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

*Author names are alphabetically ordered.

theory *State-Variable-Representation*
imports *Main Propositional-Proof-Systems.Formulas Propositional-Proof-Systems.Sema*

Propositional-Proof-Systems.CNF
begin

1 State-Variable Representation

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

type-synonym $('variable, 'domain)\ state = 'variable \rightharpoonup 'domain$

type-synonym $('variable, 'domain)\ assignment = 'variable \times 'domain$

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

type-synonym $('variable, 'domain)\ Effect = ('variable \times 'domain)\ list$

end

theory *STRIPS-Representation*
imports *State-Variable-Representation*
begin

2 STRIPS Representation

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

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

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

— 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.

[1] For the full reference on records see [4, 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-precondition*
:: *'variable strips-operator* ⇒ (*'variable*, *bool*) *assignment list*
where *to-precondition op* ≡ *map* (λ*v*. (*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* ((-$_\mathcal{V}$) [*1000*] *999*)
operators-of :: *'variable strips-operator list* ((-$_\mathcal{O}$) [*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*
⇒ *'variable strips-operator*
⇒ *'variable strips-state* (**infixl** ≫ *52*)
where *execute-operator s op*
≡ *s* ++ *map-of* (*effect-to-assignments op*)

end

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

[2] Function `effect_to_assignments` converts the operator effect to a list of assignments.

3 STRIPS Semantics

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

3.1 Serial Plan Execution Semantics

Serial plan execution is defined by primitive recursion on the plan. Definition **??** returns the given state if the state argument does note satisfy the precondition of the next operator in the plan. Otherwise it executes the rest of the plan on the successor state $s \gg op$ of the given state and operator.

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

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

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

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

definition *is-serial-solution-for-problem*
  where *is-serial-solution-for-problem* Π *π*
    ≡ (*goal-of* Π) ⊆$_m$ *execute-serial-plan* (*initial-of* Π) *π*
      ∧ *list-all* (λ*op. 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)$_I$) = *set* ((Π)$_\mathcal{V}$)
  proof −

```
    {
      let ?I = strips-problem.initial-of Π
      let ?vs = strips-problem.variables-of Π
      fix v
      have ?I v ≠ None ⟷ ListMem v ?vs
        using assms(1)
        unfolding is-valid-problem-strips-def
        by meson
      hence v ∈ dom ?I ⟷ v ∈ set ?vs
        using ListMem-iff
        by fast
    }
    thus ?thesis
      by auto
  qed

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

lemma is-valid-problem-strips-operator-variable-sets:
  fixes Π:: 'a strips-problem
  assumes is-valid-problem-strips Π
    and op ∈ set ((Π)_𝒪)
  shows set (precondition-of op) ⊆ set ((Π)_𝒱)
    and set (add-effects-of op) ⊆ set ((Π)_𝒱)
    and set (delete-effects-of op) ⊆ set ((Π)_𝒱)
    and disjnt (set (add-effects-of op)) (set (delete-effects-of op))
```

proof −
let *?ops* = *strips-problem.operators-of* Π
and *?vs* = *strips-problem.variables-of* Π
have *list-all* (*is-valid-operator-strips* Π) *?ops*
using *assms*(*1*)
unfolding *is-valid-problem-strips-def*
by *meson*
moreover have ∀ *v* ∈ *set* (*precondition-of op*). *v* ∈ *set* ((Π)$_{\mathcal{V}}$)
and ∀ *v* ∈ *set* (*add-effects-of op*). *v* ∈ *set* ((Π)$_{\mathcal{V}}$)
and ∀ *v* ∈ *set* (*delete-effects-of op*). *v* ∈ *set* ((Π)$_{\mathcal{V}}$)
and ∀ *v* ∈ *set* (*add-effects-of op*). *v* ∉ *set* (*delete-effects-of op*)
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* ((Π)$_{\mathcal{V}}$)
and *set* (*add-effects-of op*) ⊆ *set* ((Π)$_{\mathcal{V}}$)
and *set* (*delete-effects-of op*) ⊆ *set* ((Π)$_{\mathcal{V}}$)
and *disjnt* (*set* (*add-effects-of op*)) (*set* (*delete-effects-of op*))
unfolding *disjnt-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 as_1 as_2
where *as* = as_1 @ as_2
and as_1 = *map* (λ*v*. (*v*, *True*)) (*add-effects-of op*)
and as_2 = *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* ∈ *set* (*add-effects-of op*)
and *as* = *effect-to-assignments op*
obtains *a* **where** *a* ∈ *set as a* = (*v*, *True*)
proof −
let *?add-assignments* = (λ*v*. (*v*, *True*)) ‘ *set* (*add-effects-of op*)
let *?delete-assignments* = (λ*v*. (*v*, *False*)) ‘ *set* (*delete-effects-of op*)
obtain as_1 as_2
where *a1*: *as* = as_1 @ as_2
and *a2*: as_1 = *map* (λ*v*. (*v*, *True*)) (*add-effects-of op*)
and *a3*: as_2 = *map* (λ*v*. (*v*, *False*)) (*delete-effects-of op*)
using *assms*(*2*) *effect-to-assignments-ii*
by *blast*
then have *b*: *set as*
= *?add-assignments* ∪ *?delete-assignments*
by *auto*
— NOTE The existence of an assignment as proposed can be shown by the following sequence of set inclusions.
{
from *b* **have** *?add-assignments* ⊆ *set as*
by *blast*
moreover have {(*v*, *True*)} ⊆ *?add-assignments*
using *assms*(*1*) *a2*
by *blast*
ultimately have ∃ *a*. *a* ∈ *set as* ∧ *a* = (*v*, *True*)
by *blast*
}
then show *?thesis*
using *that*
by *blast*
qed

lemma *effect-to-assignments-iii-b*:
— NOTE This proof is symmetrical to the one above.
fixes *v*
assumes *v* ∈ *set* (*delete-effects-of op*)
and *as* = *effect-to-assignments op*
obtains *a* **where** *a* ∈ *set as a* = (*v*, *False*)
proof −
let *?add-assignments* = (λ*v*. (*v*, *True*)) ‘ *set* (*add-effects-of op*)
let *?delete-assignments* = (λ*v*. (*v*, *False*)) ‘ *set* (*delete-effects-of op*)
obtain as_1 as_2
where *a1*: *as* = as_1 @ as_2
and *a2*: as_1 = *map* (λ*v*. (*v*, *True*)) (*add-effects-of op*)
and *a3*: as_2 = *map* (λ*v*. (*v*, *False*)) (*delete-effects-of op*)
using *assms*(*2*) *effect-to-assignments-ii*
by *blast*
then have *b*: *set as*
= *?add-assignments* ∪ *?delete-assignments*

by *auto*

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

{

from *b* **have** *?delete-assignments* $\subseteq$ *set as*

by *blast*

moreover have $\{(v, False)\} \subseteq$ *?delete-assignments*

using *assms(1) a2*

by *blast*

ultimately have $\exists\, a.\ a \in set\ as \wedge a = (v, False)$

by *blast*

}

then show *?thesis*

using *that*

by *blast*

qed

lemma *effect--strips-i*:

fixes *op*

assumes $e = effect\text{--}strips\ op$

obtains $es_1\ es_2$

where $e = (es_1\ @\ es_2)$

and $es_1 = map\ (\lambda v.\ (v, True))\ (add\text{-}effects\text{-}of\ op)$

and $es_2 = map\ (\lambda v.\ (v, False))\ (delete\text{-}effects\text{-}of\ op)$

proof −

obtain $es_1\ es_2$ **where** *a*: $e = (es_1\ @\ es_2)$

and *b*: $es_1 = map\ (\lambda v.\ (v, True))\ (add\text{-}effects\text{-}of\ op)$

and *c*: $es_2 = map\ (\lambda v.\ (v, False))\ (delete\text{-}effects\text{-}of\ op)$

using *assms(1)*

unfolding *effect--strips-def*

by *blast*

then show *?thesis*

using *that*

by *force*

qed

lemma *effect--strips-ii*:

fixes *op*

assumes $e = ConjunctiveEffect\ (es_1\ @\ es_2)$

and $es_1 = map\ (\lambda v.\ (v, True))\ (add\text{-}effects\text{-}of\ op)$

and $es_2 = map\ (\lambda v.\ (v, False))\ (delete\text{-}effects\text{-}of\ op)$

shows $\forall\, v \in set\ (add\text{-}effects\text{-}of\ op).\ (\exists\, e' \in set\ es_1.\ e' = (v, True))$

and $\forall\, v \in set\ (delete\text{-}effects\text{-}of\ op).\ (\exists\, e' \in set\ es_2.\ e' = (v, False))$

proof

— NOTE Show that for each variable v in the add effect list, we can obtain an atomic effect with true value.

fix *v*

{

assume *a*: $v \in set\ (add\text{-}effects\text{-}of\ op)$

have *set* $es_1 = (\lambda v.\ (v,\ True))$ ‘ *set* (*add-effects-of op*)
using *assms*(*2*) *List.set-map*
by *auto*
then obtain e'
where $e' \in$ *set* es_1
and $e' = (\lambda v.\ (v,\ True))\ v$
using *a*
by *blast*
then have $\exists\, e' \in$ *set* $es_1.\ e' = (v,\ True)$
by *blast*
}
thus $v \in$ *set* (*add-effects-of op*) $\Longrightarrow \exists\, e' \in$ *set* $es_1.\ e' = (v,\ True)$
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*: $v \in$ *set* (*delete-effects-of op*)
have *set* $es_2 = (\lambda v.\ (v,\ False))$ ‘ *set* (*delete-effects-of op*)
using *assms*(*3*) *List.set-map*
by *force*
then obtain e''
where $e'' \in$ *set* es_2
and $e'' = (\lambda v.\ (v,\ False))\ v$
using *a*
by *blast*
then have $\exists\, e'' \in$ *set* $es_2.\ e'' = (v,\ False)$
by *blast*
}
thus $\forall\, v{\in}$*set* (*delete-effects-of op*). $\exists\, e'{\in}$*set* $es_2.\ e' = (v,\ False)$
by *fast*
qed

lemma *map-of-constant-assignments-dom*:
— NOTE ancillary lemma used in the proof below.
assumes $m =$ *map-of* (*map* $(\lambda v.\ (v,\ d))$ *vs*)
shows *dom* $m =$ *set vs*
proof −
let *?vs′* = *map* $(\lambda v.\ (v,\ d))$ *vs*
have *dom* $m =$ *fst* ‘ *set ?vs′*
using *assms*(*1*) *dom-map-of-conv-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 ⋀v. v ∈ set (add-effects-of op) ⟹ s' v = Some True
  proof −
    fix v
    assume a: v ∈ set (add-effects-of op)
    let ?as = effect-to-assignments op
    obtain as₁ as₂ where b: ?as = as₁ @ as₂
      and c: as₁ = map (λv. (v, True)) (add-effects-of op)
      and as₂ = map (λv. (v, False)) (delete-effects-of op)
      using effect-to-assignments-ii
      by blast
    have d: map-of ?as = map-of as₂ ++ map-of as₁
      using b Map.map-of-append
      by auto
    {
      — TODO refactor?
      let ?vs = add-effects-of op
      have ?vs ≠ []
        using a
        by force
      then have dom (map-of as₁) = set (add-effects-of op)
        using c map-of-constant-assignments-dom
        by metis
      then have v ∈ dom (map-of as₁)
        using a
        by blast
      then have map-of ?as v = map-of as₁ v
        using d
        by force
    } moreover {
      let ?f = λ-. True
      from c have map-of as₁ = (Some ∘ ?f) |` (set (add-effects-of op))
        using map-of-map-restrict
        by fast
      then have map-of as₁ v = Some True
        using a
        by auto
    }
    moreover have s' = s ++ map-of as₂ ++ map-of as₁
      using assms(1)
      unfolding execute-operator-def
      using b
      by simp
    ultimately show s' v = Some True
      by simp
  qed
```

lemma *effect--strips-iii-b*:
assumes $s' = (s \gg op)$
shows $\bigwedge v.\ v \in set\ (delete\text{-}effects\text{-}of\ op) \wedge v \notin set\ (add\text{-}effects\text{-}of\ op) \Longrightarrow s'\ v = Some\ False$
proof (*auto*)
fix *v*
assume *a1*: $v \notin set\ (add\text{-}effects\text{-}of\ op)$ **and** *a2*: $v \in set\ (delete\text{-}effects\text{-}of\ op)$
let *?as* = *effect-to-assignments op*
obtain $as_1\ as_2$ **where** *b*: $?as = as_1\ @\ as_2$
and *c*: $as_1 = map\ (\lambda v.\ (v,\ True))\ (add\text{-}effects\text{-}of\ op)$
and *d*: $as_2 = map\ (\lambda v.\ (v,\ False))\ (delete\text{-}effects\text{-}of\ op)$
using *effect-to-assignments-ii*
by *blast*
have *e*: *map-of ?as* = *map-of* as_2 ++ *map-of* as_1
using *b Map.map-of-append*
by *auto*
{
have *dom* (*map-of* as_1) = *set* (*add-effects-of op*)
using *c map-of-constant-assignments-dom*
by *metis*
then have $v \notin dom\ (map\text{-}of\ as_1)$
using *a1*
by *blast*
} **note** *f* = *this*
{
let *?vs* = *delete-effects-of op*
have $?vs \neq []$
using *a2*
by *force*
then have *dom* (*map-of* as_2) = *set ?vs*
using *d map-of-constant-assignments-dom*
by *metis*
} **note** *g* = *this*
{
have $s' = s$ ++ *map-of* as_2 ++ *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* ++ *map-of* as_2]
moreover have $s'\ v = (s$ ++ *map-of* $as_2)\ v$
using *calculation map-add-dom-app-simps*(*3*)[*OF f*, *of s* ++ *map-of* as_2]
by *blast*
moreover have $v \in dom\ (map\text{-}of\ as_2)$
using *a2 g*
by *argo*
ultimately have $s'\ v = map\text{-}of\ as_2\ v$
by *fastforce*

```
    }
    moreover
    {
      let ?f = λ-. False
      from d have map-of as₂ = (Some ∘ ?f) |` (set (delete-effects-of op))
        using map-of-map-restrict
        by fast
      then have map-of as₂ v = Some False
        using a2
        by force
    }
    ultimately show   s' v = Some False
      by argo
  qed


lemma effect--strips-iii-c:
  assumes s' = (s ≫ op)
  shows ⋀v. v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of op) ⟹ s' v =
s v
  proof (auto)
    fix v
    assume a1: v ∉ set (add-effects-of op) and a2: v ∉ set (delete-effects-of op)
    let ?as = effect-to-assignments op
    obtain as₁ as₂ where b: ?as = as₁ @ as₂
      and c: as₁ = map (λv. (v, True)) (add-effects-of op)
      and d: as₂ = map (λv. (v, False)) (delete-effects-of op)
      using effect-to-assignments-ii
      by blast
    have e: map-of ?as = map-of as₂ ++ map-of as₁
      using b Map.map-of-append
      by auto
    {
      have dom (map-of as₁) = set (add-effects-of op)
        using c map-of-constant-assignments-dom
        by metis
      then have v ∉ dom (map-of as₁)
        using a1
        by blast
    } moreover {
      have dom (map-of as₂) = set (delete-effects-of op)
        using d map-of-constant-assignments-dom
        by metis
      then have v ∉ dom (map-of as₂)
        using a2
        by blast
    }
    ultimately show s' v = s v
      using assms(1)
```

unfolding *execute-operator-def*
by (*simp add*: *b map-add-dom-app-simps*(*3*))
qed

The following theorem combines three preceding sublemmas which show that the following properties hold for the successor state $s' \equiv$ *execute-operator op s* obtained by executing an operator *op* in a state s: [3]

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

theorem *operator-effect--strips*:
assumes $s' = (s \gg op)$
shows
$\bigwedge v.$
$v \in set\ (add\text{-}effects\text{-}of\ op)$
$\Longrightarrow s'\ v = Some\ True$
and $\bigwedge v.$
$v \notin set\ (add\text{-}effects\text{-}of\ op) \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)$
$\Longrightarrow s'\ v = Some\ False$
and $\bigwedge v.$
$v \notin set\ (add\text{-}effects\text{-}of\ op) \wedge v \notin set\ (delete\text{-}effects\text{-}of\ op)$
$\Longrightarrow s'\ v = s\ v$
proof (*auto*)
show $\bigwedge v.$
$v \in set\ (add\text{-}effects\text{-}of\ op)$
$\Longrightarrow s'\ v = Some\ True$
using *assms effect--strips-iii-a*
by *fast*
next
show $\bigwedge v.$
$v \notin set\ (add\text{-}effects\text{-}of\ op)$
$\Longrightarrow v \in set\ (delete\text{-}effects\text{-}of\ op)$
$\Longrightarrow\ s'\ v = Some\ False$
using *assms effect--strips-iii-b*
by *fast*
next
show $\bigwedge v.$
$v \notin set\ (add\text{-}effects\text{-}of\ op)$
$\Longrightarrow v \notin set\ (delete\text{-}effects\text{-}of\ op)$
$\Longrightarrow s'\ v = s\ v$

[3] Lemmas `effect__strips_iii_a`, `effect__strips_iii_b`, and `effect__strips_iii_c` (not shown).

using *assms effect--strips-iii-c*
by *metis*
qed

3.2 Parallel Plan Semantics

definition *are-all-operators-applicable s ops*
≡ *list-all* (λ*op*. *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* (λ*v*. *list-ex* ((=) *v*) del_2) add_1 ∧ ¬*list-ex* (λ*v*. *list-ex* ((=) *v*) add_2) del_1

definition *are-all-operator-effects-consistent ops* ≡
list-all (λ*op*. *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 unecessary in the serial case).

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

definition *are-operators-interfering* op_1 op_2
≡ *list-ex* (λ*v*. *list-ex* ((=) *v*) (*delete-effects-of* op_1)) (*precondition-of* op_2)
∨ *list-ex* (λ*v*. *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* (λ*op'*. ¬*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* (λ*ops*. *list-all* (λ*op*.
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* (λ*op*. *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 (¬list-ex (λv. 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 (¬list-ex (λv. list-ex ((=) v) (add-effects-of op₂)) (delete-effects-of
op₁))
      = (set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {})
      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
```

```
    }
    thus ?thesis
      unfolding are-all-operator-effects-consistent-def list-all-iff
      by force
  qed

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

lemma are-all-operators-non-interfering-tail:
  assumes are-all-operators-non-interfering (op # ops)
  shows are-all-operators-non-interfering ops
  using assms
```

unfolding *are-all-operators-non-interfering-def*
by *simp*

lemma *are-operators-interfering-symmetric*:
assumes *are-operators-interfering* op_1 op_2
shows *are-operators-interfering* op_2 op_1
using *assms*
unfolding *are-operators-interfering-def list-ex-iff*
by *fast*

— A small technical characterizing operator lists with property . We show that pairs of distinct operators which interfere with one another cannot both be contained in the corresponding operator set.
lemma *are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs*:
assumes *are-all-operators-non-interfering ops*
and *are-operators-interfering* op_1 op_2
and $op_1 \neq op_2$
shows $op_1 \notin set\ ops \vee op_2 \notin set\ ops$
using *assms*
proof (*induction ops*)
case (*Cons op ops*)
thm *Cons.IH*[*OF - Cons.prems*(*2*, *3*)]
have nb_1: $\forall\, op' \in set\ ops.\ \neg$*are-operators-interfering op op'*
and nb_2: *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*) $op = op_1$
| (*B*) $op = op_2$
| (*C*) $op \neq op_1 \wedge op \neq op_2$
by *blast*
thus *?case*
proof (*cases*)
case *A*
{
assume $op_2 \in set\ (op\ \#\ ops)$
then have $op_2 \in set\ ops$
using *Cons.prems*(*3*) *A*
by *force*
then have ¬*are-operators-interfering* op_1 op_2
using nb_1 *A*
by *fastforce*
hence *False*
using *Cons.prems*(*2*)**..**
}
thus *?thesis*
by *blast*
next
case *B*

```
    {
      assume op₁ ∈ set (op # ops)
      then have op₁ ∈ set ops
        using Cons.prems(3) B
        by force
      then have ¬are-operators-interfering op₁ op₂
        using nb₁ B are-operators-interfering-symmetric
        by blast
      hence False
        using Cons.prems(2)..
    }
    thus ?thesis
      by blast
  next
    case C
    thus ?thesis
      using Cons.IH[OF nb₂ Cons.prems(2, 3)]
      by force
  qed
qed simp


lemma execute-parallel-plan-precondition-cons-i:
  fixes s :: ('variable, bool) state
  assumes ¬are-operators-interfering op op'
    and is-operator-applicable-in s op
    and is-operator-applicable-in s op'
  shows is-operator-applicable-in (s ++ map-of (effect-to-assignments op)) op'
  proof −
    let ?s' = s ++ map-of (effect-to-assignments op)
    — TODO slightly hackish to exploit the definition of execute-operator, but
we otherwise have to rewrite theorem operator-effect--strips (which is a todo as of
now).
    {
      have a: ?s' = s ≫ op
        by (simp add: execute-operator-def)
      then have ⋀v. v ∈ set (add-effects-of op) ⟹ ?s' v = Some True
        and ⋀v. v ∉ set (add-effects-of op) ∧ v ∈ set (delete-effects-of op) ⟹ ?s'
v = Some False
        and ⋀v. v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of op) ⟹ ?s'
v = s v
        using operator-effect--strips
        by metis+
    }
    note a = this
    — TODO refactor lemma not-have-interference-set.
    {
      fix v
      assume α: v ∈ set (precondition-of op')
```

```
    {
      fix v
      have ¬list-ex ((=) v) (delete-effects-of op)
        = list-all (λv'. ¬v = v') (delete-effects-of op)
        using not-list-ex-equals-list-all-not[
            where P=(=) v and xs=delete-effects-of op]
        by blast
    } moreover {
      from assms(1)
     have ¬list-ex (λv. list-ex ((=) v) (delete-effects-of op)) (precondition-of op')
        unfolding are-operators-interfering-def
        by blast
     then have list-all (λv. ¬list-ex ((=) v) (delete-effects-of op)) (precondition-of
op')
        using not-list-ex-equals-list-all-not[
         where P=λv. list-ex ((=) v) (delete-effects-of op) and xs=precondition-of
op']
        by blast
    }
    ultimately have β:
     list-all (λv. list-all (λv'. ¬v = v') (delete-effects-of op)) (precondition-of op')
      by presburger
    moreover {
      fix v
      have list-all (λv'. ¬v = v') (delete-effects-of op)
        = (∀ v' ∈ set (delete-effects-of op). ¬v = v')
        using list-all-iff [where P=λv'. ¬v = v' and x=delete-effects-of op]
        .
    }
    ultimately have ∀ v ∈ set (precondition-of op'). ∀ v' ∈ set (delete-effects-of
op). ¬v = v'
      using β list-all-iff[
        where P=λv. list-all (λv'. ¬v = v') (delete-effects-of op)
          and x=precondition-of op']
      by presburger
    then have v ∉ set (delete-effects-of op)
      using α
      by fast
  }
  note b = this
  {
    fix v
    assume a: v ∈ set (precondition-of op')
    have list-all (λv. s v = Some True) (precondition-of op')
      using assms(3)
      unfolding is-operator-applicable-in-def
        STRIPS-Representation.is-operator-applicable-in-def
      by presburger
    then have ∀ v ∈ set (precondition-of op'). s v = Some True
```

```
      using list-all-iff[where P=λv. s v = Some True and x=precondition-of
op']
      by blast
    then have s v = Some True
      using a
      by blast
  }
  note c = this
  {
    fix v
    assume d: v ∈ set (precondition-of op')
    then have ?s' v = Some True
    proof (cases v ∈ set (add-effects-of op))
      case True
      then show ?thesis
        using a
        by blast
    next
      case e: False
      then show ?thesis
      proof (cases v ∈ set (delete-effects-of op))
        case True
        then show ?thesis
          using assms(1) b d
            by fast
        next
          case False
          then have ?s' v = s v
            using a e
            by blast
          then show ?thesis
            using c d
            by presburger
        qed
    qed
  }
  then have list-all (λv. ?s' v = Some True) (precondition-of op')
     using list-all-iff[where P=λv. ?s' v = Some True and x=precondition-of
op']
    by blast
  then show ?thesis
    unfolding is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def
    by auto
qed
```

— The third assumption *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 nb_1: ∀ *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 nb_2: ∀ *op* ∈ *set ops*. ¬*are-operators-interfering a op*
using *assms*(*3*)
unfolding *are-all-operators-non-interfering-def list-all-iff*
by *simp*
have nb_3: *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*] nb_1 nb_2 nb_3
by *force*
}
then show *are-all-operators-applicable ?s' ops*
unfolding *are-all-operators-applicable-def list-all-iff*
is-operator-applicable-in-def
by *blast*
qed

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

lemma *execute-parallel-operator-cons-equals*:
assumes *are-all-operators-applicable s* (*a* # *ops*)
and *are-all-operator-effects-consistent* (*a* # *ops*)
and *are-all-operators-non-interfering* (*a* # *ops*)

shows *execute-parallel-operator s (a # ops)*
= execute-parallel-operator (s ++ map-of (effect-to-assignments a)) ops
proof −
let *?s′ = s ++ map-of (effect-to-assignments a)*
{
from *assms(1, 2)*
have *execute-parallel-operator s (Cons a ops)*
= foldl (++) s (map (map-of ∘ 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-of ∘ effect-to-assignments) a)
(map (map-of ∘ effect-to-assignments) ops)
by (*simp add*: *execute-parallel-operator-def*)
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* (λ*v*. (*v*, *True*)) (*add-effects-of op*) @ *map* (λ*v*. (*v*, *False*)) (*delete-effects-of op*)
by (*simp add*: *effect-to-assignments-i*)

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

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

```
      using ListMem-iff assms
      by fastforce
    } moreover {
     have map ?g (add-effects-of op @ delete-effects-of op)
      = map ?g (add-effects-of op) @ map ?g (delete-effects-of op)
      using map-append
      by fast
     — TODO slow.
    hence effect-to-assignments op = map ?g (add-effects-of op @ delete-effects-of
op)
      using ListMem-iff assms
      by fastforce
    }
    ultimately show effect-to-assignments op = map
     (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 ∉ fst ` set ?l
      using assms(1, 2)
      by blast
  }
  thus ?thesis
    using map-of-eq-None-iff[of ?l v]
    by blast
qed

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

lemma execute-parallel-operator-positive-effect-if:
  fixes ops
  assumes are-all-operators-applicable s ops
    and are-all-operator-effects-consistent ops
    and op ∈ set ops
    and v ∈ set (add-effects-of op)
  shows execute-parallel-operator s ops v = Some True
```

```
proof −
  let ?l = map (map-of ∘ effect-to-assignments) ops
  have set-l-is: set ?l = (map-of ∘ effect-to-assignments) ‘ set ops
    using set-map
    by fastforce
  {
    let ?m = (map-of ∘ effect-to-assignments) op
    have ?m ∈ set ?l
      using assms(3) set-l-is
      by blast
    moreover have ?m v = Some True
      using execute-parallel-operator-positive-effect-if-i[OF assms]
      by fastforce
    ultimately have ∃ m ∈ set ?l. m v = Some True
      by blast
  }
  moreover {
    fix m'
    assume m' ∈ set ?l
    then obtain op'
      where op'-in-set-ops: op' ∈ set ops
        and m'-is: m' = (map-of ∘ effect-to-assignments) op'
      by auto
    then have set (add-effects-of op) ∩ set (delete-effects-of op') = {}
      using assms(2, 3) are-all-operator-effects-consistent-set[of ops]
      by blast
    then have v ∉ set (delete-effects-of op')
      using assms(4)
      by blast
    then consider (v-in-set-add-effects) v ∈ set (add-effects-of op')
      | (otherwise) ¬v ∈ set (add-effects-of op') ∧ ¬v ∈ set (delete-effects-of op')
      by blast
    hence m' v = Some True ∨ m' v = None
      proof (cases)
        case v-in-set-add-effects
        — TODO slow.
        thus ?thesis
          using execute-parallel-operator-positive-effect-if-i[
              OF assms(1, 2) op'-in-set-ops, of v] m'-is
          by simp
      next
        case otherwise
        then have ¬v ∈ set (add-effects-of op')
          and ¬v ∈ set (delete-effects-of op')
          by blast+
        thus ?thesis
          using map-of-effect-to-assignments-is-none-if[of v op'] m'-is
          by fastforce
      qed
```

```
    }
    — TODO slow.
    ultimately show ?thesis
      unfolding execute-parallel-operator-def
      using foldl-map-append-is-some-if[of s v True ?l]
      by meson
  qed

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

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

```
  and v ∈ set (delete-effects-of op)
shows execute-parallel-operator s ops v = Some False
proof −
  let ?l = map (map-of ∘ effect-to-assignments) ops
  have set-l-is: set ?l = (map-of ∘ effect-to-assignments) ‘ set ops
    using set-map
    by fastforce
  {
    let ?m = (map-of ∘ effect-to-assignments) op
    have ?m ∈ set ?l
      using assms(3) set-l-is
      by blast
    moreover have ?m v = Some False
      using execute-parallel-operator-negative-effect-if-i[OF assms]
      by fastforce
    ultimately have ∃ m ∈ set ?l. m v = Some False
      by blast
  }
  moreover {
    fix m'
    assume m' ∈ set ?l
    then obtain op'
      where op'-in-set-ops: op' ∈ set ops
        and m'-is: m' = (map-of ∘ effect-to-assignments) op'
      by auto
    then have set (delete-effects-of op) ∩ set (add-effects-of op') = {}
      using assms(2, 3) are-all-operator-effects-consistent-set[of ops]
      by blast
    then have v ∉ set (add-effects-of op')
      using assms(4)
      by blast
    then consider (v-in-set-delete-effects) v ∈ set (delete-effects-of op')
      | (otherwise) ¬v ∈ set (add-effects-of op') ∧ ¬v ∈ set (delete-effects-of op')
      by blast
    hence m' v = Some False ∨ m' v = None
      proof (cases)
        case v-in-set-delete-effects
        — TODO slow.
        thus ?thesis
          using execute-parallel-operator-negative-effect-if-i[
              OF assms(1, 2) op'-in-set-ops, of v] m'-is
          by simp
      next
        case otherwise
        then have ¬v ∈ set (add-effects-of op')
          and ¬v ∈ set (delete-effects-of op')
          by blast+
        thus ?thesis
          using map-of-effect-to-assignments-is-none-if[of v op'] m'-is
```

```
        by fastforce
      qed
    }
    — TODO slow.
    ultimately show ?thesis
      unfolding execute-parallel-operator-def
      using foldl-map-append-is-some-if[of s v False ?l]
      by meson
  qed

lemma execute-parallel-operator-no-effect-if:
  assumes ∀ op ∈ set ops. ¬v ∈ set (add-effects-of op) ∧ ¬v ∈ set (delete-effects-of
op)
  shows execute-parallel-operator s ops v = s v
  using assms
  unfolding execute-parallel-operator-def
  proof (induction ops arbitrary: s)
    case (Cons a ops)
    let ?f = map-of ∘ effect-to-assignments
    {
      have v ∉ set (add-effects-of a) ∧ v ∉ set (delete-effects-of a)
        using Cons.prems(1)
        by force
      then have ?f a v = None
        using map-of-effect-to-assignments-is-none-if[of v a]
        by fastforce
      then have v ∉ dom (?f a)
        by blast
      hence (s ++ ?f a) v = s v
        using map-add-dom-app-simps(3)[of v ?f a s]
        by blast
    }
    moreover {
      have ∀ op∈set ops. v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of op)
        using Cons.prems(1)
        by simp
      hence foldl (++) (s ++ ?f a) (map ?f ops) v = (s ++ ?f a) v
        using Cons.IH[of s ++ ?f a]
        by blast
    }
    moreover {
      have map ?f (a # ops) = ?f a # map ?f ops
        by force
      then have foldl (++) s (map ?f (a # ops))
        = foldl (++) (s ++ ?f a) (map ?f ops)
        using foldl-Cons
        by force
    }
    ultimately show ?case
```

by *argo*
qed *fastforce*

corollary *execute-parallel-operators-strips-none-if*:
assumes $\forall$ *op* $\in$ *set ops*. $\neg$*v* $\in$ *set* (*add-effects-of op*) $\wedge$ $\neg$*v* $\in$ *set* (*delete-effects-of op*)
and *s v* = *None*
shows *execute-parallel-operator s ops v* = *None*
using *execute-parallel-operator-no-effect-if*[*OF assms*(*1*)] *assms*(*2*)
by *simp*

corollary *execute-parallel-operators-strips-none-if-contraposition*:
assumes $\neg$*execute-parallel-operator s ops v* = *None*
shows ($\exists$ *op* $\in$ *set ops*. *v* $\in$ *set* (*add-effects-of op*) $\vee$ *v* $\in$ *set* (*delete-effects-of op*))
$\vee$ *s v* $\neq$ *None*
proof $-$
let *?P* = ($\forall$ *op* $\in$ *set ops*. $\neg$*v* $\in$ *set* (*add-effects-of op*) $\wedge$ $\neg$*v* $\in$ *set* (*delete-effects-of op*))
$\wedge$ *s v* = *None*
and *?Q* = *execute-parallel-operator s ops v* = *None*
have *?P* $\Longrightarrow$ *?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 *op* $\in$ *set ops* and with *v* being an add effect of *op*, then the successor state

$$s' \equiv \textit{execute-parallel-operator } s \textit{ ops}$$

will evaluate *v* to true, that is

$$\textit{execute-parallel-operator } s \textit{ ops } v = \textit{Some True}$$

Symmetrically, if *v* is a delete effect, we have

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

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 ∈ set ops ∧ v ∈ set (add-effects-of op)
  ⟶ execute-parallel-operator s ops v = Some True
  and op ∈ set ops ∧ v ∈ set (delete-effects-of op)
    ⟶ execute-parallel-operator s ops v = Some False
  and (∀ op ∈ set ops.
    v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of op))
    ⟶ execute-parallel-operator s ops v = s v
  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 Π:: 'a strips-problem
  assumes is-parallel-solution-for-problem Π π
    and ops ∈ set π
    and op ∈ set ops
  shows op ∈ set ((Π)_𝒪)
  proof −
    have ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set (strips-problem.operators-of Π)
      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 π ≠ []
  proof (cases π)
    case (Cons a list)
    then show ?thesis
    by (cases are-all-operators-applicable I (hd π) ∧ are-all-operator-effects-consistent
(hd π)
        , simp+)
  qed simp

corollary length-trace-parallel-plan-gt-0[simp]: 0 < length (trace-parallel-plan-strips
I π)
```

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

corollary *length-trace-minus-one-lt-length-trace*[*simp*]:
length (*trace-parallel-plan-strips I* π) − *1* < *length* (*trace-parallel-plan-strips I* π)
using *diff-less*[*OF - length-trace-parallel-plan-gt-0*]
by *auto*

lemma *trace-parallel-plan-strips-head-is-initial-state*:
trace-parallel-plan-strips I π ! *0* = *I*
proof (*cases* π)
case (*Cons a list*)
then show *?thesis*
by (*cases are-all-operators-applicable I a* ∧ *are-all-operator-effects-consistent a*, *simp*+)
qed *simp*

lemma *trace-parallel-plan-strips-length-gt-one-if*:
assumes *k* < *length* (*trace-parallel-plan-strips I* π) − *1*
shows *1* < *length* (*trace-parallel-plan-strips I* π)
using *assms*
by *linarith*

— This lemma simply shows that the last element of a *trace-parallel-plan-strips execution step s* # *trace-parallel-plan-strips s*′ π always is the last element of *trace-parallel-plan-strips s*′ π since *trace-parallel-plan-strips* always returns at least a singleton list (even if π = []).
lemma *trace-parallel-plan-strips-last-cons-then*:
last (*s* # *trace-parallel-plan-strips s*′ π) = *last* (*trace-parallel-plan-strips s*′ π)
by (*cases* π, *simp*, *force*)

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

First, all parallel operators *ops* = π ! *k* for any index *k* with *k* < *length* τ − *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 *ops* would have been the last element. This would violate the assumption that *k* < *length* τ − *1* is not the last index since the index of the last element is *length* τ − *1*. [4]

lemma *trace-parallel-plan-strips-operator-preconditions*:
assumes *k* < *length* (*trace-parallel-plan-strips I* π) − *1*
shows *are-all-operators-applicable* (*trace-parallel-plan-strips I* π ! *k*) (π ! *k*)
∧ *are-all-operator-effects-consistent* (π ! *k*)
using *assms*

[4] More precisely, the index of the last element is *length* τ − *1* if τ is not empty which is however always true since the trace contains at least the initial state.

```
  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 I a ∧ are-all-operator-effects-consistent a)
      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
$k <$ *length* (*trace-parallel-plan-strips* I π), the k-th element of the trace is state reached by executing the plan prefix *take* k π consisting of the first k parallel operators of π.

```
lemma trace-parallel-plan-plan-prefix:
  assumes k < length (trace-parallel-plan-strips I π)
  shows trace-parallel-plan-strips I π ! k = execute-parallel-plan I (take k π)
  using assms
  proof (induction π arbitrary: I k)
    case (Cons a π)
    then show ?case
    proof (cases are-all-operators-applicable I a ∧ are-all-operator-effects-consistent
a)
      case True
      let ?σ = trace-parallel-plan-strips I (a # π)
        and ?I' = execute-parallel-operator I a
      have σ-equals: ?σ = I # trace-parallel-plan-strips ?I' π
        using True
        by auto
      then show ?thesis
        proof (cases k = 0)
          case False
          obtain k' where k-is-suc-of-k': k = Suc k'
            using not0-implies-Suc[OF False]
            by blast
          then have execute-parallel-plan I (take k (a # π))
            = execute-parallel-plan ?I' (take k' π)
            using True
            by simp
          moreover have trace-parallel-plan-strips I (a # π) ! k
            = trace-parallel-plan-strips ?I' π ! k'
            using σ-equals k-is-suc-of-k'
            by simp
```

```
          moreover {
            have k' < length (trace-parallel-plan-strips (execute-parallel-operator I
a) π)
              using Cons.prems σ-equals k-is-suc-of-k'
              by force
            hence trace-parallel-plan-strips ?I' π ! k'
              = execute-parallel-plan ?I' (take k' π)
              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 # π) = [I]
            using operator-precondition-violated
            by force
          moreover have execute-parallel-plan I (take k (a # π)) = I
            using Cons.prems operator-precondition-violated
            by force
          ultimately show ?thesis
            using Cons.prems nth-Cons-0
            by auto
        qed simp
      qed
    qed simp


lemma length-trace-parallel-plan-strips-lte-length-plan-plus-one:
  shows length (trace-parallel-plan-strips I π) ≤ length π + 1
  proof (induction π arbitrary: I)
    case (Cons a π)
    then show ?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 auto
          then have length (trace-parallel-plan-strips I (a # π))
            = length (trace-parallel-plan-strips ?I' π) + 1
            by simp
```

```
        moreover have length (trace-parallel-plan-strips ?I' π) ≤ length π + 1
          using Cons.IH[of ?I']
          by blast
        ultimately have length (trace-parallel-plan-strips I (a # π)) ≤ length (a
# π) + 1
          by simp
      }
      thus ?thesis
        by blast
    qed auto
  qed simp

— Show that π is at least a singleton list.
lemma plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements:
  assumes k < length (trace-parallel-plan-strips I π) − 1
  obtains ops π' where π = ops # π'
  proof −
    let ?τ = trace-parallel-plan-strips I π
    have length ?τ ≤ length π + 1
      using length-trace-parallel-plan-strips-lte-length-plan-plus-one
      by fast
    then have 0 < length π
      using trace-parallel-plan-strips-length-gt-one-if assms
      by force
    then obtain k' where length π = Suc k'
      using gr0-implies-Suc
      by meson
    thus ?thesis using that
      using length-Suc-conv[of π 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 length (trace-parallel-plan-strips I π) < length π + 1
  shows ¬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 assms
  proof (induction π arbitrary: I)
    case (Cons ops π)
    let ?τ = trace-parallel-plan-strips I (ops # π)
      and ?I' = execute-parallel-operator I ops
    show ?case
    proof (cases are-all-operators-applicable I ops ∧ are-all-operator-effects-consistent
```

```
ops)
      case True
      then have τ-is: ?τ = I # trace-parallel-plan-strips ?I' π
        by fastforce
      show ?thesis
        proof (cases length (trace-parallel-plan-strips ?I' π) < length π + 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 trace-parallel-plan-step-effect-is:
  assumes k < length (trace-parallel-plan-strips I π) − 1
  shows trace-parallel-plan-strips I π ! Suc k
    = execute-parallel-operator (trace-parallel-plan-strips I π ! k) (π ! k)
  proof −
   — NOTE Rewrite the proposition using lemma trace-parallel-plan-strips-subplan.
    {
      let ?τ = trace-parallel-plan-strips I π
      have Suc k < length ?τ
        using assms
        by linarith
      hence trace-parallel-plan-strips I π ! Suc k
        = execute-parallel-plan I (take (Suc k) π)
        using trace-parallel-plan-plan-prefix[of Suc k I π]
        by blast
    }
    moreover have execute-parallel-plan I (take (Suc k) π)
      = execute-parallel-operator (trace-parallel-plan-strips I π ! k) (π ! k)
      using assms
      proof (induction k arbitrary: I π)
        case 0
        then have execute-parallel-operator (trace-parallel-plan-strips I π ! 0) (π !
0)
            = execute-parallel-operator I (π ! 0)
          using trace-parallel-plan-strips-head-is-initial-state[of I π]
          by argo
        moreover {
          obtain ops π′ where π = ops # π′
           using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
0.prems]
            by blast
          then have take (Suc 0) π = [π ! 0]
            by simp
          hence execute-parallel-plan I (take (Suc 0) π)
            = execute-parallel-plan I [π ! 0]
            by argo
        }
        moreover {
          have 0 < length (trace-parallel-plan-strips I π) − 1
            using trace-parallel-plan-strips-length-gt-one-if 0.prems
            by fastforce
          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 I π]
        by argo
    }
    ultimately show ?case
      by 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) =
        1 + 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 I 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 all problem variables—by definition of *is-valid-problem-strips* Π—and no operator can remove a previously defined variable (only positive and negative effects are possible).

```
lemma trace-parallel-plan-strips-none-if:
  fixes Π:: 'a strips-problem
  assumes is-valid-problem-strips Π
    and is-parallel-solution-for-problem Π π
    and k < length (trace-parallel-plan-strips ((Π)_I) π)
  shows (trace-parallel-plan-strips ((Π)_I) π ! k) v = None ⟷ v ∉ set ((Π)_𝒱)
  proof −
    let ?vs = strips-problem.variables-of Π
      and ?ops = strips-problem.operators-of Π
      and ?τ = trace-parallel-plan-strips ((Π)_I) π
      and ?I = strips-problem.initial-of Π
    show ?thesis
      using assms
      proof (induction k)
        case 0
        have ?τ ! 0 = ?I
          using trace-parallel-plan-strips-head-is-initial-state
          by auto
        then show ?case
          using is-valid-problem-strips-initial-of-dom[OF assms(1)]
          by auto
      next
        case (Suc k)
        have k-lt-length-τ-minus-one: k < length ?τ − 1
          using Suc.prems(3)
          by linarith
        then have IH: (trace-parallel-plan-strips ?I π ! k) v = None ⟷ v ∉set
((Π)_𝒱)
          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-vs: 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-π_k: op ∈ set (π ! k)
             and v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)..
          then consider (A) v ∈ set (add-effects-of op)
            | (B) v ∈ set (delete-effects-of op)
            by blast
          thus False
            using execute-parallel-operator-positive-effect-if[OF
                    all-operators-applicable all-effects-consistent op-in-π_k]
                  execute-parallel-operator-negative-effect-if[OF
                    all-operators-applicable all-effects-consistent op-in-π_k]
                  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
    qed
next
  assume v-notin-vs: v ∉ set ((Π)_𝒱)
  {
    fix op
    assume op-in-π_k: 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 π ! k ∈ set π
              by simp
          }
          then have op-in-ops: op ∈ set ?ops
              using is-parallel-solution-for-problem-operator-set[OF assms(2) -
op-in-π_k]
            by force
          hence v ∉ set (add-effects-of op) and v ∉ set (delete-effects-of op)
            subgoal
                using is-valid-problem-strips-operator-variable-sets(2) assms(1)
op-in-ops
                  v-notin-vs
              by auto
            subgoal
                using is-valid-problem-strips-operator-variable-sets(3) assms(1)
op-in-ops
                  v-notin-vs
              by auto
            done
        }
        then have (?τ ! Suc k) v = (?τ ! k) v
          using execute-parallel-operator-no-effect-if τ-Suc-k-is
          by metis
        thus (?τ ! Suc k) v = None
          using IH v-notin-vs
          by fastforce
      qed
    qed
  qed
```

Finally, given initial and goal states I and G, we can show that it's equivalent to say that π is a solution for I and G—i.e. $G \subseteq_m$ *execute-parallel-plan* I π—and that the goal state is subsumed by the last element of the trace of π with initial state I.

```
lemma execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace:
  G ⊆_m execute-parallel-plan I π
    ⟷ G ⊆_m last (trace-parallel-plan-strips I π)
  proof −
    let ?LHS = G ⊆_m execute-parallel-plan I π
      and ?RHS = G ⊆_m last (trace-parallel-plan-strips I π)
    show ?thesis
      proof (rule iffI)
```

```
    assume ?LHS
    thus ?RHS
      proof (induction π arbitrary: I)
        — NOTE Nil case follows from simplification.
        case (Cons a π)
        thus ?case
          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.IH[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[of I ?I' π]
              by argo
          }
          ultimately show ?thesis
            by argo
        qed force
      qed simp
  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
        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
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.

```
lemma execute-parallel-operator-equals-execute-sequential-strips-if:
  fixes s :: ('variable, bool) state
  assumes are-all-operators-applicable s ops
    and are-all-operator-effects-consistent ops
    and are-all-operators-non-interfering ops
  shows execute-parallel-operator s ops = execute-serial-plan s ops
  using assms
  proof (induction ops arbitrary: s)
    case Nil
    have execute-parallel-operator s Nil
```

```
    = foldl (++) s (map (map-of ∘ effect-to-assignments) Nil)
    using Nil.prems(1,2)
    unfolding execute-parallel-operator-def
    by presburger
  also have ... = s
    by simp
  finally have execute-parallel-operator s Nil = s
    by blast
  moreover have execute-serial-plan s Nil = s
    by auto
  ultimately show ?case
    by simp
next
  case (Cons a ops)
  — NOTE Use the preceding lemmas to show that the premises hold for the
sublist and use the IH to obtain the theorem for the sublist ops.
  have a: is-operator-applicable-in s a
    using are-all-operators-applicable-cons Cons.prems(1)
    by blast+
  let ?s' = s ++ map-of (effect-to-assignments a)
  {
    from Cons.prems
    have are-all-operators-applicable ?s' ops
      and are-all-operator-effects-consistent ops
      and are-all-operators-non-interfering ops
      using execute-parallel-plan-precondition-cons
      by blast+
    then have execute-serial-plan ?s' ops
      = execute-parallel-operator ?s' ops
      using Cons.IH
      by presburger
  }
  moreover from Cons.prems
  have execute-parallel-operator s (Cons a ops)
    = execute-parallel-operator ?s' ops
    using execute-parallel-operator-cons-equals-corollary
    unfolding execute-operator-def
    by simp
  moreover
  from a have execute-serial-plan s (Cons a ops)
    = execute-serial-plan ?s' ops
    unfolding execute-serial-plan-def execute-operator-def
      is-operator-applicable-in-def
    by fastforce
  ultimately show ?case
    by argo
qed

lemma execute-serial-plan-split-i:
```

assumes *are-all-operators-applicable s* (*op* # *π*)
and *are-all-operators-non-interfering* (*op* # *π*)
shows *are-all-operators-applicable* (*s* ≫ *op*) *π*
using *assms*
proof (*induction π arbitrary*: *s*)
case *Nil*
then show *?case*
unfolding *are-all-operators-applicable-def*
by *simp*
next
case (*Cons op′ π*)
let *?t* = *s* ≫ *op*
{
fix *x*
assume *x* ∈ *set* (*op′* # *π*)
moreover have *op* ∈ *set* (*op* # *op′* # *π*)
by *simp*
moreover have ¬*are-operators-interfering op x*
using *Cons.prems*(*2*) *calculation*(*1*)
unfolding *are-all-operators-non-interfering-def list-all-iff*
by *fastforce*
moreover have *is-operator-applicable-in s op*
using *Cons.prems*(*1*)
unfolding *are-all-operators-applicable-def list-all-iff*
is-operator-applicable-in-def
by *force*
moreover have *is-operator-applicable-in s x*
using *are-all-operators-applicable-cons*(*2*)[*OF Cons.prems*(*1*)] *calculation*(*1*)
unfolding *are-all-operators-applicable-def list-all-iff*
is-operator-applicable-in-def
by *fast*
ultimately have *is-operator-applicable-in ?t x*
using *execute-parallel-plan-precondition-cons-i*[*of op x s*]
by (*auto simp*: *execute-operator-def*)
}
thus *?case*
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

— Show that plans π can be split into separate executions of partial plans π_1 and π_2 with $\pi = \pi_1 @ \pi_2$, if all operators in π_1 are applicable in the given state s and there is no interference between subsequent operators in π_1. This is the case because non interference ensures that no precondition for any operator in π_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 :: ('*variable*, *bool*) *state*
assumes *are-all-operators-applicable* s π_1
and *are-all-operators-non-interfering* π_1
shows *execute-serial-plan* s (π_1 @ π_2)
= *execute-serial-plan* (*execute-serial-plan* s π_1) π_2
using *assms*
proof (*induction* π_1 *arbitrary*: s)
case (*Cons op* π_1)
let *?t* = $s \gg op$
{
have *are-all-operators-applicable* ($s \gg op$) π_1
using *execute-serial-plan-split-i*[*OF Cons.prems*(*1*, *2*)]**.**
moreover have *are-all-operators-non-interfering* π_1
using *are-all-operators-non-interfering-tail*[*OF Cons.prems*(*2*)]**.**
ultimately have *execute-serial-plan ?t* (π_1 @ π_2) =
execute-serial-plan (*execute-serial-plan ?t* π_1) π_2
using *Cons.IH*[*of ?t*]
by *blast*
}
moreover have *STRIPS-Representation.is-operator-applicable-in s op*
using *Cons.prems*(*1*)
unfolding *are-all-operators-applicable-def list-all-iff*
by *fastforce*
ultimately show *?case*
unfolding *execute-serial-plan-def*
by *simp*
qed *simp*

lemma *embedding-lemma-i*:
fixes I :: ('*variable*, *bool*) *state*
assumes *is-operator-applicable-in I op*
and *are-operator-effects-consistent op op*
shows $I \gg op$ = *execute-parallel-operator I* [*op*]
proof −
have *are-all-operators-applicable I* [*op*]
using *assms*(*1*)
unfolding *are-all-operators-applicable-def list-all-iff is-operator-applicable-in-def*
by *fastforce*
moreover have *are-all-operator-effects-consistent* [*op*]
unfolding *are-all-operator-effects-consistent-def list-all-iff*
using *assms*(*2*)
by *fastforce*
moreover have *are-all-operators-non-interfering* [*op*]
by *simp*
moreover have $I \gg op$ = *execute-serial-plan I* [*op*]

```
    using assms(1)
    unfolding  is-operator-applicable-in-def
    by (simp add: assms(1) execute-operator-def)
  ultimately show ?thesis
    using execute-parallel-operator-equals-execute-sequential-strips-if
    by force
qed

lemma execute-serial-plan-is-execute-parallel-plan-ii:
  fixes I :: 'variable strips-state
  assumes ∀ op ∈ set π. are-operator-effects-consistent op op
    and G ⊆m execute-serial-plan I π
  shows G ⊆m execute-parallel-plan I (embed π)
  proof −
    show ?thesis
      using assms
      proof (induction π arbitrary: I)
        case (Cons op π)
        then show ?case
          proof (cases is-operator-applicable-in I op)
            case True
            let ?J = I ≫ op
              and ?J' = execute-parallel-operator I [op]
            {
              have G ⊆m execute-serial-plan ?J π
                using Cons.prems(2) True
                unfolding is-operator-applicable-in-def
                by (simp add: True)
              hence G ⊆m execute-parallel-plan ?J (embed π)
                using Cons.IH[of ?J] Cons.prems(1)
                by fastforce
            }
            moreover {
              have are-all-operators-applicable I [op]
                using True
                unfolding are-all-operators-applicable-def list-all-iff
                  is-operator-applicable-in-def
                by fastforce
              moreover have are-all-operator-effects-consistent [op]
                unfolding are-all-operator-effects-consistent-def list-all-iff
                using Cons.prems(1)
                by fastforce
              moreover have ?J = ?J'
                using execute-parallel-operator-equals-execute-sequential-strips-if[OF
                    calculation(1, 2)] Cons.prems(1) True
                unfolding  is-operator-applicable-in-def
                by (simp add: True)
              ultimately have execute-parallel-plan I (embed (op # π))
                = execute-parallel-plan ?J (embed π)
```

```
        by fastforce
      }
      ultimately show ?thesis
        by presburger
    next
      case False
      then have G ⊆m I
        using Cons.prems is-operator-applicable-in-def
        by simp
      moreover {
        have ¬are-all-operators-applicable I [op]
          using False
          unfolding are-all-operators-applicable-def list-all-iff
            is-operator-applicable-in-def
          by force
        hence execute-parallel-plan I (embed (op # π)) = I
          by simp
      }
      ultimately show ?thesis
        by presburger
    qed
  qed simp
qed

lemma embedding-lemma-iii:
  fixes Π:: 'a strips-problem
  assumes ∀ op ∈ set π. op ∈ set ((Π)𝒪)
  shows ∀ ops ∈ set (embed π). ∀ op ∈ set ops. op ∈ set ((Π)𝒪)
  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 ((Π)𝒪)
        using assms(1)
        by simp
    }
    thus ?thesis..
  qed
```

We show in the following theorem that—as mentioned—a serial solution π to a STRIPS problem Π corresponds directly to a parallel solution obtained by embedding each operator in π in a list (by use of function *List-Supplement.embed*). The proof shows this by first confirming that

$G \subseteq_m$ *execute-serial-plan* $((\Pi)_I)$ π
$\Longrightarrow G \subseteq_m$ *execute-serial-plan* $((\Pi)_I)$ (*embed* π)

using lemma ; and moreover by showing that

$\forall$ *ops* $\in$ *set* (*embed* π). $\forall$ *op* $\in$ *set ops*. *op* $\in (\Pi)_{\mathcal{O}}$

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

theorem *embedding-lemma*:
assumes *is-valid-problem-strips* Π
and *is-serial-solution-for-problem* Π π
shows *is-parallel-solution-for-problem* Π (*embed* π)
proof −
have *nb*$_1$: $\forall$ *op* $\in$ *set* π. *op* $\in$ *set* $((\Pi)_{\mathcal{O}})$
using *assms*(*2*)
unfolding *is-serial-solution-for-problem-def list-all-iff ListMem-iff operators-of-def*
by *blast*
{
fix *op*
assume *op* $\in$ *set* π
moreover have *op* $\in$ *set* $((\Pi)_{\mathcal{O}})$
using *nb*$_1$ *calculation*
by *fast*
moreover have *is-valid-operator-strips* Π *op*
using *assms*(*1*) *calculation*(*2*)
unfolding *is-valid-problem-strips-def is-valid-problem-strips-def list-all-iff operators-of-def*
by *meson*
moreover have *list-all* (λ*v*. $\neg$*ListMem v* (*delete-effects-of op*)) (*add-effects-of op*)
and *list-all* (λ*v*. $\neg$*ListMem v* (*add-effects-of op*)) (*delete-effects-of op*)
using *calculation*(*3*)
unfolding *is-valid-operator-strips-def*
by *meson*+
moreover have $\neg$*list-ex* (λ*v*. *ListMem v* (*delete-effects-of op*)) (*add-effects-of op*)
and $\neg$*list-ex* (λ*v*. *ListMem v* (*add-effects-of op*)) (*delete-effects-of op*)
using *calculation*(*4*, *5*) *not-list-ex-equals-list-all-not*
by *blast*+
moreover have $\neg$*list-ex* (λ*v*. *list-ex* ((=) *v*) (*delete-effects-of op*)) (*add-effects-of op*)
and $\neg$*list-ex* (λ*v*. *list-ex* ((=) *v*) (*add-effects-of op*)) (*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
  } note nb2 = this
  moreover {
    have (Π)_G ⊆_m execute-serial-plan ((Π)_I) π
      using assms(2)
      unfolding is-serial-solution-for-problem-def
      by simp
    hence (Π)_G ⊆_m execute-parallel-plan ((Π)_I) (embed π)
      using execute-serial-plan-is-execute-parallel-plan-ii nb2
      by blast
  }
  moreover have ∀ ops ∈ set (embed π). ∀ op ∈ set ops. op ∈ set ((Π)_O)
    using embedding-lemma-iii[OF nb1].
  ultimately show ?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

lemma flattening-lemma-i:
  fixes Π:: 'a strips-problem
  assumes ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set ((Π)_O)
  shows ∀ op ∈ set (concat π). op ∈ set ((Π)_O)
  proof −
    {
      fix op
      assume op ∈ set (concat π)
      moreover have op ∈ (⋃ ops ∈ set π. set ops)
        using calculation
        unfolding set-concat.
      then obtain ops where ops ∈ set π and op ∈ set ops
        using UN-iff
        by blast
      ultimately have op ∈ set ((Π)_O)
        using assms
        by blast
    }
    thus ?thesis..
  qed

lemma flattening-lemma-ii:
  fixes I :: 'variable strips-state
  assumes ∀ ops ∈ set π. ∃ op. ops = [op] ∧ is-valid-operator-strips Π op
    and G ⊆_m execute-parallel-plan I π
  shows G ⊆_m execute-serial-plan I (concat π)
  proof −
    let ?π' = concat π
```

```
  {
    fix op
    assume is-valid-operator-strips Π op
   moreover have list-all (λv. ¬ListMem v (delete-effects-of op)) (add-effects-of
op)
      and list-all (λv. ¬ListMem v (add-effects-of op)) (delete-effects-of op)
      using calculation(1)
      unfolding is-valid-operator-strips-def
      by meson+
   moreover have ¬list-ex (λv. ListMem v (delete-effects-of op)) (add-effects-of
op)
      and ¬list-ex (λv. ListMem v (add-effects-of op)) (delete-effects-of op)
      using calculation(2, 3) not-list-ex-equals-list-all-not
      by blast+
  moreover have ¬list-ex (λv. list-ex ((=) v) (delete-effects-of op)) (add-effects-of
op)
      and ¬list-ex (λv. list-ex ((=) v) (add-effects-of op)) (delete-effects-of op)
      using calculation(4, 5)
      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
  } note nb1 = this
  show ?thesis
    using assms
    proof (induction π arbitrary: I)
      case (Cons ops π)
     obtain op where ops-is: ops = [op] and is-valid-op: is-valid-operator-strips
Π op
        using Cons.prems(1)
        by fastforce
      show ?case
        proof (cases are-all-operators-applicable I ops)
          case True
          let ?J = execute-parallel-operator I [op]
            and ?J' = I ≫ op
          have nb2: is-operator-applicable-in I op
            using True ops-is
            unfolding are-all-operators-applicable-def list-all-iff
              is-operator-applicable-in-def
            by simp
          have nb3: are-operator-effects-consistent op op
            using nb1[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 Π
    and ∀ ops ∈ set π. ∃ op. ops = [op]
    and is-parallel-solution-for-problem Π π
  shows is-serial-solution-for-problem Π (concat π)
  proof −
    let ?π' = concat π
    {
      have ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set ((Π)𝒪)
        using assms(3)
        unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
```

```
    by force
  hence ∀ op ∈ set ?π'. op ∈ set ((Π)_𝒪)
    using flattening-lemma-i
    by blast
 }
 moreover {
  {
   fix ops
   assume ops ∈ set π
   moreover obtain op where ops = [op]
    using assms(2) calculation
    by blast
   moreover have op ∈ set ((Π)_𝒪)
    using assms(3) calculation
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
    by force
   moreover have is-valid-operator-strips Π op
    using assms(1) calculation(3)
    unfolding is-valid-problem-strips-def Let-def list-all-iff ListMem-iff
    by simp
   ultimately have ∃ op. ops = [op] ∧ is-valid-operator-strips Π op
    by blast
  }
  moreover have (Π)_G ⊆_m execute-parallel-plan ((Π)_I) π
   using assms(3)
   unfolding is-parallel-solution-for-problem-def
   by simp
  ultimately have (Π)_G ⊆_m execute-serial-plan ((Π)_I) ?π'
   using flattening-lemma-ii
   by blast
 }
 ultimately show is-serial-solution-for-problem Π ?π'
  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 Π, and for which both the parallel operator execution condition as well as non interference is assured at every point $k < \mathit{length}\ \pi$, the flattening of the parallel plan $\mathit{concat}\ \pi$ is a serial solution for the initial and goal state of the problem. To wit, by lemma **??** we have

```
(G ⊆_m execute-parallel-plan I π)
    = (G ⊆_m last (trace-parallel-plan-strips I π))
```

so the second assumption entails that π is a solution for the initial state and the goal state of the problem. (which implicitely means that π is a solution for the inital 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 π with arbitrary initial state.

```
theorem execute-parallel-plan-is-execute-sequential-plan-if:
  fixes I :: ('variable, bool) state
  assumes is-valid-problem Π
    and G ⊆m last (trace-parallel-plan-strips I π)
    and ∀ k < length π.
      are-all-operators-applicable (trace-parallel-plan-strips I π ! k) (π ! k)
      ∧ are-all-operator-effects-consistent (π ! k)
      ∧ are-all-operators-non-interfering (π ! k)
  shows G ⊆m execute-serial-plan I (concat π)
  using assms
  proof (induction π arbitrary: I)
    case (Cons ops π)
    let ?ops' = take (length ops) (concat (ops # π))
    let ?J = execute-parallel-operator I ops
      and ?J' = execute-serial-plan I ?ops'
    {
      have trace-parallel-plan-strips I π ! 0 = I and (ops # π) ! 0 = ops
        unfolding trace-parallel-plan-strips-head-is-initial-state
        by simp+
      then have are-all-operators-applicable I ops
        and are-all-operator-effects-consistent ops
        and are-all-operators-non-interfering ops
        using Cons.prems(3)
        by auto+
      then have trace-parallel-plan-strips I (ops # π)
        = I # trace-parallel-plan-strips ?J π
        by fastforce
    } note nb = this
    {
      have last (trace-parallel-plan-strips I (ops # π))
        = last (trace-parallel-plan-strips ?J π)
        using trace-parallel-plan-strips-last-cons-then nb
        by metis
      hence G ⊆m last (trace-parallel-plan-strips ?J π)
        using Cons.prems(2)
        by force
    }
    moreover {
      fix k
      assume k < length π
      moreover have k + 1 < length (ops # π)
        using calculation
        by force
      moreover have π ! k = (ops # π) ! (k + 1)
        by simp
      ultimately have are-all-operators-applicable
```

(*trace-parallel-plan-strips ?J* π ! k) (π ! k)
and *are-all-operator-effects-consistent* (π ! k)
and *are-all-operators-non-interfering* (π ! k)
using *Cons.prems*(3) *nb*
by *force+*
}
ultimately have $G \subseteq_m$ *execute-serial-plan ?J* (*concat* π)
using *Cons.IH*[*OF Cons.prems*(1), *of ?J*]
by *blast*
moreover {
have *execute-serial-plan I* (*concat* (*ops* # π))
= *execute-serial-plan ?J'* (*concat* π)
using *execute-serial-plan-split*[*of I 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* (*concat* (*ops* # π))
= *execute-serial-plan ?J* (*concat* π)
using *execute-serial-plan-split*[*of I ops*] *Cons.prems*(3)
by *argo*
}
ultimately show *?case*
by *argo*
qed *force*

3.4 Auxiliary lemmas about STRIPS

lemma *set-to-precondition-of-op-is*[*simp*]: *set* (*to-precondition op*)
= { (*v*, *True*) | *v*. *v* ∈ *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* ((-$_{\mathcal{V}+}$) [*1000*] *999*)
  operators-of :: (*'variable*, *'domain*) *sas-plus-operator list* ((-$_{\mathcal{O}+}$) [*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* ($\mathcal{R}_+$ - - *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*
      ; *eff* = *effect-of op*
      ; *vs* = *variables-of* Ψ
      ; *D* = *range-of* Ψ
    in list-all (λ(*v*, *a*). *ListMem v vs*) *pre*
      ∧ *list-all* (λ(*v*, *a*). (*D v* ≠ *None*) ∧ *ListMem a* (*the* (*D v*))) *pre*
      ∧ *list-all* (λ(*v*, *a*). *ListMem v vs*) *eff*
      ∧ *list-all* (λ(*v*, *a*). (*D v* ≠ *None*) ∧ *ListMem a* (*the* (*D v*))) *eff*
      ∧ *list-all* (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) *pre*) *pre*

$\wedge$ *list-all* ($\lambda(v,\ a).$ *list-all* ($\lambda(v',\ a').\ v \neq v' \vee a = a'$) *eff*) *eff*

definition *is-valid-problem-sas-plus* Ψ
$\equiv$ *let ops* = *operators-of* Ψ
; *vs* = *variables-of* Ψ
; I = *initial-of* Ψ
; G = *goal-of* Ψ
; D = *range-of* Ψ
in list-all ($\lambda v.\ D\ v \neq None$) *vs*
$\wedge$ *list-all* (*is-valid-operator-sas-plus* Ψ) *ops*
$\wedge$ ($\forall v.\ I\ v \neq None \longleftrightarrow ListMem\ v\ vs$)
$\wedge$ ($\forall v.\ I\ v \neq None \longrightarrow ListMem$ (*the* ($I\ v$)) (*the* ($D\ v$)))
$\wedge$ ($\forall v.\ G\ v \neq None \longrightarrow ListMem\ v$ (*variables-of* Ψ))
$\wedge$ ($\forall v.\ G\ v \neq None \longrightarrow ListMem$ (*the* ($G\ v$)) (*the* ($D\ v$)))

definition *is-operator-applicable-in*
:: ('*variable*, '*domain*) *state*
$\Rightarrow$ ('*variable*, '*domain*) *sas-plus-operator*
$\Rightarrow$ *bool*
where *is-operator-applicable-in s op*
$\equiv$ *map-of* (*precondition-of op*) $\subseteq_m$ *s*

definition *execute-operator-sas-plus*
:: ('*variable*, '*domain*) *state*
$\Rightarrow$ ('*variable*, '*domain*) *sas-plus-operator*
$\Rightarrow$ ('*variable*, '*domain*) *state* (**infixl** $\gg_+$ *52*)
where *execute-operator-sas-plus s op* $\equiv$ *s* ++ *map-of* (*effect-of op*)

— Set up simp rules to keep use of local parameters transparent within proofs (i.e. automatically substitute definitions).
lemma[*simp*]:
is-operator-applicable-in s op = (*map-of* (*precondition-of op*) $\subseteq_m$ *s*)
s $\gg_+$ *op* = *s* ++ *map-of* (*effect-of op*)
unfolding *initial-of-def goal-of-def variables-of-def range-of-def operators-of-def*

SAS-Plus-Representation.is-operator-applicable-in-def
SAS-Plus-Representation.execute-operator-sas-plus-def
by *simp+*

lemma *range-of-not-empty*:
(*sas-plus-problem.range-of* Ψ *v* $\neq$ *None* $\wedge$ *sas-plus-problem.range-of* Ψ *v* $\neq$ *Some* []) $\longleftrightarrow$ ($\mathcal{R}_+$ Ψ *v*) $\neq$ {}
apply (*cases sas-plus-problem.range-of* Ψ *v*)
by (*auto simp add*: *SAS-Plus-Representation.range-of'-def*)

lemma *is-valid-operator-sas-plus-then*:
fixes Ψ::('*v*,'*d*) *sas-plus-problem*

assumes *is-valid-operator-sas-plus* Ψ *op*
shows $\forall (v,\ a) \in$ *set* (*precondition-of op*). $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set* (*precondition-of op*). $(\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
and $\forall (v,\ a) \in$ *set* (*effect-of op*). $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set* (*effect-of op*). $(\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
and $\forall (v,\ a) \in$ *set* (*precondition-of op*). $\forall (v',\ a') \in$ *set* (*precondition-of op*). $v \neq v' \vee a = a'$
and $\forall (v,\ a) \in$ *set* (*effect-of op*).
$\forall (v',\ a') \in$ *set* (*effect-of op*). $v \neq v' \vee a = a'$
proof −
let *?vs* = *sas-plus-problem.variables-of* Ψ
and *?pre* = *precondition-of op*
and *?eff* = *effect-of op*
and *?D* = *sas-plus-problem.range-of* Ψ
have $\forall (v,\ a) \in$*set ?pre*. $v \in$ *set ?vs*
and $\forall (v,\ a) \in$*set ?pre*.
(*?D* $v \neq$ *None*) $\wedge$
$a \in$ *set* (*the* (*?D* v))
and $\forall (v,\ a) \in$*set ?eff*. $v \in$ *set ?vs*
and $\forall (v,\ a) \in$*set ?eff*.
(*?D* $v \neq$ *None*) $\wedge$
$a \in$ *set* (*the* (*?D* v))
and $\forall (v,\ a) \in$*set ?pre*.
$\forall (v',\ a') \in$*set ?pre*. $v \neq v' \vee a = a'$
and $\forall (v,\ a) \in$*set ?eff*.
$\forall (v',\ a') \in$*set ?eff*. $v \neq v' \vee a = a'$
using *assms*
unfolding *is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff*
by *meson+*
moreover have $\forall (v,\ a) \in$ *set ?pre*. $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set ?eff*. $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set ?pre*. $\forall (v',\ a') \in$ *set ?pre*. $v \neq v' \vee a = a'$
and $\forall (v,\ a) \in$ *set ?eff*. $\forall (v',\ a') \in$ *set ?eff*. $v \neq v' \vee a = a'$
using *calculation*
unfolding *variables-of-def*
by *blast+*
moreover {
have $\forall (v,\ a) \in$ *set ?pre*. (*?D* $v \neq$ *None*) $\wedge$ $a \in$ *set* (*the* (*?D* v))
using *assms*
unfolding *is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff*
by *argo*
hence $\forall (v,\ a) \in$ *set ?pre*. $((\mathcal{R}_+\ \Psi\ v) \neq \{\}) \wedge a \in \mathcal{R}_+\ \Psi\ v$
using *range-of'-def*
by *fastforce*
}
moreover {
have $\forall (v,\ a) \in$ *set ?eff*. (*?D* $v \neq$ *None*) $\wedge$ $a \in$ *set* (*the* (*?D* v))
using *assms*
unfolding *is-valid-operator-sas-plus-def Let-def list-all-iff ListMem-iff*

by *argo*
hence $\forall (v,\ a) \in$ *set ?eff*. $((\mathcal{R}_+\ \Psi\ v) \neq \{\}) \wedge a \in \mathcal{R}_+\ \Psi\ v$
using *range-of′-def*
by *fastforce*
}
ultimately show $\forall (v,\ a) \in$ *set* (*precondition-of op*). $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set* (*precondition-of op*). $(\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
and $\forall (v,\ a) \in$ *set* (*effect-of op*). $v \in$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall (v,\ a) \in$ *set* (*effect-of op*). $(\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
and $\forall (v,\ a) \in$ *set* (*precondition-of op*). $\forall (v',\ a') \in$ *set* (*precondition-of op*). $v \neq v' \vee a = a'$
and $\forall (v,\ a) \in$ *set* (*effect-of op*).
$\forall (v',\ a') \in$ *set* (*effect-of op*). $v \neq v' \vee a = a'$
by *blast+*
qed

lemma *is-valid-problem-sas-plus-then*:
fixes Ψ::('*v*,'*d*) *sas-plus-problem*
assumes *is-valid-problem-sas-plus* Ψ
shows $\forall v \in$ *set* $((\Psi)_{\mathcal{V}+})$. $(\mathcal{R}_+\ \Psi\ v) \neq \{\}$
and $\forall op \in$ *set* $((\Psi)_{\mathcal{O}+})$. *is-valid-operator-sas-plus* Ψ *op*
and *dom* $((\Psi)_{I+})$ = *set* $((\Psi)_{\mathcal{V}+})$
and $\forall v \in$ *dom* $((\Psi)_{I+})$. *the* $(((\Psi)_{I+})\ v) \in \mathcal{R}_+\ \Psi\ v$
and *dom* $((\Psi)_{G+}) \subseteq$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall v \in$ *dom* $((\Psi)_{G+})$. *the* $(((\Psi)_{G+})\ v) \in \mathcal{R}_+\ \Psi\ v$
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* []) ⟷ $((\mathcal{R}_+\ \Psi\ v) \neq \{\})$
by (*cases ?D v*; (*auto simp*: *range-of′-def*))
} **note** *nb* = *this*
have *nb*$_1$: $\forall v \in$ *set ?vs*. *?D v* ≠ *None*
and $\forall op \in$ *set ?ops*. *is-valid-operator-sas-plus* Ψ *op*
and $\forall v$. (*?I v* ≠ *None*) = (*v* ∈ *set ?vs*)
and *nb*$_2$: $\forall v$. *?I v* ≠ *None* ⟶ *the* (*?I v*) ∈ *set* (*the* (*?D v*))
and $\forall v$. *?G v* ≠ *None* ⟶ *v* ∈ *set ?vs*
and *nb*$_3$: $\forall v$. *?G v* ≠ *None* ⟶ *the* (*?G v*) ∈ *set* (*the* (*?D v*))
using *assms*
unfolding *SAS-Plus-Representation.is-valid-problem-sas-plus-def Let-def list-all-iff ListMem-iff*
by *argo+*
then have *G3*: $\forall op \in$ *set* $((\Psi)_{\mathcal{O}+})$. *is-valid-operator-sas-plus* Ψ *op*
and *G4*: *dom* $((\Psi)_{I+})$ = *set* $((\Psi)_{\mathcal{V}+})$

```
    and G5: dom ((Ψ)_{G+}) ⊆ set ((Ψ)_{V+})
    unfolding variables-of-def operators-of-def
    by auto+
  moreover {
    fix v
    assume v ∈ set ((Ψ)_{V+})
    then have ?D v ≠ None
      using nb_1
      by force+
  } note G6 = this
  moreover {
    fix v
    assume v ∈ dom ((Ψ)_{I+})
    moreover have ((Ψ)_{I+}) v ≠ None
      using calculation
      by blast+
    moreover {
      have v ∈ set ((Ψ)_{V+})
        using G4 calculation(1)
        by argo
      then have sas-plus-problem.range-of Ψ v ≠ None
        using range-of-not-empty
        unfolding range-of'-def
        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)_{I+}) v) ∈ R_+ Ψ v
      using nb_2
      by force
  }
  moreover {
    fix v
    assume v ∈ dom ((Ψ)_{G+})
    then have ((Ψ)_{G+}) v ≠ None
      by blast
    moreover {
      have v ∈ set ((Ψ)_{V+})
        using G5 calculation(1)
        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
  :: ('variable, 'domain) sas-plus-problem ⇒ ('variable, 'domain) sas-plus-plan ⇒
bool
  where is-serial-solution-for-problem Ψ ψ
    ≡ let
        I = sas-plus-problem.initial-of Ψ
        ; G = sas-plus-problem.goal-of Ψ
        ; ops = sas-plus-problem.operators-of Ψ
      in G ⊆_m execute-serial-plan-sas-plus I ψ
        ∧ list-all (λop. ListMem op ops) ψ
```

```
context
begin

private lemma execute-operator-sas-plus-effect-i:
  assumes is-operator-applicable-in s op
    and ∀(v, a) ∈ set (effect-of op). ∀(v', a') ∈ set (effect-of op).
      v ≠ v' ∨ a = a'
    and(v, a) ∈ set (effect-of op)
  shows (s ≫+ op) v = Some a
proof −
  let ?effect = effect-of op
  have map-of ?effect v = Some a
    using map-of-constant-assignments-defined-if[OF assms(2, 3)] try0
    by blast
  thus ?thesis
    unfolding execute-operator-sas-plus-def map-add-def
    by fastforce
qed

private lemma  execute-operator-sas-plus-effect-ii:
  assumes is-operator-applicable-in s op
    and ∀(v', a') ∈ set (effect-of op). v' ≠ v
  shows (s ≫+ op) v = s v
proof −
  let ?effect = effect-of op
  {
    have v ∉ fst ‘ set ?effect
      using assms(2)
      by fastforce
    then have v ∉ dom (map-of ?effect)
      using dom-map-of-conv-image-fst[of ?effect]
      by argo
    hence (s ++ map-of ?effect) v = s v
      using map-add-dom-app-simps(3)[of v map-of ?effect s]
      by blast
  }
  thus ?thesis
    by fastforce
qed
```

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

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

The second property is the case if the operator doesn't have an effect for a

variable v.

theorem *execute-operator-sas-plus-effect*:
  assumes *is-operator-applicable-in s op*
    and $\forall (v,\ a) \in$ *set* (*effect-of op*).
      $\forall (v',\ a') \in$ *set* (*effect-of op*). $v \neq v' \vee a = a'$
  shows $(v,\ a) \in$ *set* (*effect-of op*)
      $\longrightarrow$ $(s \gg_+ op)\ v =$ *Some a*
    and $(\forall a.\ (v,\ a) \notin$ *set* (*effect-of op*))
      $\longrightarrow$ $(s \gg_+ op)\ v = s\ v$
proof −
  show $(v,\ a) \in$ *set* (*effect-of op*)
    $\longrightarrow$ $(s \gg_+ op)\ v =$ *Some a*
    using *execute-operator-sas-plus-effect-i*[*OF assms*(*1*, *2*)]
    by *blast*
next
  show $(\forall a.\ (v,\ a) \notin$ *set* (*effect-of op*))
    $\longrightarrow$ $(s \gg_+ op)\ v = s\ v$
    using *execute-operator-sas-plus-effect-ii*[*OF assms*(*1*)]
    by *blast*
qed

end

5.2 Parallel Execution Semantics

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

definition *are-all-operators-applicable-in*
  :: ('*variable*, '*domain*) *state*
    ⇒ ('*variable*, '*domain*) *sas-plus-operator list*
    ⇒ *bool*
  where *are-all-operators-applicable-in s ops*
    ≡ *list-all* (*is-operator-applicable-in s*) *ops*

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

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

$\equiv$ *list-all* ($\lambda op.$ *list-all* (*are-operator-effects-consistent op*) *ops*) *ops*

definition *execute-parallel-operator-sas-plus*
:: (*'variable*, *'domain*) *state*
$\Rightarrow$ (*'variable*, *'domain*) *sas-plus-operator list*
$\Rightarrow$ (*'variable*, *'domain*) *state*
where *execute-parallel-operator-sas-plus s ops*
$\equiv$ *foldl* (++) *s* (*map* (*map-of* $\circ$ *effect-of*) *ops*)

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

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

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

A plan ψ is a solution for a SAS+ problem Ψ if

1. starting from the initial state Ψ, SAS+ parallel plan execution reaches a state which satisfies the described goal state Ψ_{G+}; and,

2. all parallel operators *ops* in the plan ψ 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* Ψ ψ
≡ *let*
G = *sas-plus-problem.goal-of* Ψ
; *I* = *sas-plus-problem.initial-of* Ψ
; *Ops* = *sas-plus-problem.operators-of* Ψ
in *G* $\subseteq_m$ *execute-parallel-plan-sas-plus I* ψ
$\wedge$ *list-all* (λ*ops. list-all* (λ*op. ListMem op Ops*) *ops*) ψ

context
begin

lemma *execute-parallel-operator-sas-plus-cons*[*simp*]:
execute-parallel-operator-sas-plus s (*op # 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* ψ be a trace of a parallel SAS+ plan ψ with initial state *I*, then

- the head of the trace τ ! *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* τ − *1*—the parallel operator π ! k is executable (lemma **??**); and finally,
- for all $k <$ *length* τ, the parallel execution of the plan prefix *take k* ψ with initial state *I* equals the k-th element of the trace τ ! k (lemma **??**).

lemma *trace-parallel-plan-sas-plus-head-is-initial-state*:
trace-parallel-plan-sas-plus I ψ ! *0* = *I*
proof (*cases* ψ)
case (*Cons a list*)
then show *?thesis*
by (*cases are-all-operators-applicable-in I a* $\wedge$ *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* ψ) − *1*
shows *1* < *length* (*trace-parallel-plan-sas-plus I* ψ)
using *assms*
by *linarith*

lemma *length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one*:
shows *length* (*trace-parallel-plan-sas-plus I* ψ) $\leq$ *length* ψ + *1*
proof (*induction* ψ *arbitrary*: *I*)
case (*Cons a* ψ)
then show *?case*
proof (*cases are-all-operators-applicable-in I a* $\wedge$ *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* # ψ) = *I* # *trace-parallel-plan-sas-plus ?I'* ψ
using *True*
by *auto*
then have *length* (*trace-parallel-plan-sas-plus I* (*a* # ψ))
= *length* (*trace-parallel-plan-sas-plus ?I'* ψ) + *1*
by *simp*
moreover have *length* (*trace-parallel-plan-sas-plus ?I'* ψ) $\leq$ *length* ψ + *1*
using *Cons.IH*[*of ?I'*]
by *blast*
ultimately have *length* (*trace-parallel-plan-sas-plus I* (*a* # ψ)) $\leq$ *length* (*a* # ψ) + *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* ψ) − *1*
obtains *ops* ψ' **where** ψ = *ops* # ψ'
proof −
let *?*τ = *trace-parallel-plan-sas-plus I* ψ
have *length ?*τ $\leq$ *length* ψ + *1*
using *length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one*
by *fast*
then have *0* < *length* ψ
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 $k < length\ (trace\text{-}parallel\text{-}plan\text{-}sas\text{-}plus\ I\ \pi) - 1$
shows $are\text{-}all\text{-}operators\text{-}applicable\text{-}in\ (trace\text{-}parallel\text{-}plan\text{-}sas\text{-}plus\ I\ \pi\ !\ k)\ (\pi\ !\ k)$
$\wedge\ are\text{-}all\text{-}operator\text{-}effects\text{-}consistent\ (\pi\ !\ k)$
using *assms*
proof (*induction* π *arbitrary*: $I\ k$)
— NOTE Base case yields contradiction with assumption and can be left to automation.
case (*Cons* $a\ \pi$)
then show *?case*
proof (*cases are-all-operators-applicable-in I a* $\wedge$ *are-all-operator-effects-consistent a*)
case *True*
have *trace-parallel-plan-sas-plus-cons*: *trace-parallel-plan-sas-plus I* $(a\ \#\ \pi)$
$= I\ \#$ *trace-parallel-plan-sas-plus* (*execute-parallel-operator-sas-plus I a*) π
using *True*
by *simp*
then show *?thesis*
proof (*cases k*)
case *0*
have *trace-parallel-plan-sas-plus I* $(a\ \#\ \pi)\ !\ 0 = I$
using *trace-parallel-plan-sas-plus-cons*
by *simp*
moreover have $(a\ \#\ \pi)\ !\ 0 = a$
by *simp*
ultimately show *?thesis*
using *True 0*
by *presburger*
next
case (*Suc* k')
have *trace-parallel-plan-sas-plus I* $(a\ \#\ \pi)\ !\ Suc\ k'$
$=$ *trace-parallel-plan-sas-plus* (*execute-parallel-operator-sas-plus I a*) $\pi\ !\ k'$
using *trace-parallel-plan-sas-plus-cons*
by *simp*
moreover have $(a\ \#\ \pi)\ !\ Suc\ k' = \pi\ !\ k'$
by *simp*
moreover {
let *?I'* $=$ *execute-parallel-operator-sas-plus I a*
have *length* (*trace-parallel-plan-sas-plus I* $(a\ \#\ \pi)$)
$= 1 +$ *length* (*trace-parallel-plan-sas-plus ?I'* π)
using *trace-parallel-plan-sas-plus-cons*
by *auto*
then have $k' <$ *length* (*trace-parallel-plan-sas-plus ?I'* π) $- 1$
using *Cons.prems Suc*
unfolding *Suc-eq-plus1*
by *fastforce*
hence *are-all-operators-applicable-in*
(*trace-parallel-plan-sas-plus* (*execute-parallel-operator-sas-plus I a*) $\pi\ !\ k'$)
($\pi\ !\ k'$)
$\wedge$ *are-all-operator-effects-consistent* ($\pi\ !\ k'$)

```
          using Cons.IH[of k' execute-parallel-operator-sas-plus I a] Cons.prems
Suc trace-parallel-plan-sas-plus-cons
          by simp
      }
      ultimately show ?thesis
        using Suc
        by argo
    qed
  next
    case False
    then have trace-parallel-plan-sas-plus I (a # π) = [I]
      by force
    then have length (trace-parallel-plan-sas-plus 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

lemma trace-parallel-plan-sas-plus-prefix:
  assumes k < length (trace-parallel-plan-sas-plus I ψ)
  shows trace-parallel-plan-sas-plus I ψ ! k = execute-parallel-plan-sas-plus I (take
k ψ)
  using assms
proof (induction ψ arbitrary: I k)
  case (Cons a ψ)
  then show ?case
   proof (cases are-all-operators-applicable-in I a ∧ are-all-operator-effects-consistent
a)
    case True
    let ?σ = trace-parallel-plan-sas-plus I (a # ψ)
      and ?I' = execute-parallel-operator-sas-plus I a
    have σ-equals: ?σ = I # trace-parallel-plan-sas-plus ?I' ψ
      using True
      by auto
    then show ?thesis
      proof (cases k = 0)
        case False
        obtain k' where k-is-suc-of-k': k = Suc k'
          using not0-implies-Suc[OF False]
          by blast
        then have execute-parallel-plan-sas-plus I (take k (a # ψ))
          = execute-parallel-plan-sas-plus ?I' (take k' ψ)
          using True
          by simp
        moreover have trace-parallel-plan-sas-plus I (a # ψ) ! k
          = trace-parallel-plan-sas-plus ?I' ψ ! k'
```

```
        using σ-equals k-is-suc-of-k'
        by simp
      moreover {
        have k' < length (trace-parallel-plan-sas-plus ?I' ψ)
          using Cons.prems σ-equals k-is-suc-of-k'
          by force
        hence trace-parallel-plan-sas-plus ?I' ψ ! k'
          = execute-parallel-plan-sas-plus ?I' (take k' ψ)
          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-sas-plus I (a # ψ) = [I]
        using operator-precondition-violated
        by force
      moreover have execute-parallel-plan-sas-plus I (take k (a # ψ)) = I
        using Cons.prems operator-precondition-violated
        by force
      ultimately show ?thesis
        using Cons.prems nth-Cons-0
        by auto
    qed simp
  qed
qed simp

lemma trace-parallel-plan-sas-plus-step-effect-is:
  assumes k < length (trace-parallel-plan-sas-plus I ψ) − 1
  shows trace-parallel-plan-sas-plus I ψ ! Suc k
    = execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I ψ ! k) (ψ ! k)

proof −
  let ?τ = trace-parallel-plan-sas-plus I ψ
  let ?τ_k = ?τ ! k
    and ?τ_k' = ?τ ! 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-τ: Suc k < length ?τ
      using assms
      by linarith
    hence ?τ_k' = execute-parallel-plan-sas-plus I (take (Suc k) ψ)
      using trace-parallel-plan-sas-plus-prefix[of Suc k]
```

```
    by blast
  } note rewrite-goal = this
  have execute-parallel-plan-sas-plus I (take (Suc k) ψ)
   = execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I ψ ! k) (ψ ! k)

    using assms
    proof (induction k arbitrary: I ψ)
      case 0
      obtain ops ψ' where ψ-is: ψ = ops # ψ'
          using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
0.prems]
        by force
      {
        have take (Suc 0) ψ  = [ψ ! 0]
          using ψ-is
          by simp
        hence execute-parallel-plan-sas-plus I (take (Suc 0) ψ)
          = execute-parallel-plan-sas-plus I [ψ ! 0]
          by argo
      }
      moreover {
        have trace-parallel-plan-sas-plus I ψ ! 0 = I
          using trace-parallel-plan-sas-plus-head-is-initial-state.
        moreover {
          have are-all-operators-applicable-in I (ψ ! 0)
            and are-all-operator-effects-consistent (ψ ! 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 [ψ ! 0]
            = execute-parallel-operator-sas-plus I (ψ ! 0)
            by simp
        }
      ultimately have execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus
I ψ ! 0)
            (ψ ! 0)
          = execute-parallel-plan-sas-plus I [ψ ! 0]
          by argo
      }
      ultimately show ?case
        by argo
    next
      case (Suc k)
      obtain ops ψ' where ψ-is: ψ = ops # ψ'
          using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
Suc.prems]
        by blast
      let ?I' = execute-parallel-operator-sas-plus I ops
      have execute-parallel-plan-sas-plus I (take (Suc (Suc k)) ψ)
```

```
      = execute-parallel-plan-sas-plus ?I' (take (Suc k) ψ')
      using Suc.prems ψ-is
      by fastforce
    moreover {
      thm Suc.IH[of ]
      have length (trace-parallel-plan-sas-plus I ψ)
        = 1 + length (trace-parallel-plan-sas-plus ?I' ψ')
        using ψ-is Suc.prems
        by fastforce
      moreover have k < length (trace-parallel-plan-sas-plus ?I' ψ') − 1
        using Suc.prems calculation
        by fastforce
      ultimately have execute-parallel-plan-sas-plus ?I' (take (Suc k) ψ') =
        execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus ?I' ψ' ! k)
        (ψ' ! k)
        using Suc.IH[of ?I' ψ']
        by blast
    }
    moreover have execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus
?I' ψ' ! k)
        (ψ' ! k)
      = execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I ψ ! Suc k)
        (ψ ! Suc k)
      using Suc.prems ψ-is
      by auto
    ultimately show ?case
      by argo
  qed
 thus ?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.

```
lemma execute-parallel-plan-sas-plus-reaches-goal-iff-goal-is-last-element-of-trace:
  G ⊆m execute-parallel-plan-sas-plus I ψ
  ⟷ G ⊆m last (trace-parallel-plan-sas-plus I ψ)
proof −
 let ?τ = trace-parallel-plan-sas-plus I ψ
 show ?thesis
 proof (rule iffI)
  assume G ⊆m execute-parallel-plan-sas-plus I ψ
  thus G ⊆m last ?τ
   proof (induction ψ arbitrary: I)
    — NOTE Base case follows from simplification.
    case (Cons ops ψ)
```

```
      show ?case
        proof (cases are-all-operators-applicable-in I ops
          ∧ are-all-operator-effects-consistent ops)
          case True
          let ?s = execute-parallel-operator-sas-plus I ops
          {
            have G ⊆m execute-parallel-plan-sas-plus ?s ψ
              using True Cons.prems
              by simp
            hence G ⊆m last (trace-parallel-plan-sas-plus ?s ψ)
              using Cons.IH
              by auto
          }
          moreover {
            have trace-parallel-plan-sas-plus I (ops # ψ)
              = I # trace-parallel-plan-sas-plus ?s ψ
              using True
              by simp
            moreover have trace-parallel-plan-sas-plus ?s ψ ≠ []
              using trace-parallel-plan-sas-plus.elims
              by blast
            ultimately have last (trace-parallel-plan-sas-plus I (ops # ψ))
              = last (trace-parallel-plan-sas-plus ?s ψ)
              using last-ConsR
                by simp
          }
          ultimately show ?thesis
            by argo
        next
          case False
          then have G ⊆m I
            using Cons.prems
            by force
          thus ?thesis
            using False
            by force
        qed
    qed force
next
  assume G ⊆m last ?τ
  thus G ⊆m execute-parallel-plan-sas-plus I ψ
    proof (induction ψ arbitrary: I)
      case (Cons ops ψ)
      thus ?case
        proof (cases are-all-operators-applicable-in I ops
          ∧ are-all-operator-effects-consistent ops)
          case True
          let ?s = execute-parallel-operator-sas-plus I ops
          {
```

```
        have trace-parallel-plan-sas-plus I (ops # ψ)
          = I # trace-parallel-plan-sas-plus ?s ψ
          using True
          by simp
        moreover have trace-parallel-plan-sas-plus ?s ψ ≠ []
          using trace-parallel-plan-sas-plus.elims
          by blast
        ultimately have last (trace-parallel-plan-sas-plus I (ops # ψ))
          = last (trace-parallel-plan-sas-plus ?s ψ)
          using last-ConsR
          by simp
        hence G ⊆_m execute-parallel-plan-sas-plus ?s ψ
          using Cons.IH[of ?s] Cons.prems
          by argo
      }
      moreover have execute-parallel-plan-sas-plus I (ops # ψ)
        = execute-parallel-plan-sas-plus ?s ψ
        using True
        by force
      ultimately show ?thesis
        by argo
    next
      case False
      have G ⊆_m I
        using Cons.prems False
        by simp
      thus ?thesis
        using False
        by force
    qed
  qed simp
 qed
qed

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

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

 by presburger

end
```

5.3 Serializable Parallel Plans

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

1. the embedding *List-Supplement.embed* ψ of ψ is a parallel solution for Ψ (lemma **??**); and conversely that,
2. a parallel solution to Ψ that has the form of an embedded serial plan can be concatenated to obtain a serial solution (lemma **??**).

context
begin

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

lemma *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii*:
fixes *I* :: (*'variable*, *'domain*) *state*
assumes $\forall$ *op* $\in$ *set* ψ. *are-operator-effects-consistent op op*
and $G \subseteq_m execute\text{-}serial\text{-}plan\text{-}sas\text{-}plus\ I\ \psi$
shows $G \subseteq_m execute\text{-}parallel\text{-}plan\text{-}sas\text{-}plus\ I\ (embed\ \psi)$
using *assms*
proof (*induction* ψ *arbitrary*: *I*)
case (*Cons op* ψ)
show *?case*
proof (*cases are-all-operators-applicable-in I* [*op*])
case *True*
let *?J* = *execute-operator-sas-plus I op*

```
      let ?J' = execute-parallel-operator-sas-plus I [op]
    have SAS-Plus-Representation.is-operator-applicable-in I op
      using True
      unfolding are-all-operators-applicable-in-def list-all-iff
      by force
    moreover have G ⊆m execute-serial-plan-sas-plus ?J ψ
      using Cons.prems(2) calculation(1)
      by simp
    moreover have are-all-operator-effects-consistent [op]
      unfolding are-all-operator-effects-consistent-def list-all-iff Let-def
      using Cons.prems(1)
      by simp
    moreover have execute-parallel-plan-sas-plus I ([op] # embed ψ)
      = execute-parallel-plan-sas-plus ?J' (embed ψ)
      using True calculation(3)
      by simp
    moreover {
      have is-operator-applicable-in I op
        are-operator-effects-consistent op op
        using True Cons.prems(1)
        unfolding are-all-operators-applicable-in-def
          SAS-Plus-Representation.is-operator-applicable-in-def list-all-iff
        by fastforce+
      hence ?J = ?J'
        using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i
          calculation(1)
        by blast
    }
    ultimately show ?thesis
      using Cons.IH[of ?J] Cons.prems(1)
      by simp
  next
    case False
    moreover have ¬is-operator-applicable-in I op
      using calculation
      unfolding are-all-operators-applicable-in-def
        SAS-Plus-Representation.is-operator-applicable-in-def list-all-iff
      by fastforce
    moreover have G ⊆m I
      using Cons.prems(2) calculation(2)
      unfolding is-operator-applicable-in-def
      by simp
    moreover have execute-parallel-plan-sas-plus I ([op] # embed ψ) = I
      using calculation(1)
      by fastforce
    ultimately show ?thesis
      by force
  qed
qed simp
```

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

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

theorem *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus*:
assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* Ψ ψ
shows *is-parallel-solution-for-problem* Ψ (*embed* ψ)
proof −
let $?ops = sas\text{-}plus\text{-}problem.operators\text{-}of\ \Psi$
and $?\psi' = embed\ \psi$
{
thm *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii*[*OF*]

```
    have (Ψ)_{G+} ⊆_m execute-serial-plan-sas-plus ((Ψ)_{I+}) ψ
      using assms(2)
      unfolding is-serial-solution-for-problem-def Let-def
      by simp
    moreover have ∀ op ∈ set ψ. are-operator-effects-consistent op op
     using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iii[OF assms]..
    ultimately have (Ψ)_{G+} ⊆_m execute-parallel-plan-sas-plus ((Ψ)_{I+}) ?ψ'
      using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii
      by blast
  }
  moreover {
    have ∀ op ∈ set ψ. op ∈ set ((Ψ)_{O+})
      using assms(2)
      unfolding is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
      by fastforce
    hence ∀ ops ∈ set ?ψ'. ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
      using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iv
      by blast
  }
  ultimately show ?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 Ψ :: ('v, 'd) sas-plus-problem
  assumes ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
  shows ∀ op ∈ set (concat π). op ∈ set ((Ψ)_{O+})
proof −
  {
    fix op
    assume op ∈ set (concat π)
    moreover have op ∈ (⋃ ops ∈ set π. set ops)
      using calculation
      unfolding set-concat.
    then obtain ops where ops ∈ set π and op ∈ set ops
      using UN-iff
      by blast
    ultimately have op ∈ set ((Ψ)_{O+})
      using assms
      by blast
  }
  thus ?thesis..
qed

lemma flattening-lemma-ii:
  fixes I :: ('variable, 'domain) state
```

```
  assumes ∀ ops ∈ set ψ. ∃ op. ops = [op] ∧ is-valid-operator-sas-plus Ψ op
    and G ⊆m execute-parallel-plan-sas-plus I ψ
  shows G ⊆m execute-serial-plan-sas-plus I (concat ψ)
proof −
  show ?thesis
    using assms
    proof (induction ψ arbitrary: I)
      case (Cons ops ψ)
     obtain op where ops-is: ops = [op] and is-valid-op: is-valid-operator-sas-plus
Ψ op
        using Cons.prems(1)
        by auto
      then show ?case
        proof (cases are-all-operators-applicable-in I ops)
          case True
          let ?J = execute-parallel-operator-sas-plus I [op]
            and ?J' = execute-operator-sas-plus I op
          have nb1: is-operator-applicable-in I op
            using True ops-is
           unfolding are-all-operators-applicable-in-def is-operator-applicable-in-def

              list-all-iff
            by force
          have nb2: are-operator-effects-consistent op op
            unfolding are-operator-effects-consistent-def list-all-iff Let-def
            using is-valid-operator-sas-plus-then(6)[OF is-valid-op]
            by blast
          have are-all-operator-effects-consistent ops
            using ops-is
            unfolding are-all-operator-effects-consistent-def list-all-iff
            using nb2
            by force
          moreover have G ⊆m execute-parallel-plan-sas-plus ?J ψ
            using Cons.prems(2) True calculation ops-is
            by fastforce
          moreover have execute-serial-plan-sas-plus I (concat (ops # ψ))
              = execute-serial-plan-sas-plus ?J' (concat ψ)
              using ops-is nb1 is-operator-applicable-in-def
              by simp
          moreover have ?J = ?J'
             using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-i[OF
nb1 nb2]
            by simp
          ultimately show ?thesis
            using Cons.IH[of ?J] Cons.prems(1)
            by force
        next
          case False
          moreover have G ⊆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 # ψ))
        = execute-serial-plan-sas-plus I (op # concat ψ)
        using ops-is
        by force
      ultimately have execute-serial-plan-sas-plus I (concat (ops # ψ)) = 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 Ψ
    and ∀ ops ∈ set ψ. ∃ op. ops = [op]
    and is-parallel-solution-for-problem Ψ ψ
  shows is-serial-solution-for-problem Ψ (concat ψ)
proof −
  let ?ψ' = concat ψ
  {
    have ∀ ops ∈ set ψ. ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
      using assms(3)
      unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
      by force
    hence ∀ op ∈ set ?ψ'. op ∈ set ((Ψ)_{O+})
      using flattening-lemma-i
      by blast
  }
  moreover {
    {
      fix ops
      assume ops ∈ set ψ
      moreover obtain op where ops = [op]
        using assms(2) calculation
        by blast
      moreover have op ∈ set ((Ψ)_{O+})
        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 $(\Psi)_{G+} \subseteq_m$ *execute-parallel-plan-sas-plus* $((\Psi)_{I+})$ ψ
using *assms*(*3*)
unfolding *is-parallel-solution-for-problem-def*
by *fastforce*
ultimately have $(\Psi)_{G+} \subseteq_m$ *execute-serial-plan-sas-plus* $((\Psi)_{I+})$ $?\psi'$
using *flattening-lemma-ii*
by *blast*
}
ultimately show *is-serial-solution-for-problem* Ψ $?\psi'$
unfolding *is-serial-solution-for-problem-def list-all-iff ListMem-iff*
by *fastforce*
qed
end

5.4 Auxiliary lemmata on SAS+

context
begin

— Relate the locale definition *range-of* with its corresponding implementation for valid operators and given an effect (*v*, *a*).
lemma *is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op*:
assumes *is-valid-operator-sas-plus* Ψ *op*
and (*v*, *a*) ∈ *set* (*precondition-of op*) ∨ (*v*, *a*) ∈ *set* (*effect-of op*)
shows $(\mathcal{R}_+\ \Psi\ v)$ = *set* (*the* (*sas-plus-problem.range-of* Ψ *v*))
proof −
consider (*A*) (*v*, *a*) ∈ *set* (*precondition-of op*)
| (*B*) (*v*, *a*) ∈ *set* (*effect-of op*)
using *assms*(*2*)**..**
thus *?thesis*
proof (*cases*)
case *A*
then have $(\mathcal{R}_+\ \Psi\ v) \neq \{\}$ **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
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 set ((\Psi)_{\mathcal{V}+})
  shows set (the (sas-plus-problem.range-of \Psi v)) = \mathcal{R}_+ \Psi v
proof-
  have v \in set((\Psi)_{\mathcal{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 set ((\Psi)_{\mathcal{V}+}). (\mathcal{R}_+ \Psi v) \neq {}
    and \forall op \in set ((\Psi)_{\mathcal{O}+}). is-valid-operator-sas-plus \Psi op
    and dom ((\Psi)_{I+}) = set ((\Psi)_{\mathcal{V}+})
    and \forall v \in dom ((\Psi)_{I+}). the (((\Psi)_{I+}) v) \in \mathcal{R}_+ \Psi v
    and dom ((\Psi)_{G+}) \subseteq set ((\Psi)_{\mathcal{V}+})
    and \forall v \in 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* ⇒ '*variable* ⇒ ('*variable* × '*domain*) *list*

where *possible-assignments-for* Ψ *v* ≡ [(*v*, *a*). *a* ← *the* (*range-of* Ψ *v*)]

definition *all-possible-assignments-for*
:: ('*variable*, '*domain*) *sas-plus-problem* ⇒ ('*variable* × '*domain*) *list*
where *all-possible-assignments-for* Ψ
≡ *concat* [*possible-assignments-for* Ψ *v*. *v* ← *variables-of* Ψ]

definition *state-to-strips-state*
:: ('*variable*, '*domain*) *sas-plus-problem*
⇒ ('*variable*, '*domain*) *state*
⇒ ('*variable*, '*domain*) *assignment strips-state*
(φ*S* - - *99*)
where *state-to-strips-state* Ψ *s*
≡ *let defined* = *filter* (λ*v*. *s v* ≠ *None*) (*variables-of* Ψ) *in*
map-of (*map* (λ(*v*, *a*). ((*v*, *a*), *the* (*s v*) = *a*))
(*concat* [*possible-assignments-for* Ψ *v*. *v* ← *defined*]))

definition *sasp-op-to-strips*
:: ('*variable*, '*domain*) *sas-plus-problem*
⇒ ('*variable*, '*domain*) *sas-plus-operator*
⇒ ('*variable*, '*domain*) *assignment strips-operator*
(φ*O* - - *99*)
where *sasp-op-to-strips* Ψ *op* ≡ *let*
pre = *precondition-of op*
; *add* = *effect-of op*
; *delete* = [(*v*, *a*′). (*v*, *a*) ← *effect-of op*, *a*′ ← *filter* ((≠) *a*) (*the* (*range-of* Ψ *v*))]
in STRIPS-Representation.operator-for pre add delete

definition *sas-plus-problem-to-strips-problem*

```
 :: ('variable, 'domain) sas-plus-problem ⇒ ('variable, 'domain) assignment strips-problem

 (φ - 99)
 where sas-plus-problem-to-strips-problem Ψ ≡ let
     vs = [as. v ← variables-of Ψ, as ← (possible-assignments-for Ψ) v]
     ; ops = map (sasp-op-to-strips Ψ) (operators-of Ψ)
     ; I = state-to-strips-state Ψ (initial-of Ψ)
     ; G = state-to-strips-state Ψ (goal-of Ψ)
   in STRIPS-Representation.problem-for vs ops I G

definition sas-plus-parallel-plan-to-strips-parallel-plan
 :: ('variable, 'domain) sas-plus-problem
   ⇒ ('variable, 'domain) sas-plus-parallel-plan
   ⇒ ('variable × 'domain) strips-parallel-plan
 (φ_P - - 99)
 where sas-plus-parallel-plan-to-strips-parallel-plan Ψ ψ
   ≡ [[sasp-op-to-strips Ψ op. op ← ops]. ops ← ψ]


definition strips-state-to-state
 :: ('variable, 'domain) sas-plus-problem
   ⇒ ('variable, 'domain) assignment strips-state
   ⇒ ('variable, 'domain) state
 (φ_S^-1 - - 99)
 where strips-state-to-state Ψ s
   ≡ map-of (filter (λ(v, a). s (v, a) = Some True) (all-possible-assignments-for
Ψ))


definition strips-op-to-sasp
 :: ('variable, 'domain) sas-plus-problem
   ⇒ ('variable × 'domain) strips-operator
   ⇒ ('variable, 'domain) sas-plus-operator
 (φ_O^-1 - - 99)
 where strips-op-to-sasp Ψ op
   ≡ let
       precondition = strips-operator.precondition-of op
       ; effect = strips-operator.add-effects-of op
     in (| precondition-of = precondition, effect-of = effect |)


definition 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. 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. [5]

definition *range-of-strips* Π $x \equiv \{$ *True*, *False* $\}$

context
begin

— Set-up simp rules.
lemma[*simp*]:
(*φ* Ψ) = (*let*
vs = [*as*. *v* ← *variables-of* Ψ, *as* ← (*possible-assignments-for* Ψ) *v*]
; *ops* = *map* (*sasp-op-to-strips* Ψ) (*operators-of* Ψ)
; *I* = *state-to-strips-state* Ψ (*initial-of* Ψ)
; *G* = *state-to-strips-state* Ψ (*goal-of* Ψ)
in STRIPS-Representation.problem-for vs ops I G)
and (*φ*$_S$ Ψ *s*)
= (*let defined* = *filter* (λ*v*. *s v* ≠ *None*) (*variables-of* Ψ) *in*
map-of (*map* (λ(*v*, *a*). ((*v*, *a*), *the* (*s v*) = *a*))
(*concat* [*possible-assignments-for* Ψ *v*. *v* ← *defined*])))
and (*φ*$_O$ Ψ *op*)
= (*let*
pre = *precondition-of op*
; *add* = *effect-of op*
; *delete* = [(*v*, *a*′). (*v*, *a*) ← *effect-of op*, *a*′ ← *filter* ((≠) *a*) (*the* (*range-of* Ψ *v*))]
in STRIPS-Representation.operator-for pre add delete)
and (*φ*$_P$ Ψ *ψ*) = [[*φ*$_O$ Ψ *op*. *op* ← *ops*]. *ops* ← *ψ*]
and (*φ*$_S{}^{-1}$ Ψ *s*′)= *map-of* (*filter* (λ(*v*, *a*). *s*′ (*v*, *a*) = *Some True*)
(*all-possible-assignments-for* Ψ))
and (*φ*$_O{}^{-1}$ Ψ *op*′) = (*let*
precondition = *strips-operator.precondition-of op*′
; *effect* = *strips-operator.add-effects-of op*′
in (| *precondition-of* = *precondition*, *effect-of* = *effect* |))
and (*φ*$_P{}^{-1}$ Ψ *π*) = [[*φ*$_O{}^{-1}$ Ψ *op*. *op* ← *ops*]. *ops* ← *π*]
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

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

SAS-Plus-STRIPS.sasp-op-to-strips-def
sasp-op-to-strips-def
SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
sas-plus-parallel-plan-to-strips-parallel-plan-def
SAS-Plus-STRIPS.strips-state-to-state-def
strips-state-to-state-def
SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
strips-parallel-plan-to-sas-plus-parallel-plan-def
by *blast+*

lemmas [*simp*] = *range-of'-def*

lemma *is-valid-problem-sas-plus-dom-sas-plus-problem-range-of*:
assumes *is-valid-problem-sas-plus* Ψ
shows ∀ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$). *v* ∈ *dom* (*sas-plus-problem.range-of* Ψ)
using *assms*(*1*) *is-valid-problem-sas-plus-then*(*1*)
unfolding *is-valid-problem-sas-plus-def*
by (*meson domIff list.pred-set*)

lemma *possible-assignments-for-set-is*:
assumes *v* ∈ *dom* (*sas-plus-problem.range-of* Ψ)
shows *set* (*possible-assignments-for* Ψ *v*)
= { (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* }
proof −
have *sas-plus-problem.range-of* Ψ *v* ≠ *None*
using *assms*(*1*)
by *auto*
thus *?thesis*
unfolding *possible-assignments-for-def*
by *fastforce*
qed

lemma *all-possible-assignments-for-set-is*:
assumes ∀ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$). *range-of* Ψ *v* ≠ *None*
shows *set* (*all-possible-assignments-for* Ψ)
= (⋃ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$). { (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* })
proof −
let *?vs* = *variables-of* Ψ
have *set* (*all-possible-assignments-for* Ψ) =
(⋃(*set* ‘ (λ*v*. *map* (λ(*v*, *a*). (*v*, *a*)) (*possible-assignments-for* Ψ *v*)) ‘ *set ?vs*))
unfolding *all-possible-assignments-for-def set-concat*
using *set-map*
by *auto*
also have ... = (⋃((λ*v*. *set* (*possible-assignments-for* Ψ *v*)) ‘ *set ?vs*))
using *image-comp set-map*
by *simp*

also have ... = ($\bigcup$((λv. { (v, a) | a. a ∈ $\mathcal{R}_+$ Ψ v }) ‘ set ?vs))
using *possible-assignments-for-set-is assms*
by *fastforce*
finally show *?thesis*
by *force*
qed

lemma *state-to-strips-state-dom-is-i*[*simp*]:
assumes ∀ v ∈ *set* ((Ψ)$_{\mathcal{V}+}$). v ∈ *dom* (*sas-plus-problem.range-of* Ψ)
shows *set* (*concat*
[*possible-assignments-for* Ψ v. v ← *filter* (λv. s v ≠ *None*) (*variables-of* Ψ)])
= ($\bigcup$ v ∈ { v | v. v ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ s v ≠ *None* }.
{ (v, a) | a. a ∈ $\mathcal{R}_+$ Ψ v })
proof −
let *?vs* = *variables-of* Ψ
let *?defined* = *filter* (λv. s v ≠ *None*) *?vs*
let *?l* = *concat* [*possible-assignments-for* Ψ v. v ← *?defined*]
have *nb*: *set ?defined* = { v | v. v ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ s v ≠ *None* }
unfolding *set-filter*
by *force*
have *set ?l* = $\bigcup$(*set* ‘ *set* (*map* (*possible-assignments-for* Ψ) *?defined*))
unfolding *set-concat image-Union*
by *blast*
also have ... = $\bigcup$(*set* ‘ (*possible-assignments-for* Ψ) ‘ *set ?defined*)
unfolding *set-map*
by *blast*
also have ... = ($\bigcup$ v ∈ *set ?defined*. *set* (*possible-assignments-for* Ψ v))
by *blast*
also have ... = ($\bigcup$ v ∈ { v | v. v ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ s v ≠ *None* }.
set (*possible-assignments-for* Ψ v))
using *nb*
by *argo*
finally show *?thesis*
using *possible-assignments-for-set-is*
is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(*1*)
by *fastforce*
qed

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* Ψ
shows *dom* (φ_S Ψ s)
= ($\bigcup$ v ∈ { v | v. v ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ s v ≠ *None* }.
{ (v, a) | a. a ∈ $\mathcal{R}_+$ Ψ v })
proof −
let *?vs* = *variables-of* Ψ
let *?l* = *concat* [*possible-assignments-for* Ψ v. v ← *filter* (λv. s v ≠ *None*) *?vs*]
have *nb*: ∀ v ∈ *set* ((Ψ)$_{\mathcal{V}+}$). v ∈ *dom* (*sas-plus-problem.range-of* Ψ)

```
    using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)
    by fastforce
  have dom (φ_S Ψ s) = fst ‘ set (map (λ(v, a). ((v, a), the (s v) = a)) ?l)
    unfolding state-to-strips-state-def
      SAS-Plus-STRIPS.state-to-strips-state-def
    using dom-map-of-conv-image-fst[of map (λ(v, a). ((v, a), the (s v) = a)) ?l]
    by presburger
  also have ... = fst ‘ (λ(v, a). ((v, a), the (s v) = a)) ‘ set ?l
    unfolding set-map
    by blast
  also have ... = (λ(v, a). fst ((v, a), the (s v) = a)) ‘ set ?l
    unfolding image-comp[of fst λ(v, a). ((v, a), the (s v) = a)] comp-apply[of
        fst λ(v, a). ((v, a), the (s v) = a)] prod.case-distrib
    by blast
  finally show ?thesis
    unfolding state-to-strips-state-dom-is-i[OF nb]
    by force
qed

corollary state-to-strips-state-dom-element-iff:
  assumes is-valid-problem-sas-plus Ψ
  shows (v, a) ∈ dom (φ_S Ψ s) ⟷ v ∈ set ((Ψ)_{V+})
    ∧ s v ≠ None
    ∧ a ∈ R_+ Ψ v
proof −
  let ?vs = variables-of Ψ
    and ?s' = φ_S Ψ s
  show ?thesis
    proof (rule iffI)
      assume (v, a) ∈ dom (φ_S Ψ s)
      then have v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None }
          and a ∈ R_+ Ψ v
        unfolding state-to-strips-state-dom-is[OF assms(1)]
        by force+
      moreover have v ∈ set ?vs and s v ≠ None
        using calculation(1)
        by fastforce+
      ultimately show
        v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None ∧ a ∈ R_+ Ψ v
        by force
    next
      assume v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None ∧ a ∈ R_+ Ψ v
      then have v ∈ set ((Ψ)_{V+})
        and s v ≠ None
        and a-in-range-of-v: a ∈ R_+ Ψ v
        by simp+
      then have v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None }
        by force
      thus (v, a) ∈ dom (φ_S Ψ s)
```

```
      unfolding state-to-strips-state-dom-is[OF assms(1)]
      using a-in-range-of-v
      by blast
    qed
qed

lemma state-to-strips-state-range-is:
  assumes is-valid-problem-sas-plus Ψ
    and (v, a) ∈ dom (φ_S Ψ s)
  shows (φ_S Ψ s) (v, a) = Some (the (s v) = a)
proof −
  let ?vs = variables-of Ψ
  let ?s' = φ_S Ψ s
    and ?defined = filter (λv. s v ≠ None) ?vs
  let ?l = concat [possible-assignments-for Ψ v. v ← ?defined]
  have v-in-set-vs: v ∈ set ?vs
    and s-of-v-is-not-None: s v ≠ None
    and a-in-range-of-v: a ∈ R_+ Ψ v
    using assms(2)
    unfolding state-to-strips-state-dom-is[OF assms(1)]
    by fastforce+
  moreover {
    have ∀ v ∈ set ((Ψ)_V+). v ∈ dom (sas-plus-problem.range-of Ψ)
      using assms(1) is-valid-problem-sas-plus-then(1)
      unfolding is-valid-problem-sas-plus-def
      by fastforce
    moreover have (v, a) ∈ set ?l
      unfolding state-to-strips-state-dom-is-i[OF calculation(1)]
      using s-of-v-is-not-None a-in-range-of-v v-in-set-vs
      by fastforce
    moreover have set ?l ≠ {}
      using calculation
      by fastforce
    — TODO slow.
    ultimately have (φ_S Ψ s) (v, a) = Some (the (s v) = a)
      using map-of-from-function-graph-is-some-if[of
          ?l (v, a) λ(v, a). 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 ?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* Ψ
    and $(v,\ a) \in dom\ (\varphi_S\ \Psi\ s)$
    and $(v,\ a') \in dom\ (\varphi_S\ \Psi\ s)$
    and $(\varphi_S\ \Psi\ s)\ (v,\ a) = Some\ True$
    and $(\varphi_S\ \Psi\ s)\ (v,\ a') = Some\ True$
  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* Ψ *op*
  shows *set* (*strips-operator.delete-effects-of* $(\varphi_O\ \Psi\ op)$)
    $= (\bigcup (v,\ a) \in set\ (effect\text{-}of\ op).\ \{\ (v,\ a')\ |\ a'.\ a' \in (\mathcal{R}_+\ \Psi\ v) \wedge a' \neq a\ \})$
proof −
  let *?D* = *range-of* Ψ
    and *?effect* = *effect-of op*
  let *?delete* = $[(v,\ a').\ (v,\ a) \leftarrow$ *?effect*, $a' \leftarrow$ *filter* $((\neq)\ a)\ (the\ (?D\ v))]$
  {
    fix *v a*
    assume $(v,\ a) \in$ *set ?effect*
    then have $(\mathcal{R}_+\ \Psi\ v) =$ *set* $(the\ (?D\ v))$
      using *assms*
      using *is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op*
      by *fastforce*
    hence *set* $(filter\ ((\neq)\ a)\ (the\ (?D\ v))) = \{\ a' \in \mathcal{R}_+\ \Psi\ v.\ a' \neq a\ \}$
      unfolding *set-filter*
      by *blast*
  } **note** *nb* = *this*
  {
    — TODO slow.
    have *set ?delete* $= \bigcup$(*set* ‘ $(\lambda(v,\ a).\ map\ (Pair\ v)\ (filter\ ((\neq)\ a)\ (the\ (?D\ v))))$

      ‘ (*set ?effect*))
      using *set-concat*
      by *simp*
    also have ... $= \bigcup((\lambda(v,\ a).\ Pair\ v$ ‘ *set* $(filter\ ((\neq)\ a)\ (the\ (?D\ v))))$
      ‘ (*set ?effect*))
      unfolding *image-comp*[*of set*] *set-map*
      by *auto*

— TODO slow.
also have ... = ($\bigcup (v,\ a) \in$ *set ?effect. Pair v* ‘ $\{\ a' \in \mathcal{R}_+\ \Psi\ v.\ a' \neq a\ \}$)
using *nb*
by *fast*
finally have *set ?delete* = ($\bigcup (v,\ a) \in$ *set ?effect.*
$\{\ (v,\ a')\ |\ a'.\ a' \in (\mathcal{R}_+\ \Psi\ v) \wedge a' \neq a\ \}$)
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 Π is the set of all possible assignments that are possible using the variables of $\mathcal{V}$ and the corresponding domains.
assumes *is-valid-problem-sas-plus* Ψ
shows *set* $((\varphi\ \Psi)_{\mathcal{V}})$ = ($\bigcup v \in$ *set* $((\Psi)_{\mathcal{V}+}).\ \{\ (v,\ a)\ |\ a.\ a \in \mathcal{R}_+\ \Psi\ v\ \}$)
proof −
let *?*Π = $\varphi\ \Psi$
and *?vs = variables-of* Ψ
{
have *set* (*strips-problem.variables-of ?*Π)
= *set* [*as. v* ← *?vs, as* ← *possible-assignments-for* Ψ *v*]
unfolding *sas-plus-problem-to-strips-problem-def*
SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
by *force*
also have ... = ($\bigcup$(*set* ‘ (λ*v. possible-assignments-for* Ψ *v*) ‘ *set ?vs*))
using *set-concat*
by *auto*
also have ... = ($\bigcup$((*set* ∘ *possible-assignments-for* Ψ) ‘ *set ?vs*))
using *image-comp*[*of set* λ*v. possible-assignments-for* Ψ *v set ?vs*]
by *argo*
finally have *set* (*strips-problem.variables-of ?*Π)
= ($\bigcup v \in$ *set ?vs. set* (*possible-assignments-for* Ψ *v*))
unfolding *o-apply*
by *blast*
}
moreover have $\forall\, v \in$ *set ?vs. v* $\in$ *dom* (*sas-plus-problem.range-of* Ψ)
using *is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms*
by *force*
ultimately show *?thesis*
using *possible-assignments-for-set-is*
by *force*
qed

corollary *sas-plus-problem-to-strips-problem-variable-set-element-iff*:
assumes *is-valid-problem-sas-plus* Ψ

shows $(v,\ a) \in set\ ((\varphi\ \Psi)_{\mathcal{V}}) \longleftrightarrow v \in set\ ((\Psi)_{\mathcal{V}+}) \wedge a \in \mathcal{R}_+\ \Psi\ v$
unfolding *sas-plus-problem-to-strips-problem-variable-set-is*[*OF assms*]
by *fast*

lemma *sasp-op-to-strips-effect-consistent*:
assumes $op = \varphi_O\ \Psi\ op'$
and $op' \in set\ ((\Psi)_{\mathcal{O}+})$
and *is-valid-operator-sas-plus* $\Psi\ op'$
shows $(v,\ a) \in set\ (add\text{-}effects\text{-}of\ op) \longrightarrow (v,\ a) \notin set\ (delete\text{-}effects\text{-}of\ op)$
and $(v,\ a) \in set\ (delete\text{-}effects\text{-}of\ op) \longrightarrow (v,\ a) \notin set\ (add\text{-}effects\text{-}of\ op)$
proof −
have *nb*: $(\forall (v,\ a) \in set\ (effect\text{-}of\ op').\ \forall (v',\ a') \in set\ (effect\text{-}of\ op').\ v \neq v' \vee a = a')$
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*: $(v,\ a) \in set\ (add\text{-}effects\text{-}of\ op)$
have $(v,\ a) \notin set\ (delete\text{-}effects\text{-}of\ op)$
proof (*rule ccontr*)
assume $\neg(v,\ a) \notin set\ (delete\text{-}effects\text{-}of\ op)$
moreover have $(v,\ a) \in$
$(\bigcup (v,\ a') \in set\ (effect\text{-}of\ op').\ \{\ (v,\ a'')$
$\mid a''.\ a'' \in (\mathcal{R}_+\ \Psi\ v) \wedge a'' \neq a'\ \})$
using *calculation sasp-op-to-strips-set-delete-effects-is*
assms
by *blast*
moreover obtain a' **where** $(v,\ a') \in set\ (effect\text{-}of\ op')$ **and** $a \neq a'$
using *calculation*
by *blast*
moreover have $(v,\ a') \in set\ (add\text{-}effects\text{-}of\ op)$
using *assms*(1) *calculation*(3)
unfolding *sasp-op-to-strips-def*
SAS-Plus-STRIPS.sasp-op-to-strips-def
Let-def
by *fastforce*
moreover have $(v,\ a) \in set\ (effect\text{-}of\ op')$ **and** $(v,\ a') \in set\ (effect\text{-}of\ op')$
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

```
  }
  moreover {
    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)
        moreover have (v, a) ∈ set (add-effects-of op)
          using calculation
          by blast
        moreover 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
        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(4)
          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) 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 Ψ
  shows list-all (is-valid-operator-strips (φ Ψ))
    (strips-problem.operators-of (φ Ψ))
proof −
  let ?Π = φ Ψ
  let ?vs = strips-problem.variables-of ?Π
```

```
{
 fix op
 assume op ∈ set (strips-problem.operators-of ?Π)
 — TODO slow.
 then obtain op'
   where op-is: op = φ_O Ψ op'
    and op'-in-operators: op' ∈ set ((Ψ)_{O+})
   unfolding SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
     sas-plus-problem-to-strips-problem-def
     sasp-op-to-strips-def
   by auto
 then have is-valid-op': is-valid-operator-sas-plus Ψ op'
   using sublocale-sas-plus-finite-domain-representation-ii(2)[OF assms]
   by blast
 moreover {
   fix v a
   assume (v, a) ∈ set (strips-operator.precondition-of op)
   — TODO slow.
   then have (v, a) ∈ set (sas-plus-operator.precondition-of op')
     using op-is
     unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
       sasp-op-to-strips-def
     by force
   moreover have v ∈ set ((Ψ)_{V+})
     using is-valid-op' calculation
     using is-valid-operator-sas-plus-then(1)
     by fastforce
   moreover have  a ∈ R_+ Ψ v
     using is-valid-op' calculation(1)
     using is-valid-operator-sas-plus-then(2)
     by fast
   ultimately have (v, a) ∈ set ?vs
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
     by force
 }
 moreover {
   fix v a
   assume (v, a) ∈ set (strips-operator.add-effects-of op)
   then have (v, a) ∈ set (effect-of op')
     using op-is
     unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
       sasp-op-to-strips-def
     by force
   then have v ∈ set ((Ψ)_{V+}) and a ∈ R_+ Ψ v
     using is-valid-operator-sas-plus-then is-valid-op'
     by fastforce+
   hence (v, a) ∈ set ?vs
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
     by force
```

```
    }
    moreover {
      fix v a'
       assume v-a'-in-delete-effects: (v, a') ∈ set (strips-operator.delete-effects-of
op)
      moreover have set (strips-operator.delete-effects-of op)
        = (⋃(v, a) ∈ set (effect-of op').
          { (v, a') | a'. a' ∈ (ℛ₊ Ψ v) ∧ a' ≠ 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) ∈ set (effect-of op')
          and a'-in: a' ∈ { a' ∈ ℛ₊ Ψ v. a' ≠ a }
        by blast
      moreover have is-valid-operator-sas-plus Ψ op'
        using op'-in-operators assms(1)
          is-valid-problem-sas-plus-then(2)
        by blast
      moreover have v ∈ set ((Ψ)_{𝒱₊})
        using is-valid-operator-sas-plus-then calculation(1, 3)
        by fast
      moreover have a' ∈ ℛ₊ Ψ v
        using a'-in
        by blast
      ultimately have (v, a') ∈ set ?vs
       using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
        by force
    }
    ultimately have set (strips-operator.precondition-of op) ⊆ set ?vs
      ∧ set (strips-operator.add-effects-of op) ⊆ set ?vs
      ∧ set (strips-operator.delete-effects-of op) ⊆ set ?vs
      ∧ (∀ v∈set (add-effects-of op). v ∉ set (delete-effects-of op))
      ∧ (∀ v∈set (delete-effects-of op). v ∉ set (add-effects-of op))
      using sasp-op-to-strips-effect-consistent[OF
          op-is op'-in-operators is-valid-op']
      by fast+
  }
  thus ?thesis
   unfolding is-valid-operator-strips-def STRIPS-Representation.is-valid-operator-strips-def

     list-all-iff ListMem-iff Let-def
    by blast
qed

lemma is-valid-problem-sas-plus-then-strips-transformation-too-iv:
  assumes is-valid-problem-sas-plus Ψ
  shows ∀ x. ((φ Ψ)_I) x ≠ None
```

```
    ⟷ ListMem x (strips-problem.variables-of (φ Ψ))
proof −
  let ?vs = variables-of Ψ
    and ?I = initial-of Ψ
    and ?Π = φ Ψ
  let ?vs' = strips-problem.variables-of ?Π
    and ?I' = strips-problem.initial-of ?Π
  {
    fix x
    have ?I' x ≠ None ⟷ ListMem x ?vs'
      proof (rule iffI)
        assume I'-of-x-is-not-None: ?I' x ≠ None
        then have x ∈ dom ?I'
          by blast
        moreover obtain v a where x-is: x = (v, a)
          by fastforce
        ultimately have (v, a) ∈ dom ?I'
          by blast
        then have v ∈ set ?vs
            and ?I v ≠ None
            and a ∈ R+ Ψ v
          using state-to-strips-state-dom-element-iff[OF assms(1), of v a  ?I]
          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 ?vs'
          unfolding ListMem-iff
       using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]

            x-is
          by auto
      next
        assume list-mem-x-vs': ListMem x ?vs'
        then obtain v a where x-is: x = (v, a)
          by fastforce
        then have (v, a) ∈ set ?vs'
          using list-mem-x-vs'
          unfolding ListMem-iff
          by blast
        then have v ∈ set ?vs and a ∈ R+ Ψ v
       using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
          by force+
        moreover have ?I v ≠ None
          using is-valid-problem-sas-plus-then(3) assms(1) calculation(1)
          by auto
        ultimately have (v, a) ∈ dom ?I'
          using state-to-strips-state-dom-element-iff[OF assms(1), of v a ?I]
```

```
      unfolding SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
        sas-plus-problem-to-strips-problem-def
        SAS-Plus-STRIPS.state-to-strips-state-def
        state-to-strips-state-def
      by force
    thus ?I' x ≠ None
      using x-is
      by fastforce
  qed
 }
 thus ?thesis
  by simp
qed

private lemma is-valid-problem-sas-plus-then-strips-transformation-too-v:
 assumes is-valid-problem-sas-plus Ψ
 shows ∀ x. ((φ Ψ)_G) x ≠ None
  ⟶ ListMem x (strips-problem.variables-of (φ Ψ))
proof −
 let ?vs = variables-of Ψ
  and ?D = range-of Ψ
  and ?G = goal-of Ψ
 let ?Π = φ Ψ
 let ?vs' = strips-problem.variables-of ?Π
  and ?G' = strips-problem.goal-of ?Π
 have nb: ?G' = φ_S Ψ ?G
  by simp
 {
  fix x
  assume ?G' x ≠ None
  moreover obtain v a where x = (v, a)
   by fastforce
  moreover have (v, a) ∈ dom ?G'
   using domIff calculation(1, 2)
   by blast
  moreover have v ∈ set ?vs and a ∈ R_+ Ψ v
   using state-to-strips-state-dom-is[OF assms(1), of ?G] nb calculation(3)
   by auto+
  ultimately have x ∈ set ?vs'
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
   by auto
 }
 thus ?thesis
  unfolding ListMem-iff
  by simp
qed
```

We now show that given Ψ is a valid SASPlus problem, then $\Pi \equiv \varphi\ \Psi$ is a valid STRIPS problem as well. The proof unfolds the definition of

is-valid-problem-strips and then shows each of the conjuncts for Π. These are:

- Π has at least one variable;
- Π has at least one operator;
- all operators are valid STRIPS operators;
- Π_I is defined for all variables in $\Pi_{\mathcal{V}}$; and finally,
- if $(\Pi_G)\ x$ is defined, then x is in $\Pi_{\mathcal{V}}$.

theorem
is-valid-problem-sas-plus-then-strips-transformation-too:
assumes *is-valid-problem-sas-plus* Ψ
shows *is-valid-problem-strips* (φ Ψ)
proof −
let *?*Π = φ Ψ
have *list-all* (*is-valid-operator-strips* (φ Ψ))
(*strips-problem.operators-of* (φ Ψ))
using *is-valid-problem-sas-plus-then-strips-transformation-too-iii*[*OF assms*].
moreover have ∀ *x*. (((φ Ψ)$_I$) *x* ≠ *None*) =
ListMem x (*strips-problem.variables-of* (φ Ψ))
using *is-valid-problem-sas-plus-then-strips-transformation-too-iv*[*OF assms*].
moreover have ∀ *x*. ((φ Ψ)$_G$) *x* ≠ *None* ⟶
ListMem x (*strips-problem.variables-of* (φ Ψ))
using *is-valid-problem-sas-plus-then-strips-transformation-too-v*[*OF assms*].
ultimately show *?thesis*
using *is-valid-problem-strips-def*
unfolding *STRIPS-Representation.is-valid-problem-strips-def*
by *fastforce*
qed

lemma *set-filter-all-possible-assignments-true-is*:
assumes *is-valid-problem-sas-plus* Ψ
shows *set* (*filter* (λ(*v*, *a*). *s* (*v*, *a*) = *Some True*)
(*all-possible-assignments-for* Ψ))
= (⋃ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$). *Pair v* ‘ { *a* ∈ $\mathcal{R}_+$ Ψ *v*. *s* (*v*, *a*) = *Some True* })
proof −
let *?vs* = *sas-plus-problem.variables-of* Ψ
and *?P* = (λ(*v*, *a*). *s* (*v*, *a*) = *Some True*)
let *?l* = *filter ?P* (*all-possible-assignments-for* Ψ)
have *set ?l* = *set* (*concat* (*map* (*filter ?P*) (*map* (*possible-assignments-for* Ψ)
?vs)))
unfolding *all-possible-assignments-for-def*
filter-concat[*of ?P map* (*possible-assignments-for* Ψ) (*sas-plus-problem.variables-of*
Ψ)]
by *simp*

```
  also have ... = set (concat (map (λv. filter ?P (possible-assignments-for Ψ v))
?vs))
    unfolding map-map comp-apply
    by blast
  also have ... = set (concat (map (λv. map (Pair v)
    (filter (?P ∘ 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 ... = ⋃(set ‘ ((λv. map (Pair v) (filter (λa. s (v, a) = Some True)
    (the (range-of Ψ v)))) ‘ set ?vs))
    unfolding set-concat set-map..
  also have ... = (⋃ v ∈ set ?vs. Pair v ‘ set (filter (λa. s (v, a) = Some True)
    (the (range-of Ψ v))))
    unfolding image-comp[of set] comp-apply set-map..
  also have ... = (⋃ v ∈ set ?vs. Pair v
    ‘ { a ∈ set (the (range-of Ψ v)). s (v, a) = Some True })
    unfolding set-filter..
  finally show ?thesis
    using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
    by auto
qed

lemma strips-state-to-state-dom-is:
  assumes is-valid-problem-sas-plus Ψ
  shows dom (φ_S⁻¹ Ψ s)
    = (⋃ v ∈ set ((Ψ)_{V+}).
      { v | a. a ∈ (R_+ Ψ v) ∧ s (v, a) = Some True })
proof −
  let ?vs = variables-of Ψ
    and ?s' = φ_S⁻¹ Ψ s
    and ?P = (λ(v, a). s (v, a) = Some True)
  let ?l = filter ?P (all-possible-assignments-for Ψ)
  {
    have fst ‘ set ?l = fst ‘ (⋃ v ∈ set ?vs. Pair v
      ‘ { 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
      ‘ { a ∈ R_+ Ψ v. s (v, a) = Some True })
      by blast
    also have ... = (⋃ v ∈ set ?vs. (λa. fst (Pair v a)) ‘
      { a ∈ R_+ Ψ v. s (v, a) = Some True })
      unfolding image-comp[of fst] comp-apply
      by blast
```

```
    finally have fst ‘ set ?l = (⋃ v ∈ set ((Ψ)_{V+}).
      { v | a. a ∈ (R_+ Ψ v) ∧ s (v, a) = 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 Ψ
    and v ∈ set ((Ψ)_{V+})
    and a ∈ R_+ Ψ v
    and (v, a) ∈ dom s'
    and ∀ (v, a) ∈ dom s'. ∀ (v, a') ∈ dom s'. s' (v, a) = Some True ∧ s' (v, a') =
Some True
      ⟶ (v, a) = (v, a')
  shows (φ_S^{-1} Ψ s') v = Some a ⟷ the (s' (v, a))
proof −
  let ?vs = variables-of Ψ
    and ?D = range-of Ψ
    and ?s = φ_S^{-1} Ψ s'
  let ?as = all-possible-assignments-for Ψ
  let ?l = filter (λ(v, a). s' (v, a) = Some True) ?as
  show ?thesis
    proof (rule iffI)
      assume s-of-v-is-Some-a: ?s v = Some a
      {
        have (v, a) ∈ 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) = Some True
          unfolding all-possible-assignments-for-set-is set-filter
          by blast
      }
      thus the (s' (v, a))
        by simp
    next
      assume the-of-s'-of-v-a-is: the (s' (v, a))
      then have s'-of-v-a-is-Some-true: s' (v, a) = Some True
        using assms(4) domIff
        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.  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-Some-true
        by simp
    }
    ultimately show ?s v = Some a
      using map-of-constant-assignments-defined-if[of ?l v a]
      unfolding SAS-Plus-STRIPS.strips-state-to-state-def
        strips-state-to-state-def
      by blast
  qed
qed
```

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

```
lemma strips-state-to-state-inverse-is-i:
assumes is-valid-problem-sas-plus 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 ?f = λ(v, a). the (s v) = a
    and ?l = concat (map (possible-assignments-for Ψ) (filter (λv. s v ≠ None)
?vs))
  have (v, a) ∈ dom ?s'
    using state-to-strips-state-dom-element-iff[
      OF assms(1)] assms(2, 3, 4)
    by presburger
  {
    have v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None }
```

```
    using assms(2, 3)
    by blast
  moreover have ∀ v ∈ set ((Ψ)_{V+}). v ∈ dom (sas-plus-problem.range-of Ψ)
    using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of[OF assms(1)].

  moreover have set ?l = (⋃ v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None }.
    { (v, a) |a. a ∈ R_+ Ψ v })
    unfolding state-to-strips-state-dom-is-i[OF calculation(2)]
    by blast
  ultimately have (v, a) ∈ set ?l
    using assms(4)
    by blast
 }
 moreover have set ?l ≠ {}
  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) ?f]
  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 Ψ
 and v ∈ set ((Ψ)_{V+})
 and s v = Some a
 and a ∈ R_+ Ψ v
 and a' ∈ R_+ Ψ v
 and a' ≠ a
shows (φ_S Ψ s) (v, a') = Some False
proof −
 have s v ≠ None
  using assms(3)
  by simp
 moreover have the (s v) ≠ a'
  using assms(3, 6)
  by simp
 ultimately show ?thesis
  using strips-state-to-state-inverse-is-i[OF assms(1, 2) - assms(5)]
  by force
qed
```

— NOTE Follows from the corollary above by contraposition.

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

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

    hence *set ?as* = *?As*
      unfolding *set-filter*
      using *all-possible-assignments-for-set-is*
      by *force*
  } **note** *nb* = *this*
  moreover **{**
    {
      fix $v\ v'\ a\ a'$

```
    assume (v, a) ∈ set ?as
      and (v', a') ∈ set ?as
    then have (v, a) ∈ ?As and (v', a') ∈ ?As
      using nb
      by blast+
    then have v-in-set-vs: v ∈ set ?vs and v'-in-set-vs: v' ∈ set ?vs
      and a-in-range-of-v: a ∈ ℛ₊ 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]
    by blast
}
— TODO slow.
thus ?thesis
  unfolding SAS-Plus-STRIPS.strips-state-to-state-def
    strips-state-to-state-def all-possible-assignments-for-def
```

by *simp*
qed

— Show that that $\varphi_S{}^{-1}\ \Psi$ is the inverse of $\varphi_S\ \Psi$. The additional constraints *dom s = set* $(\Psi_{\mathcal{V}^+})$ and $\forall\, v{\in}$*dom s. 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* Ψ
and *dom* $s \subseteq$ *set* $((\Psi)_{\mathcal{V}^+})$
and $\forall\, v \in$ *dom s. the* $(s\ v) \in \mathcal{R}_+\ \Psi\ v$
shows $s = (\varphi_S{}^{-1}\ \Psi\ (\varphi_S\ \Psi\ s))$
proof −
let *?vs = variables-of* Ψ
and *?D = range-of* Ψ
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*: $v \in$ *dom s*
then have *v-in-set-vs*: $v \in$ *set ?vs*
using *assms*(*2*)
by *auto*
then obtain *a*
where *the-s-v-is-a*: *s v = 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 = 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* $\in$ *dom ?s″*
then obtain *a*
where $a \in \mathcal{R}_+\ \Psi\ v$
and *?s′* (v, a) = *Some True*
using *strips-state-to-state-dom-is*[*OF assms*(*1*)]
by *blast*
then have $(v, a) \in$ *dom ?s′*
by *blast*

```
    then have s v ≠ None
      using state-to-strips-state-dom-is[OF assms(1)]
      by simp
    then obtain a where s v = Some a
      by blast
    hence ?s'' v = s v
      using nb
      by fastforce
  }
  — TODO slow.
  ultimately show ?thesis
    using map-le-antisym[of s ?s''] map-le-def
    unfolding strips-state-to-state-def
      state-to-strips-state-def
    by blast
qed

— An important lemma which shows that the submap relation does not change if
we transform the states on either side from SAS+ to STRIPS.
lemma state-to-strips-state-map-le-iff:
  assumes is-valid-problem-sas-plus Ψ
    and dom s ⊆ set ((Ψ)_{V+})
    and ∀ v ∈ dom s. the (s v) ∈ R_+ Ψ v
  shows s ⊆_m t ⟷ (φ_S Ψ s) ⊆_m (φ_S Ψ t)
proof −
  let ?vs = variables-of Ψ
    and ?D = range-of Ψ
    and ?s' = φ_S Ψ s
    and ?t' = φ_S Ψ t
  show ?thesis
    proof (rule iffI)
      assume s-map-le-t: s ⊆_m t
      {
        fix v a
        assume (v, a) ∈ dom ?s'
        moreover have v ∈ set ((Ψ)_{V+}) and s v ≠ None and a ∈ R_+ Ψ v
          using state-to-strips-state-dom-is[OF assms(1)] calculation
          by blast+
        moreover have ?s' (v, a) = Some (the (s v) = a)
          using state-to-strips-state-range-is[OF assms(1)] calculation(1)
          by meson
        moreover have v ∈ dom s
          using calculation(3)
          by auto
        moreover have s v = t v
          using s-map-le-t calculation(6)
          unfolding map-le-def
          by blast
        moreover have t v ≠ None
```

```
    using calculation(3, 7)
    by argo
  moreover have (v, a) ∈ dom ?t'
    using state-to-strips-state-dom-is[OF assms(1)] calculation(2, 4, 8)
    by blast
  moreover have ?t' (v, a) = Some (the (t v) = a)
    using state-to-strips-state-range-is[OF assms(1)] calculation(9)
    by simp
  ultimately have ?s' (v, a) = ?t' (v, a)
    by presburger
 }
 thus ?s' ⊆m ?t'
  unfolding map-le-def
  by fast
next
 assume s'-map-le-t': ?s' ⊆m ?t'
 {
  fix v
  assume v-in-dom-s: v ∈ dom s
  moreover obtain a where the-of-s-of-v-is-a: the (s v) = a
    by blast
  moreover have v-in-vs: v ∈ set ((Ψ)V+)
    and s-of-v-is-not-None: s v ≠ None
    and a-in-range-of-v: a ∈ R+ Ψ v
    using assms(2, 3) v-in-dom-s calculation
    by blast+
  moreover have (v, a) ∈ dom ?s'
    using state-to-strips-state-dom-is[OF assms(1)]
      calculation(3, 4, 5)
    by simp
  moreover have ?s' (v, a) = ?t' (v, a)
    using s'-map-le-t' calculation
    unfolding map-le-def
    by blast
  moreover have (v, a) ∈ dom ?t'
    using calculation
    unfolding domIff
    by argo
  moreover have ?s' (v, a) = Some (the (s v) = a)
    and ?t' (v, a) = Some (the (t v) = a)
    using state-to-strips-state-range-is[OF assms(1)] calculation
    by fast+
  moreover have s v = Some a
    using calculation(2, 4)
    by force
  moreover have ?s' (v, a) = Some True
    using calculation(9, 11)
    by fastforce
  moreover have ?t' (v, a) = Some True
```

```
      using calculation(7, 12)
      by argo
    moreover have the (t v) = a
      using calculation(10, 13) try0
      by force
    moreover {
      have v ∈ dom t
        using state-to-strips-state-dom-element-iff[OF assms(1)]
          calculation(8)
        by auto
      hence t v = Some a
        using calculation(14)
        by force
    }
    ultimately have s v = t v
      by argo
  }
  thus s ⊆_m t
    unfolding map-le-def
    by simp
 qed
qed
```

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

```
lemma sas-plus-operator-inverse-is:
  assumes is-valid-problem-sas-plus Ψ
    and op ∈ set ((Ψ)_{O+})
  shows (φ_O⁻¹ Ψ (φ_O Ψ op)) = op
proof −
  let ?op = φ_O⁻¹ Ψ (φ_O Ψ op)
  have precondition-of ?op = precondition-of op
    unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
      strips-op-to-sasp-def
      SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def
    by fastforce
  moreover have effect-of ?op = effect-of op
    unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
      strips-op-to-sasp-def
      SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def
    by force
  ultimately show ?thesis
    by simp
```

qed

— Note that we have to make the assumption that op' is a member of the operator set of the induced STRIPS problem $\varphi\ \Psi$. This implies that op' was transformed from an $op \in$ *operators-of* Ψ. If we don't make this assumption, then multiple STRIPS operators of the form $\langle\!|$ *precondition-of* = [], *add-effects-of* = [], *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* Ψ
and $op' \in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
shows $(\varphi_O\ \Psi\ ({\varphi_O}^{-1}\ \Psi\ op')) = op'$
proof −
let $?\Pi = \varphi\ \Psi$
obtain *op* **where** $op \in$ *set* $((\Psi)_{\mathcal{O}+})$ **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 *?thesis*
by *argo*
qed

lemma *sas-plus-equivalent-to-strips-i-a-I*:
assumes *is-valid-problem-sas-plus* Ψ
and *set ops'* $\subseteq$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *STRIPS-Semantics.are-all-operators-applicable* $(\varphi_S\ \Psi\ s)$ *ops'*
and $op \in$ *set* $[{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
shows *map-of* (*precondition-of op*) $\subseteq_m$ $({\varphi_S}^{-1}\ \Psi\ (\varphi_S\ \Psi\ s))$
proof −
let $?\Pi = \varphi\ \Psi$
and $?s' = \varphi_S\ \Psi\ s$
let $?s = {\varphi_S}^{-1}\ \Psi\ ?s'$
and $?D =$ *range-of* Ψ
and $?ops = [{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
and *?pre* = *precondition-of op*
have nb_1: $\forall (v, a) \in$ *dom* $?s'$.
$\forall (v, a') \in$ *dom* $?s'$.
$?s'\ (v, a) =$ *Some True* $\wedge$ $?s'\ (v, a') =$ *Some True*
$\longrightarrow (v, a) = (v, a')$
using *state-to-strips-state-effect-consistent*[*OF assms*(*1*)]
by *blast*
{
fix op'
assume $op' \in$ *set ops'*
moreover have $op' \in$ *set* $((?\Pi)_{\mathcal{O}})$
using *assms*(*2*) *calculation*

```
    by blast
  ultimately have ∃ op ∈ set ((Ψ)_{O+}). op' = (φ_O Ψ op)
    by auto
 } note nb_2 = this
 {
  fix op
  assume op ∈ set ?ops
  then obtain op' where op' ∈ set ops' and op = φ_O^{-1} Ψ op'
    using assms(4)
    by auto
  moreover obtain op'' where op'' ∈ set ((Ψ)_{O+}) and op' = φ_O Ψ op''
    using nb_2 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 nb_3 = 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 = φ_O^{-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 = (φ_O^{-1} Ψ op')
    ∧ (v, a) ∈ set (strips-operator.precondition-of op')
    by metis
 } note nb_4 = 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' = φ_O Ψ op
     using nb_2 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-Some-True calculation(6, 7)
      by blast
    moreover have (v, a) ∈ dom ?s'
      using s'-of-v-a-is-Some-True domIff
      by blast
    ultimately have ?s v = Some a
      using strips-state-to-state-range-is[OF assms(1) - - - nb_1]
        s'-of-v-a-is-Some-True
      by simp
  }
  hence ?s v = Some a.
} note nb_5 = 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) nb_3
      by blast
    then have is-valid-operator-sas-plus Ψ op
      using is-valid-problem-sas-plus-then(2) assms(1)
      unfolding is-valid-operator-sas-plus-def
      by auto
    hence ∀(v, a) ∈ set ?pre. ∀(v', a') ∈ set ?pre. v ≠ v' ∨ a = a'
      using is-valid-operator-sas-plus-then(5)
      unfolding is-valid-operator-sas-plus-def
```

```
      by fast
    }
    moreover have map-of ?pre v = Some a
      using map-of-constant-assignments-defined-if[of ?pre] calculation(2, 3)
      by blast
    moreover obtain op' where op' ∈ set ops'
      and (v, a) ∈ set (strips-operator.precondition-of op')
      using nb_4[OF assms(4) calculation(2)]
      by blast
    moreover have ?s v = Some a
      using nb_5 calculation(5, 6)
      by fast
    ultimately have map-of ?pre v = ?s v
      by argo
  }
  thus ?thesis
    unfolding map-le-def
    by blast
qed

lemma to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure:
  assumes is-valid-problem-sas-plus Ψ
    and set ops' ⊆ set ((φ Ψ)_𝒪)
    and op ∈ set [φ_O⁻¹ Ψ op'. op' ← ops']
  shows op ∈ set ((Ψ)_𝒪+) ∧ (∃ op' ∈ set ops'. op' = φ_O Ψ op)
proof −
  let ?Π = φ Ψ
  obtain op' where op' ∈ set ops' and op = φ_O⁻¹ Ψ op'
    using assms(3)
    by auto
  moreover have op' ∈ set ((?Π)_𝒪)
    using assms(2) calculation(1)
    by blast
  moreover obtain op'' where op'' ∈ set ((Ψ)_𝒪+) and op' = φ_O Ψ op''
    using calculation(3)
    by auto
  moreover have op = op''
    using sas-plus-operator-inverse-is[OF assms(1) calculation(4)] calculation(2,
5)
    by presburger
  ultimately show ?thesis
    by blast
qed


lemma sas-plus-equivalent-to-strips-i-a-II:
  fixes Ψ :: ('variable, 'domain) sas-plus-problem
  fixes s :: ('variable, 'domain) state
  assumes is-valid-problem-sas-plus Ψ
```

```
    and set ops' ⊆ set ((φ Ψ)_𝒪)
    and STRIPS-Semantics.are-all-operators-applicable (φ_s Ψ s) ops'
      ∧ STRIPS-Semantics.are-all-operator-effects-consistent ops'
  shows are-all-operator-effects-consistent [φ_O⁻¹ Ψ op'. op' ← ops']
proof −
  let ?s' = φ_S Ψ s
  let ?s = φ_S⁻¹ Ψ ?s'
    and ?ops = [φ_O⁻¹ Ψ op'. op' ← ops']
    and ?Π = φ Ψ
  have nb: ∀(v, a) ∈ dom ?s'.
    ∀(v, a') ∈ dom ?s'.
      ?s' (v, a) = Some True ∧ ?s' (v, a') = Some True
      ⟶ (v, a) = (v, a')
    using state-to-strips-state-effect-consistent[OF assms(1)]
    by blast
  {
    fix op₁' op₂'
    assume op₁' ∈ set ops' and op₂' ∈ set ops'
    hence STRIPS-Semantics.are-operator-effects-consistent op₁' op₂'
      using assms(3)
     unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
      by blast
  } note nb₁ = this
  {
    fix op₁ op₁' op₂ op₂'
    assume op₁-in-ops: op₁ ∈ set ?ops
      and op₁'-in-ops': op₁' ∈ set ops'
      and op₁'-is: op₁' = φ_O Ψ op₁
      and is-valid-op₁: is-valid-operator-sas-plus Ψ op₁
      and op₂-in-ops: op₂ ∈ set ?ops
      and op₂'-in-ops': op₂' ∈ set ops'
      and op₂'-is: op₂' = φ_O Ψ op₂
      and is-valid-op₂: is-valid-operator-sas-plus Ψ op₂
    have ∀(v, a) ∈ set (add-effects-of op₁'). ∀(v', a') ∈ set (add-effects-of op₂').
        v ≠ v' ∨ a = a'
      proof (rule ccontr)
        assume ¬(∀(v, a) ∈ set (add-effects-of op₁'). ∀(v', a') ∈ set (add-effects-of
op₂').
        v ≠ v' ∨ a = a')
        then obtain v v' a a' where (v, a) ∈ set (add-effects-of op₁')
          and (v', a') ∈ set (add-effects-of op₂')
          and v = v'
          and a ≠ a'
          by blast
        — TODO slow.
        moreover have (v, a) ∈ set (effect-of op₁)
          using op₁'-is op₂'-is calculation(1, 2)
          unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
            sasp-op-to-strips-def
```

```
      by force
    moreover {
      have (v', a') ∈ set (effect-of op₂)
        using op₂'-is calculation(2)
        unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
          sasp-op-to-strips-def
        by force
      hence a' ∈ ℛ₊ Ψ v
        using is-valid-operator-sas-plus-then is-valid-op₂ calculation(3)
        by fastforce
    }
    moreover have (v, a') ∈ set (delete-effects-of op₁')
      using sasp-op-to-strips-set-delete-effects-is
        op₁'-is is-valid-op₁ calculation(3, 4, 5, 6)
      by blast
      moreover have ¬STRIPS-Semantics.are-operator-effects-consistent op₁'
op₂'
     unfolding STRIPS-Semantics.are-operator-effects-consistent-def list-ex-iff

      using calculation(2, 3, 7)
      by meson
    ultimately show False
      using assms(3) op₁'-in-ops' op₂'-in-ops'
   unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
      by blast
  qed
 } note nb₃ = this
 {
  fix op₁ op₂
  assume op₁-in-ops: op₁ ∈ set ?ops and op₂-in-ops: op₂ ∈ set ?ops
  moreover have op₁-in-operators-of-Ψ: op₁ ∈ set ((Ψ)_{𝒪+})
    and op₂-in-operators-of-Ψ: op₂ ∈ set ((Ψ)_{𝒪+})
   using to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure[OF
       assms(1, 2)] calculation
    by blast+
  moreover have is-valid-operator-op₁: is-valid-operator-sas-plus Ψ op₁
    and is-valid-operator-op₂: is-valid-operator-sas-plus Ψ op₂
   using is-valid-problem-sas-plus-then(2) op₁-in-operators-of-Ψ op₂-in-operators-of-Ψ
     assms(1)
    unfolding is-valid-operator-sas-plus-def
    by auto+
  moreover obtain op₁' op₂'
    where op₁-in-ops': op₁' ∈ set ops'
      and op₁-is: op₁' = φ_O Ψ op₁
      and op₂-in-ops': op₂' ∈ set ops'
      and op₂-is: op₂' = φ_O Ψ op₂
   using to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure[OF
       assms(1, 2)] op₁-in-ops op₂-in-ops
    by blast
```

— TODO slow.
ultimately have $\forall (v, a) \in set\ (add\text{-}effects\text{-}of\ op_1{}').\ \forall (v', a') \in set\ (add\text{-}effects\text{-}of\ op_2{}').$
$v \neq v' \vee a = a'$
using nb_3
by *auto*
hence *are-operator-effects-consistent* $op_1\ op_2$
using op_1*-is* op_2*-is*
unfolding *are-operator-effects-consistent-def*
sasp-op-to-strips-def
SAS-Plus-STRIPS.sasp-op-to-strips-def
list-all-iff Let-def
by *simp*
}
thus *?thesis*
unfolding *are-all-operator-effects-consistent-def list-all-iff*
by *fast*
qed

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

lemma *sas-plus-equivalent-to-strips-i-a-IV*:
assumes *is-valid-problem-sas-plus* Ψ
and *set ops'* $\subseteq$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *STRIPS-Semantics.are-all-operators-applicable* $(\varphi_S\ \Psi\ s)$ *ops'*
$\wedge$ *STRIPS-Semantics.are-all-operator-effects-consistent ops'*
shows *are-all-operators-applicable-in* $({\varphi_S}^{-1}\ \Psi\ (\varphi_S\ \Psi\ s))\ [{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']\ \wedge$
are-all-operator-effects-consistent $[{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
proof −
let *?*$\Pi = \varphi\ \Psi$
and *?s'* $= \varphi_S\ \Psi\ s$
let *?vs'* $=$ *strips-problem.variables-of ?*Π
and *?ops'* $=$ *strips-problem.operators-of ?*Π
and *?vs* $=$ *variables-of* Ψ
and *?D* $=$ *range-of* Ψ
and *?s* $= {\varphi_S}^{-1}\ \Psi$ *?s'*
and *?ops* $= [{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
have *nb*: $\forall (v, a) \in dom\ ?s'.$
$\forall (v, a') \in dom\ (\varphi_S\ \Psi\ s).$
?s' $(v, a) =$ *Some True* $\wedge$ *?s'* $(v, a') =$ *Some True*
$\longrightarrow (v, a) = (v, a')$
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 (λop. map-of (precondition-of op) ⊆_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 (λop. 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
list-all-iff
    by blast
  ultimately have are-all-operators-applicable-in ?s ?ops
  unfolding are-all-operators-applicable-in-def is-operator-applicable-in-def list-all-iff
    by argo
 }
 moreover have are-all-operator-effects-consistent ?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 Ψ
  and dom s ⊆ set ((Ψ)_{V+})
  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_O^{-1} Ψ op'. op' ← ops'] ∧
    are-all-operator-effects-consistent [φ_O^{-1} Ψ op'. op' ← ops']
 shows STRIPS-Semantics.are-all-operators-applicable (φ_S Ψ s) ops'
proof −
 let ?vs = variables-of Ψ
  and ?D = range-of Ψ
  and ?Π = φ Ψ
  and ?ops = [φ_O^{-1} Ψ op'. op' ← ops']
  and ?s' = φ_S Ψ s
 — TODO refactor.
 {
  fix op'
  assume op' ∈ set ops'
  moreover obtain op where op ∈ set ?ops and op = φ_O^{-1} Ψ op'
    using calculation
    by force
  moreover obtain op'' where op'' ∈ set ((Ψ)_{O+}) and op' = φ_O Ψ op''
    using assms(4) calculation(1)
    by auto
  moreover have is-valid-operator-sas-plus Ψ op''
```

using *is-valid-problem-sas-plus-then*(*2*) *assms*(*1*) *calculation*(*4*)
unfolding *is-valid-operator-sas-plus-def*
by *auto*
moreover have $op = op''$
using *sas-plus-operator-inverse-is*[*OF assms*(*1*)] *calculation*(*3*, *4*, *5*)
by *blast*
ultimately have $\exists\, op \in set\ ?ops.\ op \in set\ ?ops \wedge op = ({\varphi_O}^{-1}\ \Psi\ op') \wedge is\text{-}valid\text{-}operator\text{-}sas\text{-}plus\ \Psi\ op$
by *blast*
} note $nb_1 = this$
have nb_2: $\forall (v, a) \in dom\ ?s'.\ \forall (v, a') \in dom\ ?s'.\ ?s'\ (v, a) = Some\ True \wedge ?s'\ (v, a') = Some\ True \longrightarrow (v, a) = (v, a')$
using *state-to-strips-state-effect-consistent*[*OF assms*(*1*), *of - - s*]
by *blast*
{
fix *op*
assume $op \in set\ ?ops$
hence $map\text{-}of\ (precondition\text{-}of\ op) \subseteq_m s$
using *assms*(*5*)
unfolding *are-all-operators-applicable-in-def is-operator-applicable-in-def list-all-iff*
by *blast*
} note $nb_3 = this$
{
fix op'
assume $op' \in set\ ops'$
then obtain *op* **where** *op-in-ops*: $op \in set\ ?ops$
and *op-is*: $op = ({\varphi_O}^{-1}\ \Psi\ op')$
and *is-valid-operator-op*: $is\text{-}valid\text{-}operator\text{-}sas\text{-}plus\ \Psi\ op$
using nb_1
by *force*
moreover have *preconditions-are-consistent*:
$\forall (v, a) \in set\ (precondition\text{-}of\ op).\ \forall (v', a') \in set\ (precondition\text{-}of\ op).\ v \neq v' \vee a = a'$
using *is-valid-operator-sas-plus-then*(*5*) *calculation*(*3*)
unfolding *is-valid-operator-sas-plus-def*
by *fast*
moreover {
fix *v a*
assume $(v, a) \in set\ (strips\text{-}operator.precondition\text{-}of\ op')$
moreover have *v-a-in-precondition-of-op*: $(v, a) \in set\ (precondition\text{-}of\ op)$
using *op-is calculation*
unfolding *SAS-Plus-STRIPS.strips-op-to-sasp-def strips-op-to-sasp-def*
by *auto*
moreover have $map\text{-}of\ (precondition\text{-}of\ op)\ v = Some\ a$
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 nb_3[OF op-in-ops] calculation(3)
      unfolding map-le-def
      by force
    moreover have v ∈ set ((Ψ)_{V+}) 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_S^{-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)] nb_2
      by auto
  }
  ultimately have ∀(v, a) ∈ set (strips-operator.precondition-of op'). ?s' (v, a)
= Some True
    by fast
 }
 thus ?thesis
  unfolding are-all-operators-applicable-def is-operator-applicable-in-def
    STRIPS-Representation.is-operator-applicable-in-def list-all-iff
  by simp
qed


lemma sas-plus-equivalent-to-strips-i-a-VII:
 assumes is-valid-problem-sas-plus Ψ
  and dom s ⊆ set ((Ψ)_{V+})
  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_O^{-1} Ψ op'. op' ← ops'] ∧
  are-all-operator-effects-consistent [φ_O^{-1} Ψ op'. op' ← ops']
 shows STRIPS-Semantics.are-all-operator-effects-consistent ops'
proof −
 let ?s' = φ_S Ψ s
  and ?ops = [φ_O^{-1} Ψ op'. op' ← ops']
  and ?D = range-of Ψ
  and ?Π = φ Ψ
```

— TODO refactor.
{
fix op'
assume $op' \in set\ ops'$
moreover obtain op **where** $op \in set\ ?ops$ **and** $op = {\varphi_O}^{-1}\ \Psi\ op'$
using *calculation*
by *force*
moreover obtain op'' **where** $op'' \in set\ ((\Psi)_{\mathcal{O}+})$ **and** $op' = \varphi_O\ \Psi\ op''$
using *assms*(*4*) *calculation*(*1*)
by *auto*
moreover have *is-valid-operator-sas-plus* $\Psi\ op''$
using *is-valid-problem-sas-plus-then*(*2*) *assms*(*1*) *calculation*(*4*)
unfolding *is-valid-operator-sas-plus-def*
by *auto*
moreover have $op = op''$
using *sas-plus-operator-inverse-is*[*OF assms*(*1*)] *calculation*(*3*, *4*, *5*)
by *blast*
ultimately have $\exists\, op \in set\ ?ops.\ op \in set\ ?ops \wedge op' = (\varphi_O\ \Psi\ op)$
$\wedge$ *is-valid-operator-sas-plus* $\Psi\ op$
by *blast*
} **note** $nb_1 = this$
{
fix ${op_1}'\ {op_2}'$
assume ${op_1}' \in set\ ops'$
and ${op_2}' \in set\ ops'$
and $\exists\,(v, a) \in set\ (\textit{add-effects-of}\ {op_1}').\ \exists\,(v', a') \in set\ (\textit{delete-effects-of}\ {op_2}').$
$(v, a) = (v', a')$
moreover obtain $op_1\ op_2$
where $op_1 \in set\ ?ops$
and ${op_1}' = \varphi_O\ \Psi\ op_1$
and *is-valid-operator-sas-plus* $\Psi\ op_1$
and $op_2 \in set\ ?ops$
and ${op_2}' = \varphi_O\ \Psi\ op_2$
and *is-valid-op*$_2$: *is-valid-operator-sas-plus* $\Psi\ op_2$
using nb_1 *calculation*(*1*, *2*)
by *meson*
moreover obtain $v\ v'\ a\ a'$
where $(v, a) \in set\ (\textit{add-effects-of}\ {op_1}')$
and $(v', a') \in set\ (\textit{delete-effects-of}\ {op_2}')$
and $(v, a) = (v', a')$
using *calculation*
by *blast*
moreover have $(v, a) \in set\ (\textit{effect-of}\ op_1)$
using *calculation*(*5*, *10*)
unfolding *SAS-Plus-STRIPS.sasp-op-to-strips-def*
sasp-op-to-strips-def
by *fastforce*
moreover have $v = v'$ **and** $a = a'$
using *calculation*(*12*)

by *simp+*

— The next proof block shows that $(v',\ a')$ is constructed from an effect $(v'',\ a'')$ s.t. $a' \neq a''$.

moreover {

have $(v',\ a') \in (\bigcup (v'',\ a'') \in set\ (effect\text{-}of\ op_2).$
$\{\ (v'',\ a''') \mid a'''.\ a''' \in (\mathcal{R}_+\ \Psi\ v'') \wedge\ a''' \neq a''\ \})$
using *sasp-op-to-strips-set-delete-effects-is*
calculation(*8*, *11*) *is-valid-op*$_2$
by *blast*
then obtain $v''\ a''$ **where** $(v'',\ a'') \in set\ (effect\text{-}of\ op_2)$
and $(v',\ a') \in \{\ (v'',\ a''') \mid a'''.\ a''' \in (\mathcal{R}_+\ \Psi\ v'') \wedge\ a''' \neq a''\ \}$
by *blast*
moreover have $(v',\ a'') \in set\ (effect\text{-}of\ op_2)$
using *calculation*
by *blast*
moreover have $a' \in \mathcal{R}_+\ \Psi\ v''$ **and** $a' \neq a''$
using *calculation*(*1*, *2*)
by *fast+*
ultimately have $\exists\, a''.\ (v',\ a'') \in set\ (effect\text{-}of\ op_2) \wedge a' \in (\mathcal{R}_+\ \Psi\ v')$
$\wedge\ a' \neq a''$
by *blast*
}
moreover obtain a'' **where** $(v',\ a'') \in set\ (effect\text{-}of\ op_2)$
and $a' \in \mathcal{R}_+\ \Psi\ v'$
and $a' \neq a''$
using *calculation*(*16*)
by *blast*
moreover have $\exists\, (v,\ a) \in set\ (effect\text{-}of\ op_1).\ (\exists\, (v',\ a') \in set\ (effect\text{-}of\ op_2).$
$v = v' \wedge a \neq a')$
using *calculation*(*13*, *14*, *15*, *17*, *19*)
by *blast*
moreover have $\neg$*are-operator-effects-consistent op*$_1$ *op*$_2$
unfolding *are-operator-effects-consistent-def list-all-iff*
using *calculation*(*20*)
by *fastforce*
ultimately have $\neg$*are-all-operator-effects-consistent ?ops*
unfolding *are-all-operator-effects-consistent-def list-all-iff*
by *meson*
} note nb_2 = *this*
{
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*$_2'$
proof (*rule ccontr*)
assume $\neg$*STRIPS-Semantics.are-operator-effects-consistent op*$_1'$ *op*$_2'$
then consider (*A*) $\exists\, (v,\ a) \in set\ (add\text{-}effects\text{-}of\ op_1').$
$\exists\, (v',\ a') \in set\ (delete\text{-}effects\text{-}of\ op_2').\ (v,\ a) = (v',\ a')$
| (*B*) $\exists\, (v,\ a) \in set\ (add\text{-}effects\text{-}of\ op_2').$

```
        ∃(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)_{V+})
    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_O^-1 Ψ op'. op' ← ops'] ∧
    are-all-operator-effects-consistent [φ_O^-1 Ψ op'. op' ← ops']
  shows STRIPS-Semantics.are-all-operators-applicable (φ_S 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.
      {
        have ∀ (v, a) ∈ set (effect-of op). v ∈ V
          using Cons.prems(2)
          by fastforce
        moreover have fst ‘ set (effect-of op) ⊆ V
```

```
        using calculation
        by fastforce
      ultimately have dom ?s' ⊆ V
        unfolding dom-map-add dom-map-of-conv-image-fst
        using Cons.prems(1)
        by blast
    }
    moreover have ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op). v ∈ V
      using Cons.prems(2)
      by fastforce
    ultimately have dom (execute-parallel-operator-sas-plus ?s' ops) ⊆ V
      using Cons.IH[of ?s']
      by fast
    thus ?case
      unfolding execute-parallel-operator-sas-plus-cons.
  qed
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.

```
lemma sas-plus-equivalent-to-strips-i-a-X:
  assumes dom s ⊆ V
    and V ⊆ dom D
    and ∀ v ∈ dom s. the (s v) ∈ set (the (D v))
    and ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op). v ∈ V ∧ a ∈ set (the (D v))
  shows ∀ v ∈ dom (execute-parallel-operator-sas-plus s ops).
    the (execute-parallel-operator-sas-plus s ops v) ∈ set (the (D 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)
      {
        {
          have ∀ (v, a) ∈ set (effect-of op). v ∈ V
            using Cons.prems(4)
            by fastforce
          moreover have fst ‘ set (effect-of op) ⊆ V
            using calculation
            by fastforce
```

```
      ultimately have dom ?s' ⊆ V
        unfolding dom-map-add dom-map-of-conv-image-fst
        using Cons.prems(1)
        by blast
    }
    moreover {
      fix v
      assume v-in-dom-s': v ∈ dom ?s'
      hence the (?s' v) ∈ set (the (D v))
        proof (cases v ∈ dom (map-of (effect-of op)))
          case True
          moreover have ?s' v = (map-of (effect-of op)) v
            unfolding map-add-dom-app-simps(1)[OF True]
            by blast
          moreover obtain a where (map-of (effect-of op)) v = Some a
            using calculation(1)
            by fast
          moreover have (v, a) ∈ set (effect-of op)
            using map-of-SomeD calculation(3)
            by fast
          moreover have a ∈ set (the (D v))
            using Cons.prems(4) calculation(4)
            by fastforce
          ultimately show ?thesis
            by force
        next
          case False
          then show ?thesis
            unfolding map-add-dom-app-simps(3)[OF False]
            using Cons.prems(3) v-in-dom-s'
            by fast
        qed
    }
    moreover have ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op). v ∈ V ∧ a ∈
set (the (D v))
      using Cons.prems(4)
      by auto
    ultimately have ∀ v ∈ dom (execute-parallel-operator-sas-plus ?s' ops).
      the (execute-parallel-operator-sas-plus ?s' ops v) ∈ set (the (D v))
      using Cons.IH[of s ++ map-of (effect-of op), OF - Cons.prems(2)]
      by meson
  }
  thus ?case
    unfolding execute-parallel-operator-sas-plus-cons
    by blast
 qed
qed

lemma transfom-sas-plus-problem-to-strips-problem-operators-valid:
```

```
  assumes is-valid-problem-sas-plus Ψ
    and op' ∈ set ((φ Ψ)_𝒪)
  obtains op
  where op ∈ set ((Ψ)_𝒪+)
    and op' = (φ_O Ψ op) is-valid-operator-sas-plus Ψ op
proof −
  {
    obtain op where op ∈ set ((Ψ)_𝒪+) and op' = φ_O Ψ op
      using assms
      by auto
    moreover have is-valid-operator-sas-plus Ψ op
      using is-valid-problem-sas-plus-then(2) assms(1) calculation(1)
      by auto
    ultimately have ∃ op ∈ set ((Ψ)_𝒪+). op' = (φ_O Ψ op)
      ∧ is-valid-operator-sas-plus Ψ 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 Ψ
    and op' ∈ set ((φ Ψ)_𝒪)
  shows (φ_S Ψ s) ++ map-of (effect-to-assignments op')
    = φ_S Ψ (s ++ map-of (effect-of (φ_O⁻¹ Ψ op')))
proof −
  let ?Π = φ Ψ
  let ?vs = variables-of Ψ
    and ?ops = operators-of Ψ
    and ?ops' = strips-problem.operators-of ?Π
  let ?s' = φ_S Ψ s
  let ?t = ?s' ++ map-of (effect-to-assignments op')
    and ?t' = φ_S Ψ (s ++ map-of (effect-of (φ_O⁻¹ Ψ op')))
  obtain op where op'-is: op' = (φ_O Ψ op)
    and op-in-ops: op ∈ set ((Ψ)_𝒪+)
    and is-valid-operator-op: is-valid-operator-sas-plus Ψ op
    using transfom-sas-plus-problem-to-strips-problem-operators-valid[OF assms]
    by auto
  have nb_1: (φ_O⁻¹ Ψ 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 $\ldots$ = *set* (*effect-of op*) $\cup$ *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*)
$\cup$ ($\bigcup$(*v*, *a*) $\in$ *set* (*effect-of op*). { (*v*, *a'*) | *a'*. *a'* $\in$ ($\mathcal{R}_+$ Ψ *v*) $\wedge$ *a'* $\neq$ *a* })
using *sasp-op-to-strips-set-delete-effects-is*[*OF*
is-valid-operator-op] *op'-is*
by *argo*
} **note** nb_2 = *this*
have nb_3: *dom ?t* = *dom ?s'* $\cup$ *set* (*effect-of op*)
$\cup$ ($\bigcup$(*v*, *a*) $\in$ *set* (*effect-of op*). { (*v*, *a'*) | *a'*. *a'* $\in$ ($\mathcal{R}_+$ Ψ *v*) $\wedge$ *a'* $\neq$ *a* })
unfolding nb_2 *dom-map-add*
by *blast*
— TODO refactor.
have nb_4: *dom* (*s* ++ *map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*)))
= *dom s* $\cup$ *fst* ' *set* (*effect-of op*)
unfolding *dom-map-add dom-map-of-conv-image-fst* nb_1
by *fast*
{
let *?u* = *s* ++ *map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))
have *dom ?t'* = ($\bigcup$ *v* $\in$ { *v* | *v*. *v* $\in$ *set* ((Ψ)$_{\mathcal{V}+}$) $\wedge$ *?u v* $\neq$ *None* }.
{ (*v*, *a*) | *a*. *a* $\in$ $\mathcal{R}_+$ Ψ *v* })
using *state-to-strips-state-dom-is*[*OF assms*(*1*)]
by *presburger*
} **note** nb_5 = *this*
— TODO refactor.
have nb_6: *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 nb_7: *set* (*delete-effects-of op'*) = ($\bigcup$(*v*, *a*) $\in$ *set* (*effect-of op*).
{ (*v*, *a'*) | *a'*. *a'* $\in$ ($\mathcal{R}_+$ Ψ *v*) $\wedge$ *a'* $\neq$ *a* })
using *sasp-op-to-strips-set-delete-effects-is*[*OF*
is-valid-operator-op] *op'-is*
by *argo*
— TODO refactor.
{
let *?Add* = *set* (*effect-of op*)
let *?Delete* = ($\bigcup$(*v*, *a*) $\in$ *set* (*effect-of op*).
{ (*v*, *a'*) | *a'*. *a'* $\in$ ($\mathcal{R}_+$ Ψ *v*) $\wedge$ *a'* $\neq$ *a* })
have *dom-add*: *dom* (*map-of* (*map* (λ*v*. (*v*, *True*)) (*add-effects-of op'*))) = *?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 (map (λv. (v, False)) (delete-effects-of op'))) =
?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) ∈ dom (map-of (map (λv. (v, True))
(add-effects-of op')))
      have (v, a) ∉ dom (map-of (map (λv. (v, False)) (delete-effects-of op')))
        proof (rule ccontr)
          assume ¬((v, a) ∉ dom (map-of (map (λv. (v, False)) (delete-effects-of
op'))))
          then have (v, a) ∈ ?Delete and (v, a) ∈ ?Add
            using dom-add dom-delete v-a-in-dom-add
            by argo+
          moreover have ∀(v', a') ∈ ?Add. v ≠ v' ∨ 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 disjnt (dom (map-of (map (λv. (v, True)) (add-effects-of op'))))
      (dom (map-of (map (λv. (v, False)) (delete-effects-of op'))))
      unfolding disjnt-def Int-def
      using nb_7
      by simp
  }
  hence dom (map-of (map (λv. (v, True)) (add-effects-of op'))) = ?Add
    and dom (map-of (map (λv. (v, False)) (delete-effects-of op'))) = ?Delete
    and disjnt (dom (map-of (map (λv. (v, True)) (add-effects-of op'))))
      (dom (map-of (map (λv. (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) ∈ set (effect-of op).
    { (v, a') | a'. a' ∈ (R_+ Ψ v) ∧ a' ≠ a })
  — TODO slow.
```

```
  have ∀(v, a) ∈ ?Add. map-of (effect-to-assignments op') (v, a) = Some True
    and ∀(v, a) ∈ ?Delete. map-of (effect-to-assignments op') (v, a) = Some
False
   proof −
    {
     fix v a
     assume (v, a) ∈ ?Add
     hence map-of (effect-to-assignments op') (v, a) = Some True
      unfolding effect-to-assignments-simp
     using nb6 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) ∈ ?Delete
    moreover have (v, a) ∈ dom (map-of (map (λv. (v, False)) (delete-effects-of
op')))
      using nb8(2) calculation(1)
      by argo
    moreover have (v, a) ∉ dom (map-of (map (λv. (v, True)) (add-effects-of
op')))
      using nb8
      unfolding disjnt-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']
        nb7
      by auto
    }
    ultimately show ∀(v, a) ∈ ?Add. map-of (effect-to-assignments op') (v,
a) = Some True
     and ∀(v, a) ∈ ?Delete. map-of (effect-to-assignments op') (v, a) = Some
False
     by blast+
   qed
 } note nb9 = this
 {
  fix v a
  assume (v, a) ∈ set (effect-of op)
```

moreover have $\forall (v,\ a) \in set\ (effect\text{-}of\ op).\ \forall (v',\ a') \in set\ (effect\text{-}of\ op).\ v \neq v' \vee a = a'$
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*
} **note** nb_{10} = *this*
{
fix v a
assume *v-a-in-effect-of-op*: $(v,\ a) \in set\ (effect\text{-}of\ op)$
and $(s\ ++\ map\text{-}of\ (effect\text{-}of\ ({\varphi_O}^{-1}\ \Psi\ op')))\ v \neq None$
moreover have $v \in set\ ?vs$
using *is-valid-operator-op is-valid-operator-sas-plus-then*(*3*) *calculation*(*1*)
by *fastforce*
moreover {
have *is-valid-problem-strips* $?\Pi$
using *is-valid-problem-sas-plus-then-strips-transformation-too*
assms(*1*)
by *blast*
thm *calculation*(*1*) nb_6 *assms*(*2*)
moreover have $set\ (add\text{-}effects\text{-}of\ op') \subseteq set\ ((?\Pi)_{\mathcal{V}})$
using *assms*(*2*) *is-valid-problem-strips-operator-variable-sets*(*2*)
calculation
by *blast*
moreover have $(v,\ a) \in set\ ((?\Pi)_{\mathcal{V}})$
using *v-a-in-effect-of-op* nb_6 *calculation*(*2*)
by *blast*
ultimately have $a \in \mathcal{R}_+\ \Psi\ v$
using *sas-plus-problem-to-strips-problem-variable-set-element-iff*[*OF*
assms(*1*)]
by *fast*
}
— TODO slow.
ultimately have $(v,\ a) \in dom\ (\varphi_S\ \Psi\ (s\ ++\ map\text{-}of\ (effect\text{-}of\ ({\varphi_O}^{-1}\ \Psi\ op'))))$
using *state-to-strips-state-dom-is*[*OF assms*(*1*), *of*
$s\ ++\ map\text{-}of\ (effect\text{-}of\ ({\varphi_O}^{-1}\ \Psi\ op'))$]
by *simp*
} **note** nb_{11} = *this*
{
fix v a
assume $(v,\ a) \in set\ (effect\text{-}of\ op)$
moreover have $v \in dom\ (map\text{-}of\ (effect\text{-}of\ op))$
unfolding *dom-map-of-conv-image-fst*
using *calculation*
by *force*
moreover have $(s\ ++\ map\text{-}of\ (effect\text{-}of\ ({\varphi_O}^{-1}\ \Psi\ op')))\ v$ = *Some a*

unfolding *map-add-dom-app-simps(1)*[*OF calculation(2)*] nb_1
using nb_{10} *calculation(1)*
by *blast*
moreover have (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))) *v* ≠ *None*
using *calculation(3)*
by *auto*
moreover have (*v*, *a*) ∈ *dom* (φ*S* Ψ (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))))
using nb_{11} *calculation(1*, *4)*
by *presburger*
ultimately have (φ*S* Ψ (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*)))) (*v*, *a*) = *Some True*
using *state-to-strips-state-range-is*[*OF assms(1)*]
by *simp*
} **note** nb_{12} = *this*
{
fix *v a'*
assume (*v*, *a'*) ∈ *dom* (*map-of* (*effect-to-assignments op'*))
and (*v*, *a'*) ∈ (⋃(*v*, *a*) ∈ *set* (*effect-of op*).
{ (*v*, *a'*) | *a'*. *a'* ∈ ($\mathcal{R}_+$ Ψ *v*) ∧ *a'* ≠ *a* })
moreover have *v* ∈ *dom* (*map-of* (*effect-of op*))
unfolding *dom-map-of-conv-image-fst*
using *calculation(2)*
by *force*
moreover have *v* ∈ *set ?vs*
using *calculation(3) is-valid-operator-sas-plus-then(3) is-valid-operator-op*
unfolding *dom-map-of-conv-image-fst is-valid-operator-sas-plus-def*
by *fastforce*
moreover obtain *a* **where** (*v*, *a*) ∈ *set* (*effect-of op*)
and *a'* ∈ $\mathcal{R}_+$ Ψ *v*
and *a'* ≠ *a*
using *calculation(2)*
by *blast*
moreover have (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))) *v* = *Some a*
unfolding *map-add-dom-app-simps(1)*[*OF calculation(3)*] nb_1
using nb_{10} *calculation(5)*
by *blast*
moreover have (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))) *v* ≠ *None*
using *calculation(8)*
by *auto*
— TODO slow.
moreover have (*v*, *a'*) ∈ *dom* (φ*S* Ψ (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))))
using *state-to-strips-state-dom-is*[*OF assms(1)*, *of*
s ++ map-of (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))] *calculation(4*, *6*, *9)*
by *simp*
— TODO slow.
ultimately have (φ*S* Ψ (*s ++ map-of* (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*)))) (*v*, *a'*) = *Some False*
using *state-to-strips-state-range-is*[*OF assms(1)*,
of v a' s ++ map-of (*effect-of* ($\varphi_O{}^{-1}$ Ψ *op'*))]

by *simp*
} **note** nb_{13} = *this*
{
fix *v a*
assume (*v*, *a*) ∈ *dom ?t*
and (*v*, *a*) ∉ *dom* (*map-of* (*effect-to-assignments op'*))
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*(*3*)[*OF calculation*(*2*)]**..**
ultimately have *?t* (*v*, *a*) = *Some* (*the* (*s v*) = *a*)
using *state-to-strips-state-range-is*[*OF assms*(*1*)]
by *presburger*
} **note** nb_{14} = *this*
{
fix *v a*
assume (*v*, *a*) ∈ *dom ?t*
and *v-a-not-in*: (*v*, *a*) ∉ *dom* (*map-of* (*effect-to-assignments op'*))
moreover have (*v*, *a*) ∈ *dom ?s'*
using *calculation*(*1*, *2*)
unfolding *dom-map-add*
by *blast*
moreover have (*v*, *a*) ∈ (⋃ *v* ∈ { *v* | *v*. *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ *s v* ≠ *None* }.
{ (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* })
using *state-to-strips-state-dom-is*[*OF assms*(*1*)] *calculation*(*3*)
by *presburger*
moreover have *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$) **and** *s v* ≠ *None* **and** *a* ∈ $\mathcal{R}_+$ Ψ *v*
using *calculation*(*4*)
by *blast*+
— NOTE Hasn't this been proved before?
moreover {
have *dom* (*map-of* (*effect-to-assignments op'*)) = (⋃(*v*, *a*) ∈ *set* (*effect-of op*). { (*v*, *a*) })
∪ (⋃(*v*, *a*) ∈ *set* (*effect-of op*).
{ (*v*, *a'*) | *a'*. *a'* ∈ ($\mathcal{R}_+$ Ψ *v*) ∧ *a'* ≠ *a* })
unfolding nb_2
by *blast*
also have ... = (⋃(*v*, *a*) ∈ *set* (*effect-of op*). { (*v*, *a*) }
∪ { (*v*, *a'*) | *a'*. *a'* ∈ ($\mathcal{R}_+$ Ψ *v*) ∧ *a'* ≠ *a* })
by *blast*
finally have *dom* (*map-of* (*effect-to-assignments op'*))
= (⋃(*v*, *a*) ∈ *set* (*effect-of op*). { (*v*, *a*) }
∪ { (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* })
by *auto*
then have (*v*, *a*) ∉ (⋃(*v*, *a*) ∈ *set* (*effect-of op*).
{ (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* })
using *v-a-not-in*

```
    by blast
  }
  — TODO slow.
  moreover have v ∉ dom (map-of (effect-of op))
    using dom-map-of-conv-image-fst calculation
    by fastforce
  moreover have (s ++ map-of (effect-of (φ_O⁻¹ Ψ op′))) v = s v
    unfolding nb₁ map-add-dom-app-simps(3)[OF calculation(9)]
    by simp
  — TODO slow.
  moreover have (v, a) ∈ dom ?t′
    using state-to-strips-state-dom-is[OF assms(1), of
      s ++ map-of (effect-of (φ_O⁻¹ Ψ op′))] calculation(5, 6, 7, 8, 10)
    by simp
  ultimately have ?t′ (v, a) = Some (the (s v) = a)
    using state-to-strips-state-range-is[OF assms(1)]
    by presburger
} note nb₁₅ = this
— TODO refactor.
have nb₁₆: dom ?t = (⋃ v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None }.
    { (v, a) | a. a ∈ (R₊ Ψ v) })
  ∪ set (effect-of op)
  ∪ (⋃(v, a)∈set (effect-of op).
    {(v, a′) |a′. a′ ∈ (R₊ Ψ v) ∧ a′ ≠ a})
  unfolding dom-map-add nb₂
  using state-to-strips-state-dom-is[OF assms(1), of s]
  by auto
{
  {
    fix v a
    assume (v, a) ∈ dom ?t
    then consider (A) (v, a) ∈ dom (φ_S Ψ s)
      | (B) (v, a) ∈ dom (map-of (effect-to-assignments op′))
      by fast
    hence (v, a) ∈ dom ?t′
      proof (cases)
        case A
        then have v ∈ set ((Ψ)_{V+}) and s v ≠ None and a ∈ R₊ Ψ 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 ++ map-of (effect-of (φ_O⁻¹ Ψ op′))) v ≠ None
          using calculation(2)
          by simp
        ultimately show ?thesis
          unfolding state-to-strips-state-dom-element-iff[OF assms(1)]
          by blast
      next
        case B
```

```
      then have (v, a) ∈
          set (effect-of op)
        ∪ (⋃(v, a)∈set (effect-of op). { (v, a') | a'. a' ∈ ℛ₊ Ψ v ∧ a' ≠ a })
        unfolding nb₂
        by blast
      then consider (B₁) (v, a) ∈ set (effect-of op)
        | (B₂) (v, a) ∈ (⋃(v, a)∈set (effect-of op).
        { (v, a') | a'. a' ∈ ℛ₊ Ψ v ∧ a' ≠ a })
        by blast
      thm nb₁₂ nb₁₃ nb₂
      thus ?thesis
        proof (cases)
          case B₁
          then show ?thesis
            using nb₁₂
            by fast
        next
          case B₂
          then show ?thesis
            using nb₁₃ B
            by blast
        qed
    qed
  }
  moreover {
    let ?u = s ++ map-of (effect-of ((φ_O)⁻¹ Ψ op'))
    fix v a
    assume v-a-in-dom-t': (v, a) ∈ dom ?t'
    thm nb₅
    then have v-in-vs: v ∈ set ((Ψ)_𝒱₊)
      and u-of-v-is-not-None: ?u v ≠ None
      and a-in-range-of-v: a ∈ ℛ₊ Ψ v
      using state-to-strips-state-dom-element-iff[OF assms(1)]
        v-a-in-dom-t'
      by meson+
    {
      assume (v, a) ∉ dom ?t
      then have contradiction: (v, a) ∉
        (⋃v ∈ { v | v. v ∈ set ((Ψ)_𝒱₊) ∧ s v ≠ None}. { (v, a) |a. a ∈ ℛ₊ Ψ v
})
        ∪ set (effect-of op)
        ∪ (⋃(v, a)∈set (effect-of op). {(v, a') |a'. a' ∈ ℛ₊ Ψ v ∧ a' ≠ a})
        unfolding nb₁₆
        by fast
      hence False
        proof (cases map-of (effect-of ((φ_O)⁻¹ Ψ op')) v = None)
          case True
          then have s v ≠ None
            using u-of-v-is-not-None
```

```
            by simp
          then have (v, a) ∈ (⋃ v ∈ { v | v. v ∈ set ((Ψ)_{V+}) ∧ s v ≠ None}.
            { (v, a) |a. a ∈ R+ Ψ v })
            using v-in-vs a-in-range-of-v
            by blast
          thus ?thesis
            using contradiction
            by blast
        next
          case False
          then have v ∈ dom (map-of (effect-of op))
            using u-of-v-is-not-None nb1
            by blast
          then obtain a' where map-of-effect-of-op-v-is: map-of (effect-of op) v
= Some a'
            by blast
          then have v-a'-in: (v, a') ∈ set (effect-of op)
            using map-of-SomeD
            by fast
          then show ?thesis
            proof (cases a = a')
              case True
              then have (v, a) ∈ set (effect-of op)
                using v-a'-in
                by blast
              then show ?thesis
                using contradiction
                by blast
            next
              case False
              then have (v, a) ∈ (⋃(v, a)∈set (effect-of op).
                {(v, a') |a'. a' ∈ R+ Ψ v ∧ a' ≠ a})
                using v-a'-in calculation a-in-range-of-v
                by blast
              thus ?thesis
                using contradiction
                by fast
            qed
        qed
      }
      hence (v, a) ∈ dom ?t
        by argo
    }
    moreover have dom ?t ⊆ dom ?t' and dom ?t' ⊆ 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 nb_17 = 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.
    — NOTE Split on the (disjunct) domain variable sets of map-of (effect-to-assignments
op').
      then consider (A1) (v, a) ∈ set (effect-of op)
        | (A2) (v, a) ∈ (⋃(v, a) ∈ set (effect-of op).
          { (v, a') | a'. a' ∈ (R₊ Ψ v) ∧ a' ≠ a })
        using nb_2
        by fastforce
      then show ?thesis
        proof (cases)
          case A1
          then have ?t (v, a) = Some True
            unfolding map-add-dom-app-simps(1)[OF True]
            using nb_9(1)
            by fast
          moreover have ?t' (v, a) = Some True
            using nb_12[OF A1].
          ultimately show ?thesis..
        next
          case A2
          then have ?t (v, a) = Some False
            unfolding map-add-dom-app-simps(1)[OF True]
            using nb_9(2)
            by blast
          moreover have ?t' (v, a) = Some False
            using nb_13[OF True A2].
          ultimately show ?thesis..
        qed
    next
      case False
      moreover have ?t (v, a) = Some (the (s v) = a)
        using nb_14[OF v-a-in-dom-t False].
      moreover have ?t' (v, a) = Some (the (s v) = a)
        using nb_15[OF v-a-in-dom-t False].
      ultimately show ?thesis
        by argo
    qed
} note nb_18 = this
```

```
  moreover {
    fix v a
    assume (v, a) ∈ dom ?t'
    hence ?t (v, a) = ?t' (v, a)
      using nb_17 nb_18
      by presburger
  }
  — TODO slow.
  ultimately have ?t ⊆_m ?t' and ?t' ⊆_m ?t
    unfolding map-le-def
    by fastforce+
  thus ?thesis
    using map-le-antisym[of ?t ?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'* on the corresponding STRIPS state $s' = \varphi_S\ \Psi\ s$ yields the same state as executing the transformed SAS+ parallel operator $ops = [\varphi_O{}^{-1}\ (\varphi\ \Psi)\ op'.\ op' \leftarrow ops']$ on the original SAS+ state s and the transforming the resultant SAS+ state to its corresponding STRIPS state.

```
lemma sas-plus-equivalent-to-strips-i-a-XII:
  assumes is-valid-problem-sas-plus Ψ
    and ∀ op' ∈ set ops'. op' ∈ set ((φ Ψ)_O)
  shows execute-parallel-operator (φ_S Ψ s) ops'
    = φ_S Ψ (execute-parallel-operator-sas-plus s [φ_O⁻¹ Ψ op'. op' ← ops'])
using assms
proof (induction ops' arbitrary: s)
  case Nil
  then show ?case
    unfolding execute-parallel-operator-def execute-parallel-operator-sas-plus-def
    by simp
next
  case (Cons op' ops')
  let ?Π = φ Ψ
  let ?t' = (φ_S Ψ s) ++ map-of (effect-to-assignments op')
    and ?t = s ++ map-of (effect-of (φ_O⁻¹ Ψ op'))
  have nb_1: ?t' = φ_S Ψ ?t
    using sas-plus-equivalent-to-strips-i-a-XI[OF assms(1)] Cons.prems(2)
    by force
  {
    have ∀ op' ∈ set ops'. op' ∈ set (strips-problem.operators-of ?Π)
      using Cons.prems(2)
      by simp
    then have execute-parallel-operator (φ_S Ψ ?t) ops'
      = φ_S Ψ (execute-parallel-operator-sas-plus ?t [φ_O⁻¹ Ψ x. x ← ops'])
      using Cons.IH[OF Cons.prems(1), of ?t]
      by fastforce
```

hence *execute-parallel-operator ?t′ ops′*
$= \varphi_S\ \Psi$ (*execute-parallel-operator-sas-plus ?t* $[{\varphi_O}^{-1}\ \Psi\ x.\ x \leftarrow ops']$)
using nb_1
by *argo*
}
thus *?case*
by *simp*
qed

lemma *sas-plus-equivalent-to-strips-i-a-XIII*:
assumes *is-valid-problem-sas-plus* Ψ
and $\forall\ op' \in set\ ops'.\ op' \in set\ ((\varphi\ \Psi)_{\mathcal{O}})$
and $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan*
(*execute-parallel-operator* $(\varphi_S\ \Psi\ I)$ *ops′*) π
shows $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan*
($\varphi_S\ \Psi$ (*execute-parallel-operator-sas-plus I* $[{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$)) π
proof −
let *?I′* $= (\varphi_S\ \Psi\ I)$
and *?G′* $= \varphi_S\ \Psi\ G$
and *?ops* $= [{\varphi_O}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
and *?*$\Pi = \varphi\ \Psi$
let *?J = execute-parallel-operator-sas-plus I ?ops*
{
fix *v a*
assume $(v,\ a) \in dom\ ?G'$
then have *?G′* $(v,\ a) =$ *execute-parallel-plan*
(*execute-parallel-operator ?I′ ops′*) $\pi\ (v,\ a)$
using *assms*(*3*)
unfolding *map-le-def*
by *auto*
hence *?G′* $(v,\ a) =$ *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 ${\varphi_P}^{-1}\ (\varphi\ \Psi)\ \pi$ w.r.t to the original initial state I and original goal state G.
lemma *sas-plus-equivalent-to-strips-i-a*:
assumes *is-valid-problem-sas-plus* Ψ
and *dom I* $\subseteq$ *set* $((\Psi)_{\mathcal{V}^+})$

```
    and ∀ v ∈ dom I. the (I v) ∈ ℛ₊ Ψ v
    and dom G ⊆ set ((Ψ)_{𝒱+})
    and ∀ v ∈ dom G. the (G v) ∈ ℛ₊ Ψ v
    and ∀ ops' ∈ set π. ∀ op' ∈ set ops'. op' ∈ set ((φ Ψ)_𝒪)
    and (φ_S Ψ G) ⊆_m execute-parallel-plan (φ_S Ψ I) π
  shows G ⊆_m execute-parallel-plan-sas-plus I (φ_P⁻¹ Ψ π)
proof −
  let ?vs = variables-of Ψ
    and ?ψ = φ_P⁻¹ Ψ π
  show ?thesis
    using assms
    proof (induction π arbitrary: I)
      case Nil
      then have (φ_S Ψ G) ⊆_m (φ_S Ψ I)
        by fastforce
      then have G ⊆_m I
        using state-to-strips-state-map-le-iff[OF assms(1, 4, 5)]
        by blast
      thus ?case
      unfolding SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
          strips-parallel-plan-to-sas-plus-parallel-plan-def
        by fastforce
    next
      case (Cons ops' π)
      let ?D = range-of Ψ
        and ?Π = φ Ψ
        and ?I' = φ_S Ψ I
        and ?G' = φ_S Ψ G
      let ?ops = [φ_O⁻¹ Ψ op'. op' ← ops']
      let ?J = execute-parallel-operator-sas-plus I ?ops
        and ?J' = execute-parallel-operator ?I' ops'
      have nb₁: set ops' ⊆ set ((?Π)_𝒪)
        using Cons.prems(6)
        unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def list-all-iff
ListMem-iff
        by fastforce
      {
        fix op
        assume op ∈ set ?ops
        moreover obtain op' where op' ∈ set ops' and op = φ_O⁻¹ Ψ op'
          using calculation
          by auto
        moreover have op' ∈ set ((?Π)_𝒪)
          using nb₁ calculation(2)
          by blast
        moreover obtain op'' where op'' ∈ set ((Ψ)_{𝒪+}) and op' = φ_O Ψ op''
          using calculation(4)
          by auto
        moreover have op = op''
```

```
        using sas-plus-operator-inverse-is[OF assms(1) calculation(5)] calcula-
tion(3, 6)
        by presburger
      ultimately have op ∈ set ((Ψ)_{O+}) ∧ (∃ op' ∈ set ops'. op' = φ_O Ψ op)
        by blast
    } note nb_2 = this
    {
      fix op v a
      assume op ∈ set ((Ψ)_{O+}) and (v, a) ∈ set (effect-of op)
      moreover have op ∈ set ((Ψ)_{O+})
        using nb_2 calculation(1)
        by blast
      moreover have is-valid-operator-sas-plus Ψ op
        using is-valid-problem-sas-plus-then(2) Cons.prems(1) calculation(3)
        by blast
      ultimately have v ∈ set ((Ψ)_{V+})
        using is-valid-operator-sas-plus-then(3)
        by fastforce
    } note nb_3 = this
    {
      fix op
      assume op ∈ set ?ops
      then have op ∈ set ((Ψ)_{O+})
        using nb_2
        by blast
      then have is-valid-operator-sas-plus Ψ op
        using is-valid-problem-sas-plus-then(2) Cons.prems(1)
        by blast
      hence ∀ (v, a) ∈ set (effect-of op). v ∈ set ((Ψ)_{V+})
        ∧ a ∈ R_+ Ψ v
        using is-valid-operator-sas-plus-then(3,4)
        by fast
    } note nb_4 = this
    show ?case
      proof (cases STRIPS-Semantics.are-all-operators-applicable ?I' ops'
        ∧ STRIPS-Semantics.are-all-operator-effects-consistent ops')
        case True
        {
          {
            have dom I ⊆ set ((Ψ)_{V+})
              using Cons.prems(2)
              by blast
            hence (φ_S^{-1} Ψ ?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) nb_1, of I] True
      by simp
    moreover have (φ_P^-1 Ψ (ops' # π)) = ?ops # (φ_P^-1 Ψ π)
  unfolding SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
      strips-parallel-plan-to-sas-plus-parallel-plan-def
      SAS-Plus-STRIPS.strips-op-to-sasp-def
        strips-op-to-sasp-def
      by simp
    ultimately have execute-parallel-plan-sas-plus I (φ_P^-1 Ψ (ops' # π))
      = execute-parallel-plan-sas-plus ?J (φ_P^-1 Ψ π)
      by force
  } note nb_5 = this
  — Show the goal using the IH.
  {
    have dom-J-subset-eq-vs: dom ?J ⊆ set ((Ψ)_V+)
      using sas-plus-equivalent-to-strips-i-a-IX[OF Cons.prems(2)] nb_2 nb_4
      by blast
    moreover {
      have set ((Ψ)_V+) ⊆ dom (range-of Ψ)
        using is-valid-problem-sas-plus-then(1)[OF assms(1)]
        by fastforce
      moreover have ∀ v ∈ dom I. the (I v) ∈ set (the (range-of Ψ v))
    using Cons.prems(2, 3) assms(1) set-the-range-of-is-range-of-sas-plus-if

        by force
      moreover have ∀ op ∈ set ?ops. ∀ (v, a) ∈ set (effect-of op).
        v ∈ set ((Ψ)_V+) ∧ a ∈ set (the (?D v))
        using set-the-range-of-is-range-of-sas-plus-if assms(1) nb_4
        by fastforce
     moreover have v-in-dom-J-range: ∀ v ∈ dom ?J. the (?J v) ∈ set (the
(?D v))
        using sas-plus-equivalent-to-strips-i-a-X[of
            I set ((Ψ)_V+) ?D ?ops, OF Cons.prems(2)] calculation(1, 2, 3)
        by fastforce
      {
        fix v
        assume v ∈ dom ?J
        moreover have v ∈ set ((Ψ)_V+)
          using nb_2 calculation dom-J-subset-eq-vs
          by blast
        moreover have set (the (range-of Ψ v)) = R_+ Ψ v
          using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
            calculation(2)
          by presburger
        ultimately have the (?J v) ∈ R_+ Ψ v
          using nb_3 v-in-dom-J-range
          by blast
      }
      ultimately have ∀ v ∈ dom ?J. the (?J v) ∈ R_+ Ψ v
```

```
            by fast
          }
          moreover have ∀ ops' ∈ set π. ∀ op'∈set ops'. op' ∈ set ((φ Ψ)_𝒪)
            using Cons.prems(6)
            by simp
          moreover {
            have ?G' ⊆_m execute-parallel-plan ?J' π
              using Cons.prems(7) True
              by auto
            hence (φ_S Ψ G) ⊆_m execute-parallel-plan (φ_S Ψ ?J) π
              using sas-plus-equivalent-to-strips-i-a-XIII[OF Cons.prems(1)] nb_1
              by blast
          }
          ultimately have G ⊆_m execute-parallel-plan-sas-plus I (φ_P⁻¹ Ψ (ops'
# π))
              using Cons.IH[of ?J, OF Cons.prems(1) - - Cons.prems(4, 5)]
Cons.prems(6) nb_5
            by presburger
        }
        thus ?thesis.
      next
        case False
        then have ?G' ⊆_m ?I'
          using Cons.prems(7)
          by force
        moreover {
          have dom I ⊆ set ?vs
            using Cons.prems(2)
            by simp
          hence ¬(are-all-operators-applicable-in I ?ops
            ∧ are-all-operator-effects-consistent ?ops)
              using sas-plus-equivalent-to-strips-i-a-VIII[OF Cons.prems(1) -
Cons.prems(3) nb_1]
              False
            by force
        }
        moreover {
          have (φ_P⁻¹ Ψ (ops' # π)) = ?ops # (φ_P⁻¹ Ψ π)
        unfolding SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
            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 ⊆_m execute-parallel-plan-sas-plus I (?ops # (φ_P⁻¹ Ψ π))
            ⟷ G ⊆_m I
            using calculation(2)
            by force
        }
        ultimately show ?thesis
```

```
      using state-to-strips-state-map-le-iff[OF Cons.prems(1, 4, 5)]
    unfolding SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
      strips-parallel-plan-to-sas-plus-parallel-plan-def
      SAS-Plus-STRIPS.strips-op-to-sasp-def
      strips-op-to-sasp-def
    by force
  qed
 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+ Ψ problem to the corresponding STRIPS problem Π (this is implicitely also valid by lemma *is-valid-problem-sas-plus-then-strips-transformation-too*); then,
2. get a solution π—if it exists—for the induced STRIPS problem by executing SATPlan; and finally,
3. convert π back to a solution ψ for the SAS+ problem.

```
lemma sas-plus-equivalent-to-strips-i:
 assumes is-valid-problem-sas-plus Ψ
   and STRIPS-Semantics.is-parallel-solution-for-problem
   (φ Ψ) π
 shows goal-of Ψ ⊆m execute-parallel-plan-sas-plus
   (sas-plus-problem.initial-of Ψ) (φP⁻¹ Ψ π)
proof −
 let ?vs = variables-of Ψ
   and ?I = initial-of Ψ
   and ?G = goal-of Ψ
 let ?Π = φ Ψ
 let ?G' = strips-problem.goal-of ?Π
   and ?I' = strips-problem.initial-of ?Π
 let ?ψ = φP⁻¹ Ψ π
 have dom ?I ⊆ set ?vs
   using is-valid-problem-sas-plus-then(3) assms(1)
   by auto
 moreover have ∀ v ∈ dom ?I. the (?I v) ∈ R+ Ψ v
   using is-valid-problem-sas-plus-then(4) assms(1) calculation
   by auto
 moreover have dom ?G ⊆ set ?vs and ∀ v ∈ dom ?G. the (?G v) ∈ R+ Ψ v
   using is-valid-problem-sas-plus-then(5, 6) assms(1)
   by blast+
 moreover have ∀ ops'∈set π. ∀ op'∈set ops'. op' ∈ set ((?Π)O)
   using is-parallel-solution-for-problem-operator-set[OF assms(2)]
   by simp
 moreover {
   have ?G' ⊆m execute-parallel-plan ?I' π
```

```
    using assms(2)
    unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def..
  moreover have ?G' = φ_S Ψ ?G and ?I' = φ_S Ψ ?I
    by simp+
  ultimately have (φ_S Ψ ?G) ⊆_m execute-parallel-plan (φ_S Ψ ?I) π
    by simp
 }
 ultimately show ?thesis
  using sas-plus-equivalent-to-strips-i-a[OF assms(1)]
  by simp
qed
```

— NOTE Show that the operators for a given solution π 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 Ψ
  and STRIPS-Semantics.is-parallel-solution-for-problem (φ Ψ) π
 shows list-all (list-all (λop. ListMem op (operators-of Ψ))) ((φ_P)^-1 Ψ π)
proof −
 let ?Π = φ Ψ
 let ?ops = operators-of Ψ
  and ?ψ = (φ_P)^-1 Ψ π
 have is-valid-problem-strips ?Π
  using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]
  by simp
 have nb_1: ∀ op' ∈ set ((?Π)_O). (∃ op ∈ set ?ops. op' = (φ_O Ψ op))
  by auto
 {
  fix ops' op' op
  assume ops' ∈ set π and op' ∈ set ops'
  then have op' ∈ set (strips-problem.operators-of ?Π)
   using is-parallel-solution-for-problem-operator-set[OF assms(2)]
   by simp
  then obtain op where op ∈ set ((Ψ)_O+) and op' = (φ_O Ψ op)
   by auto
  then have ((φ_O)^-1 Ψ op') ∈ set ((Ψ)_O+)
   using sas-plus-operator-inverse-is[OF assms(1)]
   by presburger
 }
 thus ?thesis
  unfolding list-all-iff ListMem-iff
   strips-parallel-plan-to-sas-plus-parallel-plan-def
   SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
   SAS-Plus-STRIPS.strips-op-to-sasp-def
   strips-op-to-sasp-def
  by auto
qed
```

We now show that for a parallel solution π of Π the SAS+ plan $\psi \equiv {\varphi_P}^{-1}$

Ψ π yielded by the STRIPS to SAS+ plan transformation is a solution for Ψ. The proof uses the definition of parallel STRIPS solutions and shows that the execution of ψ on the initial state of the SAS+ problem yields a state satisfying the problem's goal state, i.e.

$G \subseteq_m$ *execute-parallel-plan-sas-plus* I ψ

and by showing that all operators in all parallel operators of ψ are operators of the problem.

theorem
sas-plus-equivalent-to-strips:
assumes *is-valid-problem-sas-plus* Ψ
and *STRIPS-Semantics.is-parallel-solution-for-problem* (φ Ψ) π
shows *is-parallel-solution-for-problem* Ψ (${\varphi_P}^{-1}$ Ψ π)
proof −
let *?I* = *initial-of* Ψ
and *?G* = *goal-of* Ψ
and *?ops* = *operators-of* Ψ
and *?ψ* = ${\varphi_P}^{-1}$ Ψ π
show *?thesis*
unfolding *is-parallel-solution-for-problem-def Let-def*
proof (*rule conjI*)
show *?G* $\subseteq_m$ *execute-parallel-plan-sas-plus ?I ?ψ*
using *sas-plus-equivalent-to-strips-i*[*OF assms*].
next
show *list-all* (*list-all* (λ*op. ListMem op ?ops*)) *?ψ*
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* Ψ
and ∀ *op* ∈ *set ops*. *op* ∈ *set* ((Ψ)$_{\mathcal{O}^+}$)
and *op′* ∈ *set* [φ_O Ψ *op*. *op* ← *ops*]
obtains *op* **where** *op* ∈ *set ops*
and *op′* = φ_O Ψ *op*
proof −
let *?*Π = φ Ψ
let *?ops* = *operators-of* Ψ
obtain *op* **where** *op* ∈ *set ops* **and** *op′* = φ_O Ψ *op*
using *assms*(*3*)
by *auto*
thus *?thesis*
using *that*
by *blast*
qed

private corollary *strips-equivalent-to-sas-plus-i-a-II*:
assumes*is-valid-problem-sas-plus* Ψ

```
    and ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
    and op' ∈ set [φ_O Ψ op. op ← ops]
  shows op' ∈ set ((φ Ψ)_O)
    and is-valid-operator-strips (φ Ψ) op'
proof −
  let ?Π = φ Ψ
  let ?ops = operators-of Ψ
    and ?ops' = strips-problem.operators-of ?Π
  obtain op where op-in: op ∈ set ops and op'-is: op' = φ_O Ψ op
    using strips-equivalent-to-sas-plus-i-a-I[OF assms].
  then have nb: op' ∈ set ((φ Ψ)_O)
    using assms(2) op-in op'-is
    by fastforce
  thus op' ∈ set ((φ Ψ)_O)
    and is-valid-operator-strips ?Π op'
    proof −
      have ∀ op' ∈ set ?ops'. is-valid-operator-strips ?Π op'
      using is-valid-problem-sas-plus-then-strips-transformation-too-iii[OF assms(1)]
        unfolding list-all-iff.
      thus is-valid-operator-strips ?Π op'
        using nb
        by fastforce
    qed fastforce
qed


lemma strips-equivalent-to-sas-plus-i-a-III:
  assumes is-valid-problem-sas-plus Ψ
    and ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
  shows execute-parallel-operator (φ_S Ψ s) [φ_O Ψ op. op ← ops]
    = (φ_S Ψ (execute-parallel-operator-sas-plus s ops))
proof −
  {
    fix op s
    assume op ∈ set ((Ψ)_{O+})
    moreover have (φ_O Ψ op) ∈ set ((φ Ψ)_O)
      using calculation
      by simp
    moreover have (φ_S Ψ s) ++ map-of (effect-to-assignments (φ_O Ψ op))
      = (φ_S Ψ (s ++ map-of (effect-of (φ_O^{-1} Ψ (φ_O Ψ op)))))
      using sas-plus-equivalent-to-strips-i-a-XI[OF assms(1) calculation(2)]
      by blast
    moreover have (φ_O^{-1} Ψ (φ_O Ψ op)) = op
      using sas-plus-operator-inverse-is[OF assms(1) calculation(1)].
    ultimately have (φ_S Ψ s) ≫ (φ_O Ψ op)
      = (φ_S Ψ (s ≫_+ op))
      unfolding execute-operator-def execute-operator-sas-plus-def
      by simp
  } note nb_1 = this
```

```
  show ?thesis
    using assms
    proof (induction ops arbitrary: s)
      case Nil
      then show ?case
       unfolding execute-parallel-operator-def execute-parallel-operator-sas-plus-def

        by simp
    next
      case (Cons op ops)
      let ?t = s ≫₊ op
      let ?s' = φ_S Ψ s
        and ?ops' = [φ_O Ψ op. op ← op # ops]
      let ?t' = ?s' ≫ (φ_O Ψ op)
      have execute-parallel-operator ?s' ?ops'
        = execute-parallel-operator ?t' [φ_O Ψ x. x ← ops]
        unfolding execute-operator-def
        by simp
      moreover have (φ_S Ψ (execute-parallel-operator-sas-plus s (op # ops)))
        = (φ_S Ψ (execute-parallel-operator-sas-plus ?t ops))
        unfolding execute-operator-sas-plus-def
        by simp
      moreover {
        have ?t' = (φ_S Ψ ?t)
          using nb_1 Cons.prems(2)
          by simp
        hence execute-parallel-operator ?t'[φ_O Ψ x. x ← ops]
          = (φ_S Ψ (execute-parallel-operator-sas-plus ?t ops))
          using Cons.IH[of ?t] Cons.prems
          by simp
      }
      ultimately show ?case
        by argo
    qed
qed

private lemma strips-equivalent-to-sas-plus-i-a-IV:
  assumes is-valid-problem-sas-plus Ψ
    and ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
    and are-all-operators-applicable-in I ops
    ∧ are-all-operator-effects-consistent ops
  shows STRIPS-Semantics.are-all-operators-applicable (φ_S Ψ I) [φ_O Ψ op. op ←
ops]
    ∧ STRIPS-Semantics.are-all-operator-effects-consistent [φ_O Ψ op. op ← ops]
proof −
  let ?vs = variables-of Ψ
    and ?ops = operators-of Ψ
  let ?I' = φ_S Ψ I
    and ?ops' = [φ_O Ψ op. op ← ops]
```

have nb_1: $\forall\, op \in set\ ops.\ is\text{-}operator\text{-}applicable\text{-}in\ I\ op$
using *assms*(3)
unfolding *are-all-operators-applicable-in-def list-all-iff*
by *blast*
have nb_2: $\forall\, op \in set\ ops.\ is\text{-}valid\text{-}operator\text{-}sas\text{-}plus\ \Psi\ op$
using *is-valid-problem-sas-plus-then*(2) *assms*(1, 2)
unfolding *is-valid-operator-sas-plus-def*
by *auto*
have nb_3: $\forall\, op \in set\ ops.\ map\text{-}of\ (precondition\text{-}of\ op) \subseteq_m I$
using nb_1
unfolding *is-operator-applicable-in-def list-all-iff*
by *blast*
{
fix $op_1\ op_2$
assume $op_1 \in set\ ops$ **and** $op_2 \in set\ ops$
hence *are-operator-effects-consistent* $op_1\ op_2$
using *assms*(3)
unfolding *are-all-operator-effects-consistent-def list-all-iff*
by *blast*
} **note** $nb_4 = this$
{
fix $op_1\ op_2$
assume $op_1 \in set\ ops$ **and** $op_2 \in set\ ops$
hence $\forall\,(v, a) \in set\ (effect\text{-}of\ op_1).\ \forall\,(v', a') \in set\ (effect\text{-}of\ op_2).$
$v \neq v' \vee a = a'$
using nb_4
unfolding *are-operator-effects-consistent-def Let-def list-all-iff*
by *presburger*
} **note** $nb_5 = this$
{
fix $op_1{}'\ op_2{}'\ I$
assume $op_1{}' \in set\ ?ops'$
and $op_2{}' \in set\ ?ops'$
and $\exists\,(v, a) \in set\ (add\text{-}effects\text{-}of\ op_1{}').\ \exists\,(v', a') \in set\ (delete\text{-}effects\text{-}of\ op_2{}').$
$(v, a) = (v', a')$
moreover obtain $op_1\ op_2$
where $op_1 \in set\ ops$
and $op_1{}' = \varphi_O\ \Psi\ op_1$
and $op_2 \in set\ ops$
and $op_2{}' = \varphi_O\ \Psi\ op_2$
using *strips-equivalent-to-sas-plus-i-a-I*[*OF assms*(1, 2)] *calculation*(1, 2)
by *auto*
moreover have *is-valid-operator-sas-plus* $\Psi\ op_1$
and *is-valid-operator-op*$_2$: *is-valid-operator-sas-plus* $\Psi\ op_2$
using *calculation*(4, 6) nb_2
by *blast*+
moreover obtain $v\ v'\ a\ a'$
where $(v, a) \in set\ (add\text{-}effects\text{-}of\ op_1{}')$
and $(v', a') \in set\ (delete\text{-}effects\text{-}of\ op_2{}')$

and $(v,\ a) = (v',\ a')$
using *calculation*
by *blast*
moreover have $(v,\ a) \in$ *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 $(v',\ a') \in (\bigcup (v,\ a) \in$ *set* (*effect-of* op_2).
$\{\ (v,\ a') \mid a'.\ a' \in (\mathcal{R}_+\ \Psi\ v) \wedge\ a' \neq a\ \})$
using *sasp-op-to-strips-set-delete-effects-is*
calculation(*7*, *9*, *11*)
by *blast*
then obtain $v''\ a''$ **where** $(v'',\ a'') \in$ *set* (*effect-of* op_2)
and $(v',\ a') \in \{\ (v'',\ a''') \mid a'''.\ a''' \in (\mathcal{R}_+\ \Psi\ v'') \wedge\ a''' \neq a''\ \}$
by *blast*
moreover have $(v',\ a'') \in$ *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 $\exists\, a''.\ (v',\ a'') \in$ *set* (*effect-of* op_2) $\wedge\ a' \in (\mathcal{R}_+\ \Psi\ v')$
$\wedge\ a' \neq a''$
by *blast*
}
moreover obtain a'' **where** $a' \in \mathcal{R}_+\ \Psi\ v'$
and $(v',\ a'') \in$ *set* (*effect-of* op_2)
and $a' \neq a''$
using *calculation*(*16*)
by *blast*
moreover have $\exists\, (v,\ a) \in$ *set* (*effect-of* op_1). $(\exists\, (v',\ a') \in$ *set* (*effect-of* op_2).
$v = v' \wedge a \neq a')$
using *calculation*(*13*, *14*, *15*, *17*, *18*, *19*)
by *blast*
— TODO slow.
ultimately have $\exists\, op_1 \in$ *set ops*. $\exists\, op_2 \in$ *set ops*. $\neg$*are-operator-effects-consistent* $op_1\ op_2$
unfolding *are-operator-effects-consistent-def list-all-iff*
by *fastforce*
} note $nb_6 =$ *this*
show *?thesis*
proof (*rule conjI*)
{
fix op'

assume $op' \in set\ ?ops'$
moreover obtain *op* **where** *op-in*: $op \in set\ ops$
and *op'-is*: $op' = \varphi_O\ \Psi\ op$
and *op'-in*: $op' \in set\ ((\varphi\ \Psi)_{\mathcal{O}})$
and *is-valid-op'*: *is-valid-operator-strips* $(\varphi\ \Psi)\ op'$
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 nb_2 *calculation*(*2*)**..**
{
fix *v a*
assume *v-a-in-preconditions'*: $(v, a) \in set$ (*strips-operator.precondition-of op'*)
have *v-a-in-preconditions*: $(v, a) \in set$ (*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 $v \in set\ ?vs$ **and** $a \in \mathcal{R}_+\ \Psi\ v$
using *is-valid-operator-sas-plus-then*(*1*,*2*) *is-valid-op calculation*(*1*)
by *fastforce+*
moreover {
have $\forall (v, a) \in set$ (*precondition-of op*). $\forall (v', a') \in set$ (*precondition-of op*).
$v \neq v' \vee a = a'$
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 $v \in dom$ (*map-of* (*precondition-of op*))
using *calculation*(*4*)
by *blast*
moreover have $I\ v$ = *Some a*
using nb_3
unfolding *map-le-def*
using *op-in calculation*(*4*, *5*)
by *metis*
moreover have $(v, a) \in dom\ ?I'$
using *state-to-strips-state-dom-element-iff*[*OF assms*(*1*)]
calculation(*2*, *3*, *6*)
by *simp*
ultimately have $?I'\ (v, a)$ = *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 op₁' op₂'
    assume op₁'-in-ops': op₁' ∈ set ?ops' and op₂'-in-ops': op₂' ∈ set ?ops'
    have STRIPS-Semantics.are-operator-effects-consistent op₁' op₂'
     unfolding STRIPS-Semantics.are-operator-effects-consistent-def Let-def
     — TODO proof is symmetrical... refactor into nb.
     proof (rule conjI)
      show ¬list-ex (λx. list-ex ((=) x) (delete-effects-of op₂'))
       (add-effects-of op₁')
       proof (rule ccontr)
        assume ¬¬list-ex (λv. list-ex ((=) v) (delete-effects-of op₂'))
         (add-effects-of op₁')
        then have ∃(v, a) ∈ set (delete-effects-of op₂').
         ∃(v', a') ∈ set (add-effects-of op₁'). (v, a) = (v', a')
         unfolding list-ex-iff
         by fastforce
        then obtain op₁ op₂ where op₁ ∈ set ops
         and op₂ ∈ set ops
         and ¬are-operator-effects-consistent op₁ op₂
         using nb₆[OF op₁'-in-ops' op₂'-in-ops']
         by blast
        thus False
         using nb₄
         by blast
       qed
     next
      show ¬list-ex (λv. list-ex ((=) v) (add-effects-of op₂')) (delete-effects-of
op₁')
       proof (rule ccontr)
        assume ¬¬list-ex (λv. list-ex ((=) v) (add-effects-of op₂'))
         (delete-effects-of op₁')
        then have ∃(v, a) ∈ set (delete-effects-of op₁').
         ∃(v', a') ∈ set (add-effects-of op₂'). (v, a) = (v', a')
         unfolding list-ex-iff
         by fastforce
        then obtain op₁ op₂ where op₁ ∈ set ops
         and op₂ ∈ set ops
         and ¬are-operator-effects-consistent op₁ op₂
         using nb₆[OF op₂'-in-ops' op₁'-in-ops']
```

```
            by blast
          thus False
            using nb_4
            by blast
        qed
      qed
    }
    thus STRIPS-Semantics.are-all-operator-effects-consistent ?ops'
    unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
      by blast
  qed
qed

private lemma strips-equivalent-to-sas-plus-i-a-V:
  assumes is-valid-problem-sas-plus Ψ
    and ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
    and ¬(are-all-operators-applicable-in s ops
    ∧ are-all-operator-effects-consistent ops)
  shows ¬(STRIPS-Semantics.are-all-operators-applicable (φ_S Ψ s) [φ_O Ψ op. op
← ops]
    ∧ STRIPS-Semantics.are-all-operator-effects-consistent [φ_O Ψ op. op ← ops])
proof −
  let ?vs = variables-of Ψ
    and ?ops = operators-of Ψ
  let ?s' = φ_S Ψ s
    and ?ops' = [φ_O Ψ op. op ← ops]
  {
    fix op
    assume op ∈ set ops
    hence ∃ op' ∈ set ?ops'. op' = φ_O Ψ op
      by simp
  } note nb_1 = this
  {
    fix op
    assume op ∈ set ops
    then have op ∈ set ((Ψ)_{O+})
      using assms(2)
      by blast
    then have is-valid-operator-sas-plus Ψ op
      using is-valid-problem-sas-plus-then(2) assms(1)
      unfolding is-valid-operator-sas-plus-def
      by auto
    hence ∀(v, a) ∈ set (precondition-of op). ∀(v', a') ∈ set (precondition-of op).
      v ≠ v' ∨ a = a'
      using is-valid-operator-sas-plus-then(5)
      unfolding is-valid-operator-sas-plus-def
      by fast
  } note nb_2 = this
  {
```

consider (*A*) ¬*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'* = φ_O Ψ *op*
using *nb*$_1$
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*) ∈ *set* (*precondition-of op*)
using *op'-is calculation*(*3*)
unfolding *sasp-op-to-strips-def Let-def*

```
            by fastforce
          moreover have v ∈ set ?vs
            and a ∈ ℛ₊ Ψ v
            and s v ≠ None
            using state-to-strips-state-dom-element-iff[OF assms(1)]
              calculation(6)
            by simp+
          moreover have ?s' (v, a) = Some (the (s v) = a)
            using state-to-strips-state-range-is[OF
                assms(1) calculation(6)].
          moreover have the (s v) = a
            using calculation(5, 11)
            by fastforce
          moreover have s v = Some a
            using calculation(12) option.collapse[OF calculation(10)]
            by argo
          moreover have map-of (precondition-of op) v = Some a
              using map-of-constant-assignments-defined-if[OF nb₂[OF op-in]
calculation(7)].
          ultimately have map-of (precondition-of op) v = s v
            by argo
        }
        then have map-of (precondition-of op) ⊆ₘ s
          unfolding map-le-def
          by blast
        thus False
          using not-precondition-map-le-s
          by simp
      qed
    thus ?thesis
      by simp
  next
    case B
    {
      obtain op₁ op₂ v v' a a'
        where op₁ ∈ set ops
          and op₂-in: op₂ ∈ set ops
          and v-a-in: (v, a) ∈ set (effect-of op₁)
          and v'-a'-in: (v', a') ∈ set (effect-of op₂)
          and v-is: v = v' and a-is: a ≠ a'
        using B
        unfolding are-all-operator-effects-consistent-def
          are-operator-effects-consistent-def list-all-iff Let-def
        by blast
      moreover obtain op₁' op₂' where op₁' ∈ set ?ops' and op₁' = φ_O Ψ op₁
        and op₁' ∈ set ?ops' and op₂'-is: op₂' = φ_O Ψ op₂
        using nb₁[OF calculation(1)] nb₁[OF calculation(2)]
        by blast
      moreover have (v, a) ∈ set (add-effects-of op₁')
```

```
      using calculation(3, 8)
      unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
        sasp-op-to-strips-def Let-def
      by force
    moreover {
      have is-valid-operator-sas-plus Ψ op₁
     using assms(2) calculation(1) is-valid-problem-sas-plus-then(2) assms(1)
        unfolding is-valid-operator-sas-plus-def
        by auto
      moreover have is-valid-operator-sas-plus Ψ op₂
        using sublocale-sas-plus-finite-domain-representation-ii(2)[
            OF assms(1)] assms(2) op₂-in
        by blast
      moreover have a ∈ 𝒱₊ Ψ v
        using is-valid-operator-sas-plus-then(4) calculation v-a-in
        unfolding is-valid-operator-sas-plus-def
        by fastforce
      ultimately have (v, a) ∈ set (delete-effects-of op₂')
        using sasp-op-to-strips-set-delete-effects-is[of Ψ op₂]
          v'-a'-in v-is a-is
        using op₂'-is
        by blast
    }
    — TODO slow.
    ultimately have ∃ op₁' ∈ set ?ops'. ∃ op₂' ∈ set ?ops'.
     ∃(v, a) ∈ set (delete-effects-of op₂'). ∃(v', a') ∈ set (add-effects-of op₁').
      (v, a) = (v', a')
      by fastforce
  }
  then have ¬STRIPS-Semantics.are-all-operator-effects-consistent ?ops'
    unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def
     STRIPS-Semantics.are-operator-effects-consistent-def list-all-iff list-ex-iff
Let-def
    by blast
  thus ?thesis
    by simp
 qed
}
thus ?thesis
  by blast
qed


lemma strips-equivalent-to-sas-plus-i-a:
  assumes is-valid-problem-sas-plus Ψ
    and dom I ⊆ set ((Ψ)𝒱₊)
    and ∀ v ∈ dom I. the (I v) ∈ 𝒱₊ Ψ v
    and dom G ⊆ set ((Ψ)𝒱₊)
    and ∀ v ∈ dom G. the (G v) ∈ 𝒱₊ Ψ v
```

```
    and ∀ ops ∈ set ψ. ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
    and G ⊆_m execute-parallel-plan-sas-plus I ψ
  shows (φ_S Ψ G) ⊆_m execute-parallel-plan (φ_S Ψ I) (φ_P Ψ ψ)
proof −
  let ?Π = φ Ψ
    and ?G' = φ_S Ψ G
  show ?thesis
    using assms
    proof (induction ψ arbitrary: I)
      case Nil
      let ?I' = φ_S Ψ I
      have G ⊆_m I
        using Nil
        by simp
      moreover have ?G' ⊆_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
      case (Cons ops ψ)
      let ?vs = variables-of Ψ
        and ?ops = operators-of Ψ
        and ?J = execute-parallel-operator-sas-plus I ops
        and ?π = φ_P Ψ (ops # ψ)
      let ?I' = φ_S Ψ I
        and ?J' = φ_S Ψ ?J
        and ?ops' = [φ_O Ψ op. op ← ops]
      {
        fix op v a
        assume op ∈ set ops and (v, a) ∈ set (effect-of op)
        moreover have op ∈ set ?ops
          using Cons.prems(6) calculation(1)
          by simp
        moreover have is-valid-operator-sas-plus Ψ 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 ∈ set ((Ψ)_{V+})
          and a ∈ R_+ Ψ v
          using is-valid-operator-sas-plus-then(3,4)
          by fastforce+
      } note nb_1 = this
      show ?case
      proof (cases are-all-operators-applicable-in I ops
        ∧ are-all-operator-effects-consistent ops)
        case True
```

```
{
  have (φ_P Ψ (ops # ψ)) = ?ops' # (φ_P Ψ 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
      SAS-Plus-STRIPS.sasp-op-to-strips-def
    by simp
  moreover have ∀ op ∈ set ops. op ∈ set ((Ψ)_{O+})
    using Cons.prems(6)
    by simp
  moreover have STRIPS-Semantics.are-all-operators-applicable ?I' ?ops'
    and STRIPS-Semantics.are-all-operator-effects-consistent ?ops'
      using strips-equivalent-to-sas-plus-i-a-IV[OF Cons.prems(1) - True]
calculation
    by blast+
  ultimately have execute-parallel-plan ?I' ?π
    = execute-parallel-plan (execute-parallel-operator ?I' ?ops') (φ_P Ψ ψ)
    by fastforce
}
  — NOTE Instantiate the IH on the next state of the SAS+ execution
execute-parallel-operator-sas-plus I ops.
moreover
{
  {
    have dom I ⊆ set (sas-plus-problem.variables-of Ψ)
      using Cons.prems(2)
      by blast
    moreover have ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op).
      v ∈ set ((Ψ)_{V+})
      using nb_1(1)
      by blast
    ultimately have dom ?J ⊆ set ((Ψ)_{V+})
      using sas-plus-equivalent-to-strips-i-a-IX[of I set ?vs]
      by simp
  } note nb_2 = this
  moreover {
    have dom I ⊆ set (sas-plus-problem.variables-of Ψ)
      using Cons.prems(2)
      by blast
    moreover have set (sas-plus-problem.variables-of Ψ)
      ⊆ dom (range-of Ψ)
     using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)
      by auto
   moreover {
      fix v
      assume v ∈ dom I
      moreover have v ∈ set ((Ψ)_{V+})
        using Cons.prems(2) calculation
        by blast
```

```
            ultimately have the (I v) ∈ set (the (range-of Ψ v))
              using Cons.prems(3)
              using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
              by blast
            }
            moreover have ∀ op∈set ops. ∀ (v, a)∈set (effect-of op).
              v ∈ set (sas-plus-problem.variables-of Ψ) ∧ a ∈ set (the (range-of Ψ
v))
                using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)] nb₁(1)
nb₁(2)
              by force
            moreover have nb₃: ∀ v ∈ dom ?J. the (?J v) ∈ set (the (range-of Ψ
v))
              using sas-plus-equivalent-to-strips-i-a-X[of I set ?vs range-of Ψ ops]
                calculation
              by fast
            moreover {
              fix v
              assume v ∈ dom ?J
              moreover have v ∈ set ((Ψ)ᵥ₊)
                using nb₂ calculation
                by blast
              moreover have set (the (range-of Ψ v)) = ℛ₊ Ψ v
                using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
                  calculation(2)
                by presburger
              ultimately have the (?J v) ∈ ℛ₊ Ψ v
                using nb₃
                by blast
            }
            ultimately have ∀ v ∈ dom ?J. the (?J v) ∈ ℛ₊ Ψ v
              by fast
          }
          moreover have ∀ ops∈set ψ. ∀ op∈set ops. op ∈ set ?ops
            using Cons.prems(6)
            by auto
          moreover have G ⊆ₘ execute-parallel-plan-sas-plus ?J ψ
            using Cons.prems(7) True
            by simp
          ultimately have (φₛ Ψ G) ⊆ₘ execute-parallel-plan ?J' (φₚ Ψ ψ)
            using Cons.IH[of ?J, OF Cons.prems(1) - - Cons.prems(4, 5)]
            by fastforce
        }
        moreover have execute-parallel-operator ?I' ?ops' = ?J'
       using assms(1) strips-equivalent-to-sas-plus-i-a-III[OF assms(1)] Cons.prems(6)
          by auto
        ultimately show ?thesis
          by argo
      next
```

```
      case False
      then have nb: G ⊆m I
        using Cons.prems(7)
        by force
      moreover {
        have ?π = ?ops' # (φP Ψ 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
            SAS-Plus-STRIPS.sasp-op-to-strips-def Let-def
          by auto
        moreover have set ?ops' ⊆ set (strips-problem.operators-of ?Π)
         using strips-equivalent-to-sas-plus-i-a-II(1)[OF assms(1)] Cons.prems(6)
          by auto
      moreover have ¬(STRIPS-Semantics.are-all-operators-applicable ?I' ?ops'

          ∧ STRIPS-Semantics.are-all-operator-effects-consistent ?ops')
       using strips-equivalent-to-sas-plus-i-a-V[OF assms(1) - False] Cons.prems(6)
          by force
        ultimately have execute-parallel-plan ?I' ?π = ?I'
          by auto
      }
      moreover have ?G' ⊆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
qed


lemma strips-equivalent-to-sas-plus-i:
  assumes is-valid-problem-sas-plus Ψ
    and is-parallel-solution-for-problem Ψ ψ
  shows (strips-problem.goal-of (φ Ψ)) ⊆m execute-parallel-plan
    (strips-problem.initial-of (φ Ψ)) (φP Ψ ψ)
proof −
  let ?vs = variables-of Ψ
    and ?ops = operators-of Ψ
    and ?I = initial-of Ψ
    and ?G = goal-of Ψ
  let ?Π = φ Ψ
  let ?I' = strips-problem.initial-of ?Π
    and ?G' = strips-problem.goal-of ?Π
  have dom ?I ⊆ set ?vs
    using is-valid-problem-sas-plus-then(3) assms(1)
    by auto
  moreover have ∀ v∈dom ?I. the (?I v) ∈ R+ Ψ v
```

```
    using is-valid-problem-sas-plus-then(4) assms(1) calculation
    by auto
  moreover have dom ?G ⊆ set ((Ψ)_{V+})
    using is-valid-problem-sas-plus-then(5) assms(1)
    by auto
  moreover have ∀ v ∈ dom ?G. the (?G v) ∈ R_+ Ψ v
    using is-valid-problem-sas-plus-then(6) assms(1)
    by auto
  moreover have ∀ ops ∈ set ψ. ∀ op ∈ set ops. op ∈ set ?ops
    using is-parallel-solution-for-problem-plan-operator-set[OF assms(2)]
    by fastforce
  moreover have ?G ⊆_m execute-parallel-plan-sas-plus ?I ψ
    using assms(2)
    unfolding is-parallel-solution-for-problem-def
    by simp

  ultimately show ?thesis
    using strips-equivalent-to-sas-plus-i-a[OF assms(1), of ?I ?G ψ]
    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 Ψ
    and is-parallel-solution-for-problem Ψ ψ
  shows list-all (list-all (λop. ListMem op (strips-problem.operators-of (φ Ψ))))
(φ_P Ψ ψ)
proof −
  let ?ops = operators-of Ψ
  let ?Π = φ Ψ
  let ?ops' = strips-problem.operators-of ?Π
    and ?π = φ_P Ψ ψ
  have is-valid-problem-strips ?Π
    using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]
    by simp
  have nb_1: ∀ op ∈ set ?ops. (∃ op' ∈ set ?ops'. op' = (φ_O Ψ 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 ops op op'
    assume ops ∈ set ψ and op ∈ set ops
    moreover have op ∈ set ((Ψ)_{O+})
      using is-parallel-solution-for-problem-plan-operator-set[OF assms(2)]
```

```
      calculation
    by blast
  moreover obtain op' where op' ∈ set ?ops' and op' = (φ_O Ψ op)
    using nb_1 calculation(3)
    by auto
  ultimately have (φ_O Ψ op) ∈ set ?ops'
    by blast
 }
 thus ?thesis
  unfolding list-all-iff ListMem-iff Let-def
    sas-plus-problem-to-strips-problem-def
    SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
    sas-plus-parallel-plan-to-strips-parallel-plan-def
    SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def
    Let-def
  by auto
qed
```

The following lemma proves the complementary proposition to theorem **??**. Namely, given a parallel solution ψ for a SAS+ problem, the transformation to a STRIPS plan $\varphi_P\ \Psi\ \psi$ also is a solution to the corresponding STRIPS problem $\Pi \equiv \varphi\ \Psi$. In this direction, we have to show that the execution of the transformed plan reaches the goal state $G' \equiv \Pi_G$ of the corresponding STRIPS problem, i.e.

$$G' \subseteq_m \textit{execute-parallel-plan}\ I'\ \pi$$

and that all operators in the transformed plan π are operators of Π.

```
theorem
 strips-equivalent-to-sas-plus:
 assumes is-valid-problem-sas-plus Ψ
  and is-parallel-solution-for-problem Ψ ψ
 shows STRIPS-Semantics.is-parallel-solution-for-problem (φ Ψ) (φ_P Ψ ψ)
proof −
 let ?Π = φ Ψ
 let ?I' = strips-problem.initial-of ?Π
  and ?G' = strips-problem.goal-of ?Π
  and ?ops' = strips-problem.operators-of ?Π
  and ?π = φ_P Ψ ψ
 show ?thesis
  unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def
  proof (rule conjI)
   show ?G' ⊆_m execute-parallel-plan ?I' ?π
    using strips-equivalent-to-sas-plus-i[OF assms]
    by simp
  next
   show list-all (list-all (λop. ListMem op ?ops')) ?π
```

```
      using strips-equivalent-to-sas-plus-ii[OF assms].
    qed
qed

lemma embedded-serial-sas-plus-plan-operator-structure:
  assumes ops ∈ set (embed ψ)
  obtains op
  where op ∈ set ψ
    and [φ_O Ψ op. op ← ops] = [φ_O Ψ op]
proof −
  let ?ψ' = embed ψ
  {
    have ?ψ' = [[op]. op ← ψ]
      by (induction ψ; force)
    moreover obtain op where ops = [op] and op ∈ set ψ
      using assms calculation
      by fastforce
    ultimately have ∃ op ∈ set ψ. [φ_O Ψ op. op ← 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 ∈ set (φ_P Ψ (embed ψ))
  obtains op where op ∈ 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 ← ops]  | ops. ops ∈ 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' ∈ set ?ψ' and ops = [φ_O Ψ op. op ← ops']

      using assms(1) calculation
      by blast
    moreover obtain op where op ∈ set ψ and ops = [φ_O Ψ op]
      using embedded-serial-sas-plus-plan-operator-structure calculation(2, 3)
      by blast
    ultimately have ∃ op ∈ set ψ. ops = [φ_O Ψ op]
      by meson
  }
  thus ?thesis
```

using *that*..
qed

private lemma *serial-sas-plus-equivalent-to-serial-strips-ii*[*simp*]:
concat (φ_P Ψ (*embed* ψ)) = [φ_O Ψ *op*. *op* ← ψ]
proof −
let *?ψ′* = *List-Supplement.embed* ψ
have *concat* (φ_P Ψ *?ψ′*) = *map* (λ*op*. φ_O Ψ *op*) (*concat ?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
SAS-Plus-STRIPS.sasp-op-to-strips-def Let-def
map-concat
by *blast*
also have ... = *map* (λ*op*. φ_O Ψ *op*) ψ
unfolding *concat-is-inverse-of-embed*[*of* ψ]..
finally show *concat* (φ_P Ψ (*embed* ψ)) = [φ_O Ψ *op*. *op* ← ψ].
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 ψ for a SAS+ problem Ψ, the embedding theorem confirms that the embedded plan *List-Supplement.embed* ψ is an equivalent parallel solution to Ψ. By parallel plan equivalence, $\pi \equiv \varphi_P\ \Psi$ *List-Supplement.embed* ψ is a parallel solution for the corresponding STRIPS problem $\varphi\ \Psi$. Moreover, since *List-Supplement.embed* ψ is a plan consisting of singleton parallel operators, the same is true for π. Hence, the flattening lemma applies and *concat* π is a serial solution for $\varphi\ \Psi$. Since *concat* moreover can be shown to be the inverse of *List-Supplement.embed*, the term

concat π = *concat* (φ_P Ψ (*embed* ψ))

can be reduced to the intuitive form

π = [φ_O Ψ *op*. *op* ← ψ]

which concludes the proof.

theorem
serial-sas-plus-equivalent-to-serial-strips:
assumes *is-valid-problem-sas-plus* Ψ
and *SAS-Plus-Semantics.is-serial-solution-for-problem* Ψ ψ
shows *STRIPS-Semantics.is-serial-solution-for-problem* (φ Ψ) [φ_O Ψ *op*. *op* ← ψ]
proof −
let *?ψ′* = *embed* ψ

```
    and ?Π = φ Ψ
  let ?π' = φ_P 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_O Ψ 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_O Ψ op]
    using serial-sas-plus-equivalent-to-serial-strips-i[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_O⁻¹ Π op. op ← ops'] = [φ_O⁻¹ 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_O⁻¹ Π op. op ← ops'] = [φ_O⁻¹ 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_P⁻¹ Π (embed π))
  obtains op where op ∈ set π and ops = [φ_O⁻¹ Π op]
proof −
  let ?π' = embed π
```

```
  {
   have set (φ_P⁻¹ Π (embed π)) = { [φ_O⁻¹ Π op. op ← ops]  | ops. ops ∈ set ?π' }
     unfolding strips-parallel-plan-to-sas-plus-parallel-plan-def
       SAS-Plus-STRIPS.strips-parallel-plan-to-sas-plus-parallel-plan-def
       strips-op-to-sasp-def set-map
     using setcompr-eq-image
     by blast
    moreover obtain ops' where ops' ∈ set ?π' and ops = [φ_O⁻¹ Π op. op ← ops']
     using assms(1) calculation
     by blast
    moreover obtain op where op ∈ set π and ops = [φ_O⁻¹ Π op]
     using embedded-serial-strips-plan-operator-structure calculation(2, 3)
     by blast
    ultimately have ∃ op ∈ set π. ops = [φ_O⁻¹ Π op]
     by meson
  }
  thus ?thesis
    using that..
qed

private lemma serial-strips-equivalent-to-serial-sas-plus-ii[simp]:
  concat (φ_P⁻¹ Π (embed π)) = [φ_O⁻¹ Π op. op ← π]
proof −
  let ?π' = List-Supplement.embed π
  have concat (φ_P⁻¹ Π ?π') = map (λop. φ_O⁻¹ Π op) (concat ?π')
    unfolding strips-parallel-plan-to-sas-plus-parallel-plan-def
      SAS-Plus-STRIPS.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 ... = map (λop. φ_O⁻¹ Π op) π
    unfolding concat-is-inverse-of-embed[of π]..
  finally show concat (φ_P⁻¹ Π (embed π)) = [φ_O⁻¹ Π op. op ← π].
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 is-valid-problem-sas-plus Ψ
    and STRIPS-Semantics.is-serial-solution-for-problem (φ Ψ) π
  shows SAS-Plus-Semantics.is-serial-solution-for-problem Ψ [φ_O⁻¹ Ψ op. op ← π]
proof −
```

let *?π′* = *embed π*
and *?Π* = *φ Ψ*
let *?ψ′* = *φ*$_P{}^{-1}$ *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*$_O{}^{-1}$ *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*$_O{}^{-1}$ *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

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′*
:: (*'variable* × *'domain*) *strips-problem*
⇒ *nat*
⇒ (*'variable* × *'domain*) *strips-plan set*
where *bounded-solution-set-strips′* Π *k*

$\equiv \{\ \pi.\ \mathit{STRIPS\text{-}Semantics.is\text{-}serial\text{-}solution\text{-}for\text{-}problem}\ \Pi\ \pi \wedge \mathit{length}\ \pi = k\ \}$

abbreviation *bounded-solution-set-strips*
:: (*'variable* × *'domain*) *strips-problem*
⇒ *nat*
⇒ (*'variable* × *'domain*) *strips-plan set*
where *bounded-solution-set-strips* Π *N* ≡ ($\bigcup k \in \{0..N\}$. *bounded-solution-set-strips'* Π *k*)

— Show that plan transformation for all SAS Plus solutions yields a STRIPS solution for the induced STRIPS problem with same length.
We first show injectiveness of plan transformation $\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi]$ on the set of plans $P_k \equiv$ *bounded-plan-set* (*operators-of* Ψ) *k* with length bound *k*. The injectiveness of $Sol_k \equiv$ *bounded-solution-set-sas-plus* Ψ *k*—the set of solutions with length bound *k*–then follows from the subset relation $Sol_k \subseteq P_k$.
lemma *sasp-op-to-strips-injective*:
assumes $(\varphi_O\ \Psi\ op_1) = (\varphi_O\ \Psi\ op_2)$
shows $op_1 = op_2$
proof −
let $?op_1' = \varphi_O\ \Psi\ op_1$
and $?op_2' = \varphi_O\ \Psi\ op_2$
{
have *strips-operator.precondition-of* $?op_1' =$ *strips-operator.precondition-of* $?op_2'$
using *assms*
by *argo*
hence *sas-plus-operator.precondition-of* $op_1 =$ *sas-plus-operator.precondition-of* op_2
unfolding *sasp-op-to-strips-def*
SAS-Plus-STRIPS.sasp-op-to-strips-def
Let-def
by *simp*
}
moreover {
have *strips-operator.add-effects-of* $?op_1' =$ *strips-operator.add-effects-of* $?op_2'$
using *assms*
unfolding *sasp-op-to-strips-def Let-def*
by *argo*
hence *sas-plus-operator.effect-of* $op_1 =$ *sas-plus-operator.effect-of* op_2
unfolding *sasp-op-to-strips-def Let-def*
SAS-Plus-STRIPS.sasp-op-to-strips-def
by *simp*
}
ultimately show *?thesis*
by *simp*
qed

lemma *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a*:
assumes *is-valid-problem-sas-plus* Ψ

shows *inj-on* ($\lambda\psi$. [φ_O Ψ *op*. *op* $\leftarrow$ ψ]) (*bounded-plan-set* (*sas-plus-problem.operators-of* Ψ) *k*)
proof −
let *?ops* = *sas-plus-problem.operators-of* Ψ

and *?*φ_P = $\lambda\psi$. [φ_O Ψ *op*. *op* $\leftarrow$ ψ]
let *?P* = *bounded-plan-set ?ops*
{
fix ψ_1 ψ_2
assume ψ_1*-in*: ψ_1 $\in$ *?P k*
and ψ_2*-in*: ψ_2 $\in$ *?P k*
and φ_P*-of-*ψ_1*-is-*φ_P*-of-*ψ_2: (*?*φ_P ψ_1) = (*?*φ_P ψ_2)
hence ψ_1 = ψ_2
proof (*induction k arbitrary*: ψ_1 ψ_2)
case *0*
then have *length* ψ_1 = *0*
and *length* ψ_2 = *0*
using ψ_1*-in* ψ_2*-in*
unfolding *bounded-solution-set-sas-plus′-def*
by *blast+*
then show *?case*
by *blast*
next
case (*Suc k*)
moreover have *length* ψ_1 = *Suc k* **and** *length* ψ_2 = *Suc k*
using *length-Suc-conv Suc*(*2*, *3*)
unfolding *bounded-solution-set-sas-plus′-def*
by *blast+*
moreover obtain op_1 ψ_1′ **where** ψ_1 = op_1 # ψ_1′
and *set* (op_1 # ψ_1′) $\subseteq$ *set ?ops*
and *length* ψ_1′ = *k*
using *calculation*(*5*) *Suc*(*2*)
unfolding *length-Suc-conv*
by *blast*
moreover obtain op_2 ψ_2′ **where** ψ_2 = op_2 # ψ_2′
and *set* (op_2 # ψ_2′) $\subseteq$ *set ?ops*
and *length* ψ_2′ = *k*
using *calculation*(*6*) *Suc*(*3*)
unfolding *length-Suc-conv*
by *blast*
moreover have *set* ψ_1′ $\subseteq$ *set ?ops* **and** *set* ψ_2′ $\subseteq$ *set ?ops*
using *calculation*(*8*, *11*)
by *auto+*
moreover have ψ_1′ $\in$ *?P k* **and** ψ_2′ $\in$ *?P k*
using *calculation*(*9*, *12*, *13*, *14*)
by *fast+*
moreover have *?*φ_P ψ_1′ = *?*φ_P ψ_2′
using *Suc.prems*(*3*) *calculation*(*7*, *10*)
by *fastforce*

```
      moreover have ψ₁' = ψ₂'
        using Suc.IH[of ψ₁' ψ₂', OF calculation(15, 16, 17)]
        by simp
      moreover have ?φ_P ψ₁ = (φ_O Ψ op₁) # ?φ_P ψ₁'
        and ?φ_P ψ₂ = (φ_O Ψ op₂) # ?φ_P ψ₂'
        using Suc.prems(3) calculation(7, 10)
        by fastforce+
      moreover have (φ_O Ψ op₁) = (φ_O Ψ op₂)
        using Suc.prems(3) calculation(17, 19, 20)
        by simp
      moreover have op₁ = op₂
        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 Ψ
  shows inj-on (λψ. [φ_O Ψ op. op ← ψ]) (bounded-solution-set-sas-plus' Ψ k)
  proof −
    let ?ops = sas-plus-problem.operators-of Ψ
      and ?φ_P = λψ. [φ_O Ψ op. op ← ψ]
    {
      fix ψ
      assume ψ ∈ bounded-solution-set-sas-plus' Ψ k
      then have set ψ ⊆ set ?ops
        and length ψ = k
      unfolding bounded-solution-set-sas-plus'-def is-serial-solution-for-problem-def
Let-def
          list-all-iff ListMem-iff
        by fast+
      hence ψ ∈ bounded-plan-set ?ops k
        by blast
    }
    hence bounded-solution-set-sas-plus' Ψ k ⊆ bounded-plan-set ?ops k
      by blast
    moreover have inj-on ?φ_P (bounded-plan-set ?ops k)
    using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a[OF
assms(1)].
    ultimately show ?thesis
    using inj-on-subset[of ?φ_P bounded-plan-set ?ops k bounded-solution-set-sas-plus'
Ψ k]
      by fast
  qed
```

— Show that mapping plan transformation $\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi]$ over the solution set for a given SAS+ problem yields the solution set for the induced STRIPS problem.

private lemma *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c*:
assumes *is-valid-problem-sas-plus* Ψ
shows $(\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi])\ `\ (bounded\text{-}solution\text{-}set\text{-}sas\text{-}plus'\ \Psi\ k)$
$= bounded\text{-}solution\text{-}set\text{-}strips'\ (\varphi\ \Psi)\ k$
proof −
let $?\Pi = \varphi\ \Psi$
and $?\varphi_P = \lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi]$
let $?Sol_k = bounded\text{-}solution\text{-}set\text{-}sas\text{-}plus'\ \Psi\ k$
and $?Sol_k' = bounded\text{-}solution\text{-}set\text{-}strips'\ ?\Pi\ k$
{
assume $?\varphi_P\ `\ ?Sol_k \neq ?Sol_k'$
then consider $(A)\ \exists\pi \in ?\varphi_P\ `\ ?Sol_k.\ \pi \notin ?Sol_k'$
| $(B)\ \exists\pi \in ?Sol_k'.\ \pi \notin\ ?\varphi_P\ `\ ?Sol_k$
by *blast*
hence *False*
proof (*cases*)
case A
moreover obtain π **where** $\pi \in ?\varphi_P\ `\ ?Sol_k$ **and** $\pi \notin ?Sol_k'$
using *calculation*
by *blast*
moreover obtain ψ **where** $length\ \psi = k$
and *SAS-Plus-Semantics.is-serial-solution-for-problem* $\Psi\ \psi$
and $\pi = ?\varphi_P\ \psi$
using *calculation*(*2*)
unfolding *bounded-solution-set-sas-plus'-def*
by *blast*
moreover have $length\ \pi = k$ **and** *STRIPS-Semantics.is-serial-solution-for-problem* $?\Pi\ \pi$
subgoal
using *calculation*(*4*, *6*) **by** *auto*
subgoal
using *serial-sas-plus-equivalent-to-serial-strips*
assms(*1*) *calculation*(*5*) *calculation*(*6*)
by *blast*
done
moreover have $\pi \in ?Sol_k'$
unfolding *bounded-solution-set-strips'-def*
using *calculation*(*7*, *8*)
by *simp*
ultimately show *?thesis*
by *fast*
next
case B

moreover obtain π **where** $\pi \in$ *?Sol$_k$'* **and** $\pi \notin$ *?φ_P ' ?Sol$_k$*
using *calculation*
by *blast*
moreover have *STRIPS-Semantics.is-serial-solution-for-problem ?*Π π
and *length* $\pi = k$
using *calculation*(*2*)
unfolding *bounded-solution-set-strips'-def*
by *simp+*
— Construct the counter example $\psi \equiv [{\varphi_O}^{-1}$ *?*Π *op. op* $\leftarrow \pi]$ and show that $\psi \in$ *?Sol$_k$* as well as *?φ_P* $\psi = \pi$ hence $\pi \in$ *?φ_P ' ?Sol$_k$*.
moreover have *length* $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi] = k$
and *SAS-Plus-Semantics.is-serial-solution-for-problem* Ψ $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi]$
subgoal
using *calculation*(*5*)
by *simp*
subgoal
using *serial-strips-equivalent-to-serial-sas-plus*[*OF assms*(*1*)]
calculation(*4*)
by *simp*
done
moreover have $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi] \in$ *?Sol$_k$*
unfolding *bounded-solution-set-sas-plus'-def*
using *calculation*(*6*, *7*)
by *blast*

moreover {
have $\forall$ *op* $\in$ *set* π. *op* $\in$ *set* ((*?*$\Pi)_{\mathcal{O}}$)
using *calculation*(*4*)
unfolding *STRIPS-Semantics.is-serial-solution-for-problem-def list-all-iff ListMem-iff*
by *simp*
hence *?φ_P* $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi] = \pi$
proof (*induction* π)
case (*Cons op* π)
moreover have *?φ_P* $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow$ *op* # $\pi]$
$= (\varphi_O$ Ψ $({\varphi_O}^{-1}$ Ψ *op*)) # *?φ_P* $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi]$
by *simp*
moreover have *op* $\in$ *set* ((*?*$\Pi)_{\mathcal{O}}$)
using *Cons.prems*
by *simp*
moreover have $(\varphi_O$ Ψ $({\varphi_O}^{-1}$ Ψ *op*)) = *op*
using *strips-operator-inverse-is*[*OF assms*(*1*) *calculation*(*4*)]**.**
moreover have *?φ_P* $[{\varphi_O}^{-1}$ Ψ *op. op* $\leftarrow \pi] = \pi$
using *Cons.IH Cons.prems*
by *auto*
ultimately show *?case*
by *argo*
qed *simp*

```
      }
      moreover have π ∈ ?φ_P ‘ ?Sol_k
        using calculation(8, 9)
        by force
      ultimately show ?thesis
        by blast
    qed
  }
  thus ?thesis
    by blast
qed

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-d:
  assumes is-valid-problem-sas-plus Ψ
  shows card (bounded-solution-set-sas-plus' Ψ k) ≤ card (bounded-solution-set-strips'
(φ Ψ) k)
  proof −
    let ?Π = φ Ψ
      and ?φ_P = λψ. [φ_O Ψ op. op ← ψ]
    let ?Sol_k = bounded-solution-set-sas-plus' Ψ k
      and ?Sol_k' = bounded-solution-set-strips' ?Π k
    have card (?φ_P ‘ ?Sol_k) = card (?Sol_k)
    using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b[OF
assms(1)]
      card-image
     by blast
    moreover have ?φ_P ‘ ?Sol_k = ?Sol_k'
    using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c[OF
assms(1)].
    ultimately show ?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 { π. set π ⊆ set ops ∧ length π = k }
  proof (induction k)
    case (Suc k)
    let ?P = { π. set π ⊆ set ops ∧ length π = k }
      and ?P' = { π. set π ⊆ set ops ∧ length π = Suc k }
    let ?P'' = (⋃ op ∈ set ops. (⋃ π ∈ ?P. { op # π }))
    {
      have ∀ op π. finite { op # π }
        by simp
      then have ∀ op. finite (⋃ π ∈ ?P. { op # π })
        using finite-UN[of ?P] Suc
        by blast
      hence finite ?P''
        using finite-UN[of set ops]
```

```
      by blast
  }
  moreover {
   {
    fix π
    assume π ∈ ?P'
    moreover have set π ⊆ set ops
     and length π = Suc k
     using calculation
     by simp+
    moreover obtain op π' where π = op # π'
     using calculation (3)
     unfolding length-Suc-conv
     by fast
    moreover have set π' ⊆ set ops and op ∈ set ops
     using calculation(2, 4)
     by simp+
    moreover have length π' = k
     using calculation(3, 4)
     by auto
    moreover have π' ∈ ?P
     using calculation(5, 7)
     by blast
    ultimately have π ∈ ?P''
     by blast
   }
   hence ?P' ⊆ ?P''
    by blast
  }
  ultimately show ?case
   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 Ψ
 shows finite (bounded-solution-set-sas-plus' Ψ k)
proof −
 let ?Ops = set ((Ψ)_{O+})
 let ?Sol_k = bounded-solution-set-sas-plus' Ψ k
  and ?P_k = { π. set π ⊆ ?Ops ∧ length π = k }
 {
  fix ψ
  assume ψ ∈ ?Sol_k
  then have length ψ = k and set ψ ⊆ ?Ops
   unfolding bounded-solution-set-sas-plus'-def
    SAS-Plus-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff List-
```

Mem-iff
    by *fastforce+*
  hence $\psi \in ?P_k$
    by *blast*
 }
 then have $?Sol_k \subseteq ?P_k$
  by *force*
 thus *?thesis*
  using *bounded-plan-set-finite rev-finite-subset*[*of* $?P_k$ $?Sol_k$]
  by *auto*
qed

— The set of fixed length STRIPS solutions are subsets of the set of plans with fixed length and therefore also finite.
private lemma *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows *finite* (*bounded-solution-set-strips'* (φ Ψ) k)
proof −
 let $?\Pi = \varphi\ \Psi$
 let $?Ops = set\ ((?\Pi)_{\mathcal{O}})$
 let $?Sol_k$ = *bounded-solution-set-strips'* $?\Pi$ k
   and $?P_k = \{\ \pi.\ set\ \pi \subseteq ?Ops \wedge length\ \pi = k\ \}$
 {
   fix π
   assume $\pi \in ?Sol_k$
   then have *length* $\pi = k$ **and** *set* $\pi \subseteq ?Ops$
     unfolding *bounded-solution-set-strips'-def*
       STRIPS-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff List-Mem-iff
     by *fastforce+*
   hence $\pi \in ?P_k$
     by *blast*
 }
 then have $?Sol_k \subseteq ?P_k$
  by *force*
 thus *?thesis*
  using *bounded-plan-set-finite rev-finite-subset*[*of* $?P_k$ $?Sol_k$]
  unfolding *state-to-strips-state-def*
    SAS-Plus-STRIPS.state-to-strips-state-def operators-of-def
  by *blast*
qed

With the results on the equivalence of SAS+ and STRIPS solutions, we can now show that given problems in both formalisms, the solution sets have the same size. This is the property required by the definition of planning formalism equivalence presented earlier in theorem **??** (**??**) and thus end up with the desired equivalence result.

The proof uses the finiteness and disjunctiveness of the solution sets for either problem to be able to equivalently transform the set cardinality over

the union of sets of solutions with bounded lengths into a sum over the cardinality of the sets of solutions with bounded length. Moreover, since we know that for each SAS+ solution with a given length an equivalent STRIPS solution exists in the solution set of the transformed problem with the same length, both sets must have the same cardinality.

Hence the cardinality of the SAS+ solution set over all lengths up to a given upper bound N has the same size as the solution set of the corresponding STRIPS problem over all length up to a given upper bound N.

theorem
assumes *is-valid-problem-sas-plus* Ψ
shows *card* (*bounded-solution-set-sas-plus* Ψ N)
= *card* (*bounded-solution-set-strips* (φ Ψ) N)
proof −
let *?*$\Pi = \varphi\ \Psi$
and *?R* = {*0*..*N*}
— Due to the disjoint nature of the bounded solution sets for fixed plan length for different lengths, we can sum the individual set cardinality to obtain the cardinality of the overall SAS+ resp. STRIPS solution sets.
have *finite-R*: *finite ?R*
by *simp*
moreover {
have $\forall\, k \in$ *?R*. *finite* (*bounded-solution-set-sas-plus'* Ψ k)
using *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a*[*OF*
assms(*1*)]..
moreover have $\forall\, j \in$ *?R*. $\forall\, k \in$ *?R*. $j \neq k$
$\longrightarrow$ *bounded-solution-set-sas-plus'* Ψ j
$\cap$ *bounded-solution-set-sas-plus'* Ψ $k = \{\}$
unfolding *bounded-solution-set-sas-plus'-def*
by *blast*

ultimately have *card* (*bounded-solution-set-sas-plus* Ψ N)
= ($\sum k \in$ *?R*. *card* (*bounded-solution-set-sas-plus'* Ψ k))
using *card-UN-disjoint*
by *blast*
}
moreover {
have $\forall\, k \in$ *?R*. *finite* (*bounded-solution-set-strips'* *?*Π k)
using *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b*[*OF*
assms(*1*)]..
moreover have $\forall\, j \in$ *?R*. $\forall\, k \in$ *?R*. $j \neq k$
$\longrightarrow$ *bounded-solution-set-strips'* *?*Π j
$\cap$ *bounded-solution-set-strips'* *?*Π $k = \{\}$
unfolding *bounded-solution-set-strips'-def*
by *blast*

ultimately have *card* (*bounded-solution-set-strips* *?*Π N)

```
      = (∑ k ∈ ?R. card (bounded-solution-set-strips' ?Π k))
      using card-UN-disjoint
      by blast
  }
  moreover {
    fix k
    have card (bounded-solution-set-sas-plus' Ψ k)
      = card ((λψ. [φ_O Ψ op. op ← ψ])
        ` bounded-solution-set-sas-plus' Ψ k)
    using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b[OF
assms]
        card-image[symmetric]
      by blast
    hence card (bounded-solution-set-sas-plus' Ψ k)
      = card (bounded-solution-set-strips' ?Π k)
    using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c[OF
assms]
      by presburger
  }
  ultimately show ?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 (⊥) to prevent warnings.
hide-const (open) Orderings.bot-class.bot
no-notation Orderings.bot-class.bot (⊥)

— Hide constant and notation for ((-⁺)) to prevent warnings.
hide-const (open) Transitive-Closure.trancl
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 $\Phi \equiv$ *encode-to-sat* Π t denote the SATPlan encoding for a STRIPS problem Π and makespan t. Let $k < t$ and $I \equiv (\Pi)_I$ be the initial state of Π, $G \equiv (\Pi)_G$ be its goal state, $\mathcal{V} \equiv (\Pi)_\mathcal{V}$ its variable set, and $\mathcal{O} \equiv (\Pi)_\mathcal{O}$ 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 —*State k n* for state variables resp. *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 *State* constructor of .

definition *encode-state-variable*
:: *nat* ⇒ *nat* ⇒ *bool option* ⇒ *sat-plan-variable formula*
where *encode-state-variable t k v* ≡ *case v of*
Some True ⇒ *Atom* (*State t k*)
| *Some False* ⇒ ¬ (*Atom* (*State t k*))

The initial state encoding (definition **??**) is a conjunction of state variable encodings $A \equiv$ *encode-state-variable 0 n b* with $n \equiv$ *index vs v* and $b \equiv I$ $v =$ *Some True* for all $v \in \mathcal{V}$. 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 $A \vee \bot$ for which it is easy to show that we can normalize to conjunctive normal form (CNF).

definition *encode-initial-state*

```
  :: 'variable strips-problem ⇒ sat-plan-variable formula (Φ_I - 99)
  where encode-initial-state Π
    ≡ let I = initial-of Π
        ; vs = variables-of Π
      in ⋀(map (λv. encode-state-variable 0 (index vs v) (I v) ∨ ⊥)
    (filter (λv. I v ≠ None) vs))

definition encode-goal-state
  :: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula (Φ_G - 99)
  where encode-goal-state Π t
    ≡ let
      vs = variables-of Π
      ; G = goal-of Π
    in ⋀(map (λv. encode-state-variable t (index vs v) (G v) ∨ ⊥)
      (filter (λv. G v ≠ None) vs))
```

Operator preconditions are encoded using activation-implies-precondition formulation as mentioned in **??**: i.e. for each operator $op \in \mathcal{O}$ and $p \in set$ (*precondition-of op*) we have to encode

$$Atom\ (Operator\ k\ (index\ ops\ op)) \longrightarrow Atom\ (State\ k\ (index\ vs\ v))$$

We use the equivalent disjunction in the formalization to simplify conversion to CNF.

```
definition encode-operator-precondition
  :: 'variable strips-problem
    ⇒ nat
    ⇒ 'variable strips-operator
    ⇒ sat-plan-variable formula
  where encode-operator-precondition Π t op ≡ let
      vs = variables-of Π
      ; ops = operators-of Π
    in ⋀(map (λv.
        ¬ (Atom (Operator t (index ops op))) ∨ Atom (State t (index vs v)))
      (precondition-of op))

definition encode-all-operator-preconditions
  :: 'variable strips-problem
    ⇒ 'variable strips-operator list
    ⇒ nat
    ⇒ sat-plan-variable formula
  where encode-all-operator-preconditions Π ops t ≡ let
      l = List.product [0..<t] ops
    in foldr (∧) (map (λ(t, op). encode-operator-precondition Π t op) l) (¬⊥)
```

Analogously to the operator precondition, add and delete effects of operators have to be implied by operator activation. That being said, we have to encode both positive and negative effects and the effect must be active at the following time point: i.e.

$Atom\ (Operator\ k\ m) \rightarrow Atom\ (State\ (Suc\ k)\ n)$

for add effects respectively

$Atom\ (Operator\ k\ m) \rightarrow \neg Atom\ (State\ (Suc\ k)\ n)$

for delete effects. We again encode the implications as their equivalent disjunctions in definition **??**.

definition *encode-operator-effect*
:: *'variable strips-problem*
⇒ *nat*
⇒ *'variable strips-operator*
⇒ *sat-plan-variable formula*
where *encode-operator-effect* Π *t op*
≡ *let*
vs = *variables-of* Π
; *ops* = *operators-of* Π
in ⋀(*map* (λ*v*.
¬(*Atom* (*Operator t* (*index ops op*)))
∨ *Atom* (*State* (*Suc t*) (*index vs v*)))
(*add-effects-of op*)
@ *map* (λ*v*.
¬(*Atom* (*Operator t* (*index ops op*)))
∨ ¬ (*Atom* (*State* (*Suc t*) (*index vs v*))))
(*delete-effects-of op*))

definition *encode-all-operator-effects*
:: *'variable strips-problem*
⇒ *'variable strips-operator list*
⇒ *nat*
⇒ *sat-plan-variable formula*
where *encode-all-operator-effects* Π *ops t*
≡ *let l* = *List.product* [*0..<t*] *ops*
in foldr (∧) (*map* (λ(*t*, *op*). *encode-operator-effect* Π *t op*) *l*) (¬⊥)

definition *encode-operators*
:: *'variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula*
where *encode-operators* Π *t*
≡ *let ops* = *operators-of* Π
in encode-all-operator-preconditions Π *ops t* ∧ *encode-all-operator-effects* Π *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 Π t v
   ≡ let vs = variables-of Π
       ; ops = operators-of Π
       ; 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 Π t v
   ≡ let vs = variables-of Π
       ; ops = operators-of Π
       ; 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 Π t
   ≡ let l = List.product [0..<t] (variables-of Π)
     in ⋀(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)
```

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 [6] 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 Π t
   ≡ encode-initial-state Π
     ∧ (encode-operators Π t
     ∧ (encode-all-frame-axioms Π t
     ∧ (encode-goal-state Π t)))
```

7.2 Decoding Function Definitions

Decoding plans from a valuation $\mathcal{A}$ 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

[6] Not shown.

time k to true—into a parallel operator (see definition **??**). [7]

definition *decode-plan'*
:: *'variable strips-problem*
⇒ *sat-plan-variable valuation*
⇒ *nat*
⇒ *'variable strips-operator list*
where *decode-plan'* Π $\mathcal{A}$ *i*
≡ *let ops* = *operators-of* Π
; *vs* = *map* (λ*op*. *Operator i* (*index ops op*)) (*remdups ops*)
in map (λ*v*. *case v of Operator - k* ⇒ *ops* ! *k*) (*filter* $\mathcal{A}$ *vs*)

— We decode maps over range *0*, ..., $t - 1$ because the last operator takes effect in t and must therefore have been applied in step $t - (1::'a)$.

definition *decode-plan*
:: *'variable strips-problem*
⇒ *sat-plan-variable valuation*
⇒ *nat*
⇒ *'variable strips-parallel-plan* (Φ^{-1} - - - *99*)
where *decode-plan* Π $\mathcal{A}$ *t* ≡ *map* (*decode-plan'* Π $\mathcal{A}$) [*0*..<*t*]

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, $\mathcal{A}$ (*State k* (*index vs v*))) for all $v \in \mathcal{V}$.

definition *decode-state-at*
:: *'variable strips-problem*
⇒ *sat-plan-variable valuation*
⇒ *nat*
⇒ *'variable strips-state* (${\Phi_S}^{-1}$ - - - *99*)
where *decode-state-at* Π $\mathcal{A}$ *k*
≡ *let*
vs = *variables-of* Π
; *state-encoding-to-assignment* = λ*v*. (*v*, $\mathcal{A}$ (*State k* (*index vs v*)))
in map-of (*map state-encoding-to-assignment vs*)

We continue by setting up the context for the proofs of soundness and completeness.

definition *encode-transitions* ::*'variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula* (Φ_T - - *99*) **where**
encode-transitions Π *t*
≡ *SAT-Plan-Base.encode-operators* Π *t* ∧
SAT-Plan-Base.encode-all-frame-axioms Π *t*

— Immediately proof the sublocale proposition for strips in order to gain access to definitions and lemmas.

[7] This is handled by function `decode_plan'` (not shown).

— Setup simp rules.
lemma [*simp*]:
encode-transitions Π *t*
= *SAT-Plan-Base.encode-operators* Π *t* ∧
SAT-Plan-Base.encode-all-frame-axioms Π *t*
unfolding *encode-problem-def encode-initial-state-def encode-transitions-def*
encode-goal-state-def decode-plan-def decode-state-at-def
by *simp+*

context
begin

lemma *encode-state-variable-is-lit-plus-if*:
assumes *is-valid-problem-strips* Π
and *v* ∈ *dom s*
shows *is-lit-plus* (*encode-state-variable k* (*index* (*strips-problem.variables-of* Π)
v) (*s v*))
proof −
have *s v* ≠ *None*
using *is-valid-problem-strips-initial-of-dom assms*(*2*)
by *blast*
then consider (*s-of-v-is-some-true*) *s v* = *Some True*
| (*s-of-v-is-some-false*) *s v* = *Some False*
by *fastforce*
thus *?thesis*
unfolding *encode-state-variable-def*
by (*cases*, *simp+*)
qed

lemma *is-cnf-encode-initial-state*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (Φ$_I$ Π)
proof −
let *?I* = (Π)$_I$
and *?vs* = *strips-problem.variables-of* Π
let *?l* = *map* (λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥)
(*filter* (λ*v*. *?I v* ≠ *None*) *?vs*)
{
fix *C*
assume *c-in-set-l*:*C* ∈ *set ?l*
have *set ?l* = (λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥) ‘
set (*filter* (λ*v*. *?I v* ≠ *None*) *?vs*)
using *set-map*[*of* λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥
filter (λ*v*. *?I v* ≠ *None*) *?vs*]
by *blast*
then have *set ?l* = (λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥) ‘
{*v* ∈ *set ?vs*. *?I v* ≠ *None*}
using *set-filter*[*of* λ*v*. *?I v* ≠ *None ?vs*]

```
    by argo
  then obtain v
    where c-is: C = encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥
    and v-in-set-vs: v ∈ set ?vs
    and I-of-v-is-not-None: ?I v ≠ None
    using c-in-set-l
    by auto

  {
    have v ∈ dom ?I
      using I-of-v-is-not-None
      by blast
    moreover have is-lit-plus (encode-state-variable 0 (index ?vs v) (?I v))
      using encode-state-variable-is-lit-plus-if[OF - calculation] assms(1)
      by blast
    moreover have is-lit-plus ⊥
      by simp
    ultimately have is-disj C
      using c-is
      by force
  }
  hence is-cnf C
    unfolding encode-state-variable-def
    using c-is
    by fastforce
 }
 thus ?thesis
  unfolding encode-initial-state-def SAT-Plan-Base.encode-initial-state-def Let-def
initial-of-def
   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 = strips-problem.variables-of Π
 let ?l = map (λv. encode-state-variable t (index ?vs v) (?G v) ∨ ⊥)
     (filter (λv. ?G v ≠ None) ?vs)
 {
   fix C
   assume C ∈ set ?l

   moreover {
     have set ?l = (λv. encode-state-variable t (index ?vs v) (?G v) ∨ ⊥)
       ‘ set (filter (λv. ?G v ≠ None) ?vs)
```

```
      unfolding set-map
      by blast
    then have set ?l = { encode-state-variable t (index ?vs v) (?G v) ∨ ⊥
      | v. v ∈ set ?vs ∧ ?G v ≠ None }
      by auto
  }
   moreover obtain v where C-is: C = encode-state-variable t (index ?vs v)
(?G v) ∨ ⊥
    and v ∈ set ?vs
    and G-of-v-is-not-None: ?G v ≠ None
    using calculation(1)
    by auto

  moreover {
    have v ∈ dom ?G
      using G-of-v-is-not-None
      by blast
    moreover have is-lit-plus (encode-state-variable t (index ?vs v) (?G v))
      using assms(1) calculation
      by (simp add: encode-state-variable-is-lit-plus-if)
    moreover have is-lit-plus ⊥
      by simp
    ultimately have is-disj C
      unfolding C-is
      by force
  }
  ultimately have is-cnf C
    by simp
 }
 thus ?thesis
  unfolding encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def
  using is-cnf-BigAnd[of ?l]
  by simp
qed

private lemma encode-operator-precondition-is-cnf:
 is-cnf (encode-operator-precondition Π k op)
proof −
 let ?vs = strips-problem.variables-of Π
  and ?ops = strips-problem.operators-of Π
 let ?l = map (λv. ¬ (Atom (Operator k (index ?ops op))) ∨ Atom (State k (index
?vs v)))
  (precondition-of op)
 {
   have set ?l = (λv. ¬(Atom (Operator k (index ?ops op))) ∨ Atom (State k
(index ?vs v)))
    ‘ set (precondition-of op)
    using set-map
    by force
```

```
    then have set ?l = { ¬(Atom (Operator k (index ?ops op))) ∨ Atom (State k
(index ?vs v))
      | v. v ∈ set (precondition-of op) }
      using setcompr-eq-image[of
        λv. ¬(Atom (Operator k (index ?ops op))) ∨ Atom (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 = ¬(Atom (Operator k (index ?ops op))) ∨ Atom (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 Π k op) (List.product [0..<t]
ops))
    = { encode-operator-precondition Π 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 Π 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 Π k op)
      ' ({0..<t} × set ops)
      using set-map set-l'-is
      by simp
   also have ... = { encode-operator-precondition Π k op | k op. (k, op) ∈ {0..<t}
× set ops}
      using setcompr-eq-image
      by fast
    finally have set ?fs  = { encode-operator-precondition Π k op
      | k op. (k, op) ∈ ({0..<t} × set ops) }
      by blast
  }
  thus ?thesis
```

```
    by blast
qed

private lemma is-cnf-encode-all-operator-preconditions:
  is-cnf (encode-all-operator-preconditions Π (strips-problem.operators-of Π) t)
proof −
  let ?l' = List.product [0..<t] (strips-problem.operators-of Π)
  let ?fs = map (λ(k, op). encode-operator-precondition Π k op) ?l'
  have ∀ f ∈ set ?fs. is-cnf f
    using encode-operator-precondition-is-cnf
    by fastforce
  thus ?thesis
    unfolding encode-all-operator-preconditions-def
    using is-cnf-foldr-and-if[of ?fs]
    by presburger
qed


private lemma set-map-or[simp]:
  set (map (λv. A v ∨ B v) vs) = { A v ∨ B v | v. v ∈ set vs }
proof −
  let ?l = map (λv. A v ∨ B v) vs
  have set ?l = (λv. A v ∨ B v) ‘ set vs
    using set-map
    by force
  thus ?thesis
    using setcompr-eq-image
    by auto
qed

private lemma encode-operator-effects-is-cnf-i:
  is-cnf (⋀(map (λv. (¬ (Atom (Operator t (index (strips-problem.operators-of Π)
op))))
   ∨ Atom (State (Suc t) (index (strips-problem.variables-of Π) v))) (add-effects-of
op)))
proof −
  let ?fs = map (λv. ¬ (Atom (Operator t (index (strips-problem.operators-of Π)
op)))
   ∨ Atom (State (Suc t) (index (strips-problem.variables-of Π) v))) (add-effects-of
op)
  {
    fix C
    assume C ∈ set ?fs
    then obtain v
      where v ∈ set (add-effects-of op)
        and C = ¬(Atom (Operator t (index (strips-problem.operators-of Π) op)))
          ∨ Atom (State (Suc t) (index (strips-problem.variables-of Π) v))
      by auto
    hence is-cnf C
```

```
    by fastforce
  }
  thus ?thesis
    using is-cnf-BigAnd
    by blast
qed

private lemma encode-operator-effects-is-cnf-ii:
  is-cnf (⋀(map (λv. ¬(Atom (Operator t (index (strips-problem.operators-of Π)
op)))
  ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of Π) v)))) (delete-effects-of
op)))
proof −
  let ?fs = map (λv. ¬(Atom (Operator t (index (strips-problem.operators-of Π)
op)))
  ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of Π) v)))) (delete-effects-of
op)
  {
    fix C
    assume C ∈ set ?fs
    then obtain v
      where v ∈ set (delete-effects-of op)
        and C = ¬(Atom (Operator t (index (strips-problem.operators-of Π) op)))
          ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of Π) v)))
      by auto
    hence is-cnf C
      by fastforce
  }
  thus ?thesis
    using is-cnf-BigAnd
    by blast
qed

private lemma encode-operator-effect-is-cnf:
  shows is-cnf (encode-operator-effect Π t op)
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
  let ?fs = map (λv. ¬(Atom (Operator t (index ?ops op)))
    ∨ Atom (State (Suc t) (index ?vs v)))
    (add-effects-of op)
    and ?fs′ = map (λv. ¬(Atom (Operator t (index ?ops op)))
      ∨ ¬(Atom (State (Suc t) (index ?vs 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-i[of t Π op]
      encode-operator-effects-is-cnf-ii[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 ?vs = 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
  by force
 thus ?thesis
  unfolding encode-all-operator-effects-def
  using is-cnf-foldr-and-if[of ?fs]
  by presburger
qed

lemma encode-operators-is-cnf:
 assumes is-valid-problem-strips Π
 shows is-cnf (encode-operators Π t)
```

unfolding *encode-operators-def*
using *is-cnf-encode-all-operator-preconditions*[*of* Π *t*]
encode-all-operator-effects-is-cnf[*OF assms*, *of t*]
is-cnf.simps(*1*)[*of encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*
encode-all-operator-effects Π (*strips-problem.operators-of* Π) *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* (λ*op*. *Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
(*filter* (λ*op*. *ListMem v vs*) (*strips-problem.operators-of* Π)))
= { *Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))
| *op*. *op* ∈ *set* (*strips-problem.operators-of* Π) ∧ *v* ∈ *set vs* }
proof −
let *?ops* = *strips-problem.operators-of* Π
{
have *set* (*filter* (λ*op*. *ListMem v vs*) *?ops*)
= { *op* ∈ *set ?ops*. *ListMem v vs* }
using *set-filter*
by *force*
then have *set* (*filter* (λ*op*. *ListMem v vs*) *?ops*)
= { *op*. *op* ∈ *set ?ops* ∧ *v* ∈ *set vs* }
using *ListMem-iff*[*of v*]
by *blast*
}
then have *set* (*map* (λ*op*. *Atom* (*Operator t* (*index ?ops op*)))
(*filter* (λ*op*. *ListMem v vs*) *?ops*))
= (λ*op*. *Atom* (*Operator t* (*index ?ops op*))) ‘ { *op* ∈ *set ?ops*. *v* ∈ *set vs* }
using *set-map*[*of* λ*op*. *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
qed

lemma is-cnf-encode-positive-transition-frame-axiom:
  shows is-cnf (encode-positive-transition-frame-axiom Π t v)
proof −
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
  let ?adding = filter (λop. ListMem v (add-effects-of op)) ?ops
  let ?fs = map (λop. Atom (Operator t (index ?ops op))) ?adding
    and ?A = Atom (State t (index ?vs v))
```

```
    and ?B = ¬(Atom (State (Suc t) (index ?vs v)))
  {
    fix f
    assume f ∈ set ?fs

    then obtain op
      where op ∈ set ?ops
        and v ∈ set (add-effects-of op)
        and f = Atom (Operator t (index ?ops op))
      using set-map-to-operator-atom[of t Π v]
      by fastforce
    hence is-lit-plus f
      by simp
  } note nb = this
  {
    have is-disj ⋁?fs
      using is-disj-big-or-if nb
      by blast
    then have is-disj (?B ∨ ⋁?fs)
      by force
    then have is-disj (?A ∨ (?B ∨ ⋁?fs))
      by fastforce
    hence is-cnf (?A ∨ (?B ∨ ⋁?fs))
      by fastforce
  }
  thus ?thesis
    unfolding encode-positive-transition-frame-axiom-def
    by meson
qed

private lemma encode-all-frame-axioms-set[simp]:
  set (map (λ(k, v). encode-negative-transition-frame-axiom Π k v)
        (List.product [0..<t] (strips-problem.variables-of Π))
      @ (map (λ(k, v). encode-positive-transition-frame-axiom Π k v)
        (List.product [0..<t] (strips-problem.variables-of Π))))
    = { encode-negative-transition-frame-axiom Π k v
        | k v. (k, v) ∈ ({0..<t} × set (strips-problem.variables-of Π)) }
      ∪ { encode-positive-transition-frame-axiom Π k v
        | k v. (k, v) ∈ ({0..<t} × set (strips-problem.variables-of Π)) }
proof −
  let ?l = List.product [0..<t] (strips-problem.variables-of Π)
  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 Π
  have set-l-is: set ?l = {0..<t} × set ?vs
    by simp
  have set ?fs = ?A ∪ ?B
```

```
    using set-append
    by force
  moreover have ?A = { encode-negative-transition-frame-axiom Π k v
    | k v. (k, v) ∈ ({0..<t} × set ?vs) }
   using set-l-is setcompr-eq-image[of λ(k, v). encode-negative-transition-frame-axiom
Π k v
        λ(k, v). (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
        λ(k, v). (k, v) ∈ ({0..<t} × set ?vs)]
    by fast
  ultimately show ?thesis
    by argo
qed


lemma encode-frame-axioms-is-cnf:
  shows is-cnf (encode-all-frame-axioms Π t)
proof −
  let ?l = List.product [0..<t] (strips-problem.variables-of Π)
    and ?vs = strips-problem.variables-of Π
  let ?A = { encode-negative-transition-frame-axiom Π k v
    | k v. (k, v) ∈ ({0..<t} × set ?vs) }
    and ?B = { encode-positive-transition-frame-axiom Π k v
    | k v. (k, v) ∈ ({0..<t} × set ?vs) }
    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-cnf f
      using is-cnf-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_I Π)
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 $\wedge$ *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 $\mathcal{A}$ $\models$ *SAT-Plan-Base.encode-problem* Π t
shows $\mathcal{A}$ $\models$ *SAT-Plan-Base.encode-initial-state* Π
and $\mathcal{A}$ $\models$ *SAT-Plan-Base.encode-goal-state* Π t
and $\mathcal{A}$ $\models$ *SAT-Plan-Base.encode-operators* Π t
and $\mathcal{A}$ $\models$ *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)
$\subseteq$ *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-goal-state* Π t)
$\subseteq$ *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-operators* Π t)
$\subseteq$ *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-all-frame-axioms* Π t)
$\subseteq$ *cnf* (*SAT-Plan-Base.encode-problem* Π t)
unfolding *SAT-Plan-Base.encode-problem-def*
SAT-Plan-Base.encode-problem-def[*of* Π t] *SAT-Plan-Base.encode-initial-state-def*[*of* Π]
SAT-Plan-Base.encode-goal-state-def[*of* Π t] *SAT-Plan-Base.encode-operators-def*
SAT-Plan-Base.encode-all-frame-axioms-def[*of* Π 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* $(\Phi_I\ \Pi) = \bigcup$ { *cnf* (*encode-state-variable 0*
(*index* (*strips-problem.variables-of* Π) *v*) (($(\Pi)_I$) *v*))
| *v*. *v* ∈ *set* (*strips-problem.variables-of* Π) ∧ (($(\Pi)_I$) *v* ≠ *None* }
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?I* = *strips-problem.initial-of* Π
let *?ls* = *map* (λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥)
(*filter* (λ*v*. *?I v* ≠ *None*) *?vs*)
{
have *cnf* ‘ *set ?ls* = *cnf* ‘ (λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥)
‘ *set* (*filter* (λ*v*. *?I v* ≠ *None*) *?vs*)
using *set-map*[*of* λ*v*. *encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥]
by *presburger*
also have ... = (λ*v*. *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*) ∨ ⊥))
‘ *set* (*filter* (λ*v*. *?I v* ≠ *None*) *?vs*)
using *image-comp*
by *blast*
also have ... = (λ*v*. *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*)))
‘ { *v* ∈ *set ?vs*. *?I v* ≠ *None* }
using *set-filter*[*of* λ*v*. *?I v* ≠ *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* ≠ *None* }
using *setcompr-eq-image*[*of* λ*v*. *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*))]
by *presburger*
}
moreover have *cnf* $(\Phi_I\ \Pi) = \bigcup$ (*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* Π
shows *cnf* $(\Phi_I\ \Pi) = (\bigcup v \in$ *set* (*strips-problem.variables-of* Π). {{
literal-formula-to-literal (*encode-state-variable 0* (*index* (*strips-problem.variables-of* Π) *v*)
(*strips-problem.initial-of* Π *v*)) }})
proof −

let *?vs* = *strips-problem.variables-of* Π
and *?I* = *strips-problem.initial-of* Π
have nb_1: { *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 nb_2 = *this*
{
have { *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*)) | *v*. *v* ∈ *set ?vs* ∧ *?I v* ≠ *None* }
= (λ*v*. *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*))) ‘ *set ?vs*
using *setcompr-eq-image*[*of* λ*v*. *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*))
λ*v*. *v* ∈ *set ?vs* ∧ *?I v* ≠ *None*] **using** nb_1
by *presburger*
hence { *cnf* (*encode-state-variable 0* (*index ?vs v*) (*?I v*)) | *v*. *v* ∈ *set ?vs* ∧ *?I v* ≠ *None* }
= (λ*v*. {{ *literal-formula-to-literal* (*encode-state-variable 0* (*index ?vs v*) (*?I v*)) }})
‘ *set ?vs*
using nb_2
by *force*
}
thus *?thesis*
using *cnf-of-encode-initial-state-set-i*
by (*smt Collect-cong*)
qed

lemma *cnf-of-encode-initial-state-set*:
assumes *is-valid-problem-strips* Π
and *v* ∈ *dom* (*strips-problem.initial-of* Π)
shows *strips-problem.initial-of* Π *v* = *Some True* ⟶ (∃!*C*. *C* ∈ *cnf* (Φ_I Π)
∧ *C* = { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^+$ })
and *strips-problem.initial-of* Π *v* = *Some False* ⟶ (∃!*C*. *C* ∈ *cnf* (Φ_I Π)
∧ *C* = { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ })

proof −
let *?I* = $(\Pi)_I$
let *?vs* = *strips-problem.variables-of* Π
let *?*Φ_I = Φ_I Π
have *nb*$_1$: *cnf* (Φ_I Π) = $\bigcup$ { *cnf* (*encode-state-variable 0* (*index ?vs v*) (*strips-problem.initial-of* Π *v*)) | *v*. *v* ∈ *set ?vs* ∧ *?I v* ≠ *None* }
using *cnf-of-encode-initial-state-set-i*
by *blast*
{
have *v* ∈ *set ?vs*
using *is-valid-problem-strips-initial-of-dom assms*(*1*, *2*)
by *blast*
hence *v* ∈ { *v*. *v* ∈ *set ?vs* ∧ *?I v* ≠ *None* }
using *assms*(*2*)
by *auto*
} note *nb*$_2$ = *this*
show *strips-problem.initial-of* Π *v* = *Some True* ⟶ (∃!*C*. *C* ∈ *cnf* (Φ_I Π) ∧ *C* = { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^+$ })
and *strips-problem.initial-of* Π *v* = *Some False* ⟶ (∃!*C*. *C* ∈ *cnf* (Φ_I Π) ∧ *C* = { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ })
proof (*auto*)
assume *i-v-is-some-true*: *strips-problem.initial-of* Π *v* = *Some True*
then have { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^+$ } ∈ *cnf* (*encode-state-variable 0* (*index* (*strips-problem.variables-of* Π) *v*) (*?I v*))
unfolding *encode-state-variable-def*
using *i-v-is-some-true*
by *auto*
thus { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^+$ } ∈ *cnf* (Φ_I Π)
using *nb*$_1$ *nb*$_2$
by *auto*
next
assume *i-v-is-some-false*: *strips-problem.initial-of* Π *v* = *Some False*
then have { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ } ∈ *cnf* (*encode-state-variable 0* (*index* (*strips-problem.variables-of* Π) *v*) (*?I v*))
unfolding *encode-state-variable-def*
using *i-v-is-some-false*
by *auto*
thus { (*State 0* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ } ∈ *cnf* (Φ_I Π)
using *nb*$_1$ *nb*$_2$
by *auto*
qed
qed

lemma *cnf-of-operator-encoding-structure*:
cnf (*encode-operators* Π *t*) = *cnf* (*encode-all-operator-preconditions* Π

```
    (strips-problem.operators-of Π) t)
  ∪ cnf (encode-all-operator-effects Π (strips-problem.operators-of Π) t)
 unfolding encode-operators-def
 using cnf.simps(5)
 by metis

corollary cnf-of-operator-precondition-encoding-subset-encoding:
 cnf (encode-all-operator-preconditions Π (strips-problem.operators-of Π) t)
  ⊆ cnf (Φ Π t)
 using cnf-of-operator-encoding-structure cnf-of-encode-problem-structure subset-trans
 unfolding encode-problem-def
 by blast


lemma cnf-foldr-and[simp]:
 cnf (foldr (∧) fs (¬⊥)) = (⋃f ∈ set fs. cnf f)
proof (induction fs)
 case (Cons f fs)
 have ih: cnf (foldr (∧) fs (¬⊥)) = (⋃f ∈ set fs. cnf f)
  using Cons.IH
  by blast
 {
  have cnf (foldr (∧) (f # fs) (¬⊥)) = cnf (f ∧ foldr (∧) fs (¬⊥))
   by simp
  also have ... = cnf f ∪ cnf (foldr (∧) fs (¬⊥))
   by force
  finally have cnf (foldr (∧) (f # fs) (¬⊥)) = cnf f ∪ (⋃f ∈ set fs. cnf f)
   using ih
   by argo
 }
 thus ?case
  by auto
qed simp


private lemma cnf-of-encode-operator-precondition[simp]:
 cnf (encode-operator-precondition Π t op) = (⋃ v ∈ set (precondition-of op).
  {{(Operator t (index (strips-problem.operators-of Π) op))⁻¹
   , (State t (index (strips-problem.variables-of Π) v))⁺}})
proof −
 let ?vs = strips-problem.variables-of Π
  and ?ops = strips-problem.operators-of Π
  and ?Φ_P = encode-operator-precondition Π t op
 let ?fs = map (λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t
(index ?vs v)))
  (precondition-of op)
  and ?A = (λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t (index
?vs v)))
   ‘ set (precondition-of op)
```

```
  have cnf (encode-operator-precondition Π t op) = cnf (⋀ ?fs)
    unfolding encode-operator-precondition-def
    by presburger
  also have ... = ⋃ (cnf ‘ set ?fs)
    using cnf-BigAnd
    by blast
  also have ... = ⋃(cnf ‘ ?A)
    using set-map[of λv. ¬ (Atom (Operator t (index ?ops op))) ∨ Atom (State t
(index ?vs v))
        precondition-of op]
    by argo
  also have ... = (⋃ v ∈ set (precondition-of op).
    cnf (¬(Atom (Operator t (index ?ops op))) ∨ Atom (State t (index ?vs v))))
    by blast

  finally show ?thesis
    by auto
qed


lemma cnf-of-encode-all-operator-preconditions-structure[simp]:
  cnf (encode-all-operator-preconditions Π (strips-problem.operators-of Π) t)
    = (⋃(t, op) ∈ ({..<t} × set (operators-of Π)).
      (⋃ v ∈ set (precondition-of op).
        {{(Operator t (index (strips-problem.operators-of Π) op))⁻¹
          , (State t (index (strips-problem.variables-of Π) v))⁺}}))
proof −
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
  let ?l = List.product [0..<t] ?ops
   and ?Φ_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)_𝒪)
      by auto
    then have ?A = (λ(t, op). encode-operator-precondition Π t op) ‘ ({0..<t} ×
set ((Π)_𝒪))
      using set-map
      by force
  } note nb = this
  have cnf ?Φ_P = cnf (foldr (∧) (map (λ(t, op). encode-operator-precondition Π
t op) ?l) (¬⊥))
    unfolding encode-all-operator-preconditions-def
    by presburger
  also have ... = (⋃f ∈ ?A. cnf f)
    by simp

  also have ... = (⋃(k, op) ∈ ({0..<t} × set ((Π)_𝒪)).
```

cnf (*encode-operator-precondition* Π *k op*))
using *nb*
by *fastforce*

finally show *?thesis*
by *fastforce*
qed

corollary *cnf-of-encode-all-operator-preconditions-contains-clause-if*:
fixes Π::*'variable STRIPS-Representation.strips-problem*
assumes *is-valid-problem-strips* (Π::*'variable STRIPS-Representation.strips-problem*)
and $k < t$
and *op* ∈ *set* ((Π)$_{\mathcal{O}}$)
and *v* ∈ *set* (*precondition-of op*)
shows { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
∈ *cnf* (*encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*)
proof −
let *?ops* = *strips-problem.operators-of* Π
and *?vs* = *strips-problem.variables-of* Π
let *?*Φ$_P$ = *encode-all-operator-preconditions* Π *?ops t*
and *?C* = { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
{
have *nb*: (*k*, *op*) ∈ {..<*t*} × *set* ((Π)$_{\mathcal{O}}$)
using *assms*(*2*, *3*)
by *blast*
moreover {
have *?C* ∈ (⋃ *v*∈*set* (*precondition-of op*).
{{(*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$,
(*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$}})
using *UN-iff*[**where** *A*=*set* (*precondition-of op*)
and *B*=λ*v*. {{(*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$,
(*State t* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$}}] *assms*(*4*)
by *blast*
hence ∃ *x*∈{..<*t*} × *set* ((Π)$_{\mathcal{O}}$).
?C ∈ (*case x of* (*k*, *op*) ⇒ ⋃ *v*∈*set* (*precondition-of op*).
{{(*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$,
(*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$}})
using *nb*
by *blast*
}
ultimately have *?C* ∈ (⋃(*t*, *op*) ∈ ({..<*t*} × *set* ((Π)$_{\mathcal{O}}$)).
(⋃ *v* ∈ *set* (*precondition-of op*).
{{ (*Operator t* (*index ?ops op*))$^{-1}$, (*State t* (*index ?vs v*))$^{+}$ }}))
by *blast*
}
thus *?thesis*
using *cnf-of-encode-all-operator-preconditions-structure*[*of* Π *t*]

by *argo*
qed

corollary *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*:
cnf (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⊆ *cnf* (Φ Π *t*)
using *cnf-of-encode-problem-structure*(*3*) *cnf-of-operator-encoding-structure*
unfolding *encode-problem-def*
by *blast*

private lemma *cnf-of-encode-operator-effect-structure*[*simp*]:
cnf (*encode-operator-effect* Π *t op*)
= (⋃ *v* ∈ *set* (*add-effects-of op*). {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }})
∪ (⋃ *v* ∈ *set* (*delete-effects-of op*).
{{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }})
proof −
let *?fs*$_1$ = *map* (λ*v*. ¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
∨ *Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*)))
(*add-effects-of op*)
and *?fs*$_2$ = *map* (λ*v*. ¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
∨ ¬ (*Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))))
(*delete-effects-of op*)
{
have *cnf* ‘ *set ?fs*$_1$ = *cnf*
‘ (λ*v*. ¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
∨ *Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))) ‘ *set* (*add-effects-of op*)
using *set-map*
by *force*
also have ... = (λ*v*. *cnf* (¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
∨ *Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))))
‘ *set* (*add-effects-of op*)
using *image-comp*
by *blast*

finally have *cnf* ‘ *set ?fs*$_1$ = (λ*v*. {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }}) ‘ *set* (*add-effects-of op*)
by *auto*
} note *nb*$_1$ = *this*
{
have *cnf* ‘ *set ?fs*$_2$ = *cnf* ‘ (λ*v*. ¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of*

Π) *op*)))
  $\vee$ $\neg$(*Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))))
    ‘ *set* (*delete-effects-of op*)
  using *set-map*
  by *force*
also have ... = (λ*v*. *cnf* ($\neg$(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
  $\vee$ $\neg$ (*Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*)))))
    ‘ *set* (*delete-effects-of op*)
  using *image-comp*
  by *blast*

finally have *cnf* ‘ *set ?fs*$_2$ = (λ*v*. {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
  , (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }})
    ‘ *set* (*delete-effects-of op*)
  by *auto*
} note *nb*$_2$ = *this*
{
have *cnf* (*encode-operator-effect* Π *t op*) = $\bigcup$(*cnf* ‘ *set* (*?fs*$_1$ @ *?fs*$_2$))
  unfolding *encode-operator-effect-def*
  using *cnf-BigAnd*[*of ?fs*$_1$ @ *?fs*$_2$]
  by *meson*
also have ... = $\bigcup$(*cnf* ‘ *set ?fs*$_1$ $\cup$ *cnf* ‘ *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 ... = $\bigcup$(*cnf* ‘ *set ?fs*$_1$) $\cup$ $\bigcup$(*cnf* ‘ *set ?fs*$_2$)
  using *Union-Un-distrib*[*of cnf* ‘ *set ?fs*$_1$ *cnf* ‘ *set ?fs*$_2$]
  by *argo*

finally have *cnf* (*encode-operator-effect* Π *t op*)
    = ($\bigcup$ *v* $\in$ *set* (*add-effects-of op*).
      {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
        , (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }})
    $\cup$ ($\bigcup$ *v* $\in$ *set* (*delete-effects-of op*).
      {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
        , (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }})
  using *nb*$_1$ *nb*$_2$
  by *argo*
}
thus *?thesis*
  by *blast*
qed

lemma *cnf-of-encode-all-operator-effects-structure*:
  cnf (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
    = ($\bigcup$(*k*, *op*) $\in$ ({*0*..<*t*} $\times$ *set* ((Π)$_{\mathcal{O}}$)).
      ($\bigcup$ *v* $\in$ *set* (*add-effects-of op*).
        {{ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$

```
        , (State (Suc k) (index (strips-problem.variables-of Π) v))⁺ }}))
    ∪ (⋃(k, op) ∈ ({0..<t} × set ((Π)_𝒪)).
      (⋃v ∈ set (delete-effects-of op).
        {{ (Operator k (index (strips-problem.operators-of Π) op))⁻¹
        , (State (Suc k) (index (strips-problem.variables-of Π) v))⁻¹ }}))
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
  let ?Φ_E = encode-all-operator-effects Π ?ops t
    and ?l = List.product [0..<t] ?ops
  let ?fs = map (λ(t, op). encode-operator-effect Π t op) ?l
  have nb: set (List.product [0..<t] ?ops) = {0..<t} × set ?ops
    by simp
  {
    have cnf ‘ set ?fs = cnf ‘ (λ(k, op). encode-operator-effect Π k op) ‘ ({0..<t}
× set ?ops)
      by force
    also have ... = (λ(k, op). cnf (encode-operator-effect Π k op)) ‘ ({0..<t} ×
set ?ops)
      using image-comp
      by fast

    finally have cnf ‘ set ?fs = (λ(k, op).
        (⋃v ∈ set (add-effects-of op).
          {{ (Operator k (index (strips-problem.operators-of Π) op))⁻¹
          , (State (Suc k) (index (strips-problem.variables-of Π) v))⁺ }})
      ∪ (⋃v ∈ set (delete-effects-of op).
          {{ (Operator k (index (strips-problem.operators-of Π) op))⁻¹
          , (State (Suc k) (index (strips-problem.variables-of Π) v))⁻¹ }}))
      ‘ ({0..<t} × set ?ops)
      using cnf-of-encode-operator-effect-structure
      by auto
  }

  thus ?thesis
    unfolding encode-all-operator-effects-def
    using cnf-BigAnd[of ?fs]
    by auto
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)
    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* = ⋃(*k*, *op*)∈{*0*..<*t*} × *set* ((Π)$_{\mathcal{O}}$).
⋃*v*∈*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*$_E$
using *cnf-of-encode-all-operator-effects-structure*[*of* Π *t*] *Un-upper1*[*of ?Add*]
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* ∈ (⋃*v*∈*set* (*add-effects-of op*).
{{ (*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{+}$}})
using *Complete-Lattices.UN-iff*[*of ?C* λ*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*} × *set* ((Π)$_{\mathcal{O}}$))
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)$_{\mathcal{O}}$)
and *v* ∈ *set* (*delete-effects-of op*)
shows { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }
∈ *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 *?Delete* = (⋃(*k*, *op*)∈{*0*..<*t*} × *set* ((Π)$_{\mathcal{O}}$).
⋃*v*∈*set* (*delete-effects-of op*).

```
    {{ (Operator k (index ?ops op))^-1, (State (Suc k) (index ?vs v))^-1 }})
  let ?C = { (Operator k (index ?ops op))^-1, (State (Suc k) (index ?vs v))^-1 }
  have ?Delete ⊆ cnf ?Φ_E
    using cnf-of-encode-all-operator-effects-structure[of Π t] Un-upper2[of ?Delete]
    by presburger
  moreover {
    have ?C ∈ (⋃ v ∈ set (delete-effects-of op).
      {{ (Operator k (index ?ops op))^-1, (State (Suc k) (index ?vs v))^-1 }})
      using assms(4)
      by blast
    moreover have (k, op) ∈ {0..<t} × set ?ops
    using assms(2, 3)
    by force

    ultimately have ?C ∈ ?Delete
      by fastforce
  }

  ultimately show ?thesis
    using subset-eq[of ?Delete cnf ?Φ_E]
    by meson
qed


private lemma cnf-of-big-or-of-literal-formulas-is[simp]:
  assumes ∀ f ∈ set fs. is-literal-formula f
  shows cnf (⋁fs) = {{ literal-formula-to-literal f | f. f ∈ set fs }}
  using assms
proof (induction fs)
  case (Cons f fs)
  {
    have is-literal-formula-f: is-literal-formula f
      using Cons.prems(1)
      by simp
    then have cnf f = {{ literal-formula-to-literal f }}
      using cnf-of-literal-formula
      by blast
  } note nb_1 = this
  {
    have ∀ f' ∈ set fs. is-literal-formula f'
      using Cons.prems
      by fastforce
    hence cnf (⋁fs) = {{ literal-formula-to-literal f | f. f ∈ set fs }}
      using Cons.IH
      by argo
  } note nb_2 = this
  {
    have cnf (⋁(f # fs)) = (λ(g, h). g ∪ h)
      ‘ ({{ literal-formula-to-literal f}}
```

```
      × {{ 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]
    by simp
  also have ... = (λop. (Operator k (index ?ops op))⁺)
      ‘ { op. op ∈ set ?ops ∧ v ∈ set (add-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 (add-effects-of
op) }
```

```
    using setcompr-eq-image[of λop. (Operator k (index ?ops op))⁺
        λop. op ∈set ?adding-operators]
    by blast

  hence cnf (⋁?fs) = {{ (Operator k (index ?ops op))⁺
    | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }}
    using cnf-of-big-or-of-literal-formulas-is[of ?fs]
      setcompr-eq-image[of literal-formula-to-literal λf. f ∈ set ?fs]
    by force
}

 then have cnf (¬(Atom (State (Suc k) (index ?vs v))) ∨ ⋁?fs)
 = {{ (State (Suc k) (index ?vs v))⁻¹  } ∪ { (Operator k (index ?ops op))⁺
  | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }}
  by force

 then have cnf ((Atom (State k (index ?vs v)) ∨ (¬(Atom (State (Suc k) (index
?vs v))) ∨ ⋁?fs)))
   = {{ (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 simp

 moreover have cnf (encode-positive-transition-frame-axiom Π k v)
   = cnf ((Atom (State k (index ?vs v)) ∨ (¬(Atom (State (Suc k) (index ?vs v)))
∨ ⋁?fs)))
   unfolding encode-positive-transition-frame-axiom-def
   by metis

 ultimately show ?thesis
   by blast
qed

private lemma cnf-of-negative-transition-frame-axiom:
  cnf (encode-negative-transition-frame-axiom Π k v)
   = {{ (State k (index (strips-problem.variables-of Π) v))⁻¹
      , (State (Suc k) (index (strips-problem.variables-of Π) v))⁺  }
     ∪ { (Operator k (index (strips-problem.operators-of Π) op))⁺
      | op. op ∈ set (strips-problem.operators-of Π) ∧ v ∈ set (delete-effects-of op)
}}
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[of λop. Atom (Operator k (index ?ops op)) ?deleting-operators]
```

by *blast*

then have *literal-formula-to-literal ‘ set ?fs*
$= (\lambda op.\ (Operator\ k\ (index\ ?ops\ op))^+)$ *‘ set ?deleting-operators*
using *image-comp*[*of literal-formula-to-literal* $\lambda op.\ Atom\ (Operator\ k\ (index\ ?ops\ op))$
set ?deleting-operators]
by *simp*
also have $\ldots = (\lambda op.\ (Operator\ k\ (index\ ?ops\ op))^+)$
‘ $\{\ op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}$
using *set-filter-op-list-mem-vs*[*of v - ?ops*]
by *auto*

finally have *literal-formula-to-literal ‘ set ?fs*
$= \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}$
using *setcompr-eq-image*[*of* $\lambda op.\ (Operator\ k\ (index\ ?ops\ op))^+$
$\lambda op.\ op \in$*set ?deleting-operators*]
by *blast*

hence *cnf* $(\bigvee ?fs) = \{\{\ (Operator\ k\ (index\ ?ops\ op))^+$
$\mid op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}\}$
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* $(Atom\ (State\ (Suc\ k)\ (index\ ?vs\ v)) \vee \bigvee ?fs)$
$= \{\{\ (State\ (Suc\ k)\ (index\ ?vs\ v))^+\ \} \cup \{\ (Operator\ k\ (index\ ?ops\ op))^+$
$\mid op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}\}$
by *force*

then have *cnf* $((\neg(Atom\ (State\ k\ (index\ ?vs\ v))) \vee (Atom\ (State\ (Suc\ k)\ (index\ ?vs\ v)) \vee \bigvee ?fs)))$
$= \{\{\ (State\ k\ (index\ ?vs\ v))^{-1}\ \}$
$\cup \{\ (State\ (Suc\ k)\ (index\ ?vs\ v))^+\ \}$
$\cup \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}\}$
by *simp*

moreover have *cnf* $(encode\text{-}negative\text{-}transition\text{-}frame\text{-}axiom\ \Pi\ k\ v)$
$= cnf\ ((\neg(Atom\ (State\ k\ (index\ ?vs\ v))) \vee (Atom\ (State\ (Suc\ k)\ (index\ ?vs\ v)) \vee \bigvee ?fs)))$
unfolding *encode-negative-transition-frame-axiom-def*
by *metis*

ultimately show *?thesis*
by *blast*
qed

lemma *cnf-of-encode-all-frame-axioms-structure*:
cnf (*encode-all-frame-axioms* Π *t*)
= ⋃(⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)).
{{{ (*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* ((Π)$_{\mathcal{O}}$) ∧ *v* ∈ *set* (*add-effects-of op*) }}})
∪ ⋃(⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)).
{{{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
∪ { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{+}$
| *op*. *op* ∈ *set* ((Π)$_{\mathcal{O}}$) ∧ *v* ∈ *set* (*delete-effects-of op*) }}})
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?ops* = *strips-problem.operators-of* Π
and *?Φ*$_F$ = *encode-all-frame-axioms* Π *t*
let *?l* = *List.product* [*0*..<*t*] *?vs*
let *?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*
{
let *?A* = { *encode-negative-transition-frame-axiom* Π *k v*
| *k v*. (*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)) }
and *?B* = { *encode-positive-transition-frame-axiom* Π *k v*
| *k v*. (*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)) }
have *set-l*: *set ?l* = {..<*t*} × *set* ((Π)$_{\mathcal{V}}$)
using *set-product*
by *force*

have *set ?fs* = *?A* ∪ *?B*
unfolding *set-append set-map*
using *encode-all-frame-axioms-set*
by *force*
then have *cnf* ‘ *set ?fs* = *cnf* ‘ *?A* ∪ *cnf* ‘ *?B*
using *image-Un*[*of cnf ?A ?B*]
by *argo*
moreover {
have *?A* = (⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)).
{ *encode-negative-transition-frame-axiom* Π *k v* })
by *blast*
then have *cnf* ‘ *?A* = (⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)).
{ *cnf* (*encode-negative-transition-frame-axiom* Π *k v*) })
by *blast*
hence *cnf* ‘ *?A* = (⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{V}}$)).
{{{ (*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*)}}})
using *cnf-of-negative-transition-frame-axiom*[*of* Π]

```
    by presburger
  }
  moreover {
    have ?B = (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      { encode-positive-transition-frame-axiom Π k v})
      by blast
    then have  cnf ‘ ?B = (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      { cnf (encode-positive-transition-frame-axiom Π k v)  })
      by blast
    hence cnf ‘ ?B = (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      {{{ (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 cnf-of-positive-transition-frame-axiom[of Π]
      by presburger
  }

  ultimately have cnf ‘ set ?fs
    = (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      {{{ (State k (index ?vs v))⁺
          , (State (Suc k) (index ?vs v))⁻¹ }
        ∪ {(Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (add-effects-of op) }}})
    ∪ (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      {{{ (State k (index ?vs v))⁻¹
          , (State (Suc k) (index ?vs v))⁺ }
        ∪ {(Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (delete-effects-of op)}}})
    unfolding set-append set-map
    by force
}
then have cnf (encode-all-frame-axioms Π t)
  = ⋃((⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      {{{ (State k (index ?vs v))⁺
          , (State (Suc k) (index ?vs v))⁻¹ }
        ∪ {(Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (add-effects-of op) }}})
    ∪ (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
      {{{ (State k (index ?vs v))⁻¹
          , (State (Suc k) (index ?vs v))⁺ }
        ∪ {(Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (delete-effects-of op)}}}))
  unfolding encode-all-frame-axioms-def Let-def
  using cnf-BigAnd[of ?fs]
  by argo
thus ?thesis
  using Union-Un-distrib[of
      (⋃(k, v) ∈ ({0..<t} × set ((Π)_𝒱)).
```

$\{\{\{ (State\ k\ (index\ ?vs\ v))^+$
$\quad , \ (State\ (Suc\ k)\ (index\ ?vs\ v))^{-1} \}$
$\quad \cup \{(Operator\ k\ (index\ ?ops\ op))^+$
$\quad\quad | \ op.\ op \in set\ ((\Pi)_{\mathcal{O}}) \wedge v \in set\ (add\text{-}effects\text{-}of\ op) \}\}\})$
$(\bigcup (k, v) \in (\{0..<t\} \times set\ ((\Pi)_{\mathcal{V}})).$
$\{\{\{ (State\ k\ (index\ ?vs\ v))^{-1}$
$\quad , (State\ (Suc\ k)\ (index\ ?vs\ v))^+ \}$
$\quad \cup \{(Operator\ k\ (index\ ?ops\ op))^+$
$\quad\quad | \ op.\ op \in set\ ((\Pi)_{\mathcal{O}}) \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\}\}\})]$
by *argo*
qed

— A technical lemma used in .
private lemma *cnf-of-encode-goal-state-set-i*:
$cnf\ ((\Phi_G\ \Pi)\ t\ ) = \bigcup (\{\ cnf\ (encode\text{-}state\text{-}variable\ t$
$(index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v)\ (((\Pi)_G)\ v))$
$| \ v.\ v \in set\ ((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq None \})$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?G* = $(\Pi)_G$
and *?*Φ_G = $(\Phi_G\ \Pi)\ t$
let *?fs* = *map* $(\lambda v.\ encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v) \vee \bot)$
(*filter* $(\lambda v.\ ?G\ v \neq None)\ ?vs$)
{
have *cnf* ‘ *set ?fs* = *cnf* ‘ $(\lambda v.\ encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v) \vee \bot)$
‘ $\{\ v \mid v.\ v \in set\ ?vs \wedge\ ?G\ v \neq None\ \}$
unfolding *set-map*
by *force*
also have ... = $(\lambda v.\ cnf\ (encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v) \vee \bot))$
‘ $\{\ v \mid v.\ v \in set\ ?vs \wedge\ ?G\ v \neq None\ \}$
using *image-comp*[*of cnf* $(\lambda v.\ encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v) \vee \bot)$
$\{\ v \mid v.\ v \in set\ ?vs \wedge\ ?G\ v \neq None\ \}$]
by *fast*
finally have *cnf* ‘ *set ?fs* = $\{\ cnf\ (encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v))$
$| \ v.\ v \in set\ ?vs \wedge\ ?G\ v \neq None\ \}$
unfolding *setcompr-eq-image*[*of* $\lambda v.\ cnf\ (encode\text{-}state\text{-}variable\ t\ (index\ ?vs\ v)\ (?G\ v) \vee \bot)$]
by *auto*
}
moreover have *cnf* $((\Phi_G\ \Pi)\ t) = \bigcup\ (cnf$ ‘ *set ?fs*)
unfolding *encode-goal-state-def SAT-Plan-Base.encode-goal-state-def Let-def*
using *cnf-BigAnd*[*of ?fs*]
by *force*
ultimately show *?thesis*
by *simp*
qed

— A simplification lemma for the above one.

corollary *cnf-of-encode-goal-state-set-ii*:
assumes *is-valid-problem-strips* Π
shows *cnf* $((\Phi_G\ \Pi)\ t) = \bigcup(\{\{\{$ *literal-formula-to-literal*
(*encode-state-variable* t (*index* (*strips-problem.variables-of* Π) v) $(((\Pi)_G)\ v)$)
$\}\}$
$\mid v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq$ *None* $\})$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?G* = $(\Pi)_G$
and *?*Φ_G = $(\Phi_G\ \Pi)\ t$
{
fix v
assume $v \in \{\ v \mid v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge$ *?G* $v \neq$ *None* $\}$
then have $v \in$ *set* $((\Pi)_{\mathcal{V}})$ **and** *G-of-v-is-not-None*: *?G* $v \neq$ *None*
by *fast+*
then consider (A) *?G* v = *Some True*
$\mid (B)$ *?G* v = *Some False*
by *fastforce*
hence *cnf* (*encode-state-variable* t (*index ?vs* v) (*?G* v))
= $\{\{$ *literal-formula-to-literal* (*encode-state-variable* t (*index ?vs* v) (*?G* v))
$\}\}$
unfolding *encode-state-variable-def*
by (*cases*, *force+*)
} note *nb* = *this*
have *cnf* *?*Φ_G = $\bigcup(\{$ *cnf* (*encode-state-variable* t (*index ?vs* v) (*?G* v))
$\mid v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge$ *?G* $v \neq$ *None* $\})$
unfolding *cnf-of-encode-goal-state-set-i*
by *blast*
also have ... = $\bigcup((\lambda v.$ *cnf* (*encode-state-variable* t (*index ?vs* v) $(((\Pi)_G)\ v)))$
‘ $\{\ v \mid v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq$ *None* $\})$
using *setcompr-eq-image*[*of*
$\lambda v.$ *cnf* (*encode-state-variable* t (*index ?vs* v) $(((\Pi)_G)\ v)$)
$\lambda v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq$ *None*]
by *presburger*
also have ... = $\bigcup((\lambda v.\ \{\{$ *literal-formula-to-literal*
(*encode-state-variable* t (*index ?vs* v) (*?G* v)) $\}\})$
‘ $\{\ v.\ v \in$ *set* $((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq$ *None* $\})$
using *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 $v \in$ *dom* $((\Pi)_G)$
shows $((\Pi)_G)$ v = *Some True* $\longrightarrow$ $(\exists!C.\ C \in$ *cnf* $((\Phi_G\ \Pi)\ t)$
$\wedge\ C = \{\ (State\ t\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^+\ \})$
and $((\Pi)_G)$ v = *Some False* $\longrightarrow$ $(\exists!C.\ C \in$ *cnf* $((\Phi_G\ \Pi)\ t)$
$\wedge\ C = \{\ (State\ t\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^{-1}\ \})$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?G* = $(\Pi)_G$
and *?*Φ_G = $(\Phi_G\ \Pi)\ t$
have *nb*$_1$: *cnf ?*Φ_G = $\bigcup$ { *cnf* (*encode-state-variable t* (*index ?vs v*)
(*?G v*)) | *v*. *v* ∈ *set* $((\Pi)_\mathcal{V})$ ∧ *?G v* ≠ *None* }
unfolding *cnf-of-encode-goal-state-set-i*
by *auto*
have *nb*$_2$: *v* ∈ { *v*. *v* ∈ *set* $((\Pi)_\mathcal{V})$ ∧ *?G v* ≠ *None* }
using *is-valid-problem-dom-of-goal-state-is assms*(*1*, *2*)
by *auto*
have *nb*$_3$: *cnf* (*encode-state-variable t* (*index* (*strips-problem.variables-of* Π) *v*)
$((\Pi)_G)$ *v*))
⊆ ($\bigcup${ *cnf* (*encode-state-variable t* (*index ?vs v*)
(*?G v*)) | *v*. *v* ∈ *set* $((\Pi)_\mathcal{V})$ ∧ *?G v* ≠ *None* })
using *UN-upper*[*OF nb*$_2$, *of* λ*v*. *cnf* (*encode-state-variable t* (*index ?vs v*) (*?G v*))] *nb*$_2$
by *blast*
show $((\Pi)_G)$ v = *Some True* $\longrightarrow$ $(\exists!C.\ C \in$ *cnf* $((\Phi_G\ \Pi)\ t)$
$\wedge\ C = \{\ (State\ t\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^+\ \})$
and $((\Pi)_G)$ v = *Some False* $\longrightarrow$ $(\exists!C.\ C \in$ *cnf* $((\Phi_G\ \Pi)\ t)$
$\wedge\ C = \{\ (State\ t\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^{-1}\ \})$
using *nb*$_3$
unfolding *nb*$_1$ *encode-state-variable-def*
by *auto*+
qed

end

We omit the proofs that the partial encoding functions produce formulas in CNF form due to their more technical nature. The following sublocale proof confirms that definition **??** encodes a valid problem Π into a formula that can be transformed to CNF (*is-cnf* (Φ Π *t*)) and that its CNF has the required form.

7.3 Soundness of the Basic SATPlan Algorithm

lemma *valuation-models-encoding-cnf-formula-equals*:
assumes *is-valid-problem-strips* Π
shows $\mathcal{A} \models$ Φ Π *t* = *cnf-semantics* $\mathcal{A}$ (*cnf* (Φ Π *t*))

proof −
let *?Φ* = Φ Π *t*
{
have *is-cnf ?Φ*
using *is-cnf-encode-problem*[*OF assms*]**.**
hence *is-nnf ?Φ*
using *is-nnf-cnf*
by *blast*
}
thus *?thesis*
using *cnf-semantics*[*of ?Φ* $\mathcal{A}$]
by *blast*
qed

corollary *valuation-models-encoding-cnf-formula-equals-corollary*:
assumes *is-valid-problem-strips* Π
shows $\mathcal{A} \models$ (Φ Π *t*)
= (∀ *C* ∈ *cnf* (Φ Π *t*). ∃ *L* ∈ *C*. *lit-semantics* $\mathcal{A}$ *L*)
using *valuation-models-encoding-cnf-formula-equals*[*OF assms*]
unfolding *cnf-semantics-def clause-semantics-def encode-problem-def*
by *presburger*

— A couple of technical lemmas about *decode-plan*.
lemma *decode-plan-length*:
assumes $\pi = \Phi^{-1}$ Π ν *t*
shows *length* π = *t*
using *assms*
unfolding *decode-plan-def SAT-Plan-Base.decode-plan-def*
by *simp*

lemma *decode-plan′-set-is*[*simp*]:
set (*decode-plan′* Π $\mathcal{A}$ *k*)
= { (*strips-problem.operators-of* Π) ! (*index* (*strips-problem.operators-of* Π) *op*)
| *op*. *op* ∈ *set* (*strips-problem.operators-of* Π)
∧ $\mathcal{A}$ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*)) }
proof −
let *?ops* = *strips-problem.operators-of* Π
let *?f* = λ*op*. *Operator k* (*index ?ops op*)
let *?vs* = *map ?f ?ops*
{
have *set* (*filter* $\mathcal{A}$ *?vs*) = *set* (*map ?f* (*filter* ($\mathcal{A}$ ∘ *?f*) *?ops*))
unfolding *filter-map*[*of* $\mathcal{A}$ λ*op*. *Operator k* (*index ?ops op*) *?ops*]**..**
hence *set* (*filter* $\mathcal{A}$ *?vs*) = (λ*op*. *Operator k* (*index ?ops op*)) ‘
{ *op* ∈ *set ?ops*. $\mathcal{A}$ (*Operator k* (*index ?ops op*)) }
unfolding *set-map set-filter*
by *simp*
}
have *set* (*decode-plan′* Π $\mathcal{A}$ *k*) = (λ*v*. *case v of Operator k i* ⇒ *?ops* ! *i*)

‘ (λop. Operator k (index ?ops op)) ‘ { op ∈ set ?ops. 𝒜 (Operator k (index ?ops op)) }
unfolding *decode-plan′-def set-map Let-def*
by *auto*
also have ... = (λop. *case Operator k (index ?ops op) of Operator k i ⇒ ?ops ! i*)
‘ { op ∈ set ?ops. 𝒜 (Operator k (index ?ops op)) }
unfolding *image-comp comp-apply*
by *argo*
also have ... = (λop. *?ops ! (index ?ops op)*)
‘ { op ∈ set ?ops. 𝒜 (Operator k (index ?ops op)) }
by *force*
finally show *?thesis*
by *blast*
qed

lemma *decode-plan-set-is*[*simp*]:
set (Φ^{-1} Π 𝒜 t) = ($\bigcup$ k ∈ {..<t}. { *decode-plan′* Π 𝒜 k })
unfolding *decode-plan-def SAT-Plan-Base.decode-plan-def set-map*
using *atLeast-upt*
by *blast*

lemma *decode-plan-step-element-then-i*:
assumes k < t
shows *set* ((Φ^{-1} Π 𝒜 t) ! k)
= { (*strips-problem.operators-of* Π) ! (*index* (*strips-problem.operators-of* Π) op)
| op. op ∈ *set* ((Π)$_{\mathcal{O}}$) ∧ 𝒜 (*Operator k* (*index* (*strips-problem.operators-of* Π) op)) }
proof −
have (Φ^{-1} Π 𝒜 t) ! k = *decode-plan′* Π 𝒜 k
unfolding *decode-plan-def SAT-Plan-Base.decode-plan-def*
using *assms*
by *simp*
thus *?thesis*
by *force*
qed

— Show that each operator *op* in the *k*-th parallel operator in a decoded parallel plan is contained within the problem's operator set and the valuation is true for the corresponding SATPlan variable.
lemma *decode-plan-step-element-then*:
fixes Π::*′a strips-problem*
assumes k < t
and op ∈ *set* ((Φ^{-1} Π 𝒜 t) ! k)
shows op ∈ *set* ((Π)$_{\mathcal{O}}$)
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 t) ! k)
proof −
  let ?ops = strips-problem.operators-of Π
    and ?π_k = (Φ⁻¹ Π 𝒜 t) ! 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 nb_1: ?π_k = 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 nb_2 = this
  {
    have inj-on ?g (set ?vs)
      unfolding inj-on-def
      by fastforce
    hence distinct (map ?g ?vs)
      using distinct-map nb_2
      by blast
  }
  thus ?thesis
    using distinct-map-filter[of ?g ?vs 𝒜]
```

unfolding nb_1 *decode-plan'-def Let-def*
by *argo*
qed

lemma *decode-state-at-valid-variable*:
fixes Π :: *'a strips-problem*
assumes $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k)\ v \neq None$
shows $v \in set\ ((\Pi)_{\mathcal{V}})$
proof −
let *?vs* = *strips-problem.variables-of* Π
let *?f* = $\lambda v.\ (v,\mathcal{A}\ (State\ k\ (index\ ?vs\ v)))$
{
have *fst ' set (map ?f ?vs)* = *fst ' (*$\lambda v.\ (v,\mathcal{A}\ (State\ k\ (index\ ?vs\ v))))$ *' set ?vs*
by *force*
also have ... = $(\lambda v.\ fst\ (v,\mathcal{A}\ (State\ k\ (index\ ?vs\ v))))$ *' set ?vs*
by *blast*
finally have *fst ' set (map ?f ?vs) = set ?vs*
by *auto*
}
moreover have $\neg v \notin$ *fst ' set (map ?f ?vs)*
using *map-of-eq-None-iff*[*of map ?f ?vs v*] *assms*
unfolding *decode-state-at-def SAT-Plan-Base.decode-state-at-def*
by *meson*
ultimately show *?thesis*
by *fastforce*
qed

— Show that there exists an equivalence between a model $\mathcal{A}$ 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 Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
and $k \leq t$
and $v \in set\ ((\Pi)_{\mathcal{V}})$
shows $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k)\ v$
= *Some* $(\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v)))$
proof −
let *?vs* = *strips-problem.variables-of* Π
let *?l* = *map* $(\lambda x.\ (x,\mathcal{A}\ (State\ k\ (index\ ?vs\ x))))$ *?vs*
have *set ?vs* $\neq \{\}$
using *assms*(*4*)
by *fastforce*
then have *map-of ?l v = Some* $(\mathcal{A}\ (State\ k\ (index\ ?vs\ v)))$
using *map-of-from-function-graph-is-some-if*[*of ?vs v*
$\lambda v.\ \mathcal{A}\ (State\ k\ (index\ ?vs\ v))$] *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* $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k) = set\ ((\Pi)_\mathcal{V})$
proof−
let *?s* $= {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$
and *?vs* = *strips-problem.variables-of* Π
have *dom ?s* = *fst* ‘ *set* (*map* ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v))))\ ?vs$)
unfolding *decode-state-at-def SAT-Plan-Base.decode-state-at-def*
using *dom-map-of-conv-image-fst*[*of* (*map* ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v))))$) *?vs*)]
by *meson*
also have ... = *fst* ‘ ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v))))$ ‘ *set* $((\Pi)_\mathcal{V})$
using *set-map*[*of* ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v))))\ ?vs$]
by *simp*
also have ... = (*fst* ∘ ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v)))))$ ‘ *set* $((\Pi)_\mathcal{V})$
using *image-comp*[*of fst* ($\lambda v.\ (v, \mathcal{A}\ (State\ k\ (index\ ?vs\ v))))$]]
by *presburger*
finally show *?thesis*
by *force*
qed

lemma *decode-state-at-initial-state*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
shows $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ 0) = (\Pi)_I$
proof −
let *?I* $= (\Pi)_I$
let *?s* $= {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ 0$
let *?vs* = *strips-problem.variables-of* Π
let *?*$\Phi = \Phi\ \Pi\ t$
let *?*$\Phi_I = \Phi_I\ \Pi$
{
have *is-cnf ?*Φ_I **and** *cnf ?*$\Phi_I \subseteq$ *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* $\mathcal{A}$ (*cnf ?*Φ_I)
using *cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem*[*OF assms*(*1*)]
assms(*2*)

```
    by blast
  hence ∀ C ∈ cnf ?Φ_I. clause-semantics 𝒜 C
    unfolding cnf-semantics-def encode-initial-state-def
    by blast
} note nb_1 = this
{

 {
   fix v
   assume v-in-dom-i: v ∈ dom ?I
   moreover {
     have v-in-variable-set: v ∈ set ((Π)_𝒱)
       using is-valid-problem-strips-initial-of-dom assms(1) v-in-dom-i
       by auto
     hence (Φ_S⁻¹ Π 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 nb_2 = 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 ∈ cnf ?Φ_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 nb_1
         unfolding clause-semantics-def
         by fastforce
       thus ?thesis
         using nb_2 v-initially-true
         by presburger
     next
       case v-initially-false

       then obtain C
         where C ∈ cnf ?Φ_I
           and c-is: C = { (State 0 (index ?vs v))⁻¹ }
         using cnf-of-encode-initial-state-set assms(1) v-in-dom-i
         by fastforce
       hence 𝒜 (State 0 (index ?vs v)) = False
         using nb_1
         unfolding clause-semantics-def
```

```
          by fastforce
        thus ?thesis
          using nb_2 v-initially-false
          by presburger
      qed
    }
    hence ?I ⊆_m ?s
      using map-le-def
      by blast
  } moreover {
    {
      fix v
      assume v-in-dom-s: v ∈ dom ?s
      then have v-in-set-vs: v ∈ set ?vs
        using decode-state-at-dom[OF assms(1)]
        by simp
      have v-in-dom-I: v ∈ dom ?I
        using is-valid-problem-strips-initial-of-dom assms(1) v-in-set-vs
        by auto
      have s-v-is: (Φ_S⁻¹ Π 𝒜 0) v = Some (𝒜 (State 0 (index ?vs v)))
       using decode-state-at-encoding-variables-equals-some-of-valuation-if assms(1,
2)
          v-in-set-vs
        by (metis le0)
      consider (s-v-is-some-true) ?s v = Some True
        | (s-v-is-some-false) ?s v = Some False
        using v-in-dom-s
        by fastforce
      hence ?s v = ?I v
        proof (cases)
          case s-v-is-some-true
          then have 𝒜-of-s-v: lit-semantics 𝒜 ((State 0 (index ?vs v))⁺)
            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 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 ∈ cnf ?Φ_I
```

```
            and C-is: C = { (State 0 (index ?vs v))⁻¹ }
          using cnf-of-encode-initial-state-set assms(1) v-in-dom-I
          by fastforce
        hence lit-semantics 𝒜 ((State 0 (index ?vs v))⁻¹)
          using nb₁
          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 ((State 0 (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 ∈ cnf ?Φ_I
              and C-is: C = { (State 0 (index ?vs v))⁺ }
            using cnf-of-encode-initial-state-set assms(1) v-in-dom-I
            by fastforce
          hence lit-semantics 𝒜 ((State 0 (index ?vs v))⁺)
            using nb₁
            unfolding clause-semantics-def
            by fast
          thus ?thesis
            using 𝒜-of-s-v
            by fastforce
        next
          case I-v-is-some-false
          thus ?thesis
            using s-v-is-some-false
            by presburger
        qed
    qed
  }
  hence ?s ⊆_m ?I
    using map-le-def
    by blast
} ultimately show ?thesis
  using map-le-antisym
  by blast
```

qed

lemma *decode-state-at-goal-state*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
shows $(\Pi)_G \subseteq_m {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ t$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?G* = $(\Pi)_G$
and *?G′* = ${\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ t$
and *?Φ* = $\Phi\ \Pi\ t$
and *?Φ*$_G$ = $(\Phi_G\ \Pi)\ t$
{
have *is-cnf ?Φ*$_G$ **and** *cnf ?Φ*$_G \subseteq$ *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* $\mathcal{A}$ (*cnf ?Φ*$_G$)
using *cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem*[*OF assms*(*1*)]
assms(*2*)
by *blast*
hence $\forall\, C \in$ *cnf ?Φ*$_G$. *clause-semantics* $\mathcal{A}\ C$
unfolding *cnf-semantics-def encode-initial-state-def*
by *blast*
} **note** nb_1 = *this*

{
fix *v*
assume $v \in$ *set* $((\Pi)_{\mathcal{V}})$
moreover have *set ?vs* $\neq \{\}$
using *calculation*(*1*)
by *fastforce*
moreover have $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ t)$
= *map-of* (*map* (λv. (v, $\mathcal{A}$ (*State t* (*index ?vs v*)))) *?vs*)
unfolding *decode-state-at-def SAT-Plan-Base.decode-state-at-def*
by *metis*

ultimately have $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ t)\ v$ = *Some* ($\mathcal{A}$ (*State t* (*index ?vs v*)))
using *map-of-from-function-graph-is-some-if*
by *fastforce*
} **note** nb_2 = *this*
{
fix *v*

assume *v-in-dom-G*: $v \in dom\ ?G$
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*: $(\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ t)\ v = Some\ (\mathcal{A}\ (State\ t\ (index\ ?vs\ v)))$
using nb_2
by *fastforce*
consider $(A)\ ?G\ v = Some\ True$
$|\ (B)\ ?G\ v = Some\ False$
using *v-in-dom-G*
by *fastforce*
hence $?G\ v = ?G'\ v$
proof (*cases*)
case *A*
{
obtain *C* **where** $C \subseteq cnf\ ?\Phi_G$ **and** $C = \{\{\ (State\ t\ (index\ ?vs\ v))^+\ \}\}$
using *cnf-of-encode-goal-state-set(1)[OF assms(1) v-in-dom-G] A*
by *auto*
then have $\{\ (State\ t\ (index\ ?vs\ v))^+\ \} \in cnf\ ?\Phi_G$
by *blast*
then have *clause-semantics* $\mathcal{A}\ \{\ (State\ t\ (index\ ?vs\ v))^+\ \}$
using nb_1
by *blast*
then have *lit-semantics* $\mathcal{A}\ ((State\ t\ (index\ ?vs\ v))^+)$
unfolding *clause-semantics-def*
by *blast*
hence $\mathcal{A}\ (State\ t\ (index\ ?vs\ v)) = True$
by *force*
}
thus *?thesis*
using *decode-state-at-is A*
by *presburger*
next
case *B*
{
obtain *C* **where** $C \subseteq cnf\ ?\Phi_G$ **and** $C = \{\{\ (State\ t\ (index\ ?vs\ v))^{-1}\ \}\}$
using *cnf-of-encode-goal-state-set(2)[OF assms(1) v-in-dom-G] B*
by *auto*
then have $\{\ (State\ t\ (index\ ?vs\ v))^{-1}\ \} \in cnf\ ?\Phi_G$
by *blast*
then have *clause-semantics* $\mathcal{A}\ \{\ (State\ t\ (index\ ?vs\ v))^{-1}\ \}$
using nb_1
by *blast*
then have *lit-semantics* $\mathcal{A}\ ((State\ t\ (index\ ?vs\ v))^{-1})$
unfolding *clause-semantics-def*
by *blast*
hence $\mathcal{A}\ (State\ t\ (index\ ?vs\ v)) = False$
by *simp*

```
      }
      thus ?thesis
        using decode-state-at-is B
        by presburger
    qed
  }
  thus ?thesis
    using map-le-def
    by blast
qed
```

— Show that the operator activation implies precondition constraints hold at every time step of the decoded plan.

```
lemma decode-state-at-preconditions:
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ Π t
    and k < t
    and op ∈ set ((Φ⁻¹ Π 𝒜 t) ! k)
    and v ∈ set (precondition-of op)
  shows 𝒜 (State k (index (strips-problem.variables-of Π) v))
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
  let ?Φ = Φ Π t
    and ?Φ_O = encode-operators Π t
    and ?Φ_P = encode-all-operator-preconditions Π ?ops t
  {
    have 𝒜 (Operator k (index ?ops op))
      and op ∈ set ((Π)_𝒪)
      using decode-plan-step-element-then[OF assms(3, 4)]
      by blast+
    moreover obtain C
      where clause-is-in-operator-encoding: C ∈ cnf ?Φ_P
        and C = { (Operator k (index ?ops op))⁻¹,
        (State k (index ?vs v))⁺ }
     using cnf-of-encode-all-operator-preconditions-contains-clause-if[OF assms(1,
3)
          calculation(2) assms(5)]
      by blast
    moreover have clause-semantics-𝒜-Φ_P: ∀ C ∈ cnf ?Φ_P. clause-semantics 𝒜
C
      using cnf-semantics-monotonous-in-cnf-subsets-if[OF assms(2)
          is-cnf-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 $\mathcal{A} \models \Phi\ \Pi\ 0$
shows *cnf* $((\Phi_G\ \Pi)\ 0) \subseteq$ *cnf* $(\Phi_I\ \Pi)$
proof −
let *?vs* = *strips-problem.variables-of* Π
and $?I = (\Pi)_I$
and $?G = (\Pi)_G$
and $?\Phi_I = \Phi_I\ \Pi$
and $?\Phi_G = (\Phi_G\ \Pi)\ 0$
and $?\Phi = \Phi\ \Pi\ 0$

— Show that the model of the encoding is also a model of the partial encodings.
have $\mathcal{A}$-*models*-Φ_I: $\mathcal{A} \models ?\Phi_I$ **and** $\mathcal{A}$-*models*-Φ_G: $\mathcal{A} \models ?\Phi_G$
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 $\mathcal{A}$—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: $C' \in$ *cnf* $?\Phi_G$
then obtain *v*
where *v-in-vs*: $v \in$ *set ?vs*
and *G-of-v-is-not-None*: $?G\ v \neq$ *None*
and C'-*is*: $C' = \{$ *literal-formula-to-literal* (*encode-state-variable 0* (*index ?vs v*)
(*?G v*)) }
using *cnf-of-encode-goal-state-set-ii*[*OF assms*(*1*)]
by *auto*
obtain *C*
where *C-in-cnf*-Φ_I: $C \in$ *cnf* $?\Phi_I$

```
      and C-is: C = { literal-formula-to-literal (encode-state-variable 0 (index ?vs
v)
          (?I v)) }
      using cnf-of-encode-initial-state-set-ii[OF assms(1)] v-in-vs
      by auto
    {
      let ?L = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?I
v))
      have { ?L } ∈ cnf ?Φ_I
        using C-in-cnf-Φ_I C-is
        by blast
      hence lit-semantics 𝒜 ?L
        using model-then-all-singleton-clauses-modelled[OF
            is-cnf-encode-initial-state[OF assms(1)]- 𝒜-models-Φ_I]
        by blast
    } note lit-semantics-𝒜-L = this
    {
      let ?L' = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?G
v))
      have { ?L' } ∈ cnf ?Φ_G
        using C'-in-cnf-Φ_G C'-is
        by blast
      hence lit-semantics 𝒜 ?L'
        using model-then-all-singleton-clauses-modelled[OF
            encode-goal-state-is-cnf[OF assms(1)]- 𝒜-models-Φ_G]
        by blast
    } note lit-semantics-𝒜-L' = this
    {
      have ?I v = ?G v
        proof (rule ccontr)
          assume contradiction: ?I v ≠ ?G v
          moreover have ?I v ≠ None
            using v-in-vs is-valid-problem-strips-initial-of-dom assms(1)
            by auto
          ultimately consider (A) ?I v = Some True ∧ ?G v = Some False
            | (B) ?I v = Some False ∧ ?G v = Some True
            using G-of-v-is-not-None
            by force
          thus False
            using lit-semantics-𝒜-L lit-semantics-𝒜-L'
            unfolding encode-state-variable-def
            by (cases, fastforce+)
        qed
    }
    hence C' ∈ cnf ?Φ_I
      using C-is C-in-cnf-Φ_I C'-is C'-in-cnf-Φ_G
      by argo
  }
  thus ?thesis
```

by *blast*
qed

— Show that the encoding secures that for every parallel operator *ops* decoded from the plan at every time step $t <$ *length pi* the following hold:

1. *ops* is applicable, and
2. the effects of *ops* are consistent.

lemma *encode-problem-parallel-correct-ii*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
and $k <$ *length* $(\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
shows *are-all-operators-applicable* $({\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k)$
$((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
and *are-all-operator-effects-consistent* $((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?ops* = *strips-problem.operators-of* Π
and $?\pi = \Phi^{-1}\ \Pi\ \mathcal{A}\ t$
and $?s = {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$
let $?\Phi = \Phi\ \Pi\ t$
and $?\Phi_E$ = *encode-all-operator-effects* Π *?ops t*
have *k-lt-t*: $k < t$
using *decode-plan-length assms*(*3*)
by *metis*
{
{
fix *op v*
assume *op-in-kth-of-decoded-plan-set*: *op* ∈ *set* $(?\pi\ !\ k)$
and *v-in-precondition-set*: *v* ∈ *set* (*precondition-of op*)
{
have $\mathcal{A}$ (*Operator k* (*index ?ops op*))
using *decode-plan-step-element-then*[*OF k-lt-t op-in-kth-of-decoded-plan-set*]
by *blast*
hence $\mathcal{A}$ (*State k* (*index ?vs v*))
using *decode-state-at-preconditions*[
OF assms(*1*, *2*) - *op-in-kth-of-decoded-plan-set v-in-precondition-set*]
k-lt-t
by *blast*
}
moreover have $k \leq t$
using *k-lt-t*
by *auto*
moreover {
have *op* ∈ *set* $((\Pi)_{\mathcal{O}})$
using *decode-plan-step-element-then*[*OF k-lt-t op-in-kth-of-decoded-plan-set*]
by *simp*
then have *v* ∈ *set* $((\Pi)_{\mathcal{V}})$

using *is-valid-problem-strips-operator-variable-sets*(*1*) *assms*(*1*)
v-in-precondition-set
by *auto*
}
ultimately have $(\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ k)\ v = \mathit{Some\ True}$
using *decode-state-at-encoding-variables-equals-some-of-valuation-if*[*OF assms*(*1*, *2*)]
by *presburger*
}
hence *are-all-operators-applicable ?s* (*?π* ! *k*)
using *are-all-operators-applicable-set*[*of ?s ?π* ! *k*]
by *blast*
} **moreover** {
{
fix $op_1\ op_2$
assume op_1*-in-k-th-of-decoded-plan*: $op_1 \in \mathit{set}\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
and op_2*-in-k-th-of-decoded-plan*: $op_2 \in \mathit{set}\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
have op_1*-in-set-ops*: $op_1 \in \mathit{set}\ ((\Pi)_{\mathcal{O}})$
and op_2*-in-set-ops*: $op_2 \in \mathit{set}\ ((\Pi)_{\mathcal{O}})$
and op_1*-active-at-k*: $\neg \mathit{lit\text{-}semantics}\ \mathcal{A}\ ((\mathit{Operator}\ k\ (\mathit{index}\ ?ops\ op_1))^{-1})$
and op_2*-active-at-k*: $\neg \mathit{lit\text{-}semantics}\ \mathcal{A}\ ((\mathit{Operator}\ k\ (\mathit{index}\ ?ops\ op_2))^{-1})$
subgoal
using *decode-plan-step-element-then*[*OF k-lt-t* op_1*-in-k-th-of-decoded-plan*]
by *simp*
subgoal
using *decode-plan-step-element-then*[*OF k-lt-t* op_2*-in-k-th-of-decoded-plan*]
by *force*
subgoal
using *decode-plan-step-element-then*[*OF k-lt-t* op_1*-in-k-th-of-decoded-plan*]
by *simp*
subgoal
using *decode-plan-step-element-then*[*OF k-lt-t* op_2*-in-k-th-of-decoded-plan*]
by *simp*
done

{
fix *v*
assume v*-in-add-effects-set-of-*op_1: $v \in \mathit{set}\ (\mathit{add\text{-}effects\text{-}of}\ op_1)$
and v*-in-delete-effects-set-of-*op_2: $v \in \mathit{set}\ (\mathit{delete\text{-}effects\text{-}of}\ op_2)$
let $?C_1 = \{(\mathit{Operator}\ k\ (\mathit{index}\ ?ops\ op_1))^{-1},$
$(\mathit{State}\ (\mathit{Suc}\ k)\ (\mathit{index}\ ?vs\ v))^{+}\}$
and $?C_2 = \{(\mathit{Operator}\ k\ (\mathit{index}\ ?ops\ op_2))^{-1},$
$(\mathit{State}\ (\mathit{Suc}\ k)\ (\mathit{index}\ ?vs\ v))^{-1}\}$
have $?C_1 \in \mathit{cnf}\ ?\Phi_E$ **and** $?C_2 \in \mathit{cnf}\ ?\Phi_E$
subgoal
using *cnf-of-operator-effect-encoding-contains-add-effect-clause-if*[*OF assms*(*1*) *k-lt-t* op_1*-in-set-ops v-in-add-effects-set-of-*op_1]
by *blast*
subgoal

using *cnf-of-operator-effect-encoding-contains-delete-effect-clause-if*[*OF* *assms*(*1*) *k-lt-t* op_2*-in-set-ops v-in-delete-effects-set-of-*op_2]
by *blast*
done
then have $?C_1 \in cnf\ ?\Phi$ **and** $?C_2 \in cnf\ ?\Phi$
using *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*
by *blast*+
then have C_1*-true*: *clause-semantics* $\mathcal{A}$ $?C_1$ **and** C_2*-true*: *clause-semantics* $\mathcal{A}$ $?C_2$
using *valuation-models-encoding-cnf-formula-equals*[*OF assms*(*1*)] *assms*(*2*)
unfolding *cnf-semantics-def*
by *blast*+
have *lit-semantics* $\mathcal{A}$ $((State\ (Suc\ k)\ (index\ ?vs\ v))^+)$
and *lit-semantics* $\mathcal{A}$ $((State\ (k + 1)\ (index\ ?vs\ v))^{-1})$
subgoal
using op_1*-active-at-k* C_1*-true*
unfolding *clause-semantics-def*
by *blast*
subgoal
using op_2*-active-at-k* C_2*-true*
unfolding *clause-semantics-def*
by *fastforce*
done
hence *False*
by *auto*
} moreover {
fix *v*
assume *v-in-delete-effects-set-of-*op_1: $v \in set\ (delete\text{-}effects\text{-}of\ op_1)$
and *v-in-add-effects-set-of-*op_2: $v \in set\ (add\text{-}effects\text{-}of\ op_2)$
let $?C_1 = \{(Operator\ k\ (index\ ?ops\ op_1))^{-1}, (State\ (Suc\ k)\ (index\ ?vs\ v))^{-1}\}$
and $?C_2 = \{(Operator\ k\ (index\ ?ops\ op_2))^{-1}, (State\ (Suc\ k)\ (index\ ?vs\ v))^{+}\}$
have $?C_1 \in cnf\ ?\Phi_E$ **and** $?C_2 \in cnf\ ?\Phi_E$
subgoal
using *cnf-of-operator-effect-encoding-contains-delete-effect-clause-if*[*OF* *assms*(*1*) *k-lt-t* op_1*-in-set-ops v-in-delete-effects-set-of-*op_1]
by *fastforce*
subgoal
using *cnf-of-operator-effect-encoding-contains-add-effect-clause-if*[*OF* *assms*(*1*) *k-lt-t* op_2*-in-set-ops v-in-add-effects-set-of-*op_2]
by *simp*
done
then have $?C_1 \in cnf\ ?\Phi$ **and** $?C_2 \in cnf\ ?\Phi$
using *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*
by *blast*+
then have C_1*-true*: *clause-semantics* $\mathcal{A}$ $?C_1$ **and** C_2*-true*: *clause-semantics* $\mathcal{A}$ $?C_2$
using *valuation-models-encoding-cnf-formula-equals*[*OF assms*(*1*)] *assms*(*2*)
unfolding *cnf-semantics-def*

```
      by blast+
    have lit-semantics 𝒜 ((State (Suc k) (index ?vs v))⁻¹)
      and lit-semantics 𝒜 ((State (k + 1) (index ?vs v))⁺)
      subgoal
        using op₁-active-at-k C₁-true
        unfolding clause-semantics-def
        by blast
      subgoal
        using op₂-active-at-k  C₂-true
        unfolding clause-semantics-def
        by fastforce
      done
    hence False
      by simp
  }
  ultimately have set (add-effects-of op₁) ∩ set (delete-effects-of op₂) = {}
    and set (delete-effects-of op₁) ∩ set (add-effects-of op₂) = {}
    by blast+
 }
 hence are-all-operator-effects-consistent (?π ! k)
  using are-all-operator-effects-consistent-set[of ?π ! k]
  by blast
}
ultimately show are-all-operators-applicable ?s (?π ! k)
 and are-all-operator-effects-consistent (?π ! k)
 by blast+
qed
```

— Show that for all operators *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 *Suc k*.

```
lemma encode-problem-parallel-correct-iii:
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ Π t
    and k < length (Φ⁻¹ Π 𝒜 t)
    and op ∈ set ((Φ⁻¹ Π 𝒜 t) ! k)
  shows v ∈ set (add-effects-of op)
    ⟶ (Φ_S⁻¹ Π 𝒜 (Suc k)) v = Some True
  and v ∈ set (delete-effects-of op)
    ⟶ (Φ_S⁻¹ Π 𝒜 (Suc k)) v = Some False
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
  let ?Φ_F = encode-all-operator-effects Π ?ops t
    and ?A = (⋃(t, op)∈{0..<t} × set ((Π)_𝒪).
      {{{ (Operator t (index ?ops op))⁻¹, (State (Suc t) (index ?vs v))⁺ }}
        | v. v ∈ set (add-effects-of op)})
    and ?B = (⋃(t, op)∈{0..<t} × set ((Π)_𝒪).
      {{{ (Operator t (index ?ops op))⁻¹,
```

```
        (State (Suc t) (index ?vs v))⁻¹ }}
      | v. v ∈ set (delete-effects-of op)})
  have k-lt-t: k < t
    using decode-plan-length assms(3)
    by metis
  have op-is-valid: op ∈ set ((Π)_𝒪)
    using decode-plan-step-element-then[OF k-lt-t assms(4)]
    by blast
  have k-op-included: (k, op) ∈ ({0..<t} × set ((Π)_𝒪))
    using k-lt-t op-is-valid
    by fastforce
  thus  v ∈ set (add-effects-of op)
    ⟶ (Φ_S⁻¹ Π 𝒜 (Suc k)) v = Some True
    and v ∈ set (delete-effects-of op)
      ⟶ (Φ_S⁻¹ Π 𝒜 (Suc k)) v = Some False
    proof (auto)
      assume v-is-add-effect: v ∈ 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))⁺}}
          ∈ {{{(Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁺}}
            | v. v ∈ 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))⁺}} ∈ ?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))⁺}}
              - {0..<t} × set ((Π)_𝒪)]
          by blast

       then have {(Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁺}
∈ ⋃ ?A
          using Union-iff[of {(Operator k (index ?ops op))⁻¹, (State (Suc k) (index
?vs v))⁺}]
          by blast

        moreover have ⋃ ?A ⊆ cnf ?Φ_F
          using cnf-of-encode-all-operator-effects-structure
          by blast
        ultimately have {(Operator k (index ?ops op))⁻¹, (State (Suc k) (index
?vs v))⁺} ∈ cnf ?Φ_F
          using in-mono[of ⋃ ?A cnf ?Φ_F]
          by presburger
      }
```

ultimately have $\mathcal{A}$ (*State* (*Suc k*) (*index ?vs v*))
using *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*
assms(*2*)[*unfolded valuation-models-encoding-cnf-formula-equals-corollary*[*OF assms*(*1*)]]
unfolding *Bex-def*
by *fastforce*
thus ($\Phi_S{}^{-1}$ Π $\mathcal{A}$ (*Suc k*)) *v* = *Some True*
using *assms*(*1*) *assms*(*2*)
decode-state-at-encoding-variables-equals-some-of-valuation-if
is-valid-problem-strips-operator-variable-sets(*2*) *k-lt-t op-is-valid subsetD*
v-is-add-effect
by *fastforce*
next
assume *v-is-delete-effect*: *v* ∈ *set* (*delete-effects-of op*)
have $\mathcal{A}$ (*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*))$^{-1}$}}
∈ {{{(*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$}}
| *v*. *v* ∈ *set* (*delete-effects-of op*)}
using *v-is-delete-effect*
by *blast*

then have {{(*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$}} ∈ *?B*
using *k-op-included cnf-of-encode-all-operator-effects-structure*
UN-iff[*of* {{(*Operator t* (*index ?ops op*))$^{-1}$, (*State* (*Suc t*) (*index ?vs v*))$^{+}$}}
- {*0*..<*t*} × *set* ((Π)$_{\mathcal{O}}$)]
by *blast*

then have {(*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$} ∈ ⋃ *?B*
using *Union-iff*[*of* {(*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$}]
by *blast*

moreover have ⋃ *?B* ⊆ *cnf* ?Φ_F
using *cnf-of-encode-all-operator-effects-structure Un-upper2*[*of* ⋃ *?B* ⋃ *?A*]
by *fast*
ultimately have {(*Operator k* (*index ?ops op*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$} ∈ *cnf* ?Φ_F
using *in-mono*[*of* ⋃ *?B cnf* ?Φ_F]
by *presburger*
}

ultimately have ¬$\mathcal{A}$ (*State* (*Suc k*) (*index ?vs v*))

using *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem valuation-models-encoding-cnf-formula-equals-corollary*[*OF assms*(*1*)] *assms*(*2*)
by *fastforce*
moreover have *Suc k* ≤ *t*
using *k-lt-t*
by *fastforce*
moreover have *v* ∈ *set*((Π)$_{\mathcal{V}}$)
using *v-is-delete-effect is-valid-problem-strips-operator-variable-sets*(*3*) *assms*(*1*) *op-is-valid*
by *auto*
ultimately show (Φ$_S$$^{-1}$ Π $\mathcal{A}$ (*Suc k*)) *v* = *Some False*
using *decode-state-at-encoding-variables-equals-some-of-valuation-if*[*OF assms*(*1*, *2*)]
by *auto*
qed
qed

— In broad strokes, this lemma shows that the operator frame axioms ensure that state is propagated—i.e. the valuation of a variable does not change inbetween time steps—, if there is no operator active which has an effect on a given variable a: i.e.

$$\mathcal{A} \vDash (\neg a_i \wedge a_{i+1}) \longrightarrow \bigvee \{op_i, k : op_i \text{ has add effect } a\}$$
$$\mathcal{A} \vDash (a_i \wedge \neg a_{i+1}) \longrightarrow \bigvee \{op_i, k : op_i \text{ has delete effect } a\}$$

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

$$\mathcal{A} \vDash \neg(\neg a_i \wedge a_{i+1}) \equiv \mathcal{A} \vDash a_i \vee \neg a_{i+1}$$
$$\mathcal{A} \vDash \neg(a_i \wedge \neg a_{i+1}) \equiv \mathcal{A} \vDash \neg a_i \vee a_{i+1}$$

hence

$$\mathcal{A} \vDash (\neg a_i \vee a_{i+1}) \wedge (a_i \vee \neg a_{i+1})$$
$$\rightsquigarrow \mathcal{A} \vDash \{\{\neg a_i, a_{i+1}\}, \{a_i, \neg a_{i+1}\}\}$$

The lemma characterizes this simplification. [8]

lemma *encode-problem-parallel-correct-iv*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and $\mathcal{A}$ ⊨ Φ Π *t*
and *k* < *t*
and *v* ∈ *set* ((Π)$_{\mathcal{V}}$)
and ¬(∃ *op* ∈ *set* ((Φ^{-1} Π $\mathcal{A}$ *t*) ! *k*).
v ∈ *set* (*add-effects-of op*) ∨ *v* ∈ *set* (*delete-effects-of op*))
shows *cnf-semantics* $\mathcal{A}$ {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$

[8] This part of the soundness proof is only treated very briefly in [3, theorem 3.1, p.1044]

, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^+$ }}
and *cnf-semantics* $\mathcal{A}$ {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^+$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }}
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?ops* = *strips-problem.operators-of* Π
let *?*Φ = Φ Π *t*
and *?*Φ_F = *encode-all-frame-axioms* Π *t*
and *?*π_k = (Φ^{-1} Π $\mathcal{A}$ *t*) ! *k*
and *?A* = $\bigcup$(*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*) }}}
and *?B* = $\bigcup$(*k*, *v*) ∈ ({*0..<t*} × *set ?vs*).
{{{ (*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*) }}}
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*) }
and *?C′* = { (*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*) }

have *k-v-included*: (*k*, *v*) ∈ ({*..<t*} × *set* ((Π)$_\mathcal{V}$))
using *assms*(*3*, *4*)
by *blast*
have *operator-encoding-subset-encoding*: *cnf ?*Φ_F ⊆ *cnf ?*Φ
using *cnf-of-encode-problem-structure*(*4*)
unfolding *encode-problem-def*
by *fast*
— Given the premise that no operator in π_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 π_k (then it doesn't have effect on *v* and therefore is not in either of the sets), or if is not, then $\mathcal{A}$ (*Operator k* (*index ?ops op*) = ⊥ by definition of *decode-plan*; moreover,
- assuming *op* is not in *set ?ops*—this is implicitely encoded as *Operator k* (*length ?ops*) and $\mathcal{A}$ (*Operator k* (*length ?ops*)) may or may not be true—, then it's not in either of the sets.

. Altogether, we have the situation that the sets only have members *Operator k* (*index ?ops op*) with $\mathcal{A}$ (*Operator k* (*index ?ops op*)) = ⊥, 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:

∀ *op*. *op* ∈ { ((*Operator k* (*index ?ops op*))$^+$)
| *op*. *op* ∈ *set ?ops* ∧ *v* ∈ *set* (*add-effects-of op*)}
⟶ ¬*lit-semantics* $\mathcal{A}$ *op*

and

$$\forall\, op.\ op \in \{\ ((Operator\ k\ (index\ ?ops\ op))^+)\\ \quad \mid op.\ op \in set\ ?ops \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\}\\ \quad \longrightarrow \neg lit\text{-}semantics\ \mathcal{A}\ op$$

Hence, the operators are irrelevant for *cnf-semantics* $\mathcal{A}$ { C } where C is a clause encoding a positive or negative transition frame axiom for a given variable v of the problem.

```
{
 let ?add = { ((Operator k (index ?ops op))⁺)
    | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }
  and ?delete = { ((Operator k (index ?ops op))⁺)
    | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) }
 {
  fix op
  assume operator-encoding-in-add: (Operator k (index ?ops op))⁺ ∈ ?add
  hence ¬lit-semantics 𝒜 ((Operator k (index ?ops op))⁺)
   proof (cases op ∈ set ?π_k)
    case True
    then have v ∉ set (add-effects-of op)
     using assms(5)
     by simp
    then have (Operator k (index ?ops op))⁺ ∉ ?add
     by fastforce
    thus ?thesis
     using operator-encoding-in-add
     by blast
   next
    case False
    then show ?thesis
     proof (cases op ∈ set ?ops)
      case True
      {
       let ?A = { ?ops ! index ?ops op |op.
          op ∈ set ((Π)_𝒪) ∧ 𝒜 (Operator k (index ?ops op))}
       assume lit-semantics 𝒜 ((Operator k (index ?ops op))⁺)
       moreover have operator-active-at-k: 𝒜 (Operator k (index ?ops op))
         using calculation
         by auto
       moreover have op ∈ set ((Π)_𝒪)
         using True
         by force
       moreover have (?ops ! index ?ops op) ∈ ?A
         using calculation(2, 3)
         by blast
       ultimately have op ∈ set ?π_k
         using decode-plan-step-element-then-i[OF assms(3)]
         by auto
```

```
          hence False
            using False
            by blast
        }
        thus ?thesis
          by blast
      next
        case False
        then have op ∉ {op ∈ set ?ops. v ∈ set (add-effects-of op)}
          by blast
        moreover have ?add =
          (λop. (Operator k (index ?ops op))⁺)
            ` { op ∈ set ?ops. v ∈ set (add-effects-of op) }
          using setcompr-eq-image[of λop. (Operator k (index ?ops op))⁺
              λop. op ∈ set ?ops ∧ v ∈ set (add-effects-of op)]
          by blast

        ultimately have (Operator k (index ?ops op))⁺ ∉ ?add
          by force
        thus ?thesis using operator-encoding-in-add
          by blast
      qed
    qed
} moreover {
  fix op
 assume operator-encoding-in-delete: ((Operator k (index ?ops op))⁺) ∈ ?delete
  hence ¬lit-semantics 𝒜 ((Operator k (index ?ops op))⁺)
    proof (cases op ∈ set ?π_k)
      case True
      then have v ∉ set (delete-effects-of op)
        using assms(5)
        by simp
      then have (Operator k (index ?ops op))⁺ ∉ ?delete
        by fastforce
      thus ?thesis
        using operator-encoding-in-delete
        by blast
    next
      case False
      then show ?thesis
        proof (cases op ∈ set ?ops)
          case True
          {
            let ?A = { ?ops ! index ?ops op |op.
                op ∈ set ((Π)_𝒪) ∧ 𝒜 (Operator k (index ?ops op))}
            assume lit-semantics 𝒜 ((Operator k (index ?ops op))⁺)
           moreover have operator-active-at-k: 𝒜 (Operator k (index ?ops op))
              using calculation
              by auto
```

```
        moreover have op ∈ set ((Π)_𝒪)
          using True
          by force
        moreover have (?ops ! index ?ops op) ∈ ?A
          using calculation(2, 3)
          by blast
        ultimately have op ∈ set ?π_k
          using decode-plan-step-element-then-i[OF assms(3)]
          by auto
        hence False
          using False
          by blast
      }
      thus ?thesis
        by blast
    next
      case False
      then have op ∉ { op ∈ set ?ops. v ∈ set (delete-effects-of op) }
        by blast
      moreover have ?delete =
        (λop. (Operator k (index ?ops op))⁺)
          ` { op ∈ set ?ops. v ∈ set (delete-effects-of op) }
        using setcompr-eq-image[of λop. (Operator k (index ?ops op))⁺
            λop. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op)]
        by blast

      ultimately have (Operator k (index ?ops op))⁺ ∉ ?delete
        by force
      thus ?thesis using operator-encoding-in-delete
        by blast
    qed
  qed
 }
 ultimately have ∀ op. op ∈ ?add ⟶ ¬lit-semantics 𝒜 op
 and ∀ op. op ∈ ?delete ⟶ ¬lit-semantics 𝒜 op
  by blast+
} note nb = this
{
 let ?Ops = { (Operator k (index ?ops op))⁺
  | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of op) }
 have ?Ops ⊆ ?C
  by blast
 moreover have ?C − ?Ops = { (State k (index ?vs v))⁺ , (State (Suc k)
(index ?vs v))⁻¹ }
  by fast
 moreover have ∀ L ∈ ?Ops. ¬ lit-semantics 𝒜 L
  using nb(1)
  by blast
```

ultimately have *clause-semantics* $\mathcal{A}$ *?C*
= *clause-semantics* $\mathcal{A}$ { (*State k* (*index ?vs v*))$^+$, (*State* (*Suc k*) (*index ?vs v*))$^{-1}$ }
using *lit-semantics-reducible-to-subset-if*[*of ?Ops ?C*]
by *presburger*
} moreover {
let *?Ops′* = { (*Operator k* (*index ?ops op*))$^+$
| *op*. *op* ∈ *set ?ops* ∧ *v* ∈ *set* (*delete-effects-of op*) }
have *?Ops′* ⊆ *?C′*
by *blast*
moreover have *?C′* − *?Ops′* = { (*State k* (*index ?vs v*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^+$ }
by *fast*
moreover have ∀ *L* ∈ *?Ops′*. ¬ *lit-semantics* $\mathcal{A}$ *L*
using *nb*(*2*)
by *blast*

ultimately have *clause-semantics* $\mathcal{A}$ *?C′*
= *clause-semantics* $\mathcal{A}$ { (*State k* (*index ?vs v*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^+$ }
using *lit-semantics-reducible-to-subset-if*[*of ?Ops′ ?C′*]
by *presburger*
} moreover {
have *cnf-semantics-*$\mathcal{A}$*-*Φ*:cnf-semantics* $\mathcal{A}$ (*cnf ?*Φ)
using *valuation-models-encoding-cnf-formula-equals*[*OF assms*(*1*)] *assms*(*2*)
by *blast*
have *k-v-included*: (*k*, *v*) ∈ ({..<*t*} × *set* ((Π)$_\mathcal{V}$))
using *assms*(*3*, *4*)
by *blast*

have *c-in-un-a*: *?C* ∈ ⋃ *?A* **and** *c′-in-un-b*: *?C′* ∈ ⋃ *?B*
using *k-v-included*
by *force*+

then have *?C* ∈ *cnf ?*Φ$_F$ **and** *?C′* ∈ *cnf ?*Φ$_F$
subgoal
using *cnf-of-encode-all-frame-axioms-structure UnI1*[*of ?C* ⋃ *?A* ⋃ *?B*] *c-in-un-a*
by *metis*
subgoal
using *cnf-of-encode-all-frame-axioms-structure UnI2*[*of ?C′* ⋃ *?B* ⋃ *?A*] *c′-in-un-b*
by *metis*
done
then have { *?C* } ⊆ *cnf ?*Φ$_F$ **and** *c′-subset-frame-axiom-encoding*: { *?C′* } ⊆ *cnf ?*Φ$_F$
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* $\mathcal{A}$ { *?C* } **and** *cnf-semantics* $\mathcal{A}$ { *?C′* }
using *cnf-semantics-*$\mathcal{A}$*-*Φ *model-for-cnf-is-model-of-all-subsets*
by *fastforce+*
}
ultimately show *cnf-semantics* $\mathcal{A}$ {{ (*State k* (*index ?vs v*))$^{-1}$, (*State* (*Suc k*) (*index ?vs v*))$^{+}$ }}
and *cnf-semantics* $\mathcal{A}$ {{ (*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 $\mathcal{A} \models \Phi\ \Pi\ t$
and $k <$ *length* ($\Phi^{-1}\ \Pi\ \mathcal{A}\ t$)
shows (${\Phi_S}^{-1}\ \Pi\ \mathcal{A}$ (*Suc k*)) = *execute-parallel-operator* (${\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$) (($\Phi^{-1}\ \Pi\ \mathcal{A}\ t$) ! *k*)
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?ops* = *strips-problem.operators-of* Π
and *?*π = $\Phi^{-1}\ \Pi\ \mathcal{A}\ t$
and *?*s_k = ${\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$
and *?*$s_k{}'$ = ${\Phi_S}^{-1}\ \Pi\ \mathcal{A}$ (*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*: $k \leq t$ **and** *suc-k-lte-t*: *Suc k* $\leq t$
using *decode-plan-length*[*of ?*π Π $\mathcal{A}$ *t*] *assms*(*3*)
by (*argo*, *fastforce+*)
then have *operator-preconditions-hold*:
are-all-operators-applicable ?s_k *?*π_k $\wedge$ *are-all-operator-effects-consistent ?*π_k
using *encode-problem-parallel-correct-ii*[*OF assms*(*1*, *2*, *3*)]
by *blast*
— We show the goal in classical fashion by proving that

${\Phi_S}^{-1}\ \Pi\ \mathcal{A}$ (*Suc k*) *v*
= *execute-parallel-operator* (${\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$)
(($\Phi^{-1}\ \Pi\ \mathcal{A}\ t$) ! *k*) *v*

—i.e. the state decoded at time $k + 1$ is equivalent to the state obtained by executing the parallel operator ($\Phi^{-1}\ \Pi\ \mathcal{A}\ t$) ! k on the previous state ${\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k$—for all variables v given $k < t$, a model $\mathcal{A}$, and makespan t.
moreover {

```
  {
    fix v
    assume v-in-dom-s_k':v ∈ dom ?s_k'
    then have s_k'-not-none: ?s_k' v ≠ None
      by blast
    hence ?s_k' v = ?t_k' v
        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-π_k: op ∈ set ?π_k
        and v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)
        by blast
      then consider (v-is-add-effect) v ∈ set (add-effects-of op)
        | (v-is-delete-effect) v ∈ set (delete-effects-of op)
        by blast
      then show ?thesis
        proof (cases)
          case v-is-add-effect
          then have ?s_k' v = Some True
                using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)
op-in-π_k]
            v-is-add-effect
          by blast
         moreover have are-all-operators-applicable (Φ_S⁻¹ Π 𝒜 k) ((Φ⁻¹ Π 𝒜
t) ! k)
          and are-all-operator-effects-consistent ((Φ⁻¹ Π 𝒜 t) ! k)
          using operator-preconditions-hold v-is-add-effect
          by blast+
         moreover have ?t_k' v = Some True
          using execute-parallel-operator-positive-effect-if[of
             Φ_S⁻¹ Π 𝒜 k (Φ⁻¹ Π 𝒜 t) ! k] op-in-π_k
            v-is-add-effect calculation(2, 3)
          by blast
         ultimately show ?thesis
          by argo
        next
          case v-is-delete-effect
          then have ?s_k' v = Some False
                using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)
op-in-π_k]
            v-is-delete-effect
          by blast
         moreover have are-all-operators-applicable (Φ_S⁻¹ Π 𝒜 k) ((Φ⁻¹ Π 𝒜
t) ! k)
          and are-all-operator-effects-consistent ((Φ⁻¹ Π 𝒜 t) ! k)
          using operator-preconditions-hold
          by blast+
         moreover have ?t_k' v = Some False
```

```
            using execute-parallel-operator-effect(2) op-in-π_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-π_k opera-
tor-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 ∈ set ((Π)_𝒱)
            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)_𝒱)
       then have (Φ_S⁻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
       have op-in: op ∈ set ((Π)_𝒪)
          using op-in decode-plan-step-element-then(1) k-lt-t
          by blast
       consider (A) v ∈ set (add-effects-of op)
         | (B) v ∈ set (delete-effects-of op)
         using v-is-or
         by blast
       hence False
```

```
      proof (cases)
        case A
        then have v ∈ set ((Π)_𝒱)
          using is-valid-problem-strips-operator-variable-sets(2)[OF
              assms(1)] op-in A
          by blast
        thus False
          using contradiction
          by blast
      next
        case B
        then have v ∈ set ((Π)_𝒱)
          using is-valid-problem-strips-operator-variable-sets(3)[OF
              assms(1)] op-in B
          by blast
        thus False
          using contradiction
          by blast
      qed
    }
  }
  hence v-in-set-vs: v ∈ set ((Π)_𝒱)
    by blast
  hence ?t_k' v = ?s_k' v
  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-set-π_k: op ∈ set ?π_k
     and v-options: v ∈ set (add-effects-of op) ∨ v ∈ set (delete-effects-of op)
      by blast
    then have op ∈ set ((Π)_𝒪)
      using decode-plan-step-element-then[OF k-lt-t]
      by blast
    then consider (v-is-add-effect) v ∈ set (add-effects-of op)
      | (v-is-delete-effect) v ∈ set (delete-effects-of op)
      using v-options
      by blast
    thus ?thesis
      proof (cases)
        case v-is-add-effect
        then have ?t_k' v = Some True
          using execute-parallel-operator-positive-effect-if[OF - - op-in-set-π_k]
            operator-preconditions-hold
          by blast
        moreover have ?s_k' v = Some True
              using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)
op-in-set-π_k]
            v-is-add-effect
```

```
            by blast
          ultimately show ?thesis
            by argo
        next
          case v-is-delete-effect
          then have ?t_k' v = Some False
            using execute-parallel-operator-negative-effect-if[OF - - op-in-set-π_k]
              operator-preconditions-hold
            by blast
          moreover have ?s_k' v = Some False
                using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)
op-in-set-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
          then consider (s_k-v-is-some-true) ?s_k v = Some True
            | (s_k-v-is-some-false) ?s_k v = Some False
            by fastforce
        }
        then show ?thesis
          proof (cases)
            case s_k-v-is-some-true
```

```
        moreover {
          have lit-semantics 𝒜 ((State k (index ?vs v))⁺)
        using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
              assms(1, 2) k-lte-t v-in-set-vs] s_k-v-is-some-true
          by simp
          then have lit-semantics 𝒜 ((State (Suc k) (index ?vs v))⁺)
          using state-propagation-positive
          unfolding cnf-semantics-def clause-semantics-def
          by fastforce
          then have ?s_k' v = Some True
        using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
              assms(1, 2) suc-k-lte-t v-in-set-vs]
          by fastforce
        }
        ultimately show ?thesis
          using t_k'-equals-s_k
          by simp
      next
        case s_k-v-is-some-false
        moreover {
          have lit-semantics 𝒜 ((State k (index ?vs v))⁻¹)
        using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
              assms(1, 2) k-lte-t v-in-set-vs] s_k-v-is-some-false
          by simp
          then have lit-semantics 𝒜 ((State (Suc k) (index ?vs v))⁻¹)
          using state-propagation-negative
          unfolding cnf-semantics-def clause-semantics-def
          by fastforce
          then have ?s_k' v = Some False
        using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
              assms(1, 2) suc-k-lte-t v-in-set-vs]
          by fastforce
        }
        ultimately show ?thesis
          using t_k'-equals-s_k
          by simp
      qed
    qed
  }
  hence ?t_k' ⊆_m ?s_k'
  using map-le-def
  by blast
 }
 ultimately show ?thesis
  using map-le-antisym
  by blast
qed

lemma encode-problem-parallel-correct-vi:
```

```
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ Π t
    and k < length (trace-parallel-plan-strips ((Π)_I) (Φ⁻¹ Π 𝒜 t))
  shows trace-parallel-plan-strips ((Π)_I) (Φ⁻¹ Π 𝒜 t) ! k
    = Φ_S⁻¹ Π 𝒜 k
  using assms
proof −
  let ?I = (Π)_I
    and ?π = Φ⁻¹ Π 𝒜 t
  let ?τ = trace-parallel-plan-strips ?I ?π
  show ?thesis
    using assms
    proof (induction k)
      case 0
      hence ?τ ! 0 = ?I
        using trace-parallel-plan-strips-head-is-initial-state
        by blast
      moreover have Φ_S⁻¹ Π 𝒜 0 = ?I
        using decode-state-at-initial-state[OF assms(1, 2)]
        by simp
      ultimately show ?case
        by simp
    next
      case (Suc k)
      let ?τ_k = trace-parallel-plan-strips ?I ?π ! k
        and ?s_k = Φ_S⁻¹ Π 𝒜 k
     have k-lt-length-τ-minus-one: k < length ?τ − 1 and k-lt-length-τ: k < length
?τ
        using Suc.prems(3)
        by linarith+
    — Use the induction hypothesis to obtain the proposition for the previous step
k. Then, show that applying the k-th parallel operator in the plan π on either the
state obtained from the trace or decoded from the model yields the same successor
state.
      {
        have ?τ ! k = execute-parallel-plan ?I (take k ?π)
          using trace-parallel-plan-plan-prefix k-lt-length-τ
          by blast
        hence ?τ_k = ?s_k
          using Suc.IH[OF assms(1, 2) k-lt-length-τ]
          by blast
      }
      moreover have trace-parallel-plan-strips ?I ?π ! Suc k
        = execute-parallel-operator ?τ_k (?π ! k)
        using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
        by blast
      moreover {
        thm Suc.prems(3)
        have length (trace-parallel-plan-strips ?I ?π) ≤ length ?π + 1
```

```
      using length-trace-parallel-plan-strips-lte-length-plan-plus-one
      by blast
    then have k < length ?π
      using Suc.prems(3)
      unfolding Suc-eq-plus1
      by linarith
    hence Φ_S⁻¹ Π 𝒜 (Suc k)
      = execute-parallel-operator ?s_k (?π ! k)
      using encode-problem-parallel-correct-v[OF assms(1, 2)]
      by simp
  }
  ultimately show ?case
    by argo
 qed
qed

lemma encode-problem-parallel-correct-vii:
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ Π t
  shows length (map (decode-state-at Π 𝒜)
      [0..<Suc (length (Φ⁻¹ Π 𝒜 t))])
    = length (trace-parallel-plan-strips ((Π)_I) (Φ⁻¹ Π 𝒜 t))
proof −
  let ?I = (Π)_I
    and ?π = Φ⁻¹ Π 𝒜 t
  let ?σ = map (decode-state-at Π 𝒜) [0..<Suc (length ?π)]
    and ?τ = trace-parallel-plan-strips ?I ?π
  let ?l = length ?τ
  let ?k = ?l − 1
  show ?thesis
    proof (rule ccontr)
      assume length-σ-neq-length-τ: length ?σ ≠ length ?τ
      {
        have length ?σ = length ?π + 1
          by fastforce
        moreover have length ?τ ≤ length ?π + 1
          using length-trace-parallel-plan-strips-lte-length-plan-plus-one
          by blast
        moreover have length ?τ < length ?π + 1
          using length-σ-neq-length-τ calculation
          by linarith
      } note nb_1 = this
      {
        have 0 < length ?τ
          using trace-parallel-plan-strips-not-nil..
        then have length ?τ − 1 < length ?π
          using nb_1
          by linarith
      } note nb_2 = 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 nb_2
        by blast
    } note nb_3 = 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)] nb_3
      moreover have (Φ_S⁻¹ Π 𝒜 ?k) = ?τ ! ?k
        using encode-problem-parallel-correct-vi[OF assms(1, 2)
            length-trace-minus-one-lt-length-trace]..
      ultimately have (Φ_S⁻¹ Π 𝒜 ?k) = execute-parallel-plan ?I (take ?k ?π)
        by argo
    } note nb_4 = this
    {
      have are-all-operators-applicable (Φ_S⁻¹ Π 𝒜 ?k) (?π ! ?k)
        and are-all-operator-effects-consistent (?π ! ?k)
        using encode-problem-parallel-correct-ii(1, 2)[OF assms(1, 2)] nb_3
        by blast+
      — Unsure why calculation(1, 2) is needed for this proof step. Should just
require the default proof.
      moreover have ¬are-all-operators-applicable (Φ_S⁻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 nb_1]
          calculation(1, 2)
        unfolding nb_3 nb_4
        by blast+
      ultimately have False
        by blast
    }
    thus False.
  qed
qed

lemma encode-problem-parallel-correct-x:
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ Π t
  shows map (decode-state-at Π 𝒜)
      [0..<Suc (length (Φ⁻¹ Π 𝒜 t))]
    = trace-parallel-plan-strips ((Π)_I) (Φ⁻¹ Π 𝒜 t)
proof −
  let ?I = (Π)_I
    and ?π = Φ⁻¹ Π 𝒜 t
```

```
  let ?σ = map (decode-state-at Π 𝒜) [0..<Suc (length ?π)]
    and ?τ = trace-parallel-plan-strips ?I ?π
  {
    have length ?τ = length ?σ
      using encode-problem-parallel-correct-vii[OF assms]..
    moreover {
      fix k
      assume k-lt-length-τ: k < length ?τ
      then have trace-parallel-plan-strips ((Π)_I) (Φ⁻¹ Π 𝒜 t) ! k
        = Φ_S⁻¹ Π 𝒜 k
        using encode-problem-parallel-correct-vi[OF assms]
        by blast
      moreover {
        have length ?τ ≤ length ?π + 1
          using length-trace-parallel-plan-strips-lte-length-plan-plus-one
          by blast
        then have k < length ?π + 1
          using k-lt-length-τ
          by linarith
        then have k < Suc (length ?π) − 0
          by simp
        hence ?σ ! k = Φ_S⁻¹ Π 𝒜 k
          using nth-map-upt[of k Suc (length ?π) 0]
          by auto
      }
      ultimately have ?τ ! k = ?σ ! k
        by argo
    }
    ultimately have ?τ = ?σ
      using list-eq-iff-nth-eq[of ?τ ?σ]
      by blast
  }
  thus ?thesis
    by argo
qed

lemma encode-problem-parallel-correct-xi:
  fixes Π:: 'a strips-problem
  assumes is-valid-problem-strips Π
   and 𝒜 ⊨ Φ Π t
   and ops ∈ set (Φ⁻¹ Π 𝒜 t)
   and op ∈ set ops
 shows op ∈ set ((Π)_𝒪)
proof −
  let ?π = Φ⁻¹ Π 𝒜 t
  have length ?π = t
    using decode-plan-length
    by force
  moreover obtain k where k < length ?π and ops = ?π ! k
```

using *in-set-conv-nth*[*of ops ?π*] *assms*(*3*)
unfolding *calculation*
by *blast*
ultimately show *?thesis*
using *assms*(*4*) *decode-plan-step-element-then*(*1*)
by *force*
qed

To show soundness, we have to prove the following: given the existence of a model $\mathcal{A}$ of the basic SATPlan encoding $\Phi\ \Pi\ t$ for a given valid problem Π and hypothesized plan length t, the decoded plan $\pi \equiv \Phi^{-1}\ \Pi\ \mathcal{A}\ t$ is a parallel solution for Π.

We show this theorem by showing equivalence between the execution trace of the decoded plan and the sequence of states

$$\sigma = map\ (\lambda\ k.\ {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k)\ [0..<Suc\ (length\ ?\pi)]$$

decoded from the model $\mathcal{A}$. Let

$$\tau \equiv trace\text{-}parallel\text{-}plan\text{-}strips\ I\ \pi$$

be the trace of π. Theorem **??** first establishes the equality $\sigma = \tau$ of the decoded state sequence and the trace of π. We can then derive that $G \subseteq_m last\ \sigma$ 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 π is a solution for I and G.

Moreover, we show that all operators *op* in all parallel operators $ops \in set\ \pi$ are also contained in $\mathcal{O}$. This is the case because the plan decoding function reverses the encoding function (which only encodes operators in $\mathcal{O}$).

By definition **??** this means that π is a parallel solution for Π. Moreover π has length t as confirmed by lemma . [9]

theorem *encode-problem-parallel-sound*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
shows *is-parallel-solution-for-problem* Π $(\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
proof −
let *?ops* = *strips-problem.operators-of* Π
and *?I* = $(\Pi)_I$
and *?G* = $(\Pi)_G$
and *?π* = $\Phi^{-1}\ \Pi\ \mathcal{A}\ t$
let *?σ* = *map* $(\lambda\ k.\ {\Phi_S}^{-1}\ \Pi\ \mathcal{A}\ k)$ [*0..<Suc* (*length ?π*)]
and *?τ* = *trace-parallel-plan-strips ?I ?π*
{

[9] This lemma is used in the proof but not shown.

have *?σ = ?τ*
using *encode-problem-parallel-correct-x*[*OF assms*].
moreover {
have *length ?π = t*
using *decode-plan-length*
by *auto*
then have *?G* $\subseteq_m$ *last ?σ*
using *decode-state-at-goal-state*[*OF assms*]
by *simp*
}
ultimately have $((\Pi)_G) \subseteq_m$ *execute-parallel-plan* $((\Pi)_I)$ $(\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
using *execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace*
by *auto*
}
moreover have $\forall$ *ops* $\in$ *set ?π*. $\forall$ *op* $\in$ *set ops*. *op* $\in$ *set* $((\Pi)_{\mathcal{O}})$
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* ($\mathcal{A}_0$)
where *empty-valuation* $\equiv$ (λ-. *False*)

abbreviation *valuation-for-state*
:: *'variable list*
$\Rightarrow$*'variable strips-state*
$\Rightarrow$ *nat*
$\Rightarrow$ *'variable*
$\Rightarrow$ *sat-plan-variable valuation*
$\Rightarrow$ *sat-plan-variable valuation*
where *valuation-for-state vs s k v* $\mathcal{A}$
$\equiv \mathcal{A}$(*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 $k' \in \{$*length* τ..*length* π + *1*$\}$ for all $v \in \mathcal{V}$ (see $\mathcal{A}_2$).
definition *valuation-for-state-variables*
:: *'variable strips-problem*
$\Rightarrow$ *'variable strips-operator list list*
$\Rightarrow$ *'variable strips-state list*

```
  ⇒ sat-plan-variable valuation
 where valuation-for-state-variables Π π τ ≡ let
    t' = length τ
    ; τΩ = τ ! (t' − 1)
    ; vs = variables-of Π
    ; V1 = { State k (index vs v) | k v. k ∈ {0..<t'} ∧ v ∈ set vs }
    ; V2 = { State k (index vs v) | k v. k ∈ {t'..(length π + 1)} ∧ v ∈ set vs }
    ; A1 = foldr
      (λ(k, v) A. valuation-for-state (variables-of Π) (τ ! k) k v A)
      (List.product [0..<t'] vs)
      A0
    ; A2 = foldr
      (λ(k, v) A. valuation-for-state (variables-of Π) τΩ k v A)
      (List.product [t'..<length π + 2] vs)
      A0
  in override-on (override-on A0 A1 V1) A2 V2
```

— The valuation is left to yield false for the potentially remaining $k' \in \{length\ \tau..length\ \pi + 1\}$ since no more operators are executed after the trace ends anyway. The definition of $\mathcal{A}_0$ as the valuation that is false for every argument ensures this implicitely.

```
definition valuation-for-operator-variables
 :: 'variable strips-problem
  ⇒ 'variable strips-operator list list
  ⇒ 'variable strips-state list
  ⇒ sat-plan-variable valuation
 where valuation-for-operator-variables Π π τ ≡ let
    ops = operators-of Π
    ; Op = { Operator k (index ops op) | k op. k ∈ {0..<length τ − 1} ∧ op ∈
set ops }
  in override-on
    A0
    (foldr
      (λ(k, op) A. A(Operator k (index ops op) := True))
      (concat (map (λk. map (Pair k) (π ! k)) [0..<length τ − 1]))
      A0)
    Op
```

The completeness proof requires that we show that the SATPlan encoding $\Phi\ \Pi\ t$ of a problem Π has a model $\mathcal{A}$ in case a solution π with length t exists. Since a plan corresponds to a state trace $\tau \equiv trace\text{-}parallel\text{-}plan\text{-}strips\ I\ \pi$ with

$$\tau\ !\ k = execute\text{-}parallel\text{-}plan\ I\ (take\ k\ \pi)$$

for all $k < length\ \tau$ we can construct a valuation $\mathcal{A}_V$ modeling the state sequence in τ by letting

$$\mathcal{A}(State\ k\ (index\ vs\ v) := (s\ v = Some\ True))$$

or all $v \in \mathcal{V}$ where $s \equiv \tau\ !\ k$. [10]
Similarly to $\mathcal{A}_V$, we obtain an operator valuation $\mathcal{A}_O$ by defining

$\mathcal{A}(Operator\ k\ (index\ ops\ op) := True)$

for all operators $op \in \mathcal{O}$ s.t. $op \in set\ (\pi\ !\ k)$ for all $k < length\ \tau - 1$.
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*
⇒ *'variable strips-operator list list*
⇒ *sat-plan-variable valuation*
where *valuation-for-plan* $\Pi\ \pi \equiv$ *let*
vs = *variables-of* Π
; *ops* = *operators-of* Π
; τ = *trace-parallel-plan-strips* (*initial-of* Π) π
; *t* = *length* π
; t' = *length* τ
; $\mathcal{A}_V$ = *valuation-for-state-variables* $\Pi\ \pi\ \tau$
; $\mathcal{A}_O$ = *valuation-for-operator-variables* $\Pi\ \pi\ \tau$
; *V* = { *State k* (*index vs v*)
| *k v*. $k \in \{0..<t + 1\} \wedge v \in set\ vs$ }
; *Op* = { *Operator k* (*index ops op*)
| *k op*. $k \in \{0..<t\} \wedge op \in set\ ops$ }
in override-on (*override-on* $\mathcal{A}_0\ \mathcal{A}_V$ *V*) $\mathcal{A}_O$ *Op*

— Show that in case of an encoding with makespan zero, it suffices to show that a given model satisfies the initial state and goal state encodings.

lemma *model-of-encode-problem-makespan-zero-iff*:
$\mathcal{A} \models \Phi\ \Pi\ 0 \longleftrightarrow \mathcal{A} \models \Phi_I\ \Pi \wedge (\Phi_G\ \Pi)\ 0$
proof −
have *encode-operators* $\Pi\ 0 = \neg\bot \wedge \neg\bot$
unfolding *encode-operators-def encode-all-operator-effects-def*
encode-all-operator-preconditions-def
by *simp*
moreover have *encode-all-frame-axioms* $\Pi\ 0 = \neg\bot$
unfolding *encode-all-frame-axioms-def*
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

[10] It is helpful to remember at this point, that the trace elements of a solution contain the states reached by plan prefix execution (lemma **??**).

lemma *empty-valution-is-False*[*simp*]: $\mathcal{A}_0\ v = \mathit{False}$
unfolding *empty-valuation-def*..

lemma *model-initial-state-set-valuations*:
assumes *is-valid-problem-strips* Π
shows *set* (*map* (λv. *case* $((\Pi)_I)\ v$ *of Some b*
$\Rightarrow \mathcal{A}_0$(*State 0* (*index* (*strips-problem.variables-of* Π) v) := b)
| - $\Rightarrow \mathcal{A}_0$)
(*strips-problem.variables-of* Π))
= { $\mathcal{A}_0$(*State 0* (*index* (*strips-problem.variables-of* Π) v) := *the* $(((\Pi)_I)\ v)$)
| v. $v \in$ *set* $((\Pi)_\mathcal{V})$ }
proof −
let *?I* = $(\Pi)_I$
and *?vs* = *strips-problem.variables-of* Π
let *?f* = λv. *case* $((\Pi)_I)\ v$ *of Some b*
$\Rightarrow \mathcal{A}_0$(*State 0* (*index ?vs v*) := b) | - $\Rightarrow \mathcal{A}_0$
and *?g* = λv. $\mathcal{A}_0$(*State 0* (*index ?vs v*) := *the* (*?I v*))
let *?As* = *map ?f ?vs*
have nb_1: *dom ?I* = *set* $((\Pi)_\mathcal{V})$
using *is-valid-problem-strips-initial-of-dom assms*
by *fastforce*
{
{
fix v
assume $v \in$ *dom ?I*
hence *?f v* = *?g v*
using nb_1
by *fastforce*
}
hence *?f* ‘ *set* $((\Pi)_\mathcal{V})$ = *?g* ‘ *set* $((\Pi)_\mathcal{V})$
using nb_1
by *force*
}
then have *set ?As* = *?g* ‘ *set* $((\Pi)_\mathcal{V})$
unfolding *set-map*
by *simp*
thus *?thesis*
by *blast*
qed

lemma *valuation-of-state-variable-implies-lit-semantics-if*:
assumes $v \in$ *dom S*
and $\mathcal{A}$ (*State k* (*index vs v*)) = *the* (*S v*)
shows *lit-semantics* $\mathcal{A}$ (*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
              by blast
          } moreover {
            — Follows from modus tollens on the definition of inj-on.
            have f a ≠ f x
              using Cons.prems False
              by force
            moreover have foldr (λx A. A(f x := g x)) (a # xs) A
```

```
        = (foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜)(f a := g a)
        by simp
      ultimately have foldr (λx 𝒜. 𝒜(f x := g x)) (a # xs) 𝒜 (f x)
        = (foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜) (f x)
        unfolding fun-upd-def
        by presburger
    } ultimately show ?thesis
      by argo
  qed simp
 qed
qed fastforce

lemma foldr-fun-no-upd:
  assumes inj-on f (set xs)
    and y ∉ f ‘ set xs
  shows foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜 y = 𝒜 y
  using assms
proof (induction xs)
  case (Cons a xs)
  {
    have inj-on f (set xs) and y ∉ f ‘ set xs
      using Cons.prems
      by (fastforce, simp)
    hence foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜 y = 𝒜 y
      using Cons.IH
      by blast
  }
  moreover {
    have f a ≠ y
      using Cons.prems(2)
      by auto
    moreover have foldr (λx 𝒜. 𝒜(f x := g x)) (a # xs) 𝒜
      = (foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜)(f a := g a)
      by simp
    ultimately have foldr (λx 𝒜. 𝒜(f x := g x)) (a # xs) 𝒜 y
      = (foldr (λx 𝒜. 𝒜(f x := g x)) xs 𝒜) y
      unfolding fun-upd-def
      by presburger
  }
  ultimately show ?case
    by argo
qed simp

— We only use the part of the characterization of 𝒜 which pertains to the state variables here.
lemma encode-problem-parallel-complete-i:
  fixes Π::'a strips-problem
  assumes is-valid-problem-strips Π
    and (Π)_G ⊆_m execute-parallel-plan ((Π)_I) π
```

```
    ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
     ⟶ (A (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
       ∧ (¬A (State k (index (strips-problem.variables-of Π) v))
        ⟷ ((trace-parallel-plan-strips ((Π)_I) π ! k) v ≠ Some True))
  shows A ⊨ Φ_I Π
proof −
  let ?vs = strips-problem.variables-of Π
    and ?I = (Π)_I
    and ?G = (Π)_G
    and ?Φ_I = Φ_I Π
  let ?τ = trace-parallel-plan-strips ?I π
  {
    fix C
    assume C ∈ cnf ?Φ_I
    then obtain v
      where v-in-set-vs: v ∈ set ?vs
      and C-is: C = { literal-formula-to-literal (encode-state-variable 0 (index ?vs
v) (?I v)) }
      using cnf-of-encode-initial-state-set-ii[OF assms(1)]
      by auto
    {
      have 0 < length ?τ
        using trace-parallel-plan-strips-not-nil
        by blast
      then have A (State 0 (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! 0) v = Some True
        and ¬A (State 0 (index (strips-problem.variables-of Π) v))
        ⟷ ((trace-parallel-plan-strips ((Π)_I) π ! 0) v ≠ Some True)
        using assms(3)
        by (presburger+)
    } note nb = this
    {
      let ?L = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?I
v))
      have τ-0-is: ?τ ! 0 = ?I
        using trace-parallel-plan-strips-head-is-initial-state
        by blast
      have v-in-dom-I: v ∈ dom ?I
        using is-valid-problem-strips-initial-of-dom assms(1) v-in-set-vs
        by fastforce
      then consider (I-v-is-Some-True) ?I v = Some True
        | (I-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* $\mathcal{A}$ *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 $\mathcal{A}$ Otherwise, we have to first show that the goal was reached (albeit early) and that our valuation $\mathcal{A}$ reflects the termination of plan execution after the time point at which the goal was reached.

lemma *encode-problem-parallel-complete-ii*:
fixes Π::*'a strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m$ *execute-parallel-plan* $((\Pi)_I)$ π
and $\forall$ *v k*. *k* < *length* (*trace-parallel-plan-strips* $((\Pi)_I)$ π)
$\longrightarrow$ ($\mathcal{A}$ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))
$\longleftrightarrow$ (*trace-parallel-plan-strips* $((\Pi)_I)$ π ! *k*) *v* = *Some True*)
and $\forall$ *v l*. *l* ≥ *length* (*trace-parallel-plan-strips* $((\Pi)_I)$ π) ∧ *l* < *length* π + *1*
$\longrightarrow$ $\mathcal{A}$ (*State l* (*index* (*strips-problem.variables-of* Π) *v*))
= $\mathcal{A}$ (*State* (*length* (*trace-parallel-plan-strips* $((\Pi)_I)$ π) − *1*)
(*index* (*strips-problem.variables-of* Π) *v*))
shows $\mathcal{A} \models (\Phi_G\ \Pi)(\textit{length}\ \pi)$
proof −
let *?vs* = *strips-problem.variables-of* Π
and *?I* = $(\Pi)_I$
and *?G* = $(\Pi)_G$
and *?*Φ_I = Φ_I Π
and *?t* = *length* π
and *?*Φ_G = $(\Phi_G\ \Pi)$ (*length* π)
let *?*τ = *trace-parallel-plan-strips ?I* π
let *?t′* = *length ?*τ
{
fix *v*
assume *G-of-v-is-not-None*: *?G v* ≠ *None*
have *?G* $\subseteq_m$ *last ?*τ
using *execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace assms*(*2*)
by *blast*
also have ... = *?*τ ! (*?t′* − *1*)
using *last-conv-nth*[*OF trace-parallel-plan-strips-not-nil*].
finally have *?G* $\subseteq_m$ *?*τ ! (*?t′* − *1*)
by *argo*
hence (*?*τ ! (*?t′* − *1*)) *v* = *?G v*

using *G-of-v-is-not-None*
unfolding *map-le-def*
by *force*
} **note** nb_1 = *this*

— Discriminate on whether the trace has full length or not and show that the model valuation of the state variables always correspond to the (defined) goal state values.
{
fix *v*
assume *G-of-v-is-not-None*: *?G v* ≠ *None*
hence $\mathcal{A}$ (*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 $\mathcal{A}$ (*State ?t* (*index ?vs v*)) ⟷ (*?τ* ! *?t*) *v* = *Some True*
using *assms*(*3*) *calculation*(*2*)
by *blast*
ultimately show *?thesis*
using nb_1[*OF G-of-v-is-not-None*]
by *force*
next
case *False*
{
have *?t′* < *?t* + *1*
using *length-trace-parallel-plan-strips-lte-length-plan-plus-one False*
le-neq-implies-less
by *blast*
moreover have $\mathcal{A}$ (*State ?t* (*index ?vs v*)) = $\mathcal{A}$ (*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 $\mathcal{A}$ (*State* (*?t′* − *1*) (*index ?vs v*)) ⟷ (*?τ* ! (*?t′* − *1*)) *v* = *Some True*
using *assms*(*3*) *calculation*(*3*)
by *blast*
ultimately have $\mathcal{A}$ (*State ?t* (*index ?vs v*)) ⟷ (*?τ* ! (*?t′* − *1*)) *v* = *Some True*
by *blast*
}
thus *?thesis*
using nb_1[*OF G-of-v-is-not-None*]
by *presburger*

```
    qed
  } note nb_2 = this
  {
    fix C
    assume C-in-cnf-of-Φ_G: C ∈ cnf ?Φ_G

    moreover obtain v
      where v ∈ set ?vs
        and G-of-v-is-not-None: ?G v ≠ None
      and C-is: C = { literal-formula-to-literal (encode-state-variable ?t (index ?vs
v)
        (?G v)) }
      using cnf-of-encode-goal-state-set-ii[OF assms(1)] calculation
      by auto
    consider (G-of-v-is-Some-True) ?G v = Some True
      | (G-of-v-is-Some-False) ?G v = Some False
      using G-of-v-is-not-None
      by fastforce
    then have clause-semantics A C
      using nb_2 C-is
      unfolding clause-semantics-def encode-state-variable-def
      by (cases, force+)
  }
  thus ?thesis
    using cnf-semantics[OF is-nnf-cnf[OF encode-goal-state-is-cnf[OF assms(1)]]]
    unfolding cnf-semantics-def
    by blast
qed

— We are not using the full characterization of A here since it's not needed.

lemma encode-problem-parallel-complete-iii-a:
  fixes Π::'a strips-problem
  assumes is-valid-problem-strips Π
    and (Π)_G ⊆_m execute-parallel-plan ((Π)_I) π
    and C ∈ cnf (encode-all-operator-preconditions Π (strips-problem.operators-of
Π) (length π))
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ A (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ l op. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) − 1 ∧ l < length π
      ⟶ ¬A (Operator l (index (strips-problem.operators-of Π) op))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (A (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
  shows clause-semantics A C
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
```

```
  and ?t = length π
let ?τ = trace-parallel-plan-strips ((Π)_I) π

obtain k op
  where k-and-op-are: (k, op) ∈ ({0..<?t} × set ((Π)_O))
    and C ∈ (⋃ v ∈ set (precondition-of op). {{ (Operator k (index ?ops op))^-1
      , (State k (index ?vs v))^+ }})
  using cnf-of-encode-all-operator-preconditions-structure assms(3)
    UN-E[of C ]
  by auto
then obtain v
  where v-in-preconditions-of-op: v ∈ set (precondition-of op)
    and C-is: C = { (Operator k (index ?ops op))^-1, (State k (index ?vs v))^+ }
  by blast
thus ?thesis
  proof (cases k < length ?τ − 1)
    case k-lt-length-τ-minus-one: True
    thus ?thesis
      proof (cases op ∈ set (π ! k))
        case True
        {
          have are-all-operators-applicable (?τ ! k) (π ! k)
        using trace-parallel-plan-strips-operator-preconditions k-lt-length-τ-minus-one
            by blast
          then have (?τ ! k) v = Some True
            using are-all-operators-applicable-set v-in-preconditions-of-op True
            by fast
          hence A (State k (index ?vs v))
            using assms(6) k-lt-length-τ-minus-one
            by force
        }
        thus ?thesis
          using C-is
          unfolding clause-semantics-def
          by fastforce
      next
        case False
        then have ¬A (Operator k (index ?ops op))
          using assms(4) k-lt-length-τ-minus-one
          by blast
        thus ?thesis
          using C-is
          unfolding clause-semantics-def
          by fastforce
      qed
  next
    case False
    then have k ≥ length ?τ − 1 k < ?t
      using k-and-op-are
```

```
      by(force, simp)
    then have ¬𝒜 (Operator k (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 𝒜 here since it's not needed.

lemma encode-problem-parallel-complete-iii-b:
  fixes Π::'a strips-problem
  assumes is-valid-problem-strips Π
    and (Π)_G ⊆_m execute-parallel-plan ((Π)_I) π
     and C ∈ cnf (encode-all-operator-effects Π (strips-problem.operators-of Π)
(length π))
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ 𝒜 (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ l op. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) − 1 ∧ l < length π
      ⟶ ¬𝒜 (Operator l (index (strips-problem.operators-of Π) op))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (𝒜 (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
  shows clause-semantics 𝒜 C
proof −
  let ?ops = strips-problem.operators-of Π
    and ?vs = strips-problem.variables-of Π
    and ?t = length π
  let ?τ = trace-parallel-plan-strips ((Π)_I) π
  let ?A = (⋃(k, op) ∈ {0..<?t} × set ((Π)_𝒪).
    ⋃v ∈ set (add-effects-of op).
      {{ (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁺ }})
    and ?B = (⋃(k, op) ∈ {0..<?t} × set ((Π)_𝒪).
      ⋃v ∈ set (delete-effects-of op).
        {{ (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁻¹ }})
  consider (C-in-A) C ∈ ?A
    | (C-in-B) C ∈ ?B
    using Un-iff[of C ?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 op
        where k-and-op-are: (k, op) ∈ {0..<?t} × set((Π)_𝒪)
          and C ∈ (⋃v ∈ set (add-effects-of op).
```

```
      {{ (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁺ }})
    by blast
  then obtain v where v-in-add-effects-of-op: v ∈ set (add-effects-of op)
    and C-is: C = { (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs
v))⁺ }
    by blast
  thus ?thesis
    proof (cases k < length ?τ − 1)
      case k-lt-length-τ-minus-one: True
      thus ?thesis
        proof (cases op ∈ set (π ! k))
          case True
          {
            then have are-all-operators-applicable (?τ ! k) (π ! k)
              and are-all-operator-effects-consistent (π ! k)
           using trace-parallel-plan-strips-operator-preconditions k-lt-length-τ-minus-one
              by blast+
            hence execute-parallel-operator (?τ ! k) (π ! k) v = Some True
              using execute-parallel-operator-positive-effect-if[
                OF - - True v-in-add-effects-of-op, of ?τ ! k]
              by blast
          }
          then have τ-Suc-k-is-Some-True: (?τ ! Suc k) v = Some True
            using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
            by argo
          have A (State (Suc k) (index ?vs v))
            using assms(6) k-lt-length-τ-minus-one τ-Suc-k-is-Some-True
            by fastforce
          thus ?thesis
            using C-is
            unfolding clause-semantics-def
            by fastforce
        next
          case False
          then have ¬A (Operator k (index ?ops op))
            using assms(4) k-lt-length-τ-minus-one
            by blast
          thus ?thesis
            using C-is
            unfolding clause-semantics-def
            by force
        qed
    next
      case False
      then have k ≥ length ?τ − 1 and k < ?t
        using k-and-op-are
        by auto
      then have ¬A (Operator k (index ?ops op))
        using assms(5)
```

```
        by blast
      thus ?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 op
      where k-and-op-are: (k, op) ∈ {0..<?t} × set ((Π)_𝒪)
        and C ∈ (⋃ v ∈ set (delete-effects-of op).
          {{ (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs v))⁻¹ }})
      by blast
    then obtain v where v-in-delete-effects-of-op: v ∈ set (delete-effects-of op)
      and C-is: C = { (Operator k (index ?ops op))⁻¹, (State (Suc k) (index ?vs
v))⁻¹ }
      by blast
    thus ?thesis
      proof (cases k < length ?τ − 1)
        case k-lt-length-τ-minus-one: True
        thus ?thesis
          proof (cases op ∈ set (π ! k))
            case True
            {
              then have are-all-operators-applicable (?τ ! k) (π ! k)
                and are-all-operator-effects-consistent (π ! k)
             using trace-parallel-plan-strips-operator-preconditions k-lt-length-τ-minus-one
                by blast+
              hence execute-parallel-operator (?τ ! k) (π ! k) v = Some False
                using execute-parallel-operator-negative-effect-if[
                  OF - - True v-in-delete-effects-of-op, of ?τ ! k]
                by blast
            }
            then have τ-Suc-k-is-Some-True: (?τ ! Suc k) v = Some False
              using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
              by argo
            have ¬𝒜 (State (Suc k) (index ?vs v))
              using assms(6) k-lt-length-τ-minus-one τ-Suc-k-is-Some-True
              by fastforce
            thus ?thesis
              using C-is
              unfolding clause-semantics-def
              by fastforce
          next
            case False
            then have ¬𝒜 (Operator k (index ?ops op))
              using assms(4) k-lt-length-τ-minus-one
              by blast
```

```
        thus ?thesis
          using C-is
          unfolding clause-semantics-def
          by force
      qed
    next
      case False
      then have k ≥ length ?τ − 1 and k < ?t
        using k-and-op-are
        by auto
      then have ¬𝒜 (Operator k (index ?ops op))
        using assms(5)
        by blast
      thus ?thesis
        using C-is
        unfolding clause-semantics-def
        by fastforce
    qed
  qed
qed


lemma encode-problem-parallel-complete-iii:
  fixes Π::'a strips-problem
  assumes is-valid-problem-strips Π
    and (Π)_G ⊆_m execute-parallel-plan ((Π)_I) π
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ 𝒜 (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ l op. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) − 1 ∧ l < length π
      ⟶ ¬𝒜 (Operator l (index (strips-problem.operators-of Π) op))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (𝒜 (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
  shows 𝒜 ⊨ encode-operators Π (length π)
proof −
  let ?t = length π
    and ?ops = strips-problem.operators-of Π
  let ?Φ_O = encode-operators Π ?t
    and ?Φ_P = encode-all-operator-preconditions Π ?ops?t
    and ?Φ_E = encode-all-operator-effects Π ?ops ?t
  {
    fix C
    assume C ∈ cnf ?Φ_O
    then consider (C-in-precondition-encoding) C ∈ cnf ?Φ_P
      | (C-in-effect-encoding) C ∈ cnf ?Φ_E
      using cnf-of-operator-encoding-structure
      by blast
    hence clause-semantics 𝒜 C
```

```
    proof (cases)
      case C-in-precondition-encoding
      thus ?thesis
        using encode-problem-parallel-complete-iii-a[OF assms(1, 2) - assms(3,
4, 5)]
        by blast
    next
      case C-in-effect-encoding
      thus ?thesis
       using encode-problem-parallel-complete-iii-b[OF assms(1, 2) - assms(3, 4,
5)]
        by blast
    qed
  }
  thus ?thesis
    using encode-operators-is-cnf[OF assms(1)] is-nnf-cnf cnf-semantics
    unfolding cnf-semantics-def
    by blast
qed


lemma encode-problem-parallel-complete-iv-a:
  fixes Π :: 'a strips-problem
  assumes STRIPS-Semantics.is-parallel-solution-for-problem Π π
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ 𝒜 (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (𝒜 (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
    and ∀ v l. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) ∧ l < length π + 1
      ⟶ 𝒜 (State l (index (strips-problem.variables-of Π) v))
        = 𝒜 (State
          (length (trace-parallel-plan-strips ((Π)_I) π) − 1)
          (index (strips-problem.variables-of Π) v))
    and C ∈ ⋃ (⋃(k, v) ∈ {0..<length π} × set ((Π)_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 ((Π)_𝒪) ∧ v ∈ set (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)_I) 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** $C' \in ?A$ **and** *C-in-C'*: $C \in C'$
using *Union-iff assms*(5)
by *auto*
then obtain k v
where $(k, v) \in \{0..<?t\} \times set\ ?vs$
and $C' \in \{\{\{\ (State\ k\ (index\ ?vs\ v))^+, (State\ (Suc\ k)\ (index\ ?vs\ v))^{-1}\ \}$
$\cup \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (add\text{-}effects\text{-}of$
op) }}}
using *UN-E*
by *blast*
hence $\exists\, k\ v.$
$k \in \{0..<?t\}$
$\wedge\ v \in set\ ?vs$
$\wedge\ C = \{\ (State\ k\ (index\ ?vs\ v))^+, (State\ (Suc\ k)\ (index\ ?vs\ v))^{-1}\ \}$
$\cup \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (add\text{-}effects\text{-}of$
op) }
using *C-in-C'*
by *blast*
}
then obtain k v
where *k-in*: $k \in \{0..<?t\}$
and *v-in-vs*: $v \in set\ ?vs$
and *C-is*: $C = \{\ (State\ k\ (index\ ?vs\ v))^+, (State\ (Suc\ k)\ (index\ ?vs\ v))^{-1}\ \}$
$\cup \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (add\text{-}effects\text{-}of$
op) }
by *blast*
show *?thesis*
proof (*cases* $k < length\ ?\tau - 1$)
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* $(?\tau\ !\ k)\ (\pi\ !\ k)$
and *all-operator-effects-consistent*: *are-all-operator-effects-consistent* $(\pi\ !\ k)$
using *trace-parallel-plan-strips-operator-preconditions*[*OF k-lt-length-τ-minus-one*]
by *simp+*
then consider (*A*) $\exists\, op \in set\ (\pi\ !\ k).\ v \in set\ (add\text{-}effects\text{-}of\ op)$
| (*B*) $\exists\, op \in set\ (\pi\ !\ k).\ v \in set\ (delete\text{-}effects\text{-}of\ op)$
| (*C*) $\forall\, op \in set\ (\pi\ !\ k).\ v \notin set\ (add\text{-}effects\text{-}of\ op) \wedge v \notin set\ (delete\text{-}effects\text{-}of$
op)
by *blast*
thus *?thesis*
proof (*cases*)
case *A*
moreover obtain *op*

where *op-in-*π_k: $op \in set\ (\pi\ !\ k)$
and *v-is-add-effect*: $v \in set\ (add\text{-}effects\text{-}of\ op)$
using *A*
by *blast*
moreover {
have $(\pi\ !\ k) \in set\ \pi$
using *k-lt-t*
by *simp*
hence $op \in set\ ?ops$
using *is-parallel-solution-for-problem-operator-set*[*OF assms*(*1*) *op-in-*π_k]
by *blast*
}
ultimately have $(Operator\ k\ (index\ ?ops\ op))^+$
$\in \{\ (Operator\ k\ (index\ ?ops\ op))^+ \mid op.\ op \in set\ ?ops \wedge v \in set\ (add\text{-}effects\text{-}of\ op)\ \}$
using *v-is-add-effect*
by *blast*
then have $(Operator\ k\ (index\ ?ops\ op))^+ \in C$
using *C-is*
by *auto*
moreover have $\mathcal{A}\ (Operator\ k\ (index\ ?ops\ op))$
using *assms*(*2*) *k-lt-length-*τ*-minus-one op-in-*π_k
by *blast*
ultimately show *?thesis*
unfolding *clause-semantics-def*
by *force*
next
case *B*
then obtain *op*
where *op-in-*π_k: $op \in set\ (\pi\ !\ k)$
and *v-is-delete-effect*: $v \in set\ (delete\text{-}effects\text{-}of\ op)$**..**
then have $\neg(\exists\ op \in set\ (\pi\ !\ k).\ v \in set\ (add\text{-}effects\text{-}of\ op))$
using *all-operator-effects-consistent are-all-operator-effects-consistent-set*
by *fast*
then have *execute-parallel-operator* $(?\tau\ !\ k)\ (\pi\ !\ k)\ v$
$= Some\ False$
using *execute-parallel-operator-negative-effect-if*[*OF all-operators-applicable*
all-operator-effects-consistent op-in-π_k *v-is-delete-effect*]
by *blast*
moreover have $(?\tau\ !\ Suc\ k)\ v = execute\text{-}parallel\text{-}operator\ (?\tau\ !\ k)\ (\pi\ !\ k)\ v$
using *trace-parallel-plan-step-effect-is*[*OF k-lt-length-*τ*-minus-one*]
by *simp*
ultimately have $\neg\mathcal{A}\ (State\ (Suc\ k)\ (index\ ?vs\ v))$
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
      proof (cases (?τ ! k) v = Some True)
       case True
       then have 𝒜 (State k (index ?vs v))
        using assms(3) k-lt-length-τ-minus-one
        by force
       thus ?thesis
        using C-is
        unfolding clause-semantics-def
        by fastforce
      next
       case False
       {
        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
        hence (?τ ! Suc k) v ≠ Some True
         using False
         by argo
       }
       then have ¬𝒜 (State (Suc k) (index ?vs v))
        using assms(3) k-lt-length-τ-minus-one
        by auto
       thus ?thesis
        using C-is
        unfolding clause-semantics-def
        by fastforce
      qed
    qed
next
  case k-gte-length-τ-minus-one: False
  show ?thesis
    proof (cases 𝒜 (State (length ?τ − 1) (index ?vs v)))
     case True
     {
      have 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 True
          by blast
      }
      thus ?thesis
        using C-is
        unfolding clause-semantics-def
        by simp
    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 fastforce
    qed
  qed
qed


lemma encode-problem-parallel-complete-iv-b:
  fixes Π :: 'a strips-problem
  assumes is-parallel-solution-for-problem Π π
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ 𝒜 (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (𝒜 (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
    and ∀ v l. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) ∧ l < length π + 1
      ⟶ 𝒜 (State l (index (strips-problem.variables-of Π) v))
        = 𝒜 (State
          (length (trace-parallel-plan-strips ((Π)_I) π) − 1)
          (index (strips-problem.variables-of Π) v))
    and C ∈ ⋃ (⋃(k, v) ∈ {0..<length π} × set ((Π)_𝒱).
      {{{ (State k (index (strips-problem.variables-of Π) v))^-1
```

```
        , (State (Suc k) (index (strips-problem.variables-of Π) v))⁺ }
      ∪ { (Operator k (index (strips-problem.operators-of Π) op))⁺
        |op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (delete-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 (initial-of Π) π
  let ?A = (⋃(k, v) ∈ {0..<?t} × set ?vs.
    {{{ (State k (index ?vs v))⁻¹, (State (Suc k) (index ?vs v))⁺ }
      ∪ { (Operator k (index ?ops op))⁺
        | op. op ∈ set ((Π)_𝒪) ∧ v ∈ 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} × set ?vs
      and C' ∈ {{{ (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) }}}
      using UN-E
      by fastforce
    hence ∃ k v.
      k ∈ {0..<?t}
      ∧ v ∈ set ?vs
      ∧ C = { (State k (index ?vs v))⁻¹, (State (Suc k) (index ?vs v))⁺ }
        ∪ { (Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ 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 ∈ set ((Π)_𝒱)
      and C-is: C = { (State k (index ?vs v))⁻¹, (State (Suc k) (index ?vs v))⁺ }
        ∪ { (Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (delete-effects-of op) }
    by auto
  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* (*delete-effects-of op*)
| (*B*) ∃ *op* ∈ *set* (*π* ! *k*). *v* ∈ *set* (*add-effects-of op*)
| (*C*) ∀ *op* ∈ *set* (*π* ! *k*). *v* ∉ *set* (*add-effects-of op*) ∧ *v* ∉ *set* (*delete-effects-of op*)
by *blast*
thus *?thesis*
proof (*cases*)
case *A*
moreover obtain *op*
where *op-in-*π_k: *op* ∈ *set* (*π* ! *k*)
and *v-is-delete-effect*: *v* ∈ *set* (*delete-effects-of op*)
using *A*
by *blast*
moreover {
have (*π* ! *k*) ∈ *set π*
using *k-lt-t*
by *simp*
hence *op* ∈ *set ?ops*
using *is-parallel-solution-for-problem-operator-set*[*OF assms*(*1*) *op-in-*π_k]
by *auto*
}
ultimately have (*Operator k* (*index ?ops op*))$^+$
∈ { (*Operator k* (*index ?ops op*))$^+$
| *op*. *op* ∈ *set ?ops* ∧ *v* ∈ *set* (*delete-effects-of op*) }
using *v-is-delete-effect*
by *blast*
then have (*Operator k* (*index ?ops op*))$^+$ ∈ *C*
using *C-is*
by *auto*
moreover have $\mathcal{A}$ (*Operator k* (*index ?ops op*))
using *assms*(*2*) *k-lt-length-τ-minus-one op-in-*π_k
by *blast*
ultimately show *?thesis*
unfolding *clause-semantics-def*
by *force*
next
case *B*
then obtain *op*
where *op-in-*π_k: *op* ∈ *set* (*π* ! *k*)
and *v-is-add-effect*: *v* ∈ *set* (*add-effects-of op*)**..**
then have ¬(∃ *op* ∈ *set* (*π* ! *k*). *v* ∈ *set* (*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)
v
        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
    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
        qed
    qed
qed


lemma encode-problem-parallel-complete-iv:
  fixes Π::'a strips-problem
```

```
  assumes is-valid-problem-strips Π
    and is-parallel-solution-for-problem Π π
    and ∀ k op. k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
      ⟶ 𝒜 (Operator k (index (strips-problem.operators-of Π) op)) = (op ∈ set
(π ! k))
    and ∀ v k. k < length (trace-parallel-plan-strips ((Π)_I) π)
      ⟶ (𝒜 (State k (index (strips-problem.variables-of Π) v))
        ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True)
    and ∀ v l. l ≥ length (trace-parallel-plan-strips ((Π)_I) π) ∧ l < length π + 1
      ⟶ 𝒜 (State l (index (strips-problem.variables-of Π) v))
       = 𝒜 (State
         (length (trace-parallel-plan-strips ((Π)_I) π) − 1)
         (index (strips-problem.variables-of Π) v))
  shows 𝒜 ⊨ encode-all-frame-axioms Π (length π)
proof −
  let ?Φ_F = encode-all-frame-axioms Π (length π)
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
    and ?t = length π
  let ?A = ⋃ (⋃(k, v) ∈ {0..<?t} × set ((Π)_𝒱).
    {{{ (State k (index ?vs v))⁺, (State (Suc k) (index ?vs v))⁻¹ }
      ∪ { (Operator k (index ?ops op))⁺
        | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (add-effects-of op) }}})
    and ?B = ⋃ (⋃(k, v) ∈ {0..<?t} × set ((Π)_𝒱).
      {{{ (State k (index ?vs v))⁻¹, (State (Suc k) (index ?vs v))⁺ }
        ∪ { (Operator k (index ?ops op))⁺
          | op. op ∈ set ((Π)_𝒪) ∧ v ∈ set (delete-effects-of op) }}})

  have cnf-Φ_F-is-A-union-B: cnf ?Φ_F = ?A ∪ ?B
    using cnf-of-encode-all-frame-axioms-structure
    by (simp add: cnf-of-encode-all-frame-axioms-structure)
  {
    fix C
    assume C ∈ cnf ?Φ_F
    then consider (C-in-A) C ∈ ?A
      | (C-in-B) C ∈ ?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-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)_I) π) − 1
    and op ∈ set ((Π)_O)
  shows valuation-for-operator-variables Π π (trace-parallel-plan-strips ((Π)_I) 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)_I) 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
          moreover {
            have k'-lt: k' < length ?τ − 1
              using k'-op'-in
              by fastforce

            have op-in: op ∈ set (π ! k)
              using k-op-in
```

```
      by force

    then have op'-in: op' ∈ set (π ! k)
      using k'-op'-in k-is-k'
      by auto
    {
      have length-τ-gt-1: length ?τ > 1
        using assms(2)
        by linarith
      have length ?τ − Suc 0 ≤ length π + 1 − Suc 0
        using length-trace-parallel-plan-strips-lte-length-plan-plus-one
        using diff-le-mono
        by blast
      then have length ?τ − 1 ≤ length π
        by fastforce
      then have k' < length π
        using length-τ-gt-1 k'-lt
        by linarith
      hence π ! k' ∈ set π
        by simp
    }
    moreover have op ∈ set ?ops and op' ∈ set ?ops
      using is-parallel-solution-for-problem-operator-set[OF assms(1)] op-in
op'-in k-is-k'
       calculation
      by auto
    ultimately have op = op'
      using index-op-is-index-op'
      by force
  }
  ultimately show (k, op) = (k', op')
    by blast
 qed fast
}

hence inj-on ?f (set ?l)
  unfolding inj-on-def fst-def snd-def
  by fast
} note inj-on-f-set-l = this

{
 have set ?l = ⋃ (set ‘ set (map (λk. map (Pair k) (π ! k)) [0..<length ?τ −
1]))
   using set-concat
   by metis
 also have ... = ⋃ (set ‘ (λk. map (Pair k) (π ! k)) ‘ {0..<length ?τ − 1})
   by force
 also have ... = ⋃ ((λk. (Pair k) ‘ set (π ! k)) ‘ {0..<length ?τ − 1})
   by force
```

```
    also have ... = ⋃((λk. { (k, op) | op. op ∈ set (π ! k) }) ` {0..<length ?τ −
1})
      by blast
    also have ... = ⋃({{ (k, op) } | k op. k ∈ {0..<length ?τ − 1} ∧ op ∈ set
(π ! k) })
      by blast

    finally have set ?l = ⋃((λ(k, op). { (k, op) })
      ` { (k, op). k ∈ {0..<length ?τ − 1} ∧ op ∈ set (π ! 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 Π 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 , sow
we have to do some extra work to convince Isabelle of the truth of the statement.
      moreover {
        let ?g = λ-. True
        thm foldr-fun-upd[OF inj-on-f-set-l k-op-in]
        have ?v = Operator (fst (k, op)) (index ?ops (snd (k, op)))
          by simp
        moreover have (λ(k, op) A. A(Operator k (index ?ops op) := True))
          = (λx A. A(Operator (fst x) (index ?ops  (snd x)) := True))
          by fastforce
        moreover have foldr (λx A. A(Operator (fst x) (index ?ops  (snd x)) :=
?g x))
          ?l A0 (Operator (fst (k, op)) (index ?ops (snd (k, op)))) = True
          unfolding foldr-fun-upd[OF inj-on-f-set-l k-op-in]..
        ultimately have valuation-for-operator-variables Π π ?τ ?v = True
          using nb
          by argo
      }
      thus ?thesis
        using True
        by blast
```

```
    next
      case False
      {
        have (k, op) ∉ set ?l
          using False set-l-is
          by fast
        moreover {
          fix k' op'
          assume (k', op') ∈ set ?l
            and ?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 A₀ (Operator k (index ?ops op)) = False
          by simp
        moreover have (λ(k, op) A. A(Operator k (index ?ops op) := True))
          = (λx A. A(Operator (fst x) (index ?ops (snd x)) := True))
          by fastforce
       ultimately have foldr (λ(k, op) A. A(Operator k (index ?ops op) := True))
?l A₀ ?v = False
          using foldr-fun-no-upd[OF inj-on-f-set-l operator-not-in-f-image-set-l, of
λ-. True A₀]
          by presburger
      }
      thus ?thesis
      using nb False
      by blast
    qed
qed


lemma encode-problem-parallel-complete-vi-a:
  fixes Π :: 'a strips-problem
  assumes is-parallel-solution-for-problem Π π
    and k < length (trace-parallel-plan-strips ((Π)_I) π) − 1
  shows valuation-for-plan Π π (Operator k (index (strips-problem.operators-of Π)
op))
    = (op ∈ set (π ! k))
proof −
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
```

```
    and ?t = length π
    and ?τ = trace-parallel-plan-strips ((Π)_I) π
  let ?A_π = valuation-for-plan Π π
    and ?A_O = valuation-for-operator-variables Π π ?τ
    and ?Op = { Operator k (index ?ops op) | k op. k ∈ {0..<?t} ∧ op ∈ set ?ops
}
    and ?V = { State k (index ?vs v) | k v. k ∈ {0..<?t + 1} ∧ v ∈ set ?vs }
    and ?v = Operator k (index ?ops op)
  {
    have length ?τ ≤ length π + 1
      using length-trace-parallel-plan-strips-lte-length-plan-plus-one.
    then have length ?τ − 1 ≤ length π
      by simp
    then have k < ?t
      using assms
      by fastforce
  } note k-lt-length-π = this
  show ?thesis
    proof (cases op ∈ set ((Π)_O))
      case True
      {
        have ?v ∈ ?Op
          using k-lt-length-π True
          by auto

        hence ?A_π ?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 ¬index ?ops op < length ?ops due to the assumption that ¬op
∈ set ?ops. Hence ¬k ∈ {0..<?t and therefore ?v ∉ ?Op.
          have ?Op = (λ(k, op). Operator k (index ?ops op)) ‘ ({0..<?t} × set ?ops)
            by fast
          moreover have ¬index ?ops op < length ?ops
            using False
            by simp
          ultimately have ?v ∉ ?Op
            by fastforce
        }
        moreover have ?v ∉ ?V
          by force
```

```
      ultimately have ?\mathcal{A}_\pi ?v = \mathcal{A}_0 ?v
        unfolding valuation-for-plan-def override-on-def
        by metis
      hence \neg?\mathcal{A}_\pi ?v
        unfolding empty-valuation-def
        by blast
    }
    moreover have (\pi ! k) \in set \pi
      using k-lt-length-\pi
      by simp
    moreover have op \notin set (\pi ! k)
        using is-parallel-solution-for-problem-operator-set[OF assms(1) calcula-
tion(2)] False
      by blast
    ultimately show ?thesis
      by blast
  qed
qed


lemma encode-problem-parallel-complete-vi-b:
  fixes \Pi :: 'a strips-problem
  assumes is-parallel-solution-for-problem \Pi \pi
    and l \geq length (trace-parallel-plan-strips ((\Pi)_I) \pi) - 1
    and l < length \pi
  shows \neg valuation-for-plan \Pi \pi (Operator l (index (strips-problem.operators-of
\Pi) op))
proof -

  let ?vs = strips-problem.variables-of \Pi
    and ?ops = strips-problem.operators-of \Pi
    and ?t = length \pi
    and ?\tau = trace-parallel-plan-strips ((\Pi)_I) \pi
  let ?\mathcal{A}_\pi = valuation-for-plan \Pi \pi
    and ?\mathcal{A}_O = valuation-for-operator-variables \Pi \pi ?\tau
    and ?Op = { Operator k (index ?ops op) | k op. k \in {0..<?t} \wedge op \in set ?ops
}
    and ?Op' = { Operator k (index ?ops op) | k op. k \in {0..<length ?\tau - 1} \wedge
op \in set ?ops }
    and ?V = { State k (index ?vs v) | k v. k \in {0..<?t + 1} \wedge v \in set ?vs }
    and ?v = Operator l (index ?ops op)
  show ?thesis
    proof (cases op \in set ((\Pi)_\mathcal{O}))
      case True
      {
        {
          have ?v \in ?Op
            using assms(3) True
```

```
      by auto

    hence ?A_π ?v = ?A_O ?v
      unfolding valuation-for-plan-def override-on-def Let-def
      by simp
  }
  moreover {
    have l ∉ {0..<length ?τ − 1}
      using assms(2)
      by simp
    then have ?v ∉ ?Op'
      by blast
    hence ?A_O ?v = A_0 ?v
      unfolding valuation-for-operator-variables-def override-on-def
      by meson
  }
  ultimately have ¬?A_π ?v
    unfolding empty-valuation-def
    by blast
}
then show ?thesis
  by blast
next

  case False
  {
    {
      — We have ¬index ?ops op < length ?ops due to the assumption that ¬op
∈ set ?ops. Hence ¬k ∈ {0..<?t and therefore ?v ∉ ?Op.
      have ?Op = (λ(k, op). Operator k (index ?ops op)) ‘ ({0..<?t} × set ?ops)
        by fast
      moreover have ¬index ?ops op < length ?ops
        using False
        by simp
      ultimately have ?v ∉ ?Op
        by fastforce
    }
    moreover have ?v ∉ ?V
      by force

    ultimately have ?A_π ?v = A_0 ?v
      unfolding valuation-for-plan-def override-on-def
      by metis
    hence ¬?A_π ?v
      unfolding empty-valuation-def
      by blast
  }
  thus ?thesis
    by blast
```

qed
qed

— As a corollary from lemmas and we obtain the result that the constructed valuation $\mathcal{A} \equiv$ *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* π.
corollary *encode-problem-parallel-complete-vi-d*:

fixes Π :: *'variable strips-problem*
assumes *is-parallel-solution-for-problem* $\Pi\ \pi$
and $k <$ *length* π
and *op* $\notin$ *set* $(\pi\ !\ k)$
shows $\neg$*valuation-for-plan* $\Pi\ \pi$ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))
using *encode-problem-parallel-complete-vi-a*[*OF assms*(*1*)] *assms*(*3*)
encode-problem-parallel-complete-vi-b[*OF assms*(*1*) - *assms*(*2*)] *assms*(*3*)
by (*cases k <* length (*trace-parallel-plan-strips* $((\Pi)_I)\ \pi) - 1$; *fastforce*)

lemma *list-product-is-nil-iff*: *List.product xs ys* = [] $\longleftrightarrow$ *xs* = [] $\vee$ *ys* = []
proof (*rule iffI*)
assume *product-xs-ys-is-Nil*: *List.product xs ys* = []
show *xs* = [] $\vee$ *ys* = []
proof (*rule ccontr*)
assume $\neg$(*xs* = [] $\vee$ *ys* = [])
then have *xs* $\neq$ [] **and** *ys* $\neq$ []
by *simp+*
then obtain *x xs' y ys'* **where** *xs* = *x* # *xs'* **and** *ys* = *y* # *ys'*
using *list.exhaust*
by *metis*
then have *List.product xs ys* = (*x*, *y*) # *map* (*Pair x*) *ys'* @ *List.product xs'* (*y* # *ys'*)
by *simp*
thus *False*
using *product-xs-ys-is-Nil*
by *simp*
qed
next
assume *xs* = [] $\vee$ *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
qed
```

— We keep the state abstract by requiring a function s which takes the index k and returns state. This makes the lemma cover both cases, i.e. dynamic (e.g. the k-th trace state) as well as static state (e.g. final trace state).

```
lemma valuation-for-state-variables-is:
  assumes k ∈ 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
      assume x-in-ps: x ∈ set ?ps and y-in-ps: y ∈ set ?ps
        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''
        using x-is-not-y
        by blast
      hence False
        proof (cases)
          case A
          then have ?f x ≠ ?f y
            using x-is y-is
```

```
        by simp
      thus ?thesis
        using f-x-is-f-y
        by argo
    next
      case B
      have v' ∈ set vs and v'' ∈ set vs
        using x-in-ps x-is y-in-ps y-is set-product
        by blast+
      then have index vs v' ≠ index vs v''
        using B
        by force
      then have ?f x ≠ ?f y
        using x-is y-is
        by simp
      thus False
        using f-x-is-f-y
        by blast
    qed
  }
  hence inj-on ?f (set ?ps)
    using inj-on-def
    by blast
} note nb₂ = this
{
  have foldr (λx. valuation-for-state vs (s (fst x)) (fst x) (snd x))
   (List.product ks vs) 𝒜₀ (State (fst (k, v)) (index vs (snd (k, v)))) =
  (s (fst (k, v)) (snd (k, v)) = Some True)
    using foldr-fun-upd[OF nb₂ nb₁, of ?g 𝒜₀]
    by blast
  moreover have (λx. valuation-for-state vs (s (fst x)) (fst x) (snd x))
    = (λ(k, v). valuation-for-state vs (s k) k v)
    by fastforce
  ultimately have ?𝒜 (?f (k, v)) = ?g (k, v)
    by simp
}
thus ?thesis
  by simp
qed


lemma encode-problem-parallel-complete-vi-c:
  fixes Π :: 'a strips-problem
  assumes is-valid-problem-strips Π
    and is-parallel-solution-for-problem Π π
    and k < length (trace-parallel-plan-strips ((Π)_I) π)
  shows valuation-for-plan Π π (State k (index (strips-problem.variables-of Π) v))
    ⟷ (trace-parallel-plan-strips ((Π)_I) π ! k) v = Some True
proof −
```

```
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
    and ?τ = trace-parallel-plan-strips ((Π)_I) π
  let ?t = length π
    and ?t' = length ?τ
  let ?A_π = valuation-for-plan Π π
    and ?A_V = valuation-for-state-variables Π π ?τ
    and ?A_O = valuation-for-state-variables Π π ?τ
    and ?A_1 = foldr
      (λ(k, v) A. valuation-for-state ?vs (?τ ! k) k v A)
      (List.product [0..<?t'] ?vs) A_0
   and ?Op = { Operator k (index ?ops op) | k op. k ∈ {0..<?t} ∧ op ∈ set ((Π)_O)
}
    and ?Op' = { Operator k (index ?ops op) | k op. k ∈ {0..<?t' − 1} ∧ op ∈ set
((Π)_O) }
    and ?V = { State k (index ?vs v) | k v. k ∈ {0..<?t + 1} ∧ v ∈ set ((Π)_V) }
    and ?V_1 = { State k (index ?vs v) | k v. k ∈ {0..<?t'} ∧ v ∈ set ((Π)_V) }
    and ?V_2 = { State k (index ?vs v) | k v. k ∈ {?t'..(?t + 1)} ∧ v ∈ set ((Π)_V)
}
    and ?v = State k (index ?vs v)
  have v-notin-Op: ?v ∉ ?Op
    by blast
  have k-lte-length-π-plus-one: k < length π + 1
   using less-le-trans length-trace-parallel-plan-strips-lte-length-plan-plus-one assms(3)
    by blast
  show ?thesis
    proof (cases v ∈ set ((Π)_V))
      case True
      {

        {
          have ?v ∈ ?V ?v ∉ ?Op
            using k-lte-length-π-plus-one True
            by force+
          hence ?A_π ?v = ?A_V ?v
            unfolding valuation-for-plan-def override-on-def Let-def
            by simp
        }
        moreover {
          have ?v ∈ ?V_1 ?v ∉ ?V_2
            using assms(3) True
            by fastforce+
          hence ?A_V ?v = ?A_1 ?v
            unfolding valuation-for-state-variables-def override-on-def Let-def
            by force
        }
        ultimately have ?A_π ?v = ?A_1 ?v
          by blast
```

```
    }
    moreover have k ∈ set [0..<?t']
      using assms(3)
      by simp
    moreover have v ∈ set (strips-problem.variables-of Π)
      using True
      by simp

    ultimately show ?thesis
      using valuation-for-state-variables-is[of k [0..<?t']]
      by fastforce
  next
    case False
    {
      {
        have ¬ index ?vs v < length ?vs
          using False index-less-size-conv
          by simp
        hence ?v ∉ ?V
          by fastforce
      }
      then have ¬?A_π ?v
        using v-notin-Op
      unfolding valuation-for-plan-def override-on-def empty-valuation-def Let-def
          variables-of-def operators-of-def
        by presburger
    }
    moreover have ¬(?τ ! k) v = Some True
      using trace-parallel-plan-strips-none-if[of Π π k v] assms(1, 2, 3) False
      unfolding initial-of-def
      by force
    ultimately show ?thesis
      by blast
  qed
qed


lemma encode-problem-parallel-complete-vi-f:
  fixes Π :: 'a strips-problem
  assumes is-valid-problem-strips Π
    and is-parallel-solution-for-problem Π π
    and l ≥ length (trace-parallel-plan-strips ((Π)_I) π)
    and l < length π + 1
  shows valuation-for-plan Π π (State l (index (strips-problem.variables-of Π) v))
    = valuation-for-plan Π π
      (State (length (trace-parallel-plan-strips ((Π)_I) π) - 1)
      (index (strips-problem.variables-of Π) v))
proof -
```

```
  let ?vs = strips-problem.variables-of Π
    and ?ops = strips-problem.operators-of Π
    and ?τ = trace-parallel-plan-strips ((Π)_I) π
  let ?t = length π
    and ?t' = length ?τ
  let ?τ_Ω = ?τ ! (?t' − 1)
    and ?A_π = valuation-for-plan Π π
    and ?A_V = valuation-for-state-variables Π π ?τ
    and ?A_O = valuation-for-state-variables Π π ?τ
  let ?A_2 = foldr
    (λ(k, v) A. valuation-for-state (strips-problem.variables-of Π) ?τ_Ω k v A)
    (List.product [?t'..<length π + 2] ?vs)
    A_0
   and ?Op = { Operator k (index ?ops op) | k op. k ∈ {0..<?t} ∧ op ∈ set ((Π)_O)
}
   and ?Op' = { Operator k (index ?ops op) | k op. k ∈ {0..<?t' − 1} ∧ op ∈ set
((Π)_O) }
   and ?V = { State k (index ?vs v) | k v. k ∈ {0..<?t + 1} ∧ v ∈ set ((Π)_V) }
   and ?V_1 = { State k (index ?vs v) | k v. k ∈ {0..<?t'} ∧ v ∈ set ((Π)_V) }
   and ?V_2 = { State k (index ?vs v) | k v. k ∈ {?t'..(?t + 1)} ∧ v ∈ set ((Π)_V)
}
   and ?v = State l (index ?vs v)
  have v-notin-Op: ?v ∉ ?Op
    by blast
  show ?thesis
    proof (cases v ∈ set ((Π)_V))
      case True
      {

        {
          have ?v ∈ ?V ?v ∉ ?Op
            using assms(4) True
            by force+

          hence ?A_π ?v = ?A_V ?v
            unfolding valuation-for-plan-def override-on-def Let-def
            by simp
        }
        moreover {
          have ?v ∉ ?V_1 ?v ∈ ?V_2
            using assms(3, 4) True
            by force+

          hence ?A_V ?v = ?A_2 ?v
            unfolding valuation-for-state-variables-def override-on-def Let-def
            by auto
        }
        ultimately have ?A_π ?v = ?A_2 ?v
          by blast
```

```
  } note nb = this
  moreover
  {
    have l ∈ set [?t'..<?t + 2]
      using assms(3, 4)
      by auto

    hence ?A_2 ?v ⟷ ?τ_Ω v = Some True
      using valuation-for-state-variables-is[of l [?t'..<?t + 2]] True nb
      by fastforce
  }
  ultimately have ?A_π ?v ⟷ ?τ_Ω v = Some True
    by fast
  moreover {
    have 0 < ?t'
      using trace-parallel-plan-strips-not-nil
      by blast
    then have ?t' − 1 < ?t'
      using diff-less
      by presburger
  }
  ultimately show ?thesis
    using encode-problem-parallel-complete-vi-c[of - - ?t' − 1, OF assms(1, 2)]
    by blast
next
  case False
  {
    {
      have ¬ index ?vs v < length ?vs
        using False index-less-size-conv
        by auto
      hence ?v ∉ ?V
        by fastforce
    }
    then have ¬?A_π ?v
      using v-notin-Op
   unfolding valuation-for-plan-def override-on-def empty-valuation-def Let-def
       variables-of-def operators-of-def
      by presburger
  }
  moreover {
    have 0 < ?t'
      using trace-parallel-plan-strips-not-nil
      by blast
    then have ?t' − 1 < ?t'
      by simp
  }
  moreover have ¬((?τ ! (?t' − 1)) v = Some True)
      using trace-parallel-plan-strips-none-if[of - - ?t' − 1 v, OF - assms(2)
```

calculation(*2*)]
assms(*1*) *False*
by *simp*
ultimately show *?thesis*
using *encode-problem-parallel-complete-vi-c*[*of - - ?t′ − 1*, *OF assms*(*1*, *2*)]
by *blast*
qed
qed

Let now $\tau \equiv$ *trace-parallel-plan-strips* $I\ \pi$ be the trace of the plan π, $t \equiv$ *length* π, and $t' \equiv$ *length* τ.

Any model of the SATPlan encoding $\mathcal{A}$ must satisfy the following properties: [11]

1. for all k and for all op with $k < t' - (1::'a)$

   $$\mathcal{A}\ (\mathit{Operator}\ k\ (\mathit{index}\ (\mathit{operators\text{-}of}\ \Pi)\ op)) = op \in \mathit{set}\ (\pi\ !\ k)$$

2. for all l and for all op with $t' - (1::'a) \leq l$ and $l <$ *length* π we require

   $$\mathcal{A}\ (\mathit{Operator}\ l\ (\mathit{index}\ (\mathit{operators\text{-}of}\ \Pi)\ op))$$

3. for all v and for all k with $k < t'$ we require

   $$\mathcal{A}\ (\mathit{State}\ k\ (\mathit{index}\ (\mathit{variables\text{-}of}\ \Pi)\ v)) \longrightarrow ((\tau\ !\ k)\ v = \mathit{Some}\ \mathit{True})$$

4. and finally for all v and for all l with $t' \leq l$ and $l < t + (1::'a)$ we require

   $$\mathcal{A}\ (\mathit{State}\ l\ (\mathit{index}\ (\mathit{variables\text{-}of}\ \Pi)\ v)) = \mathcal{A}\ (\mathit{State}\ (t' - 1)\ (\mathit{index}\ (\mathit{variables\text{-}of}\ \Pi)\ v))$$

Condition "1." states that the model must reflect operator activation for all operators in the parallel operator lists $\pi\ !\ k$ of the plan π 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 τ). Meaning that $\mathcal{A}$ (*State* k (*index* $(\Pi_{\mathcal{V}})\ v)$) for the SAT plan variable of every state variable v at time point k if and only if $(\tau\ !\ k)\ v =$ *Some True* for the corresponding state $\tau\ !\ k$ at time k (and $\neg\ \mathcal{A}$ (*State* k (*index* $(\Pi_{\mathcal{V}})\ v)$) otherwise).

The second respectively fourth condition cover early plan termination by negating operator activation and propagating the last reached state. Note

[11] Cf. [3, Theorem 3.1, p. 1044] for the construction of $\mathcal{A}$.

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 *length* (*trace-parallel-plan-strips* (Π_I) π) $-$ *1* and the remaining states take up the indexes to *length* π + *1*.

value *stop*

— To show completeness—i.e. every valid parallel plan π corresponds to a model for the SATPlan encoding Φ Π (*length* π)—, 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* Π
and *is-parallel-solution-for-problem* Π π
shows *valuation-for-plan* Π π $\models$ Φ Π (*length* π)
proof −
let *?t* = *length* π
and *?I* = $(\Pi)_I$
and *?G* = $(\Pi)_G$
and *?A* = *valuation-for-plan* Π π
have *nb*: *?G* $\subseteq_m$ *execute-parallel-plan ?I* π
using *assms*(*2*)
unfolding *is-parallel-solution-for-problem-def*
by *force*
have *?A* $\models$ Φ_I Π
using *encode-problem-parallel-complete-i*[*OF assms*(*1*) *nb*]
encode-problem-parallel-complete-vi-c[*OF assms*(*1*, *2*)]
by *presburger*
moreover have *?A* $\models$ (Φ_G Π) *?t*
using *encode-problem-parallel-complete-ii*[*OF assms*(*1*) *nb*]
encode-problem-parallel-complete-vi-c[*OF assms*(*1*, *2*)]
encode-problem-parallel-complete-vi-f[*OF assms*(*1*, *2*)]
by *presburger*
moreover have *?A* $\models$ *encode-operators* Π *?t*
using *encode-problem-parallel-complete-iii*[*OF assms*(*1*) *nb*]
encode-problem-parallel-complete-vi-a[*OF assms*(*2*)]
encode-problem-parallel-complete-vi-b[*OF assms*(*2*)]
encode-problem-parallel-complete-vi-c[*OF assms*(*1*, *2*)]
by *presburger*
moreover have *?A* $\models$ *encode-all-frame-axioms* Π *?t*
using *encode-problem-parallel-complete-iv*[*OF assms*(*1*, *2*)]
encode-problem-parallel-complete-vi-a[*OF assms*(*2*)]
encode-problem-parallel-complete-vi-c[*OF assms*(*1*, *2*)]
encode-problem-parallel-complete-vi-f[*OF assms*(*1*, *2*)]
by *presburger*
ultimately show *?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

$$\neg(Atom\ (Operator\ k\ (index\ ops\ op_1))$$
$$\vee\ \neg(Atom\ (Operator\ k\ (index\ ops\ op_2))$$

for all pairs of distinct interfering operators op_1, op_2 for all time points $k < t$ for a given estimated plan length t. Definitions **??** and **??** implement the encoding for operator pairs resp. for all interfering operator pairs and all time points.

definition *encode-interfering-operator-pair-exclusion*
:: *'variable strips-problem*
⇒ *nat*
⇒ *'variable strips-operator*
⇒ *'variable strips-operator*
⇒ *sat-plan-variable formula*
where *encode-interfering-operator-pair-exclusion* Π *k* op_1 op_2
≡ *let ops* = *operators-of* Π *in*
¬(*Atom* (*Operator k* (*index ops* op_1)))
∨ ¬(*Atom* (*Operator k* (*index ops* op_2)))

definition *encode-interfering-operator-exclusion*
:: *'variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula*
where *encode-interfering-operator-exclusion* Π *t* ≡ *let*
ops = *operators-of* Π
; *interfering* = *filter* (λ(op_1, op_2). *index ops* op_1 ≠ *index ops* op_2
∧ *are-operators-interfering* op_1 op_2) (*List.product ops ops*)
in foldr (∧) [*encode-interfering-operator-pair-exclusion* Π *k* op_1 op_2.
(op_1, op_2) ← *interfering*, *k* ← [*0*..<*t*]] (¬⊥)

A SATPlan encoding with interfering operator pair exclusion can now be defined by simplying adding the conjunct *encode-interfering-operator-exclusion* Π *t* to the basic SATPlan encoding.

definition *encode-problem-with-operator-interference-exclusion*

:: *'variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula*
(Φ$_\forall$ - - *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 op*$_1$ *op*$_2$) = {{
(*Operator k* (*index* (*strips-problem.operators-of* Π) *op*$_1$))$^{-1}$
, (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*$_2$))$^{-1}$ }}
proof −
let *?ops* = *strips-problem.operators-of* Π
have *cnf* (*encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$)
= *cnf* (¬(*Atom* (*Operator k* (*index ?ops op*$_1$))) ∨ ¬(*Atom* (*Operator k* (*index ?ops op*$_2$))))
unfolding *encode-interfering-operator-pair-exclusion-def*
by *metis*
also have ... = { *C* ∪ *D* | *C D*.
C ∈ *cnf* (¬(*Atom* (*Operator k* (*index ?ops op*$_1$))))
∧ *D* ∈ *cnf* (¬(*Atom* (*Operator k* (*index ?ops op*$_2$)))) }
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 op*$_1$ *op*$_2$.
(*op*$_1$, *op*$_2$) ← *filter* (λ(*op*$_1$, *op*$_2$).
index (*strips-problem.operators-of* Π) *op*$_1$ ≠ *index* (*strips-problem.operators-of* Π) *op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$)
(*List.product* (*strips-problem.operators-of* Π) (*strips-problem.operators-of* Π))
, *k* ← [*0*..<*t*]]
= (⋃(*op*$_1$, *op*$_2$)
∈ { (*op*$_1$, *op*$_2$) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π).
index (*strips-problem.operators-of* Π) *op*$_1$ ≠ *index* (*strips-problem.operators-of* Π) *op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
(λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$) ` {*0*..<*t*})
proof −

let *?ops* = *strips-problem.operators-of* Π
let *?interfering* = *filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$) (*List.product ?ops ?ops*)
let *?fs* = [*encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$.
(*op*$_1$, *op*$_2$) ← *?interfering*, *k* ← [*0*..<*t*]]
have *set ?fs* = ⋃(*set*
‘(λ(*op*$_1$, *op*$_2$). *map* (λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$)
[*0*..<*t*])
‘(*set* (*filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$ ∧ *are-operators-interfering*
op$_1$ *op*$_2$)
(*List.product ?ops ?ops*))))
unfolding *set-concat set-map*
by *blast*
— TODO slow.
also have ... = ⋃((λ(*op*$_1$, *op*$_2$).
set (*map* (λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$) [*0*..<*t*]))
‘(*set* (*filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$ ∧ *are-operators-interfering*
op$_1$ *op*$_2$)
(*List.product ?ops ?ops*))))
unfolding *image-comp*[*of*
set λ(*op*$_1$, *op*$_2$). *map* (λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$
op$_2$) [*0*..<*t*]]
comp-apply
by *fast*
also have ... = ⋃((λ(*op*$_1$, *op*$_2$).
(λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$) ‘ {*0*..<*t*})
‘(*set* (*filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$ ∧ *are-operators-interfering*
op$_1$ *op*$_2$)
(*List.product ?ops ?ops*))))
unfolding *set-map*[*of* - [*0*..<*t*]] *atLeastLessThan-upt*[*of 0 t*]
by *blast*
also have ... = ⋃((λ(*op*$_1$, *op*$_2$).
(λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$) ‘ {*0*..<*t*})
‘(*Set.filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$ ∧ *are-operators-interfering*
op$_1$ *op*$_2$)
(*set* (*List.product ?ops ?ops*))))
unfolding *set-filter*[*of* λ(*op*$_1$, *op*$_2$). *are-operators-interfering op*$_1$ *op*$_2$ *List.product*
?ops ?ops]
by *force*
— TODO slow.
finally show *?thesis*
unfolding *operators-of-def set-product*[*of ?ops ?ops*]
by *fastforce*
qed

lemma *cnf-of-encode-interfering-operator-exclusion-is-iii*[*simp*]:

fixes Π :: *'variable strips-problem*

shows *cnf ‘ set* [*encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$.
(*op*$_1$, *op*$_2$) ← *filter* (λ(*op*$_1$, *op*$_2$).
index (*strips-problem.operators-of* Π) *op*$_1$ ≠ *index* (*strips-problem.operators-of* Π) *op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$)
(*List.product* (*strips-problem.operators-of* Π) (*strips-problem.operators-of* Π))
, *k* ← [*0*..<*t*]]
= (⋃(*op*$_1$, *op*$_2$)
∈ { (*op*$_1$, *op*$_2$) ∈ *set* (*strips-problem.operators-of* Π) × *set* (*strips-problem.operators-of* Π).
index (*strips-problem.operators-of* Π) *op*$_1$ ≠ *index* (*strips-problem.operators-of* Π) *op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
{{{ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*$_1$))$^{-1}$
, (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*$_2$))$^{-1}$ }} | *k*. *k* ∈ {*0*..<*t*}})
proof −
let *?ops* = *strips-problem.operators-of* Π
let *?interfering* = *filter* (λ(*op*$_1$, *op*$_2$). *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$) (*List.product ?ops ?ops*)
let *?fs* = [*encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$.
(*op*$_1$, *op*$_2$) ← *?interfering*, *k* ← [*0*..<*t*]]
have *cnf ‘ set ?fs* = *cnf ‘* (⋃(*op*$_1$, *op*$_2$) ∈ { (*op*$_1$, *op*$_2$).
(*op*$_1$, *op*$_2$) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π) ∧ *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
(λ*k*. *encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$) ‘ {*0*..<*t*})
unfolding *cnf-of-encode-interfering-operator-exclusion-is-ii*
by *blast*
also have ... = (⋃(*op*$_1$, *op*$_2$) ∈ { (*op*$_1$, *op*$_2$).
(*op*$_1$, *op*$_2$) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π) ∧ *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
(λ*k*. *cnf* (*encode-interfering-operator-pair-exclusion* Π *k op*$_1$ *op*$_2$)) ‘ {*0*..<*t*})
unfolding *image-Un image-comp comp-apply*
by *blast*
also have ... = (⋃(*op*$_1$, *op*$_2$) ∈ { (*op*$_1$, *op*$_2$).
(*op*$_1$, *op*$_2$) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π) ∧ *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
(λ*k*. {{ (*Operator k* (*index ?ops op*$_1$))$^{-1}$, (*Operator k* (*index ?ops op*$_2$))$^{-1}$ }})
‘ {*0*..<*t*})
by *simp*
also have ... = (⋃(*op*$_1$, *op*$_2$) ∈ { (*op*$_1$, *op*$_2$).
(*op*$_1$, *op*$_2$) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π) ∧ *index ?ops op*$_1$ ≠ *index ?ops op*$_2$
∧ *are-operators-interfering op*$_1$ *op*$_2$ }.
(λ*k*. {{ (*Operator k* (*index ?ops op*$_1$))$^{-1}$, (*Operator k* (*index ?ops op*$_2$))$^{-1}$ }})

```
      ‘ { k | k. k ∈ {0..<t}})
   by blast
 — TODO slow.
 finally show ?thesis
   unfolding operators-of-def setcompr-eq-image[of - λk. k ∈ {0..<t}]
   by force
qed

lemma cnf-of-encode-interfering-operator-exclusion-is:
 cnf (encode-interfering-operator-exclusion Π t) = ⋃(⋃(op₁, op₂)
     ∈ { (op₁, op₂) ∈ set (operators-of Π) × set (operators-of Π).
     index (strips-problem.operators-of Π) op₁ ≠ index (strips-problem.operators-of
Π) op₂
         ∧ are-operators-interfering op₁ op₂ }.
   {{{ (Operator k (index (strips-problem.operators-of Π) op₁))⁻¹
      , (Operator k (index (strips-problem.operators-of Π) op₂))⁻¹ }} | k. k ∈
{0..<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 Π k op₁ op₂.
   (op₁, op₂) ← ?interfering, k ← [0..<t]]
 have cnf (encode-interfering-operator-exclusion Π t) = cnf (foldr (∧) ?fs (¬⊥))
   unfolding encode-interfering-operator-exclusion-def
   by metis
 also have ... = ⋃(cnf ‘ set ?fs)
   unfolding cnf-foldr-and[of ?fs]..
 finally show ?thesis
   unfolding cnf-of-encode-interfering-operator-exclusion-is-iii[of Π t]
   by blast
qed

lemma cnf-of-encode-interfering-operator-exclusion-contains-clause-if:

 fixes Π :: 'variable strips-problem
 assumes k < t
  and op₁ ∈ set (strips-problem.operators-of Π) and op₂ ∈ 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 Π k op₁ op₂.
   (op₁, op₂) ← ?interfering, k ← [0..<t]]
  let ?Fs = (⋃(op₁, op₂)
     ∈ { (op₁, op₂) ∈ set (operators-of Π) × set (operators-of Π). are-operators-interfering
op₁ op₂ }.
     (λk. encode-interfering-operator-pair-exclusion Π k op₁ op₂) ‘ {0..<t})
  {
   fix f
   assume f ∈ set ?fs
   then have f ∈ ?Fs
     unfolding cnf-of-encode-interfering-operator-exclusion-is-ii
```

by *blast*
then obtain op_1 op_2
where $(op_1,\ op_2) \in$ *set* (*operators-of* Π) $\times$ *set* (*operators-of* Π)
and *are-operators-interfering* op_1 op_2
and $f \in (\lambda k.$ *encode-interfering-operator-pair-exclusion* Π k op_1 $op_2)$ ‘ $\{0..<t\}$
by *fast*
then obtain k **where** $f =$ *encode-interfering-operator-pair-exclusion* Π k op_1 op_2
by *blast*
then have $f = \neg($*Atom* (*Operator* k (*index ?ops* op_1))) $\vee$ $\neg($*Atom* (*Operator* k (*index ?ops* op_2)))
unfolding *encode-interfering-operator-pair-exclusion-def*
by *metis*
hence *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 *is-cnf-encode-problem-with-operator-interference-exclusion*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* ($\Phi_\forall$ Π t)
using *is-cnf-encode-problem is-cnf-encode-interfering-operator-exclusion assms*
unfolding *encode-problem-with-operator-interference-exclusion-def SAT-Plan-Base.encode-problem-def is-cnf.simps*(*1*)
by *blast*

lemma *cnf-of-encode-problem-with-operator-interference-exclusion-structure*:
shows *cnf* (Φ_I Π) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* ((Φ_G Π) t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-operators* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-all-frame-axioms* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-interfering-operator-exclusion* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
unfolding *encode-problem-with-operator-interference-exclusion-def encode-problem-def SAT-Plan-Base.encode-problem-def*
encode-initial-state-def
encode-goal-state-def
by *auto+*

lemma *encode-problem-with-operator-interference-exclusion-has-model-then-also-partial-encodings*:
assumes $\mathcal{A} \models \Phi_\forall$ Π t
shows $\mathcal{A} \models$ *SAT-Plan-Base.encode-initial-state* Π
and $\mathcal{A} \models$ *SAT-Plan-Base.encode-operators* Π t
and $\mathcal{A} \models$ *SAT-Plan-Base.encode-all-frame-axioms* Π t
and $\mathcal{A} \models$ *encode-interfering-operator-exclusion* Π t

and $\mathcal{A} \models$ *SAT-Plan-Base.encode-goal-state* Π 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).

lemma *encode-problem-serializable-sound-i*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi_\forall$ Π t
and $k < t$
and *ops* $\in$ *set* (*subseqs* ((Φ^{-1} Π $\mathcal{A}$ t) ! k))
shows *are-all-operators-non-interfering ops*
proof −
let *?ops* = *strips-problem.operators-of* Π
and *?*π = Φ^{-1} Π $\mathcal{A}$ t
and *?*Φ_X = *encode-interfering-operator-exclusion* Π t
let *?*π_k = (Φ^{-1} Π $\mathcal{A}$ t) ! k

{
fix C
assume *C-in*: $C \in$ *cnf ?*Φ_X
have *cnf-semantics* $\mathcal{A}$ (*cnf ?*Φ_X)
using *cnf-semantics-monotonous-in-cnf-subsets-if*[*OF assms*(*2*)
is-cnf-encode-problem-with-operator-interference-exclusion[*OF assms*(*1*)]
cnf-of-encode-problem-with-operator-interference-exclusion-structure(*5*)].
hence *clause-semantics* $\mathcal{A}$ C
unfolding *cnf-semantics-def*
using *C-in*
by *fast*

```
  } note nb_1 = this
  {
    fix op_1 op_2
    assume op_1 ∈ set ?π_k and op_2 ∈ set ?π_k
      and index-op_1-is-not-index-op_2: index ?ops op_1 ≠ index ?ops op_2
   moreover have op_1-in: op_1 ∈ set ?ops and 𝒜-models-op_1:𝒜 (Operator k (index
?ops op_1))
      and op_2-in: op_2 ∈ set ?ops and 𝒜-models-op_2: 𝒜 (Operator k (index ?ops
op_2))
      using decode-plan-step-element-then[OF assms(3)] calculation
      unfolding decode-plan-def
      by blast+
    moreover {
     let ?C = { (Operator k (index ?ops op_1))^-1, (Operator k (index ?ops op_2))^-1
}
      assume are-operators-interfering op_1 op_2
      moreover have ?C ∈ cnf ?Φ_X
        using cnf-of-encode-interfering-operator-exclusion-contains-clause-if[OF
            assms(3) op_1-in op_2-in index-op_1-is-not-index-op_2] calculation
        by blast
      moreover have ¬clause-semantics 𝒜 ?C
        using 𝒜-models-op_1 𝒜-models-op_2
        unfolding clause-semantics-def
        by auto
      ultimately have False
        using nb_1
        by blast
    }
    ultimately have ¬are-operators-interfering op_1 op_2
      by blast
  } note nb_3 = this
  show ?thesis
    using assms
    proof (induction ops)
      case (Cons op_1 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 op_2
        assume op_2-in-ops: op_2 ∈ set ops
        moreover have op_1-in-π_k: op_1 ∈ set ?π_k and op_2-in-π_k: op_2 ∈ set ?π_k
          using element-of-subseqs-then-subset[OF Cons.prems(4)] calculation(1)
          by auto+
        moreover
        {
          have distinct (op_1 # ops)
            using subseqs-distinctD[OF Cons.prems(4)]
              decode-plan-step-distinct[OF Cons.prems(3)]
```

```
        unfolding decode-plan-def
        by blast
      moreover have op₁ ∈ set ?ops and op₂ ∈ set ?ops
        using decode-plan-step-element-then(1)[OF Cons.prems(3)] op₁-in-π_k
op₂-in-π_k
        unfolding decode-plan-def
        by force+
      moreover have op₁ ≠ op₂
        using op₂-in-ops calculation(1)
        by fastforce
      ultimately have index ?ops op₁ ≠ index ?ops op₂
        using index-eq-index-conv
        by auto
    }
    ultimately have ¬are-operators-interfering op₁ op₂
      using nb₃
      by blast
  }
  ultimately show ?case
    using list-all-iff
    by auto
 qed simp
qed

theorem encode-problem-serializable-sound:
  assumes is-valid-problem-strips Π
    and 𝒜 ⊨ Φ∀ 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
      and 𝒜 ⊨ SAT-Plan-Base.encode-operators Π t
      and 𝒜 ⊨ SAT-Plan-Base.encode-all-frame-axioms Π t
      and 𝒜 ⊨ SAT-Plan-Base.encode-goal-state Π t
      using assms(2)
      unfolding encode-problem-with-operator-interference-exclusion-def
      by simp+
    then have 𝒜 ⊨ SAT-Plan-Base.encode-problem Π t
      unfolding SAT-Plan-Base.encode-problem-def
      by simp
  }
  thus is-parallel-solution-for-problem Π (Φ⁻¹ Π 𝒜 t)
    using encode-problem-parallel-sound assms(1, 2)
    unfolding decode-plan-def
    by blast
next
  let ?π = Φ⁻¹ Π 𝒜 t
```

```
  {
    fix k
    assume k < t
    moreover have ?π ! k ∈ set (subseqs (?π ! k))
      using subseqs-refl
      by blast
    ultimately have are-all-operators-non-interfering (?π ! k)
      using encode-problem-serializable-sound-i[OF assms]
      unfolding SAT-Plan-Base.decode-plan-def decode-plan-def
      by blast
  }
  moreover have length ?π = t
    unfolding SAT-Plan-Base.decode-plan-def decode-plan-def
    by simp
  ultimately show ∀ k < length ?π. are-all-operators-non-interfering (?π ! k)
    by simp
qed
```

8.2 Completeness

```
lemma encode-problem-with-operator-interference-exclusion-complete-i:
  assumes is-valid-problem-strips Π
    and is-parallel-solution-for-problem Π π
    and ∀ k < length π. are-all-operators-non-interfering (π ! k)
  shows valuation-for-plan Π π ⊨ encode-interfering-operator-exclusion Π (length
π)
proof −
  let ?A = valuation-for-plan Π π
    and ?Φ_X = encode-interfering-operator-exclusion Π (length π)
    and ?ops = strips-problem.operators-of Π
    and ?t = length π
  let ?τ = trace-parallel-plan-strips ((Π)_I) π
  let ?Ops = { (op_1, op_2). (op_1, op_2) ∈ set (operators-of Π) × set (operators-of
Π)
    ∧ index ?ops op_1 ≠ index ?ops op_2
    ∧ are-operators-interfering op_1 op_2 }
    and ?f = λ(op_1, op_2). {{{ (Operator k (index ?ops op_1))^-1, (Operator k (index
?ops op_2))^-1 }}
      | k. k ∈ {0..<length π} }
  let ?A = ⋃(?f ‘ ?Ops)
  let ?B = ⋃ ?A
  have nb_1: ∀ 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** nb_2 = *this*
{
fix *k* op_1 op_2
assume *k* < *length* π
and op_1 ∈ *set* (π ! *k*)
and *index ?ops* op_1 ≠ *index ?ops* op_2
and *are-operators-interfering* op_1 op_2
moreover have *are-all-operators-non-interfering* (π ! *k*)
using *assms*(*3*) *calculation*(*1*)
by *blast*
moreover have op_1 ≠ op_2
using *calculation*(*3*)
by *blast*
ultimately have op_2 ∉ *set* (π ! *k*)
using *are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs*
assms(*3*)
by *blast*
} **note** nb_3 = *this*
{
fix *C*
assume *C* ∈ *cnf* ?Φ_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 op_1 op_2 **where** (op_1, op_2) ∈ *set* (*operators-of* Π) × *set* (*operators-of* Π)
and *index-op*$_1$*-is-not-index-op*$_2$: *index ?ops* op_1 ≠ *index ?ops* op_2
and *are-operators-interfering-op*$_1$*-op*$_2$: *are-operators-interfering* op_1 op_2
and *C′-in*: *C′* ∈ {{{(*Operator k* (*index ?ops* op_1))$^{-1}$, (*Operator k* (*index ?ops* op_2))$^{-1}$}}
| *k*. *k* ∈ {*0*..<*length* π}}
using *UN-iff*[*of C′ ?f ?Ops*]
by *blast*
then obtain *k* **where** *k* ∈ {*0*..<*length* π}
and *C-is*: *C* = { (*Operator k* (*index ?ops* op_1))$^{-1}$, (*Operator k* (*index ?ops* op_2))$^{-1}$ }
using *C-in C′-in*
by *blast*
then have *k-lt-length-π*: *k* < *length* π
by *simp*
consider (*A*) op_1 ∈ *set* (π ! *k*)
| (*B*) op_2 ∈ *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)
   using nb₃ k-lt-length-π calculation index-op₁-is-not-index-op₂ are-operators-interfering-op₁-op₂
      by blast
    moreover have ¬ ?A (Operator k (index ?ops op₂))
      using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length-π]
        calculation(2)
      by blast
    ultimately show ?thesis
      using C-is
      unfolding clause-semantics-def
      by force
  next
    case B
    moreover have op₁ ∉ set (π ! k)
   using nb₃ k-lt-length-π calculation index-op₁-is-not-index-op₂ are-operators-interfering-op₁-op₂
      by blast
    moreover have ¬ ?A (Operator k (index ?ops op₁))
      using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length-π]
        calculation(2)
      by blast
    ultimately show ?thesis
      using C-is
      unfolding clause-semantics-def
      by force
  next
    case C
    then show ?thesis
      proof (rule disjE)
        assume op₁ ∉ set (π ! k)
        then have ¬ ?A (Operator k (index ?ops op₁))
         using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length-π]
          by blast
        thus clause-semantics (valuation-for-plan Π π) C
          using C-is
          unfolding clause-semantics-def
          by force
      next
        assume op₂ ∉ set (π ! k)
        then have ¬ ?A (Operator k (index ?ops op₂))
         using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length-π]
          by blast
        thus clause-semantics (valuation-for-plan Π π) C
          using C-is
          unfolding clause-semantics-def
```

```
          by force
        qed
      qed
  }
  then have cnf-semantics ?A (cnf ?Φ_X)
    unfolding cnf-semantics-def..
  thus ?thesis
   using cnf-semantics[OF is-nnf-cnf[OF is-cnf-encode-interfering-operator-exclusion]]
    by fast
qed
```

Similar to the soundness proof, we may reuse the previously established facts about the valuation for the completeness proof of the basic SATPlan encoding (**??**). To make it clearer why this is true we have a look at the form of the clauses for interfering operator pairs op_1 and op_2 at the same time index k which have the form shown below:

$$\{\ (\mathit{Operator}\ k\ (\mathit{index}\ \mathit{ops}\ \mathit{op}_1))^{-1},\ (\mathit{Operator}\ k\ (\mathit{index}\ \mathit{ops}\ \mathit{op}_2))^{-1}\ \}$$

where $\mathit{ops} \equiv \Pi_{\mathcal{O}}$. Now, consider an operator op_1 that is contained in the k-th plan step $\pi\ !\ k$ (symmetrically for op_2). Since π 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 $\mathcal{A} \equiv \mathit{valuation\text{-}for\text{-}plan}\ \Pi\ \pi$ it holds that

$$\mathcal{A} \models \neg\ \mathit{Atom}\ (\mathit{Operator}\ k\ (\mathit{index}\ \mathit{ops}\ \mathit{op}_2))$$

and $\mathcal{A}$ therefore models the clause.

Furthermore, if neither is present, than $\mathcal{A}$ 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 $\mathcal{A}$. 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
    and ∀ k < length π. are-all-operators-non-interfering (π ! k)
  shows valuation-for-plan Π π ⊨ Φ_∀ Π (length π)
proof −
  let ?A = valuation-for-plan Π π
    and ?Φ_X = encode-interfering-operator-exclusion Π (length π)
  have ?A ⊨ SAT-Plan-Base.encode-problem Π (length π)
    using assms(1, 2) encode-problem-parallel-complete
    by auto
  moreover have ?A ⊨ ?Φ_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

value stop


lemma encode-problem-forall-step-decoded-plan-is-serializable-i:
 assumes is-valid-problem-strips Π
   and 𝒜 ⊨ Φ∀ Π t
 shows (Π)G ⊆m execute-serial-plan ((Π)I) (concat (Φ⁻¹ Π 𝒜 t))
proof −
 let ?G = (Π)G
   and ?I = (Π)I
   and ?π = Φ⁻¹ Π 𝒜 t
 let ?π' = concat (Φ⁻¹ Π 𝒜 t)
   and ?τ = trace-parallel-plan-strips ?I ?π
   and ?σ = map (decode-state-at Π 𝒜) [0..<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 (Φ⁻¹ Π 𝒜 t))
proof −
 let ?π = Φ⁻¹ Π 𝒜 t
 let ?π' = concat ?π

 {
  have set ?π' = ⋃(set ‘ (⋃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 set ?π' = (⋃k < t. set (decode-plan' Π 𝒜 k))
   by blast
 } note nb = this
 {
  fix op
  assume op ∈ set ?π'
  then obtain k where k < t and op ∈ set (decode-plan' Π 𝒜 k)
   using nb
   by blast
  moreover have op ∈ set (decode-plan Π 𝒜 t ! k)
   using calculation
   unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
   by simp
  ultimately have op ∈ set (operators-of Π)
```

using *decode-plan-step-element-then*(*1*)
unfolding *operators-of-def decode-plan-def*
by *blast*
}
thus *?thesis*
unfolding *list-all-iff ListMem-iff operators-of-def*
by *blast*
qed

Given the soundness and completeness of the SATPlan encoding with interfering operator exclusion $\Phi_\forall\ \Pi\ t$, we can now conclude this part with showing that for a parallel plan $\pi \equiv \Phi^{-1}\ \Pi\ \mathcal{A}\ t$ that was decoded from a model $\mathcal{A}$ of $\Phi_\forall\ \Pi\ t$ the serialized plan $\pi' \equiv concat\ \pi$ is a serial solution for Π. To this end, we have to show that

- the state reached by serial execution of π' subsumes G, and
- all operators in π' are operators contained in $\mathcal{O}$.

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 π and therefore have to show that G is subsumed by the last state of the trace of π'

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

and moreover that at every step of the parallel plan execution, the parallel operator execution condition as well as non interference are met

$$\forall\, k < length\ \pi.\ are\text{-}all\text{-}operators\text{-}non\text{-}interfering\ (\pi\ !\ k)$$

. [12] Note that the parallel operator execution condition is implicit in the existence of the parallel trace for π with

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

warranted by the soundness of $\Phi_\forall\ \Pi\ t$.

theorem *serializable-encoding-decoded-plan-is-serializable*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi_\forall\ \Pi\ t$
shows *is-serial-solution-for-problem* Π (*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*

[12] These propositions are shown in lemmas `encode_problem_forall_step_decoded_plan_is_serializable_ii` and `encode_problem_forall_step_decoded_plan_is_serializable_i` which have been omitted for brevity.

initial-of-def decode-plan-def
by *blast*

end

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* Ψ
and $\mathcal{A} \models \Phi_\forall\ (\varphi\ \Psi)\ t$
shows *is-serial-solution-for-problem* Ψ $[{\varphi_O}^{-1}\ \Psi\ op.\ op \leftarrow concat\ (\Phi^{-1}\ (\varphi\ \Psi)\ \mathcal{A}\ t)]$
proof −
let $?\Pi = \varphi\ \Psi$
and $?\pi' = concat\ (\Phi^{-1}\ (\varphi\ \Psi)\ \mathcal{A}\ t)$
let $?\psi = [{\varphi_O}^{-1}\ \Psi\ op.\ op \leftarrow ?\pi']$
{
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-iff-ii*:
assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* $\Psi\ \psi$
and $h = length\ \psi$
shows $\exists \mathcal{A}.\ (\mathcal{A} \models \Phi_\forall\ (\varphi\ \Psi)\ h)$
proof −
let $?\Pi = \varphi\ \Psi$
and $?\pi = \varphi_P\ \Psi\ (embed\ \psi)$
let $?\mathcal{A} = valuation\text{-}for\text{-}plan\ ?\Pi\ ?\pi$
let $?t = length\ \psi$

```
  have nb: length ψ = length ?π
    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 ψ; auto)
  have is-valid-problem-strips ?Π
    using assms(1) is-valid-problem-sas-plus-then-strips-transformation-too
    by blast
  moreover have STRIPS-Semantics.is-parallel-solution-for-problem ?Π ?π
   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 < length ?π
    moreover obtain ops' where ops' = ?π ! k
      by simp
    moreover have ops' ∈ set ?π
      using calculation nth-mem
      by blast
    moreover have ?π = [[φ_O Ψ op. op ← ops]. ops ← embed ψ]
      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' = [φ_O Ψ op. op ← ops]
        and ops ∈ set (embed ψ)
      using calculation(3, 4)
      by auto
    moreover have ops ∈ { [op] | op. op ∈ set ψ }
      using calculation(6) set-of-embed-is
      by blast
    moreover obtain op
      where ops = [op] and op ∈ set ψ
      using calculation(7)
      by blast
    ultimately have are-all-operators-non-interfering (?π ! 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 Ψ can be solved using the planning as satisfiability framework.

A solution ψ for the SAS+ problem Ψ 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 $\Phi_\forall\ \varphi\ \Psi\quad t$ exist.

theorem *sas-plus-problem-has-serial-solution-iff*:
assumes *is-valid-problem-sas-plus* Ψ
shows $(\exists \psi.$ *is-serial-solution-for-problem* $\Psi\ \psi) \longleftrightarrow (\exists \mathcal{A}\ t.\ \mathcal{A} \models \Phi_\forall\ (\varphi\ \Psi)\ t)$
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* ≡ (|*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 Π ≡
(|*SAS-Plus-Representation.sas-plus-problem.variables-of* = *SAS-Plus-Representation.sas-plus-problem.varia*
Π,
SAS-Plus-Representation.sas-plus-problem.operators-of = *empty-sasp-action*
SAS-Plus-Representation.sas-plus-problem.operators-of Π,
SAS-Plus-Representation.sas-plus-problem.initial-of = *SAS-Plus-Representation.sas-plus-problem.initial-of*
Π,
SAS-Plus-Representation.sas-plus-problem.goal-of = *SAS-Plus-Representation.sas-plus-problem.goal-of*
Π,
SAS-Plus-Representation.sas-plus-problem.range-of = *SAS-Plus-Representation.sas-plus-problem.range-of*
Π|)

lemma *sasp-noops-in-noop-problem*: *set* (*replicate n empty-sasp-action*) ⊆ *set* (*SAS-Plus-Representation.sas-pl*
(*prob-with-noop* Π))
by (*induction n*) (*auto simp*: *prob-with-noop-def*)

lemma *noops-complete*:
SAS-Plus-Semantics.is-serial-solution-for-problem Ψ π $\Longrightarrow$
SAS-Plus-Semantics.is-serial-solution-for-problem (*prob-with-noop* Ψ) ((*replicate* *n empty-sasp-action*) @ π)
by(*induction n*)
(*auto simp*: *SAS-Plus-Semantics.is-serial-solution-for-problem-def insert list.pred-set*
sasp-exec-noops prob-with-noop-def Let-def empty-sasp-action-def
elem)

definition *rem-noops* $\equiv$ *filter* (λ*op*. *op* $\neq$ *empty-sasp-action*)

lemma *sasp-filter-empty-action*:
execute-serial-plan-sas-plus s (*rem-noops* π*s*) = *execute-serial-plan-sas-plus s* π*s*
by (*induction* π*s arbitrary*: *s*)
(*auto simp*: *empty-sasp-action-def rem-noops-def*)

lemma *noops-sound*:
SAS-Plus-Semantics.is-serial-solution-for-problem (*prob-with-noop* Ψ) π*s* $\Longrightarrow$
SAS-Plus-Semantics.is-serial-solution-for-problem Ψ (*rem-noops* π*s*)
by(*induction* π*s*)
(*fastforce simp*: *SAS-Plus-Semantics.is-serial-solution-for-problem-def insert*
list.pred-set
prob-with-noop-def ListMem-iff rem-noops-def
sasp-filter-empty-action[*unfolded empty-sasp-action-def rem-noops-def*]
empty-sasp-action-def)+

lemma *noops-valid*: *is-valid-problem-sas-plus* Ψ $\Longrightarrow$ *is-valid-problem-sas-plus* (*prob-with-noop* Ψ)
by (*auto simp*: *is-valid-problem-sas-plus-def prob-with-noop-def Let-def*
empty-sasp-action-def is-valid-operator-sas-plus-def list.pred-set)

lemma *sas-plus-problem-has-serial-solution-iff-i'*:
assumes *is-valid-problem-sas-plus* Ψ
and $\mathcal{A}$ $\models$ $\Phi_\forall$ (φ (*prob-with-noop* Ψ)) *t*
shows *SAS-Plus-Semantics.is-serial-solution-for-problem* Ψ
(*rem-noops*
(*map* (λ*op*. ${\varphi_O}^{-1}$ (*prob-with-noop* Ψ) *op*)
(*concat* (Φ^{-1} (φ (*prob-with-noop* Ψ)) $\mathcal{A}$ *t*))))
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* Ψ
and *SAS-Plus-Semantics.is-serial-solution-for-problem* Ψ ψ
and *length* ψ $\leq$ *h*
shows $\exists \mathcal{A}$. ($\mathcal{A}$ $\models$ $\Phi_\forall$ (φ (*prob-with-noop* Ψ)) *h*)
using *assms*
by(*fastforce*
intro!: *assms noops-valid noops-complete*

sas-plus-problem-has-serial-solution-iff-ii
[**where** ψ = (*replicate* (*h* − *length* ψ) *empty-sasp-action*) @ ψ])
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 represnetation 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* ≡ λ(*pre*, -, -, -). *pre* = []

definition *is-standard-operator* :: *ast-operator* ⇒ *bool*
where *is-standard-operator* ≡ λ(-, -, *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*)

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*

$x) \Longrightarrow$
$m1 \mathbin{++} m2 \subseteq_m x \Longrightarrow m1 \subseteq_m x$
using *map-add-le-mapE map-add-comm*
by *force*

lemma *subsuming-states-map-add*:
$(\bigwedge x.\ x \in dom\ m1 \cap dom\ m2 \Longrightarrow m1\ x = m2\ x) \Longrightarrow$
$m1 \mathbin{++} m2 \subseteq_m s \longleftrightarrow (m1 \subseteq_m s \wedge m2 \subseteq_m s)$
by(*auto simp*: *map-add-le-mapI intro*: *first-map-add-submap map-add-le-mapE*)

lemma *consistent-map-lists*:
$[\![distinct\ (map\ fst\ (xs1 @ xs2));\ x \in dom\ (map\text{-}of\ xs1) \cap dom\ (map\text{-}of\ xs2)]\!] \Longrightarrow$

$(map\text{-}of\ xs1)\ x = (map\text{-}of\ xs2)\ x$
apply(*induction xs1*)
apply (*simp-all add*: *consistent-map-lists-def image-def*)
using *map-of-SomeD*
by *fastforce*

lemma *subsuming-states-append*:
$distinct\ (map\ fst\ (xs @ ys)) \Longrightarrow$
$(map\text{-}of\ (xs @ ys)) \subseteq_m s \longleftrightarrow ((map\text{-}of\ ys) \subseteq_m s \wedge (map\text{-}of\ xs) \subseteq_m s)$
unfolding *map-of-append*
apply(*intro subsuming-states-map-add*)
apply (*auto simp add*: *image-def*)
by (*metis* (*mono-tags*, *lifting*) *IntI empty-iff fst-conv mem-Collect-eq*)

definition *consistent-pres-op* **where**
$consistent\text{-}pres\text{-}op\ op \equiv (case\ op\ of\ (name, pres, effs, cost) \Rightarrow$
$distinct\ (map\ fst\ (pres @ (implicit\text{-}pres\ effs)))$
$\wedge\ consistent\text{-}map\text{-}lists\ pres\ (implicit\text{-}pres\ effs))$

definition *consistent-pres-op′* **where**
$consistent\text{-}pres\text{-}op'\ op \equiv (case\ op\ of\ (name, pres, effs, cost) \Rightarrow$
$consistent\text{-}map\text{-}lists\ pres\ (implicit\text{-}pres\ effs))$

lemma *consistent-pres-op-then′*: $consistent\text{-}pres\text{-}op\ op \Longrightarrow consistent\text{-}pres\text{-}op'\ op$
by(*auto simp add*: *consistent-pres-op′-def consistent-pres-op-def*)

lemma *rem-implicit-pres-ops-valid-states*:
$ast\text{-}problem.valid\text{-}states\ (rem\text{-}implicit\text{-}pres\text{-}ops\ prob) = ast\text{-}problem.valid\text{-}states\ prob$
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*:
$ast\text{-}problem.lookup\text{-}operator\ (vars, init, goal, ops)\ name = None \longleftrightarrow$

ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name*
= *None*
by (*induction ops*) (*auto simp*: *ast-problem.lookup-operator-def ast-problem.astδ-def*)

lemma *rem-implicit-pres-ops-lookup-op-Some-1*:
ast-problem.lookup-operator (*vars*, *init*, *goal*, *ops*) *name* = *Some* (*n*,*p*,*vp*,*e*) ⟹
ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name*
=
Some (*rem-implicit-pres* (*n*,*p*,*vp*,*e*))
by (*induction ops*) (*fastforce simp*: *ast-problem.lookup-operator-def ast-problem.astδ-def*)+

lemma *rem-implicit-pres-ops-lookup-op-Some-1′*:
ast-problem.lookup-operator prob name = *Some* (*n*,*p*,*vp*,*e*) ⟹
ast-problem.lookup-operator (*rem-implicit-pres-ops prob*) *name* =
Some (*rem-implicit-pres* (*n*,*p*,*vp*,*e*))
apply(*cases prob*)
using *rem-implicit-pres-ops-lookup-op-Some-1*
by *simp*

lemma *implicit-pres-empty*: *implicit-pres* (*map rem-effect-implicit-pres effs*) = []
by (*induction effs*) (*auto simp*: *implicit-pres-def*)

lemma *rem-implicit-pres-ops-lookup-op-Some-2*:
ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name*
= *Some op*
⟹ ∃ *op′*. *ast-problem.lookup-operator* (*vars*, *init*, *goal*, *ops*) *name* = *Some op′*
∧
(*op* = *rem-implicit-pres op′*)
by (*induction ops*) (*auto simp*: *ast-problem.lookup-operator-def ast-problem.astδ-def*
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 ∈ *ast-problem.subsuming-states prob ps* = (*s* ∈ (*ast-problem.valid-states prob*)
∧ *ps* ⊆*m s*)
by (*auto simp add*: *ast-problem.subsuming-states-def*)

lemma *rem-implicit-pres-ops-enabled-1*:
⟦(⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*);
ast-problem.enabled prob name s⟧ ⟹
ast-problem.enabled (*rem-implicit-pres-ops prob*) *name s*
by (*fastforce simp*: *ast-problem.enabled-def rem-implicit-pres-ops-valid-states sub-*

suming-states-def'
implicit-pres-empty
intro!: *map-add-le-mapI*
dest: *rem-implicit-pres-ops-lookup-op-Some-1'*
split: *option.splits*)+

context *ast-problem*
begin

lemma *lookup-Some-inδ*: *lookup-operator π = Some op ⟹ op∈set astδ*
by(*auto simp*: *find-Some-iff in-set-conv-nth lookup-operator-def*)

end

lemma *rem-implicit-pres-ops-enabled-2*:
assumes (⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*)
shows *ast-problem.enabled* (*rem-implicit-pres-ops prob*) *name s* ⟹
ast-problem.enabled prob name s
using *assms*[*OF ast-problem.lookup-Some-inδ*, *unfolded consistent-pres-op-def*]
apply(*auto simp*: *subsuming-states-append rem-implicit-pres-ops-valid-states subsuming-states-def'*
ast-problem.enabled-def
dest!: *rem-implicit-pres-ops-lookup-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*:
(⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*) ⟹
ast-problem.enabled (*rem-implicit-pres-ops prob*) *name s* = *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*]:
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*)

end

lemma *is-standard-operator-rem-implicit*: *is-standard-operator* (*n*,*p*,*vp*,*v*) ⟹
is-standard-operator (*rem-implicit-pres* (*n*,*p*,*vp*,*v*))
by (*induction vp*) (*auto simp*: *is-standard-operator-def is-standard-effect-def*)

lemma *is-standard-operator-rem-implicit-pres-ops*:
⟦(⋀*op*. *op* ∈ *set* (*ast-problem.astδ* (*a*,*b*,*c*,*d*)) ⟹ *is-standard-operator op*);
op ∈ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops* (*a*,*b*,*c*,*d*)))⟧
⟹ *is-standard-operator op*
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′*:
⟦*op* ∈ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*));
(⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *is-standard-operator op*)⟧
⟹ *is-standard-operator op*
apply(*cases prob*)
using *is-standard-operator-rem-implicit-pres-ops*
by *blast*

lemma *in-rem-implicit-pres-δ*:
op ∈ *set* (*ast-problem.astδ prob*) ⟹
rem-implicit-pres op ∈ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*))
by(*auto simp add*: *ast-problem.astδ-def*)

lemma *rem-implicit-pres-ops-execute*:
assumes
(⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *is-standard-operator op*) **and**
s ∈ *ast-problem.valid-states prob*
shows *ast-problem.execute* (*rem-implicit-pres-ops prob*) *name s* = *ast-problem.execute prob name s*
proof−
have (*n*,*ps*,*es*,*c*) ∈ *set* (*ast-problem.astδ prob*) ⟹
(*filter* (*ast-problem.eff-enabled prob s*) *es*) = *es* **for** *n ps es c*
using *assms*(*2*)
by (*auto simp add*: *ast-problem.std-eff-enabled dest*!: *assms*(*1*))
moreover have (*n*,*ps*,*es*,*c*) ∈ *set* (*ast-problem.astδ prob*) ⟹
(*filter* (*ast-problem.eff-enabled* (*rem-implicit-pres-ops prob*) *s*) (*map rem-effect-implicit-pres es*))
= *map rem-effect-implicit-pres es* **for** *n ps es c*
using *assms*
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*)) ∘ *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*
dest: *rem-implicit-pres-ops-lookup-op-Some-2′ ast-problem.lookup-Some-inδ*)
qed

lemma *rem-implicit-pres-ops-path-to*:
  wf-ast-problem prob $\Longrightarrow$
    $(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow consistent\text{-}pres\text{-}op\ op) \Longrightarrow$
    $(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow is\text{-}standard\text{-}operator\ op) \Longrightarrow$
    $s \in ast\text{-}problem.valid\text{-}states\ prob \Longrightarrow$
    ast-problem.path-to (*rem-implicit-pres-ops prob*) $s\ \pi s\ s' = ast\text{-}problem.path\text{-}to$ *prob* $s\ \pi s\ s'$
 by (*induction* πs *arbitrary*: s)
   (*auto simp*: *rem-implicit-pres-ops-execute rem-implicit-pres-ops-enabled*
         ast-problem.path-to.simps wf-ast-problem.execute-preserves-valid)

lemma *rem-implicit-pres-ops-astG*[*simp*]: *ast-problem.astG* (*rem-implicit-pres-ops prob*) =
      ast-problem.astG prob
 apply(*cases prob*)
 by (*auto simp add*: *ast-problem.astG-def*)

lemma *rem-implicit-pres-ops-goal*[*simp*]: *ast-problem.G* (*rem-implicit-pres-ops prob*) = *ast-problem.G prob*
 apply(*cases prob*)
 using *rem-implicit-pres-ops-valid-states*
 by (*auto simp add*: *ast-problem.G-def ast-problem.astG-def subsuming-states-def'*)

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.I-def ast-problem.astI-def subsuming-states-def'*)

lemma *rem-implicit-pres-ops-init*[*simp*]: *ast-problem.I* (*rem-implicit-pres-ops prob*) = *ast-problem.I prob*
 apply(*cases prob*)
 by (*auto simp add*: *ast-problem.I-def ast-problem.astI-def*)

lemma *rem-implicit-pres-ops-valid-plan*:
 assumes *wf-ast-problem prob*
    $(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow consistent\text{-}pres\text{-}op\ op)$
    $(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow is\text{-}standard\text{-}operator\ op)$
 shows *ast-problem.valid-plan* (*rem-implicit-pres-ops prob*) πs = *ast-problem.valid-plan prob* πs
 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*
ast-problem.wf-operator prob op
shows
ast-problem.wf-operator (*rem-implicit-pres-ops prob*) (*rem-implicit-pres 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.numVals* (*rem-implicit-pres-ops prob*) *x*
if (*x*,*v*) ∈ *set* ((*implicit-pres effs*) @ *pres*) **for** *x v*
using *that asses*
by (*auto dest*!: *in-implicit-pres simp*: *ast-problem.wf-partial-state-def ast-problem.wf-operator-def*)
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 (*map* (λ(-, *v*, -, -). *v*) *effs*) =
(*map* (λ(-, *v*, -, -). *v*) (*map rem-effect-implicit-pres effs*))
by *auto*
hence *distinct* (*map* (λ(-, *v*, -, -). *v*) (*map 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 (∃ *vp*. (*epres*,*x*,*vp*,*v*)∈*set effs*) ⟷ (*epres*,*x*,*None*,*v*)∈*set* (*map rem-effect-implicit-pres effs*)
for *epres x v*
by *force*
ultimately show *?thesis*

using *assms*(*2*)
by (*auto simp*: *op ast-problem.wf-operator-def rem-implicit-pres-ops-wf-partial-state*

split: *prod.splits*)
qed

lemma *rem-implicit-pres-ops-inδD*: *op* ∈ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*))
⟹ (∃ *op'*. *op'* ∈ *set* (*ast-problem.astδ prob*) ∧ *op* = *rem-implicit-pres op'*)
by (*cases prob*) (*force simp*: *ast-problem.astδ-def*)

lemma *rem-implicit-pres-ops-well-formed*:
assumes (⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*)
ast-problem.well-formed prob
shows *ast-problem.well-formed* (*rem-implicit-pres-ops prob*)
proof−
have *map fst* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*)) = *map fst* (*ast-problem.astδ prob*)
by (*cases prob*) (*auto simp*: *ast-problem.astδ-def*)
thus *?thesis*
using *assms*
by(*auto simp add*: *ast-problem.well-formed-def rem-implicit-pres-ops-wf-partial-state*
simp del: *rem-implicit-pres.simps*
dest!: *rem-implicit-pres-ops-inδD*
intro!: *rem-implicit-pres-wf-operator*)
qed

definition *is-standard-effect'*
:: *ast-effect* ⇒ *bool*
where *is-standard-effect'* ≡ λ(*pre*, -, *vpre*, -). *pre* = [] ∧ *vpre* = *None*

definition *is-standard-operator'*
:: *ast-operator* ⇒ *bool*
where *is-standard-operator'* ≡ λ(-, -, *effects*, -). *list-all is-standard-effect' effects*

lemma *rem-implicit-pres-is-standard-operator'*:
is-standard-operator (*n*,*p*,*es*,*c*) ⟹ *is-standard-operator'* (*rem-implicit-pres* (*n*,*p*,*es*,*c*))
by (*induction es*) (*auto simp*: *is-standard-operator'-def is-standard-operator-def is-standard-effect-def*
is-standard-effect'-def)

lemma *rem-implicit-pres-ops-is-standard-operator'*:
(⋀*op*. *op* ∈ *set* (*ast-problem.astδ* (*vs*, *I*, *G*, *ops*)) ⟹ *is-standard-operator op*)
⟹
π∈*set* (*ast-problem.astδ* (*rem-implicit-pres-ops* (*vs*, *I*, *G*, *ops*))) ⟹ *is-standard-operator' π*
by (*cases ops*) (*auto simp*: *ast-problem.astδ-def dest*!: *rem-implicit-pres-is-standard-operator'*)

locale *abs-ast-prob* = *wf-ast-problem* +

assumes *no-cond-effs*: $\forall \pi \in set\ ast\delta.\ is\text{-}standard\text{-}operator'\ \pi$

context *ast-problem*
begin

definition *abs-ast-variable-section* = $[0..<(length\ astDom)]$

definition *abs-range-map*
:: $(nat \rightharpoonup nat\ list)$
where *abs-range-map* $\equiv$
map-of (*zip abs-ast-variable-section*
($map\ ((\lambda vals.\ [0..<length\ vals])\ o\ snd\ o\ snd)$
astDom))

end

context *abs-ast-prob*
begin

lemma *is-valid-vars-1*: $astDom \neq [] \Longrightarrow abs\text{-}ast\text{-}variable\text{-}section \neq []$
by(*simp add*: *abs-ast-variable-section-def*)

end

lemma *upt-eq-Nil-conv'*[*simp*]: $([] = [i..<j]) = (j = 0 \vee j \leq i)$
by(*induct j*)*simp-all*

lemma *map-of-zip-map-Some*:
$v < length\ xs$
$\Longrightarrow (map\text{-}of\ (zip\ [0..<length\ xs]\ (map\ f\ xs))\ v) = Some\ (f\ (xs\ !\ v))$
by (*induction xs rule*: *rev-induct*) (*auto simp add*: *nth-append map-add-Some-iff*)

lemma *map-of-zip-Some*:
$v < length\ xs$
$\Longrightarrow (map\text{-}of\ (zip\ [0..<length\ xs]\ xs)\ v) = Some\ (xs\ !\ v)$
by (*induction xs rule*: *rev-induct*) (*auto simp add*: *nth-append map-add-Some-iff*)

lemma *in-set-zip-lengthE*:
$(x,y) \in set(zip\ [0..<length\ xs]\ xs) \Longrightarrow (\llbracket\ x < length\ xs;\ xs\ !\ x = y\ \rrbracket \Longrightarrow R) \Longrightarrow R$
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* ($\lambda v.$ *abs-range-map* $v \neq None$) *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* ≡ *map-of* (*zip* [*0*..<*length astI*] *astI*)

end

context *abs-ast-prob*
begin

lemma *valid-abs-init-1*: *abs-ast-initial-state v* ≠ *None* ⟷ *v* ∈ *set abs-ast-variable-section*
by (*simp add*: *abs-ast-variable-section-def numVars-def wf-initial*(*1*) *abs-ast-initial-state-def*)

lemma *abs-range-map-Some*:
shows *v* ∈ *set abs-ast-variable-section* ⟹
(*abs-range-map v*) = *Some* [*0*..<*length* (*snd* (*snd* (*astDom* ! *v*)))]
by (*simp add*: *numVars-def abs-range-map-def o-def abs-ast-variable-section-def map-of-zip-map-Some*)

lemma *in-abs-v-sec-length*: *v* ∈ *set abs-ast-variable-section* ⟷ *v* < *length astDom*
by (*simp add*: *abs-ast-variable-section-def*)

lemma [*simp*]: *v* < *length astDom* ⟹ (*abs-ast-initial-state v*) = *Some* (*astI* ! *v*)
using *wf-initial*(*1*)[*simplified numVars-def*, *symmetric*]
by (*auto simp add*: *map-of-zip-Some abs-ast-initial-state-def split*: *prod.splits*)

lemma [*simp*]: *v* < *length astDom* ⟹ *astI* ! *v* < *length* (*snd* (*snd* (*astDom* ! *v*)))
using *wf-initial*(*1*)[*simplified numVars-def*, *symmetric*] *wf-initial*
by (*auto simp add*: *numVals-def abs-ast-initial-state-def*
split: *prod.splits*)

lemma [*intro*!]: *v* ∈ *set abs-ast-variable-section* ⟹ *x* < *length* (*snd* (*snd* (*astDom* ! *v*))) ⟹
x ∈ *set* (*the* (*abs-range-map v*))
using *abs-range-map-Some*
by (*auto simp add*:)

lemma [*intro*!]: *x*<*length astDom* ⟹ *astI* ! *x* < *length* (*snd* (*snd* (*astDom* ! *x*)))
using *wf-initial*[*unfolded numVars-def numVals-def*]
by *auto*

lemma [*simp*]: *abs-ast-initial-state v* = *Some a* ⟹ *a* < *length* (*snd* (*snd* (*astDom* ! *v*)))
by(*auto simp add*: *abs-ast-initial-state-def*
wf-initial(*1*)[*unfolded numVars-def numVals-def*, *symmetric*]
elim!: *in-set-zip-lengthE*)

lemma *valid-abs-init-2*:
abs-ast-initial-state v ≠ *None* ⟹ (*the* (*abs-ast-initial-state v*)) ∈ *set* (*the* (*abs-range-map v*))
using *valid-abs-init-1*
by *auto*

end

context *ast-problem*
begin

definition *abs-ast-goal*
:: *nat-sas-plus-state*
where *abs-ast-goal* ≡ *map-of astG*

end

context *abs-ast-prob*
begin

lemma [*simp*]: *wf-partial-state s* ⟹ (*v*, *a*) ∈ *set s* ⟹ *v* ∈ *set abs-ast-variable-section*
by (*auto simp add*: *wf-partial-state-def abs-ast-variable-section-def numVars-def split*: *prod.splits*)

lemma *valid-abs-goal-1*: *abs-ast-goal v* ≠ *None* ⟹ *v* ∈ *set abs-ast-variable-section*
using *wf-goal*
by (*auto simp add*: *abs-ast-goal-def dest*!: *map-of-SomeD*)

lemma *in-abs-rangeI*: *wf-partial-state s* ⟹ (*v*, *a*) ∈ *set s* ⟹ (*a* ∈ *set* (*the* (*abs-range-map v*)))
by (*auto simp add*: *abs-range-map-Some wf-partial-state-def numVals-def split*: *prod.splits*)

lemma *valid-abs-goal-2*:
abs-ast-goal v ≠ *None* ⟹ (*the* (*abs-ast-goal v*)) ∈ *set* (*the* (*abs-range-map v*))
using *wf-goal*
by (*auto simp add*: *map-of-SomeD weak-map-of-SomeI abs-ast-goal-def intro*!: *in-abs-rangeI*)

end

context *ast-problem*
begin

definition *abs-ast-operator*
:: *ast-operator* ⇒ *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*[*intro*!]:
wf-operator op ⟹ *list-all* (λ(*v*, *a*). *ListMem v abs-ast-variable-section*)
(*precondition-of* (*abs-ast-operator op*))
by (*cases op*; *auto simp add*: *abs-ast-operator-def wf-operator-def list.pred-set ListMem-iff*)

lemma *wf-operator-preD*: *wf-operator* (*name*, *pres*, *effs*, *cost*) ⟹ *wf-partial-state pres*
by (*simp add*: *wf-operator-def*)

lemma *abs-valid-operator-2*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). (∃ *y*. *abs-range-map v* = *Some y*) ∧ *ListMem a* (*the* (*abs-range-map v*)))
(*precondition-of* (*abs-ast-operator op*))
by(*cases op*,
auto dest!: *wf-operator-preD simp*: *list.pred-set ListMem-iff abs-ast-operator-def*
intro!: *abs-rangeI*[*simplified not-None-eq*] *in-abs-rangeI*)

lemma *wf-operator-effE*: *wf-operator* (*name*, *pres*, *effs*, *cost*) ⟹
(⟦*distinct* (*map* (λ(-, *v*, -, -). *v*) *effs*);
⋀*epres x vp v*. (*epres*,*x*,*vp*,*v*)∈*set effs* ⟹ *wf-partial-state epres*;
⋀*epres x vp v*.(*epres*,*x*,*vp*,*v*)∈*set effs* ⟹ *x* < *numVars*;
⋀*epres x vp v*. (*epres*,*x*,*vp*,*v*)∈*set effs* ⟹ *v* < *numVals x*;
⋀*epres x vp v*. (*epres*,*x*,*vp*,*v*)∈*set effs* ⟹
case vp of None ⇒ *True* | *Some v* ⇒ *v*<*numVals x*⟧
⟹ *P*)
⟹ *P*
unfolding *wf-operator-def*
by (*auto split*: *prod.splits*)

lemma *abs-valid-operator-3′*:
wf-operator (*name*, *pre*, *eff*, *cost*) ⟹
list-all (λ(*v*, *a*). *ListMem v abs-ast-variable-section*) (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)

by (*fastforce simp add*: *list.pred-set ListMem-iff abs-ast-variable-section-def image-def numVars-def*
elim!: *wf-operator-effE split*: *prod.splits*)

lemma *abs-valid-operator-3*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). *ListMem v abs-ast-variable-section*) (*effect-of* (*abs-ast-operator op*))
by (*cases op*, *simp add*: *abs-ast-operator-def abs-valid-operator-3'*)

lemma *wf-abs-eff*: *wf-operator* (*name*, *pre*, *eff*, *cost*) ⟹ *wf-partial-state* (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
by (*elim wf-operator-effE*, *induction eff*)
(*fastforce simp*: *wf-partial-state-def image-def o-def split*: *prod.split-asm*)+

lemma *abs-valid-operator-4'*:
wf-operator (*name*, *pre*, *eff*, *cost*) ⟹
list-all (λ(*v*, *a*). (*abs-range-map v* ≠ *None*) ∧ *ListMem a* (*the* (*abs-range-map v*))) (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
apply(*subst list.pred-set ListMem-iff*)+
apply(*drule wf-abs-eff*)
by (*metis* (*mono-tags*, *lifting*) *abs-rangeI case-prodI2 in-abs-rangeI*)

lemma *abs-valid-operator-4*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). (∃ *y*. *abs-range-map v* = *Some y*) ∧ *ListMem a* (*the* (*abs-range-map v*)))
(*effect-of* (*abs-ast-operator op*))
using *abs-valid-operator-4'*
by (*cases op*, *simp add*: *abs-ast-operator-def*)

lemma *consistent-list-set*: *wf-partial-state s* ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) *s*) *s*
by (*auto simp add*: *list.pred-set wf-partial-state-def eq-key-imp-eq-value split*: *prod.splits*)

lemma *abs-valid-operator-5'*:
wf-operator (*name*, *pre*, *eff*, *cost*) ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) *pre*) *pre*
apply(*drule wf-operator-preD*)
by (*intro consistent-list-set*)

lemma *abs-valid-operator-5*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) (*precondition-of* (*abs-ast-operator op*)))
(*precondition-of* (*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*) $\Longrightarrow$
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* $\neq$ *v'* $\vee$ *a* = *a'*) *s*) *s*
by (*auto simp add*: *list.pred-set wf-partial-state-def eq-key-imp-eq-value split*: *prod.splits*)

lemma *abs-valid-operator-6'*:
assumes *wf-operator* (*name*, *pre*, *eff*, *cost*)
shows *list-all* (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* $\neq$ *v'* $\vee$ *a* = *a'*) (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*))
(*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
proof−
have $*$: *map fst* (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*) = (*map* (λ(-, *v*,-,-). *v*) *eff*)
by (*induction eff*) *auto*
show *?thesis*
using *assms*
apply(*elim wf-operator-effE*)
apply(*intro consistent-list-set-2*)
by (*subst* $*$)
qed

lemma *abs-valid-operator-6*[*intro*!]:
wf-operator op $\Longrightarrow$
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* $\neq$ *v'* $\vee$ *a* = *a'*) (*effect-of* (*abs-ast-operator op*)))
(*effect-of* (*abs-ast-operator op*))
using *abs-valid-operator-6'*
by (*cases op*, *simp add*: *abs-ast-operator-def*)

end

context *ast-problem*
begin

definition *abs-ast-operator-section*
:: *nat-sas-plus-operator list*
where *abs-ast-operator-section* $\equiv$ [*abs-ast-operator op*. *op* $\leftarrow$ *astδ*]

definition *abs-prob* :: *nat-sas-plus-problem*
where *abs-prob* = (|
variables-of = *abs-ast-variable-section*,
operators-of = *abs-ast-operator-section*,
initial-of = *abs-ast-initial-state*,
goal-of = *abs-ast-goal*,
range-of = *abs-range-map*
|)

end

context *abs-ast-prob*
begin

lemma [*simp*]: *op* ∈ *set ast*δ ⟹ (*is-valid-operator-sas-plus abs-prob*) (*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* ⇒ *nat-sas-plus-plan*
where *abs-ast-plan* π *s*
≡ *map* (*abs-ast-operator o the o lookup-operator*) π *s*

lemma *std-then-implici-effs*[*simp*]: *is-standard-operator′* (*name*, *pres*, *effs*, *layer*) ⟹ *implicit-pres effs* = []
apply(*induction effs*)
by (*auto simp add*: *is-standard-operator′-def implicit-pres-def is-standard-effect′-def*)

lemma [*simp*]: *enabled* π *s* ⟹ *lookup-operator* π = *Some* (*name*, *pres*, *effs*, *layer*) ⟹
is-standard-operator′ (*name*, *pres*, *effs*, *layer*) ⟹
(*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* (λ(-,*x*,-,*v*). (*x*,*v*)) *effs*)
by (*auto simp add*: *abs-ast-operator-def*
split: *option.splits prod.splits*)

lemma *exect-eq-abs-execute*:
⟦*enabled* π *s*; *lookup-operator* π = *Some* (*name*, *preconds*, *effs*, *layer*);
is-standard-operator′(*name*, *preconds*, *effs*, *layer*)⟧ ⟹
execute π *s* = (*execute-operator-sas-plus s* ((*abs-ast-operator o the o lookup-operator*) π))
using *effs-eq-abs-effs*
by (*auto simp add*: *execute-def execute-operator-sas-plus-def*)

lemma *enabled-then-sas-applicable*:
enabled π $s \Longrightarrow$ *SAS-Plus-Representation.is-operator-applicable-in s* ((*abs-ast-operator o the o lookup-operator*) π)
by (*auto simp add*: *subsuming-states-def enabled-def lookup-operator-def SAS-Plus-Representation.is-operator-applicable-in-def abs-ast-operator-def split*: *option.splits prod.splits*)

lemma *path-to-then-exec-serial*: $\forall \pi \in$ *set* πs. *lookup-operator* $\pi \neq$ *None* $\Longrightarrow$ *path-to s* πs $s' \Longrightarrow$ $s' \subseteq_m$ *execute-serial-plan-sas-plus s* (*abs-ast-plan* πs)
proof(*induction* πs *arbitrary*: *s* s')
case (*Cons a* πs)
then show *?case*
by (*force simp*: *exect-eq-abs-execute abs-ast-plan-def lookup-Some-in*δ *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*:
(*None* = *map-of xys x*) = ($x \notin$ *fst* ‘ (*set xys*))
by (*induct xys*) *simp-all*

lemma [*simp*]: *I* = *abs-ast-initial-state*
apply(*intro HOL.ext*)
by (*auto simp*: *map-of-eq-None-iff set-map*[*symmetric*] *I-def abs-ast-initial-state-def map-of-zip-Some dest*: *map-of-SomeD*)

lemma [*simp*]: $\forall \pi \in$ *set* πs. *lookup-operator* $\pi \neq$ *None* $\Longrightarrow$ *op*$\in$*set* (*abs-ast-plan* πs) $\Longrightarrow$ *op* $\in$ *set abs-ast-operator-section*
by (*induction* πs) (*auto simp*: *abs-ast-plan-def abs-ast-operator-section-def lookup-Some-in*δ)

end

context *ast-problem*
begin

lemma *path-to-then-lookup-Some*: ($\exists s' \in G$. *path-to s* πs s') $\Longrightarrow$ ($\forall \pi \in$ *set* πs. *lookup-operator* $\pi \neq$ *None*)
by (*induction* πs *arbitrary*: *s*) (*force simp add*: *enabled-def split*: *option.splits*)+

lemma *valid-plan-then-lookup-Some*: *valid-plan* $\pi s \Longrightarrow$ ($\forall \pi \in$ *set* πs. *lookup-operator* $\pi \neq$ *None*)
using *path-to-then-lookup-Some*
by(*simp add*: *valid-plan-def*)

end

context *abs-ast-prob*

begin

theorem *valid-plan-then-is-serial-sol*:
assumes *valid-plan* πs
shows *is-serial-solution-for-problem abs-prob* (*abs-ast-plan* πs)
using *valid-plan-then-lookup-Some*[*OF assms*] *assms*
by (*auto simp add*: *is-serial-solution-for-problem-def valid-plan-def initial-of-def abs-prob-def abs-ast-goal-def G-def subsuming-states-def list-all-iff ListMem-iff map-le-trans path-to-then-exec-serial*
simp del: *sas-plus-problem.select-defs*)

end

11.2 Translating SAS+ represnetation to Fast-Downward's

context *ast-problem*
begin

definition *lookup-action*:: *nat-sas-plus-operator* ⇒ *ast-operator option* **where**
lookup-action op ≡
find (λ(-, *pres*, *effs*, -). *precondition-of op* = *pres* ∧
map (λ(*v*,*a*). ([], *v*, *None*, *a*)) (*effect-of op*) = *effs*)
astδ

end

context *abs-ast-prob*
begin

lemma *find-Some*: *find P xs* = *Some x* ⟹ *x* ∈ *set xs* ∧ *P x*
by (*auto simp add*: *find-Some-iff*)

lemma *distinct-find*: *distinct* (*map f xs*) ⟹ *x* ∈ *set xs* ⟹ *find* (λ*x'*. *f x'* = *f x*) *xs* = *Some x*
by (*induction xs*) (*auto simp*: *image-def*)

lemma *lookup-operator-find*: *lookup-operator nme* = *find* (λ*op*. *fst op* = *nme*) *astδ*
by (*auto simp*: *lookup-operator-def intro*!: *arg-cong*[**where** *f* = (λ*x*. *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*)

lemma *std-lookup-action*: *is-standard-operator′ ast-op* ⟹ *ast-op* ∈ *set astδ* ⟹
∃ *ast-op′*. *lookup-action* (*abs-ast-operator ast-op*) = *Some ast-op′*
unfolding *lookup-action-def*
apply(*intro mem-find-Some*)
by (*auto split*: *prod.splits simp*: *o-def*)

lemma *is-applicable-then-enabled-1*:
ast-op ∈ *set astδ* ⟹
∃ *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*) ⟹ *name* = *name′*
using *lookup-operator-wf*(*2*)
by *fastforce*

lemma *eq-name-eq-pres*: (*name*, *pres*, *effs*, *layer*) ∈ *set astδ* ⟹ (*name*, *pres′*, *effs′*, *layer′*) ∈ *set astδ*
⟹ *pres* = *pres′*
using *eq-key-imp-eq-value*[*OF wf-operators*(*1*)]
by *auto*

lemma *eq-name-eq-effs*:
name = name′ ⟹ (name, pres, effs, layer) ∈ set astδ ⟹ (name′, pres′, effs′, layer′) ∈ set astδ
⟹ effs = effs′
using *eq-key-imp-eq-value*[*OF wf-operators*(*1*)]
by *auto*

lemma *is-applicable-then-subsumes*:
s ∈ valid-states ⟹
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator* (*name, pres, effs, layer*)) *⟹*
s ∈ subsuming-states (*map-of pres*)
by (*simp add*: *subsuming-states-def SAS-Plus-Representation.is-operator-applicable-in-def abs-ast-operator-def*)

lemma *eq-name-eq-pres′*:
⟦*s ∈ valid-states* ; *is-standard-operator′* (*name, pres, effs, layer*); (*name, pres, effs, layer*) *∈ set astδ* ;
lookup-operator ((*fst o the o lookup-action o abs-ast-operator*) (*name, pres, effs, layer*)) = *Some* (*name′, pres′, effs′, layer′*)⟧
⟹ pres = pres′
using *lookup-operator-eq-name lookup-operator-works-2*
by (*fastforce dest*!: *std-lookup-action simp*: *eq-name-eq-pres*[*OF lookup-action-Some-in-δ lookup-Some-inδ*])

lemma *is-applicable-then-enabled-2*:
⟦*s ∈ valid-states* ; *ast-op ∈ set astδ* ;
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator ast-op*);
lookup-operator ((*fst o the o lookup-action o abs-ast-operator*) *ast-op*) = *Some* (*name, pres, effs, layer*)⟧
⟹ s∈subsuming-states (*map-of pres*)
apply(*cases ast-op*)
using *eq-name-eq-pres′ is-applicable-then-subsumes no-cond-effs*
by *fastforce*

lemma *is-applicable-then-enabled-3*:
⟦*s ∈ valid-states*;
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-inδ 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*)⟧

$\Longrightarrow$ *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*) $\in$ *set astδ*
s $\in$ *valid-states*
shows *effs* = *effs'*
using *std-lookup-action*[*OF assms*(*2*,*3*)] *assms*
by (*auto simp*: *lookup-operator-works-3*[*OF assms*(*2*,*3*)]
eq-name-eq-effs[*OF lookup-operator-eq-name lookup-action-Some-in-δ lookup-Some-inδ*])

lemma *std-eff-enabled'*[*simp*]:
is-standard-operator' (*name*, *pres*, *effs*, *layer*) $\Longrightarrow$ *s* $\in$ *valid-states* $\Longrightarrow$ (*filter* (*eff-enabled s*) *effs*) = *effs*
by (*induction effs*) (*auto simp*: *is-standard-operator'-def is-standard-effect'-def eff-enabled-def subsuming-states-def*)

lemma *execute-abs*:
⟦*s* $\in$ *valid-states*; *ast-op* $\in$ *set astδ*;
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator ast-op*)⟧
$\Longrightarrow$
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*) =
(*SAS-Plus-Representation.is-operator-applicable-in s op* ∧
sat-preconds-as (*execute-operator-sas-plus s op*) *ops*)

lemma *exec-serial-then-path-to'*:
⟦*s* $\in$ *valid-states*;
∀ *op*∈*set ops*. ∃ *ast-op*∈ *set astδ*. *op* = *abs-ast-operator ast-op*;
(*sat-preconds-as s ops*)⟧ $\Longrightarrow$
path-to s (*map* (*fst o the o lookup-action*) *ops*) (*execute-serial-plan-sas-plus s ops*)
proof(*induction ops arbitrary*: *s*)
case (*Cons a ops*)

```
  then show ?case
    using execute-abs is-applicable-then-enabled execute-preserves-valid
    apply simp
    by metis
qed auto

end

fun rem-condless-ops where
  rem-condless-ops s [] = []
| rem-condless-ops s (op#ops) =
     (if SAS-Plus-Representation.is-operator-applicable-in s op then
      op # (rem-condless-ops (execute-operator-sas-plus s op) ops)
      else [])

context abs-ast-prob
begin

lemma exec-rem-consdless: execute-serial-plan-sas-plus s (rem-condless-ops s ops)
= execute-serial-plan-sas-plus s ops
  by (induction ops arbitrary: s) auto

lemma rem-conless-sat: sat-preconds-as s (rem-condless-ops s ops)
  by (induction ops arbitrary: s) auto

lemma set-rem-condlessD: x ∈ set (rem-condless-ops s ops) ⟹ x ∈ set ops
  by (induction ops arbitrary: s) auto

lemma exec-serial-then-path-to:
  ⟦s ∈ valid-states;
   ∀ op∈set ops. ∃ ast-op∈ set astδ. op = abs-ast-operator ast-op⟧ ⟹
   path-to s (((map (fst o the o lookup-action)) o rem-condless-ops s) ops)
            (execute-serial-plan-sas-plus s ops)
  using  rem-conless-sat
  by (fastforce dest!: set-rem-condlessD
                intro!: exec-serial-then-path-to'
                        [where s = s and ops = rem-condless-ops s ops,
                         unfolded exec-rem-consdless])

lemma is-serial-solution-then-abstracted:
  is-serial-solution-for-problem abs-prob ops
   ⟹ ∀ op∈set ops. ∃ ast-op∈ set astδ. op = abs-ast-operator ast-op
  by(auto simp: is-serial-solution-for-problem-def abs-prob-def Let-def list.pred-set
                ListMem-iff abs-ast-operator-section-def
        split: if-splits)

lemma lookup-operator-works-1': lookup-action op = Some π' ⟹ ∃ op. lookup-operator
(fst π') = 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*)⟧ ⟹
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*⟧ ⟹
(∃ $s' \in G$. *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

context *ast-problem*
begin

definition *decode-abs-plan* ≡ (*map* (*fst o the o lookup-action*) *o rem-condless-ops I*)

end

context *abs-ast-prob*
begin

theorem *is-serial-sol-then-valid-plan*:
⟦*is-serial-solution-for-problem abs-prob ops*⟧ ⟹
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

theory *Solve-SASP*

imports *AST-SAS-Plus-Equivalence SAT-Solve-SAS-Plus*
HOL−Data-Structures.RBT-Map HOL−Library.Code-Target-Nat HOL.String
AI-Planning-Languages-Semantics.SASP-Checker Set2-Join-RBT
begin

11.3 SAT encoding works for Fast-Downward's representation

context *abs-ast-prob*
begin

theorem *is-serial-sol-then-valid-plan-encoded*:
$\mathcal{A} \models \Phi_\forall\ (\varphi\ (\textit{prob-with-noop abs-prob}))\ t \Longrightarrow$
valid-plan
(*decode-abs-plan*
(*rem-noops*
(*map* (λ*op*. ${\varphi_O}^{-1}$ (*prob-with-noop abs-prob*) *op*)
(*concat* (${\Phi}^{-1}$ (φ (*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* πs = *length* (*abs-ast-plan* πs)
by (*auto simp*: *abs-ast-plan-def*)

theorem *valid-plan-then-is-serial-sol-encoded*:
valid-plan $\pi s \Longrightarrow$ *length* $\pi s \leq\ h \Longrightarrow \exists\, \mathcal{A}.\ \mathcal{A} \models \Phi_\forall\ (\varphi\ (\textit{prob-with-noop abs-prob}))$
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*: ((*l*::*nat*) < *n*) $\Longrightarrow$ ($\exists\, m.\ n$ = *Suc m* $\wedge$ *l* $\leq$ *m*)
by (*induction n*) *auto*

locale *cnf-to-dimacs* =
fixes *h* :: *nat* **and** *n-ops* :: *nat*
begin

fun *var-to-dimacs* **where**
var-to-dimacs (*Operator t k*) = *1* + *t* + *k* * *h*
| *var-to-dimacs* (*State t k*) = *1* + *n-ops* * *h* + *t* + *k* * (*h*)

definition *dimacs-to-var* **where**
dimacs-to-var v ≡
if v < *1* + *n-ops* ∗ *h then*
Operator ((*v* − *1*) *mod* (*h*)) ((*v* − *1*) *div* (*h*))
else
(*let k* = ((*v* − *1*) − *n-ops* ∗ *h*) *in*
State (*k mod* (*h*)) (*k div* (*h*)))

fun *valid-state-var* **where**
valid-state-var (*Operator t k*) ⟷ *t* < *h* ∧ *k* < *n-ops*
| *valid-state-var* (*State t k*) ⟷ *t* < *h*

lemma *State-works*:
valid-state-var (*State t k*) ⟹
dimacs-to-var (*var-to-dimacs* (*State t k*)) =
(*State t k*)
by (*induction k*) (*auto simp add*: *dimacs-to-var-def add.left-commute Let-def*)

lemma *Operator-works*:
valid-state-var (*Operator t k*) ⟹
dimacs-to-var (*var-to-dimacs* (*Operator t k*)) =
(*Operator t k*)
by (*induction k*) (*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 ⟹
dimacs-to-var (*var-to-dimacs sv*) = *sv*
apply(*cases sv*)
using *State-works Operator-works*
by *auto*

end

lemma *changing-atoms-works*:
(⋀*x*. *P x* ⟹ (*f o g*) *x* = *x*) ⟹ (∀ *x*∈*atoms phi*. *P x*) ⟹ *M* ⊨ *phi* ⟷ *M o f* ⊨ *map-formula g phi*
by (*induction phi*) *auto*

lemma *changing-atoms-works′*:
M o g ⊨ *phi* ⟷ *M* ⊨ *map-formula g phi*
by (*induction phi*) *auto*

context *cnf-to-dimacs*
begin

lemma *sat-plan-to-dimacs*:
(⋀*sv*. *sv*∈*atoms sat-plan-formula* ⟹ *valid-state-var sv*) ⟹
M ⊨ *sat-plan-formula*

$\longleftrightarrow$ *M o dimacs-to-var* $\models$ *map-formula var-to-dimacs sat-plan-formula*
by(*auto intro*!: *changing-atoms-works*[**where** *P* = *valid-state-var*] *simp*: *sat-plan-to-dimacs-works*)

lemma *dimacs-to-sat-plan*:
M o var-to-dimacs $\models$ *sat-plan-formula*
$\longleftrightarrow$ *M* $\models$ *map-formula var-to-dimacs sat-plan-formula*
using *changing-atoms-works*′ **.**

end

locale *sat-solve-sasp* = *abs-ast-prob* Π + *cnf-to-dimacs Suc h Suc* (*length astδ*)
for Π *h*
begin

lemma *encode-initial-state-valid*:
sv ∈ *atoms* (*encode-initial-state Prob*) ⟹ *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*: *length* (*operators-of* (*φ* (*prob-with-noop abs-prob*))) = *Suc* (*length astδ*)
by(*simp add*: *abs-prob-def abs-ast-operator-section-def sas-plus-problem-to-strips-problem-def prob-with-noop-def*)

lemma *encode-operator-effect-valid-1*: *t* < *h* ⟹ *op* ∈ *set* (*operators-of* (*φ* (*prob-with-noop abs-prob*))) ⟹
sv ∈ *atoms*
(⋀(*map* (λ*v*.
¬(*Atom* (*Operator t* (*index* (*operators-of* (*φ* (*prob-with-noop abs-prob*))) *op*)))
∨ *Atom* (*State* (*Suc t*) (*index vs v*)))
asses)) ⟹
valid-state-var sv
using *length-operators*
by (*induction asses*) (*auto simp*: *simp add*: *cnf-to-dimacs.valid-state-var.simps*)

lemma *encode-operator-effect-valid-2*: *t* < *h* ⟹ *op* ∈ *set* (*operators-of* (*φ* (*prob-with-noop abs-prob*))) ⟹
sv ∈ *atoms*
(⋀(*map* (λ*v*.
¬(*Atom* (*Operator t* (*index* (*operators-of* (*φ* (*prob-with-noop abs-prob*))) *op*)))
∨ ¬ (*Atom* (*State* (*Suc t*) (*index vs v*))))
asses)) ⟹
valid-state-var sv
using *length-operators*
by (*induction asses*) (*auto simp*: *simp add*: *cnf-to-dimacs.valid-state-var.simps*)

end

lemma *atoms-And-append*: *atoms* (⋀ (*as1* @ *as2*)) = *atoms* (⋀ *as1*) ∪ *atoms* (⋀ *as2*)
by (*induction as1*) *auto*

context *sat-solve-sasp*
begin

lemma *encode-operator-effect-valid*:
sv ∈ *atoms* (*encode-operator-effect* (φ (*prob-with-noop abs-prob*)) *t op*) ⟹
t < *h* ⟹ *op* ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹
valid-state-var sv
by (*force simp*: *encode-operator-effect-def Let-def atoms-And-append*
intro!: *encode-operator-effect-valid-1 encode-operator-effect-valid-2*)

end

lemma *foldr-And*: *foldr* (∧) *as* (¬ ⊥) = (⋀ *as*)
by (*induction as*) *auto*

context *sat-solve-sasp*
begin

lemma *encode-all-operator-effects-valid*:
t < *Suc h* ⟹
sv ∈ *atoms* (*encode-all-operator-effects* (φ (*prob-with-noop abs-prob*)) (*operators-of* (φ (*prob-with-noop abs-prob*))) *t*) ⟹
valid-state-var sv
unfolding *encode-all-operator-effects-def foldr-And*
by (*force simp add*: *encode-operator-effect-valid*)

lemma *encode-operator-precondition-valid-1*:
t < *h* ⟹ *op* ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹
sv ∈ *atoms*
(⋀(*map* (λ*v*.
¬ (*Atom* (*Operator t* (*index* (*operators-of* (φ (*prob-with-noop abs-prob*))) *op*))) ∨ *Atom* (*State t* (*f v*)))
asses)) ⟹
valid-state-var sv
using *length-operators*
by (*induction asses*) (*auto simp*: *simp add*: *cnf-to-dimacs.valid-state-var.simps*)

lemma *encode-operator-precondition-valid*:
sv ∈ *atoms* (*encode-operator-precondition* (φ (*prob-with-noop abs-prob*)) *t op*) ⟹

t < *h* ⟹ *op* ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹
valid-state-var sv
by (*force simp*: *encode-operator-precondition-def Let-def*

intro!: *encode-operator-precondition-valid-1*)

lemma *encode-all-operator-preconditions-valid*:
$t < Suc\ h \Longrightarrow$
$sv \in atoms\ (encode\text{-}all\text{-}operator\text{-}preconditions\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))$
$(operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob)))\ t) \Longrightarrow$
valid-state-var sv
unfolding *encode-all-operator-preconditions-def foldr-And*
by (*force simp add*: *encode-operator-precondition-valid*)

lemma *encode-operators-valid*:
$sv \in atoms\ (encode\text{-}operators\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t) \Longrightarrow t < Suc\ h$
$\Longrightarrow$
valid-state-var sv
unfolding *encode-operators-def Let-def*
by (*force simp add*: *encode-all-operator-preconditions-valid encode-all-operator-effects-valid*)

lemma *encode-negative-transition-frame-axiom'*:
$t < h \Longrightarrow$
$set\ deleting\text{-}operators \subseteq set\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))) \Longrightarrow$
$sv \in atoms$
$(\neg(Atom\ (State\ t\ v\text{-}idx))$
$\vee\ (Atom\ (State\ (Suc\ t)\ v\text{-}idx)$
$\vee \bigvee (map\ (\lambda op.\ Atom\ (Operator\ t\ (index\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop$
$abs\text{-}prob)))\ op))))$
$deleting\text{-}operators))) \Longrightarrow$
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*:
$sv \in atoms\ (encode\text{-}negative\text{-}transition\text{-}frame\text{-}axiom\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))$
$t\ v) \Longrightarrow\ t < h \Longrightarrow$
valid-state-var sv
unfolding *encode-negative-transition-frame-axiom-def Let-def*
apply(*intro encode-negative-transition-frame-axiom'*[*of t*])
by *auto*

lemma *encode-positive-transition-frame-axiom-valid*:
$sv \in atoms\ (encode\text{-}positive\text{-}transition\text{-}frame\text{-}axiom\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))$
$t\ v) \Longrightarrow t < h \Longrightarrow$
valid-state-var sv
unfolding *encode-positive-transition-frame-axiom-def Let-def*
apply(*intro encode-negative-transition-frame-axiom'*[*of t*])
by *auto*

lemma *encode-all-frame-axioms-valid*:
$sv \in atoms\ (encode\text{-}all\text{-}frame\text{-}axioms\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t) \Longrightarrow t <$
$Suc\ h \Longrightarrow$
valid-state-var sv

unfolding *encode-all-frame-axioms-def Let-def atoms-And-append*
by (*force simp add*: *encode-negative-transition-frame-axiom-valid encode-positive-transition-frame-axiom-valid*

lemma *encode-goal-state-valid*:
sv ∈ *atoms* (*encode-goal-state Prob t*) ⟹ *t* < *Suc h* ⟹ *valid-state-var sv*
by (*auto simp add*: *encode-state-variable-def Let-def encode-goal-state-def split*:
sat-plan-variable.splits bool.splits)

lemma *encode-problem-valid*:
sv ∈ *atoms* (*encode-problem* (φ (*prob-with-noop abs-prob*)) *h*) ⟹ *valid-state-var*
sv
unfolding *encode-problem-def*
using *encode-initial-state-valid encode-operators-valid encode-all-frame-axioms-valid*
encode-goal-state-valid
by *fastforce*

lemma *encode-interfering-operator-pair-exclusion-valid*:
sv ∈ *atoms* (*encode-interfering-operator-pair-exclusion* (φ (*prob-with-noop abs-prob*))
t op$_1$ *op*$_2$) ⟹ *t* < *Suc h* ⟹
op$_1$ ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹ *op*$_2$ ∈ *set*
(*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹
valid-state-var sv
by (*auto simp*: *encode-interfering-operator-pair-exclusion-def Let-def length-operators*[*symmetric*]
cnf-to-dimacs.valid-state-var.simps)

lemma *encode-interfering-operator-exclusion-valid*:
sv ∈ *atoms* (*encode-interfering-operator-exclusion* (φ (*prob-with-noop abs-prob*))
t) ⟹ *t* < *Suc h* ⟹
valid-state-var sv
unfolding *encode-interfering-operator-exclusion-def Let-def foldr-And*
by (*force simp add*: *encode-interfering-operator-pair-exclusion-valid*)

lemma *encode-problem-with-operator-interference-exclusion-valid*:
sv ∈ *atoms* (*encode-problem-with-operator-interference-exclusion* (φ (*prob-with-noop*
abs-prob)) *h*) ⟹ *valid-state-var sv*
unfolding *encode-problem-with-operator-interference-exclusion-def*
using *encode-initial-state-valid encode-operators-valid encode-all-frame-axioms-valid*
encode-goal-state-valid
encode-interfering-operator-exclusion-valid
by *fastforce*

lemma *planning-by-cnf-dimacs-complete*:
valid-plan π *s* ⟹ *length* π *s* ≤ *h* ⟹
∃ *M*. *M* ⊨ *map-formula var-to-dimacs* (Φ$_\forall$ (φ (*prob-with-noop abs-prob*)) *h*)
using *valid-plan-then-is-serial-sol-encoded*
sat-plan-to-dimacs[*OF encode-problem-with-operator-interference-exclusion-valid*]
by *meson*

lemma *planning-by-cnf-dimacs-sound*:

$\mathcal{A} \models$ *map-formula var-to-dimacs* ($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *t*) $\Longrightarrow$
valid-plan
(*decode-abs-plan*
(*rem-noops*
(*map* (λ*op*. ${\varphi_O}^{-1}$ (*prob-with-noop abs-prob*) *op*)
(*concat* (${\Phi}^{-1}$ (φ (*prob-with-noop abs-prob*)) ($\mathcal{A}$ *o var-to-dimacs*) *t*)))))
using *changing-atoms-works'*
by (*fastforce intro*!: *is-serial-sol-then-valid-plan-encoded*)

end

12.1 Going from Formualae to DIMACS-like CNF

We now represent the CNF formulae into a very low-level representation that is reminiscent to the DIMACS representation, where a CNF formula is a list of list of integers.

fun *disj-to-dimacs*::*nat formula* $\Rightarrow$ *int list* **where**
disj-to-dimacs ($\varphi_1 \vee \varphi_2$) = *disj-to-dimacs* φ_1 @ *disj-to-dimacs* φ_2
| *disj-to-dimacs* $\bot$ = []
| *disj-to-dimacs* (*Not* $\bot$) = [−*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 ($\varphi_1 \wedge \varphi_2$) = *cnf-to-dimacs* φ_1 @ *cnf-to-dimacs* φ_2
| *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* $\Longrightarrow$ *x* ≤ *find-max xs* (**is** *?P* $\Longrightarrow$ *?Q*)
proof−
have *x* ∈ *set xs* $\Longrightarrow$ (*x*::*nat*) ≤ (*fold max xs m*) **for** *m*
unfolding *max-def*
apply (*induction xs arbitrary*: *m rule*: *rev-induct*)
using *nat-le-linear*
by (*auto dest*: *le-trans simp add*:)
thus *?P* $\Longrightarrow$ *?Q*
by(*auto simp add*: *find-max-def max-def*)
qed

fun *formula-vars* **where**
formula-vars ($\bot$) = [] |
formula-vars (*Atom k*) = [*k*] |
formula-vars (*Not F*) = *formula-vars F* |
formula-vars (*And F G*) = *formula-vars F* @ *formula-vars G* |
formula-vars (*Imp F G*) = *formula-vars F* @ *formula-vars G* |

formula-vars (*Or F G*) = *formula-vars F* @ *formula-vars G*

lemma *atoms-formula-vars*: *atoms f* = *set* (*formula-vars f*)
by (*induction f*) *auto*

lemma *max-var*: *v* ∈ *atoms* (*f*::*nat formula*) ⟹ *v* ≤ *find-max* (*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* ≤ *a* ⟹ (*b*::*nat*) ≤ *fold* (λ*x m*. *if m* ≤ *x then x 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 ls cs ≡ (∀ *c*∈*set cs*. (∃ *l*∈*set ls*. *l* ∈ *set c*)) ∧
distinct (*map dimacs-lit-to-var ls*)

fun *model-to-dimacs-model* **where**
model-to-dimacs-model M (*v*#*vs*) = (*if M v then int v else* − (*int v*)) # (*model-to-dimacs-model M vs*)
| *model-to-dimacs-model* - [] = []

lemma *model-to-dimacs-model-append*:
set (*model-to-dimacs-model M* (*vs* @ *vs*′)) = *set* (*model-to-dimacs-model M vs*) ∪ *set* (*model-to-dimacs-model M vs*′)
by (*induction vs*) *auto*

lemma *upt-append-sing*: *xs* @ [*x*] = [*a*..<*n-vars*] ⟹ *a* < *n-vars* ⟹ (*xs* = [*a*..<*n-vars* − *1*] ∧ *x* = *n-vars*−*1* ∧ *n-vars* > *0*)
by (*induction n-vars*) *auto*

lemma *upt-eqD*: *upt a b* = *upt a b*′ ⟹ (*b* = *b*′ ∨ *b*′ ≤ *a* ∨ *b* ≤ *a*)
by (*induction b*) (*auto dest*!: *upt-append-sing split*: *if-splits*)

lemma *pos-in-model*: *M n* ⟹ *0* < *n* ⟹ *n* < *n-vars* ⟹ *int n* ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*])
by (*induction n-vars*) (*auto simp add*: *less-Suc-eq model-to-dimacs-model-append*
)

lemma *neg-in-model*: ¬ *M n* ⟹ *0* < *n* ⟹ *n* < *n-vars* ⟹ − (*int n*) ∈ *set*

(*model-to-dimacs-model M* [*1*..<*n-vars*])
by (*induction n-vars*) (*auto simp add*: *less-Suc-eq model-to-dimacs-model-append*)

lemma *in-model*: *0* < *n* ⟹ *n* < *n-vars* ⟹ *int n* ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*]) ∨ − (*int n*) ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*])
using *pos-in-model neg-in-model*
by *metis*

lemma *model-to-dimacs-model-all-vars*:
(∀ *v*∈*atoms f*. *0* < *v* ∧ *v* < *n-vars*) ⟹ *is-cnf f* ⟹ *M* ⊨ *f* ⟹
(∀ *n*<*n-vars*. *0* < *n* ⟶ (*int n* ∈ *set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]) ∨
− (*int n*) ∈ *set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*])))
using *in-model neg-in-model pos-in-model*
by (*auto simp add*: *le-less model-to-dimacs-model-append split*: *if-splits*)

lemma *cnf-And*: *set* (*cnf-to-dimacs* (*f1* ∧ *f2*)) = *set* (*cnf-to-dimacs f1*) ∪ *set* (*cnf-to-dimacs f2*)
by *auto*

lemma *one-always-in*:
1 < *n-vars* ⟹ *1* ∈ *set* (*model-to-dimacs-model M* ([*1*..<*n-vars*])) ∨ − *1* ∈ *set* (*model-to-dimacs-model M* ([*1*..<*n-vars*]))
by (*induction n-vars*) (*auto simp add*: *less-Suc-eq model-to-dimacs-model-append*)

lemma [*simp*]: (*disj-to-dimacs* (*f1* ∨ *f2*)) = (*disj-to-dimacs f1*) @ (*disj-to-dimacs f2*)
by *auto*

lemma [*simp*]: (*atoms* (*f1* ∨ *f2*)) = *atoms f1* ∪ *atoms f2*
by *auto*

lemma *isdisj-disjD*: (*is-disj* (*f1* ∨ *f2*)) ⟹ *is-disj f1* ∧ *is-disj f2*
by (*cases f1*; *auto*)

lemma *disj-to-dimacs-sound*:
1 < *n-vars* ⟹ (∀ *v*∈*atoms f*. *0* < *v* ∧ *v* < *n-vars*) ⟹ *is-disj f* ⟹ *M* ⊨ *f*
⟹ ∃ *l*∈*set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]). *l* ∈ *set* (*disj-to-dimacs f*)
apply(*induction f*)
using *neg-in-model pos-in-model one-always-in*
by (*fastforce elim*!: *is-lit-plus.elims dest*!: *isdisj-disjD*)+

lemma *is-cnf-disj*: *is-cnf* (*f1* ∨ *f2*) ⟹ (⋀*f*. *f1* ∨ *f2* = *f* ⟹ *is-disj f* ⟹ *P*) ⟹ *P*
by *auto*

lemma *cnf-to-dimacs-disj*: *is-disj f* ⟹ *cnf-to-dimacs f* = [*disj-to-dimacs f*]
by (*induction f*) *auto*

lemma *model-to-dimacs-model-all-clauses*:
1 < n-vars $\Longrightarrow$ ($\forall$ *v*$\in$*atoms f. 0 < v* $\wedge$ *v < n-vars*) $\Longrightarrow$ *is-cnf f* $\Longrightarrow$ *M* $\models$ *f* $\Longrightarrow$
c$\in$*set* (*cnf-to-dimacs f*) $\Longrightarrow$ $\exists$ *l*$\in$*set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]).
l $\in$ *set c*
proof(*induction f arbitrary*:)
case (*Not f*)
then show *?case*
using *in-model neg-in-model*
by (*fastforce elim*!: *is-lit-plus.elims*)+
next
case (*Or f1 f2*)
then show *?case*
using *cnf-to-dimacs-disj disj-to-dimacs-sound*
by(*elim is-cnf-disj*, *simp*)
qed (*insert in-model neg-in-model pos-in-model*, *auto*)

lemma *upt-eq-Cons-conv*:
(*x*#*xs* = [*i*..<*j*]) = (*i < j* $\wedge$ *i* = *x* $\wedge$ [*i*+*1*..<*j*] = *xs*)
using *upt-eq-Cons-conv*
by *metis*

lemma *model-to-dimacs-model-append'*:
(*model-to-dimacs-model M* (*vs* @ *vs'*)) = (*model-to-dimacs-model M vs*) @ (*model-to-dimacs-model M vs'*)
by (*induction vs*) *auto*

lemma *model-to-dimacs-neg-nin*:
n-vars $\leq$ *x* $\Longrightarrow$ *int x* $\notin$ *set* (*model-to-dimacs-model M* [*a*..<*n-vars*])
by (*induction n-vars arbitrary*: *a*) (*auto simp*: *model-to-dimacs-model-append'*)

lemma *model-to-dimacs-pos-nin*:
n-vars $\leq$ *x* $\Longrightarrow$ $-$ *int x* $\notin$ *set* (*model-to-dimacs-model M* [*a*..<*n-vars*])
by (*induction n-vars arbitrary*: *a*) (*auto simp*: *model-to-dimacs-model-append'*)

lemma *int-cases2'*:
z $\neq$ *0* $\Longrightarrow$ ($\bigwedge$*n*. *0* $\neq$ (*int n*) $\Longrightarrow$ *z* = *int n* $\Longrightarrow$ *P*) $\Longrightarrow$ ($\bigwedge$*n*. *0* $\neq$ $-$ (*int n*) $\Longrightarrow$
z = $-$ (*int n*) $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
by (*metis* (*full-types*) *int-cases2*)

lemma *model-to-dimacs-model-distinct*:
1 < n-vars $\Longrightarrow$ *distinct* (*map dimacs-lit-to-var* (*model-to-dimacs-model M* [*1*..<*n-vars*]))
by (*induction n-vars*)
(*fastforce elim*!: *int-cases2'*
simp add: *dimacs-lit-to-var-def model-to-dimacs-model-append'*
model-to-dimacs-neg-nin model-to-dimacs-pos-nin)+

lemma *model-to-dimacs-model-sound*:
1 < n-vars $\Longrightarrow$ ($\forall$ *v*$\in$*atoms f. 0 < v* $\wedge$ *v < n-vars*) $\Longrightarrow$ *is-cnf f* $\Longrightarrow$ *M* $\models$ *f* $\Longrightarrow$

dimacs-model (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]) (*cnf-to-dimacs f*)
unfolding *dimacs-model-def*
using *model-to-dimacs-model-all-vars model-to-dimacs-model-all-clauses model-to-dimacs-model-distinct*
by *auto*

lemma *model-to-dimacs-model-sound-exists*:
$1 < n\text{-}vars \Longrightarrow (\forall v{\in}atoms\ f.\ 0 < v \wedge v < n\text{-}vars) \Longrightarrow is\text{-}cnf\ f \Longrightarrow M \models f \Longrightarrow$
$\exists M\text{-}dimacs.\ dimacs\text{-}model\ M\text{-}dimacs\ (cnf\text{-}to\text{-}dimacs\ f)$
using *model-to-dimacs-model-sound*
by *metis*

definition *dimacs-to-atom* ::*int* ⇒ *nat formula* **where**
dimacs-to-atom l ≡ *if* (*l* < *0*) *then Not* (*Atom* (*nat* (*abs l*))) *else Atom* (*nat* (*abs l*))

definition *dimacs-to-disj*::*int list* ⇒ *nat formula* **where**
dimacs-to-disj f ≡ ⋁ (*map dimacs-to-atom f*)

definition *dimacs-to-cnf*::*int list list* ⇒ *nat formula* **where**
dimacs-to-cnf f ≡ ⋀*map dimacs-to-disj f*

definition *dimacs-model-to-abs dimacs-M M* ≡
fold (λ*l M*. *if* (*l* > *0*) *then M*((*nat* (*abs l*)):= *True*) *else M*((*nat* (*abs l*)):= *False*)) *dimacs-M M*

lemma *dimacs-model-to-abs-atom*:
0 < *x* ⟹ *int x* ∈ *set dimacs-M* ⟹ *distinct* (*map dimacs-lit-to-var dimacs-M*) ⟹ *dimacs-model-to-abs dimacs-M M x*
proof (*induction dimacs-M arbitrary*: *M rule*: *rev-induct*)
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 *dimacs-model-to-abs-atom′*:
0 < *x* ⟹ −(*int x*) ∈ *set dimacs-M* ⟹ *distinct* (*map dimacs-lit-to-var dimacs-M*) ⟹ ¬ *dimacs-model-to-abs dimacs-M M x*
proof (*induction dimacs-M arbitrary*: *M rule*: *rev-induct*)
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 ∈ *atoms* (*map-formula var-to-dimacs f*) ⟹ *0* < *v*
by (*induction f*) *auto*

lemma *map-is-disj*: *is-disj f* ⟹ *is-disj* (*map-formula F f*)
by (*induction f*) (*auto elim*: *is-lit-plus.elims*)

lemma *map-is-cnf*: *is-cnf f* $\Longrightarrow$ *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 πs $\Longrightarrow$ *length* $\pi s \leq h$ $\Longrightarrow$
let cnf-formula = (*map-formula var-to-dimacs*
($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *h*))
in
$\exists$ *dimacs-M*. *dimacs-model dimacs-M* (*cnf-to-dimacs cnf-formula*)
unfolding *Let-def*
by (*fastforce simp*: *var-to-dimacs-pos*
dest!: *planning-by-cnf-dimacs-complete*
intro: *model-to-dimacs-model-sound-max-var map-is-cnf*
is-cnf-encode-problem-with-operator-interference-exclusion
is-valid-problem-sas-plus-then-strips-transformation-too
noops-valid abs-prob-valid)

lemma *planning-dimacs-sound*:
let cnf-formula =
(*map-formula var-to-dimacs*
($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *h*))
in
dimacs-model dimacs-M (*cnf-to-dimacs cnf-formula*) $\Longrightarrow$
valid-plan
(*decode-abs-plan*
(*rem-noops*
(*map* (λ*op*. ${\varphi_O}^{-1}$ (*prob-with-noop abs-prob*) *op*)
(*concat*
(Φ^{-1} (φ (*prob-with-noop abs-prob*)) ((*dimacs-model-to-abs dimacs-M*
(λ-. *False*)) *o var-to-dimacs*) *h*)))))
by(*fastforce simp*: *var-to-dimacs-pos Let-def*
intro: *planning-by-cnf-dimacs-sound model-to-dimacs-model-complete-max-var*
map-is-cnf is-cnf-encode-problem-with-operator-interference-exclusion

is-valid-problem-sas-plus-then-strips-transformation-too abs-prob-valid
noops-valid)

end

13 Code Generation

We now generate SML code equivalent to the functions that encode a problem as a CNF formula and that decode the model of the given encodings into a plan.

lemma [*code*]:
dimacs-model ls cs $\equiv$ (*list-all* (λ*c*. *list-ex* (λ*l*. *ListMem l c*) *ls*) *cs*) $\wedge$
distinct (*map dimacs-lit-to-var ls*)
unfolding *dimacs-model-def*

by (*auto simp*: *list.pred-set ListMem-iff list-ex-iff*)

definition
SASP-to-DIMACS h prob ≡
  cnf-to-dimacs
    (*map-formula*
      (*cnf-to-dimacs.var-to-dimacs* (*Suc h*) (*Suc* (*length* (*ast-problem.astδ prob*))))
        (Φ∀ (φ (*prob-with-noop* (*ast-problem.abs-prob prob*))) *h*))

lemma *planning-dimacs-complete-code*:
  ⟦*ast-problem.well-formed prob*;
    ∀ π∈*set* (*ast-problem.astδ prob*). *is-standard-operator'* π;
    ast-problem.valid-plan prob π *s*;
    length π *s* ≤ *h*⟧ ⟹
   let cnf-formula = (*SASP-to-DIMACS h prob*) *in*
      ∃ *dimacs-M*. *dimacs-model dimacs-M cnf-formula*
  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* ≡ *SASP-to-DIMACS h* (*rem-implicit-pres-ops prob*)

lemma *planning-dimacs-complete-code'*:
  ⟦*ast-problem.well-formed prob*;
    (⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*);
    (⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *is-standard-operator op*);
    ast-problem.valid-plan prob π *s*;
    length π *s* ≤ *h*⟧ ⟹
   let cnf-formula = (*SASP-to-DIMACS' h prob*) *in*
      ∃ *dimacs-M*. *dimacs-model dimacs-M cnf-formula*
  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-inδD*)

A function that does the checks required by the completeness theorem above, and returns appropriate error messages if any of the checks fail.

definition
  encode 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:
  ⟦ast-problem.valid-plan prob πs; length πs ≤ h;
      encode h prob = Inl cnf-formula⟧ ⟹
       (∃ 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:
  encode h prob = Inr err ⟹
    ¬(ast-problem.well-formed prob ∧ (∀ op ∈ set (ast-problem.astδ prob). consis-
tent-pres-op op) ∧
    (∀ 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) ⟹ match-pres pres s ⟷ 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
  tree-map-of updatea T [] = T
| 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* ⊆$_m$ *s* ⟷ *m2* ⊆$_m$ *s*
by *presburger*

lemma *match-pres-submap*:

match-pres (*inorder* (*M.tree-map-of'* *empty pres*)) *s* ⟷ *Map.map-of pres* ⊆$_m$ *s*
using *match-pres-distinct*[*OF M.tree-map-of-distinct*]
by (*smt M.invar-def M.invar-empty M.tree-map-of-invar M.tree-map-of-works map-le-cong map-of-eq map-of-lookup*)

lemma [*code*]:
SAS-Plus-Representation.is-operator-applicable-in s op ⟷
match-pres (*inorder* (*M.tree-map-of'* *empty* (*SAS-Plus-Representation.precondition-of op*))) *s*
by (*simp add*: *match-pres-submap SAS-Plus-Representation.is-operator-applicable-in-def*)

definition *decode-DIMACS-model dimacs-M h prob* ≡
(*ast-problem.decode-abs-plan prob*
(*rem-noops*
(*map* (λ*op*. *φ*$_O{}^{-1}$ (*prob-with-noop* (*ast-problem.abs-prob prob*)) *op*)
(*concat*
(Φ^{-1} (*φ* (*prob-with-noop* (*ast-problem.abs-prob prob*)))
((*dimacs-model-to-abs dimacs-M* (λ-. *False*)) *o*
(*cnf-to-dimacs.var-to-dimacs* (*Suc h*)
(*Suc* (*length* (*ast-problem.astδ prob*)))))
h)))))

lemma *planning-dimacs-sound-code*:
⟦*ast-problem.well-formed prob*;
∀ *π*∈*set* (*ast-problem.astδ prob*). *is-standard-operator' π*⟧ ⟹
let
cnf-formula = (*SASP-to-DIMACS h prob*);
decoded-plan = *decode-DIMACS-model dimacs-M h prob*
in
(*dimacs-model dimacs-M cnf-formula* ⟶ *ast-problem.valid-plan prob decoded-plan*)
unfolding *SASP-to-DIMACS-def decode-DIMACS-model-def Let-def*
apply(*rule 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 de-*

```
coded-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-inδD)+
```

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-def 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 (∀ op∈set (ast-problem.astδ prob). consistent-pres-op op) then
        if (∀ op∈set (ast-problem.astδ prob). is-standard-operator op) then
          if (dimacs-model M (SASP-to-DIMACS' h prob)) then
            Inl (decode-DIMACS-model' M h prob)
          else Inr (STR ''Error: Model does not solve the problem!'')
        else
          Inr (STR ''Error: Conditional effects!'')
      else
        Inr (STR ''Error: Preconditions inconsistent'')
```

```
    else
      Inr (STR ''Error: Problem malformed!'')

lemma decode-sound:
  decode M h prob = Inl plan ⟹
        ast-problem.valid-plan prob plan
  unfolding decode-def
  apply (auto split: if-splits simp: list.pred-set)
  using planning-dimacs-sound-code'
  by auto

lemma decode-complete:
  decode M h prob = Inr err ⟹
        ¬ (ast-problem.well-formed prob ∧
          (∀ op ∈ set (ast-problem.astδ prob). consistent-pres-op op) ∧
          (∀ π∈set (ast-problem.astδ prob). is-standard-operator π) ∧
          dimacs-model M (SASP-to-DIMACS' h prob))
  unfolding decode-def
  by (auto split: if-splits simp: list.pred-set)

lemma [code]:
  ListMem x' []= False
  ListMem x' (x#xs) = (x' = x ∨ ListMem x' xs)
  by (simp add: ListMem-iff)+

lemmas [code] = SASP-to-DIMACS-def ast-problem.abs-prob-def
          ast-problem.abs-ast-variable-section-def ast-problem.abs-ast-operator-section-def
              ast-problem.abs-ast-initial-state-def ast-problem.abs-range-map-def
              ast-problem.abs-ast-goal-def cnf-to-dimacs.var-to-dimacs.simps
          ast-problem.astδ-def ast-problem.astDom-def ast-problem.abs-ast-operator-def
            ast-problem.astI-def ast-problem.astG-def ast-problem.lookup-action-def
          ast-problem.I-def execute-operator-sas-plus-def ast-problem.decode-abs-plan-def

definition nat-opt-of-integer :: integer ⇒ nat option where
      nat-opt-of-integer i = (if (i ≥ 0) then Some (nat-of-integer i) else None)

definition max-var :: int list ⇒ int where
      max-var xs ≡ fold (λ(x::int) (y::int). if abs x ≥ abs y then (abs x) else y) xs
(0::int)

export-code encode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode
   String.implode max-var concat char-of-nat Int.nat integer-of-int length int-of-integer
  in SML module-name exported file-prefix SASP-to-DIMACS

export-code decode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode
   String.implode max-var concat char-of-nat Int.nat integer-of-int length int-of-integer
  in SML module-name exported file-prefix decode-DIMACS-model

end
```

References

[1] M. Abdulaziz and F. Kurz. Formally verified sat-based ai planning, 2020.

[2] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[3] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

[4] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2018. https://isabelle.in.tum.de/doc/isar-ref.pdf.