by simp
moreover have cnf (encode-negative-transition-frame-axiom \( \Pi k v \))
\[ = \text{cnf}\left((\neg \text{Atom}(\text{State k (index \( ?v \))}) \lor \text{Atom}(\text{State (Suc k) (index \( ?v \}))\lor \bigvee \text{?fs})\right) \]

unfolding encode-negative-transition-frame-axiom-def
by metis

ultimately show ?thesis
by blast
qed

lemma cnf-of-encode-all-frame-axioms-structure:
\[ \text{cnf}\left(\text{encode-all-frame-axioms \( \Pi t \)}\right) = \bigcup (k, v) \in (\{0..<t\} \times \text{set}(\Pi_v)).
\{ (\text{State k (index (strips-problem.variables-of \( \Pi \)) v)})^+, 
(\text{State (Suc k) (index (strips-problem.variables-of \( \Pi \)) v)})^{-1}
\} \]
\[ \cup \{(\text{Operator k (index (strips-problem.operators-of \( \Pi \)) op)})^+
| \text{op. op \( \in \) set}(\Pi_O) \land v \in \text{set}\left(\text{add-effects-of op}\right)\}\}\]
\[ \cup \{(\text{State k (index (strips-problem.variables-of \( \Pi \)) v)})^{-1}
(\text{State (Suc k) (index (strips-problem.variables-of \( \Pi \)) v)})^+
\} \]
\[ \cup \{(\text{Operator k (index (strips-problem.operators-of \( \Pi \)) op)})^+
| \text{op. op \( \in \) set}(\Pi_O) \land v \in \text{set}\left(\text{delete-effects-of op}\right)\}\}\]

proof –
let \( ?v = \text{strips-problem.variables-of \( \Pi \)}\)
and \( ?ops = \text{strips-problem.operators-of \( \Pi \)}\)
and \( ?\Phi_F = \text{encode-all-frame-axioms \( \Pi t \)}\)
let \( l = \text{List.product (0..<t) \( ?v \)}\)
let \( fs = \text{map}(\lambda (k, v). \text{encode-negative-transition-frame-axiom \( \Pi k v \)} \) \( @ l \)
\[ \circ \text{map}(\lambda (k, v). \text{encode-positive-transition-frame-axiom \( \Pi k v \)} \) \( @ l \)
\[
\{ \\
\begin{array}{l}
\text{let } A = \{ \text{encode-negative-transition-frame-axiom \( \Pi k v \)} \\
| k. v. (k, v) \in (\{0..<t\} \times \text{set}(\Pi_v))\} \\
\text{and } B = \{ \text{encode-positive-transition-frame-axiom \( \Pi k v \)} \\
| k. v. (k, v) \in (\{0..<t\} \times \text{set}(\Pi_v))\} \\
\text{have set-l: set } l = \{0..<t\} \times \text{set}(\Pi_v) \\
\text{using set-product} \\
\text{by force} \\
\end{array}
\]

have set \( ?fs = A \cup B \)
unfolding set-append set-map
using encode-all-frame-axioms-set
by force
then have \( \text{cnf}\cdot \text{set } ?fs = \text{cnf } A \cup \text{cnf } B \)
using image-Union of cnf \( ?A \cup ?B \)
by argo
moreover \{ \\
\begin{array}{l}
\text{have } A = (\bigcup (k, v) \in (\{0..<t\} \times \text{set}(\Pi_v)). \\
\{ \text{encode-negative-transition-frame-axiom \( \Pi k v \)} \} \\
\text{by blast} \\
\end{array}
\]

208
then have \( \text{cnf} \ ?A = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) \). \\
\{ \text{cnf} (\text{encode-negative-transition-frame-axiom} \Pi k v) \} \) \\
by blast \\
hence \( \text{cnf} \ ?A = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{-1} \\
\quad \text{(State } \text{Suc } k (\text{index } ?vs v)\}^{-1} \} \\
\cup \{(\text{Operator } k (\text{index } ?ops op))^{+} \\
\quad \text{op. op } \in \text{set } ?ops \wedge v \in \text{set } (\text{delete-effects-of } op)\})) \\
using \text{cnf-of-negative-transition-frame-axiom[of } \Pi \} \\
by presburger \\
\}

moreover \{ \\
\text{have } ?B = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{ \text{ encode-positive-transition-frame-axiom } \Pi k v \} \) \\
by blast \\
then have \( \text{cnf} \ ?B = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{ \text{cnf} (\text{encode-positive-transition-frame-axiom } \Pi k v) \} \) \\
by blast \\
hence \( \text{cnf} \ ?B = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{+} \\
\quad \text{(State } \text{Suc } k (\text{index } ?vs v)\}^{-1} \} \\
\cup \{(\text{Operator } k (\text{index } ?ops op))^{+} \\
\quad \text{op. op } \in \text{set } ?ops \wedge v \in \text{set } (\text{add-effects-of } op)\})))) \\
using \text{cnf-of-positive-transition-frame-axiom[of } \Pi \} \\
by presburger \\
\}

ultimately have \( \text{cnf} \ set ?fs = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{+} \\
\quad \text{(State } \text{Suc } k (\text{index } ?vs v)\}^{-1} \} \\
\cup \{(\text{Operator } k (\text{index } ?ops op))^{+} \\
\quad \text{op. op } \in \text{set } ((\Pi)\_o) \wedge v \in \text{set } (\text{add-effects-of } op)\})\} \\
\cup (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{-1} \\
\quad \text{(State } \text{Suc } k (\text{index } ?vs v)\}^{+} \} \\
\cup \{(\text{Operator } k (\text{index } ?ops op))^{+} \\
\quad \text{op. op } \in \text{set } ((\Pi)\_o) \wedge v \in \text{set } (\text{delete-effects-of } op)\})\} \\
\} \\
\} \\
\text{ unfolding set-append set-map} \\
by force \\
\}

then have \( \text{cnf} (\text{encode-all-frame-axioms } \Pi t) = (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{+} \\
\quad \text{(State } \text{Suc } k (\text{index } ?vs v)\}^{-1} \} \\
\cup \{(\text{Operator } k (\text{index } ?ops op))^{+} \\
\quad \text{op. op } \in \text{set } ((\Pi)\_o) \wedge v \in \text{set } (\text{add-effects-of } op)\})\} \\
\cup (\bigcup (k, v) \in \{(0..<t) \times set ((II)\_v)) . \\
\{\{ \text{(State } k (\text{index } ?vs v)\}^{-1} \\
\} \} \\
\}

209
(State (Suc k) (index ?vs v))^+ \cup \{(Operator k (index ?ops op))^+ \\
| op. op \in set ((\Pi)_O) \land v \in set (\text{delete-effects-of op})\}\}

unfolding encode-all-frame-axioms-def Let-def
using cnf-BigAnd[of ?fs]
by argo
thus \textbf{thesis}
using Union-Un-distrib[of \\
(\bigcup (k, v) \in (\{0..<t\} \times set ((\Pi)_V)) \\cup \{(State (Suc k) (index ?vs v))^-1 \} \\
\cup \{(Operator k (index ?ops op))^+ \\
| op. op \in set ((\Pi)_O) \land v \in set (\text{add-effects-of op})\}\}
(\bigcup (k, v) \in (\{0..<t\} \times set ((\Pi)_V)).
\{(State (k (index ?vs v))^-1 \\
\cup \{(Operator k (index ?ops op))^+ \\
| op. op \in set ((\Pi)_O) \land v \in set (\text{delete-effects-of op})\}\}\)

by argo
qed

— A technical lemma used in.

private lemma cnf-of-encode-goal-state-set-i:
\begin{align*}
\text{cnf } (\Phi_G (\Pi)_t) &= \bigcup \{\text{cnf} (\text{encode-state-variable } t) \\\n&\quad (\text{index } \text{strips-problem.variables-of } \Pi) v \land (\Pi)_G v \neq \text{None} \} \\
\text{proof} &\quad \text{let } ?vs = \text{strips-problem.variables-of } \Pi \\
&\quad \text{and } ?G = (\Pi)_G \\
&\quad \text{and } ?G_G = (\Phi_G (\Pi)_t) \\
&\quad \text{let } ?fs = \text{map } (\lambda v. \text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v} \lor \bot) \\
&\quad (\text{filter } (\lambda v. ?G v \neq \text{None}) \ ?us) \\
&\quad \{ \text{have } \text{cnf } \text{set } ?fs = \text{cnf } (\lambda v. \text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v} \lor \bot) \}} \\
&\quad \quad \quad \text{'} \{ v \mid v. v \in \text{set } ?vs \land ?G v \neq \text{None} \} \\
&\quad \quad \quad \text{unfolding set-map} \\
&\quad \quad \quad \text{by force} \\
&\quad \quad \quad \text{also have } \ldots = (\lambda v. \text{cnf} (\text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v} \lor \bot)) \\
&\quad \quad \quad \text{'} \{ v \mid v. v \in \text{set } ?vs \land ?G v \neq \text{None} \} \\
&\quad \quad \quad \text{using image-comp[of cnf } (\lambda v. \text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v} \lor \bot) \} \\
&\quad \quad \quad \text{'} \{ v \mid v. v \in \text{set } ?vs \land ?G v \neq \text{None} \} \\
&\quad \quad \quad \text{by fast} \\
&\quad \quad \text{finally have } \text{cnf } \text{set } ?fs = \{ \text{cnf} (\text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v})) \\
&\quad \quad \text{'} \{ v \mid v. v \in \text{set } ?vs \land ?G v \neq \text{None} \} \\
&\quad \quad \text{unfolding setcompr-eq-image[of } \lambda v. \text{cnf } (\text{encode-state-variable } t (\text{index } ?vs v) (\text{?G v}) \lor \bot) \} \\
&\quad \quad \text{by auto} 
\end{align*}
moreover have \( \text{cnf} \ (\Phi G \Pi) t) = \bigcup (\text{cnf} \ \text{set} \ \varsigma) \)

unfolding \text{encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def}

by force

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

by \text{simp}

qed

— A simplification lemma for the above one.

corollary \text{cnf-of-encode-goal-state-set-ii}:

assumes \text{is-valid-problem-strips} \Pi

shows \( \text{cnf} \ ((\Phi G \Pi) t) = \bigcup (\text{\{ literal-formula-to-literal}

\ (\text{encode-state-variable} \ t \ (\text{index} \ (\text{strips-problem.variables-of} \ \Pi) \ v) \ (((\Pi) G) v))) \)

\} \ | \ v. \ v \in \text{set} \ ((\Pi) V) \land ((\Pi) G) v \neq \text{None} \)

proof –

let \( ?\text{vs} = \text{strips-problem.variables-of} \ \Pi \)

and \( ?G = ((\Pi) G) \)

and \( ?\Phi G = (\Phi G \Pi) t \)

{ \n
  \fix \ v

  assume \ v \in \{ \ v \ | \ v. \ v \in \text{set} \ ((\Pi) V) \land ?G v \neq \text{None} \} \)

  then have \ v \in \text{set} \ ((\Pi) V) \land \text{G-of-v-is-not-None}: \ ?G v \neq \text{None} \n
  by \text{fast+}

  then consider \ (A) \ ?G v = \text{Some True}

  \ | \ (B) ?G v = \text{Some False}

  by \text{fastforce}

  hence \( \text{cnf} \ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (\text{?G v})) \)

  = \{ \{ \ \text{literal-formula-to-literal} \ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (\text{?G v})) \}

}\)

unfolding \text{encode-state-variable-def}

by \text{cases, force+}

} note nb = this

have \( \varphi G = \bigcup (\text{\{ cnf} \ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (\text{?G v})) \)

\ | \ v. \ v \in \text{set} \ ((\Pi) V) \land ?G v \neq \text{None} \}

unfolding \text{cnf-of-encode-goal-state-set-ii}

by \text{blast}

also have \ldots = \bigcup ((\lambda v. \text{cnf} \ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (((\Pi) G) v)))

\ | \ v. \ v \in \text{set} \ ((\Pi) V) \land ((\Pi) G) v \neq \text{None} \}

using \text{setcompr-eq-image[of}

\lambda v. \text{cnf} \ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (((\Pi) G) v))

\lambda v. \ v \in \text{set} \ ((\Pi) V) \land ((\Pi) G) v \neq \text{None}

by \text{presburger}

also have \ldots = \bigcup ((\lambda v. \{ \text{literal-formula-to-literal}

\ (\text{encode-state-variable} \ t \ (\text{index} \ ?\text{vs} v) \ (\text{?G v}) \}) \}

\ | \ v. \ v \in \text{set} \ ((\Pi) V) \land ((\Pi) G) v \neq \text{None} \}

using \text{nb}
by simp
finally show ?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 Π :: 'a strips-problem
  assumes is-valid-problem-strips Π
  and v ∈ dom ((Π) G)
  shows ((Π) G) v = Some True → (∃!C. C ∈ cnf ((Φ G Π) t)
    ∧ C = { (State t (index (strips-problem.variables-of Π) v))'' })
  and ((Π) G) v = Some False → (∃!C. C ∈ cnf ((Φ G Π) t)
    ∧ C = { (State t (index (strips-problem.variables-of Π) v))' })
proof
  let ?vs = strips-problem.variables-of Π
  and ?G = (Π) G
  and ?Φ G = (Φ G Π) t
  have nb1: cnf ?Φ G = ∪ { cnf (encode-state-variable t (index ?vs v)
    (?G v)) | v. v ∈ set ((Π)V) ∧ ?G v ≠ None }
    unfolding cnf-of-encode-goal-state-set-i
    by auto
  have nb2: v ∈ { v. v ∈ set ((Π)V) ∧ ?G v ≠ None }
    using is-valid-problem-dom-of-goal-state-is assms(1, 2)
    by auto
  have nb3: cnf (encode-state-variable t (index (strips-problem.variables-of Π) v)
    ((Π) G) v)) ≤ ∪ { cnf (encode-state-variable t (index ?vs v)
      (?G v)) | v. v ∈ set ((Π)V) ∧ ?G v ≠ None }
    using UN-upper[OF nb2, of λv. cnf (encode-state-variable t (index ?vs v) (?G v))]
    nb2
    by blast
  show ((Π) G) v = Some True → (∃!C. C ∈ cnf ((Φ G Π) t)
    ∧ C = { (State t (index (strips-problem.variables-of Π) v))'' })
  and ((Π) G) v = Some False → (∃!C. C ∈ cnf ((Φ G Π) t)
    ∧ C = { (State t (index (strips-problem.variables-of Π) v))' })
    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 II into a formula that can be transformed to CNF (is-cnf (Φ II t)) and that its CNF has the required form.

7.3 Soundness of the Basic SATPlan Algorithm

**Lemma** valuation-models-encoding-cnfformula-equals:
assumes is-valid-problem-strips II
shows A |= Φ II t = cnf-semantics A (cnf (Φ II t))
proof –
let ?Φ = Φ II 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-cnfformula-equals-corollary:
assumes is-valid-problem-strips II
shows A |= (Φ II t) = (∀ C ∈ cnf (Φ II t). ∃ L ∈ C. lit-semantics A L)
using valuation-models-encoding-cnfformula-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 II 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' II A k)
= { (strips-problem.operators-of II) ! (index (strips-problem.operators-of II) op) | op. op ∈ set (strips-problem.operators-of II) ^ A (Operator k (index (strips-problem.operators-of II) op))) }
proof –
let ?ops = strips-problem.operators-of II
let ?f = λop. Operator k (index ?ops op)
let ?vs = map ?f ?ops
have \( \{ \) set \((\text{filter } A \ ?v)\) = set \((\text{map} \? f (\text{filter } (\lambda \cdot ?v) \ ?ops))\)

unfolding \( \text{filter-map[of } \lambda \cdot \text{ops}. \text{Operator} k \ (\text{index } \?ops \ ?op) \ ?ops]\)..

hence \( \{ \) set \((\text{filter } A \ ?v)\) = \((\lambda \cdot \text{ops}. \text{Operator} k \ (\text{index } \?ops \ ?op)) \)

unfolding \( \text{set-map set-filter} \)

by simp 

\}

have \( \{ \) set \((\text{decode-plan'} \ \Pi \ A \ k)\) = \((\lambda \cdot \text{case } \text{Operator} k \ (\text{index } \?ops \ ?op) \ \Rightarrow \ ?ops ! \ ?i)\)

\((\lambda \cdot \text{Operator} k \ (\text{index } \?ops \ ?op)) \)

unfolding \( \text{decode-plan'-def set-map Let-def} \)

by auto

also have \( \ldots = \) \((\lambda \cdot \text{ops!} \ (\text{index } \?ops \ ?op))\)

\( \{ \) \op \in \text{set } \text{ops}. \text{A} \ (\text{Operator} k \ (\text{index } \?ops \ ?op)) \)

by force

finally show \( ?\)thesis

by blast

qed

lemma \( \text{decode-plan-set-is[simp]}: \)

set \((\Phi^{-1} \ \Pi \ A \ t)\) = \((\bigcup k \in \{..<t\}. \{ \text{decode-plan'} \ \Pi \ A \ k \})\)

unfolding \( \text{decode-plan-def SAT-Plan-Base.decode-plan-def set-map} \)

using \(\text{atLeast-upt}\)

by blast

lemma \( \text{decode-plan-step-element-then-i}: \)

assumes \( k < t \)

shows \( \{ \text{set } ((\Phi^{-1} \ \Pi \ A \ t) ! k) \)

= \((\{ \text{strips-problem.operators-of } \Pi \) \ (\text{index } \text{strips-problem.operators-of } \Pi \ ?op) \)

\mid \text{op. op} \in \text{set } ((\Pi)_{\text{op}}) \land \text{A} \ (\text{Operator} k \ (\text{index } \text{strips-problem.operators-of } \Pi \ ?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 simp

thus \( ?\)thesis

by force

qed

Show that each operator \text{op} in the \text{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 Π::′a strips-problem
  assumes k < t
    and op ∈ set ((Φ⁻¹ Π A t) ! k)
  shows op ∈ set ((Π O)"
    and A (Operator k (index (strips-problem.operators-of Π) op))

proof –
  let ?ops = strips-problem.operators-of Π
  let ?Ops = { ?ops ! (index ?ops op)
    | op. op ∈ set ((Π O) ∧ A (Operator k (index ?ops op))) }
  have op ∈ ?Ops
    using assms(2)
    unfolding decode-plan-step-element-then-i[OF assms(1)] assms
    by blast
  moreover have op ∈ set ((Π O)
    and A (Operator k (index ?ops op))
    using calculation
    by fastforce+
  ultimately show op ∈ set ((Π O)
    and A (Operator k (index ?ops op))
    by 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 distinct ((Φ⁻¹ Π A k) ! k)

proof –
  let ?ops = strips-problem.operators-of Π
  and ?πk = (Φ⁻¹ Π A k) ! k
  let ?f = λop. Operator k (index ?ops op)
  and ?g = λv. case v of Operator - k ⇒ ?ops ! k
  let ?vs = map ?f (remdups ?ops)
  have nb1: ?πk = decode-plan′ Π 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 (Φ_S^{-1} II ?A k) v ≠ None
  shows v ∈ set ((II)₁)
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] assms
  unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
  by meson
ultimately show ?thesis
  by fastforce
qed

— 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 II :: ’a strips-problem
  assumes is-valid-problem-strips II
  and ?A ⊨ Φ II t
  and k ≤ t
  and ?v ∈ set ((II)₁)
  shows (Φ_S^{-1} II ?A k) ?v
    = Some (?A (State k (index (strips-problem.variables-of II) ?v)))
proof –
  let ?vs = strips-problem.variables-of II
  let ?l = map (λx. (?x,?A (State k (index ?vs x)))) ?vs
have set ?vs ≠ { }
  using assms(4)
  by fastforce
then have map-of ?l v = Some (A (State k (index ?vs v)))
  using map-of-from-function-graph-is-some-if[of ?vs v
     λv. A (State k (index ?vs v))] assms(4)
  by fastforce
thus ?thesis
  unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
  by meson
qed

lemma decode-state-at-dom:
  assumes is-valid-problem-strips Π
  shows dom (Φ_{S^{-1}} Π A k) = set ((Π)v)
proof –
  let ?s = Φ_{S^{-1}} Π A k
  and ?vs = strips-problem.variables-of Π
  have dom ?vs = fst ' set (map (λv. (v, A (State k (index ?vs v)))) ?vs)
    using dom-map-of-conv-image-fst[of set-map[of (map (λv. (v, A (State k (index ?vs v)))) ?vs]
    unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
    by meson
  also have ... = fst ' (λv. (v, A (State k (index ?vs v)))) ' set ((Π)v)
    using set-map[of (map (λv. (v, A (State k (index ?vs v)))) ?vs]
    by simp
  also have ... = (fst ◦ (λv. (v, A (State k (index ?vs v))))) ' set ((Π)v)
    using image-comp[of fst (λv. (v, A (State k (index ?vs v))))]
    by presburger
  finally show ?thesis
    by force
qed

lemma decode-state-at-initial-state:
  assumes is-valid-problem-strips Π
  and A |= Φ Π t
  shows (Φ_{S^{-1}} Π A 0) = (Π)_I
proof –
  let I = (Π)_I
  let ?s = Φ_{S^{-1}} Π A 0
  let ?vs = strips-problem.variables-of Π
  let ?Φ = Φ Π t
  let ?Φ_I = Φ_I Π
{ have is-cnf ?Φ_I and cnf ?Φ_I ⊆ cnf ?Φ
    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 \( A (cnf \ ?\Phi_I) \)
  using cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF
assms(1)]
  assms(2)
by blast
hence \( \forall C \in cnf \ ?\Phi_I, \) clause-semantics \( A C \)
unfolding cnf-semantics-def encode-initial-state-def
by blast
} note nb1 = this
{

{ fix \( v \)
  assume \( v\)-in-dom-i: \( v \in dom \ ?I \)
  moreover { 
    have \( v\)-in-variable-set: \( v \in set ((\Pi)v) \)
      using is-valid-problem-strips-initial-of-dom assms(1) \( v\)-in-dom-i
      by auto
    hence \( (\Phi_S^{-1} \Pi A 0) v = Some (A (State 0 (index \ ?vs v))) \)
      using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
        assms(1, 2) - \( v\)-in-variable-set]
      by fast
  } note nb2 = this
  consider \( (v\)-initially-true) \ ?I v = Some True 
  | \( (v\)-initially-false) \ ?I v = Some False 
  using \( v\)-in-dom-i 
  by fastforce
  hence \( ?I v = ?s v \)
  proof (cases)
    case \( v\)-initially-true
      then obtain \( C \)
        where \( C \in cnf \ ?\Phi_I \)
        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 \( C \in \text{cnf } \Phi_{I} \)
and \( \text{c-is} \): \( C = \{ (\text{State } 0 (\text{index } ?v s))^{-1} \} \)
using \( \text{cnf-of-encode-initial-state-set assms(1)} \) \( \text{v-in-dom-i} \)
by \( \text{fastforce} \)

hence \( \mathcal{A} (\text{State } 0 (\text{index } ?v s)) = \text{False} \)
using \( \text{nb}_{1} \)
unfolding \( \text{clause-semantics-def} \)
by \( \text{fastforce} \)
thus \( \text{?thesis} \)
using \( \text{nb}_{2} \) \( \text{v-initially-false} \)
by \( \text{presburger} \)
qed

hence \( ?I \subseteq_{m} ?s \)
using \( \text{map-le-def} \)
by \( \text{blast} \)

moreover {
{
fix \( v \)
assume \( \text{v-in-dom-s}; v \in \text{dom } ?s \)
then have \( \text{v-in-set-vs}; v \in \text{set } ?v s \)
using \( \text{decode-state-at-dom[OF assms(1)]} \)
by \( \text{simp} \)

have \( \text{v-in-dom-I}; v \in \text{dom } ?I \)
using \( \text{is-valid-problem-strips-initial-of-dom assms(1)} \) \( \text{v-in-set-vs} \)
by \( \text{auto} \)

have \( \text{s-v-is}; (\Phi_{S^{-1}} \Pi \mathcal{A } 0) v = \text{Some } (\mathcal{A} (\text{State } 0 (\text{index } ?v s))) \)
using \( \text{decode-state-at-encoding-variables-equals-some-of-valuation-if assms(1, 2)} \)
  \( \text{v-in-set-vs} \)
by \( \text{metis le0} \)

consider \( \text{(s-v-is-some-true) } ?s v = \text{Some True} \)
  \| \( \text{(s-v-is-some-false) } ?s v = \text{Some False} \)
using \( \text{v-in-dom-s} \)
by \( \text{fastforce} \)

hence \( ?s v = ?I v \)

proof (cases)
case \( \text{s-v-is-some-true} \)
then have \( \mathcal{A}.of-s-v; \text{lit-semantics } \mathcal{A} ((\text{State } 0 (\text{index } ?v s))^{+}) \)
using \( \text{s-v-is} \)
by \( \text{fastforce} \)

consider \( \text{(l-v-is-some-true) } ?I v = \text{Some True} \)
  \| \( \text{(l-v-is-some-false) } ?I v = \text{Some False} \)
using \( \text{v-in-dom-I} \)
by \( \text{fastforce} \)
thus \( \text{?thesis} \)

proof (cases)
case $I$-v-is-some-true
  then show ?thesis
    using $s$-v-is-some-true
    by argo
next
case $I$-v-is-some-false

  then obtain $C$
    where $C$-in-encode-initial-state: $C \in \text{cnf } \Phi_I$
    and $C$-is: $C = \{ (\text{State } 0 (\text{index } ?vs v))^{-1} \}$
    using cnf-of-encode-initial-state-set assms($I$) $v$-in-dom-$I$
    by fastforce
  hence lit-semantics $A ((\text{State } 0 (\text{index } ?vs v))^{-1})$
    using $nb_1$
    unfolding clause-semantics-def
    by fast
  thus ?thesis
    using $A$-of-$s$-$v$
    by fastforce
qed

next
case $s$-v-is-some-false

  then have $A$-of-$s$-$v$: lit-semantics $A ((\text{State } 0 (\text{index } ?vs v))^{-1})$
    using $s$-v-is
    by fastforce
  consider ($I$-v-is-some-true) ?$I$ $v$ = Some True
    | ($I$-v-is-some-false) ?$I$ $v$ = Some False
    using $v$-in-dom-$I$
    by fastforce
  thus ?thesis
proof (cases)
case $I$-v-is-some-true

  then obtain $C$
    where $C$-in-encode-initial-state: $C \in \text{cnf } \Phi_I$
    and $C$-is: $C = \{ (\text{State } 0 (\text{index } ?vs v))^{+} \}$
    using cnf-of-encode-initial-state-set assms($I$) $v$-in-dom-$I$
    by fastforce
  hence lit-semantics $A ((\text{State } 0 (\text{index } ?vs v))^{+})$
    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
lemma decode-state-at-goal-state:
assumes "is-valid-problem-strips Π" and "A |- Φ Π t"
shows "(Π)G ⊆m ΦS⁻¹ Π A t"
proof -
  let "?vs = strips-problem.variables-of Π"
  and "?G = (Π)G"
  and "?G' = ΦS⁻¹ Π A t"
  and "?Φ = Φ Π t"
  and "?ΦG = (ΦG Π) t"
  { 
  have "is-cnf ?ΦG and cnf ?ΦG ⊆ cnf ?Φ"
    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 (?ΦG)"
    using "cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF assms(1)]"
    by blast
  hence "∀ C ∈ cnf ?ΦG. clause-semantics A C"
    unfolding "cnf-semantics-def encode-initial-state-def"
    by blast
  } note "nb1 = this"

{ 
  fix v
  assume v ∈ set "(Π)v"
  moreover have "?vs ≠ {}"
    using "calculation(1)"
    by fastforce
  moreover have "(ΦS⁻¹ Π A t) = map-of (map (λv. (v, A (State t (index ?vs v)))) ?vs)
  qed
  qed
}
unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
by metis

ultimately have \( (\Phi_S^{-1} \Pi A t) v = \text{Some} (A (State t (index ?vs v))) \)
using map-of-from-function-graph-is-some-if
by fastforce

\{ note nb2 = this
\{
  fix v
  assume v-in-dom-G: v \in \text{dom} ?G
  then have v-in-vs: v \in set ?vs
  using is-valid-problem-dom-of-goal-state-is assms(1)
  by auto
  then have decode-state-at-is: \( (\Phi_S^{-1} \Pi A t) v = \text{Some} (A (State t (index ?vs v))) \)
  using nb2
  by fastforce
  consider \( (A) ?G v = \text{Some} \text{True} \)
  | \( (B) ?G v = \text{Some} \text{False} \)
  using v-in-dom-G
  by fastforce
  hence \( ?G v = ?G' v \)

  proof (cases)
  case A
  \{
    obtain C where C \subseteq \text{cnf} ?\Phi_G \text{ and } C = \{ \{ \text{State t (index ?vs v)} \} \} \}
    using cnf-of-encode-goal-state-set(1)[OF assms(1)] A
    by auto
    then have \( \{ \text{State t (index ?vs v)} \} \in \text{cnf} ?\Phi_G \)
      by blast
    then have clause-semantics A \( \{ \text{State t (index ?vs v)} \} \)
      using nb1
      by blast
    then have lit-semantics A \( (\text{State t (index ?vs v)}) \)
      unfolding clause-semantics-def
      by blast
    hence \( A (\text{State t (index ?vs v)}) = \text{True} \)
      by force
  \}
  thus \text{thesis}
    using decode-state-at-is A
    by presburger

  next
  case B
  \{
    obtain C where C \subseteq \text{cnf} ?\Phi_G \text{ and } C = \{ \{ \text{State t (index ?vs v)} \}^{-1} \} \}
    using cnf-of-encode-goal-state-set(2)[OF assms(1)] B
    by auto
    then have \( \{ \text{State t (index ?vs v)} \}^{-1} \in \text{cnf} ?\Phi_G \)

222
by blast 
then have \( \text{clause-semantics } A \{ (\text{State } t (\text{index } ?vs v))^{-1} \} \)
using \( \text{nb}_1 \)
by blast 
then have \( \text{lit-semantics } A \{ ((\text{State } t (\text{index } ?vs v))^{-1}) \} \)
unfolding \( \text{clause-semantics-def} \)
by blast 
hence \( A (\text{State } t (\text{index } ?vs v)) = \text{False} \)
by simp 
}
thus \(?thesis\)
using \( \text{decode-state-at-is } B \)
by \(\text{presburger}\) 
qed 

— Show that the operator activation implies precondition constraints hold at every time step of the decoded plan.

\text{lemma} \( \text{decode-state-at-preconditions}: \)
\text{assumes} \( \text{is-valid-problem-strips } \Pi \)
and \( A \models \Phi \Pi \, t \)
and \( k < t \)
and \( o \in \text{set } ((\Phi^{-1} \Pi \, A \, t) \setminus k) \)
and \( v \in \text{set } (\text{precondition-of } o) \)
\text{shows} \( A (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) \, v)) \)
\text{proof} –
let \( \text{?ops} = \text{strips-problem.operators-of } \Pi \)
and \( \text{?vs} = \text{strips-problem.variables-of } \Pi \)
let \( \Phi = \Phi \Pi \, t \)
and \( \Phi_O = \text{encode-operators } \Pi \, t \)
and \( \Phi_P = \text{encode-all-operator-preconditions } \Pi \, ?ops \, t \)
\{ 
\text{have } A (\text{Operator } k (\text{index } ?ops \, o))
and \( o \in \text{set } ((\Pi)_O) \)
using \( \text{decode-plan-step-element-then}[\text{OF assms}(3, 4)] \)
by blast +
moreover obtain \( C \)
where \( \text{clause-is-in-operator-encoding}: C \in \text{cnf } ?\Phi_P \)
and \( C = \{ (\text{Operator } k (\text{index } ?ops \, o))^{-1}, (\text{State } k (\text{index } ?vs v))^+ \} \)
using \( \text{cnf-of-encode-all-operator-preconditions-contains-clause-if}[\text{OF assms}(1, 3)] \)
calculation(2) \text{ assms}(5)] 
by blast
moreover have \( \text{clause-semantics } A \cdot \Phi_P \cdot \forall C \in \text{cnf } ?\Phi_P, \text{clause-semantics } A \, C \)
using cnf-semantics-monotonous-in-cnf-subsets-if[OF assms(2)]
is-cnfs-de-encode-problem[OF assms(1)]
cnf-of-operator-precondition-encoding-subset-encoding
unfolding cnf-semantics-def
by blast

ultimately have lit-semantics A (Pos (State k (index ?vs v)))
unfolding clause-semantics-def
by fastforce
}
thus ?thesis
unfolding lit-semantics-def
by fastforce
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 encode-problem-parallel-correct-i:
assumes is-valid-problem-strips Π
and A|= Φ Π 0
shows cnf ((Φ G Π) 0) ⊆ cnf (Φ I Π)
proof –
let ?vs = strips-problem.variables-of Π
and Π I = (Π)I
and ?G = (Π)G
and ?Φ I = Φ I Π
and ?Φ G = (Φ G Π) 0
and ?Φ = Φ Π 0
— Show that the model of the encoding is also a model of the partial encodings.
using assms(2) encode-problem-has-model-then-also-partial-encodings(1, 2)
unfolding encode-problem-def encode-initial-state-def encode-goal-state-def
by blast+
— Show that every clause in the CNF of the goal state encoding Φ G is also in the
CNF of the initial state encoding Φ I thus making it a subset. We can conclude this
from the fact that both Φ I and Φ 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.
{
  fix C’
  assume C’-in-cnf-Φ G: C’ ∈ cnf ?Φ G
  then obtain v
    where v-in-us: v ∈ set ?vs

224
and \( G \neq \text{None} \) \( \Rightarrow \) \( G \neq \text{None} \)
and \( C \)'-is: \( C' = \{ \text{literal-formula-to-literal (encode-state-variable 0 (index} \( \text{?vs v})\) \}
using \( \text{cnf-of-encode-goal-set}\text{-ii}[\text{OF assms(1)}] \)
by \( \text{auto} \)
obtain \( C \)
where \( C \)-in-cnf-\( \Phi_I; C \in \text{cnf} \[ \Phi_I \]
and \( C \)-is: \( C = \{ \text{literal-formula-to-literal (encode-state-variable 0 (index} \( \text{?vs v})\) \}
using \( \text{cnf-of-encode-initial-set}\text{-ii}[\text{OF assms(1)}] \text{ v-in-vs} \)
by \( \text{auto} \)
\{ \let \( ?L = \text{literal-formula-to-literal (encode-state-variable 0 (index} \( \text{?vs v}) \text{ (?I v)} \)
have \( \{ \text{?L} \} \in \text{cnf} \[ \Phi_I \]
using \( \text{C-in-cnf-}\Phi_I \text{ C-is} \)
by \( \text{blast} \)
\}
\}
\let \( ?L' = \text{literal-formula-to-literal (encode-state-variable 0 (index} \( \text{?vs v}) \text{ (?G v)} \)
have \( \{ \text{?L'} \} \in \text{cnf} \[ \Phi_G \]
using \( \text{C'-in-cnf-}\Phi_G \text{ C'-is} \)
by \( \text{blast} \)
\}
\}
\let \( ?I v = ?G v \)
proof (rule \( \text{ccontr} \))
assume contradiction: \( ?I v \neq ?G v \)
moreover have \( ?I v \neq \text{None} \)
using \( \text{v-in-vs is-valid-problem-strips-initial-of-dom assms(1)} \)
by \( \text{auto} \)
ultimately consider \( (A) ?I v = \text{Some True} \land ?G v = \text{Some False} \)
\mid \( (B) ?I v = \text{Some False} \land ?G v = \text{Some True} \)
using \( \text{G-of-v-is-not-None} \)
by force
thus \( \text{False} \)
using \( \text{lit-semantics-}\Phi_I \text{-L lit-semantics-}\Phi_I \text{-L'} \)
unfolding \( \text{encode-state-variable-def} \)

225
by \((cases, \text{fastforce}+)\)

qed

\}

hence \(C' \in \text{cnf } ?\Phi_I\)

using \(C \text{-is } C \text{-in-cnf-}\Phi_I \quad C' \text{-is } C' \text{-in-cnf-}\Phi_G\)

by \texttt{argo}\}

thus \(?\text{thesis}\)

by \texttt{blast}\qend

— Show that the encoding secures that for every parallel operator \(\text{ops}\) decoded from the plan at every time step \(t < \text{length } \pi\) the following hold:

1. \(\text{ops}\) is applicable, and

2. the effects of \(\text{ops}\) are consistent.

\textbf{lemma} \texttt{encode-problem-parallel-correct-ii}:

\textbf{assumes} \(\text{is-valid-problem-strips } \Pi\)

\text{and} \(\mathcal{A} \models \Phi \Pi t\)

\text{and} \(k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t)\)

\textbf{shows} \(\text{are-all-operators-applicable } (\Phi_{S^{-1}} \Pi \mathcal{A} k)\)

\((\Phi^{-1} \Pi \mathcal{A} t) ! k)\)

\text{and} \(\text{are-all-operator-effects-consistent } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)\)

\textbf{proof} –

let \(?\text{vs } = \text{strips-problem.variables-of } \Pi\)

\text{and} \(?\text{ops } = \text{strips-problem.operators-of } \Pi\)

\text{and} \(?\pi = \Phi^{-1} \Pi \mathcal{A} t\)

\text{and} \(?s = \Phi_{S^{-1}} \Pi \mathcal{A} k\)

let \(?\Phi = \Phi \Pi t\)

\text{and} \(?\Phi_E = \text{encode-all-operator-effects } \Pi ?\text{ops } t\)

\textbf{have} \(k-\text{lt-}t: k < t\)

\text{using} \(\text{decode-plan-length } \text{assms}(3)\)

\text{by} \texttt{metis}\}

\{

\{\text{fix } op \ v\)

\text{assume} \(\text{op-in-kth-of-decoded-plan-set: } op \in \text{set } (?\pi ! k)\)

\text{and} \(v\text{-in-precondition-set: } v \in \text{set } \text{(precondition-of } op)\)

\{

\text{have } \mathcal{A} (\text{Operator } k (\text{index } ?\text{ops } op))\)

\text{using} \(\text{decode-plan-step-element-then}[OF \ k-\text{lt-}t \ \text{op-in-kth-of-decoded-plan-set}]\)

\text{by} \texttt{blast}\)

\text{hence } \mathcal{A} (\text{State } k (\text{index } ?\text{vs } v))\)

\text{using} \(\text{decode-state-at-preconditions}[\)

\(OF \ \text{assms}(1, 2) \ - \ \text{op-in-kth-of-decoded-plan-set } v\text{-in-precondition-set}]\)

\(k-\text{lt-}t\)

\text{by} \texttt{blast}\}

\}

226
moreover have \( k \leq t \)
using \( k \lt t \)
by auto
moreover {
  have \( \text{op} \in \text{set } ((\Pi)_{\Theta}) \)
  using decode-plan-step-element-then[\( OF \ k \lt t \ \text{op-in-kth-of-decoded-plan-set} \)]
  by simp
  then have \( v \in \text{set } ((\Pi)_{\nu}) \)
  using is-valid-problem-strips-operator-variable-sets(\( I \)) assms(\( I \))
  \( v \)-in-precondition-set
  by simp
}
ultimately have \( (\Phi_{S^{-1}} \ \Pi \ A \ k) \ v = \text{Some True} \)
using decode-state-at-encoding-variables-equals-some-of-valuation-if[\( OF \ assms(1, 2) \)]
by presburger
}
hence are-all-operators-applicable \( ?s \ (? \pi \ k) \)
using are-all-operators-applicable-set[\( of \ ?s \ ? \pi \ k \)]
by blast
} moreover {
{
  fix \( \text{op}_1 \ \text{op}_2 \)
  assume \( \text{op}_1\text{-in-k-th-of-decoded-plan}: \text{op}_1 \in \text{set } ((\Phi^{-1} \ \Pi \ A \ t) \ ! \ k) \)
  and \( \text{op}_2\text{-in-k-th-of-decoded-plan}: \text{op}_2 \in \text{set } ((\Phi^{-1} \ \Pi \ A \ t) \ ! \ k) \)
  have \( \text{op}_1\text{-in-set-ops}: \text{op}_1 \in \text{set } ((\Pi)_{\Theta}) \)
  and \( \text{op}_2\text{-in-set-ops}: \text{op}_2 \in \text{set } ((\Pi)_{\Theta}) \)
  and \( \text{op}_1\text{-active-at-k}: \neg \text{lit-semantics } A (((\text{Operator } k \ (\text{index } ?\text{ops } \text{op}_1))^{-1}) \)
  and \( \text{op}_2\text{-active-at-k}: \neg \text{lit-semantics } A (((\text{Operator } k \ (\text{index } ?\text{ops } \text{op}_2))^{-1}) \)
  subgoal
  using decode-plan-step-element-then[\( OF \ k \lt t \ \text{op}_1\text{-in-k-th-of-decoded-plan} \)]
  by simp
  subgoal
  using decode-plan-step-element-then[\( OF \ k \lt t \ \text{op}_2\text{-in-k-th-of-decoded-plan} \)]
  by force
  subgoal
  using decode-plan-step-element-then[\( OF \ k \lt t \ \text{op}_1\text{-in-k-th-of-decoded-plan} \)]
  by simp
  subgoal
  using decode-plan-step-element-then[\( OF \ k \lt t \ \text{op}_2\text{-in-k-th-of-decoded-plan} \)]
  by simp
  done
{
  fix \( v \)
  assume \( v\text{-in-add-effects-set-of-\text{op}_1}: v \in \text{set } (\text{add-effects-of } \text{op}_1) \)
  and \( v\text{-in-delete-effects-set-of-\text{op}_2}: v \in \text{set } (\text{delete-effects-of } \text{op}_2) \)
  let \( \Phi_{C_1} = ((\text{Operator } k \ (\text{index } ?\text{ops } \text{op}_1))^{-1}, \)
  \( (\text{State } (\text{Suc } k \ (\text{index } ?\text{vs } v)))^{+} \)}
\[ \text{and } \mathcal{C}_2 = \{(\text{Operator } k (\text{index } \mathcal{ops} \ op_2))^{-1}, \]
\[(\text{State } (\text{Suc } k) (\text{index } \mathcal{vs} \ v))^{-1}\} \]

have \( \mathcal{C}_1 \in \text{cnf } \Phi_E \) and \( \mathcal{C}_2 \in \text{cnf } \Phi_E \)

subgoal
- using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[\( \text{OF assms(1) } k\lt-t \ op_1\text{-in-set-ops } v\text{-in-add-effects-set-of-op}_1 \)]
  by blast

subgoal
- using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[\( \text{OF assms(1) } k\lt-t \ op_2\text{-in-set-ops } v\text{-in-delete-effects-set-of-op}_2 \)]
  by blast

done

then have \( \mathcal{C}_1 \in \text{cnf } \Phi \) and \( \mathcal{C}_2 \in \text{cnf } \Phi \)

using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
by blast+

then have \( C_1\text{-true: }\text{clause-semantics } \mathcal{A} \) \( \mathcal{C}_1 \) and \( C_2\text{-true: }\text{clause-semantics } \mathcal{A} \) \( \mathcal{C}_2 \)

using valuation-models-encoding-cnf-formula-equals[\( \text{OF assms(1)} \) assms(2)]

unfolding cnf-semantics-def
by blast+

have lit-semantics \( \mathcal{A} \) \((\text{State } (\text{Suc } k) (\text{index } \mathcal{vs} \ v)^+)\)

and lit-semantics \( \mathcal{A} \) \((\text{State } (k + 1) (\text{index } \mathcal{vs} \ v)^-1)\)

subgoal
- using \( \text{op}_1\text{-active-at-k } C_1\text{-true} \)

unfolding clause-semantics-def
by blast

subgoal
- using \( \text{op}_2\text{-active-at-k } C_2\text{-true} \)

unfolding clause-semantics-def
by fastforce

done

hence False
by auto

} moreover { 
fix \( v \)
assume \( v\text{-in-delete-effects-set-of-op}_1: v \in \text{set } (\text{delete-effects-of } \text{op}_1) \)
and \( v\text{-in-add-effects-set-of-op}_2: v \in \text{set } (\text{add-effects-of } \text{op}_2) \)

let \( \mathcal{C}_1 = \{(\text{Operator } k (\text{index } \mathcal{ops} \ op_1))^{-1}, (\text{State } (\text{Suc } k) (\text{index } \mathcal{vs} \ v))^{-1}\} \)

and \( \mathcal{C}_2 = \{(\text{Operator } k (\text{index } \mathcal{ops} \ op_2))^{-1}, (\text{State } (\text{Suc } k) (\text{index } \mathcal{vs} \ v))^{-1}\} \)

have \( \mathcal{C}_1 \in \text{cnf } \Phi_E \) and \( \mathcal{C}_2 \in \text{cnf } \Phi_E \)

subgoal
- using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[\( \text{OF assms(1) } k\lt-t \ op_1\text{-in-set-ops } v\text{-in-delete-effects-set-of-op}_1 \)]
  by fastforce

subgoal
- using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[\( \text{OF assms(1) } k\lt-t \ op_2\text{-in-set-ops } v\text{-in-add-effects-set-of-op}_2 \)]
  by simp

228
done
then have \(?C_1 \in \text{cnf } \Phi\) and \(?C_2 \in \text{cnf } \Phi\)
using \(\text{cnf-of-encode-all-operator-effects-subset-cnfxof-encode-problem}\)
by blast+
then have \(C_1\text{-true: clause-semantics } \mathcal{A}\) \(\text{and}\)
\(C_2\text{-true: clause-semantics } \mathcal{A}\)
using \(\text{valuation-models-encoding-cnfx-formula-equals}[\text{OF assms(1)}] \text{assms(2)}\)
unfolding \(\text{cnf-semantics-def}\)
by blast+
have \(\text{lit-semantics } \mathcal{A}\) \(((\text{State } \text{Suc } k) \text{ (index } \text{?vs } v)\leftarrow)\)
and \(\text{lit-semantics } \mathcal{A}\) \(((\text{State } (k + 1) \text{ (index } \text{?vs } v)\leftarrow)\)
subgoal
using \(\text{op1-active-at-k } C_1\text{-true}\)
unfolding \(\text{clause-semantics-def}\)
by blast
subgoal
using \(\text{op2-active-at-k } C_2\text{-true}\)
unfolding \(\text{clause-semantics-def}\)
by fastforce
done
hence \(\text{False}\)
by simp
}
ultimately have \(\text{set}(\text{add-effects-of } \text{op1}) \cap \text{set}(\text{delete-effects-of } \text{op2}) = \{\}\)
and \(\text{set}(\text{delete-effects-of } \text{op1}) \cap \text{set}(\text{add-effects-of } \text{op2}) = \{\}\)
by blast+
}

hence \(\text{are-all-operator-effects-consistent } (?\pi ! k)\)
using \(\text{are-all-operator-effects-consistent-set}[\text{of } ?\pi ! k]\)
by blast
}
ultimately show \(\text{are-all-operators-applicable } ?s ( ?\pi ! k)\)
and \(\text{are-all-operator-effects-consistent } (?\pi ! k)\)
by blast+
qed

— Show that for all operators \(\text{op}\) at timestep \(k\) of the plan \(\Phi^{-1} \Pi \mathcal{A} t\) decoded from the model \(\mathcal{A}\), both add effects as well as delete effects will hold in the next timestep \text{Suc } k.

**lemma** \text{encode-problem-parallel-correct-iii:}
assumes \(\text{is-valid-problem-strips } \Pi\)
and \(\mathcal{A} \models \Phi \Pi t\)
and \(k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t)\)
and \(\text{op } \in \text{set } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)\)
shows \(v \in \text{set } (\text{add-effects-of } \text{op})\)
\(\rightarrow (\Phi_{S^{-1}} \Pi \mathcal{A} \text{ (Suc } k)) v = \text{Some True}\)
and \(v \in \text{set } (\text{delete-effects-of } \text{op})\)
\(\rightarrow (\Phi_{S^{-1}} \Pi \mathcal{A} \text{ (Suc } k)) v = \text{Some False}\)
proof –
let ?ops = strips-problem.operators-of II
and ?vs = strips-problem.variables-of II
let ?ΦF = encode-all-operator-effects II ?ops t
and ?A = \( \bigcup (t, op) \in \{0..<t\} \times set ((II)O) .
\{\{ (Operator t (index ?ops op))^{-1}, (State (Suc t) (index ?vs v))^+ \} \mid v. v \in set (add-effects-of op)\} \)
and ?B = \( \bigcup (t, op) \in \{0..<t\} \times set ((II)O) .
\{\{ (Operator t (index ?ops op))^{-1},
(State (Suc t) (index ?vs v))^{-1} \} \mid v. v \in set (delete-effects-of op)\} \)
have k-lt-t: k < t
using decode-plan-length assms (3)
by metis
have op-is-valid: op \in set ((II)O)
using decode-plan-step-element-then[OF k-lt-t assms (4)]
by blast
have k-op-included: (k, op) \in \{\{ (Operator t (index ?ops op))^{-1},
(State (Suc t) (index ?vs v))^{-1} \} \mid v. v \in set (add-effects-of op)\}
by fastforce
thus \( v \in set (add-effects-of op) \)
\( \rightarrow (\PhiF^{-1} II A (Suc k)) \ v = Some True \)
and \( v \in set (delete-effects-of op) \)
\( \rightarrow (\PhiF^{-1} II A (Suc k)) \ v = Some False \)
proof (auto)
assume v-is-add-effect: \( v \in set (add-effects-of op) \)
have \( A (Operator k (index ?ops op)) \)
using decode-plan-step-element-then[OF k-lt-t assms (4)]
by blast
moreover {
have \( \{\{ (Operator k (index ?ops op))^{-1}, (State (Suc k) (index ?vs v))^+ \} \}
\in \{\{ (Operator k (index ?ops op))^{-1}, (State (Suc k) (index ?vs v))^+ \} \mid v. v \in set (add-effects-of op)\} \)
using v-is-add-effect
by blast
then have \( \{\{ (Operator k (index ?ops op))^{-1}, (State (Suc k) (index ?vs v))^+ \} \} \in \ ?A \)
using k-op-included cnf-of-operator-encoding-structure
UN-iff[of \{\{ (Operator t (index ?ops op))^{-1}, (State (Suc t) (index ?vs v))^{-1} \} \mid t. t \in set ((II)O)\}]
by blast
then have \( \{\{ (Operator k (index ?ops op))^{-1}, (State (Suc k) (index ?vs v))^+ \} \}\in \ \bigcup \ ?A \)
using Union-iff[of \{\{ (Operator k (index ?ops op))^{-1}, (State (Suc k) (index ?vs v))^+ \} \}]
by blast

230
moreover have $\bigcup \mathcal{B} \subseteq \text{cnf } \Phi_F$
  using \text{cnf-of-encode-all-operator-effects-structure}
  by blast
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 \text{in-monono}[\bigcup \mathcal{A} \text{ cnf } \Phi_F]
  by presburger

ultimately have $\mathcal{A} \ (\text{State } \text{(Suc } k \ (\text{index } ?vs \ v))$
  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 Bex-def
  by fastforce
thus $(\Phi_{S}^{-1} \Pi \mathcal{A} \ (\text{Suc } k) = \text{Some True}$
  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 fastforce
next
  assume v-is-delete-effect; $v \in \text{set}(\text{delete-effects-of op})$
  have $\mathcal{A} \ (\text{Operator } k \ (\text{index } ?ops \ op))$
    using decode-plan-step-element-then[\text{OF } k-lt-t \text{ assms}(4)]
    by blast
moreover {
  have $\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } \text{(Suc } k \ (\text{index } ?vs \ v))^{-1}\} \in \{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } \text{(Suc } k \ (\text{index } ?vs \ v))^{-1}\}
  \mid v, v \in \text{set}(\text{delete-effects-of op})\}$
  using v-is-delete-effect
  by blast
  then have $\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } \text{(Suc } k \ (\text{index } ?vs \ v))^{-1}\} \in \mathcal{B}$
    using k-op-included \text{cnf-of-encode-all-operator-effects-structure}
    UN-iff[of $\{(\text{Operator } t \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } \text{(Suc } t \ (\text{index } ?vs \ v))^{-1}\}$
      $\times \text{set}((\Pi)O)]$
    by blast
  then have $\{(\text{Operator } k \ (\text{index } ?ops \ op))^{-1}, \ (\text{State } \text{(Suc } k \ (\text{index } ?vs \ v))^{-1}\} \in \bigcup \mathcal{B}$
    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 $\bigcup \mathcal{B} \subseteq \text{cnf } \Phi_F$
  using \text{cnf-of-encode-all-operator-effects-structure} \text{Un-upper2}[of $\bigcup \mathcal{B} \bigcup \mathcal{A}$]

231
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
\quad \text{using } \text{in-mono[of } \bigcup B \text{ cnf } \Phi_F]
\quad \text{by } \text{presburger}
}

ultimately have \neg \mathcal{A} \ (\text{State } (\text{Suc } k) \ (\text{index } ?vs \ v))
\quad \text{using } \text{cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem}
\quad \text{valuation-models-encoding-cnf-formula-equals-corollary}[\text{OF } \text{assms}(1)]
\quad \text{assms}(2)
\quad \text{by } \text{fastforce}
\quad \text{moreover have } \text{Suc } k \leq t
\quad \text{using } \text{k-lt-t}
\quad \text{by } \text{fastforce}
\quad \text{moreover have } v \in \text{set}(\prod \nu)
\quad \text{using } \text{v-is-delete-effect is-valid-problem-strips-operator-variable-sets}(3)
\quad \text{assms}(1)
\quad \text{op-is-valid}
\quad \text{by } \text{auto}
\quad \text{ultimately show } (\Phi_S^{-1} \prod \mathcal{A} \ (\text{Suc } k)) \ v = \text{Some False}
\quad \text{using } \text{decode-state-at-encoding-variables-equals-some-of-valuation-if}[\text{OF}
\quad \text{assms}(1, 2)]
\quad \text{by } \text{auto}
\quad \text{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.

\begin{align*}
\mathcal{A} \models (\neg a_i \land a_{i+1}) & \rightarrow \bigvee \{op_i, k : op_i \text{ has add effect } a\} \\
\mathcal{A} \models (a_i \land \neg a_{i+1}) & \rightarrow \bigvee \{op_i, k : op_i \text{ has delete effect } a\}
\end{align*}

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

\begin{align*}
\mathcal{A} \models (\neg a_i \land a_{i+1}) & \equiv \mathcal{A} \models a_i \lor \neg a_{i+1} \\
\mathcal{A} \models (a_i \land \neg a_{i+1}) & \equiv \mathcal{A} \models \neg a_i \lor a_{i+1}
\end{align*}

hence

\begin{align*}
\mathcal{A} \models (\neg a_i \lor a_{i+1}) \land (a_i \lor \neg a_{i+1}) \\
\models \mathcal{A} \models \{\{\neg a_i, a_{i+1}\}, \{a_i, \neg a_{i+1}\}\}
\end{align*}

The lemma characterizes this simplification. ⁹

⁹This part of the soundness proof is only treated very briefly in [3, theorem 3.1, p.1044]
lemma encode-problem-parallel-correct-iv:
fixes II :: 'a strips-problem
assumes is-valid-problem-strips II
and A |= \Phi II t
and k < t
and v \in set ((II)_{\nu})
and \neg (\exists op \in set ((\Phi^{-1} II A t) \setminus k).
\neg v \in set (add-effects-of op) \cup v \in set (delete-effects-of op))
shows cnf-semantics A \{ (State k (index (strips-problem.variables-of II) v))^{-1} , (State (Suc k) (index (strips-problem.variables-of II) v))^{+} \}
and cnf-semantics A \{ (State k (index (strips-problem.variables-of II) v))^{+} , (State (Suc k) (index (strips-problem.variables-of II) v))^{-1} \}
proof
let ?vs = strips-problem.variables-of II
and ?ops = strips-problem.operators-of II
let ?\Phi = \Phi II t
and ?\Phi_F = encode-all-frame-axioms II t
and ?\Phi_k = (\Phi^{-1} II A t) \setminus k
and ?A = \bigcup (k, v) \in (\{0..<t\} \times set ?vs).
\{ (State k (index ?vs v))^{+} , (State (Suc k) (index ?vs v))^{-1} \}
\cup \{ (Operator k (index ?ops op))^{+} | op. op \in set ?ops \land v \in set (add-effects-of op) \}
and ?B = \bigcup (k, v) \in (\{0..<t\} \times set ?vs).
\{ (State k (index ?vs v))^{-1} , (State (Suc k) (index ?vs v))^{+} \}
\cup \{ (Operator k (index ?ops op))^{-1} | op. op \in set ?ops \land v \in set (delete-effects-of op) \}
and ?C = \{ (State k (index ?vs v))^{+} , (State (Suc k) (index ?vs v))^{-1} \}
\cup \{ (Operator k (index ?ops op))^{+} | op. op \in set ?ops \land v \in set (add-effects-of op) \}
and ?C' = \{ (State k (index ?vs v))^{-1} , (State (Suc k) (index ?vs v))^{+} \}
\cup \{ (Operator k (index ?ops op))^{-1} | op. op \in set ?ops \land v \in set (delete-effects-of op) \}
have k-v-included: (k, v) \in (\{0..<t\} \times set ((II)_{\nu}))
using assms(3, 4)
by blast
have operator-encoding-subset-encoding: cnf ?\Phi_F \subseteq cnf ?\Phi
using cnf-of-encode-problem-structure(4)
unfolding encode-problem-def
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 op is in set ?ops, either 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 \A (Operator k (index ?ops op) = \bot by definition of decode-plan; moreover,

- assuming op is not in set ?ops—this is implicitly encoded as Operator k (length ?ops) and \A (Operator k (length ?ops)) may or may not be true—, then it’s not in either of the sets.

233
Altogether, we have the situation that the sets only have members $\text{Operator } k (\text{index } ?ops \ op)$ with $A (\text{Operator } k (\text{index } ?ops \ op)) = \bot$, 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:

$$\forall \ op. \ op \in \{(\text{Operator } k (\text{index } ?ops \ op))^+\}$$
$$\quad | \ op. \ op \in \text{set } ?ops \wedge v \in \text{set } (\text{add-effects-of } op) \}$$
$$\rightarrow \neg \text{lit-semantics } A \ op$$

and

$$\forall \ op. \ op \in \{(\text{Operator } k (\text{index } ?ops \ op))^+\}$$
$$\quad | \ op. \ op \in \text{set } ?ops \wedge v \in \text{set } (\text{delete-effects-of } op) \}$$
$$\rightarrow \neg \text{lit-semantics } A \ op$$

Hence, the operators are irrelevant for $\text{cnf-semantics } A \{ C \}$ where $C$ is a clause encoding a positive or negative transition frame axiom for a given variable $v$ of the problem.

$$\{$$
let $?add = \{(\text{Operator } k (\text{index } ?ops \ op))^+\}$
$$\quad | \ op. \ op \in \text{set } ?ops \wedge v \in \text{set } (\text{add-effects-of } op) \}$$
and $?delete = \{(\text{Operator } k (\text{index } ?ops \ op))^+\}$
$$\quad | \ op. \ op \in \text{set } ?ops \wedge v \in \text{set } (\text{delete-effects-of } op) \}$$

$$\{$$
fix $op$
assume operator-encoding-in-add: $(\text{Operator } k (\text{index } ?ops \ op))^+ \in ?add$
hence $\neg \text{lit-semantics } A ((\text{Operator } k (\text{index } ?ops \ op))^+)$
proof (cases $op \in \text{set } ?\pi_k$)
case $True$
then have $v \not\in \text{set } (\text{add-effects-of } op)$
using assms(5)
by simp
then have $(\text{Operator } k (\text{index } ?ops \ op))^+ \not\in ?add$
by fastforce
thus $?thesis$
using operator-encoding-in-add
by blast
next
case $False$
then show $?thesis$
proof (cases $op \in \text{set } ?ops$)
case $True$
$\{$
let $?A = \{ ?ops \mid \text{index } ?ops \ op \mid op.$
$$\quad op \in \text{set } ((\Pi)\O) \wedge A (\text{Operator } k (\text{index } ?ops \ op))\}$$
assume lit-semantics $A ((\text{Operator } k (\text{index } ?ops \ op))^+)$
moreover have operator-active-at-k: $A (\text{Operator } k (\text{index } ?ops \ op))$
using calculation

234
by auto
moreover have $op \in \text{set } (\Pi)_{\sigma}$
  using True
by force
moreover have $(?\text{ops } \text{! } \text{index } ?\text{ops } op) \in ?A$
  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 $?\text{thesis}$
by blast
next
case False
then have $op \notin \{op \in \text{set } ?\text{ops}. v \in \text{set } (\text{add-effects-of } op)\}$
by blast
moreover have $?\text{add} = (\lambda op. (\text{Operator } k (\text{index } ?\text{ops } op))^+)$
  " \{ op \in \text{set } ?\text{ops}. v \in \text{set } (\text{add-effects-of } op) \}
using setcompr_eq_image[of \lambda op. (\text{Operator } k (\text{index } ?\text{ops } op))^+
  \lambda op. op \in \text{set } ?\text{ops } \land v \in \text{set } (\text{add-effects-of } op)]
by blast
ultimately have $(\text{Operator } k (\text{index } ?\text{ops } op))^+ \notin ?\text{add}$
by force
thus $?\text{thesis}$ using operator-encoding-in-add
by blast
qed
qed
moreover {
fix op
assume operator-encoding-in-delete: $(\text{Operator } k (\text{index } ?\text{ops } op))^+ \in ?\text{delete}$
hence ~lit-semantics $A ((\text{Operator } k (\text{index } ?\text{ops } op))^+)$
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 $(\text{Operator } k (\text{index } ?\text{ops } op))^+ \notin ?\text{delete}$
by fastforce
thus $?\text{thesis}$
  using operator-encoding-in-delete
by blast
next
case False

then show \( \text{thesis} \)
proof (cases \( \text{op} \in \text{set} \ ?\text{ops} \))
  case True
  
  let \( \?A = \{ \ ?\text{ops} ! \ \text{index} \ ?\text{ops} \ \text{op} \ |
  \ \text{op} \in \text{set} \ ((\Pi)O) \ \land \ ?A \ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op})) \} \)
assume \( \text{lit-semantics} \ ?A \ ((\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+) \)
moreover have \( \text{operator-active-at-k} : \ ?A \ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op})) \)
  using calculation
by auto
moreover have \( \text{op} \in \text{set} \ ((\Pi)O) \)
by force
ultimately have \( \text{op} \in \text{set} \ ?\pi_k \)
using \( \text{decode-plan-step-element-then-i[OF assms(\_)]} \)
by auto
hence False
using False
by blast

thus \( \text{thesis} \)
by blast

next
  case False
  then have \( \text{op} \notin \{ \ \text{op} \in \text{set} \ ?\text{ops}. \ \text{v} \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \} \)
  by blast
moreover have \( \?\text{delete} = \)
  \( \lambda \text{op}. \ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \)
  \( \cdot \{ \ \text{op} \in \text{set} \ ?\text{ops}. \ \text{v} \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \} \)
using setcompr-eq-image[of \( \lambda \text{op}. \ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \)
\( \lambda \text{op}. \ \text{op} \in \text{set} \ ?\text{ops} \ \land \ \text{v} \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \)]
by blast
ultimately have \( (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \notin \ ?\text{delete} \)
by force
thus \( \text{thesis using \ operator-encoding-in-delete} \)
by blast
qed

\text{note nb = this}

\{ \let \?\text{Ops} = \{ \ (\text{Operator} \ k \ (\text{index} \ ?\text{ops} \ \text{op}))^+ \} \}
have \( ?\text{Ops} \subseteq ?C \)
by blast
moreover have \( ?C - ?\text{Ops} = \{ \text{(State} \ k \ \text{(index} \ ?v \ v) \text{)}^{+}, \ \text{(State} \ \text{(Suc} \ k) \ \text{(index} \ ?v \ v) \text{)}^{-1} \) 
by fast
moreover have \( \forall L \in ?\text{Ops}. \neg \text{lit-semantics} \ A \ L \)
using \( \text{nb}(1) \)
by blast

ultimately have \( \text{clause-semantics} \ A \ ?C \)
= \( \text{clause-semantics} \ A \ \{ \text{(State} \ k \ \text{(index} \ ?v \ v) \text{)}^{+}, \ \text{(State} \ \text{(Suc} \ k) \ \text{(index} \ ?v \ v) \text{)}^{-1} \) 
using \( \text{lit-semantics-reducible-to-subset-if}[\text{of} \ ?\text{Ops} \ ?C] \)
by presburger

} moreover {
let \( ?\text{Ops}' = \{ \text{(Operator} \ k \ \text{(index} \ ?ops \ op) \text{)}^{+} \mid \text{op. op} \in \text{set} \ ?\text{ops} \ \land \ v \in \text{set} \ (\text{delete-effects-of} \ \text{op}) \} \)
have \( ?\text{Ops}' \subseteq ?C' \)
by blast
moreover have \( ?C' - ?\text{Ops}' = \{ \text{(State} \ k \ \text{(index} \ ?v \ v) \text{)}^{-1}, \ \text{(State} \ \text{(Suc} \ k) \ \text{(index} \ ?v \ v) \text{)}^{+} \) 
by fast
moreover have \( \forall L \in ?\text{Ops}'. \neg \text{lit-semantics} \ A \ L \)
using \( \text{nb}(2) \)
by blast

ultimately have \( \text{clause-semantics} \ A \ ?C' \)
= \( \text{clause-semantics} \ A \ \{ \text{(State} \ k \ \text{(index} \ ?v \ v) \text{)}^{-1}, \ \text{(State} \ \text{(Suc} \ k) \ \text{(index} \ ?v \ v) \text{)}^{+} \) 
using \( \text{lit-semantics-reducible-to-subset-if}[\text{of} \ ?\text{Ops}' \ ?C'] \)
by presburger

} moreover {
have \( \text{cnf-semantics-} A \cdot \Phi: \text{cnf-semantics} \ A \ \{ \text{cnf} \ ?\Phi \} \)
using \( \text{valuation-models-encoding-cnf-formula-equals}[\text{OF} \ \text{assms}(1)] \ \text{assms}(2) \)
by blast
have \( k-v\text{-included}: (k, v) \in \{\ldots<t\} \times \text{set} \ (\Pi) v) \)
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 \( k-v\text{-included} \)
by force+

\text{then have} \ ?C \in \text{cnf} \ ?\Phi_F \ \text{and} \ ?C' \in \text{cnf} \ ?\Phi_F \)
subgoal
using \( \text{cnf-of-encode-all-frame-axioms-structure} \ \text{UnI1}[\text{of} \ ?C \ \bigcup ?A \ \bigcup ?B] \)
c-in-un-a
by metis

237
  by metis
done
then have { ?C } ⊆ cnf ?Φ and c'-subset-frame-axiom-encoding: { ?C' } ⊆ cnf ?Φ
  by blast+
then have { ?C } ⊆ cnf ?Φ and { ?C' } ⊆ cnf ?Φ
subgoal using operator-encoding-subset-encoding
  by fast
subgoal using c'-subset-frame-axiom-encoding operator-encoding-subset-encoding
  by fast
done

hence cnf-semantics A { ?C } and cnf-semantics A { ?C' }
  using cnf-semantics-AΦ model-for-cnf-is-model-of-all-subsets
  by fastforce+
}
ultimately show cnf-semantics A {{ (State k (index ?vs v))^{-1}, (State (Suc k) (index ?vs v))^+ } }
  and cnf-semantics A {{ (State k (index ?vs v))^+, (State (Suc k) (index ?vs v))^{-1} } }
  unfolding cnf-semantics-def
  by blast+
qed

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 (Φ_{S}^{-1} Π A (Suc k)) = execute-parallel-operator (Φ_{S}^{-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 ?s_k = Φ_{S}^{-1} Π A k
and ?s_k' = Φ_{S}^{-1} Π A (Suc k)
let ?t_k' = execute-parallel-operator ?s_k (?π ! k)
and ?π_k = ?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 ?s_k ?π_k ∧ are-all-operator-effects-consistent ?π_k
  using encode-problem-parallel-correct-ii[OF assms(1, 2, 3)]
We show the goal in classical fashion by proving that

\[ \Phi_{S^{-1}} \Pi A \quad (\text{Suc } k) \ v = \text{execute-parallel-operator} \ (\Phi_{S^{-1}} \Pi A \ k) \ ((\Phi^{-1} \Pi A \ t) ! k) \ v \]

— i.e. the state decoded at time \( k + 1 \) is equivalent to the state obtained by executing the parallel operator \((\Phi^{-1} \Pi A \ t) ! k \) on the previous state \( \Phi_{S^{-1}} \Pi A \ k \)—for all variables \( v \) given \( k < t \), a model \( A \), and makespan \( t \).

moreover {

\{ 

\fix \ v
\assumex v \in \text{dom-}s_{k}'\ v \in \text{dom } ?s_{k}'
\thenx ?s_{k}' \not= \text{None}
\by blast
\hence (?s_{k}' \ v)
\proof (\cases \exists \text{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 \text{op}
where \text{op-in-}\pi_{k} : \text{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})
\by blast
\then consider (v-is-add-effect) \ v \in \text{set } (\text{add-effects-of op})
| (v-is-delete-effect) \ v \in \text{set } (\text{delete-effects-of op})
\by blast
\then show \ ?thesis
\proof (\cases)
\case v-is-add-effect
\thenx ?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 A \ k) ((\Phi^{-1} \Pi A \ t) ! k)
and are-all-operator-effects-consistent ((\Phi^{-1} \Pi 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 A \ k (\Phi^{-1} \Pi A \ t) ! k \ \text{op-in-}\pi_{k}]
\v-is-add-effect \ calculation(2, 3)
\by blast
ultimately show \ ?thesis
\by argo
next
\case v-is-delete-effect
\thenx ?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 \((\Phi S^{-1} \Pi A) t \) ! \(k\)
and are-all-operator-effects-consistent \(((\Phi^{-1} \Pi A) t) ! k\)
using operator-preconditions-hold
by blast+
moreover have \(?t_k' v = Some False\)
using execute-parallel-operator-effect(2) op-in-\(\pi_k\)
v-is-delete-effect calculation(2, 3)
by fast
moreover have \(?t_k' v = Some False\)
by (meson execute-parallel-operator-negative-effect-if op-in-\(\pi_k\)
operator-preconditions-hold v-is-delete-effect)
ultimately show \(?thesis\)
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: \(v \in set (\Pi)_{\forall}\) 
using decode-state-at-valid-variable[OF \(s_k'\)-not-none].
then 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 - 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 lit-semantics A \((State (Suc k) (index ?vs v))^{+}\)
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 lit-semantics A \((State k (index ?vs v))^{+}\)
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 ~lit-semantics A ((State (Suc k) (index ?vs 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 ~lit-semantics A ((State k (index ?vs 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
}
hence {s_k' ⊆ m {t_k'}}
using map-le-def
by blast
}
moreover {
{
fix v
assume v ∈ dom {t_k'}
then have t_k'-not-none: {t_k'} v ≠ None
by blast
{
{
assume contradiction: v ∉ set ((Π)য)
then have (Φ^−1 Π A k) v = None
using decode-state-at-valid-variable
by fastforce
then obtain op
where op-in: op ∈ set ((Φ^−1 Π A t) ! k)
and v-is-or: v ∈ set (add-effects-of op)
∨ v ∈ set (delete-effects-of op)
using execute-parallel-operators-strips-none-if-contraposition[OF
t_k'-not-none]
by blast

241
have \( \text{op-in:} \ \text{op} \in \text{set (}\Pi_{\text{O}}\text{)} \)
using \( \text{op-in decode-plan-step-element-then(1) k-lt-t} \)
by blast

consider \((A) \ v \in \text{set (add-effects-of op)} \)
| \((B) \ v \in \text{set (delete-effects-of op)} \)
using \( v\text{-is-or} \)
by blast

hence False

proof (cases)
  case \(A\)
  then have \( v \in \text{set ((}\Pi_{\text{V}})\text{)} \)
  using \( \text{is-valid-problem-strips-operator-variable-sets(2)[OF assms(1)] op-in A} \)
  by blast
  thus False
  using contradiction
  by blast

next
  case \(B\)
  then have \( v \in \text{set ((}\Pi_{\text{V}})\text{)} \)
  using \( \text{is-valid-problem-strips-operator-variable-sets(3)[OF assms(1)] op-in B} \)
  by blast
  thus False
  using contradiction
  by blast

qed

}\}
hence \( v\text{-in-set-vs:} \ v \in \text{set ((}\Pi_{\text{V}})\text{)} \)
by blast

hence \( ?_{k'} v = ?_{k'} v \)

proof (cases \( \exists \text{op}\in\text{set } ?_{\pi_k}. v \in \text{set (add-effects-of op)} \lor v \in \text{set (delete-effects-of op)} \))

  case True
  then obtain \( \text{op} \)
  where \( \text{op-in-set-}\pi_k: \ \text{op} \in \text{set } ?_{\pi_k} \)
  and \( v\text{-options:} \ v \in \text{set (add-effects-of op)} \lor v \in \text{set (delete-effects-of op)} \)
  by blast
  then have \( \text{op} \in \text{set (}\Pi_{\text{O}}\text{)} \)
  using \( \text{decode-plan-step-element-then}[OF k-lt-t] \)
  by blast
  then consider \((v\text{-is-add-effect}) \ v \in \text{set (add-effects-of op)} \)
  | \((v\text{-is-delete-effect}) \ v \in \text{set (delete-effects-of op)} \)
  using \( v\text{-options} \)
  by blast
  thus \( \?\text{thesis} \)
  proof (cases)
  case \(v\text{-is-add-effect} \)
then have ?t′ v = Some True
  using execute-parallel-operator-positive-effect-if[OF - - op-in-set-π_k]
  operator-preconditions-hold
  by blast
moreover have ?s′ v = Some True
  using encode-problem-parallel-correct-iii[OF assms(1, 2, 3)]

op-in-set-π_k
  v-is-add-effect
  by blast
ultimately show ?thesis
  by argo
next
case v-is-delete-effect
then have ?t′ v = Some False
  using execute-parallel-operator-negative-effect-if[OF - - op-in-set-π_k]
  operator-preconditions-hold
  by blast
moreover have ?s′ v = Some False
  using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)]

op-in-set-π_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-π_k-have-no-effect:
    ∀ op ∈ set ?π_k. v ∉ set (add-effects-of op) ∧ v ∉ 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-π_k-have-no-effect]
  by blast
} note t_k′-equals-s_k = this
{
  have ?s_k v ≠ None
  using t_k′-not-none t_k′-equals-s_k
  by argo

243
then consider \( (s_k \text{-} v \text{-} is \text{-} some \text{-} true) \? s_k \ v = \text{Some True} \)
| \( (s_k \text{-} v \text{-} is \text{-} some \text{-} false) \? s_k \ v = \text{Some False} \)
by fastforce

\}
then show \( \text{thesis} \)
proof (cases)
case \( s_k \text{-} v \text{-} is \text{-} some \text{-} true \)
moreover {
  have lit-semantics \( A ((\text{State } k (\text{index } ?v \text{~} v)))^+ \)
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF 
    \( \text{assms}(1, 2) \ k \text{-} lte \ t \ v \text{-} \text{in} \text{-} \text{set} \text{-} vs \) \( s_k \text{-} v \text{-} is \text{-} some \text{-} true \)
  by simp
  then have lit-semantics \( A ((\text{State } (\text{Suc } k) (\text{index } ?v \text{~} v)))^+ \)
  using state-propagation-positive
  unfolding cnf-semantics-def clause-semantics-def
  by fastforce
  then have \( ?s_k' \ v = \text{Some True} \)
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF 
    \( \text{assms}(1, 2) \ \text{suc} \text{-} k \text{-} lte \ t \ v \text{-} \text{in} \text{-} \text{set} \text{-} vs \)  
  by fastforce

  ultimately show \( \text{thesis} \)
  using \( t_k' \text{-} \text{equals} \text{-} s_k \)
  by simp
next
  case \( s_k \text{-} v \text{-} is \text{-} some \text{-} false \)
  moreover {
    have lit-semantics \( A ((\text{State } k (\text{index } ?v \text{~} v)))^{-1} \)
    using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF 
      \( \text{assms}(1, 2) \ k \text{-} lte \ t \ v \text{-} \text{in} \text{-} \text{set} \text{-} vs \) \( s_k \text{-} v \text{-} is \text{-} some \text{-} false \)
    by simp
    then have lit-semantics \( A ((\text{State } (\text{Suc } k) (\text{index } ?v \text{~} v)))^{-1} \)
    using state-propagation-negative
    unfolding cnf-semantics-def clause-semantics-def
    by fastforce
    then have \( ?s_k' \ v = \text{Some False} \)
    using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF 
      \( \text{assms}(1, 2) \ \text{suc} \text{-} k \text{-} lte \ t \ v \text{-} \text{in} \text{-} \text{set} \text{-} vs \) 
    by fastforce

    ultimately show \( \text{thesis} \)
    using \( t_k' \text{-} \text{equals} \text{-} s_k \)
    by simp
  qed
qede
hence \( ?t_k' \subseteq m \ ?s_k' \)
using map-le-def
by blast
ultimately show ?thesis 
using map-le-antisym 
by blast
qed

lemma encode-problem-parallel-correct-\pi:
assumes is-valid-problem-strips \Pi
and \mathcal{A} \models \Phi \Pi t
and k < length (trace-parallel-plan-strips ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t))
shows trace-parallel-plan-strips ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t) ! k = \Phi_{S^{-1}} \Pi \mathcal{A} k
using assms
proof –
let \mathcal{I} = (\Pi)_I
and \mathcal{\pi} = \Phi^{-1} \Pi \mathcal{A} t
let \mathcal{\tau} = trace-parallel-plan-strips \mathcal{I} \mathcal{\pi}
show ?thesis 
using assms
proof (induction k)
case 0
hence \mathcal{\tau} ! 0 = \mathcal{I}
using trace-parallel-plan-strips-head-is-initial-state
by blast
moreover have \Phi_{S^{-1}} \Pi \mathcal{A} 0 = \mathcal{I}
using decode-state-at-initial-state[OF assms(1, 2)]
by simp
ultimately show ?case
by simp
next
case (Suc k)
let \mathcal{\tau}_k = trace-parallel-plan-strips \mathcal{I} \mathcal{\tau} ! k
and \mathcal{s}_k = \Phi_{S^{-1}} \Pi \mathcal{A} k
have k-lt-length-\tau-minus-one: k < length \mathcal{\tau} - 1 and k-lt-length-\tau: k < length \mathcal{\tau}
using Suc.prems(\tau)
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 \mathcal{\tau} ! k = execute-parallel-plan \mathcal{I} (take k \mathcal{\pi})
using trace-parallel-plan-plan-prefix k-lt-length-\tau
by blast
hence \mathcal{\tau}_k = \mathcal{s}_k
using Suc.IH[OF assms(1, 2) k-lt-length-\tau]
by blast
}
moreover have \( \text{trace-parallel-plan-strips } ?I \ ?\pi ! \text{Suc } k \) 
= execute-parallel-operator \(? ?k \ (?\pi ! k) \)
using \( \text{trace-parallel-plan-step-effect-is}(\text{OF } k\text{-lt-length-}\tau\text{-minus-one}) \)
by blast
moreover {
  thm Suc.prems(3)
  have length (trace-parallel-plan-strips \(?I \ ?\pi\)) \leq length \(?\pi + 1 \)
  using length-trace-parallel-plan-strips-\text{lte-length-plan-plus-one} 
  by blast
then have \( k < \text{length } ?\pi \)
  using Suc.prems(3)
  unfolding Suc-eq-plus1
  by linarith
  hence \( \Phi_{S^{-1}} \Pi.A \text{(Suc } k) \)
  = execute-parallel-operator \(?s_k \ (?\pi ! k) \)
  using encode-problem-parallel-correct-\text{v}[\text{OF } \text{assms}(1, 2)] 
  by simp
}
ultimately show \(?\text{case} \)
by argo
qed

lemma encode-problem-parallel-correct-\text{vii}:
assumes \( \text{is-valid-problem-strips } \Pi \) 
and \( \mathcal{A} \models \Phi \Pi t \)
shows length (map (decode-state-at \( \Pi.A \)) \([0..<\text{Suc} (\text{length } (\Phi^{-1} \Pi.A t))]) = length (\text{trace-parallel-plan-strips }((\Pi)_I) (\Phi^{-1} \Pi.A t)) 
proof –
  let \( ?I = (\Pi)_I \)
  and \( ?\pi = \Phi^{-1} \Pi.A t \)
  let \( ?\sigma = \text{map (decode-state-at } \Pi.A) \[0..<\text{Suc} (\text{length } ?\pi)\] 
  and \( ?\tau = \text{trace-parallel-plan-strips } ?I \ ?\pi \)
  let \( ?l = \text{length } ?\tau \)
  let \( ?k = ?l - 1 \)
  show \(?\text{thesis} \)
proof (rule ccontr)
  assume \( \text{length-}\sigma\text{-eq-length-}\tau:\text{ length } ?\sigma \neq \text{length } ?\tau \)
  
  have length \( ?\sigma = \text{length } ?\pi + 1 \)
  by fastforce
  moreover have length \( ?\tau \leq \text{length } ?\pi + 1 \)
  using length-trace-parallel-plan-strips-\text{lte-length-plan-plus-one} 
  by blast
  moreover have length \( ?\tau < \text{length } ?\pi + 1 \)
  using length-\sigma\text{-eq-length-\tau-calculation} 
  by linarith
} note \( nb_1 = \text{this} \)
\begin{verbatim}
{ have 0 < length ?τ using trace-parallel-plan-strips-not-nil. then have length ?τ - 1 < length ?π using nb₁ by linarith } note nb₂ = this { obtain k' where length ?τ = Suc k' using less-imp-Suc-add[OF length-trace-parallel-plan-gt-0] by blast hence ?k < length ?π using nb₂ by blast } note nb₃ = this { have ?τ ! ?k = execute-parallel-plan ?I (take ?k ?π) using trace-parallel-plan-plan-prefix[of ?k] length-trace-minus-one-lt-length-trace by blast thm encode-problem-parallel-correct-vi[OF assms(1, 2)] nb₃ moreover have (Φₛ⁻¹ Π A ?k) = ?τ ! ?k using encode-problem-parallel-correct-vi[OF assms(1, 2)] length-trace-minus-one-lt-length-trace by blast ultimately have (Φₛ⁻¹ Π A ?k) = execute-parallel-plan ?I (take ?k ?π) by argo } note nb₄ = this { have are-all-operators-applicable (Φₛ⁻¹ Π A ?k) (?π ! ?k) and are-all-operator-effects-consistent (?π ! ?k) using encode-problem-parallel-correct-ii[OF assms(1, 2)] nb₃ nb₄ by blast+ — Unsure why calculation(1, 2) is needed for this proof step. Should just require the default proof. moreover have ¬are-all-operators-applicable (Φₛ⁻¹ Π A ?k) (?π ! ?k) and ¬are-all-operator-effects-consistent (?π ! ?k) using length-trace-parallel-plan-strips-lt-length-plan-plus-one-then[OF nb₁] calculation(1, 2) unfolding nb₃ nb₄ by blast+ ultimately have False by blast } thus False. qed qed

lemma encode-problem-parallel-correct-x: assumes is-valid-problem-strips Π
\end{verbatim}
and $A \vDash \Phi \Pi t$

shows $\map{\text{map}}{\text{decode-state-at} \Pi A}{[0..\text{Suc} (\text{length} (\Phi^{-1} \Pi A t))] = \text{trace-parallel-plan-strips} ((\Pi)_I) (\Phi^{-1} \Pi A t)}$

proof

let $?I = (\Pi)_I$

and $?\pi = \Phi^{-1} \Pi A t$

let $?\sigma = \map{\text{map}}{\text{decode-state-at} \Pi A}{[0..\text{Suc} (\text{length} ?\pi)]}$

and $?\tau = \text{trace-parallel-plan-strips} ?I ?\pi$

\{ have length $?\tau = \text{length} ?\sigma$

using encode-problem-parallel-correct-vii[OF assms].. moreover \{

fix $k$

assume $k$-lt-length-$\tau$: $k < \text{length} ?\tau$

then have $\text{trace-parallel-plan-strips} ((\Pi)_I) (\Phi^{-1} \Pi A t) ! k = \Phi S^{-1} \Pi A k$

using encode-problem-parallel-correct-vi[OF assms] by blast

moreover \{

have length $?\tau \leq \text{length} ?\pi + 1$

using length-trace-parallel-plan-strips-lte-length-plan-plus-one by blast

then have $k < \text{length} ?\pi + 1$

using k-ll-length-$\tau$

by linarith

then have $k < \text{Suc} (\text{length} ?\pi) - 0$

by simp

hence $?\sigma ! k = \Phi S^{-1} \Pi A k$

using nth-map-upt[of $k \text{Suc} (\text{length} ?\pi) \emptyset]$

by auto

\}

ultimately have $?\tau ! k = ?\sigma ! k$

by argo

\}

ultimately have $?\tau = ?\sigma$

using list-eq-iff-nth-eq[of $?\tau \ ?\sigma]$

by blast

\}

thus $?\text{thesis}$

by argo

qed

lemma encode-problem-parallel-correct-xi:

fixes II: ‘a strips-problem

assumes is-valid-problem-strips II

and $A \vDash \Phi \Pi t$

and $\text{ops} \in \text{set} (\Phi^{-1} \Pi A t)$

and $\text{op} \in \text{set} \text{ops}$

248
shows $op \in \text{set } ((\Pi)_O)$

proof

- let $?\pi = \Phi^{-1} \Pi A t$
- have length $?\pi = t$
  - using decode-plan-length
  - by force

moreover obtain $k$ where $k < \text{length } ?\pi$ and $\text{ops} = ?\pi \uparrow k$
  - using in-set-conv-nth[of $\text{ops} \ ?\pi$] 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 $A$ of the basic SATPlan encoding $\Phi \Pi t$ for a given valid problem $\Pi$ and hypothesized plan length $t$, the decoded plan $\pi \equiv \Phi^{-1} \Pi A t$ 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

$$\sigma = \text{map } (\lambda k. \Phi_{S^{-1}} \Pi A k) [0..<\text{Suc } (\text{length } ?\pi)]$$

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 $\text{ops} \in \text{set } \pi$ 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 $\text{is-valid-problem-strips } \Pi$
  and $A \models \Phi \Pi t$
  shows $\text{is-parallel-solution-for-problem } \Pi (\Phi^{-1} \Pi A t)$

proof

\footnote{This lemma is used in the proof but not shown.}
let ?ops = strips-problem.operators-of Π
and ?I = (Π)_I
and ?G = (Π)_G
and ?π = Φ⁻¹ Π A t
let ?σ = map (λ k. Φ⁻¹ Π A k) [0..<Suc (length ?π)]
{
  have ?σ = ?τ
    using encode-problem-parallel-correct-x[OF assms].
  moreover {
    have length ?π = τ
      using decode-plan-length
      by auto
    then have ?G ⊆ m last ?σ
      using decode-state-at-goal-state[OF assms]
      by simp
  }
  ultimately have ((Π)_G) ⊆ m execute-parallel-plan ((Π)_I) (Φ⁻¹ Π A t)
    using execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace
    by auto
  }
  moreover have ∀ ops ∈ set ?π. ∀ op ∈ set ops. op ∈ set ((Π)_O)
    using encode-problem-parallel-correct-xi[OF assms(1, 2)]
    by auto
  ultimately show ?thesis
    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 \( A_2 \)).

**definition valuation-for-state-variables**

:: `variable strips-problem
\Rightarrow `variable strips-operator list list
\Rightarrow `variable strips-state list
\Rightarrow sat-plan-variable valuation

**where valuation-for-state-variables** \( \Pi \pi \tau \equiv let \)

\[ t' = \text{length } \tau \]
\[ \tau_\Omega = \tau ! (t' - 1) \]
\[ vs = \text{variables-of } \Pi \]
\[ V_1 = \{ \text{State } k \ (\text{index } vs v) \mid k v. k \in \{0..<t'\} \land v \in \text{set } vs \} \]
\[ V_2 = \{ \text{State } k \ (\text{index } vs v) \mid k v. k \in \{t'..(\text{length } \pi + 1)\} \land v \in \text{set } vs \} \]
\[ A_1 = \text{foldr} \]
\[ (\lambda(k, v) A. \text{valuation-for-state } (\text{variables-of } \Pi) (\tau ! k) k v A) \]
\[ (\text{List.product } [0..<t'] vs) \]
\[ A_0 \]
\[ A_2 = \text{foldr} \]
\[ (\lambda(k, v) A. \text{valuation-for-state } (\text{variables-of } \Pi) \tau_\Omega k v A) \]
\[ (\text{List.product } [t'..<\text{length } \pi + 2] vs) \]
\[ A_0 \]

in override-on (override-on \( A_0 A_1 V_1 \) \( A_2 V_2 \))

— The valuation is left to yield false for the potentially remaining \( k' \in \{ \text{length } \tau..\text{length } \pi + 1 \} \) since no more operators are executed after the trace ends anyway. The definition of \( A_0 \) as the valuation that is false for every argument ensures this implicitly.

**definition valuation-for-operator-variables**

:: `variable strips-problem
\Rightarrow `variable strips-operator list list
\Rightarrow `variable strips-state list
\Rightarrow sat-plan-variable valuation

**where valuation-for-operator-variables** \( \Pi \pi \tau \equiv let \)

\( \text{ops = operators-of } \Pi \)
\[ \text{Ops} = \{ \text{Operator } k \ (\text{index } \text{ops } op) \mid k \text{ op. } k \in \{0..<\text{length } \tau - 1\} \land \text{op } \in \text{set } \text{ops} \} \]

\( \text{ops} \)

in override-on

\( A_0 \)

\( (\text{foldr}) \)
\[ (\lambda(k, op) A. A(\text{Operator } k \ (\text{index } \text{ops } op) := True)) \]
\( (\text{concat } (\text{map } (\lambda k. \text{map } (\text{Pair } k) (\pi ! k)) [0..<\text{length } \tau - 1]))) \]
\( A_0 \)
\( \text{Ops} \)

The completeness proof requires that we show that the SATPlan encoding \( \Phi \Pi t \) of a problem \( \Pi \) has a model \( A \) in case a solution \( \pi \) with length \( t \) exists. Since a plan corresponds to a state trace \( \tau \equiv \text{trace-parallel-plan-strips } I \pi \) with

251
\( \tau ! k = \text{execute-parallel-plan } I \) (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 \( v \in \mathcal{V} \) 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 \( op \in \mathcal{O} \) 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

:: 'variable strips-problem

\[ \Rightarrow ' \text{variable strips-operator list list} \]

\[ \Rightarrow \text{sat-plan-variable valuation} \]

**where** valuation-for-plan \( \Pi \pi \equiv \text{let} \)

\[ vs = \text{variables-of } \Pi \]

\[ ; ops = \text{operators-of } \Pi \]

\[ ; \tau = \text{trace-parallel-plan-strips } (\text{initial-of } \Pi) \pi \]

\[ ; t = \text{length } \pi \]

\[ ; t' = \text{length } \tau \]

\[ ; \mathcal{A}_V = \text{valuation-for-state-variables } \Pi \pi \tau \]

\[ ; \mathcal{A}_O = \text{valuation-for-operator-variables } \Pi \pi \tau \]

\[ ; \mathcal{V} = \{ \text{State } k \ (\text{index } vs \ v) \mid k v, k \in \{0..<t + 1\} \land v \in \text{set } vs \} \]

\[ ; \mathcal{O} = \{ \text{Operator } k \ (\text{index } ops \ op) \mid k op, k \in \{0..<t\} \land op \in \text{set } ops \} \]

\[ \text{in override-on} (\text{override-on } \mathcal{A}_0 \mathcal{A}_V \mathcal{V}) \mathcal{A}_O \mathcal{O} \]

— 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.

**lemma** model-of-encode-problem-makespan-zero-iff:

\[ \mathcal{A} \models \Phi \Pi \theta \leftrightarrow \mathcal{A} \models \Phi_I \Pi \land (\Phi_G \Pi) \theta \]

**proof**

\[ \text{have encode-operators } \Pi \theta = \neg \bot \land \neg \bot \]

**unfolding** encode-operators-def encode-all-operator-effects-def encode-all-operator-preconditions-def

**by** simp

**moreover have** encode-all-frame-axioms \( \theta = \neg \bot \)

**unfolding** encode-all-frame-axioms-def

\(^{11}\)It is helpful to remember at this point, that the trace elements of a solution contain the states reached by plan prefix execution (lemma ??).
by simp
ultimately show ?thesis
unfolding encode-problem-def SAT-Plan-Base.encode-problem-def encode-initial-state-def
   encode-goal-state-def
by simp
qed

lemma empty-valuation-is-False[simp]: \( \mathcal{A}_0 \ v = \text{False} \)
unfolding empty-valuation-def.

lemma model-initial-state-set-valuations:
assumes is-valid-problem-strips \( \Pi \)
shows set (map (\( \lambda v. \text{case } ((\Pi)_I) v \text{ of } \text{Some } b \)) \( v \)) := b)
   \[ \Rightarrow \mathcal{A}_0(\text{State 0 } (\text{index } (\text{strips-problem.variables-of } \Pi) v) := \text{b}) \]
   \[ | - \Rightarrow \mathcal{A}_0 \]
   \( (\text{strips-problem.variables-of } \Pi)) \)
   \[ = \{ \mathcal{A}_0(\text{State 0 } (\text{index } (\text{strips-problem.variables-of } \Pi) v) := \text{the } ((\Pi)_I) v)) \]
   \[ | v. v \in \text{set } (((\Pi)_I)) \} \]
proof -
let \( ?I = ((\Pi)_I) \)
and \( ?\text{vs } = \text{strips-problem.variables-of } \Pi \)
let \( ?f = \lambda v. \text{case } ((\Pi)_I) v \text{ of } \text{Some } b \)
\[ \Rightarrow \mathcal{A}_0(\text{State 0 } (\text{index } ?\text{vs } v) := b) \] \[ | - \Rightarrow \mathcal{A}_0 \]
and \( ?g = \lambda v. \mathcal{A}_0(\text{State 0 } (\text{index } ?\text{vs } v) := \text{the } ((\Pi)_I) v) \)
let \( ?\text{As } = \text{map } ?f \ ?\text{vs} \)
have \( \text{nb}_1: \text{dom } ?I = \text{set } (((\Pi)_I)) \)
using is-valid-problem-strips-initial-of-dom assms
by fastforce
\{ 
  \{ 
    fix v 
    assume v \in \text{dom } ?I 
    hence \( ?f v = ?g v \)
    using \( \text{nb}_1 \)
    by fastforce 
  \}
  hence \( ?f \ ?\text{set } (((\Pi)_I)) = ?g \ ?\text{set } (((\Pi)_I)) \)
  using \( \text{nb}_1 \)
  by force 
\}
then have set ?\text{As } = ?g \ ?\text{set } (((\Pi)_I))
  unfolding set-map
  by simp
thus ?thesis
  by blast
qed

253
lemma valuation-of-state-variable-implies-lit-semantics-if:
  assumes v ∈ dom S
  and A (State k (index vs v)) = the (S v)
  shows lit-semantics A (literal-formula-to-literal (encode-state-variable k (index vs v) (S v)))
proof –
  let ?L = literal-formula-to-literal (encode-state-variable k (index vs v) (S v))
  consider (True) S v = Some True
  | (False) S v = Some False
  using assms(1)
  by fastforce
  thus ?thesis
  unfolding encode-state-variable-def
  using assms(2)
  by (cases, force+) 
qed

lemma foldr-fun-upd:
  assumes inj-on f (set xs)
  and x ∈ set xs
  shows foldr (λx A. A(f x := g x)) 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
  qed
by blast

moreover {  
— Follows from modus tollens on the definition of inj-on.

have \( f a \neq f x \)
  using Cons.prems False
  by force
moreover have \( \text{foldr} (\lambda x. A(f x := g x)) (a \# xs) A \)
  = (\text{foldr} (\lambda x. A(f x := g x)) xs A)(f a := g a) 
  by simp
ultimately have \( \text{foldr} (\lambda x. A(f x := g x)) (a \# xs) A (f x) \)
  = (\text{foldr} (\lambda x. A(f x := g x)) xs A) (f x)
  unfolding fun-upd-def
  by presburger
} ultimately show \(?thesis
  by argo
qed simp

lemma foldr-fun-no-upd:
  assumes \( \text{inj-on \( f \)} \ (\text{set \( xs \)}) 
  \text{and} y \notin f' \text{ set \( xs \)}
  shows \( \text{foldr} (\lambda x. A(f x := g x)) xs A y = A y \)
  using assms
proof (induction \( xs \))
  case (Cons a xs)
  {  
    have \( \text{inj-on \( f \)} \ (\text{set \( xs \)}) \text{ and} y \notin f' \text{ set \( xs \)}
      using Cons.prems
      by (fastforce, simp)
    hence \( \text{foldr} (\lambda x. A(f x := g x)) xs A y = A y \)
      using Cons.IH
      by blast
  }
moreover {  
  have \( f a \neq y \)
    using Cons.prems(2)
    by auto
moreover have \( \text{foldr} (\lambda x. A(f x := g x)) (a \# xs) A \)
  = (\text{foldr} (\lambda x. A(f x := g x)) xs A)(f a := g a) 
  by simp
ultimately have \( \text{foldr} (\lambda x. A(f x := g x)) (a \# xs) A y \)
  = (\text{foldr} (\lambda x. A(f x := g x)) xs A) y
  unfolding fun-upd-def
  by presburger
}
ultimately show \(?case
  by argo
qed simp

255
— We only use the part of the characterization of \( \mathcal{A} \) which pertains to the state variables here.

**lemma** encode-problem-parallel-complete-i:

**fixes** II::'a strips-problem

**assumes** is-valid-problem-strips II

\[ (\Pi)_{II} \subseteq_{m} \text{execute-parallel-plan } ((\Pi)_{I}) \pi \]

\[ \forall v, k \text{. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_{I}) \pi) \]

\[ \implies (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)) \]

\[ \iff (\text{trace-parallel-plan-strips } ((\Pi)_{I}) \pi ! k) v = \text{Some True} \]

\[ \wedge (\neg \mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)) \]

\[ \iff ((\text{trace-parallel-plan-strips } ((\Pi)_{I}) \pi ! k) v \neq \text{Some True}) \]

**shows** \( \mathcal{A} \models \Phi_{I} \Pi \)

**proof**

- let \( ?vs = \text{strips-problem.variables-of } \Pi \)
- and \( ?I = (\Pi)_{I} \)
- and \( ?G = (\Pi)_{G} \)
- and \( \Phi_{I} = \Phi_{I} \Pi \)

- let \( ?\tau = \text{trace-parallel-plan-strips } ?I \pi \)

\{ 

- fix \( C \)
  - assume \( C \in \text{cnf } \Phi_{I} \)
  - then obtain \( v \)
  - where \( v\text{-in-set-vs: } v \in \text{set } ?vs \)
  - and \( C\text{-is: } C = \{ \text{literal-formula-to-literal } (\text{encode-state-variable } 0 (\text{index } ?vs v)) (\text{?I } v) \} \)
  - using \( \text{cnf-of-encode-initial-state-set-ii[OF assms(I)]} \)
  - by \text{auto}

\{ 

- have \( 0 < \text{length } ?\tau \)
  - using \text{trace-parallel-plan-strips-not-nil}
  - by \text{blast}
  - then have \( \mathcal{A} (\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v)) \)
  - \( \iff (\text{trace-parallel-plan-strips } ((\Pi)_{I}) \pi ! 0) v = \text{Some True} \)
  - and \( \neg \mathcal{A} (\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v)) \)
  - \( \iff ((\text{trace-parallel-plan-strips } ((\Pi)_{I}) \pi ! 0) v \neq \text{Some True}) \)
  - using \text{assms(3)}
  - by \text{(presburger+)}

- note \( \text{nb = this} \)

\{ 

- let \( ?L = \text{literal-formula-to-literal } (\text{encode-state-variable } 0 (\text{index } ?vs v)) (\text{?I v}) \)

- have \( \tau\text{-0-is: } ?\tau ! 0 = ?I \)
  - using \text{trace-parallel-plan-strips-head-is-initial-state}
  - by \text{blast}

- have \( v\text{-in-dom-}\text{I}: v \in \text{dom } ?I \)
  - using \( \text{is-valid-problem-strips-initial-of-dom assms(I)} \text{ v-in-set-vs} \)
  - by \text{fastforce}

- then consider \( (I\text{-v-is-Some-True} ) ?I v = \text{Some True} \)
| (1-v-is-Some-False) ?I v = Some False by fastforce hence lit-semantics A ?L unfolding encode-state-variable-def using assms(3) τ-0-is nb by (cases, force+) } hence clause-semantics A C unfolding clause-semantics-def C-is by blast } thus ?thesis using is-cnf-encode-initial-state[OF assms(1)] is-nnf-cnf cnf-semantics unfolding cnf-semantics-def by blast 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 A Otherwise, we have to first show that the goal was reached (albeit early) and that our valuation 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) \( G \subseteq_m \) execute-parallel-plan ((Π)I) π and \( \forall v. k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)I) \pi) \rightarrow (A (\text{State } k (\text{index} (\text{strips-problem.variables-of} \Pi) v)) \leftrightarrow (\text{trace-parallel-plan-strips} ((\Pi)I) \pi ! k) v = \text{Some True}) \) and \( \forall v. l \geq \text{length} (\text{trace-parallel-plan-strips} ((\Pi)I) \pi) \land l < \text{length} \pi + 1 \rightarrow A (\text{State } l (\text{index} (\text{strips-problem.variables-of} \Pi) v)) = A (\text{State } (\text{length} (\text{trace-parallel-plan-strips} ((\Pi)I) \pi) - 1) (\text{index} (\text{strips-problem.variables-of} \Pi) v)) \) shows A \( \models (\Phi_G \Pi)(\text{length } \pi) \)

**proof** —
let \(?us = \text{strips-problem.variables-of} \Pi \)
and \(?I = (\Pi)_I\) and \(?G = (\Pi)_G\) and \(?\Phi_I = \Phi_I \Pi\) and \(?\pi = \text{length } \pi\) and \(?\Phi_G = (\Phi_G \Pi) (\text{length } \pi)\) let \(?\tau = \text{trace-parallel-plan-strips } \Pi I\)
let \(?\tau' = \text{length } ?\tau\)
{ fix v assume G-of-v-is-not-None: ?G v \( \neq \) None have ?G \( \subseteq_m \) last ?\tau
using `execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace` assms (2) by `blast` also have ... = ?τ ! (?τ' - 1) using `last-conv-nth[OF trace-parallel-plan-strips-not-nil]`, finally have ?G ⊆m ?τ ! (?τ' - 1) by `argo` hence (?τ ! (?τ' - 1)) v = ?G v using `G-of-v-is-not-None` unfolding `map-le-def` by `force` }

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)
case `True` moreover have ?t < ?t' using `calculation` by `fastforce` moreover have `A (State ?t (index ?vs v)) ⇔ (?τ ! ?t) v = Some True` using `assms(3)` calculation(2) by `blast` ultimately show `?thesis` using `nb1[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-neg-implies-less by `blast` moreover have `A (State ?t (index ?vs v)) = 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 ?τ] less-diff-conv2[of 1 ?t' ?t']` by `force` moreover have `A (State (?t' - 1) (index ?vs v)) = Some True` using `assms(3)` calculation(3) by `blast` }
ultimately have \( A (\text{State } \tau t (\text{index } \tau v s v)) \leftrightarrow (?r ! (?t' - 1)) \) \( v = \text{Some True} \)

by blast

\}

thus \(?thesis \)
using \( nb_1[\text{OF G-of-v-is-not-None}] \)
by presburger

qed

\}

note \( nb_2 = \text{this} \)

\{

fix \( C \)
assume \( \text{C-in-cnfs-of-} \Phi_G \): \( C \in \text{cnf } ?\Phi_G \)

moreover obtain \( v \)
where \( v \in \text{set } ?v s \)
and \( G\text{-of-v-is-not-None}: \text{G v} \neq \text{None} \)
and \( C\text{-is: } C = \{ \text{literal-formula-to-literal (encode-state-variable } \tau t (\text{index } \tau v s) v) \} \)
using \( \text{cnf-of-encode-goal-state-set-ii}\)[OF assms(1)]
calculation
by auto

consider \( G\text{-of-v-is-Some-True} \): \( \text{G v} = \text{Some True} \)
| \( G\text{-of-v-is-Some-False} \): \( \text{G v} = \text{Some False} \)
using \( G\text{-of-v-is-not-None} \)
by fastforce

then have \( \text{clause-semantics } A C \)
using \( nb_2 C\text{-is} \)
unfolding \( \text{clause-semantics-def encode-state-variable-def} \)
by (cases, force+)

\}

thus \(?thesis \)
using \( \text{cnf-semantics}[\text{OF is-nnf-cnfs}[\text{OF encode-goal-state-is-cnfs}[\text{OF assms(1)]]}] \)
unfolding \( \text{cnf-semantics-def} \)
by blast

qed

— We are not using the full characterization of \( A \) here since it’s not needed.

lemma \( \text{encode-problem-parallel-complete-iii-a}: \)

fixes \( \Pi::'a \text{strips-problem} \)
assumes \( \text{is-valid-problem-strips } \Pi \)
and \( (\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi \)
and \( C \in \text{cnf } \langle \text{encode-all-operator-preconditions } \Pi (\text{strips-problem.operators-of II} (\text{length } \pi)) \rangle \)
and \( \forall k \text{ op. } k < \text{length (trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \rightarrow A (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } II) \text{ op})) = (\text{op } \in \text{set } (\pi ! k)) \)
and \( \forall l \text{ op. } l \geq \text{length (trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \land l < \text{length } \pi \rightarrow \neg A (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } II) \text{ op})) \)
and \( \forall v \ k . \ k < \text{length} (\text{trace-parallel-plan-strips} ((\Pi)_1) \pi) \)
\[ \rightarrow (A (\text{State} k (\text{index} (\text{strips-problem.variables-of} \ Pi) v))) \]
\[ \leftarrow (\text{trace-parallel-plan-strips} ((\Pi)_1) \pi ! k) \ v = \text{Some True} \]

shows clause-semantics \( A \ C \)

proof –

let \( \text{ops} = \text{strips-problem.operators-of} \ Pi \)
and \( \text{vs} = \text{strips-problem.variables-of} \ Pi \)
and \( \ell = \text{length} \pi \)

let \( \tau = \text{trace-parallel-plan-strips} ((\Pi)_1) \pi \)

obtain \( k \ op \)
where \( k-\text{and-op-are:} (k, op) \in (\{0..<\ell\} \times \text{set} ((\Pi)_C)) \)
and \( C \in (\bigcup v \in \text{set} (\text{precondition-of op}). \{(\text{Operator} k (\text{index} \text{ops op}))^{-1}
, (\text{State} k (\text{index} \text{vs v}))^+ \})\}) \)

using cnf-of-encode-all-operator-preconditions-structure assms(3)

UN-E\[A C\]

by auto

then obtain \( v \)
where \( v-\text{in-preconditions-of-op:} v \in \text{set} (\text{precondition-of op}) \)
and \( \text{C-is:} C = \{(\text{Operator} k (\text{index} \text{ops op}))^{-1}, (\text{State} k (\text{index} \text{vs v}))^+ \}\}

by blast

thus \( ?\text{thesis} \)

proof (cases \( k < \text{length} \tau - 1 \))

case \( k-\text{lt-length-}\tau-\text{-minus-one:} \text{True} \)

thus \( ?\text{thesis} \)

proof (cases \( \text{op} \in \text{set} (\pi ! k) \))

case \( \text{True} \)

\{ 

have are-all-operators-applicable (\( \tau ! k \)) (\( \pi ! k \))
using trace-parallel-plan-strips-operator-preconditions k-lt-length-\( \tau \)-minus-one
by blast
then have (\( \tau ! k \)) \( v = \text{Some True} \)
using are-all-operators-applicable-set v-in-preconditions-of-op True
by fast
hence \( A (\text{State} k (\text{index} \text{vs v})) \)
using assms(6) k-lt-length-\( \tau \)-minus-one
by force
\}

thus \( ?\text{thesis} \)

using C-is

unfolding clause-semantics-def
by fastforce

next

case \( \text{False} \)
then have \( \neg A (\text{Operator} k (\text{index} \text{ops op})) \)
using assms(4) k-lt-length-\( \tau \)-minus-one
by blast

thus \( ?\text{thesis} \)

using C-is

260
unfolding clause-semantics-def
by fastforce
qed
next
case False
then have \( k \geq \text{length } \tau - 1 \)
using k-and-op-are
by (force, simp)
then have \( \neg \mathcal{A} (\text{Operator } k (\text{index } ?ops op)) \)
using assms(5)
by blast
thus \(?thesis\)
unfolding clause-semantics-def
using C-is
by fastforce
qed
qed

— We are not using the full characterization of \( \mathcal{A} \) here since it's not needed.

lemma encode-problem-parallel-complete-iii-b:
fixes \( \Pi :: \text{strips-problem} \)
assumes is-valid-problem-strips \( \Pi \)
and \( (\Pi) G \subseteq m \) execute-parallel-plan \((\Pi) I \) \( \pi \)
and \( C \in \text{cnf} \ (\text{encode-all-operator-effects } \Pi \ (\text{strips-problem.operators-of } \Pi \) (\text{length } \pi)) \)
and \( \forall k \ op. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi) I) \pi) - 1 \)
\( \rightarrow \mathcal{A} (\text{Operator } k (\text{index } \text{strips-problem.operators-of } \Pi \) op)) = (op \in \text{set } (\pi ! k)) \)
and \( \forall l \ op. l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi) I) \pi) - 1 \land l < \text{length } \pi \)
\( \rightarrow \neg \mathcal{A} (\text{Operator } l (\text{index } \text{strips-problem.operators-of } \Pi \) op)) \)
and \( \forall v k. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi) I) \pi) \)
\( \rightarrow (\mathcal{A} (\text{State } k (\text{index } \text{strips-problem.variables-of } \Pi \) v)) \leftrightarrow (\text{trace-parallel-plan-strips } ((\Pi) I) \pi ! k) v = \text{Some } True) \)
shows clause-semantics \( \mathcal{A} C \)
proof —
let \( ?ops = \text{strips-problem.operators-of } \Pi \)
and \( ?vs = \text{strips-problem.variables-of } \Pi \)
and \( ?t = \text{length } \pi \)
let \( ?A = (\bigcup (k, op) \in [0..<?t] \times \text{set } ((\Pi)_{\Box}). \cup v \in \text{set } (\text{add-effects-of op}). \{ (\text{Operator } k (\text{index } ?ops op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^{+} \}) \)
and \( ?B = (\bigcup (k, op) \in [0..<?t] \times \text{set } ((\Pi)_{\Box}). \cup v \in \text{set } (\text{delete-effects-of op}). \{ (\text{Operator } k (\text{index } ?ops op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^{-1} \}) \)
consider (C-in-A) \( C \in ?A \)
| (C-in-B) \( C \in ?B \)
using Un-iff[of \( ?A \) \( ?B \)] cnf-of-encode-all-operator-effects-structure assms(3)
by (metis C-in-A C-in-B)
thus ?thesis
proof (cases)
case C-in-A
then obtain $k \cdot op$
  where $k-and-op-are$:
    $(k, op) \in \{0 .. ?t \times \text{set}((\Pi)_{\mathcal{O}})\)$
    and $C \in (\bigcup v \in \text{set} \cdot (\text{add-effects-of} \cdot op))$
    $\{\{ (\text{Operator } k \cdot (\text{index } ?\text{ops} \cdot op))^{-1}, (\text{State } (\text{Suc } k) \cdot (\text{index } ?\text{vs} \cdot v)) \}}\}
  by blast
then obtain $v$ where $v-in-add-effects-of-op$:
  $v \in \text{set} \cdot \text{add-effects-of} \cdot op$
  and $C-is$:
    $C = \{ (\text{Operator } k \cdot (\text{index } ?\text{ops} \cdot op))^{-1}, (\text{State } (\text{Suc } k) \cdot (\text{index } ?\text{vs} \cdot v)) \}$
  by blast
thus ?thesis
proof (cases $k < \text{length } ?\tau - 1$
  case $k-lt-length-\tau-minus-one$:
    True
    thus ?thesis
    proof (cases $op \in \text{set} \cdot (\pi ! k)$)
      case True
      { then have are-all-operators-applicable $\cdot (\pi ! k)$
        and are-all-operator-effects-consistent $\cdot (\pi ! k)$
        using trace-parallel-plan-strips-operator-preconditions $k-lt-length-\tau-minus-one$
        by blast+
        hence execute-parallel-operator $\cdot (\pi ! k)$ $\cdot v = \text{Some } True$
        using execute-parallel-operator-positive-effect-if
          $\cdot [OF \cdot - \cdot \text{True } v-in-add-effects-of-op, of \cdot ?\tau \cdot k]$
        by blast
      }
    } then have $\tau-Suc-k-is-Some-True$: $\cdot (\tau ! \text{Suc } k)$ $\cdot v = \text{Some } True$
    using trace-parallel-plan-step-effect-is $\cdot [OF \cdot k-lt-length-\tau-minus-one]$
    by argo
    have $A$ $\cdot (\text{State } (\text{Suc } k) \cdot (\text{index } ?\text{vs} \cdot v))$
    using assms(6) $k-lt-length-\tau-minus-one$ $\cdot \tau-Suc-k-is-Some-True$
    by fastforce
    thus ?thesis
    using $C-is$
    unfolding clause-semantics-def
    by fastforce
next
case False
then have $\neg A$ $\cdot (\text{Operator } k \cdot (\text{index } ?\text{ops} \cdot op))$
    using assms(4) $k-lt-length-\tau-minus-one$
    by blast
thus ?thesis
using $C-is$
unfolding clause-semantics-def
by force
qed

262
next
case False
then have \( k \geq \text{length } ?\tau - 1 \text{ and } k < ?t \)
  using k-and-op-are
  by auto
then have \( \neg A \ (\text{Operator } k \ (\text{index } ?\text{ops op})) \)
  using assms(5)
  by blast
thus \( ?\text{thesis} \)
  using C-is
  unfolding clause-semantics-def
  by fastforce
qed

next
— This case is completely symmetrical to the one above.
case C-in-B
then obtain \( k \) \( \text{op} \)
where k-and-op-are: \( (k, \text{op}) \in \{0..<?t\} \times \text{set } (\Pi) \)\( \)and \( C \in (\bigcup \{v \in \text{set } (\text{delete-effects-of } \text{op}) \})\)

\{ \( (\text{Operator } k \ (\text{index } ?\text{ops op}))^{-1}, (\text{State } (\text{Suc } k) \ (\text{index } ?\text{vs } v))^{-1} \}) \}
by blast
then obtain \( v \) \( \text{where } v\text{-in-delete-effects-of-op: } v \in \text{set } (\text{delete-effects-of } \text{op}) \)
and \( C\text{-is: } C = \{ (\text{Operator } k \ (\text{index } ?\text{ops op}))^{-1}, (\text{State } (\text{Suc } k) \ (\text{index } ?\text{vs } v))^{-1} \} \)
by blast
thus \( ?\text{thesis} \)
proof (cases \( k < \text{length } ?\tau - 1 \))
case k-lt-length-\( \tau\)-minus-one: True
thus \( ?\text{thesis} \)
proof (cases \( \text{op} \in \text{set } (\pi ! k) \))
case True
{\n  then have \( \text{are-all-operators-applicable } (?\tau ! k) \ (\pi ! k) \)
  and \( \text{are-all-operator-effects-consistent } (\pi ! k) \)
  using trace-parallel-plan-strips-operator-preconditions k-lt-length-\( \tau\)-minus-one
  by blast
  hence \( \text{execute-parallel-operator } (?\tau ! k) \ (\pi ! k) \ v = \text{Some False } \)
  using \( \text{execute-parallel-operator-negative-effect-if} \)
  \( \text{OF - - True } v\text{-in-delete-effects-of-op, of } ?\tau ! k \)
  by blast
}\then have \( \tau\text{-Suc-k-is-Some-True: } (?\tau ! \text{Suc } k) \ v = \text{Some False } \)
  using trace-parallel-plan-step-effect-is[OF k-lt-length-\( \tau\)-minus-one]
  by argo
have \( \neg A \ (\text{State } (\text{Suc } k) \ (\text{index } ?\text{vs } v)) \)
  using assms(6) \( k\text{-lt-length-}\( \tau\)-minus-one \( \tau\text{-Suc-k-is-Some-True } \)
  by fastforce
thus \( ?\text{thesis} \)
  using C-is

263
unfolding clause-semantics-def
by fastforce
next
case False
then have \( \neg A \ (\text{Operator } k \ (\text{index } ?\text{ops } op)) \)
using assms(4) k-it-length-\( \tau \)-minus-one
by blast
thus ?thesis
using C-is
unfolding clause-semantics-def
by force
qed
next
case False
then have \( k \geq \text{length } ?\tau - 1 \) and \( k < ?t \)
using k-and-op-are
by auto
then have \( \neg A \ (\text{Operator } k \ (\text{index } ?\text{ops } op)) \)
using assms(5)
by blast
thus ?thesis
using C-is
unfolding clause-semantics-def
by fastforce
qed
qed

lemma encode-problem-parallel-complete-iii:
fixes \( \Pi \) :: 'a strips-problem
assumes is-valid-problem-strips \( \Pi \)
and \( (\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi \)
and \( \forall k \text{ op. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \)
\(-\rightarrow A \ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op } \in \text{set } (\pi ! k)) \)
and \( \forall l \text{ op. } l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \land l < \text{length } \pi \)
\(-\rightarrow \neg A \ (\text{Operator } l \ (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op})) \)
and \( \forall v k. \ k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \)
\(-\rightarrow (A \ (\text{State } k \ (\text{index } (\text{strips-problem.variables-of } \Pi) \text{ v}))), v = \text{Some True} \)
shows \( A \models \text{encode-operators } \Pi \ (\text{length } \pi) \)
proof —
let \( ?t = \text{length } \pi \)
and \( ?\text{ops} = \text{strips-problem.operators-of } \Pi \)
let \( ?\Phi_O = \text{encode-operators } \Pi \ ?t \)
and \( ?\Phi_P = \text{encode-all-operator-preconditions } \Pi \ ?\text{ops} \ ?t \)
and \( ?\Phi_E = \text{encode-all-operator-effects } \Pi \ ?\text{ops} \ ?t \)
fix $C$
assume $C \in \text{cnf } \Phi_O$
then consider (C-in-precondition-encoding) $C \in \text{cnf } \Phi_P$
| (C-in-effect-encoding) $C \in \text{cnf } \Phi_E$
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 cnf-semantics-def
by blast
qed

lemma encode-problem-parallel-complete-iv-a:
fixes $\Pi :: 'a \text{strips-problem}$
assumes STRIPS-Semantics.is-parallel-solution-for-problem $\Pi \pi$
and $\forall k \ op. k < \text{length } \text{trace-parallel-plan-strips } ((\Pi)_I) \pi - 1$
$\rightarrow A (\text{Operator } k \ (\text{index } \text{strips-problem.operators-of } \Pi) \ op) = (\ op \in \text{set } (\pi ! k))$
and $\forall v \ l. \ l \geq \text{length } \text{trace-parallel-plan-strips } ((\Pi)_I) \pi \land l < \text{length } \pi + 1$
$\rightarrow A (\text{State } l \ (\text{index } \text{strips-problem.variables-of } \Pi) v) = A (\text{State}\
\quad (\text{length } \text{trace-parallel-plan-strips } ((\Pi)_I) \pi - 1)\
\quad (\text{index } \text{strips-problem.variables-of } \Pi) v))$
and $C \in \bigcup \{ (\bigcup(k, v) \in \{0..<\text{length } \pi \} \times \text{set } ((\Pi)_V).\}
\quad \{(\text{State } k \ (\text{index } \text{strips-problem.variables-of } \Pi) v)v)\} +$
\quad , (\text{State } (\text{Suc } k) \ (\text{index } \text{strips-problem.variables-of } \Pi) v) v)\} - 1 \}$
\quad \cup \{(\text{Operator } k \ (\text{index } \text{strips-problem.operators-of } \Pi) \ op) v)\} +$
\quad | op, \ op \in \text{set } ((\Pi)_O) \land v \in \text{set } (\text{add-effects-of } \op)\} \} \})\}
shows clause-semantics $A \ C$
proof --
let ?vs = strips-problem.variables-of Π
and ?ops = strips-problem.operators-of Π
and ?t = length π
let ?τ = trace-parallel-plan-strips ((Π), π)
let ?A = (∪(k, v) ∈ {0..<?t} × set ?vs.
 {{(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) }})

{ 
  obtain C' where C' ∈ ?A and C-in-C': C ∈ C'
  using Union-iff assms(5)
  by auto 
  then obtain k v 
    where (k, v) ∈ {0..<?t} × set ?vs
    and 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 UN-E
    by blast
  hence ∃k v.
    k ∈ {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) (π ! k)
and all-operator-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+
then consider (A) ∃op ∈ set (π ! k). v ∈ set (add-effects-of op)
| (B) ∃op ∈ set (π ! k). v ∈ set (delete-effects-of op)
\[
(C) \quad \forall \, op \in \text{set} \,(\pi \! k), \, v \notin \text{set} \,(\text{add-effects-of } op) \land v \notin \text{set} \,(\text{delete-effects-of } op)
\]

by blast

thus \(?thesis\)

proof (cases)

\begin{itemize}
\item \textbf{case } A
\item moreover obtain \(op\)
  \begin{itemize}
  \item where \(op-in-\pi_k\): \(op \in \text{set} \,(\pi \! k)\)
  \item and \(v\text{-is-add-effect}: v \in \text{set} \,(\text{add-effects-of } op)\)
  \item using \(A\)
  \item by blast
  \end{itemize}
\item moreover \{
  \begin{itemize}
  \item have \((\pi \! k) \in \text{set} \pi\)
  \item using \(k\text{-lt-t}\)
  \item by simp
  \item hence \(op \in \text{set} \, ?ops\)
  \item using \(\text{is-parallel-solution-for-problem-operator-set}[OF \, \text{assms}(1) - op-in-\pi_k]\)
  \item by blast
  \end{itemize}
\item ultimately have \((\text{Operator } k \, (\text{index } ?ops \, op))^{+} \in \{ (\text{Operator } k \, (\text{index } ?ops \, op))^{+} \mid op, \, op \in \text{set} \, ?ops \land v \in \text{set} \,(\text{add-effects-of } op) \}\)
  \begin{itemize}
  \item using \(v\text{-is-add-effect}\)
  \item by blast
  \end{itemize}
\item then have \((\text{Operator } k \, (\text{index } ?ops \, op))^{+} \in C\)
  \begin{itemize}
  \item using \(C\text{-is}\)
  \item by auto
  \end{itemize}
\item moreover have \(A \,(\text{Operator } k \, (\text{index } ?ops \, op))\)
  \begin{itemize}
  \item using \(\text{assms}(2) \, k\text{-lt-length-}\tau\text{-minus-one } op-in-\pi_k\)
  \item by blast
  \end{itemize}
\item ultimately show \(?thesis\)
  \begin{itemize}
  \item unfolding \(\text{clause-semantics-def}\)
  \item by force
  \end{itemize}
\end{itemize}

next

\begin{itemize}
\item \textbf{case } B
\item then obtain \(op\)
  \begin{itemize}
  \item where \(op-in-\pi_k\): \(op \in \text{set} \,(\pi \! k)\)
  \item and \(v\text{-is-delete-effect}: v \in \text{set} \,(\text{delete-effects-of } op)\)
  \item then have \(\neg(\exists \, op \in \text{set} \,(\pi \! k), \, v \in \text{set} \,(\text{add-effects-of } op))\)
  \begin{itemize}
  \item using \(\text{all-operator-effects-consistent are-all-operator-effects-consistent-set}\)
  \item by fast
  \end{itemize}
\item then have \(\text{execute-parallel-operator} \,(?\tau \! k) \,(\pi \! k) \, v = \text{Some } False\)
  \begin{itemize}
  \item using \(\text{execute-parallel-operator-negative-effect-if}[OF \, \text{all-operators-applicable all-operator-effects-consistent op-in-}\pi_k \, v\text{-is-delete-effect}]\)
  \item by blast
  \end{itemize}
\item moreover have \((?\tau \! Suc \, k) \, v = \text{execute-parallel-operator} \,(?\tau \! k) \,(\pi \! k) \, v\)
\end{itemize}

267
using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
by simp
ultimately have \( \neg A\ (\text{State} (\text{Suc} k) (\text{index} \ ?\ vs \ v)) \)
using assms(3) k-lt-length-τ-minus-one
by simp
thus \( \text{thesis} \)
using C-is
unfolding clause-semantics-def
by simp
next
case C
show \( \text{thesis} \)
proof (cases (\( ?\tau ! \ k \) \( v = \text{Some True} \))
  case True
  then have \( A\ (\text{State} k (\text{index} \ ?\ vs \ v)) \)
  using assms(3) k-lt-length-τ-minus-one
  by force
  thus \( \text{thesis} \)
  using C-is
  unfolding clause-semantics-def
  by fastforce
next
case False
  
  have \( (\ ?\tau ! \ \text{Suc} \ k ) = \text{execute-parallel-operator} \ (\ ?\tau ! \ k ) (\pi ! \ k ) \)
  using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one].
  then have \( (\ ?\tau ! \ \text{Suc} \ k ) \ v = (\ ?\tau ! \ k ) \ v \)
  using execute-parallel-operator-no-effect-if C
  by fastforce
  hence \( (\ ?\tau ! \ \text{Suc} \ k ) \ v \neq \text{Some True} \)
  using False
  by argo
  
  then have \( \neg A\ (\text{State} (\text{Suc} k) (\text{index} \ ?\ vs \ v)) \)
  using assms(3) k-lt-length-τ-minus-one
  by auto
  thus \( \text{thesis} \)
  using C-is
  unfolding clause-semantics-def
  by fastforce
qed
qed
next
case k-gte-length-τ-minus-one: False
show \( \text{thesis} \)
proof (cases A (\( \text{State} ((\text{length} \ ?\tau - 1) (\text{index} \ ?\ vs \ v)) )\))
  case True
  
  have \( A\ (\text{State} k (\text{index} \ ?\ vs \ v)) = A\ (\text{State} ((\text{length} \ ?\tau - 1) (\text{index} \ ?\ vs \ v)) )\)
proof (cases k = length ?τ - 1)
  case False
  then have length ?τ ≤ k and k < ?t + 1
    using k-gte-length-τ-minus-one k-in
    by fastforce+
  thus ?thesis
    using assms(4)
    by blast
  qed blast
  hence A (State k (index ?vs v))
    using True
    by blast
  qed blast
next
  case False
  { 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 False
      by blast
  }
  thus ?thesis
    using C-is
    unfolding clause-semantics-def
    by simp
next qed
lemma encode-problem-parallel-complete-iv-b:
  fixes II :: 'a strips-problem
  assumes is-parallel-solution-for-problem II π
    and ∀ k op. k < length (trace-parallel-plan-strips ((II)₁) π) - 1
    → A (Operator k (index (strips-problem.operators-of II) op)) = (op ∈ set (π
      ! k))
    and ∀ v k. k < length (trace-parallel-plan-strips ((II)₁) π)
    → (A (State k (index (strips-problem.variables-of II) v))
      ↔ (trace-parallel-plan-strips ((II)₁) π ! k) v = Some True)
∀ v l. l ≥ length (trace-parallel-plan-strips ((II)I) π) ∧ l < length π + 1
→ A (State l (index (strips-problem.variables-of II) v))

= A (State
 (length (trace-parallel-plan-strips ((II)I) π) - 1)
 (index (strips-problem.variables-of II) v))

and C ∈ \bigcup \bigl( \bigcup (k, v) \in \{0..<\text{length π}\} \times \text{set ((II)I)} \bigr).

\{\{ (State k (index (strips-problem.variables-of II) v))^-1
 , (State (Suc k) (index (strips-problem.variables-of II) v))^+ \}
 \cup \{ (Operator k (index (strips-problem.operators-of II) op))^+
 | op. op ∈ \text{set ((II)C)} ∧ v ∈ \text{set (delete-effects-of op)} \} \})

shows clause-semantics A C

proof

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

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

and ?t = length π

let ?A = \{\{ (State k (index ?vs v))^-1
 , (State (Suc k) (index ?vs v))^+ \}
 \cup \{ (Operator k (index ?ops op))^+
 | op. op ∈ \text{set ((II)C)} ∧ v ∈ \text{set (delete-effects-of op)} \} \}

obtain C' where C' ∈ ?A and C-in-C': C ∈ C'

using Union-iff assms(5)

by auto

then obtain k v

where (k, v) ∈ \{0..<?t\} \times \text{set ?vs}

and C' ∈ \{\{ (State k (index ?vs v))^-1
 , (State (Suc k) (index ?vs v))^+ \}
 \cup \{ (Operator k (index ?ops op))^+
 | op. op ∈ \text{set ?ops ∧ v ∈ set (delete-effects-of op)} \} \}

using UN-E

by fastforce

hence \exists k v.

k ∈ \{0..<?t\}

∧ v ∈ \text{set ?vs}

∧ C = \{ (State k (index ?vs v))^-1
 , (State (Suc k) (index ?vs v))^+ \}
 \cup \{ (Operator k (index ?ops op))^+
 | op. op ∈ \text{set ((II)C)} ∧ v ∈ \text{set (delete-effects-of op)} \}

using C-in-C'

by auto

then obtain k v

where k-in: k ∈ \{0..<?t\}

and v-in-vs: v ∈ \text{set ((II)I)}

and C-is: C = \{ (State k (index ?vs v))^-1
 , (State (Suc k) (index ?vs v))^+ \}
 \cup \{ (Operator k (index ?ops op))^+
 | op. op ∈ \text{set ((II)C)} ∧ v ∈ \text{set (delete-effects-of op)} \}
by auto
show ?thesis
proof (cases \( k < \text{length } \tau - 1 \))
  case \( k\text{-lt-length-}\tau\text{-minus-one} : \text{True} \)
    then have \( k\text{-lt-t} : k < \tau \)
      using \( k\text{-in} \)
      by force
    have all-operators-applicable: \( \text{are-all-operators-applicable } (\tau ! k) (\pi ! k) \)
      and all-operator-effects-consistent: \( \text{are-all-operator-effects-consistent } (\pi ! k) \)
      using trace-parallel-plan-strips-operator-preconditions[OF \( k\text{-lt-length-}\tau\text{-minus-one} \)]
      by simp+
    then consider \( (A) \exists op \in \text{set } (\tau ! k). \ v \in \text{set } (\text{delete-effects-of } op) \)
      \( | (B) \exists op \in \text{set } (\pi ! k). \ v \in \text{set } (\text{add-effects-of } op) \)
      \( | (C) \forall op \in \text{set } (\pi ! k). \ v \notin \text{set } (\text{add-effects-of } op) \land v \notin \text{set } (\text{delete-effects-of } op) \)
      by blast
    thus ?thesis
  proof (cases)
    case A
    moreover obtain op
      where op-in-\( \pi_k \): \( op \in \text{set } (\tau ! k) \)
      and v-is-delete-effect: \( v \in \text{set } (\text{delete-effects-of } op) \)
      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 is-parallel-solution-for-problem-operator-set[OF assms(1) - op-in-\( \pi_k \)]
        by auto
    }
    ultimately have \( (\text{Operator } k (\text{index } \text{ops } op))^+ \in \{ (\text{Operator } k (\text{index } \text{ops } op))^+ \mid \text{op. } op \in \text{set } \text{ops } \land v \in \text{set } (\text{delete-effects-of } op) \} \)
      using 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 assms(2) k-lt-length-\( \tau \text{-minus-one } \text{op-in-}\pi_k \)
      by blast
    ultimately show ?thesis
      unfolding clause-semantics-def
      by force
next
  case B
then obtain $op$
where $op$-in-$π_k$: $op \in \text{set}(π_k)$
and $v$-is-add-effect: $v \in \text{set}(\text{add-effects-of } op)$.
then have $¬(\exists op \in \text{set}(π_k). v \in \text{set}(\text{delete-effects-of } op))$
using all-operator-effects-consistent are-all-operator-effects-consistent-set
by fast
then have execute-parallel-operator (?τ ! k) (π ! k) $v$ = Some True
using execute-parallel-operator-positive-effect-if[OF all-operators-applicable
all-operator-effects-consistent op-in-π_k v-is-add-effect]
by blast
moreover have (?τ ! Suc k) $v$ = execute-parallel-operator (?τ ! k) (π ! k)

using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
by simp
ultimately have A (State (Suc 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 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

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 ?thesis
  using C-is
  unfolding clause-semantics-def
  by fastforce
next
case False

{ have A (State k (index ?vs v)) = A (State (length ?τ - 1) (index ?vs v))
  proof (cases k = length ?τ - 1)
    case False
    then have length ?τ ≤ k and k < ?t + 1
      using k-gte-length-τ-minus-one k-in
      by fastforce+
    thus ?thesis
      using assms(4)
      by blast
    qed blast
    hence ¬A (State k (index ?vs v))
      using False
      by blast
  }
  thus ?thesis
  using C-is
  unfolding clause-semantics-def
  by simp
lemma encode-problem-parallel-complete-iv:
fixes II::'a strips-problem
assumes is-valid-problem-strips II
and is-parallel-solution-for-problem II π
and \( \forall k \) op. \( k < \text{length} \) (trace-parallel-plan-strips ((II) I) π) = 1
\( \rightarrow \) A (Operator k (index (strips-problem.operators-of II) op)) = (op \in \text{set} (π ! k))
\( \forall v \) k. \( k < \text{length} \) (trace-parallel-plan-strips ((II) I) π)
\( \rightarrow \) A (State k (index (strips-problem.variables-of II) v))
\( \leftrightarrow \) (trace-parallel-plan-strips ((II) I) π ! k) v = Some True
\( \forall v \) l. \( l \geq \text{length} \) (trace-parallel-plan-strips ((II) I) π) \( \land \) l < length π + 1
\( \rightarrow \) A (State l (index (strips-problem.variables-of II) v)) = A (State (length (trace-parallel-plan-strips ((II) I) π) − 1) (index (strips-problem.variables-of II) v))
shows A \models encode-all-frame-axioms II (length π)
proof –
let \( ?Φ_F = \text{encode-all-frame-axioms II (length π) \text{ using } cnf-of-encode-all-frame-axioms-structure} \) by (simp add: cnf-of-encode-all-frame-axioms-structure)
\{ fix C assume C \in cnf ?Φ_F then consider \( (C\text{-in-A}) \) C \in ?A | \( (C\text{-in-B}) \) C \in ?B using Un-iff[of C ?A ?B] cnf-Φ-F-is-A-union-B by argo hence clause-semantics A C proof (cases) case C\text{-in-A} then show ?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 ?thesis
  using encode-problem-parallel-complete-iv-b[OF assms(2, 3, 4, 5) C-in-B]
  by blast
qed

thus ?thesis
  using encode-frame-axioms-is-cnf is-nnf-cnf cnf-semantics
  unfolding cnf-semantics_def
  by blast
qed

lemma valuation-for-operator-variables-is:
  fixes Π :: ′a strips-problem
  assumes is-parallel-solution-for-problem Π π
  and k < length (trace-parallel-plan-strips ((Π)t) π) − 1
  and op ∈ set ((Π)O)
  shows valuation-for-operator-variables Π π (trace-parallel-plan-strips ((Π)t) π)
    (Operator k (index (strips-problem.operators-of Π) op))
    = (op ∈ set (π ! k))
proof
  let ?ops = strips-problem.operators-of Π
  and ?τ = trace-parallel-plan-strips ((Π)t) π
  let ?v = Operator k (index ?ops op)
  and ?Op = { Operator k (index ?ops op) | k op. k ∈ {0..<length ?τ − 1} ∧ op ∈ set ((Π)O) }
  let ?l = concat (map (λk. map (Pair k) (π ! k)) [0..<length ?τ − 1])
  and ?f = λx. Operator (fst x) (index ?ops (snd x))
  — show that our operator construction function is injective on set (concat (map (λk. map (Pair k) (π ! k)) [0..<length ?τ − 1])).
  have k-in: k ∈ {0..<length ?τ − 1}
  using assms(2)
  by fastforce
{

  { 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')
      then have k-is-k': k = k'
      by fast

275
moreover { 
    have $k' \lt$: $k' < \text{length } ?\tau - 1$
    using $k'\text{-op'}\text{-in}$
    by fastforce
}

have $op\text{-in: } op \in \text{set } (\pi \mid k)$
    using $k\text{-op-in}$
    by force

then have $op'\text{-in: } op' \in \text{set } (\pi \mid k)$
    using $k'\text{-op'}\text{-in } k\text{-is-k'}$
    by auto
{
    have $\text{length}-\tau\text{-gt-1}: \text{length } ?\tau > 1$
        using $\text{assms}(2)$
        by linarith
    have $\text{length } ?\tau - \text{Suc 0} \leq \text{length } \pi + 1 - \text{Suc 0}$
        using $\text{length-parallel-plan-strips-}\text{ltc-length-plan-plus-one}$
        using $\text{diff-le-mono}$
        by blast
    then have $\text{length } ?\tau - 1 \leq \text{length } \pi$
        by fastforce
    then have $k' < \text{length } \pi$
        using $\text{length-}\tau\text{-gt-1 } k'\text{-lt}$
        by linarith
    hence $\pi \mid k' \in \text{set } \pi$
        by simp
}

moreover have $op \in \text{set } ?ops$ and $op' \in \text{set } ?ops$
    using $\text{is-parallel-solution-for-problem-operator-set}[\text{OF } \text{assms}(1)]$
    $op\text{-in}$
    $op'\text{-in } k\text{-is-k'}$
        calculation
        by auto
    ultimately have $op = op'$
        using $\text{index-op-is-index-op'}$
        by force
}

ultimately show $(k, op) = (k', op')$
    by blast
qed fast
}

hence $\text{inj-on } ?f (\text{set } ?l)$

unfolding $\text{inj-on-def } \text{fst-def } \text{snd-def}$
    by fast
}

note $\text{inj-on-f-set-l = this}$

{ 
    have $\text{set } ?l = \bigcup (\text{set } (\text{map } (\lambda k. \text{map } (\text{Pair } k) (\pi \mid k))) [0..<\text{length } ?\tau -
using set-concat
by metis
also have \ldots = \bigcup (set ' (λ k. map (Pair k) (π ! k)) \cdot \{0..<\text{length } ?τ - 1\})
by force
also have \ldots = \bigcup ((λ k. (Pair k) \cdot set (π ! k)) \cdot \{0..<\text{length } ?τ - 1\})
by force
also have \ldots = \bigcup ((λ k. \{ (k, op) | op \in set (π ! k) \}) \cdot \{0..<\text{length } ?τ - 1\})
by blast
also have \ldots = \bigcup (\{ (k, op) \} | k \in \{0..<\text{length } ?τ - 1\} \land op \in set (π ! k))
by blast
finally have set ?l = \bigcup (\{ (k, op) \} | (k, op) \in \{0..<\text{length } ?τ - 1\} \land op \in set (\pi ! k))
using setcompr-eq-image[of λ (k, op). \{ (k, op) \}]
by auto
} note set-l-is = this
{
  have Operator k (index ?ops op) ∈ ?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))
case True
  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 (λx A. A (Operator k (index ?ops op) := True)) (?l A0 (Operator (fst x) (index ?ops (snd x)) := ?g x))
    = (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 ∨ 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 ?f (k′, op′) = ?f (k, op)
    hence (k′, op′) = (k, op)
    using inj-on-f-set-l assms(3)
    by simp
  }

ultimately have Operator k (index ?ops op) ∉ ?f ' set ?l
  using image-iff
  by force
} note operator-not-in-f-image-set-l = this
{
  have A_0 (Operator k (index ?ops op)) = False
    by simp
  moreover have (λ(k, op) A. A (Operator k (index ?ops op) := True))
    = (λx. 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))
    ?l A_0 ?v = False
    using foldr-fun-no-upd[OF inj-on-f-set-l operator-not-in-f-image-set-l, of
    λ-. True A_0]
    by presburger
  }
  thus ?thesis
  using nb False
  by blast
qed

lemma encode-problem-parallel-complete-vi-a:
  fixes II :: 'a strips-problem
  assumes is-parallel-solution-for-problem II π
and $k < \text{length } \text{trace-parallel-plan-strips } (((\Pi)_{\pi}) \pi) - 1$

shows valuation-for-plan $\Pi \pi (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) \text{op})) = (\text{op } \in \text{set } (\pi ! k))$

proof –

let $?vs = \text{strips-problem.variables-of } \Pi$
and $?ops = \text{strips-problem.operators-of } \Pi$
and $?t = \text{length } \pi$
and $?\tau = \text{trace-parallel-plan-strips } ((\Pi)_{\pi})$

let $?A_{\pi} = \text{valuation-for-plan } \Pi \pi$
and $?A_{O} = \text{valuation-for-operator-variables } \Pi \pi$
and $?Op = \{ \text{Operator } k (\text{index } ?ops \text{op}) \mid k \text{op. } k \in \{0..<?t\} \land \text{op } \in \text{set } ?ops \}$

and $?V = \{ \text{State } k (\text{index } ?vs \text{v}) \mid k \text{v. } k \in \{0..<?t+1\} \land \text{v } \in \text{set } ?vs \}$
and $?v = \text{Operator } k (\text{index } ?ops \text{op})$

{ have length $?\tau \leq \text{length } \pi + 1
  then have length $?\tau - 1 \leq \text{length } \pi$
    by simp
  then have $k < $?t
    using assms
    by fastforce
} note $k$-lt-length-$\pi = \text{this}$

show $?thesis$

proof (cases $\text{op } \in \text{set } ((\Pi)_{O})$)
  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 $\neg$-index $?ops \text{op } < \text{length } ?ops$ due to the assumption that $\neg$-op $\in \text{set } ?ops$. Hence $\neg k \in \{0..<?t\}$ and therefore $?v \notin ?Op.$
    have $?Op = (\lambda (k, \text{op}). \text{Operator } k (\text{index } ?ops \text{op})) \cdot (\{0..<?t\} \times \text{set } ?ops)$
      by fast
    moreover have $\neg$-index $?ops \text{op } < \text{length } ?ops$
using False
by simp
ultimately have \( ?v \notin ?Op \)
by fastforce
}
moreover have \( ?v \notin ?V \)
by force
ultimately have \( ?A \pi \ ?v \in A_0 \ ?v \)
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-It-length-\(\pi\)
by simp
moreover have \(op \notin \text{set} \ (\pi \! k)\)
using is-parallel-solution-for-problem-operator-set[OF assms(1) calculation(2)] False
by blast
ultimately show \(?\text{thesis}\)
by blast
qed
qed

lemma encode-problem-parallel-complete-vi-b:
fixes \( II :: 'a \text{strips-problem} \)
assumes is-parallel-solution-for-problem \( II \ \pi \)
and \( l \geq \text{length} \ (\text{trace-parallel-plan-strips} ((II)_I) \ \pi) - 1 \)
and \( l \lt \text{length} \ \pi \)
shows \( \neg \text{valuation-for-plan} \ II \ \pi \ (\text{Operator} \ l \ (\text{index} \ (\text{strips-problem.operators-of} \ II) \ op)) \)
proof –

let \( ?vs = \text{strips-problem.variables-of} \ II \)
and \( ?ops = \text{strips-problem.operators-of} \ II \)
and \( ?t = \text{length} \ \pi \)
and \( ?\tau = \text{trace-parallel-plan-strips} ((II)_I) \ \pi \)
let \( ?A_\pi = \text{valuation-for-plan} \ II \ \pi \)
and \( ?A_\omega = \text{valuation-for-operator-variables} \ II \ \pi \ \pi \)
and \( ?Op = \{ \ \text{Operator} \ k \ (\text{index} \ ?ops \ op) \mid k \ \text{op.} \ k \in \{0..<?t\} \land \text{op} \in \text{set} \ ?ops \} \)
and \( ?Op^' = \{ \ \text{Operator} \ k \ (\text{index} \ ?ops \ op) \mid k \ \text{op.} \ k \in \{0..<\text{length} \ \pi \ \tau - 1\} \land \text{op} \in \text{set} \ ?ops \} \)
and \( ?V = \{ \ \text{State} \ k \ (\text{index} \ ?vs \ v) \mid k \ v. \ k \in \{0..<?t + 1\} \land v \in \text{set} \ ?vs \} \)
and \( ?v = \text{Operator} \ l \ (\text{index} \ ?ops \ op) \)
show \( \text{thesis} \)

\textbf{proof} (cases \( \text{op} \in \text{set} \ (\Pi \circ) \))

\textbf{case} \( \text{True} \)

\{ 
  
  \{ 
    
    have \( \text{?v} \in \text{?Op} \)
    using \text{assms(3)} \text{ True}
    by auto

    hence \( \text{?A} \pi \text{?v} = \text{?A} \circ \text{?v} \)
    unfolding \text{valuation-for-plan-def override-on-def Let-def}
    by simp
  \}

  moreover \{ 
    
    have \( \text{l} \notin \{0..< \text{length } ?t - 1\} \)
    using \text{assms(2)}
    by simp
    
    then have \( \text{?v} \notin \text{?Op}' \)
    by blast
    
    hence \( \text{?A} \circ \text{?v} = \text{A} \circ \text{?v} \)
    unfolding \text{valuation-for-operator-variables-def override-on-def}
    by meson
  \}

  ultimately have \( \neg \text{?A} \pi \text{?v} \)
  unfolding \text{empty-valuation-def}
  by blast

\}

then show \( \text{thesis} \)
by blast

\textbf{next}

\textbf{case} \( \text{False} \)

\{ 

  \{ 
    
    \text{— We have } \neg \text{index } \text{?ops} \text{ op } < \text{length } \text{?ops} \text{ due to the assumption that } \neg \text{op } 
    \text{in set } \text{?ops}. \text{ Hence } \neg \text{k } \in \{0..< \text{?t} \} \text{ and therefore } \text{?v} \notin \text{?Op}.
  \}

  
  \text{have } \text{?Op} = (\lambda(\text{k}, \text{op}). \text{Operator } \text{k } (\text{index } \text{?ops} \text{ op} )) \times \{0..< \text{?t} \} \times \text{set } \text{?ops}
  by fast

  moreover have \( \neg \text{index } \text{?ops} \text{ op } < \text{length } \text{?ops} \)
  using \text{False}
  by simp

  ultimately have \( \text{?v} \notin \text{?Op} \)
  by fastforce

\}

moreover have \( \text{?v} \notin \text{?V} \)
by force

ultimately have \( \text{?A} \pi \text{?v} = \text{A} \circ \text{?v} \)
unfolding \text{valuation-for-plan-def override-on-def}
by metis
hence \( \neg \exists_A \pi \ ?v \)
unfolding empty-valuation-def
by blast
}
thus \( \?thesis \)
by blast
qed

— As a corollary from lemmas and we obtain the result that the constructed valuation \( A \equiv valuation-for-plan \Pi \pi \) valuates SATPlan operator variables as false if they are not contained in any operator set \( \pi ! k \) for any time point \( k < length \pi \).
corollary encode-problem-parallel-complete-vi-d:

fixes \( \Pi :: \) 'variable strips-problem
assumes is-parallel-solution-for-problem \( \Pi \pi \)
and \( k < length \pi \)
and \( op / \notin set (\pi ! k) \)
shows \( \neg valuation-for-plan \Pi \pi (\text{Operator } k (\text{index } (\text{strips-problem.operators-of} \Pi) op)) \)
using encode-problem-parallel-complete-vi-a[\text{OF assms (1)}] assms(3)
encode-problem-parallel-complete-vi-b[\text{OF assms(1) - assms(2)}] assms(3)
by (cases \( k < length (\text{trace-parallel-plan-strips } (\Pi)_I \pi) - 1 \}; fastforce)

lemma list-product-is-nil-iff: List.product \( xs ys = [] \) if \( xs = [] \lor ys = [] \)
proof (rule iffI)
assume product-xs-ys-is-Nil: List.product \( xs ys = [] \)
show \( xs = [] \lor ys = [] \)
proof (rule ccontr)
assume \( \neg (xs = [] \lor ys = []) \)
then have \( xs \neq [] \land ys \neq [] \)
by simp+
then obtain \( x x' y y' \) where \( xs = x \# xs' \) and \( ys = y \# ys' \)
using list.exhaust
by metis
then have List.product \( xs ys = (x, y) \# \text{map } (\text{Pair } x) \ ys' @ List.product \) \( xs' (y \# ys') \)
by simp
thus False
using product-xs-ys-is-Nil
by simp
qed

next
assume \( xs = [] \lor ys = [] \)
thus 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 List.product xs ys = []
  proof (induction xs)
    case (Cons x xs)
      have List.product (x # xs) ys = map (Pair x) ys @ List.product xs ys
        by simp
    also have ... = [] @ List.product xs ys
      using Nil-is-map-conv ys-is-Nil
      by blast
  finally show ?case
    using Cons.IH
    by force
  qed auto
  qed simp

— 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 ∈ set ks
  and v ∈ set vs
  shows foldr (λ(k, v) A. valuation-for-state vs (s k) k v A) (List.product ks vs) A0
              (State k (index vs v)) ←→ (s k) v = Some True
  proof
    let ?v = State k (index vs v)
    and ?ps = List.product ks vs
    let ?A = foldr (λ(k, v) A. valuation-for-state vs (s k) k v A) ?ps A0
    and ?f = λx. State (fst x) (index vs (snd x))
    and ?g = λx. (s (fst x)) (snd x) = Some True
    have nb1: (k, v) ∈ set ?ps
      using assms(1, 2) set-product
      by simp
    moreover {
      
      fix x y
      and ~(?f x = ?f y → x = y)
      then have f-x-is-f-y: ?f x = ?f y and x-is-not-y: x ≠ y
        by blast
      then obtain k’ k” v’ v’’
        where x-is: x = (k’, v’)
        and y-is: y = (k”, v’’)
        by fastforce
      then consider (A) k’ ≠ k”
        | (B) v’ ≠ v’’
    }

283
using \( x \neq y \)
by blast

hence False

proof (cases)
  case A
  then have \(? f \neq ? f\)
  using \( x \neq y \)
  by simp
  thus \(? f\)
  using \(? f \neq f\)
  by argo

next
  case B
  have \( v' \in \text{set vs} \) and \( v'' \in \text{set vs} \)
  using \( x \in \text{ps} \)
  by blast
  then have \( \text{index vs } v' \neq \text{index vs } v'' \)
  using B
  by force
  then have \(? f \neq ? f\)
  using \( x \neq y \)
  by simp
  thus False
  using \(? f \neq f\)
  by blast

qed

} hence inj-on \(? f \) (set \(? ps\))
using inj-on-def
by blast

} note \( nb_2 = \text{this} \)

{ have foldr (\( \lambda x. \text{valuation-for-state vs } (s \text{ (fst x)})) (\text{fst x}) (\text{snd x})\)
(List.product ks vs) \( A_0 \) (State (\( \text{fst} (k, v)\)) (\text{index vs } (\text{snd } (k, v)))) =
(\( s \text{ (fst } k, v)\)) (\( \text{snd } (k, v)\)) = Some True
  using foldr-fun-upd[\( OF nb_2 \) \( nb_1 \), \text{of } \(? g A_0\)]
  by blast
moreover have (\( \lambda x. \text{valuation-for-state vs } (s \text{ (fst x)})) (\text{fst x}) (\text{snd x})\)
= (\( \lambda(k, v). \text{valuation-for-state vs } (s \text{ } k \text{ } v)\)
  by fastforce
ultimately have \(? A \) (\( ? f (k, v)\)) = \(? g (k, v)\)
  by simp

thus \(? f\)
by simp
qed


lemma encode-problem-parallel-complete-vi-c:
fixes $\Pi :: 'a$ strips-problem
assumes $\text{is-valid-problem-strips } \Pi$
\hspace{1em} and $\text{is-parallel-solution-for-problem } \Pi \pi$
\hspace{1em} and $k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$
shows $\text{valuation-for-plan } \Pi \pi (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))$
\hspace{1em} $\leftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$

proof –

let $?\text{vs} = \text{strips-problem.variables-of } \Pi$
\hspace{1em} and $?\text{ops} = \text{strips-problem.operators-of } \Pi$
\hspace{1em} and $?\tau = \text{trace-parallel-plan-strips } ((\Pi)_I) \pi$
let $?t = \text{length } \pi$
\hspace{1em} and $?t' = \text{length } ?\tau$
let $?A_\pi = \text{valuation-for-plan } \Pi$
\hspace{1em} and $?A_V = \text{valuation-for-state-variables } \Pi ?\tau$
\hspace{1em} and $?A_O = \text{valuation-for-state-variables } \Pi ?\tau$
\hspace{1em} and $?A_1 = \text{foldr} (\lambda (k, v). A. \text{valuation-for-state } ?\text{vs} (?\tau ! k) k v A)$
\hspace{1em} $(\text{List.product } [0..<?t'] ?\text{vs}) A_0$
\hspace{1em} and $?\text{Op} = \{ \text{Operator } k (\text{index } ?\text{ops op}) | k \text{ op. } k \in \{0..<?t'\} \land \text{op } \in \text{set } ((\Pi)_O) \}$
\hspace{1em} and $?\text{Op}' = \{ \text{Operator } k (\text{index } ?\text{ops op}) | k \text{ op. } k \in \{0..<?t' - 1\} \land \text{op } \in \text{set } ((\Pi)_O) \}$
\hspace{1em} and $?V = \{ \text{State } k (\text{index } ?\text{vs v}) | k \text{ v. } k \in \{0..<?t + 1\} \land \text{v } \in \text{set } ((\Pi)_V) \}$
\hspace{1em} and $?V_1 = \{ \text{State } k (\text{index } ?\text{vs v}) | k \text{ v. } k \in \{0..<?t'\} \land \text{v } \in \text{set } ((\Pi)_V) \}$
\hspace{1em} and $?V_2 = \{ \text{State } k (\text{index } ?\text{vs v}) | k \text{ v. } k \in \{?t'.(?t + 1)\} \land \text{v } \in \text{set } ((\Pi)_V) \}$
\hspace{1em} and $?v = \text{State } k (\text{index } ?\text{vs v})$

have $?\text{vs} \notin \text{Op}$
\hspace{1em} by blast
have $k \text{-lte-length-}\pi\text{-plus-one}: k < \text{length } \pi + 1$
using $\text{less-le-trans length-trace-parallel-plan-strips-lte-length-plan-plus-one assms(3)}$
\hspace{1em} by blast
show $?\text{thesis}$

proof (cases $v \in \text{set } ((\Pi)_V)$)
case True

{ 
  have $?v \in ?V ?v \notin ?\text{Op}$
\hspace{1em} using $k\text{-lte-length-}\pi\text{-plus-one True}$
\hspace{1em} by force+
\hspace{1em} unfolding $\text{valuation-for-plan-def override-on-def Let-def}$
\hspace{1em} by simp
}

moreover {
  have $?v \in ?V_1 ?v \notin ?V_2$
\hspace{1em} using assms(3) True
\hspace{1em} by fastforce+
}
hence \( \mathcal{A}_V \ ?v = \mathcal{A}_1 \ ?v \)
unfolding valuation-for-state-variables-def override-on-def Let-def
by force
}
ultimately have \( \mathcal{A}_\pi \ ?v = \mathcal{A}_1 \ ?v \)
by blast
}
moreover have \( k \in \text{set } [0..<?t'] \)
using assms(3)
by simp
moreover have \( v \in \text{set } \text{strips-problem.variables-of } \Pi \)
using True
by simp
ultimately show \( \text{thesis} \)
using valuation-for-state-variables-is[of k [0..<?t']]
by fastforce
next
case False
{
{
have \( \neg \ \text{index } ?vs \ v < \text{length } ?vs \)
using False index-less-size-conv
by simp
hence \( ?v \not\in \ ?V \)
by fastforce
}
then have \( \neg \mathcal{A}_\pi \ ?v \)
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 \( \neg (\exists ! k) \ v = \text{Some True} \)
using trace-parallel-plan-strips-none-if[of II \( \pi \ k \ v \) assms(1, 2, 3) False
unfolding initial-of-def
by force
ultimately show \( \text{thesis} \)
by blast
qed

lemma encode-problem-parallel-complete-vi-f:
fixes II :: 'a strips-problem
assumes is-valid-problem-strips II
and is-parallel-solution-for-problem II \( \pi \)
and \( l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi) \)
and \( l < \text{length } \pi + 1 \)
shows valuation-for-plan $\Pi v (\text{State}_l \index{strips-problem.variables-of \Pi v})$
$\ = \ valuation-for-plan \Pi v$
$(\text{State} \ length (\text{trace-parallel-plan-strips} ((\Pi)_1) \pi - 1))$
$(\index{strips-problem.variables-of \Pi v})$

proof —

let $\text{?vs} = \text{strips-problem.variables-of} \Pi$
and $\text{?ops} = \text{strips-problem.operators-of} \Pi$
and $\text{?r} = \text{trace-parallel-plan-strips} ((\Pi)_1) \pi$
let $\text{?t} = \text{length} \pi$
and $\text{?t}' = \text{length} \text{?r}$
let $\text{?r}_\Omega = \text{?r} ! (\text{?t}' - 1)$
and $\text{?A}_\pi = \text{valuation-for-plan} \Pi \pi$
and $\text{?A}_V = \text{valuation-for-state-variables} \Pi \pi \text{?r}$
and $\text{?A}_O = \text{valuation-for-state-variables} \Pi \pi \text{?r}$
let $\text{?A}_2 = \text{foldr}$
$(\lambda (k, v) \ A. \text{valuation-for-state} \text{?r}_\Omega k v A)$
$(\text{List.product} [?t'..<\text{length} \pi + 2] \text{?vs})$
$A_0$
and $\text{?O} = \{ \text{Operator} k \ (\text{index} \ ?ops \ op) \ | \ k \ op. \ k \in \{0..<?t\} \land \ op \in \text{set} ((\Pi)_O) \}$
and $\text{?O}' = \{ \text{Operator} k \ (\text{index} \ ?ops \ op) \ | \ k \ op. \ k \in \{0..<?t' - 1\} \land \ op \in \text{set} ((\Pi)_O) \}$
and $\text{?V} = \{ \text{State} k \ (\text{index} \ ?vs \ v) \ | \ k \ v. \ k \in \{0..<?t + 1\} \land \ v \in \text{set} ((\Pi)_V) \}$
and $\text{?V}_1 = \{ \text{State} k \ (\text{index} \ ?vs \ v) \ | \ k \ v. \ k \in \{0..<?t'\} \land \ v \in \text{set} ((\Pi)_V) \}$
and $\text{?V}_2 = \{ \text{State} k \ (\text{index} \ ?vs \ v) \ | \ k \ v. \ k \in \{?t'.(?t + 1)\} \land \ v \in \text{set} ((\Pi)_V) \}$
and $\text{?v} = \text{State}_l \ (\text{index} \ ?vs \ v)$

have v-notin-Op: $\text{?v} \notin \text{?O}$
by blast
show $\text{?thesis}$
proof (cases $v \in \text{set} ((\Pi)_V)$)
case True
{}

{ have $\text{?v} \in \text{?V} \text{?v} \notin \text{?O}$
using assms(4) True
by force+
hence $\text{?A}_\pi \text{?v} = \text{?A}_V \text{?v}$
unfolding valuation-for-plan-def override-on-def Let-def
by simp }
moreover {
have $\text{?v} \notin \text{?V}_1 \text{?v} \in \text{?V}_2$
using assms(3, 4) True
by force+
hence $\text{?A}_V \text{?v} = \text{?A}_2 \text{?v}$

287
ultimately have \( \mathcal{A}_\pi \ ?v = \mathcal{A}_2 \ ?v \)
by blast
} note \( nb = this \)
moreover
{
  have \( l \in set [\ ?t'.<?t + 2] \)
  using \( \text{assms}(3, 4) \)
  by auto

  hence \( \mathcal{A}_2 \ ?v \leftrightarrow \mathcal{\tau}_\Omega \ v = \text{Some True} \)
  using \( \text{valuation-for-state-variables-is-of l [\ ?t'.<?t + 2]} \ \text{True nb} \)
  by fastforce
}
ultimately have \( \mathcal{A}_\pi \ ?v \leftrightarrow \mathcal{\tau}_\Omega \ v = \text{Some True} \)
by fast
moreover {
  have \( 0 < \ ?t' \)
  using \( \text{trace-parallel-plan-strips-not-nil} \)
  by blast
  then have \( \ ?t' - 1 < \ ?t' \)
  using \( \text{diff-less} \)
  by presburger
}
ultimately show \( \ ?\text{thesis} \)
using \( \text{encode-problem-parallel-complete-vi-c[of - - \ ?t' - 1, OF \text{assms}(1, 2)]} \)
by blast
next
case False
{
  {
    have \( \neg \text{index} \ ?v < \text{length} \ ?vs \)
    using \( \text{False index-less-size-conv} \)
    by auto
    hence \( \ ?v \notin \ ?V \)
    by fastforce
  }
  then have \( \neg \mathcal{A}_\pi \ ?v \)
  using \( \text{v-notin-Op} \)
  unfolding \( \text{valuation-for-plan-def override-on-def empty-valuation-def Let-def} \)
  \( \text{variables-of-def operators-of-def} \)
  by presburger
}
moreover {
  have \( 0 < \ ?t' \)
  using \( \text{trace-parallel-plan-strips-not-nil} \)
  by blast
}
then have \(?t' - 1 < ?t'\)
by simp

moreover have \(\neg((\tau ! (\ ?t' - 1)) \ v = \text{Some True})\)
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
qed

Let now \(\tau \equiv \text{trace-parallel-plan-strips} \ I \ \pi\) be the trace of the plan \(\pi\), \(t \equiv \text{length} \ \pi\), and \(t' \equiv \text{length} \ \tau\).

Any model of the SATPlan encoding \(\mathcal{A}\) must satisfy the following properties:

1. for all \(k\) and for all \(op\) with \(k < t' - (1::'a)\)

\[\mathcal{A} (\text{Operator } k (\text{index (operators-of } \Pi \text{) } op)) = op \in \text{set} (\tau ! k)\]

2. for all \(l\) and for all \(op\) with \(t' - (1::'a) \leq l \text{ and } l < \text{length } \pi\) we require

\[\mathcal{A} (\text{Operator } l (\text{index (operators-of } \Pi \text{) } op))\]

3. for all \(v\) and for all \(k\) with \(k < t'\) we require

\[\mathcal{A} (\text{State } k (\text{index (variables-of } \Pi \text{) } v)) \rightarrow ((\tau ! k) \ v = \text{Some True})\]

4. and finally for all \(v\) and for all \(l\) with \(t' \leq l \text{ and } l < t + (1::'a)\) we require

\[\mathcal{A} (\text{State } l (\text{index (variables-of } \Pi \text{) } v)) = \mathcal{A} (\text{State } (t' - 1) (\text{index (variables-of } \Pi \text{) } v))\]

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 \(\mathcal{A} (\text{State } k (\text{index (} \Pi \text{) } v))\) for the SAT plan variable of every state variable \(v\)

\[^{12}\text{Cf. [3, Theorem 3.1, p. 1044] for the construction of } \mathcal{A}.\]
at time point $k$ if and only if $(\tau \mid k) v = \text{Some True}$ for the corresponding state $\tau \mid k$ at time $k$ (and $\neg \mathcal{A} (\text{State } k (\text{index } (\Pi v)) v)$ 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**

**encode-problem-parallel-complete:**

**assumes** is-valid-problem-strips $\Pi$

and is-parallel-solution-for-problem $\Pi \pi$

**shows** valuation-for-plan $\Pi \pi \models \Phi \Pi (\text{length } \pi)$

**proof**

- let $?t = \text{length } \pi$
  and $?I = (\Pi)_I$
  and $?G = (\Pi)_G$
  and $?A = \text{valuation-for-plan } \Pi \pi$

- have $\text{nb} \colon ?G \subseteq_m \text{execute-parallel-plan } ?I \pi$
  using $\text{assms}(2)$
  unfolding $\text{is-parallel-solution-for-problem-def}$
  by force

- have $?A \models \Phi_I \Pi$
  using $\text{encode-problem-parallel-complete-}\text{i}(\text{OF assms}(1) \text{ nb})$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{c}(\text{OF assms}(1, 2))$
  by $\text{presburger}$

- moreover have $?A \models (\Phi_G \Pi) ?t$
  using $\text{encode-problem-parallel-complete-}\text{ii}(\text{OF assms}(1) \text{ nb})$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{c}(\text{OF assms}(1, 2))$
  by $\text{presburger}$

- moreover have $?A \models \text{encode-operators } \Pi ?t$
  using $\text{encode-problem-parallel-complete-}\text{iii}(\text{OF assms}(1) \text{ nb})$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{a}(\text{OF assms}(2))$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{b}(\text{OF assms}(2))$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{c}(\text{OF assms}(1, 2))$
  by $\text{presburger}$

- moreover have $?A \models \text{encode-all-frame-axioms } \Pi ?t$
  using $\text{encode-problem-parallel-complete-}\text{iv}(\text{OF assms}(1, 2))$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{a}(\text{OF assms}(2))$
  $\text{encode-problem-parallel-complete-}\text{vi-}\text{c}(\text{OF assms}(1, 2))$
  by $\text{presburger}$
encode-problem-parallel-complete-vi-f[OF assms(1, 2)]
by presburger
ultimately show \(?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

$$
\neg(\text{Atom (Operator } k \text{ (index ops } op_1))
\lor \neg(\text{Atom (Operator } k \text{ (index ops } op_2))
$$

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 \Rightarrow \text{nat} \Rightarrow \text{variable strips-operator} \Rightarrow \text{variable strips-operator} \Rightarrow \text{sat-plan-variable formula}
where

encode-interfering-operator-pair-exclusion \(\Pi \ k \ op_1 \ op_2\)
\equiv let \ops = \text{operators-of } \Pi \ in
\neg(\text{Atom (Operator } k \text{ (index ops } op_1)))
\lor \neg(\text{Atom (Operator } k \text{ (index ops } op_2)))

definition encode-interfering-operator-exclusion
:: 'variable strips-problem \Rightarrow \text{nat} \Rightarrow \text{sat-plan-variable formula}
where

encode-interfering-operator-exclusion \(\Pi \ t \equiv \text{let}\)
\ops = \text{operators-of } \Pi
; \interfering = \text{filter (λ(op_1, op_2). index ops \(op_1\) \(\neq\) index ops \(op_2\)} \\land \text{are-operators-interfering \(op_1\ \(op_2\) (List.product \(ops\ \(ops\)) )} in \text{foldr (\(\land\)) \[encode-interfering-operator-pair-exclusion \(\Pi \ k \ op_1 \ op_2\) \(\(op_1, op_2\) \leftarrow \interfering, \(0..<t\) \(\neg\)\]}

291
A SATPlan encoding with interfering operator pair exclusion can now be defined by simply adding the conjunct `encode-interfering-operator-exclusion Π t` to the basic SATPlan encoding.

— NOTE This is the quadratic size encoding for the ∀-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 ∀-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`

(Φ ∀ - 52)

where `encode-problem-with-operator-interference-exclusion Π t` ≡ `encode-initial-state Π` ∧ (`encode-operators Π t` ∧ (`encode-all-frame-axioms Π t` ∧ (`encode-interfering-operator-exclusion Π t` ∧ (`encode-goal-state Π 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]:`

cnf (`encode-interfering-operator-pair-exclusion Π k op1 op2`) = {{
(Operator k (index (strips-problem.operators-of Π) op1))⁻¹,
(Operator k (index (strips-problem.operators-of Π) op2))⁻¹ }}

proof —
let ?ops = strips-problem.operators-of Π
have cnf (`encode-interfering-operator-pair-exclusion Π k op1 op2`) = cnf `(¬(Atom (Operator k (index ?ops op1))) ∨ ¬(Atom (Operator k (index ?ops op2))))`

unfolding `encode-interfering-operator-pair-exclusion-def` by metis
also have . . . = { C ∪ D | C D, C ∈ cnf `(¬(Atom (Operator k (index ?ops op1))))` ∧ D ∈ cnf `(¬(Atom (Operator k (index ?ops op2))))` } by simp
finally show ?thesis
by auto
qed

lemma `cnf-of-encode-interfering-operator-exclusion-is-ii [simp]:`

set [encode-interfering-operator-pair-exclusion Π k op1 op2. (op1, op2) ← filter (λ(op1, op2).
(index (strips-problem.operators-of Π) op1 ≠ index (strips-problem.operators-of Π) op2)]]

292
∧ are-operators-interfering op₁ op₂
(List.product (strips-problem.operators-of II) (strips-problem.operators-of II))
, k ← [0..<t]
= (⋃ (op₁, op₂)
∈ \{ (op₁, op₂) ∈ set (operators-of II) × set (operators-of II).
index (strips-problem.operators-of II) op₁ ≠ index (strips-problem.operators-of II) op₂
∧ are-operators-interfering op₁ op₂ }.
(λk. encode-interfering-operator-pair-exclusion II k op₁ op₂) \{ [0..<t]})

proof –
let ?ops = strips-problem.operators-of II
let ?interfering = filter (λ(op₁, op₂). index ?ops op₁ ≠ index ?ops op₂
∧ are-operators-interfering op₁ op₂) (List.product ?ops ?ops)
let ?fs = [encode-interfering-operator-pair-exclusion II k op₁ op₂.
(op₁, op₂) ← ?interfering, k ← [0..<t]]
have set ?fs = \bigcup (set
  ' (λ(op₁, op₂). map (λk. encode-interfering-operator-pair-exclusion II k op₁ op₂)
  \{ [0..<t]})
  ' (set (filter (λ(op₁, op₂). index ?ops op₁ ≠ index ?ops op₂ ∧ are-operators-interfering
  op₁ op₂)
  (List.product ?ops ?ops))))
  unfolding set-concat set-map
  by blast
  — TODO slow.
also have . . . = \bigcup ((λ(op₁, op₂).
  set (map (λk. encode-interfering-operator-pair-exclusion II k op₁ op₂) \{ [0..<t]})
  ' (set (filter (λ(op₁, op₂). index ?ops op₁ ≠ index ?ops op₂ ∧ are-operators-interfering
  op₁ op₂)
  (List.product ?ops ?ops))))
  unfolding image-comp[of
  set λ(op₁, op₂). map (λk. encode-interfering-operator-pair-exclusion II k op₁
  op₂) \{ [0..<t]}
  comp-apply
  by fast
also have . . . = \bigcup ((λ(op₁, op₂).
  (λk. encode-interfering-operator-pair-exclusion II k op₁ op₂) \{ [0..<t]})
  ' (set (filter (λ(op₁, op₂). index ?ops op₁ ≠ index ?ops op₂ ∧ are-operators-interfering
  op₁ op₂)
  (List.product ?ops ?ops))))
  unfolding set-map[of - [0..<t]] atLeastLess-than-upt[of 0 t]
  by blast
also have . . . = \bigcup ((λ(op₁, op₂).
  (λk. encode-interfering-operator-pair-exclusion II k op₁ op₂) \{ [0..<t]})
  ' (set (List.product ?ops ?ops))))
  unfolding set-filter[of λ(op₁, op₂). are-operators-interfering op₁ op₂ List.product
  ?ops ?ops]

293
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 II k op1 op2.
    (op1, op2) ← filter (λ(op1, op2).
      index (strips-problem/operators-of II) op1 ≠ index (strips-problem/operators-of
      II) op2
    ∧ are-operators-interfering op1 op2)
    (List.product (strips-problem/operators-of II) (strips-problem/operators-of
    II))
    , k ← [0..<t]]
      = (⋃ (op1, op2)
        ∈ { (op1, op2) ∈ set (strips-problem/operators-of II) × set (strips-problem/operators-of
        II).
          index (strips-problem/operators-of II) op1 ≠ index (strips-problem/operators-of
        II) op2
        ∧ are-operators-interfering op1 op2 }.
        {{(Operator k (index (strips-problem/operators-of II) op1))−1
          , (Operator k (index (strips-problem/operators-of II) op2))−1 } | k. k ∈
          {0..<t}})

proof –
  let ?ops = strips-problem/operators-of II
  let ?interfering = filter (λ(op1, op2). index ?ops op1 ≠ index ?ops op2
    ∧ are-operators-interfering op1 op2) (List.product ?ops ?ops)
  let ?fs = [encode-interfering-operator-pair-exclusion II k op1 op2.
    (op1, op2) ← ?interfering, k ← [0..<t]]
  have cnf ' set ?fs = cnf ' (⋃ (op1, op2) ∈ { (op1, op2).
    (op1, op2) ∈ set (operators-of II) × set (operators-of II) ∧ index ?ops op1 ≠
    index ?ops op2
    ∧ are-operators-interfering op1 op2 }.
    (λk. encode-interfering-operator-pair-exclusion II k op1 op2) ' {0..<t})
      unfolding cnf-of-encode-interfering-operator-exclusion-is-ii
    by blast
  also have ... = (⋃ (op1, op2) ∈ { (op1, op2).
    (op1, op2) ∈ set (operators-of II) × set (operators-of II) ∧ index ?ops op1 ≠
    index ?ops op2
    ∧ are-operators-interfering op1 op2 }.
    (λk. cnf (encode-interfering-operator-pair-exclusion II k op1 op2)) ' {0..<t})
      unfolding image-Un image-comp comp-apply
    by blast
  also have ... = (⋃ (op1, op2) ∈ { (op1, op2).
\[(op_1, op_2) \in \text{set (operators-of } \Pi) \times \text{set (operators-of } \Pi) \land \text{index } \?ops \, op_1 \neq \text{index } \?ops \, op_2 \land \text{are-operators-interfering } op_1 \, op_2 \}\]

\[(\lambda k. \{ (\text{Operator } k (\text{index } \?ops \, op_1))^{-1}, (\text{Operator } k (\text{index } \?ops \, op_2))^{-1} \} \} \cdot \{0..<t\})\]

by simp
also have \ldots = \bigcup (op_1, op_2) \in \{ (op_1, op_2) \land \text{index } \?ops \, op_1 \neq \text{index } \?ops \, op_2 \land \text{are-operators-interfering } op_1 \, op_2 \}

\[(\lambda k. \{ (\text{Operator } k (\text{index } \?ops \, op_1))^{-1}, (\text{Operator } k (\text{index } \?ops \, op_2))^{-1} \} \} \cdot \{ k | k. k \in \{0..<t\}\})\]

by blast
— TODO slow.
finally show \?thesis
unfolding operators-of-def setcompr-eq-image[of - \lambda k. k. k \in \{0..<t\}]
by force
qed

lemma cnf-of-encode-interfering-operator-exclusion-is:
\[\text{cnf (encode-interfering-operator-exclusion } \Pi \, t) = \bigcup (\text{set (operators-of } \Pi) \times \text{set (operators-of } \Pi) \land \text{index } \?ops \, op_1 \neq \text{index } \?ops \, op_2 \land \text{are-operators-interfering } op_1 \, op_2 \}

\[(\{ (\text{Operator } k (\text{index } \?ops \, op_1))^{-1}, (\text{Operator } k (\text{index } \?ops \, op_2))^{-1} \} | k. k \in \{0..<t\})\}\]

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) \leftarrow \?interfering, k \leftarrow \{0..<t\}]

have cnf (encode-interfering-operator-exclusion \Pi \, t) = \text{cnf (foldr } (\land) \, ?fs \, (\neg \bot))

unfolding encode-interfering-operator-exclusion-def
by metis
also have \ldots = \bigcup (\text{cnf } \cdot \text{set } ?fs)

unfolding cnf-foldr-and[of \, ?fs].

finally show \?thesis
unfolding cnf-of-encode-interfering-operator-exclusion-is-iii[of \, \Pi \, t]
by blast
qed

lemma cnf-of-encode-interfering-operator-exclusion-contains-clause-if:

fixes \Pi :: \text{‘variable strips-problem}
assumes \k. \k < \t
and \op_1 \in \text{set (strips-problem.operators-of } \Pi) \, \text{and } \op_2 \in \text{set (strips-problem.operators-of}
Π)
and index (strips-problem.operators-of Π) op₁ ≠ index (strips-problem.operators-of Π) op₂
and are-operators-interfering op₁ op₂
shows { (Operator k (index (strips-problem.operators-of Π) op₁))⁻¹
 , (Operator k (index (strips-problem.operators-of Π) op₂))⁻¹ } ∈ cnf (encode-interfering-operator-exclusion Π t) proof –
let ?ops = strips-problem.operators-of Π
and ?Φ_X = encode-interfering-operator-exclusion Π t
let ?Ops = { (op₁, op₂) ∈ set (operators-of Π) × set (operators-of Π).
 index ?ops op₁ ≠ index ?ops op₂ ∧ are-operators-interfering op₁ op₂ }
and ?f = λ(op₁, op₂). {{ (Operator k (index ?ops op₁))⁻¹, (Operator k (index
 ?ops op₂))⁻¹ }}
 | k. k ∈ {0..<t} }
let ?A = (∪ (op₁, op₂) ∈ ?Ops. ?f (op₁, op₂))
let ?B = ∪ ?A
and ?C = { (Operator k (index ?ops op₁))⁻¹, (Operator k (index ?ops op₂))⁻¹ }

{ have (op₁, op₂) ∈ ?Ops
 using assms(2, 3, 4, 5)
 unfolding operators-of-def
 by force
 moreover have { ?C } ∈ ?f (op₁, op₂)
 using assms(1)
 by auto
 moreover have { ?C } ∈ ?A
 using UN-iff[of ?C - ?Ops] calculation(1, 2)
 by blast

ultimately have ∃ X ∈ ?A. ?C ∈ X
 by auto
}

thus ?thesis
 unfolding cnf-of-encode-interfering-operator-exclusion-is
 using Union-iff[of ?C ?A]
 by auto

qed

lemma is-cnf-encode-interfering-operator-exclusion:

fixes Π :: 'variable strips-problem
shows is-cnf (encode-interfering-operator-exclusion Π t)
proof –
let ?ops = strips-problem.operators-of Π
let ?interfering = filter (λ(op₁, op₂). index ?ops op₁ ≠ index ?ops op₂
 ∧ are-operators-interfering op₁ op₂) (List.product ?ops ?ops)
let \(?fs = \{ encode-interfering-operator-pair-exclusion \Pi k op_1 op_2 \}.

let \(?Fs = \bigcup \{ (op_1, op_2) \in \text{set (operators-of } \Pi) \times \text{set (operators-of } \Pi), \text{are-operators-interfering op}_1 op_2 \}.\)

\((\lambda k. \text{encode-interfering-operator-pair-exclusion } \Pi k op_1 op_2) \setminus \{0..<t\})\)

{ }

fix \(f\)
assume \(f \in \text{set } ?fs\)
then have \(f \in ?Fs\)
  unfolding cnf-of-encode-interfering-operator-exclusion-is-ii
  by blast
then obtain \(op_1 op_2\)
  where \((op_1, op_2) \in \text{set (operators-of } \Pi) \times \text{set (operators-of } \Pi)\)
  and are-operators-interfering \(op_1 op_2\)
  and \(f \in (\lambda k. \text{encode-interfering-operator-pair-exclusion } \Pi k op_1 op_2) \setminus \{0..<t\}\)
  by fast
then obtain \(k\) where \(f = \text{encode-interfering-operator-pair-exclusion } \Pi k op_1 op_2\)
  by blast
then have \(f = \neg (\text{Atom (Operator } k \text{ (index } ?ops op_1)) \lor \neg (\text{Atom (Operator } k \text{ (index } ?ops op_2)))\)
  unfolding encode-interfering-operator-pair-exclusion-def
  by metis
hence \(\text{is-cnf } f\)
  by force

thus \(?thesis\)
  unfolding encode-interfering-operator-exclusion-def
  using 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 \(\text{is-cnf (} \Phi \_ \Pi t\)\)
using \(\text{is-cnf-encode-problem is-cnf-encode-interfering-operator-exclusion axsms}\)
unfolding encode-problem-with-operator-interference-exclusion-def SAT-Plan-Base.encode-problem-def
  is-cnf.simps(1)
  by blast

lemma \(\text{cnf-of-encode-problem-with-operator-interference-exclusion-structure}:\)
shows \(\text{cnf (} \Phi \_ t\Pi \subseteq \text{cnf (} \Phi \_ t\Pi t\)\)
and \(\text{cnf (} \Phi \_ t\Pi t \subseteq \text{cnf (} \Phi \_ t\Pi t\)\)
and \(\text{cnf (} \text{encode-operators } t\Pi t \subseteq \text{cnf (} \Phi \_ t\Pi t\)\)
and \(\text{cnf (} \text{encode-all-frame-axioms } t\Pi t \subseteq \text{cnf (} \Phi \_ t\Pi t\)\)
and \(\text{cnf (} \text{encode-interfering-operator-exclusion } t\Pi t \subseteq \text{cnf (} \Phi \_ t\Pi t\)\)
unfolding encode-problem-with-operator-interference-exclusion-def encode-problem-def
SAT-Plan-Base.encode-problem-def

297
By auto

**Lemma** encode-problem-with-operator-interference-exclusion-has-model-then-also-partial-encodings:

**Assumes** $A \models \Phi \Pi t$

**Shows** $A \models SAT-Plan-Base.encode-initial-state \Pi t$

and $A \models SAT-Plan-Base.encode-operators \Pi t$

and $A \models encode-interfering-operator-exclusion \Pi t$

and $A \models SAT-Plan-Base.encode-goal-state \Pi t$

**Using** `assms`

**Unfolding** encode-problem-with-operator-interference-exclusion-def encode-problem-def

SAT-Plan-Base.encode-problem-def

by `simp`

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 \neq ops \in set (subseqs (\Phi^{-1} \Pi 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 \in set (subseqs (\Phi^{-1} \Pi A t ! k))` using lemma `subseqs-distinctD`.

**Lemma** encode-problem-serializable-sound-i:

**Assumes** is-valid-problem-strips II

and $A \models \Phi \Pi t$

and $k < t$

and `ops \in set (subseqs ((\Phi^{-1} \Pi A t) ! k))`

**Shows** are-all-operators-non-interfering `ops`

**Proof** —
let \( ?\text{ops} = \text{strips-problem.operators-of } \Pi \)
and \( ?\pi = \Phi^{-1} \Pi \mathcal{A} t \)
and \( ?\Phi_X = \text{encode-interfering-operator-exclusion } \Pi t \)
let \( ?\pi_k = (?\Phi^{-1} \Pi \mathcal{A} t) ! k \)

{ 
    fix \( C \)
    assume \( C\text{-in}: C \in \text{cnf } ?\Phi_X \)
    have \( \text{cnf-semantics } \mathcal{A} \left( \text{cnf } ?\Phi_X \right) \)
        using \( \text{cnf-semantics-monotonous-in-cnfs-subsets-if } \) [OF \( \text{assms}(2) \)]
        \( \text{is-cnfs-encode-problem-with-operator-interference-exclusion } \) [OF \( \text{assms}(1) \)]
        \( \text{cnf-of-encode-problem-with-operator-interference-exclusion-structure } \) [5].
    hence \( \text{clause-semantics } \mathcal{A} C \)
        unfolding \( \text{cnf-semantics-def} \)
        using \( C\text{-in} \)
        by fast
} note \( nb_1 = \text{this} \)

{ 
    fix \( op_1, op_2 \)
    assume \( op_1 \in \text{set } ?\pi_k \) and \( op_2 \in \text{set } ?\pi_k \)
    and \( \text{index-op}_1\text{-is-not-index-op}_2: \text{index } ?\text{ops } op_1 \neq \text{index } ?\text{ops } op_2 \)
    moreover have \( \text{op}_1\text{-in}: op_1 \in \text{set } ?\text{ops and } \mathcal{A}\text{-models-op}_1: \mathcal{A} \left( \text{Operator } k \left( \text{index } ?\text{ops } op_1 \right) \right) \)
        and \( \text{op}_2\text{-in}: op_2 \in \text{set } ?\text{ops and } \mathcal{A}\text{-models-op}_2: \mathcal{A} \left( \text{Operator } k \left( \text{index } ?\text{ops } op_2 \right) \right) \)
        using \( \text{decode-plan-step-element-then-if } \) [OF \( \text{assms}(3) \)]
        calculation
        unfolding \( \text{decode-plan-def} \)
        by blast+
    moreover { 
        let \( ?C = \{ \left( \text{Operator } k \left( \text{index } ?\text{ops } op_1 \right) \right)^{-1}, \left( \text{Operator } k \left( \text{index } ?\text{ops } op_2 \right) \right)^{-1} \} \)
        assume \( \text{are-operators-interfering } \text{op}_1 \text{ op}_2 \)
        moreover have \( ?C \in \text{cnf } ?\Phi_X \)
            using \( \text{cnf-of-encode-interfering-operator-exclusion-contains-clause-if } \) [OF \( \text{assms}(3) \) \( \text{op}_1\text{-in } \text{op}_2\text{-in } \text{index-op}_1\text{-is-not-index-op}_2 \)]
            calculation
            by blast
        moreover have \( \neg \text{clause-semantics } \mathcal{A} ?C \)
            using \( \mathcal{A}\text{-models-op}_1 \mathcal{A}\text{-models-op}_2 \)
            unfolding \( \text{clause-semantics-def} \)
            by auto
        ultimately have \( \text{False} \)
            using \( nb_1 \)
            by blast
    }
    ultimately have \( \neg \text{are-operators-interfering } \text{op}_1 \text{ op}_2 \)
        by blast
} note \( nb_3 = \text{this} \)
show \( ?\text{thesis} \)
using \( \text{assms} \)

299
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-πk
       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 Π
     and A⊨ Φγ Π t
  shows is-parallel-solution-for-problem Π(Φ⁻¹ Π A t)
     and ∀ k < length (Φ⁻¹ Π A t). are-all-operators-non-interfering ((Φ⁻¹ Π A t)
               ! k)
proof –
  { 
    have A ⊨ SAT-Plan-Base.encode-initial-state Π
    qed
and \( A \models SAT-Plan-Base.\!\! encode-operators \Pi t \)
and \( A \models SAT-Plan-Base.\!\! encode-all-frame-axioms \Pi t \)
and \( A \models SAT-Plan-Base.\!\! encode-goal-state \Pi t \)
using assms(2)
unfolding encode-problem-with-operator-interference-exclusion-def
by simp+
then have \( A \models SAT-Plan-Base.\!\! encode-problem \Pi t \)
unfolding SAT-Plan-Base.\!\! encode-problem-def
by simp
\}
thus is-parallel-solution-for-problem \( \Pi^{-1} \Pi A t \)
using encode-problem-parallel-sound assms(1, 2)
unfolding decode-plan-def
by blast
next
let \(?\pi = \Phi^{-1} \Pi A t\)
\{
  \fix k
  assume \( k < t \)
  moreover have \(?\pi ! k \in \text{set} (\text{subseqs} (?\pi ! k))\)
  using subseqs-refl
  by blast
  ultimately have are-all-operators-non-interfering (?\pi ! k)
  using encode-problem-serializable-sound-i[OF assms]
  unfolding SAT-Plan-Base.\!\! decode-plan-def decode-plan-def
  by blast
\}
moreover have \( \text{length} ?\pi = t \)
unfolding SAT-Plan-Base.\!\! decode-plan-def decode-plan-def
by simp
ultimately show \( \forall k < \text{length} ?\pi.\!\! \text{are-all-operators-non-interfering} (?\pi ! k) \)
by simp
qed

8.2 Completeness

lemma encode-problem-with-operator-interference-exclusion-complete-i:
assumes is-valid-problem-strips \( \Pi \)
and is-parallel-solution-for-problem \( \Pi \pi \)
and \( \forall k < \text{length} \pi.\!\! \text{are-all-operators-non-interfering} (\pi ! k) \)
shows valuation-for-plan \( \Pi \pi \models encode-interfering-operator-exclusion \Pi (\text{length} \pi) \)
proof –
let \(?A = valuation-for-plan \Pi \pi\)
and \(?\Phi_X = encode-interfering-operator-exclusion \Pi (\text{length} \pi)\)
and \(?\ops = \text{strips-problem.operators-of} \Pi \)
and \(?t = \text{length} \pi\)
let \(?\pi = \text{trace-parallel-plan-strips} ((\Pi I)1) \pi\)
let \(?\ops = \{ (op_1, op_2).\!\! (op_1, op_2) \in \text{set} (\text{operators-of} \Pi) \times \text{set} (\text{operators-of} \Pi)\)
Π)
∧ index ?ops op1 ≠ index ?ops op2
∧ are-operators-interfering op1 op2 }
and ?f = λ(op1, op2).
\{ \{ (Operator k (index ?ops op1))^{-1}, (Operator k (index
?ops op2))^{-1} \} | k. k ∈ \{0..<length π \} \}
let ?A = ∪(if ?'Ops)
let ?B = ∪ ?A
have nb1: ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set (operators-of Π)
  using is-parallel-solution-for-problem-operator-set[OF assms(2)]
  unfolding operators-of-def
  by blast
{
  fix k op
  assume k < length π and op ∈ set (π ! k)
  hence lit-semantics ?A ((Operator k (index ?ops op))^{+}) = (k < length ?τ − 1)
    using encode-problem-parallel-complete-vi-a[OF assms(2)]
    encode-problem-parallel-complete-vi-b[OF assms(2)] initial-of-def
  by(cases k < length ?τ − 1; simp)
  note nb2 = this
}
{
  fix k op1 op2
  assume k < length π
  and op1 ∈ set (π ! k)
  and index ?ops op1 ≠ index ?ops op2
  and are-operators-interfering op1 op2
  moreover have are-all-operators-non-interfering (π ! k)
    using assms(3) calculation(1)
    by blast
  moreover have op1 ≠ op2
    using calculation(3)
    by blast
  ultimately have op2 ∉ set (π ! k)
    using are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs
    assms(3)
    by blast
  note nb3 = this
}
{
  fix C
  assume C ∈ cnf ?Ψ_X
  then have C ∈ ?B
    using cnf-of-encode-interfering-operator-exclusion-is[of Π length π]
    by argo
  then obtain C' where C' ∈ ?A and C-in: C ∈ C'
    using Union-iff[of C ?A]
    by meson
  then obtain op1 op2 where (op1, op2) ∈ set (operators-of Π) × set (operators-of
Π)

302
and index-op₁-is-not-index-op₂: index ?ops op₁ ≠ index ?ops op₂
and are-operators-interfering-op₁-op₂: are-operators-interfering op₁ op₂
and C'-in: C' ∈ \{((Operator k (index ?ops op₁))⁻¹, (Operator k (index ?ops op₂)))⁻¹\}
| k. k ∈ \{0..<\text{length} π\}
using UN-iff[of C' if ?Ops]
by blast
then obtain k where k ∈ \{0..<\text{length} π\}
and C-is: C = \{(Operator k (index ?ops op₁))⁻¹, (Operator k (index ?ops op₂))⁻¹\}
using C-in C'-in
by blast
then have k-lt-length-π: k < \text{length} π
by simp
consider (A) op₁ ∈ set (π ! k)
| (B) op₂ ∈ set (π ! k)
| (C) ¬op₁ ∈ set (π ! k) ∨ ¬op₂ ∈ set (π ! k)
by linarith
hence clause-semantics ?A C
proof (cases)
  case A
  moreover have op₂ /∈ set (π ! k)
  by simp
  moreover have ¬?A (Operator k (index ?ops op₂))
  by blast
  ultimately show ‹thesis›
  by simp
next
  case B
  moreover have op₁ /∈ set (π ! k)
  by simp
  moreover have ¬?A (Operator k (index ?ops op₁))
  by blast
  ultimately show ‹thesis›
  by simp
next
  case C
  then show ‹thesis›
  proof (rule disjE)

303
\[ \text{assume } op_1 \notin \text{ set } (\pi ! k) \]
\[ \text{then have } \neg \forall A \ (\text{Operator } k \ (\text{index } \text{ops } op_1)) \]
\[ \text{using } \text{encode-problem-parallel-complete-vi-d}(\text{OF } \text{assms}(2) \ k\text{-lt-length-}\pi] \]
\[ \text{by blast} \]
\[ \text{thus } \text{clause-semantics } (\text{valuation-for-plan } \Pi \pi) \ C \]
\[ \text{using } C\text{-is} \]
\[ \text{unfolding } \text{clause-semantics-def} \]
\[ \text{by force} \]
\[ \text{next} \]
\[ \text{assume } op_2 \notin \text{ set } (\pi ! k) \]
\[ \text{then have } \neg \forall A \ (\text{Operator } k \ (\text{index } \text{ops } op_2)) \]
\[ \text{using } \text{encode-problem-parallel-complete-vi-d}(\text{OF } \text{assms}(2) \ k\text{-lt-length-}\pi] \]
\[ \text{by blast} \]
\[ \text{thus } \text{clause-semantics } (\text{valuation-for-plan } \Pi \pi) \ C \]
\[ \text{using } C\text{-is} \]
\[ \text{unfolding } \text{clause-semantics-def} \]
\[ \text{by force} \]
\[ \text{qed} \]
\[ \text{qed} \]
\[ \{ \ (\text{Operator } k \ (\text{index } \text{ops } op_1))^{-1}, \ (\text{Operator } k \ (\text{index } \text{ops } op_2))^{-1} \ \} \]

where \( \text{ops} \equiv \Pi_\text{O} \). 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 \( A \equiv \text{valuation-for-plan } \Pi \pi \) it holds that
\[ A \models \neg \text{Atom } (\text{Operator } k \ (\text{index } \text{ops } op_2)) \]
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 ??).
**Theorem** encode-problem-serializable-complete:

**Assumes** is-valid-problem-strips Π

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

∀ k < length π. are-all-operators-non-interfering (π ! k)

**Shows** valuation-for-plan Π π |= \( \Phi \) Π (length π)

**Proof**

- Let \( ?A = \text{valuation-for-plan} \ Π \ π \)

  and \( ?\Phi_X = \text{encode-interfering-operator-exclusion} \ Π \ (\text{length} \ π) \)

- Have \( ?A = \text{SAT-Plan-Base.encode-problem} \ Π \ (\text{length} \ π) \)

  using assms(1, 2) encode-problem-parallel-complete

  by auto

  moreover have \( ?A = ?\Phi_X \)

  using encode-problem-with-operator-interference-exclusion-complete-i[OF assms]

  **Ultimately show** ?thesis

  unfolding encode-problem-with-operator-interference-exclusion-def encode-problem-def

  SAT-Plan-Base.encode-problem-def

  by force

**QED**

**Lemma** encode-problem-forall-step-decoded-plan-is-serializable-i:

**Assumes** is-valid-problem-strips Π

and \( A \models \Phi \ Π \ t \)

**Shows** (Π) \( G \subseteq_m \text{execute-serial-plan} \ ((Π)I) \ (\text{concat} \ (Φ^{-1} \ Π \ A \ t)) \)

**Proof**

- Let \( ?G = (Π)_G \)

  and \( ?I = (Π)_I \)

  and \( ?π = Φ^{-1} \ Π \ A \ t \)

- Let \( ?π' = \text{concat} \ (Φ^{-1} \ Π \ A \ t) \)

  and \( ?τ = \text{trace-parallel-plan-strips} \ ?I \ ?π \)

  and \( ?σ = \text{map (decode-state-at Π A)} [0..<\text{Suc (length} \ ?π)] \)

  \{ fix k 

  assume k-lt-length-π: k < length ?π

  moreover have \( A = SAT-Plan-Base.encode-problem Π t \)

  using assms(2)

  unfolding encode-problem-with-operator-interference-exclusion-def

  encode-problem-def SAT-Plan-Base.encode-problem-def

  by simp

  moreover have length ?σ = length ?τ

  using encode-problem-parallel-correct-vii assms(1) calculation

  unfolding decode-state-at-def decode-plan-def initial-of-def

  by fast

  **Ultimately have** k < length ?τ - 1 and k < t

  unfolding decode-plan-def SAT-Plan-Base.decode-plan-def

  by force+ \}

**Note** nb = this
{ 
  have ?G ⊆ m execute-parallel-plan ?I ?π 
    using encode-problem-serializable-sound assms 
  unfolding is-parallel-solution-for-problem-def decode-plan-def 
  goal-of-def initial-of-def 
  by blast 
  hence ?G ⊆ m last (trace-parallel-plan-strips ?I ?π) 
    using execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace 
    by fast 
}

moreover { 
  fix k 
  assume k-lt-length-π: k < length ?π 
  moreover have k < length ?τ − 1 and k < t 
    using nb calculation 
    by blast+ 
  moreover have are-all-operators-applicable (?τ ! k) (?π ! k) 
    and are-all-operator-effects-consistent (?π ! k) 
    using trace-parallel-plan-strips-operator-preconditions calculation(2) 
    by blast+ 
  moreover have are-all-operators-non-interfering (?π ! k) 
    using encode-problem-serializable-sound(2)[OF assms(1, 2)] k-lt-length-π 
    by blast 
  ultimately have are-all-operators-applicable (?τ ! k) (?π ! k) 
    and are-all-operator-effects-consistent (?π ! k) 
    and are-all-operators-non-interfering (?π ! 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 Π :: 'variable strips-problem 
  shows list-all (λop. ListMem op (strips-problem.operators-of Π)) 
    (concat (?τ⁻¹ Π A t)) 
proof – 
  let ?π = Φ⁻¹ Π A t 
  let ?π' = concat ?π 

  { 
    have set ?π' = ∪(set i. (∪ k < t. { decode-plan' Π A k }))) 
    unfolding decode-plan-def decode-plan-set-is set-concat 
    by auto 
    also have ... = ∪(∪ k < t. { set (decode-plan' Π A k) }) 
    by blast 
  }
finally have \( \text{set} \ ?\pi' = (\bigcup k < t. \text{set} (\text{decode-plan}' \Pi A k)) \)
by blast
\}
note \( nb = this \)
\{
fix \( op \)
assume \( op \in \text{set} \ ?\pi' \)
then obtain \( k \) where \( k < t \) and \( op \in \text{set} (\text{decode-plan}' \Pi A k) \)
using \( nb \)
by blast
moreover have \( op \in \text{set} (\text{decode-plan} \Pi A t ! k) \)
using \text{calculation}
unfolding \text{decode-plan-def SAT-Plan-Base.decode-plan-def} 
by simp
ultimately have \( op \in \text{set} (\text{operators-of} \Pi) \)
using \text{decode-plan-step-element-then(I)}
unfolding \text{operators-of-def decode-plan-def} 
by blast
\}
thus ?thesis
unfolding \text{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 \( \pi = \Phi^{-1} \Pi A t \) 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 straight forward, 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

\[ \text{13} \] These propositions are shown in lemmas \text{encode_problem_forall_step_decoded_plan_is_serializable_ii} and \text{encode_problem_forall_step_decoded_plan_is_serializable_i} which have been omitted for brevity.
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 \( \mathcal{A} \models \Phi \forall \Pi t \)
- **Shows** is-serial-solution-for-problem \( \Pi (\text{concat} (\Phi^{-1} \Pi \mathcal{A} t)) \)
- **Using** encode-problem-forall-step-decoded-plan-is-serializable-i(OF assms)
- encode-problem-forall-step-decoded-plan-is-serializable-ii
- unfolding is-serial-solution-for-problem-def goal-of-def
- initial-of-def decode-plan-def
- **By** blast

**Theory** SAT-Solve-SAS-Plus

- imports SAS-Plus-STRIPS
  - SAT-Plan-Extensions
- **Begin**

9 SAT-Solving of SAS+ Problems

**Lemma** sas-plus-problem-has-serial-solution-iff-i:

- **Assumes** is-valid-problem-sas-plus \( \Psi \)
- and \( \mathcal{A} \models \Phi \forall (\varphi \Psi) t \)
- **Shows** is-serial-solution-for-problem \( \Psi [\varphi_{O^{-1}} \Psi \text{ op. op} \leftarrow \text{concat} (\Phi^{-1} (\varphi \Psi) \mathcal{A} t)] \)
- **Proof**
  - let \( ?\Pi = \varphi \Psi \)
  - and \( ?\pi' = \text{concat} (\Phi^{-1} (\varphi \Psi) \mathcal{A} t) \)
  - let \( ?\psi = [\varphi_{O^{-1}} \Psi \text{ op. op} \leftarrow ?\pi'] \)
  - **Have** is-valid-problem-strips ?\Pi
    - **Using** is-valid-problem-sas-plus-then-strips-transformation-too(OF assms(1)).
  - **Moreover Have** STRIPS-Semantics.is-serial-solution-for-problem ?\Pi ?\pi'
    - **Using** calculation serializable-encoding-decoded-plan-is-serializable[OF
      - assms(2)]
    - **Unfolding** decode-plan-def
    - **By** simp
  - **Ultimately Have** SAS-Plus-Semantics.is-serial-solution-for-problem \( \Psi ?\psi \)
    - **Using** assms(1) serial-strips-equivalent-to-serial-sas-plus
    - **By** blast
  - **Thus** ?thesis
    - **Using** serial-strips-equivalent-to-serial-sas-plus[OF assms(1)]
    - **By** blast
- **Qed**
lemma sas-plus-problem-has-serial-solution-if-iff:
assumes is-valid-problem-sas-plus $\Psi$
and is-serial-solution-for-problem $\Psi$ $\psi$
and $h = \text{length } \psi$
shows $\exists A. (A \models \Phi \phi \ (\phi \Psi) h)$
proof
let $\Pi = \phi \Psi$
and $\pi = \phi P \Psi \ (\text{embed } \psi)$
let $\forall A = \text{valuation-for-plan } ?\pi$
let $?i = \text{length } \psi$

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 $ops' \text{ where } ops' = ?\pi ! k$
by simp
moreover have $ops' \in \text{set } ?\pi$
using calculation nth-mem
by blast
moreover have $?\pi = [[\phi \Psi \ op. op \leftarrow ops], ops \leftarrow \text{embed } \psi]]$
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 $ops$
where $ops' = [\phi \Psi \ op. op \leftarrow ops]$
and $ops \in \text{set } (\text{embed } \psi)$
using calculation(3, 4)
by auto
moreover have $ops \in \{ \ [op] \mid op. op \in \text{set } \psi \}$
using calculation(6) set-of-embed-is
by blast
moreover obtain $op$
where $ops = [op]$ and $op \in \text{set } \psi$

309
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))
qed

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 \( \mathcal{A} \) and a
hypothesized plan length \( t \) exist s.t.

\[ \mathcal{A} \models \Phi \forall (\varphi \Psi) \ t \]

for the serializable SATPlan encoding of the corresponding STRIPS problem
\( \Phi \varphi \Psi \ t \) exist.

**Theorem** sas-plus-problem-has-serial-solution-iff:

assumes is-valid-problem-sas-plus \( \Psi \)
s-shows (\exists \psi. is-serial-solution-for-problem \( \Psi \) \( \psi \)) \iff (\exists \mathcal{A} \ t. \mathcal{A} \models \Phi \varphi \Psi \ t)

using sas-plus-problem-has-serial-solution-iff-i OF assms
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 \( \equiv (\SAS-Plus-Representation.sas-plus-operator.precondition-of = [])\)

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 \( \Pi \equiv (\SAS-Plus-Representation.sas-plus-problem.variables-of = \SAS-Plus-Representation.sas-plus-problem.variables-of \Pi, \Pi)\)
\[
\text{lemma sasp-noops-in-noop-problem: } \text{set } (\text{replicate } n \text{ empty-sasp-action}) \subseteq \text{set } (\text{SAS-Plus-Representation.operators-of } (\text{prob-with-noop } \Pi)) \\
\text{by } (\text{induction } n) (\text{auto simp: prob-with-noop-def})
\]

\[
\text{definition rem-noops } \equiv \text{filter } (\lambda \, \text{op. op \neq \text{empty-sasp-action}})
\]

\[
\text{lemma sasp-filter-empty-action: } \text{execute-serial-plan-sas-plus } s (\text{rem-noops } \pi s) = \text{execute-serial-plan-sas-plus } s \pi s \\
\text{by } (\text{induction } \pi s \text{ arbitrary: } s) \\
(\text{auto simp: empty-sasp-action-def rem-noops-def})
\]

\[
\text{lemma noops-sound: } \text{SAS-Plus-Semantics.is-serial-solution-for-problem } (\text{prob-with-noop } \Psi) \pi s \implies \text{SAS-Plus-Semantics.is-serial-solution-for-problem } \Psi (\text{rem-noops } \pi s) \\
\text{by } (\text{induction } \pi s) \\
(\text{fastforce simp: SAS-Plus-Semantics.is-serial-solution-for-problem-def insert list.pred-set} \\
\text{prob-with-noop-def ListMem-iff rem-noops-def} \\
\text{sasp-filter-empty-action[unfolded empty-sasp-action-def rem-noops-def]} \\
\text{empty-sasp-action-def}+)
\]

\[
\text{lemma noops-valid: } \text{is-valid-problem-sas-plus } \Psi \implies \text{is-valid-problem-sas-plus } (\text{prob-with-noop } \Psi) \\
\text{by } (\text{auto simp: is-valid-problem-sas-plus-def prob-with-noop-def Let-def empty-sasp-action-def is-valid-operator-sas-plus-def list.pred-set})
\]

\[
\text{lemma sas-plus-problem-has-serial-solution-iff-1':} \\
\text{assumes } \text{is-valid-problem-sas-plus } \Psi \\
\text{and } A \models \Phi_\tau (\varphi (\text{prob-with-noop } \Psi)) t \\
\text{shows } \text{SAS-Plus-Semantics.is-serial-solution-for-problem } \Psi (\text{rem-noops})
\]
\[
\begin{aligned}
& (\text{map} \ (\lambda \varphi \cdot \varphi^{-1} \ (\text{prob-with-noop} \ \Psi)) \ \text{op}) \\
& (\text{concat} \ (\Phi^{-1} \ (\varphi \ (\text{prob-with-noop} \ \Psi))) \ A \ t)) \\
\end{aligned}
\]

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 \ \psi
and length \psi \leq h
shows \exists A. (A \models \Phi \ (\varphi \ (\text{prob-with-noop} \ \Psi)) \ h)
using assms
by (fastforce
intro!: assms noops-valid noops-complete
SAS-Plus-Semantics.is-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.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 ⇒ bool
where is-standard-effect ≡ \lambda (pre, -, -, -). \ pre = []

definition is-standard-operator :: ast-operator ⇒ bool
where is-standard-operator ≡ \lambda (-, -, effects, -). \ list-all is-standard-effect effects

fun rem-effect-implicit-pres :: ast-effect ⇒ ast-effect where
rem-effect-implicit-pres (preconds, v, implicit-pre, eff) = (preconds, v, None, eff)

fun rem-implicit-pres :: ast-operator ⇒ ast-operator where
rem-implicit-pres (name, preconds, effects, cost) =
(name, (implicit-pres effects) @ preconds, map rem-effect-implicit-pres effects, cost)

312
fun rem-implicit-pres-ops :: ast-problem ⇒ ast-problem where
rem-implicit-pres-ops (vars, init, goal, ops) = (vars, init, goal, map rem-implicit-pres ops)

definition consistent-map-lists xs1 xs2 ≡ (∀ (x1,x2) ∈ set xs1. ∀ (y1,y2)∈ set xs2. x1 = y1 → x1 = y2)

lemma map-add-comm: (∀ x. x ∈ dom m1 ∧ x ∈ dom m2 ⇒ m1 x = m2 x) ⇒ m1 ++ m2 = m2 ++ m1
by (fastforce simp add: map-add-def split: option.splits)

lemma first-map-add-submap: (∀ x. x ∈ dom m1 ∧ x ∈ dom m2 ⇒ m1 x = m2 x) ⇒ m1 ++ m2 ⊆ m x
using map-add-le-mapE map-add-comm
by force

lemma subsuming-states-map-add:
(∀ x. x ∈ dom m1 ∩ dom m2 ⇒ m1 x = m2 x) ⇒ m1 ++ m2 ⊆ m s ←→ (m1 ⊆ m s ∧ m2 ⊆ m s)
by (auto simp: map-add-le-mapI intro: first-map-add-submap map-add-le-mapE)

lemma consistent-map-lists:
[distinct (map fst (xs1 @ xs2)); x ∈ dom (map-of xs1) ∩ dom (map-of xs2)] ⇒
(map-of xs1) x = (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:
distinct (map fst (xs @ ys)) ⇒
(map-of (xs @ ys)) ⊆ m s ←→ ((map-of ys) ⊆ m s ∧ (map-of xs) ⊆ 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 ≡ (case op of (name, pres, effs, cost) ⇒
distinct (map fst (pres @ (implicit-pres effs))))
∧ consistent-map-lists pres (implicit-pres effs))

definition consistent-pres-op’ where
consistent-pres-op’ op ≡ (case op of (name, pres, effs, cost) ⇒
consistent-map-lists pres (implicit-pres effs))

lemma consistent-pres-op-then’: consistent-pres-op op ⇒ consistent-pres-op’ op
by (auto simp add: consistent-pres-op' def consistent-pres-op-def)

lemma rem-implicit-pres-ops-valid-states:
  ast-problem.valid-states (rem-implicit-pres-ops prob) = ast-problem.valid-states 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
  ast-problem.range-of-var-def ast-problem.numVals-def)

lemma rem-implicit-pres-ops-lookup-op-None:
  ast-problem.lookup-operator (vars, init, goal, ops) name = None ⇔
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name = None

lemma rem-implicit-pres-ops-lookup-op-Some-1:
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name = Some op
  ⇒ ∃ op'. ast-problem.lookup-operator (vars, init, goal, ops) name = Some op'
  ∧ (op = rem-implicit-pres op')
  implicit-pres-empty image-def)

lemma rem-implicit-pres-ops-lookup-op-Some-1'::
  ast-problem.lookup-operator prob name = Some (n,p,v,e) ⇒
  ast-problem.lookup-operator (rem-implicit-pres-ops prob) name = Some (rem-implicit-pres (n,p,v,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
  ⇒ ∃ op'. ast-problem.lookup-operator (vars, init, goal, ops) name = Some op'
  ∧ (op = rem-implicit-pres op')
  implicit-pres-empty image-def)

lemma rem-implicit-pres-ops-lookup-op-Some-2'::
  ast-problem.lookup-operator (rem-implicit-pres-ops prob) name = Some (n,p,e,c)
  ⇒ ∃ op'. ast-problem.lookup-operator prob name = Some op'
  ∧ ((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':
  \( s \in \text{ast-problem.subsuming-states \ prob \ ps} = (s \in (\text{ast-problem.valid-states \ prob}) \land ps \subseteq_m s) \)
  by (auto simp add: ast-problem.subsuming-states-def)

lemma rem-implicit-pres-ops-enabled-1:
  \( \left( (\forall \text{op. op} \in \text{set (ast-problem.ast}^{\delta} \text{ prob)} \implies \text{consistent-pres-op op}); \right) \)
  \( \text{ast-problem.enabled \ prob \ name \ s} \)
  \( \implies \text{ast-problem.enabled (rem-implicit-pres-ops \ prob) \ name \ s} \)
  by (fastforce simp: ast-problem.enabled-def rem-implicit-pres-ops-valid-states subsuming-states-def')

lemma lookup-Some-in-delta: lookup-operator \( \pi = \text{Some \ op} = \implies \text{op} \in \text{set ast}^{\delta} \)
  by (auto simp: find-Some-iff in-set-conv-nth lookup-operator-def)

end

lemma rem-implicit-pres-ops-enabled-2:
  \( (\forall \text{op. op} \in \text{set (ast-problem.ast}^{\delta} \text{ prob)} \implies \text{consistent-pres-op op}); \)
  \( \text{ast-problem.enabled (rem-implicit-pres-ops \ prob) \ name \ s} \)
  \( \implies \text{ast-problem.enabled \ prob \ name \ s} \)
  using assms[OF ast-problem.lookup-Some-in-delta, unfolded consistent-pres-op-def]
apply(auto simp: subsuming-states-append rem-implicit-pres-ops-valid-states subsuming-states-def')
  dest!: rem-implicit-pres-ops-append-op-Some-2'
  split: option.splits)
using subsuming-states-map-add consistent-map-lists
apply (metis Map.map-add-comm dom-map-of-conv-image-fst map-add-le-mapE)
using map-add-le-mapE by blast

lemma rem-implicit-pres-ops-enabled:
  \( (\forall \text{op. op} \in \text{set (ast-problem.ast}^{\delta} \text{ prob)} \implies \text{consistent-pres-op op}); \)
  \( \text{ast-problem.enabled (rem-implicit-pres-ops \ prob) \ name \ s = \text{ast-problem.enabled \ prob \ name \ s}} \)
  using rem-implicit-pres-ops-enabled-1 rem-implicit-pres-ops-enabled-2
  by blast

context ast-problem
begin
lemma std-eff-enabled[simp]:
  \(\text{is-standard-operator (name, pres, effs, layer)} \implies s \in \text{valid-states} \implies (\text{filter (eff-enabled s) effs) = effs}\)
  
  \text{by (induction effs) (auto simp: is-standard-operator-def is-standard-effect-def eff-enabled-def subsuming-states-def)}

end

lemma is-standard-operator-rem-implicit:
  \(\text{is-standard-operator (n,p,vp,v)} \implies \text{is-standard-operator (rem-implicit-pres (n,p,vp,v))}\)
  
  \text{by (induction vp) (auto simp: is-standard-operator-def is-standard-effect-def)}

lemma is-standard-operator-rem-implicit-pres-ops:
\[
[(\forall \text{op} \in \text{set (ast-problem.astδ (a,b,c,d)) \implies \text{is-standard-operator op});
\begin{align*}
\text{op} &\in \text{set (ast-problem.astδ (rem-implicit-pres-ops (a,b,c,d)))} \\
\implies &\text{is-standard-operator op}
\end{align*}
]\]

\text{by (induction d) (fastforce simp add: ast-problem.astδ-def image-def dest: is-standard-operator-rem-implicit)+}

lemma is-standard-operator-rem-implicit-pres-ops':
\[
[(\forall \text{op} \in \text{set (ast-problem.astδ (rem-implicit-pres-ops prob))};
\begin{align*}
\text{op} &\in \text{set (ast-problem.astδ prob)} \implies \text{is-standard-operator op} \\
\implies &\text{is-standard-operator op}
\end{align*}
]\]

\text{apply (cases prob) using is-standard-operator-rem-implicit-pres-ops by blast}

lemma in-rem-implicit-pres-δ:
\(\text{op} \in \text{set (ast-problem.astδ prob)} \implies \text{rem-implicit-pres op} \in \text{set (ast-problem.astδ (rem-implicit-pres-ops prob))}\)

\text{by (auto simp add: ast-problem.astδ-def)}

lemma rem-implicit-pres-ops-execute:
\text{assumes}
\(\begin{align*}
\forall \text{op} \in \text{set (ast-problem.astδ prob)} \implies \text{is-standard-operator op} \end{align*}\) \text{and}
\(\text{s} \in \text{ast-problem.valid-states prob}\)
\text{shows ast-problem.execute (rem-implicit-pres-ops prob) name s = ast-problem.execute prob name s}\n
\text{proof–}
\text{have (n,ps,es,c) }\in \text{set (ast-problem.astδ prob)} \implies
\begin{align*}
\text{(filter (ast-problem.eff-enabled prob s) es) } = \text{es for n ps es c} \\
\text{using assms(2)}
\end{align*}

\text{by (auto simp add: ast-problem.std-eff-enabled dest!: assms(1))}
\text{moreover have (n,ps,es,c) }\in \text{set (ast-problem.astδ prob)} \implies
\begin{align*}
\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 ps es c} \\
\text{using assms}
\end{align*}

\text{by (fastforce simp add: ast-problem.std-eff-enabled rem-implicit-pres-ops-valid-states)}
dest!: is-standard-operator-rem-implicit-pres-ops'
dest: in-rem-implicit-pres-δ

moreover have map-of (map ((λ(x,-,v). (x,v)) o rem-effect-implicit-pres) effs) =
map-of (map (λ(x,-,v). (x,v)) effs) for effs
by (induction effs) auto
ultimately show thesis
by (auto simp add: ast-problem.execute-def rem-implicit-pres-ops-lookup-op-Some-1'
split: option.splits
qed

lemma rem-implicit-pres-ops-path-to:
wf-ast-problem prob
(∀ op. op ∈ set (ast-problem.astδ prob) ⇒ consistent-pres-op op)
(∀ op. op ∈ set (ast-problem.astδ prob) ⇒ is-standard-operator op)
s ∈ ast-problem.valid-states prob
⇒ ast-problem.path-to (rem-implicit-pres-ops prob) s π s s' = ast-problem.path-to prob s π s'
by (induction π s arbitrary: s)
(auto simp: 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) =
est-problem.astG prob
apply(cases prob)
by (auto simp add: ast-problem.astG-def)

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.astI-def subsuming-states-def')

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
(∀ op. op ∈ set (ast-problem.astδ prob) ⇒ consistent-pres-op op)
(∀ op. op ∈ set (ast-problem.astδ prob) ⇒ is-standard-operator op)
shows ast-problem.valid-plan (rem-implicit-pres-ops prob) π s = ast-problem.valid-plan prob π 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]:
ast-problem.numVals (rem-implicit-pres-ops prob) x = ast-problem.numVals prob x
by (cases prob) (simp add: ast-problem.numVals-def ast-problem.astDom-def)

lemma in-implicit-pres:
(x, a) ∈ set (implicit-pres effs) ⟹ (∃ epres v vp. (epres,x,vp,v) ∈ set effs ∧ vp = Some a)
by (induction effs) (fastforce simp: implicit-pres-def image-def split: if-splits)+

lemma pair4-eqD: (a1,a2,a3,a4) = (b1,b2,b3,b4) ⟹ a3 = b3
by simp

lemma rem-implicit-pres-ops-wf-partial-state:
ast-problem.wf-partial-state (rem-implicit-pres-ops prob) s =
ast-problem.wf-partial-state prob s
by (auto simp: ast-problem.wf-partial-state-def)

lemma rem-implicit-pres-wf-operator:
assumes consistent-pres-op op
shows ast-problem.wf-operator prob op
proof−
obtain name pres effs cost where op: op = (name, pres, effs, cost)
by (cases op)
hence asses: consistent-pres-op (name, pres, effs, cost)
ast-problem.wf-operator prob (name, pres, effs, cost)
using assms
by auto
hence distinct (map fst ((implicit-pres effs) @ pres))
by (simp only: consistent-pres-op-def) auto
moreover have x < ast-problem.numVars (rem-implicit-pres-ops prob)
v < ast-problem.numVars (rem-implicit-pres-ops prob) x
if (x,v) ∈ set ((implicit-pres effs) @ pres) for x v
using that asses
ultimately have ast-problem.wf-partial-state (rem-implicit-pres-ops prob) ((implicit-pres effs) @ pres)
by (auto simp only: ast-problem.wf-partial-state-def)
moreover have \((\text{map } (\lambda (-, v, -). v) \text{ effs}) = (\text{map } (\lambda (-, v, -). v) (\text{map } \text{ rem-effect-implicit-pres effs}))\)

by auto

hence \((\text{distinct } (\text{map } (\lambda (-, v, -). v) (\text{map } \text{ rem-effect-implicit-pres effs}))\)

using assms(2)

by (auto simp only: op ast-problem.wf-operator-def rem-implicit-pres.simps dest!: pair4-eqD)

moreover have \((\exists \text{ epres } x \text{ v}. (\text{epres,x,v})\in \text{set effs}) \iff (\text{epres,x,}\text{None,v})\in \text{set } (\text{map } \text{rem-effect-implicit-pres effs})\)

for epres x v

by force

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

using assms(2)


qed

lemma rem-implicit-pres-ops-inδD: \(\text{op} \in \text{set } (\text{ast-problem.astδ } (\text{rem-implicit-pres-ops prob})) \implies (\exists \text{ op′. } \text{op′} \in \text{set } (\text{ast-problem.astδ prob}) \land \text{op} = \text{rem-implicit-pres op′})\)

by (cases prob) (force simp: ast-problem.astδ-def)

lemma rem-implicit-pres-ops-well-formed:

assumes \((\forall \text{ op. } \text{op} \in \text{set } (\text{ast-problem.astδ prob}) \implies \text{consistent-pres-op op})\)

shows ast-problem.well-formed (rem-implicit-pres-ops prob)

proof−

have \(\text{map \ fst } (\text{ast-problem.astδ } (\text{rem-implicit-pres-ops prob})) = \text{map \ fst } (\text{ast-problem.astδ prob})\)

by (cases prob) (auto simp: ast-problem.astδ-def)

thus \(?\text{thesis}\)

using assms


qed

definition is-standard-effect' :: ast-effect ⇒ bool

where is-standard-effect' ≡ \(\lambda (\text{pre}, -, \text{vpre}, -). \text{pre} = [] \land \text{vpre} = \text{None}\)

definition is-standard-operator' :: ast-operator ⇒ bool

where is-standard-operator' ≡ \(\lambda (\_, -, \text{effects}, -). \text{list-all is-standard-effect' effects}\)

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 vs) (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':
  \( (\forall \text{op} \in \text{set (ast-problem.ast} \delta (\text{vs, I, G, ops})) \implies \text{is-standard-operator op}) \)
  \( \implies \pi \in \text{set (ast-problem.ast} \delta (\text{rem-implicit-pres-ops (vs, I, G, ops)}) \implies \text{is-standard-operator'} \pi \)
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: \( \forall \pi \in \text{set ast} \delta. \text{is-standard-operator'} \pi \)

context ast-problem
begin

definition abs-ast-variable-section = [0..<(length astDom)]

definition abs-range-map :: (nat \to nat list)
where abs-range-map \equiv
  map-of (zip abs-ast-variable-section
    (map ((\lambda \text{vals. [0..<length vals]} o snd o snd)
      astDom))

end

context abs-ast-prob
begin

lemma is-valid-vars-1: astDom \( \neq [] \implies \text{abs-ast-variable-section} \neq [] \)
  by (simp add: abs-ast-variable-section-def)

end

lemma upt-eq-Nil-conv'[simp]: \( [[i..<j]] = (j = 0 \lor j \leq i) \)
  by (induct j) simp-all

lemma map-of-zip-map-Some:
  \( v < \text{length xs} \)
  \( \implies (\text{map-of (zip [0..<length xs] (map f xs)) v} = \text{Some (f (xs ! v))} \)
by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)

lemma map-of-zip-Some:
  \( v < \text{length xs} \)
  \( \implies (\text{map-of (zip [0..<length xs] xs) v} = \text{Some (xs ! v)} \)
by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)
lemma in-set-zip-lengthE:
\[(x,y) \in \text{set}(\text{zip} \; [0..\langle \text{length} \; xs \rangle] \; xs) \implies (\langle x < \text{length} \; xs ; \; xs ! x = y \rangle \implies R) \implies R\]
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 \((\lambda v. \text{abs-range-map} \; v \neq \text{None}) \; \text{abs-ast-variable-section}\)
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 \equiv \text{map-of} (\text{zip} \; [0..\langle \text{length} \; \text{astI} \rangle] \; \text{astI})
end

context abs-ast-prob
begin

lemma valid-abs-init-1:
abs-ast-initial-state \(v\) \(\neq\) \(\text{None}\) \(\iff\) \(v\) \(\in\) \text{set} \(\text{abs-ast-variable-section}\)
by (simp add: abs-ast-variable-section-def numVars-def wf-initial(1) abs-ast-initial-state-def)

lemma abs-range-map-Some:
shows \(v \in \text{set} \; \text{abs-ast-variable-section} \iff\) \(\text{abs-range-map} \; v = \text{Some} \; [0..\langle \text{length} \; (\text{snd} \; \text{snd} \; \text{astDom} \; ! \; v))])\)
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:
\(v \in \text{set} \; \text{abs-ast-variable-section} \iff v < \text{length} \; \text{astDom}\)
by (simp add: abs-ast-variable-section-def)

lemma [simp]: \(v < \text{length} \; \text{astDom} \Rightarrow \text{abs-ast-initial-state} \; v = \text{Some} \; (\text{astI} \; ! \; v)\)
using wf-initial(1)[simplified numVars-def, symmetric]
by (auto simp add: map-of-zip-Some abs-ast-initial-state-def split: prod.splits)

lemma [simp]: \(v < \text{length} \; \text{astDom} \Rightarrow \text{astI} \; ! \; v < \text{length} \; (\text{snd} \; (\text{snd} \; (\text{astDom} \; ! \; v)))\)
using wf-initial(1)[simplified numVars-def, symmetric] wf-initial
by (auto simp add: numVals-def abs-ast-initial-state-def split: prod.splits)

lemma [intro!]: \(v \in \text{set} \; \text{abs-ast-variable-section} \Rightarrow x < \text{length} \; (\text{snd} \; (\text{snd} \; (\text{astDom} \; ! \; v)))\) \(\Rightarrow\)
\(x \in \text{set} \; (\text{the} \; (\text{abs-range-map} \; v))\)
using abs-range-map-Some

321
lemma [intro]: \( x < \text{length } \text{astDom} \Rightarrow \text{astI} \; ! \; x < \text{length} \; (\text{snd} \; (\text{snd} \; (\text{astDom} \; ! \; x))) \)
using \( \text{wf-initial[unfolded numVars-def numVals-def]} \)
by auto

lemma [simp]: \( \text{abs-ast-initial-state } v = \text{Some } a \Rightarrow a < \text{length} \; (\text{snd} \; (\text{snd} \; (\text{astDom} \; ! \; v))) \)
by (auto simp add: \( \text{abs-ast-initial-state-def \; wf-initial(1)[unfolded numVars-def numVals-def, symmetric]} \)
elim!: \( \text{in-set-zip-lengthE} \))

lemma valid-\( \text{abs-init-2} \): \( \text{abs-ast-initial-state } v \neq \text{None} \Rightarrow (\text{the}(\text{abs-ast-initial-state } v)) \in \text{set} \; (\text{the}(\text{abs-range-map } v)) \)
using valid-\( \text{abs-init-1} \)
by auto

end

context ast-problem
begin

definition abs-ast-goal :: nat-sas-plus-state
where abs-ast-goal \equiv \text{map-of } \text{astG}
end

context abs-ast-prob
begin

lemma [simp]: \( \text{wf-partial-state } s \Rightarrow (v, a) \in \text{set } s \Rightarrow v \in \text{set} \; \text{abs-ast-variable-section} \)
by (auto simp add: \( \text{wf-partial-state-def \; abs-ast-variable-section-def \; numVars-def} \)
split: prod.splits)

lemma valid-\( \text{abs-goal-1} \): \( \text{abs-ast-goal } v \neq \text{None} \Rightarrow v \in \text{set} \; \text{abs-ast-variable-section} \)
using \( \text{wf-goal} \)
by (auto simp add: \( \text{abs-ast-goal-def \; dest!: \; map-of-SomeD} \))

lemma in-\( \text{abs-rangeI} \): \( \text{wf-partial-state } s \Rightarrow (v, a) \in \text{set } s \Rightarrow (a \in \text{set} \; (\text{the}(\text{abs-range-map } v))) \)
by (auto simp add: \( \text{abs-range-map-Some \; wf-partial-state-def \; numVals-def \; split: prod.splits} \))

lemma valid-\( \text{abs-goal-2} \): \( \text{abs-ast-goal } v \neq \text{None} \Rightarrow (\text{the}(\text{abs-ast-goal } v)) \in \text{set} \; (\text{the}(\text{abs-range-map } v)) \)
using \( \text{wf-goal} \)
by (auto simp add: \( \text{map-of-SomeD \; weak-map-of-SomeI \; abs-ast-goal-def \; intro!:} \))
in-abs-rangeI

end

context ast-problem
begin
definition abs-ast-operator :: ast-operator ⇒ nat-sas-plus-operator
  where abs-ast-operator ≡ λ(name, preconditions, effects, cost).
    precondition-of = preconditions,
    effect-of = [(v, x). (- v, - x) ← 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[introl]:
  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[introl]:
  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);
  \∀ epres x vp v. (epres,x,vp,v)∈ set effs =⇒ wf-partial-state epres;
  \∀ epres x vp (epres,x,vp,v)∈ set effs =⇒ x < numVars;
  \∀ epres x vp v. (epres,x,vp,v)∈ set effs =⇒ v < numVals x;
\( \forall \text{epres } x, \text{vp } v. (\text{epres}, x, \text{vp}, v) \in \text{set effs} \implies \) 

\[
\text{case } \text{vp of None } \Rightarrow \text{True } | \text{ Some } v \Rightarrow v < \text{numVals } x \]

\[\implies P)\]

\( \implies P\)

**unfolding** \text{wf-operator-def}

**by** (auto split: prod.splits)

**lemma** \text{abs-valid-operator-3’}:

\text{wf-operator} (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies 

\[\text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{map } (\lambda(-, v, -, a). (v, a)) \text{ eff})\]

**by** (fastforce simp add: list.pred-set ListMem-iff abs-ast-variable-section-def image-def numVars-def

\text{elim1: wf-operator-effE split: prod.splits})

**lemma** \text{abs-valid-operator-3[introl]}:

\text{wf-operator } op \implies 

\[\text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{effect-of } (\text{abs-ast-operator } op))\]

**by** (cases op, simp add: abs-ast-operator-def abs-valid-operator-3’)

**lemma** \text{wf-abs-eff}: \text{wf-operator} (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies \text{wf-partial-state} (\text{map } (\lambda(-, v, -, a). (v, a)) \text{ eff})

**by** (elim \text{wf-operator-effE}, induction eff)

\text{fastforce simp add: wf-partial-state-def image-def o-def split: prod.split-asm}+ 

**lemma** \text{abs-valid-operator-4’}:

\text{wf-operator} (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies 

\[\text{list-all } (\lambda(v, a). (\text{abs-range-map } v \neq \text{None}) \land \text{ListMem } a (\text{the } (\text{abs-range-map } v))) (\text{map } (\lambda(-, v, -, a). (v, a)) \text{ eff})\]

**apply** (subt list.pred-set ListMem-iff)+ 

**apply** (drule wf-abs-eff)

**by** (metis (mono-tags, lifting) abs-rangeI case-prodI2 in-abs-rangeI)

**lemma** \text{abs-valid-operator-4[introl]}:

\text{wf-operator } op \implies 

\[\text{list-all } (\lambda(v, a). (\exists y. \text{abs-range-map } v = \text{Some } y) \land \text{ListMem } a (\text{the } (\text{abs-range-map } v)))\]

**using** \text{abs-valid-operator-4’}

**by** (cases op, simp add: abs-ast-operator-def)

**lemma** \text{consistent-list-set}: \text{wf-partial-state } s \implies 

\[\text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v’, a’). v \neq v’ \lor a = a’) s) s\]

**by** (auto simp add: list.pred-set wf-partial-state-def eq-key-imp-eq-value split: prod.splits)

**lemma** \text{abs-valid-operator-5’}:

\text{wf-operator} (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies
\[
\text{list-all} \ (\lambda (v, a). \ \text{list-all} \ (\lambda (v', a'). \ v \neq v' \lor a = a') \ \text{pre}) \ \text{pre}
\]
apply(drule wf-operator-preD)
by (intro consistent-list-set)

lemma abs-valid-operator-5[introl]:
wf-operator op \implies
\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)))
(\text{precondition-of} \ (\text{abs-ast-operator} \ op))
using abs-valid-operator-5
by (cases op, simp add: abs-ast-operator-def)

lemma consistent-list-set-2: distinct (map fst s) \implies
\text{list-all} \ (\lambda (v, a). \ \text{list-all} \ (\lambda (v', a'). \ v \neq v' \lor a = a') \ 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':
asumes \text{wf-operator} \ (\text{name}, \ \text{pre}, \ \text{eff}, \ \text{cost})
shows \text{list-all} \ (\lambda (v, a). \ \text{list-all} \ (\lambda (v', a'). \ v \neq v' \lor a = a') \ (\text{map} \ (\lambda (\cdot, v, \cdot, a). \ (v, a)) \ \text{eff}))
(\text{map} \ (\lambda (\cdot, v, v, \cdot, a). \ (v, a)) \ \text{eff})
proof-
have *: \text{map} \ \text{fst} \ (\text{map} \ (\lambda (\cdot, v, \cdot, a). \ (v, a)) \ \text{eff}) = \ (\text{map} \ (\lambda (\cdot, v, v, a). \ v) \ \text{eff})
by (induction \ \text{eff}) \ \text{auto}
show \ ?thesis
using assms
apply(elim wf-operator-effE)
apply(intro consistent-list-set-2)
by (subst *)
qed

lemma abs-valid-operator-6[introl]:
\text{wf-operator} \ op \ \implies
\text{list-all} \ (\lambda (v, a). \ \text{list-all} \ (\lambda (v', a'). \ v \neq v' \lor a = a') \ (\text{effect-of} \ (\text{abs-ast-operator} \ op)))
(\text{effect-of} \ (\text{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 \equiv \ [\text{abs-ast-operator} \ op. \ op \leftarrow \ \ast \delta]
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 |
end

context abs-ast-prob begin

lemma \[ simp \]: \( op \in \text{set ast} \implies (\text{is-valid-operator-sas-plus abs-prob} \ (\text{abs-ast-operator}\ op)) \)
apply(cases op)
apply(subst is-valid-operator-sas-plus-def Let-def)+
using wf-operators(2)
by(fastforce simp add: abs-prob-def)+

lemma abs-ast-operator-section-valid:
  list-all (is-valid-operator-sas-plus abs-prob) abs-ast-operator-section
by (auto simp: abs-ast-operator-section-def list.pred-set)

lemma abs-prob-valid: is-valid-problem-sas-plus abs-prob
using valid-abs-goal-1 valid-abs-goal-2 valid-abs-init-1 is-valid-vars-2
  abs-ast-operator-section-valid\[unfolded abs-prob-def\]
by (auto simp add: is-valid-problem-sas-plus-def Let-def ListMem-iff abs-prob-def)

definition abs-ast-plan
:: SASP-Semantics.plan \(\Rightarrow\) nat-sas-plus-plan
where abs-ast-plan \(\pi\ s\)
  \(\equiv\) map (abs-ast-operator o the o lookup-operator) \(\pi\ s\)

lemma \[ simp \]: enabled \(\pi\ s\) \(\implies\) lookup-operator \(\pi\) = Some (name, pres, effs, layer)
\(\implies\) implicit-pres effs = []
apply(induction effs)
by (auto simp add: is-standard-operator-′-def implicit-pres-def is-standard-effect-′-def)

lemma \[ simp \]: enabled \(\pi\ s\) \(\implies\) lookup-operator \(\pi\) = Some (name, pres, effs, layer)
\(\implies\) is-standard-operator′ (name, pres, effs, layer) \(\implies\)
  (filter (eff-enabled s) effs) = effs
by(auto simp add: enabled-def is-standard-operator′-def eff-enabled-def is-standard-effect′-def
filter-id-conv list.pred-set)

lemma effs-eq-abs-effs: (effect-of (abs-ast-operator (name, pres, effs, layer))) =
  (map (\(\lambda\)(-,-,-,v). \(\langle\ x,v\rangle\)) effs)
by (auto simp add: abs-ast-operator-def
split: option.splits prod.splits)

lemma exect-eq-abs-execute:
\[
\text{enabled } \pi \text{ s; lookup-operator } \pi = \text{Some (name, preconds, effs, layer);} \\
\text{is-standard-operator}'(\text{name, preconds, effs, layer}) \implies \\
exect \pi \text{ s} = (\text{execute-operator-sas-plus s (abs-ast-operator o the o lookup-operator)} \pi)
\]
using effs-eq-abs-effs
by (auto simp add: execute-def execute-operator-sas-plus-def)

lemma enabled-then-sas-applicable:
\[
\text{enabled } \pi \text{ s} \implies \text{SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator o the o lookup-operator)} \pi
\]
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:
\[
\forall \pi \in \text{set } \pi \text{ s. lookup-operator } \pi \neq \text{None} \implies \\
\text{path-to s } \pi \text{ s s'} \implies \\
\text{s' } \subseteq \text{m execute-serial-plan-sas-plus s (abs-ast-plan } \pi \text{ s)}
\]
proof (induction \pi \text{ s arbitrary: } s s')
case (Cons a \pi \text{ s})
then show ?case
by (force simp: exect-eq-abs-execute abs-ast-plan-def lookup-Some-in
\no-cond-effs
\dest: enabled-then-sas-applicable)
qed (auto simp: execute-serial-plan-sas-plus-def abs-ast-plan-def)

lemma map-of-eq-None-iff:
\[
(\text{None } = \text{map-of } \text{xyz } x) = (x \notin \text{fst } (\text{set } \text{xyz}))
\]
by (induct \text{xyz}) simp-all

lemma [simp]: I = abs-ast-initial-state
apply (intro HOL.extr)
by (auto simp: map-of-eq-None-iff set-map[symmetric] I-def abs-ast-initial-state-def
\map-of-\text{zip-Some}
\dest: map-of-SomeD)

lemma [simp]: \forall \pi \in \text{set } \pi \text{ s. lookup-operator } \pi \neq \text{None} \implies \\
\text{op } \subseteq \text{ set (abs-ast-plan } \pi \text{ s)} \implies \text{op } \in \text{ set abs-ast-operator-section}
by (induction \pi \text{ s}) (auto simp: abs-ast-plan-def abs-ast-operator-section-def lookup-Some-in)

end

context ast-problem
begin

lemma path-to-then-lookup-Some:
\[
(\exists s' \in G. \text{ path-to s } \pi \text{ s s'}) \implies (\forall \pi \in \text{ set } \pi \text{ s}.
\]

327
lemma valid-plan-then-lookup-Some: valid-plan π s ⇒ (∀ π ∈ set s. lookup-operator π ≠ None)
  using path-to-then-lookup-Some
  by (simp add: valid-plan-def)

end

context abs-ast-prob
begin

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(v, a). ([][], v, None, a)) (effect-of op) = effs

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')
  ⇒ pres = pres'
  by (auto simp: lookup-action-def abs-ast-operator-def dest: find-Some)

lemma [simp]: is-standard-operator' (name, pres, effs, layer) ⨪ 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) ⨪ (name, pres, effs, layer) ∈ set astδ ⨪ lookup-action (abs-ast-operator (name, pres, effs, layer)) = Some (name', pres', effs', layer')
  ⇒ effs = effs'
  by (auto simp: is-standard-operator'-def lookup-action-def dest: find-Some)

lemma mem-find-Some: x ∈ set xs ⇒ P x ⇒ ∃ 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)

  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 ⇒ ∃ 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 ⇒ 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) \rightarrow name = name'  
  using lookup-operator-wf(2)  
  by fastforce

lemma eq-name-eq-pres: (name, pres, effs, layer) \in set ast\delta \rightarrow (name, pres', effs', layer') \in set ast\delta \rightarrow \text{pres} = \text{pres}'  
  using eq-key-imp-eq-value[OF wf-operators(1)]  
  by auto

lemma eq-name-eq-effs: name = name' \rightarrow (name, pres, effs, layer) \in set ast\delta \rightarrow (name', pres', effs', layer') \in set ast\delta \rightarrow \text{effs} = \text{effs}'  
  using eq-key-imp-eq-value[OF wf-operators(1)]  
  by auto

lemma is-applicable-then-subsumes:  
\[
  s \in \text{valid-states} \implies \text{SAS-Plus-Representation.is-operator-applicable-in } s \text{ (abs-ast-operator (name, pres, effs, layer))} \implies \text{SAS-Plus-Representation.is-operator-applicable-in-def abs-ast-operator-def}
\]

lemma eq-name-eq-pres': \[
\begin{align*}
  [s \in \text{valid-states} ; \text{is-standard-operator'} (name, pres, effs, layer); (name, pres, effs, layer) \in set ast\delta ; \\
  \text{lookup-operator} ((\text{fst o the o lookup-action o abs-ast-operator}) (name, pres, effs, layer)) = \text{Some (name', pres', effs', layer')}] \implies \text{pres} = \text{pres}'
\end{align*}
\]
  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:  
\[
\begin{align*}
  [s \in \text{valid-states} ; \text{ast-op} \in set ast\delta ; \\
  \text{SAS-Plus-Representation.is-operator-applicable-in } s \text{ (abs-ast-operator ast-op)}; \\
  \text{lookup-operator} ((\text{fst o the o lookup-action o abs-ast-operator}) \text{ ast-op}) = \text{Some (name, pres, effs, layer)}] \implies \text{s} \in \text{subsuming-states (map-of pres)}
\end{align*}
\]
  apply(cases ast-op)  
  using eq-name-eq-pres' is-applicable-then-subsumes no-cond-effs 
  by fastforce

lemma is-applicable-then-enabled-3:  
\[
\begin{align*}
  [s \in \text{valid-states} ;
\end{align*}
\]
lookup-operator ((fst o the o lookup-action o abs-ast-operator) ast-op) = Some
(name, pres, effs, layer)
⇒ s ∈ 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-δ
simp: subsuming-states-def)

lemma is-applicable-then-enabled:
[s ∈ valid-states; ast-op ∈ set astδ; SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator ast-op)]
⇒ enabled ((fst o the o lookup-action o abs-ast-operator) 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) ∈ set astδ
s ∈ 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)]

lemma std-eff-enabled[simp]:
is-standard-operator′ (name, pres, effs, layer) ⇒ s ∈ valid-states ⇒ (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 ∈ valid-states; ast-op ∈ set astδ; SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator ast-op)]
⇒ execute ((fst o the o lookup-action o abs-ast-operator) ast-op) s =
execute-operator-sas-plus s (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) =

331
lemma exec-serial-then-path-to':
  \[
  \begin{align*}
  (s \in \text{valid-states}; \\
  \forall op \in \text{set ops}. \exists \text{ast-op} \in \text{set ast}. \text{op} = \text{abs-ast-operator ast-op}; \\
  (\text{sat-preconds-as s ops}) \implies \\
  \text{path-to s ((map (fst o the o lookup-action) ops) (execute-serial-plan-sas-plus s ops))} \\
  \end{align*}
  \]

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-condless: 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 \in \text{set (rem-condless-ops s ops)} \implies x \in \text{set ops}
  by (induction ops arbitrary; s) auto

lemma exec-serial-then-path-to:
  \[
  \begin{align*}
  (s \in \text{valid-states}; \\
  \forall op \in \text{set ops}. \exists \text{ast-op} \in \text{set ast}. \text{op} = \text{abs-ast-operator ast-op}; \\
  (\text{sat-preconds-as s ops}) \implies \\
  \text{path-to s ((map (fst o the o lookup-action) ops) o rem-condless-ops s ops)} \\
  \text{(execute-serial-plan-sas-plus s ops)} \\
  \end{align*}
  \]

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-condless])
lemma is-serial-solution-then-abstracted:
is-serial-solution-for-problem abs-prob ops
  \implies \forall op \in set ops. \exists ast-op \in set ast' 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)

lemma lookup-operator-works-1': lookup-action op = Some \pi' \implies \exists op. lookup-operator (fst \pi') = op
  using lookup-operator-works-1 by auto

lemma is-serial-sol-then-valid-plan-1:
is-serial-solution-for-problem abs-prob ops;
\pi \in set ((map (fst o the o lookup-action) o rem-condless-ops I) ops) \implies
lookup-operator \pi \neq None
using std-lookup-action lookup-operator-works-1 no-cond-effs
by (fastforce dest!: set-rem-condlessD is-serial-solution-then-abstracted
  simp: valid-plan-def list.pred-set ListMem-iff)

lemma is-serial-sol-then-valid-plan-2:
is-serial-solution-for-problem abs-prob ops \implies
(\exists s' \in G. path-to I ((map (fst o the o lookup-action) o rem-condless-ops I) ops)
  s')
using I-valid
by (fastforce intro: path-to-pres-valid exec-serial-then-path-to
  intro!: bexI [where x = execute-serial-plan-sas-plus I ops]
  dest: is-serial-solution-then-abstracted
  simp: list.pred-set ListMem-iff abs-ast-operator-section-def
  G-def subsuming-states-def is-serial-solution-for-problem-def
  abs-prob-def abs-ast-goal-def)+
end

cellent ast-problem
begin

definition decode-abs-plan \equiv (map (fst o the o lookup-action) o rem-condless-ops I)
end

cellent abs-ast-prob
begin

theorem is-serial-sol-then-valid-plan:
is-serial-solution-for-problem abs-prob ops \implies
valid-plan (decode-abs-plan ops)
using is-serial-sol-then-valid-plan-1 is-serial-sol-then-valid-plan-2
by (simp add: valid-plan-def decode-abs-plan-def)

end

end


begin

11.3 SAT encoding works for Fast-Downward’s representation

context abs-ast-prob

begin

theorem is-serial-sol-then-valid-plan-encoded:
  \( \mathcal{A} \models \Phi_\varphi (\varphi (\text{prob-with-noop abs-prob})) t \longrightarrow \)
  valid-plan
  (decode-abs-plan
   (rem-noops
    (map (\lambda op. \varphi^{-1} (\text{prob-with-noop abs-prob}) op)
      (concat (\Phi^{-1} (\varphi (\text{prob-with-noop abs-prob}) \mathcal{A} t))))))

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 \( \pi s = \text{length (abs-ast-plan } \pi s) \)
  by (auto simp: abs-ast-plan-def)

theorem valid-plan-then-is-serial-sol-encoded:
  valid-plan \( \pi s \Rightarrow \text{length } \pi s \leq h \Rightarrow \exists A. \mathcal{A} \models \Phi_\varphi (\varphi (\text{prob-with-noop abs-prob})) \)

h
  apply (subst (asm) length-abs-ast-plan)

by (fastforce intro!: sas-plus-problem-has-serial-solution-iff-ii' 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 $gtD$: $(\forall m. n = Suc m \land l \leq m) 
\quad \text{by (induction } n) \ \text{auto}$

locale cnf-to-dimacs = 
  fixes $h :: nat$ and $n\text{-ops :: nat}$
begin 

fun var-to-dimacs where 
  var-to-dimacs $(\text{Operator } t \ \text{k}) = 1 + t + k \ast h$
| var-to-dimacs $(\text{State } t \ \text{k}) = 1 + n\text{-ops} \ast h + t + k \ast (h)$

definition dimacs-to-var where 
  dimacs-to-var $v \equiv$
  if $v < 1 + n\text{-ops} \ast h$ then
    $(\text{Operator } ((v - 1) \mod (h))) (\text{((v - 1) \div (h))))$
  else
    $(\text{let } k = ((v - 1) - n\text{-ops} \ast h) \text{ in}$
    $(\text{State } (k \mod (h))) (k \div (h)))$

fun valid-state-var where 
  valid-state-var $(\text{Operator } t \ \text{k}) \longleftrightarrow t < h \land k < n\text{-ops}$
| valid-state-var $(\text{State } t \ \text{k}) \longleftrightarrow t < h$

lemma State-works:
valid-state-var $(\text{State } t \ \text{k}) 
\quad \text{dimacs-to-var } (\text{var-to-dimacs } (\text{State } t \ \text{k})) =$
\quad $(\text{State } t \ \text{k})$
\quad \text{by (induction } k) \ \text{(auto simp add: dimacs-to-var-def add.left-commute Let-def)}$

lemma Operator-works:
valid-state-var $(\text{Operator } t \ \text{k}) 
\quad \text{dimacs-to-var } (\text{var-to-dimacs } (\text{Operator } t \ \text{k})) =$
\quad $(\text{Operator } t \ \text{k})$
\quad \text{by (induction } k) \ \text{(auto simp add: algebra-simps dimacs-to-var-def gr0-conv-Suc
nat-le-iff-add dest!: $gtD$)}$

lemma sat-plan-to-dimacs-works:
valid-state-var $sv 
\quad \text{dimacs-to-var } (\text{var-to-dimacs } sv) = sv$
\quad apply(cases $sv$)
\quad using State-works Operator-works
\quad \text{by auto}$

end

lemma changing-atoms-works:
$(\forall x. P \ x \equiv (f \ o \ g) \ x = x) 
\quad \text{by (induction } phi) \ \text{auto}$

335
lemma changing-atoms-works':
\[ M \circ g \models \phi \iff M \models \text{map-formula } g \phi \]
by (induction \( \phi \)) auto

context cnf-to-dimacs
begin

lemma sat-plan-to-dimacs:
\[ (\forall sv. sv \in \text{atoms } \text{sat-plan-formula} \Rightarrow \text{valid-state-var } sv) \Rightarrow M \models \text{sat-plan-formula} \]
\[ \iff M \circ \text{dimacs-to-var} \models \text{map-formula } \text{var-to-dimacs } \text{sat-plan-formula} \]
by (auto intro!: changing-atoms-works\[ P = \text{valid-state-var} \] simp: sat-plan-to-dimacs-works)

lemma dimacs-to-sat-plan:
\[ M \circ \text{var-to-dimacs} \models \text{sat-plan-formula} \iff M \models \text{map-formula } \text{var-to-dimacs } \text{sat-plan-formula} \]
using changing-atoms-works'.

end

locale sat-solve-sasp = abs-ast-prob \( \Pi + \) cnf-to-dimacs Suc h Suc (length \( \delta \))
for \( \Pi \) h
begin

lemma encode-initial-state-valid:
\[ sv \in \text{atoms } (\text{encode-initial-state } \text{Prob}) \Rightarrow \text{valid-state-var } sv \]
by (auto simp add: encode-state-variable-def Let-def encode-initial-state-def split: sat-plan-variable.splits bool.splits)

lemma length-operators:
\[ \text{length } (\text{operators-of } (\varphi (\text{prob-with-noop abs-prob}))) = \text{Suc } (\text{length } \delta) \]

lemma encode-operator-effect-valid-1:
\[ t < h \Rightarrow \text{op} \in \text{set } (\text{operators-of } (\varphi (\text{prob-with-noop abs-prob}))) \]
\[ \forall sv \in \text{atoms } \left( \bigwedge (\text{map } (\lambda v. \neg (\text{Atom } (\text{Operator } t (\text{index } (\text{operators-of } (\varphi (\text{prob-with-noop abs-prob})))) \text{op})))) \right) \]
\[ \lor \text{Atom } (\text{State } (\text{Suc } t) (\text{index } vs v))) \Rightarrow \text{valid-state-var } sv \]
using length-operators
by (induction \( \text{asses} \)) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)

lemma encode-operator-effect-valid-2:
\[ t < h \Rightarrow \text{op} \in \text{set } (\text{operators-of } (\varphi (\text{prob-with-noop abs-prob}))) \]
\[ \forall sv \in \text{atoms } \left( \bigwedge (\text{map } (\lambda v. \neg (\text{Atom } (\text{Operator } t (\text{index } (\text{operators-of } (\varphi (\text{prob-with-noop abs-prob})))) \text{op})))) \right) \]
\[ \lor \text{Atom } (\text{State } (\text{Suc } t) (\text{index } vs v))) \Rightarrow \text{valid-state-var } sv \]
using length-operators
by (induction \( \text{asses} \)) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)
abs-prob)) \implies
sv \in \text{atoms}
(\bigwedge (\lambda v. \\
\neg (\text{Atom} (\text{Operator} \ t \ (\text{index} \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob})))) \op))) \\
\lor \neg (\text{Atom} (\text{State} \ (\text{Suc} \ t \ (\text{index} \ vs \ v))))
\text{assess}) \implies
valid-state-var \ sv
\text{using length-operators}
\text{by (induction assess)} \ (\text{auto simp: simp add: cnf-to-dimacs.valid-state-var.simps})
\end

\textbf{lemma atoms-And-append:} \ \text{atoms} (\bigwedge (as1 @ as2)) = \text{atoms} (\bigwedge as1) \cup \text{atoms} (\bigwedge as2) 
\text{by (induction as1) auto}

\textbf{context sat-solve-sasp}
\textbf{begin}

\textbf{lemma encode-operator-effect-valid:}
sv \in \text{atoms} \ (\text{encode-operator-effect} \ (\varphi \ (\text{prob-with-noop abs-prob})) \ t \ op) \implies
\ t < h \implies op \in \text{set (operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \implies
valid-state-var \ sv
\end

\textbf{lemma foldr-And:} foldr (\bigwedge) as (\neg \bot) = (\bigwedge as) 
\text{by (induction as) auto}

\textbf{context sat-solve-sasp}
\textbf{begin}

\textbf{lemma encode-all-operator-effects-valid:}
\ t < \text{Suc} \ h \implies
\ sv \in \text{atoms} \ (\text{encode-all-operator-effects} \ (\varphi \ (\text{prob-with-noop abs-prob})) \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob})))) \ t \implies
valid-state-var \ sv
\text{unfolding encode-all-operator-effects-def foldr-And}
\text{by (force simp add: encode-operator-effect-valid)}

\textbf{lemma encode-operator-precondition-valid-1:}
\ t < h \implies op \in \text{set (operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \implies
\ sv \in \text{atoms}
(\bigwedge (\lambda v. \\
\neg (\text{Atom} (\text{Operator} \ t \ (\text{index} \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob})))) \op))) \lor \text{Atom} (\text{State} \ t \ (f \ v)))

337
\[
\begin{aligned}
\text{asses} & \implies \\
\text{valid-state-var sv}
\end{aligned}
\]

**using** length-operators

**by** (induction asses) (auto simp: simp-to-dimacs.valid-state-var.simps)

**lemma** encode-operator-precondition-valid:

\[
sv \in \text{atoms} \ (\text{encode-operator-precondition} \ (\varphi \ (\text{prob-with-noop abs-prob})) \ t \ op) \implies \\
t < h \implies \text{op} \in \text{set} \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \implies \\
\text{valid-state-var sv}
\]

**by** (force simp: encode-operator-precondition-def Let-def

introl: encode-operator-precondition-valid-1)

**lemma** encode-all-operator-preconditions-valid:

\[
t < \operatorname{Suc} h \implies \\
sv \in \text{atoms} \ (\text{encode-all-operator-preconditions} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \ t) \implies \\
\text{valid-state-var sv}
\]

**unfolding** encode-all-operator-preconditions-def foldr-And

**by** (force simp add: encode-operator-precondition-valid)

**lemma** encode-operators-valid:

\[
sv \in \text{atoms} \ (\text{encode-operators} \ (\varphi \ (\text{prob-with-noop abs-prob})) \ t) \implies \\
t < \operatorname{Suc} h \implies \\
\text{valid-state-var sv}
\]

**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'1:

\[
l < h \implies \\
\text{set deleting-operators} \subseteq \text{set} \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \implies \\
sv \in \text{atoms} \\
\ \\
(\neg \text{Atom} \ (\text{State} \ t \ \text{v-idx}) \\
\lor \text{Atom} \ (\text{State} \ (\operatorname{Suc} t) \ \text{v-idx}) \\
\lor \bigvee \ (\text{map} \ \lambda \text{op}. \text{Atom} \ (\text{Operator} \ t \ (\text{index} \ (\text{operators-of} \ (\varphi \ (\text{prob-with-noop abs-prob})))) \ \text{op}))) \\
\text{deleting-operators} \implies \\
\text{valid-state-var sv}
\]

**by** (induction deleting-operators) (auto simp: length-operators[symmetric] cnf-to-dimacs.valid-state-var.simps)

**lemma** encode-negative-transition-frame-axiom-valid:

\[
sv \in \text{atoms} \ (\text{encode-negative-transition-frame-axiom} \ (\varphi \ (\text{prob-with-noop abs-prob}))) \\
t v) \implies \ t < h \implies \\
\text{valid-state-var sv}
\]

**unfolding** encode-negative-transition-frame-axiom-def Let-def

**apply**(intro encode-negative-transition-frame-axiom'[of l])

**by** auto

**lemma** encode-positive-transition-frame-axiom-valid:
\[ sv \in \text{atoms} (\text{encode-positive-transition-frame-axiom} (\varphi (\text{prob-with-noop abs-prob}))) \]
\[ t \Rightarrow t < h \Rightarrow \text{valid-state-var} sv \]

**unfolding** \text{encode-positive-transition-frame-axiom-def} \text{Let-def} 

**apply** \text{intro encode-negative-transition-frame-axiom\{of \[t\]\}} 

**by** \text{auto} 

**lemma** \text{encode-all-frame-axioms-valid}: 
\[ sv \in \text{atoms} (\text{encode-all-frame-axioms} (\varphi (\text{prob-with-noop abs-prob})) t) \Rightarrow t < \text{Suc} h \Rightarrow \text{valid-state-var} sv \]

**unfolding** \text{encode-all-frame-axioms-def} \text{Let-def} \text{atoms-And-append} 

**by** \text{force simp add: encode-negative-transition-frame-axiom-valid encode-positive-transition-frame-axiom-valid} 

**lemma** \text{encode-goal-state-valid}: 
\[ sv \in \text{atoms} (\text{encode-goal-state} \varphi t) \Rightarrow t < \text{Suc} h \Rightarrow \text{valid-state-var} sv \]

**unfolding** \text{encode-goal-state-def} 

**using** \text{encode-initial-state-valid encode-operators-valid encode-all-frame-axioms-valid encode-goal-state-valid} 

**by** \text{fastforce} 

**lemma** \text{encode-interfering-operator-pair-exclusion-valid}: 
\[ sv \in \text{atoms} (\text{encode-interfering-operator-pair-exclusion} (\varphi (\text{prob-with-noop abs-prob})) t \in \text{set} (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \Rightarrow \text{valid-state-var} sv \]

**by** \text{auto simp add: encode-interfering-operator-pair-exclusion-def Let-def length-operators[symmetric] cnf-to-dimacs.valid-state-var.simps} 

**lemma** \text{encode-interfering-operator-exclusion-valid}: 
\[ sv \in \text{atoms} (\text{encode-interfering-operator-exclusion} (\varphi (\text{prob-with-noop abs-prob})) t) \Rightarrow t < \text{Suc} h \Rightarrow \text{valid-state-var} sv \]

**unfolding** \text{encode-interfering-operator-exclusion-def} \text{Let-def foldr-And} 

**by** \text{force simp add: encode-interfering-operator-pair-exclusion-valid} 

**lemma** \text{encode-problem-with-operator-interference-exclusion-valid}: 
\[ sv \in \text{atoms} (\text{encode-problem-with-operator-interference-exclusion} (\varphi (\text{prob-with-noop abs-prob}))) \Rightarrow \text{valid-state-var} sv \]

**unfolding** \text{encode-problem-with-operator-interference-exclusion-def} \text{Let-def foldr-And} 

**by** \text{force simp add: encode-interfering-operator-pair-exclusion-valid}
encode-interfering-operator-exclusion-valid

by fastforce

lemma planning-by-cnf-dimacs-complete:
valid-plan π s \implies length π s \leq h \implies
\exists M. M \models map-formula var-to-dimacs (Φ ∀ (ϕ (prob-with-noop abs-prob)) h)

using valid-plan-then-is-serial-sol-encoded

by meson

lemma planning-by-cnf-dimacs-sound:
A \models map-formula var-to-dimacs (Φ ∀ (ϕ (prob-with-noop abs-prob)) t) \implies
valid-plan
(decode-abs-plan
(rem-noops
(map (λop. ϕ₀⁻¹ (prob-with-noop abs-prob) op)
(concat (Φ⁻¹ (ϕ (prob-with-noop abs-prob)) (A o var-to-dimacs) t)))))

using changing-atoms-works'

by (fastforce intro: is-serial-sol-then-valid-plan-encoded)

end

12.1 Going from Formulae 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 lists of integers.

fun disj-to-dimacs :: nat formula \Rightarrow int list where
  disj-to-dimacs (ϕ₁ ∨ ϕ₂) = disj-to-dimacs ϕ₁ @ disj-to-dimacs ϕ₂
  | disj-to-dimacs ⊥ = []
  | disj-to-dimacs (Not ⊥) = [−1::int, 1::int]
  | disj-to-dimacs (Atom v) = [int v]
  | disj-to-dimacs (Not (Atom v)) = [−(int v)]

fun cnf-to-dimacs :: nat formula \Rightarrow int list list where
  cnf-to-dimacs (ϕ₁ ∧ ϕ₂) = cnf-to-dimacs ϕ₁ @ cnf-to-dimacs ϕ₂
  | cnf-to-dimacs d = [disj-to-dimacs d]

definition dimacs-lit-to-var l ≡ nat (abs l)

definition find-max (xs::nat list)≡ (fold max xs 1)

lemma find-max-works:
x ∈ set xs \Rightarrow x ≤ find-max xs (is ?P \implies ?Q)

proof—
  have x ∈ set xs \Rightarrow (x::nat) ≤ (fold max xs m) for m
  unfolding max-def
  apply (induction xs arbitrary: m rule: rev-induct)
  using nat-le-linear

340
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
formula-vars (\(\bot\)) = [] |
formula-vars (\(\text{Atom} \ k\)) = [k] |
formula-vars (\(\text{Not} \ F\) = formula-vars \(\text{F}\) |
formula-vars (\(\text{And} \ F \ G\) = formula-vars \(\text{F}\) @ formula-vars \(\text{G}\) |
formula-vars (\(\text{Imp} \ F \ G\) = formula-vars \(\text{F}\) @ formula-vars \(\text{G}\) |
formula-vars (\(\text{Or} \ F \ G\) = formula-vars \(\text{F}\) @ formula-vars \(\text{G}\)

lemma atoms-formula-vars: atoms \(\text{f}\) = set (formula-vars \(\text{f}\))
  by (induction \(\text{f}\)) auto

lemma max-var: \(v \in \text{atoms} \ (f::\text{nat} \ \text{formula}) \Longrightarrow v \leq \text{find-max} \ (\text{formula-vars} \ f)\)
  using find-max-works
  by (simp add: atoms-formula-vars)

definition dimacs-max-var cs ≡ find-max (map (find-max o (map (nat o abs))) cs)

lemma fold-max-ge: \(b \leq a \Longrightarrow (b::\text{nat}) \leq \text{fold} \ (\lambda x \ m. \ \text{if} \ m \leq x \ \text{then} \ x \ \text{else} \ m) \ ys \ a\)
  by (induction ys arbitrary: a b) auto

lemma find-max-append: find-max (xs @ ys) = max (find-max xs) (find-max ys)
  apply (simp only: Max.set-eq-fold[symmetric] append-Cons[symmetric] set-append find-max-def)
  by (metis List.finite-set Max.union Un-absorb Un-insert-left Un-insert-right list.distinct(1) list.simps(15) set-empty)

definition dimacs-model::int list ⇒ int list list ⇒ bool where
  dimacs-model \(\text{ls} \ \text{cs}\) ≡ (\(\forall c \in \text{set} \ \text{cs}\). (\(\exists l \in \text{set} \ \text{ls}. \ l \in \text{set} \ c\)) ∧
  distinct (map dimacs-lit-to-var \(\text{ls}\))

fun model-to-dimacs-model where
  model-to-dimacs-model \(\text{M} \ (v\#\ \text{vs}\) = (if \(\text{M} \ v\) then \(\text{int} \ v\) else - (\(\text{int} \ v\)) #
  (model-to-dimacs-model \(\text{M} \ \text{vs}\) |
  model-to-dimacs-model - [] = []

lemma model-to-dimacs-model-append:
  set (model-to-dimacs-model \(\text{M} \ (\text{vs} @ \text{vs}')\)) = set (model-to-dimacs-model \(\text{M} \ \text{vs}\) ⊖
  set (model-to-dimacs-model \(\text{M} \ \text{vs}'\))
  by (induction \(\text{vs}\)) auto

lemma upt-append-sing: \(\text{xs} @ [x] = [a..<\text{n-vars}] \Longrightarrow a < \text{n-vars} \Longrightarrow (\text{xs} = [a..<\text{n-vars} - 1] ∧ x = \text{n-vars} - 1 ∧ \text{n-vars} > 0)\)
  by (induction \(\text{n-vars}\)) auto
lemma upt-eqD: \( \text{upt } a \ b = \text{upt } a \ b' \implies (b = b' \lor b' \leq a \lor b \leq a) \)
by (induction b) (auto dest: upt-append-sing split: if-splits)

lemma pos-in-model: \( M \ n \implies 0 < n \implies n < \text{n-vars} \implies \text{int } n \in \text{set } (\text{model-to-dimacs-model } M \ [1..<\text{n-vars}]) \)
by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma neg-in-model: \( \neg M \ n \implies 0 < n \implies n < \text{n-vars} \implies -\text{(int } n) \in \text{set } (\text{model-to-dimacs-model } M \ [1..<\text{n-vars}]) \)
using pos-in-model neg-in-model
by (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma in-model: \( 0 < n \implies n < \text{n-vars} \implies \text{int } n \in \text{set } (\text{model-to-dimacs-model } M \ [1..<\text{n-vars}]) \lor -\text{(int } n) \in \text{set } (\text{model-to-dimacs-model } M \ [1..<\text{n-vars}]) \)
using pos-in-model neg-in-model
by (auto simp add: le-less model-to-dimacs-model-append)

lemma model-to-dimacs-model-all-vars:
(\( \forall v \in \text{atoms } f \. \ 0 < v \land v < \text{n-vars} \) \implies \text{is-cnf } f \implies M \models f \implies 
(\( \forall n < \text{n-vars} \. \ 0 < n \implies (\text{int } n) \in \text{set } (\text{model-to-dimacs-model } M \ [(1::\text{nat})..<\text{n-vars}]) \)
\lor 
(\( \neg (\text{int } n) \in \text{set } (\text{model-to-dimacs-model } M \ [(1::\text{nat})..<\text{n-vars}]) \))
)
using in-model neg-in-model pos-in-model
by (auto simp add: le-less model-to-dimacs-model-append)

lemma cnf-And: \( \text{set } (\text{cnf-to-dimacs } (f1 \land f2)) = \text{set } (\text{cnf-to-dimacs } f1) \cup \text{set } (\text{cnf-to-dimacs } f2) \)
by auto

lemma [simp]: \( \text{disj-to-dimacs } (f1 \lor f2) = (\text{disj-to-dimacs } f1) @ (\text{disj-to-dimacs } f2) \)
by auto

lemma one-always-in:
\( I < \text{n-vars} \implies I \in \text{set } (\text{model-to-dimacs-model } M \ [(1..<\text{n-vars}]) \lor -I \in \text{set } (\text{model-to-dimacs-model } M \ [(1..<\text{n-vars}]) \)
by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma [simp]: \( \text{atoms } (f1 \lor f2) = \text{atoms } f1 \cup \text{atoms } f2 \)
by auto

lemma isdisj-disjD: \( \text{is-disj } (f1 \lor f2) \implies \text{is-disj } f1 \land \text{is-disj } f2 \)
by (cases f1; auto)

lemma disj-to-dimacs-sound:
\( I < \text{n-vars} \implies (\forall v \in \text{atoms } f \. \ 0 < v \land v < \text{n-vars} \) \implies \text{is-disj } f \implies M \models f \implies \exists l \in \text{set } (\text{model-to-dimacs-model } M \ [(1::\text{nat})..<\text{n-vars}]). \ l \in \text{set } (\text{disj-to-dimacs } f) \)

342
apply \((\text{induction } f)\)
using \textit{neg-in-model }\textit{pos-in-model }\textit{one-always-in}
by \((\text{fastforce elim!: is-lit-plus.elims dest!: isdisj-disjD})+\)

\textbf{lemma} \textit{is-cnf-disj}: \textit{is-cnf }\(f1 \lor f2\) \(\Rightarrow\) \((\forall f. f1 \lor f2 = f \Rightarrow \textit{is-disj }f \Rightarrow P) \Rightarrow P\)
by \(\text{auto}\)

\textbf{lemma} \textit{cnf-to-dimacs-disj}: \textit{is-disj }f \Rightarrow \textit{cnf-to-dimacs }f = [\text{disj-to-dimacs }f]
by \((\text{induction } f) \text{ auto}\)

\textbf{lemma} \textit{model-to-dimacs-model-all-clauses}:
\(1 < n\text{-vars} \Rightarrow (\forall v \in \text{atoms } f. 0 < v \land v < n\text{-vars}) \Rightarrow \textit{is-cnf }f \Rightarrow M \models f \Rightarrow v \in \text{set } (\textit{cnf-to-dimacs }f) \Rightarrow \exists l \in \text{set } (\textit{model-to-dimacs-model }M [(1::nat)..<n\text{-vars}])\).
\(l \in \text{set } c\)
\textbf{proof}(\textit{induction } f \text{ arbitrary: })

\textbf{case} \((\text{Not } f)\)
then show \(?case\)
using \textit{in-model }\textit{neg-in-model}
by \((\text{fastforce elim!: is-lit-plus.elims})+\)
\textbf{next}

\textbf{case} \((\text{Or } f1 f2)\)
then show \(?case\)
using \textit{cnf-to-dimacs-disj }\textit{disj-to-dimacs-sound}
by \((\text{elim is-cnf-disj, simp})\)
\textbf{qed} \((\text{insert in-model neg-in-model pos-in-model, auto})\)

\textbf{lemma} \textit{upt-eq-Cons-conv}:
\((x \# xs = [i..<j]) = (i < j \land i = x \land [i+1..<j] = xs)\)
using \textit{upt-eq-Cons-conv}
by \textit{metis}\n
\textbf{lemma} \textit{model-to-dimacs-model-append}:
\((\textit{model-to-dimacs-model }M (vs @ vs')) = (\textit{model-to-dimacs-model }M vs) @ (\textit{model-to-dimacs-model }M vs')\)
by \((\text{induction } vs) \text{ auto}\)

\textbf{lemma} \textit{model-to-dimacs-neg-nin}:
\(n\text{-vars} \leq x \Rightarrow \textit{int }x \notin \text{ set } (\textit{model-to-dimacs-model }M [a..<n\text{-vars}])\)
by \((\text{induction } n\text{-vars} \text{ arbitrary: } a) (\text{auto simp: model-to-dimacs-model-append})\)

\textbf{lemma} \textit{model-to-dimacs-pos-nin}:
\(n\text{-vars} \leq x \Rightarrow \neg \textit{int }x \notin \text{ set } (\textit{model-to-dimacs-model }M [a..<n\text{-vars}])\)
by \((\text{induction } n\text{-vars} \text{ arbitrary: } a) (\text{auto simp: model-to-dimacs-model-append})\)

\textbf{lemma} \textit{int-cases2}:
\(z \neq 0 \Rightarrow (\forall n. 0 \neq (\text{int }n) \Rightarrow z = \text{ int }n \Rightarrow P) \Rightarrow (\forall n. 0 \neq (\text{int }n) \Rightarrow P) \Rightarrow P\)
by \(\text{metis (full-types int-cases2)}\)
lemma model-to-dimacs-model-distinct:
  \( 1 < n\text{-vars} \implies \text{distinct} (\map \text{dimacs-lit-to-var} (\text{model-to-dimacs-model} M \{1..<n\text{-vars}\})) \)
by (induction \( n\text{-vars} \))
  (fastforce elim: int-cases2' 
    simp add: dimacs-lit-to-var-def model-to-dimacs-model-append' 
    model-to-dimacs-model-distinct 
    model-to-dimacs-model-distinct' model-to-dimacs-model-distinct')+

lemma model-to-dimacs-model-sound:
  \( 1 < n\text{-vars} \implies (\\forall v \in \text{atoms} f. \ 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnf} f \implies M \models f \implies \text{dimacs-model} (\text{model-to-dimacs-model} M \{1::\text{nat},..<n\text{-vars}\}) (\text{cnf-to-dimacs} f) \)
unfolding dimacs-model-def 
using model-to-dimacs-model-sound 
by auto

lemma model-to-dimacs-model-sound-exists:
  \( 1 < n\text{-vars} \implies (\\forall v \in \text{atoms} f. \ 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnf} f \implies M \models f \implies \exists M\text{-dimacs. dimacs-model} M\text{-dimacs} (\text{cnf-to-dimacs} f) \)
using model-to-dimacs-model-sound 
by metis

definition dimacs-to-atom ::\(\text{int} \Rightarrow \text{nat} \ \text{formula} \) where 
dimacs-to-atom \( l \equiv \) if \( (l < 0) \) then \( \text{Not} (\text{Atom} (\text{nat} (\text{abs} l))) \) else \( \text{Atom} (\text{nat} (\text{abs} l)) \)

definition dimacs-to-disj::\(\text{int list} \Rightarrow \text{nat} \ \text{formula} \) where 
dimacs-to-disj \( f \equiv \bigvee (\map \text{dimacs-to-atom} f) \)

definition dimacs-to-cnf::\(\text{int list list} \Rightarrow \text{nat} \ \text{formula} \) where 
dimacs-to-cnf \( f \equiv \bigwedge (\map \text{dimacs-to-disj} f) \)

definition dimacs-model-to-abs dimacs-M M \equiv 
fold \( (\lambda M. \text{if } (l > 0) \text{ then } M((\text{nat} (\text{abs} l)):= \text{True}) \text{ else } M((\text{nat} (\text{abs} l)):= \text{False})) \) dimacs-M M

lemma dimacs-model-to-abs-atom:
  \( 0 < x \Rightarrow \ \text{int} x \in \text{set} \ \text{dimacs-M} \implies \text{distinct} (\map \text{dimacs-lit-to-var} \ \text{dimacs-M}) \implies \text{dimacs-model-to-abs} \ \text{dimacs-M} M x \)
proof (induction dimacs-M arbitrary: M rule: rev-induct)
  case (snooc 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':
  \( 0 < x \Rightarrow \neg (\text{int} x) \in \text{set} \ \text{dimacs-M} \implies \text{distinct} (\map \text{dimacs-lit-to-var} \ \text{dimacs-M}) \implies \neg \ \text{dimacs-model-to-abs} \ \text{dimacs-M} M x \)
proof (induction dimacs-M arbitrary: M rule: rev-induct)

344
case (snoc 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:
(∀ v∈atoms f. 0 < v ∧ v < n-vars) ⇒ is-disj f ⇒ distinct (map dimacs-lit-to-var dimacs-M)
⇒ dimacs-model dimacs-M (cnf-to-dimacs f) ⇒ dimacs-model-to-abs dimacs-M (λ-. False) | f
by (induction f)
  (fastforce elim!: is-lit-plus.elims dest!: isdisj-disjD
    simp: cnf-to-dimacs-disj dimacs-model-def dimacs-model-to-abs-atom'
    dimacs-model-to-abs-atom)+

lemma model-to-dimacs-model-complete:
(∀ v∈atoms f. 0 < v ∧ v < n-vars) ⇒ is-cnf f ⇒ distinct (map dimacs-lit-to-var dimacs-M)
⇒ dimacs-model dimacs-M (cnf-to-dimacs f) ⇒ dimacs-model-to-abs dimacs-M (λ-. False) | f
proof (induction f)
case (Not f)
then show ?case
  by (auto elim!: is-lit-plus.elims simp add: dimacs-model-to-abs-atom' dimacs-model-def)
next
case (Or f1 f2)
then show ?case
  using cnf-to-dimacs-disj model-to-dimacs-model-complete-disj
  by (elim is-cnf-disj, simp add: dimacs-model-def)
qed (insert dimacs-model-to-abs-atom, auto simp: dimacs-model-def)

lemma model-to-dimacs-model-complete-max-var:
(∀ v∈atoms f. 0 < v) ⇒ is-cnf f ⇒
  dimacs-model dimacs-M (cnf-to-dimacs f) ⇒
  dimacs-model-to-abs dimacs-M (λ-. False) | f
using le-imp-less-Suc[OF max-var]
by (auto intro!: model-to-dimacs-model-complete simp: dimacs-model-def)

lemma model-to-dimacs-model-sound-max-var:
(∀ v∈atoms f. 0 < v) ⇒ is-cnf f ⇒ M | f ⇒
  dimacs-model (model-to-dimacs-model M [(1::nat).<(find-max (formula-vars f) + 2)])
  (cnf-to-dimacs f)
using le-imp-less-Suc[unfolded Suc-eq-plus1, OF max-var]
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:
  \( v \in \text{atoms} \rightarrow (\text{map-formula var-to-dimacs } f) \rightarrow 0 < v \)
  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 \( \pi \) \( \Rightarrow \) length \( \pi \) \( \leq \) \( h \)
  let cnf-formula = (map-formula var-to-dimacs
    \( (\Phi_{\forall} \ (\varphi \ (\text{prob-with-noop abs-prob}) \ h)) \))
  in
    \( \exists \dimacs-M. \dimacs-model \dimacs-M \ (\text{cnf-to-dimacs cnf-formula}) \)

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
      \( (\Phi_{\forall} \ (\varphi \ (\text{prob-with-noop abs-prob}) \ h)) \))
  in
    \( \dimacs-model \dimacs-M \ (\text{cnf-to-dimacs cnf-formula}) \rightarrow \)
    valid-plan
      (decode-abs-plan
        (rem-noops
          (map (\text{\lambda op. } \varphi_{\varphi}^{-1} \ (\text{prob-with-noop abs-prob}) \text{op})
            (concat
                \( (\Phi_{\forall}^{-1} \ (\varphi \ (\text{prob-with-noop abs-prob}) \ ((\dimacs-model-to-abs dimacs-M \lambda- \text{False}) \ o \var-to-dimacs \ h)))) \)))
  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.

**Definition**
\[\text{SASP-to-DIMACS } h \text{ prob } \equiv \]
\[
\begin{align*}
\text{cnf-to-dimacs} \\
(\text{cnf-to-dimacs.var-to-dimacs} (\text{Suc } h) (\text{Suc} (\text{length} (\text{ast-problem.ast} \delta \text{ prob})))) \\
(\Phi \forall (\varphi (\text{prob-with-noop} (\text{ast-problem.abs-prob prob})) h))
\end{align*}
\]

**Lemma** planning-dimacs-complete-code:
\[
\begin{align*}
\text{ast-problem.well-formed prob; } \\
\forall \pi \in \text{set} (\text{ast-problem.ast} \delta \text{ prob}), \text{is-standard-operator } \pi; \\
\text{ast-problem.valid-plan prob } \pi s; \\
\text{length } \pi s \leq h \implies \\
\text{let cnf-formula } = (\text{SASP-to-DIMACS } h \text{ prob}) \text{ in } \\
\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}
\end{align*}
\]

**Unfolding** SASP-to-DIMACS-def Let-def

**Apply** (rule sat-solve-sasp.planning-dimacs-complete[unfolded Let-def])

**Apply** unfold-locales

**By** auto

**Definition** SASP-to-DIMACS’ h prob \(\equiv\) SASP-to-DIMACS h (rem-implicit-pres-ops prob)

**Lemma** planning-dimacs-complete-code’:
\[
\begin{align*}
\text{ast-problem.well-formed prob; } \\
(\forall \text{op. op } \in \text{set} (\text{ast-problem.ast} \delta \text{ prob}) \implies \text{consistent-pres-op op}; \\
(\forall \text{op. op } \in \text{set} (\text{ast-problem.ast} \delta \text{ prob}) \implies \text{is-standard-operator op}); \\
\text{ast-problem.valid-plan prob } \pi s; \\
\text{length } \pi s \leq h \implies \\
\text{let cnf-formula } = (\text{SASP-to-DIMACS’ } h \text{ prob}) \text{ in } \\
\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}
\end{align*}
\]

**Unfolding** Let-def SASP-to-DIMACS’-def

**By** (auto simp add: rem-implicit-pres-ops-valid-plan[symmetric] wf-ast-problem-def

**Simp del:** rem-implicit-pres.simps

**Intro:** rem-implicit-pres-is-standard-operator’

**Planning-dimacs-complete-code[unfolded Let-def]**

**Rem-implicit-pres-ops-well-formed**

**Dest!:** rem-implicit-pres-ops-inD)
A function that does the checks required by the completeness theorem above, and returns appropriate error messages if any of the checks fail.

**Definition**

```haskell
code h prob ≡
  if ast-problem.well-formed prob then
    if (∀ op ∈ set (ast-problem.astδ prob). consistent-pres-op op) then
      if (∀ op ∈ set (ast-problem.astδ prob). is-standard-operator op) then
        Inl (SASP-to-DIMACS' h prob)
      else
        Inr (STR "Error: Conditional effects!")
    else
      Inr (STR "Error: Preconditions inconsistent")
  else
    Inr (STR "Error: Problem malformed!")
```

**Lemma** encode-sound:

\[
\text{[ast-problem.valid-plan prob π s; length π s ≤ h;}
\]
\[
\text{code h prob = Inl cnf-formula] \implies (\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula})}
\]

**Unfolding** encode-def

by (auto split: if-splits simp: list.pred-set
  intro: planning-dimacs-complete-code[unfolded Let-def])

**Lemma** encode-complete:

\[
\text{encode h prob = Inr err \implies}
\]
\[
\neg (\text{ast-problem.well-formed prob} \land (∀ op ∈ set (ast-problem.astδ prob). consistent-pres-op op}) \land
\]
\[
(\forall op ∈ set (ast-problem.astδ prob). is-standard-operator op))
\]

**Unfolding** encode-def

by (auto split: if-splits simp: list.pred-set
  intro: planning-dimacs-complete-code[unfolded Let-def])

**Definition** match-pre where

match-pre ≡ λ(x,v) s. s x = Some v

**Definition** match-pres where

match-pres-pres s ≡ ∀ pre ∈ set pres. match-pre pre s

**Lemma** match-pres-distinct:

distinct (map fst pres) \implies match-pres-pres s \iff Map.map-of pres ⊆ₘ s

**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

\[
\text{tree-map-of updatea T [] = T}
\]

348
tree-map-of updatea T ((v,a)#m) = updatea v a (tree-map-of updatea T m)

context Map
begin

abbreviation tree-map-of' ≡ tree-map-of update

lemma tree-map-of-invar: invar T ⇒ 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 = 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 ⇒ 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: map-empty 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 = set (inorder t)
  by (induction t) auto

lemma map-of-eq:
  map-of xs = Map.map-of xs
  by (induction xs) (auto simp: map-of-simps split: option.split)

lemma lookup-someD: lookup T x = Some y ⇒ ∃ p. p ∈ set (inorder T) ∧ p = (x, y)
  by (induction T) (auto split: if-splits)

lemma map-of-lookup: sorted1 (inorder T) ⇒ Map.map-of (inorder T) = 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: (∀x. m1 x = m2 x) → m1 ≤ₗ m s ⟷ m2 ≤ₗ m s
by presburger

lemma match-pres-submap:
  match-pres (inorder (M.tree-map-of′ empty pres)) s ⟷ Map.map-of pres ⊆ₗ 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. φ₀⁻¹ (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.ast₀ prob)))))))
        h))))

lemma planning-dimacs-sound-code:
  [ast-problem.well-formed prob;
∀ π∈set (ast-problem.ast₀ 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 impI 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. 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);
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
apply(subst rem-implicit-pres-ops-valid-plan|symmetric)
by(fastforce simp only: rem-implicit-pres-ops-valid-plan wf-ast-problem-def
 intro: rem-implicit-pres-is-standard-operator'
 rem-implicit-pres-ops-well-formed
 rev-iffD2[OF - rem-implicit-pres-ops-valid-plan]
 planning-dimacs-sound-code wf-ast-problem.intro
dest!: rem-implicit-pres-ops-in δD)+

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!

fun list-to-rbt :: int list ⇒ int rbt where
list-to-rbt [] = Leaf |
list-to-rbt (x#xs) = insert-rbt x (list-to-rbt xs)

lemma inv-list-to-rbt:
invc (list-to-rbt xs) ∧ invh (list-to-rbt xs)
by (induction xs) (auto simp: RBT.inv-insert)

lemma Tree2-list-to-rbt: Tree2.bst (list-to-rbt xs)
by (induction xs) (auto simp: RBT.bst-insert)

lemma set-list-to-rbt: Tree2.set-tree (list-to-rbt xs) = set xs
by (induction xs) (simp add: RBT.set-tree-insert Tree2-list-to-rbt)+

The following
lemma dimacs-model-code[code]:
dimacs-model ls cs ←→
(let tls = list-to-rbt ls in
 (∀ c∈set cs. size (inter-rbt (tls) (list-to-rbt c)) ≠ 0) ∧
distinct (map dimacs-lit-to-var ls))
using RBT.set-tree-inter[OF Tree2-list-to-rbt Tree2-list-to-rbt]
apply (auto simp: dimacs-model-def Let-def set-list-to-rbt inter-rbt-def)
apply (metis IntI RBT.set-empty empty-iff)
by (metis Tree2.eq-set-tree-empty disjoint-iff-not-equal)

definition decode M h prob ≡
if ast-problem.well-formed prob then
if (\( \forall \, op \in \text{set} \) \( (\text{ast-problem.ast} \delta \text{ prob}) \). consistent-pres-op op) then
  if (\( \forall \, op \in \text{set} \) \( (\text{ast-problem.ast} \delta \text{ prob}) \). is-standard-operator op) then
    if (\( \text{dimacs-model} \, M \) \( (\text{SASP-to-DIMACS'} \, h \, \text{prob}) \)) then
      \( \text{Inl} \) \( (\text{decode-DIMACS-model} \, M \, h \, \text{prob}) \)
    else \( \text{Inr} \) \( (\text{STR} \, \text{"Error: Model does not solve the problem!"}) \)
  else
    \( \text{Inr} \) \( (\text{STR} \, \text{"Error: Conditional effects!"}) \)
else
  \( \text{Inr} \) \( (\text{STR} \, \text{"Error: Preconditions inconsistent"}) \)
else
  \( \text{Inr} \) \( (\text{STR} \, \text{"Error: Problem malformed!"}) \)

\textbf{lemma} decode-sound:
\( \text{decode} \, M \, h \, \text{prob} = \text{Inl} \, \text{plan} \implies \)
\( \text{ast-problem.valid-plan} \, \text{prob} \, \text{plan} \)
\textbf{unfolding} decode-def
\textbf{apply} (\text{auto} \, \text{split: if-splits simp: list.pred-set})
\textbf{using} planning-dimacs-sound-code'
by auto

\textbf{lemma} decode-complete:
\( \text{decode} \, M \, h \, \text{prob} = \text{Inr} \, \text{err} \implies \)
\( \neg \, (\text{ast-problem.well-formed} \, \text{prob} \land \)
  (\( \forall \, op \in \text{set} \) \( (\text{ast-problem.ast} \delta \text{ prob}) \). consistent-pres-op op) \land
  (\( \forall \, \pi \in \text{set} \) \( (\text{ast-problem.ast} \delta \text{ prob}) \). is-standard-operator \( \pi \) \land
  \text{dimacs-model} \, M \) \( (\text{SASP-to-DIMACS'} \, h \, \text{prob}) \))
\textbf{unfolding} decode-def
by (\text{auto} \, \text{split: if-splits simp: list.pred-set})

\textbf{lemma} [code]:
\( \text{ListMem} \, x' \, ['] = \text{False} \)
\( \text{ListMem} \, x' \, (x#xs) = (x' = x \lor \text{ListMem} \, x' \, xs) \)
by (simp add: ListMem iff)+

\textbf{lemmas} [code] = \( \text{SASP-to-DIMACS-def ast-problem.abs-prob-def} \)
  \( \text{ast-problem.abs-ast-variable-section-def ast-problem.abs-ast-operator-section-def} \)
  \( \text{ast-problem.abs-ast-initial-state-def ast-problem.abs-range-map-def} \)
  \( \text{ast-problem.abs-ast-goal-def cnf-to-dimacs.var-to-dimacs.simps} \)
  \( \text{ast-problem.ast-\delta-def ast-problem.astDom-def ast-problem.abs-ast-operator-def} \)

\textbf{definition} nat-opt-of-integer :: integer \Rightarrow nat \, option \, where
\( \text{nat-opt-of-integer} \, i = (\text{if} \, (i \geq 0) \, \text{then} \, \text{Some} \, \text{(nat-of-integer} \, i) \, \text{else} \, \text{None}) \)

\textbf{definition} max-var :: \text{int list} \Rightarrow \text{int} \, where
\( \text{max-var} \, xs = \text{fold} \, (\lambda x :: \text{int} \, (y :: \text{int}). \, \text{if} \, \text{abs} \, x \geq \text{abs} \, y \, \text{then} \, (\text{abs} \, x) \, \text{else} \, y) \, xs \)
\( (\emptyset :: \text{int}) \)
References

