

Verified SAT-Based AI Planning

Mohammad Abdulaziz and Friedrich Kurz*

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

Contents

1 State-Variable Representation	3
2 STRIPS Representation	3
3 STRIPS Semantics	5
3.1 Serial Plan Execution Semantics	5
3.2 Parallel Plan Semantics	15
3.3 Serializable Parallel Plans	46
3.4 Auxiliary lemmas about STRIPS	58
4 SAS+ Representation	58
5 SAS+ Semantics	64
5.1 Serial Execution Semantics	64
5.2 Parallel Execution Semantics	66
5.3 Serializable Parallel Plans	76
5.4 Auxiliary lemmata on SAS+	83
6 SAS+/STRIPS Equivalence	85
6.1 Translation of SAS+ Problems to STRIPS Problems	85
6.2 Equivalence of SAS+ and STRIPS	166

* Author names are alphabetically ordered.

7	The Basic SATPlan Encoding	176
7.1	Encoding Function Definitions	177
7.2	Decoding Function Definitions	180
7.3	Soundness of the Basic SATPlan Algorithm	212
7.4	Completeness	250
8	Serializable SATPlan Encodings	291
8.1	Soundness	298
8.2	Completeness	301
9	SAT-Solving of SAS+ Problems	308
10	Adding Noop actions to the SAS+ problem	310
11	Proving Equivalence of SAS+ representation and Fast-Downward's Multi-Valued Problem Representation	312
11.1	Translating Fast-Downward's represnetation to SAS+	312
11.2	Translating SAS+ represnetation to Fast-Downward's	328
11.3	SAT encoding works for Fast-Downward's representation	333
12	DIMACS-like semantics for CNF formulae	334
12.1	Going from Formualae to DIMACS-like CNF	340
13	Code Generation	346

```

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  $\rightarrow$  '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-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.

¹For the full reference on records see [4, 11.6, pp.260-265]

```

abbreviation operator-for
  :: 'variable list  $\Rightarrow$  'variable list  $\Rightarrow$  'variable list  $\Rightarrow$  'variable strips-operator
where operator-for pre add delete  $\equiv$  (
  precondition-of = pre
  , add-effects-of = add
  , delete-effects-of = delete )

definition to-precondition
  :: 'variable strips-operator  $\Rightarrow$  ('variable, bool) assignment list
where to-precondition op  $\equiv$  map ( $\lambda v.$  ( $v$ , True)) (precondition-of op)

definition to-effect
  :: 'variable strips-operator  $\Rightarrow$  ('variable, bool) Effect
where to-effect op = [( $v_a$ , True).  $v_a \leftarrow$  add-effects-of op] @ [( $v_d$ , False).  $v_d \leftarrow$  delete-effects-of op]

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

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

value stop

As discussed in ??, the effect of a STRIPS operator can be normalized to a conjunction of atomic effects. We can therefore construct the successor state by simply converting the list of add effects to assignments to True resp. converting the list of delete effect to a list of assignments to False and then adding the map corresponding to the assignments to the given state  $s$  as shown below in definition ??.2

definition execute-operator
  :: 'variable strips-state
   $\Rightarrow$  'variable strips-operator
   $\Rightarrow$  'variable strips-state (infixl < $\gg$ > 52)
where execute-operator  $s$  op
   $\equiv$   $s$  ++ map-of (effect-to-assignments op)

end

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

```

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

3 STRIPS Semantics

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

3.1 Serial Plan Execution Semantics

Serial plan execution is defined by primitive recursion on the plan. Definition ?? returns the given state if the state argument does not satisfy the precondition of the next operator in the plan. Otherwise it executes the rest of the plan on the successor state $s \gg op$ 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  $\Rightarrow$  'variable strips-plan  $\Rightarrow$  '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  $\Pi \pi$ 
     $\equiv$  (goal-of  $\Pi$ )  $\subseteq_m$  execute-serial-plan (initial-of  $\Pi$ )  $\pi$ 
     $\wedge$  list-all ( $\lambda op. ListMem op (operators-of \Pi)$ )  $\pi$ 
```

```
lemma is-valid-problem-strips-initial-of-dom:
  fixes  $\Pi$ :: 'a strips-problem
  assumes is-valid-problem-strips  $\Pi$ 
  shows dom  $((\Pi)_I) = set ((\Pi)_V)$ 
  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 disjoint (set (add-effects-of op)) (set (delete-effects-of op))

```

```

proof -
let ?ops = strips-problem.operators-of  $\Pi$ 
and ?vs = strips-problem.variables-of  $\Pi$ 
have list-all (is-valid-operator-strips  $\Pi$ ) ?ops
  using assms(1)
  unfolding is-valid-problem-strips-def
  by meson
moreover have  $\forall v \in \text{set}(\text{precondition-of } op). v \in \text{set}((\Pi)_v)$ 
  and  $\forall v \in \text{set}(\text{add-effects-of } op). v \in \text{set}((\Pi)_v)$ 
  and  $\forall v \in \text{set}(\text{delete-effects-of } op). v \in \text{set}((\Pi)_v)$ 
  and  $\forall v \in \text{set}(\text{add-effects-of } op). v \notin \text{set}(\text{delete-effects-of } op)$ 
  and  $\forall v \in \text{set}(\text{delete-effects-of } op). v \notin \text{set}(\text{add-effects-of } op)$ 
  using assms(2) calculation
  unfolding is-valid-operator-strips-def list-all-iff Let-def ListMem-iff
  using variables-of-def
  by auto+
ultimately show set (precondition-of op)  $\subseteq$  set (( $\Pi$ ) $_v$ )
  and set (add-effects-of op)  $\subseteq$  set (( $\Pi$ ) $_v$ )
  and set (delete-effects-of op)  $\subseteq$  set (( $\Pi$ ) $_v$ )
  and disjoint (set (add-effects-of op)) (set (delete-effects-of op))
  unfolding disjoint-def
  by fast+
qed

```

```

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

```

```

lemma effect-to-assignments-ii:
— NOTE effect-to-assignments can be simplified drastically given that only atomic effects and the add-effects as well as delete-effects lists only consist of variables.
assumes as = effect-to-assignments op
obtains as1 as2
where as = as1 @ as2
and as1 = map ( $\lambda v. (v, \text{True})$ ) (add-effects-of op)
and as2 = map ( $\lambda v. (v, \text{False})$ ) (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 = (λv. (v, True)) ` set (add-effects-of op)
  let ?delete-assignments = (λv. (v, False)) ` set (delete-effects-of op)
  obtain as1 as2
    where a1: as = as1 @ as2
      and a2: as1 = map (λv. (v, True)) (add-effects-of op)
      and a3: as2 = map (λv. (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 = (λv. (v, True)) ` set (add-effects-of op)
  let ?delete-assignments = (λv. (v, False)) ` set (delete-effects-of op)
  obtain as1 as2
    where a1: as = as1 @ as2
      and a2: as1 = map (λv. (v, True)) (add-effects-of op)
      and a3: as2 = map (λv. (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 ⊆ set as  
    by blast  
  moreover have {(v, False)} ⊆ ?delete-assignments  
    using assms(1) a2  
    by blast  
  ultimately have ∃ a. a ∈ set as ∧ a = (v, False)  
    by blast  
}  
then show ?thesis  
  using that  
  by blast  
qed
```

lemma effect--strips-i:

```
fixes op  
assumes e = effect--strips op  
obtains es1 es2  
where e = (es1 @ es2)  
and es1 = map (λv. (v, True)) (add-effects-of op)  
and es2 = map (λv. (v, False)) (delete-effects-of op)  
proof –  
  obtain es1 es2 where a: e = (es1 @ es2)  
    and b: es1 = map (λv. (v, True)) (add-effects-of op)  
    and c: es2 = map (λv. (v, False)) (delete-effects-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 (es1 @ es2)  
and es1 = map (λv. (v, True)) (add-effects-of op)  
and es2 = map (λv. (v, False)) (delete-effects-of op)  
shows ∀ v ∈ set (add-effects-of op). (∃ e' ∈ set es1. e' = (v, True))  
  and ∀ v ∈ set (delete-effects-of op). (∃ e' ∈ set es2. e' = (v, False))  
proof
```

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

```
fix v  
{  
  assume a: v ∈ set (add-effects-of op)
```

```

have set es1 = ( $\lambda v. (v, \text{True})$ ) ` set (add-effects-of op)
  using assms(2) List.set-map
  by auto
then obtain e'
  where e' ∈ set es1
  and e' = ( $\lambda v. (v, \text{True})$ ) v
  using a
  by blast
then have  $\exists e' \in \text{set } es1. e' = (v, \text{True})$ 
  by blast
}
thus  $v \in \text{set } (\text{add-effects-of } op) \implies \exists e' \in \text{set } es1. e' = (v, \text{True})$ 
  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:  $v \in \text{set } (\text{delete-effects-of } op)$ 
have set es2 = ( $\lambda v. (v, \text{False})$ ) ` set (delete-effects-of op)
  using assms(3) List.set-map
  by force
then obtain e''
  where e'' ∈ set es2
  and e'' = ( $\lambda v. (v, \text{False})$ ) v
  using a
  by blast
then have  $\exists e'' \in \text{set } es2. e'' = (v, \text{False})$ 
  by blast
}
thus  $\forall v \in \text{set } (\text{delete-effects-of } op). \exists e' \in \text{set } es2. e' = (v, \text{False})$ 
  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 o ?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 ≈ op)
  shows ⋀v. v ∈ set (delete-effects-of op) ∧ v ∉ set (add-effects-of op) ==> s' v =
Some False
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
} note f = this
{
  let ?vs = delete-effects-of op
  have ?vs ≠ []
    using a2
    by force
  then have dom (map-of as2) = set ?vs
    using d map-of-constant-assignments-dom
    by metis
} note g = this
{
  have s' = s ++ map-of as2 ++ map-of as1
    using assms(1)
    unfolding execute-operator-def
    using b
    by simp
thm f map-add-dom-app-simps(3)[OF f, of s ++ map-of as2]
moreover have s' v = (s ++ map-of as2) v
  using calculation map-add-dom-app-simps(3)[OF f, of s ++ map-of as2]
  by blast
moreover have v ∈ dom (map-of as2)
  using a2 g
  by argo
ultimately have s' v = map-of as2 v
  by fastforce

```

```

}
moreover
{
let ?f = λ-. False
from d have map-of as2 = (Some o ?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 $s' \equiv \text{execute-operator } op \ s$ 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
 $\bigwedge v.$ 
 $v \in \text{set (add-effects-of } op\text{)}$ 
 $\implies s' v = \text{Some True}$ 
and  $\bigwedge v.$ 
 $v \notin \text{set (add-effects-of } op\text{)} \wedge v \in \text{set (delete-effects-of } op\text{)}$ 
 $\implies s' v = \text{Some False}$ 
and  $\bigwedge v.$ 
 $v \notin \text{set (add-effects-of } op\text{)} \wedge v \notin \text{set (delete-effects-of } op\text{)}$ 
 $\implies s' v = s v$ 
proof (auto)
show  $\bigwedge v.$ 
 $v \in \text{set (add-effects-of } op\text{)}$ 
 $\implies s' v = \text{Some True}$ 
using assms effect--strips-iii-a
by fast
next
show  $\bigwedge v.$ 
 $v \notin \text{set (add-effects-of } op\text{)}$ 
 $\implies v \in \text{set (delete-effects-of } op\text{)}$ 
 $\implies s' v = \text{Some False}$ 
using assms effect--strips-iii-b
by fast
next
show  $\bigwedge v.$ 
 $v \notin \text{set (add-effects-of } op\text{)}$ 
 $\implies v \notin \text{set (delete-effects-of } op\text{)}$ 
 $\implies s' v = s v$ 

```

³Lemmas `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
     $\equiv$  list-all ( $\lambda op.$  is-operator-applicable-in s op) ops

```

```

definition are-operator-effects-consistent op1 op2  $\equiv$  let
    add1 = add-effects-of op1
    ; add2 = add-effects-of op2
    ; del1 = delete-effects-of op1
    ; del2 = delete-effects-of op2
    in  $\neg$ list-ex ( $\lambda v.$  list-ex ((=) v) del2) add1  $\wedge$   $\neg$ list-ex ( $\lambda v.$  list-ex ((=) v) add2)
    del1

```

```

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

```

```

definition execute-parallel-operator
:: 'variable strips-state
 $\Rightarrow$  'variable strips-operator list
 $\Rightarrow$  'variable strips-state
where execute-parallel-operator s ops
 $\equiv$  foldl (++) s (map (map-of  $\circ$  effect-to-assignments) ops)

```

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

```

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

```

```

definition are-operators-interfering op1 op2
 $\equiv$  list-ex ( $\lambda v.$  list-ex ((=) v) (delete-effects-of op1)) (precondition-of op2)
 $\vee$  list-ex ( $\lambda v.$  list-ex ((=) v) (precondition-of op1)) (delete-effects-of op2)

```

```

primrec are-all-operators-non-interfering
:: 'variable strips-operator list  $\Rightarrow$  bool

```

```

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

```

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

```

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

```

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

```

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

```

```

lemma are-all-operators-applicable-set:
are-all-operators-applicable s ops
 $\longleftrightarrow (\forall op \in set. ops. \forall v \in set. (precondition-of op). s v = Some True)$ 
unfolding are-all-operators-applicable-def
  STRIPS-Representation.is-operator-applicable-in-def list-all-iff
by presburger

```

```

lemma are-all-operators-applicable-cons:
assumes are-all-operators-applicable s (op # ops)
shows is-operator-applicable-in s op
  and are-all-operators-applicable s ops
proof –
  from assms have a: list-all (λop. is-operator-applicable-in s op) (op # ops)
  unfolding are-all-operators-applicable-def is-operator-applicable-in-def
    STRIPS-Representation.is-operator-applicable-in-def
  by blast
  then have is-operator-applicable-in s op
  by fastforce
moreover {

```

```

from a have list-all ( $\lambda op. \text{is-operator-applicable-in } s op$ ) 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 ( $\lambda v. \text{list-ex } ((=) v) (\text{delete-effects-of } op_2)$ ) (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 ( $\lambda v. \text{list-ex } ((=) v) (\text{add-effects-of } op_2)$ ) (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
 $\longleftrightarrow (\forall op_1 \in \text{set ops. } \forall op_2 \in \text{set ops.}$ 
  (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
  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 op1 op2
  shows are-operators-interfering op2 op1
  using assms
  unfolding are-operators-interfering-def list-ex-iff
  by fast

```

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

```

lemma are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs:
  assumes are-all-operators-non-interfering ops
    and are-operators-interfering op1 op2
    and op1 ≠ op2
  shows op1 ∉ set ops ∨ op2 ∉ set ops
  using assms
  proof (induction ops)
    case (Cons op ops)
      thm Cons.IH[OF - Cons.prems(2, 3)]
      have nb1: ∀ op' ∈ set ops. ¬are-operators-interfering op op'
        and nb2: are-all-operators-non-interfering ops
        using Cons.prems(1)
        unfolding are-all-operators-non-interfering.simps(2) list-all-iff
        by blast+
      then consider (A) op = op1
        | (B) op = op2
        | (C) op ≠ op1 ∧ op ≠ op2
        by blast
      thus ?case
        proof (cases)
          case A
          {
            assume op2 ∈ set (op # ops)
            then have op2 ∈ set ops
              using Cons.prems(3) A
              by force
            then have ¬are-operators-interfering op1 op2
              using nb1 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  $\neg \text{list-ex} ((=) v) (\text{delete-effects-of } op)$ 
    =  $\text{list-all} (\lambda v'. \neg v = v') (\text{delete-effects-of } op)$ 
    using not-list-ex-equals-list-all-not[
      where  $P=(=) v$  and  $xs=\text{delete-effects-of } op]$ 
    by blast
} moreover {
  from assms(1)
  have  $\neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{delete-effects-of } op)) (\text{precondition-of } op')$ 
    unfolding are-operators-interfering-def
    by blast
  then have  $\text{list-all} (\lambda v. \neg \text{list-ex} ((=) v) (\text{delete-effects-of } op)) (\text{precondition-of } op')$ 
    using not-list-ex-equals-list-all-not[
      where  $P=\lambda v. \text{list-ex} ((=) v) (\text{delete-effects-of } op)$  and  $xs=\text{precondition-of } op]$ 
    by blast
}
ultimately have  $\beta$ :
 $\text{list-all} (\lambda v. \text{list-all} (\lambda v'. \neg v = v') (\text{delete-effects-of } op)) (\text{precondition-of } op')$ 
  by presburger
moreover {
  fix v
  have  $\text{list-all} (\lambda v'. \neg v = v') (\text{delete-effects-of } op)$ 
    =  $(\forall v' \in \text{set} (\text{delete-effects-of } op). \neg v = v')$ 
    by (simp add: list-all-iff)
}
ultimately have  $\forall v \in \text{set} (\text{precondition-of } op'). \forall v' \in \text{set} (\text{delete-effects-of } op). \neg v = v'$ 
  using  $\beta$ 
  by (simp add: list-all-iff)
then have  $v \notin \text{set} (\text{delete-effects-of } op)$ 
  using  $\alpha$ 
  by fast
}
note  $b = \text{this}$ 
{
  fix v
  assume  $a: v \in \text{set} (\text{precondition-of } op')$ 
  have  $\text{list-all} (\lambda v. s v = \text{Some True}) (\text{precondition-of } op')$ 
    using assms(3)
    unfolding is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def
    by presburger
  then have  $\forall v \in \text{set} (\text{precondition-of } op'). s v = \text{Some True}$ 
    by (simp add: list-all-iff)
  then have  $s v = \text{Some True}$ 
    using  $a$ 
}

```

```

    by blast
}
note c = this
{
fix v
assume d:  $v \in \text{set}(\text{precondition-of } op')$ 
then have ? $s' v = \text{Some True}$ 
proof (cases  $v \in \text{set}(\text{add-effects-of } op))$ 
  case True
  then show ?thesis
    using a
    by blast
next
case e: False
then show ?thesis
proof (cases  $v \in \text{set}(\text{delete-effects-of } op))$ 
  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 ( $\lambda v. ?s' v = \text{Some True}$ ) (precondition-of op')
  by (simp add: list-all-iff)
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 nb1:  $\forall op \in set (a \# ops). is\text{-operator\text{-}applicable\text{-}in } s op$ 
    using assms(1) are-all-operators-applicable-set
    unfolding are-all-operators-applicable-def is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def list-all-iff
    by blast
  have nb2:  $\forall op \in set ops. \neg are\text{-operators\text{-}interfering } a op$ 
    using assms(3)
    unfolding are-all-operators-non-interfering-def list-all-iff
    by simp
  have nb3: is-operator-applicable-in s a
    using assms(1) are-all-operators-applicable-set
    unfolding are-all-operators-applicable-def is-operator-applicable-in-def
      STRIPS-Representation.is-operator-applicable-in-def list-all-iff
    by force
  {
    fix op
    assume op-in-ops: op  $\in$  set ops
    hence is-operator-applicable-in ?s' op
      using execute-parallel-plan-precondition-cons-i[of a op] nb1 nb2 nb3
      by force
  }
  then show are-all-operators-applicable ?s' ops
    unfolding are-all-operators-applicable-def list-all-iff
      is-operator-applicable-in-def
    by blast
  qed

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

lemma execute-parallel-operator-cons-equals:
  assumes are-all-operators-applicable s (a  $\#$  ops)
  and are-all-operator-effects-consistent (a  $\#$  ops)
  and are-all-operators-non-interfering (a  $\#$  ops)
  shows execute-parallel-operator s (a  $\#$  ops)
  = execute-parallel-operator (s ++ map-of (effect-to-assignments a)) ops
proof -
  let ?s' = s ++ map-of (effect-to-assignments a)
  {
    from assms(1, 2)
    have execute-parallel-operator s (Cons a ops)
  }

```

```

= foldl (++) s (map (map-of o effect-to-assignments) (Cons a ops))
  unfolding execute-parallel-operator-def
  by presburger
also have ... = foldl (++) (?s')
  (map (map-of o effect-to-assignments) ops)
  by auto
finally have execute-parallel-operator s (Cons a ops)
= foldl (++) (?s')
  (map (map-of o effect-to-assignments) ops)
using execute-parallel-operator-def
by blast
}
— NOTE the precondition of for  $a \# ops$  is also true for the tail list and state  $?s'$  as shown in lemma . Hence the precondition for the r.h.s. of the goal also holds.
moreover have execute-parallel-operator ?s' ops
= foldl (++) (s ++ (map-of o effect-to-assignments) a)
  (map (map-of o effect-to-assignments) ops)
  by (simp add: execute-parallel-operator-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
 $= \text{map}(\lambda v. (v, \text{True})) (\text{add-effects-of } op) @ \text{map}(\lambda v. (v, \text{False})) (\text{delete-effects-of } op)$
by (simp add: effect-to-assignments-i)

lemma effect-to-assignments-set-is[simp]:

```

set (effect-to-assignments op) = { ((v, a), True) | v a. (v, a) ∈ set (add-effects-of
op) }
    ∪ { ((v, a), False) | v a. (v, a) ∈ set (delete-effects-of op) }

proof –
  obtain as where effect--strips op = as
  and as = map (λv. (v, True)) (add-effects-of op)
    @ map (λv. (v, False)) (delete-effects-of op)
  unfolding effect--strips-def
  by blast
  moreover have as
  = map (λv. (v, True)) (add-effects-of op) @ map (λv. (v, False)) (delete-effects-of
op)
  using calculation(2)
  unfolding map-append map-map comp-apply
  by auto
  moreover have effect-to-assignments op = as
  unfolding effect-to-assignments-def calculation(1, 2)
  by auto
  ultimately show ?thesis
  unfolding set-map
  by auto
qed

corollary effect-to-assignments-construction-from-function-graph:
assumes set (add-effects-of op) ∩ set (delete-effects-of op) = {}
shows effect-to-assignments op = map
  (λv. (v, if ListMem v (add-effects-of op) then True else False))
  (add-effects-of op @ delete-effects-of op)
and effect-to-assignments op = map
  (λv. (v, if ListMem v (delete-effects-of op) then False else True))
  (add-effects-of op @ delete-effects-of op)

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

```

— TODO slow.

hence *effect-to-assignments op = map ?g (add-effects-of op @ delete-effects-of op)*
using *ListMem-iff assms*
by *fastforce*

}

ultimately show *effect-to-assignments op = map*
 $(\lambda v. (v, \text{if } \text{ListMem } v (\text{add-effects-of } op) \text{ then True else False}))$
 $(\text{add-effects-of } op @ \text{delete-effects-of } op)$
and *effect-to-assignments op = map*
 $(\lambda v. (v, \text{if } \text{ListMem } v (\text{delete-effects-of } op) \text{ then False else True}))$
 $(\text{add-effects-of } op @ \text{delete-effects-of } op)$
by *blast+*

qed

corollary *map-of-effect-to-assignments-is-none-if:*
assumes $\neg v \in \text{set}(\text{add-effects-of } op)$
and $\neg v \in \text{set}(\text{delete-effects-of } op)$
shows *map-of (effect-to-assignments op) v = None*
proof –

let $?l = \text{effect-to-assignments op}$

{

have *set ?l = { (v, True) | v. v ∈ set (add-effects-of op) }*
 $\cup \{ (v, \text{False}) | v. v \in \text{set}(\text{delete-effects-of } op) \}$
by *auto*

then have *fst ‘ set ?l*
 $= (\text{fst ‘ } \{ (v, \text{True}) | v. v \in \text{set}(\text{add-effects-of } op) \})$
 $\cup (\text{fst ‘ } \{ (v, \text{False}) | v. v \in \text{set}(\text{delete-effects-of } op) \})$
using *image-Un[of fst { (v, True) | v. v ∈ set (add-effects-of op) }]*
 $\{ (v, \text{False}) | v. v \in \text{set}(\text{delete-effects-of } op) \}]$
by *presburger*

— TODO slow.

also have $\dots = (\text{fst ‘ } (\lambda v. (v, \text{True})) ‘ \text{set}(\text{add-effects-of } op))$
 $\cup (\text{fst ‘ } (\lambda v. (v, \text{False})) ‘ \text{set}(\text{delete-effects-of } op))$
using *setcompr-eq-image[of $\lambda v. (v, \text{True})$ $\lambda v. v \in \text{set}(\text{add-effects-of } op)$]*
setcompr-eq-image[of $\lambda v. (v, \text{False})$ $\lambda v. v \in \text{set}(\text{delete-effects-of } op)$]
by *simp*

— TODO slow.

also have $\dots = id ‘ \text{set}(\text{add-effects-of } op) \cup id ‘ \text{set}(\text{delete-effects-of } op)$
by *force*

— TODO slow.

finally have *fst ‘ set ?l = set (add-effects-of op) ∪ set (delete-effects-of op)*
by *auto*

hence $v \notin \text{fst ‘ set ?l}$
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 o effect-to-assignments) ops
    have set-l-is: set ?l = (map-of o effect-to-assignments) ` set ops
      using set-map
      by fastforce

```

```

{
  let ?m = (map-of o 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 o 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.prem(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.prem(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 "op ∈ set ops. ¬v ∈ set (add-effects-of op) ∧ ¬v ∈ set (delete-effects-of op)
and s v = None"

```

```

shows execute-parallel-operator s ops v = None
using execute-parallel-operator-no-effect-if[OF assms(1)] assms(2)
by simp

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

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

will evaluate *v* to true, that is

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

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

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

under the same condition as for the positive effect. Lastly, if *v* is neither an add effect nor a delete effect for any operator in the operator set corresponding to *ops*, then the state after parallel operator execution remains unchanged, i.e.

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

```

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)○)
  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 $s \# \text{trace-parallel-plan-strips } s' \pi$ always is the last element of $\text{trace-parallel-plan-strips } s' \pi$ since *trace-parallel-plan-strips* always returns at least a singleton list (even if $\pi = []$).

lemma trace-parallel-plan-strips-last-cons-then:

```

  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 $\tau \equiv \text{trace-parallel-plan-strips } I \pi$ be a trace for a parallel plan π with initial state I .

First, all parallel operators $ops = \pi ! k$ for any index k with $k < \text{length } \tau - 1$ (meaning that k is not the index of the last element). ops 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 $k < \text{length } \tau - 1$ is not the last index since the index of the last element is $\text{length } \tau - 1$.⁴

lemma trace-parallel-plan-strips-operator-preconditions:

```

  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
  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)

```

⁴More precisely, the index of the last element is $\text{length } \tau - 1$ if τ is not empty which is however always true since the trace contains at least the initial state.

```

case True
have trace-parallel-plan-strips-cons: trace-parallel-plan-strips I (a #  $\pi$ )
  = I # trace-parallel-plan-strips (execute-parallel-operator I a)  $\pi$ 
  using True
  by simp
then show ?thesis
proof (cases k)
  case 0
  have trace-parallel-plan-strips I (a #  $\pi$ ) ! 0 = I
    using trace-parallel-plan-strips-cons
    by simp
  moreover have (a #  $\pi$ ) ! 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 #  $\pi$ ) ! Suc k' = trace-parallel-plan-strips
      ?I'  $\pi$  ! k'
      using trace-parallel-plan-strips-cons
      by simp
    moreover have (a #  $\pi$ ) ! Suc k' =  $\pi$  ! k'
      by simp
    moreover {
      have length (trace-parallel-plan-strips I (a #  $\pi$ ))
        = 1 + length (trace-parallel-plan-strips ?I'  $\pi$ )
        unfolding trace-parallel-plan-strips-cons
        by simp
      then have k' < length (trace-parallel-plan-strips ?I'  $\pi$ ) - 1
        using Suc Cons.prems
        by fastforce
        hence are-all-operators-applicable (trace-parallel-plan-strips ?I'  $\pi$  ! k')
      ( $\pi$  ! k)
         $\wedge$  are-all-operator-effects-consistent ( $\pi$  ! 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 #  $\pi$ ) = [I]
    by force
  then have length (trace-parallel-plan-strips I (a #  $\pi$ )) - 1 = 0
    by simp

```

```

— NOTE Thesis follows from contradiction with assumption.
then show ?thesis
  using Cons.prem
  by force
qed
qed auto

```

Another interesting property that we verify below is that elements of the trace store the result of plan prefix execution. This means that for an index k with

$k < \text{length}(\text{trace-parallel-plan-strips } I \pi)$, the k -th element of the trace is state reached by executing the plan prefix $\text{take } k \pi$ consisting of the first k parallel operators of π .

```

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

```

```

        using 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  $\neg \text{are-all-operators-applicable}$ 
  ( $\text{execute-parallel-plan } ?I'$ 
   ( $\text{take}(\text{length}(\text{trace-parallel-plan-strips } ?I' \pi) - 1) \pi$ ))
  ( $\pi ! (\text{length}(\text{trace-parallel-plan-strips } ?I' \pi) - 1)$ )
 $\vee \neg \text{are-all-operator-effects-consistent}$ 
  ( $\pi ! (\text{length}(\text{trace-parallel-plan-strips } ?I' \pi) - 1)$ )
using Cons.IH[of ?I']
by blast
moreover have  $\text{trace-parallel-plan-strips } ?I' \pi \neq []$ 
using trace-parallel-plan-strips-not-nil
by blast
ultimately show ?thesis
unfolding take-Cons'
by simp
next
case False
then have  $\text{length}(\text{trace-parallel-plan-strips } ?I' \pi) \geq \text{length } \pi + 1$ 
by fastforce
thm Cons.prems
moreover have  $\text{length}(\text{trace-parallel-plan-strips } I (\text{ops} \# \pi))$ 
 $= 1 + \text{length}(\text{trace-parallel-plan-strips } ?I' \pi)$ 
using True
by force
moreover have  $\text{length}(\text{trace-parallel-plan-strips } ?I' \pi)$ 
 $< \text{length}(\text{ops} \# \pi)$ 
using Cons.prems calculation(2)
by force
ultimately have False
by fastforce
thus ?thesis..
qed
next
case False
then have  $\tau\text{-is-singleton}: ?\tau = [I]$ 
using False
by auto
then have  $\text{ops} = (\text{ops} \# \pi) ! (\text{length } ?\tau - 1)$ 
by fastforce
moreover have  $\text{execute-parallel-plan } I (\text{take}(\text{length } ?\tau - 1) \pi) = I$ 
using  $\tau\text{-is-singleton}$ 
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-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-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.prem]
    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-length-gt-one-if 0.prem
      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  $\pi'$  where  $\pi$ -split:  $\pi = \text{ops} \# \pi'$ 
    using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
Suc.prems]
        by blast
let  $?I' = \text{execute-parallel-operator } I \text{ ops}$ 
{
    have length (trace-parallel-plan-strips I  $\pi$ ) =
         $1 + \text{length}(\text{trace-parallel-plan-strips } ?I' \pi')$ 
    using Suc.prems  $\pi$ -split
        by fastforce
    then have  $k < \text{length}(\text{trace-parallel-plan-strips } ?I' \pi')$ 
    using Suc.prems
        by fastforce
    moreover have trace-parallel-plan-strips I  $\pi ! Suc k$ 
         $= \text{trace-parallel-plan-strips } ?I' \pi' ! k$ 
    using Suc.prems  $\pi$ -split
        by force
    ultimately have trace-parallel-plan-strips I  $\pi ! Suc k$ 
         $= \text{execute-parallel-plan } ?I' (\text{take } k \pi')$ 
    using trace-parallel-plan-plan-prefix[of k  $?I' \pi'$ ]
        by argo
}
moreover have execute-parallel-plan I ( $\text{take}(\text{Suc } (\text{Suc } k)) \pi$ )
     $= \text{execute-parallel-plan } ?I' (\text{take } (\text{Suc } k) \pi')$ 
    using Suc.prems  $\pi$ -split
    by fastforce
moreover {
    have  $0 < \text{length}(\text{trace-parallel-plan-strips } I \pi) - 1$ 
    using Suc.prems
        by linarith
    hence are-all-operators-applicable I ( $\pi ! 0$ )
         $\wedge$  are-all-operator-effects-consistent ( $\pi ! 0$ )
    using trace-parallel-plan-strips-operator-preconditions[of 0 I  $\pi$ ]
        trace-parallel-plan-strips-head-is-initial-state[of I  $\pi$ ]
    by argo
}
ultimately show ?case
    using Suc.IH Suc.prems  $\pi$ -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  $\Pi$ :: 'a strips-problem
  assumes is-valid-problem-strips  $\Pi$ 
    and is-parallel-solution-for-problem  $\Pi \pi$ 
    and  $k < \text{length}(\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
  shows (trace-parallel-plan-strips  $((\Pi)_I) \pi ! k$ )  $v = \text{None} \longleftrightarrow v \notin \text{set } ((\Pi)_V)$ 
  proof -
    let ?vs = strips-problem.variables-of  $\Pi$ 
    and ?ops = strips-problem.operators-of  $\Pi$ 
    and ? $\tau$  = trace-parallel-plan-strips  $((\Pi)_I) \pi$ 
    and ?I = strips-problem.initial-of  $\Pi$ 
    show ?thesis
      using assms
      proof (induction k)
        case 0
          have ? $\tau$  ! 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  $k < \text{length } ?\tau - 1$ 
              using Suc.prems(3)
              by linarith
            then have IH: (trace-parallel-plan-strips ?I  $\pi ! k$ )  $v = \text{None} \longleftrightarrow v \notin \text{set } ((\Pi)_V)$ 
              using Suc.IH[OF Suc.prems(1, 2)]
              by force
            have  $\tau\text{-Suc-}k\text{-is: } (?\tau ! \text{Suc } k) = \text{execute-parallel-operator } (?\tau ! k) (\pi ! k)$ 
              using trace-parallel-plan-step-effect-is[OF k-lt-length- $\tau$ -minus-one].
            have all-operators-applicable: are-all-operators-applicable  $(?\tau ! k) (\pi ! k)$ 
              and all-effects-consistent: are-all-operator-effects-consistent  $(\pi ! k)$ 
            using trace-parallel-plan-strips-operator-preconditions[OF k-lt-length- $\tau$ -minus-one]
              by simp+
            show ?case
              proof (rule iffI)
                assume  $\tau\text{-Suc-}k\text{-of-}v\text{-is-None: } (?\tau ! \text{Suc } k) v = \text{None}$ 
                show  $v \notin \text{set } ((\Pi)_V)$ 
                proof (rule ccontr)
                  assume  $\neg v \in \text{set } ((\Pi)_V)$ 
                  then have v-in-set-vs:  $v \in \text{set } ((\Pi)_V)$ 
                  by blast
                  show False
                  proof (cases  $\exists op \in \text{set } (\pi ! k)$ .

```

```

 $v \in \text{set}(\text{add-effects-of } op) \vee v \in \text{set}(\text{delete-effects-of } op)$ 
case True
then obtain op
  where op-in- $\pi_k$ :  $op \in \text{set}(\pi ! k)$ 
    and  $v \in \text{set}(\text{add-effects-of } op) \vee v \in \text{set}(\text{delete-effects-of } op)..$ 
then consider (A)  $v \in \text{set}(\text{add-effects-of } op)$ 
  | (B)  $v \in \text{set}(\text{delete-effects-of } op)$ 
  by blast
thus False
  using execute-parallel-operator-positive-effect-if[ $OF$ 
    all-operators-applicable all-effects-consistent op-in- $\pi_k$ ]
  execute-parallel-operator-negative-effect-if[ $OF$ 
    all-operators-applicable all-effects-consistent op-in- $\pi_k$ ]
   $\tau\text{-Suc-}k\text{-of-}v\text{-is-None}$   $\tau\text{-Suc-}k\text{-is}$ 
  by (cases, fastforce+)
next
  case False
  then have  $\forall op \in \text{set}(\pi ! k).$ 
     $v \notin \text{set}(\text{add-effects-of } op) \wedge v \notin \text{set}(\text{delete-effects-of } op)$ 
    by blast
  then have ( $\exists \tau ! Suc k$ )  $v = (\exists \tau ! k) v$ 
    using execute-parallel-operator-no-effect-if  $\tau\text{-Suc-}k\text{-is}$ 
    by fastforce
  then have  $v \notin \text{set}((\Pi)_V)$ 
    using IH  $\tau\text{-Suc-}k\text{-of-}v\text{-is-None}$ 
    by simp
  thus False
    using v-in-set-vs
    by blast
  qed
qed
next
assume v-notin-vs:  $v \notin \text{set}((\Pi)_V)$ 
{
  fix op
  assume op-in- $\pi_k$ :  $op \in \text{set}(\pi ! k)$ 
  {
    have  $1 < \text{length } ?\tau$ 
    using trace-parallel-plan-strips-length-gt-one-if[ $OF k\text{-lt-length-}\tau\text{-minus-one}$ ].
    then have  $0 < \text{length } ?\tau - 1$ 
      using k-lt-length- $\tau\text{-minus-one}$ 
      by linarith
    moreover have  $\text{length } ?\tau - 1 \leq \text{length } \pi$ 
      using length-trace-parallel-plan-strips-lte-length-plan-plus-one
    le-diff-conv
      by blast
    then have  $k < \text{length } \pi$ 
      using k-lt-length- $\tau\text{-minus-one}$ 
      by force
  }
}

```

```

      hence  $\pi ! k \in \text{set } \pi$ 
      by simp
    }
    then have op-in-ops:  $op \in \text{set } ?ops$ 
      using is-parallel-solution-for-problem-operator-set[OF assms(2) -
op-in-πk]
      by force
      hence  $v \notin \text{set } (\text{add-effects-of } op) \text{ and } v \notin \text{set } (\text{delete-effects-of } op)$ 
      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. $G \subseteq_m \text{execute-parallel-plan } I \pi$ —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:

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

proof –

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

show $?thesis$

proof (rule iffI)

assume $?LHS$
 thus $?RHS$

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

case ($Cons a \pi$)
 thus $?case$
 using Cons.prem

```

proof (cases are-all-operators-applicable I a  $\wedge$  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  $\wedge$  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.preds
    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.preds
    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.preds(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  $\gg$  op
  {
    fix x
    assume x  $\in$  set (op' # π)
    moreover have op  $\in$  set (op # op' # π)
      by simp
    moreover have  $\neg$ 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 ($\pi_1 @ \pi_2$)

```

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

```

```

lemma embedding-lemma-i:
  fixes I :: ('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

$$\begin{aligned} G &\subseteq_m \text{execute-serial-plan } ((\Pi)_I) \pi \\ &\implies G \subseteq_m \text{execute-serial-plan } ((\Pi)_I) (\text{embed } \pi) \end{aligned}$$

using lemma ; and moreover by showing that

$$\forall ops \in set (embed \pi). \forall op \in set ops. op \in (\Pi)_O$$

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

theorem *embedding-lemma*:

```

assumes is-valid-problem-strips  $\Pi$ 
and is-serial-solution-for-problem  $\Pi \pi$ 
shows is-parallel-solution-for-problem  $\Pi$  (embed  $\pi$ )
proof -
  have  $nb_1: \forall op \in \text{set } \pi. op \in \text{set } ((\Pi)_O)$ 
  using assms(2)
  unfolding is-serial-solution-for-problem-def list-all-iff ListMem-iff operators-of-def
  by blast
  {
    fix  $op$ 
    assume  $op \in \text{set } \pi$ 
    moreover have  $op \in \text{set } ((\Pi)_O)$ 
    using  $nb_1$  calculation
    by fast
    moreover have is-valid-operator-strips  $\Pi op$ 
    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  $(\lambda v. \neg \text{ListMem } v (\text{delete-effects-of } op))$  (add-effects-of  $op$ )
    and list-all  $(\lambda v. \neg \text{ListMem } v (\text{add-effects-of } op))$  (delete-effects-of  $op$ )
    using calculation(3)
    unfolding is-valid-operator-strips-def
    by meson+
    moreover have  $\neg \text{list-ex}$   $(\lambda v. \text{ListMem } v (\text{delete-effects-of } op))$  (add-effects-of  $op$ )
    and  $\neg \text{list-ex}$   $(\lambda v. \text{ListMem } v (\text{add-effects-of } op))$  (delete-effects-of  $op$ )
    using calculation(4, 5) not-list-ex-equals-list-all-not
    by blast+
    moreover have  $\neg \text{list-ex}$   $(\lambda v. \text{list-ex } ((=) v) (\text{delete-effects-of } op))$  (add-effects-of  $op$ )
    and  $\neg \text{list-ex}$   $(\lambda v. \text{list-ex } ((=) v) (\text{add-effects-of } op))$  (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  $nb_2 = \text{this}$ 
  moreover {
    have  $(\Pi)_G \subseteq_m \text{execute-serial-plan} ((\Pi)_I) \pi$ 
    using assms(2)
    unfolding is-serial-solution-for-problem-def
    by simp
  }

```

```

hence  $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I)$  (embed  $\pi$ )
      using execute-serial-plan-is-execute-parallel-plan-ii  $nb_2$ 
      by blast
}
moreover have  $\forall ops \in set (\text{embed } \pi)$ .  $\forall op \in set ops$ .  $op \in set ((\Pi)_{\mathcal{O}})$ 
      using embedding-lemma-iii[OF  $nb_1$ ].
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  $\Pi :: 'a \text{strips-problem}$ 
assumes  $\forall ops \in set \pi$ .  $\forall op \in set ops$ .  $op \in set ((\Pi)_{\mathcal{O}})$ 
shows  $\forall op \in set (\text{concat } \pi)$ .  $op \in set ((\Pi)_{\mathcal{O}})$ 
proof -
{
  fix  $op$ 
  assume  $op \in set (\text{concat } \pi)$ 
  moreover have  $op \in (\bigcup_{ops \in set \pi} set ops)$ 
    using calculation
    unfolding set-concat.
  then obtain  $ops$  where  $ops \in set \pi$  and  $op \in set ops$ 
    using UN-iff
    by blast
  ultimately have  $op \in set ((\Pi)_{\mathcal{O}})$ 
    using assms
    by blast
}
thus ?thesis..
qed

lemma flattening-lemma-ii:
fixes  $I :: 'variable \text{strips-state}$ 
assumes  $\forall ops \in set \pi$ .  $\exists op. ops = [op] \wedge \text{is-valid-operator-strips } \Pi op$ 
      and  $G \subseteq_m \text{execute-parallel-plan } I \pi$ 
shows  $G \subseteq_m \text{execute-serial-plan } I (\text{concat } \pi)$ 
proof -
  let  $?{\pi}' = \text{concat } \pi$ 

  {
    fix  $op$ 
    assume is-valid-operator-strips  $\Pi op$ 
    moreover have list-all  $(\lambda v. \neg \text{ListMem } v (\text{delete-effects-of } op)) (\text{add-effects-of } op)$ 
      and list-all  $(\lambda v. \neg \text{ListMem } v (\text{add-effects-of } op)) (\text{delete-effects-of } op)$ 
      using calculation(1)
      unfolding is-valid-operator-strips-def
  }

```

```

    by meson+
  moreover have  $\neg \text{list-ex} (\lambda v. \text{ListMem } v (\text{delete-effects-of } op)) (\text{add-effects-of } op)$ 
    and  $\neg \text{list-ex} (\lambda v. \text{ListMem } v (\text{add-effects-of } op)) (\text{delete-effects-of } op)$ 
    using calculation(2, 3) not-list-ex-equals-list-all-not
    by blast+
  moreover have  $\neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{delete-effects-of } op)) (\text{add-effects-of } op)$ 
    and  $\neg \text{list-ex} (\lambda v. \text{list-ex} ((=) v) (\text{add-effects-of } op)) (\text{delete-effects-of } op)$ 
    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  $\pi$  arbitrary: I)
  case (Cons ops  $\pi$ )
  obtain op where ops-is: ops = [op] and is-valid-op: is-valid-operator-strips
 $\Pi$  op
  using Cons.prem(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  $\gg$  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  $G \subseteq_m \text{execute-parallel-plan } ?J \pi$ 
        using Cons.prem(2) ops-is True
        by fastforce
    }
    moreover have execute-serial-plan I (concat (ops #  $\pi$ ))
      = execute-serial-plan ?J' (concat  $\pi$ )
      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.prem
  by force
next
  case False
  moreover have G ⊆m I
    using Cons.prem(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)ο)
      using assms(3)
    unfolding is-parallel-solution-for-problem-def list-all-iff ListMem-iff
      by force
    hence ∀ op ∈ set ?π'. op ∈ set ((Π)ο)
      using flattening-lemma-i
      by blast
  }
  moreover {
    {
      fix ops

```

```

assume  $ops \in set \pi$ 
moreover obtain  $op$  where  $ops = [op]$ 
  using  $assms(2)$  calculation
  by  $blast$ 
moreover have  $op \in set ((\Pi)_O)$ 
  using  $assms(3)$  calculation
  unfolding  $is-parallel-solution-for-problem-def$   $list-all-iff ListMem-iff$ 
  by  $force$ 
moreover have  $is-valid-operator-strips \Pi op$ 
  using  $assms(1)$  calculation(3)
  unfolding  $is-valid-problem-strips-def$   $Let-def$   $list-all-iff ListMem-iff$ 
  by  $simp$ 
ultimately have  $\exists op. ops = [op] \wedge is-valid-operator-strips \Pi op$ 
  by  $blast$ 
}
moreover have  $(\Pi)_G \subseteq_m execute-parallel-plan ((\Pi)_I) \pi$ 
  using  $assms(3)$ 
  unfolding  $is-parallel-solution-for-problem-def$ 
  by  $simp$ 
ultimately have  $(\Pi)_G \subseteq_m execute-serial-plan ((\Pi)_I) ?\pi'$ 
  using  $flattening-lemma-ii$ 
  by  $blast$ 
}
ultimately show  $is-serial-solution-for-problem \Pi ?\pi'$ 
  unfolding  $is-serial-solution-for-problem-def$   $list-all-iff ListMem-iff$ 
  by  $simp$ 
qed

```

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

$$(G \subseteq_m execute-parallel-plan I \pi) \\ = (G \subseteq_m last (trace-parallel-plan-strips I \pi))$$

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

```

theorem  $execute-parallel-plan-is-execute-sequential-plan-if:$ 
fixes  $I :: ('variable, bool) state$ 
assumes  $is-valid-problem \Pi$ 
and  $G \subseteq_m last (trace-parallel-plan-strips I \pi)$ 

```

```

and  $\forall k < \text{length } \pi.$ 
  are-all-operators-applicable (trace-parallel-plan-strips I  $\pi ! k$ ) ( $\pi ! k$ )
   $\wedge$  are-all-operator-effects-consistent ( $\pi ! k$ )
   $\wedge$  are-all-operators-non-interfering ( $\pi ! k$ )
shows  $G \subseteq_m \text{execute-serial-plan } I (\text{concat } \pi)$ 
using assms
proof (induction  $\pi$  arbitrary: I)
  case (Cons ops  $\pi$ )
    let  $?ops' = \text{take}(\text{length } \text{ops}) (\text{concat}(\text{ops} \# \pi))$ 
    let  $?J = \text{execute-parallel-operator } I \text{ops}$ 
      and  $?J' = \text{execute-serial-plan } I ?ops'$ 
    {
      have trace-parallel-plan-strips I  $\pi ! 0 = I$  and ( $\text{ops} \# \pi$ )  $! 0 = \text{ops}$ 
        unfolding trace-parallel-plan-strips-head-is-initial-state
        by simp+
      then have are-all-operators-applicable I  $\text{ops}$ 
        and are-all-operator-effects-consistent  $\text{ops}$ 
        and are-all-operators-non-interfering  $\text{ops}$ 
        using Cons.prems(3)
        by auto+
      then have trace-parallel-plan-strips I ( $\text{ops} \# \pi$ )
         $= I \# \text{trace-parallel-plan-strips } ?J \pi$ 
        by fastforce
    } note nb = this
    {
      have last (trace-parallel-plan-strips I ( $\text{ops} \# \pi$ ))
         $= \text{last}(\text{trace-parallel-plan-strips } ?J \pi)$ 
        using trace-parallel-plan-strips-last-cons-then nb
        by metis
      hence  $G \subseteq_m \text{last}(\text{trace-parallel-plan-strips } ?J \pi)$ 
        using Cons.prems(2)
        by force
    }
    moreover {
      fix  $k$ 
      assume  $k < \text{length } \pi$ 
      moreover have  $k + 1 < \text{length } (\text{ops} \# \pi)$ 
        using calculation
        by force
      moreover have  $\pi ! k = (\text{ops} \# \pi) ! (k + 1)$ 
        by simp
      ultimately have are-all-operators-applicable
        (trace-parallel-plan-strips ?J  $\pi ! k$ ) ( $\pi ! k$ )
        and are-all-operator-effects-consistent ( $\pi ! k$ )
        and are-all-operators-non-interfering ( $\pi ! k$ )
        using Cons.prems(3) nb
        by force+
    }
    ultimately have  $G \subseteq_m \text{execute-serial-plan } ?J (\text{concat } \pi)$ 

```

```

using Cons.IH[OF Cons.prems(1), of ?J]
by blast
moreover {
  have execute-serial-plan I (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 ((¬v+) [1000] 999)

```

```

operators-of :: ('variable, 'domain) sas-plus-operator list ( $\langle \neg \circ_+ \rangle$  [1000] 999)
initial-of :: ('variable, 'domain) state ( $\langle \neg I_+ \rangle$  [1000] 999)
goal-of :: ('variable, 'domain) state ( $\langle \neg G_+ \rangle$  [1000] 999)
range-of :: 'variable  $\rightarrow$  'domain list

definition range-of' :: ('variable, 'domain) sas-plus-problem  $\Rightarrow$  'variable  $\Rightarrow$  'domain
set ( $\langle \mathcal{R}_+ - \rightarrow 52 \rangle$ )
where
range-of'  $\Psi$  v  $\equiv$ 
  (case sas-plus-problem.range-of  $\Psi$  v of None  $\Rightarrow$  {}
  | Some as  $\Rightarrow$  set as)

definition to-precondition
:: ('variable, 'domain) sas-plus-operator  $\Rightarrow$  ('variable, 'domain) assignment list
where to-precondition  $\equiv$  precondition-of

definition to-effect
:: ('variable, 'domain) sas-plus-operator  $\Rightarrow$  ('variable, 'domain) Effect
where to-effect op  $\equiv$  [(v, a) . (v, a)  $\leftarrow$  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 ( $\langle \varrho \rangle$ )
where empty-operator  $\equiv$  () precondition-of = [], effect-of = [] []

definition is-valid-operator-sas-plus
:: ('variable, 'domain) sas-plus-problem  $\Rightarrow$  ('variable, 'domain) sas-plus-operator
 $\Rightarrow$  bool
where is-valid-operator-sas-plus  $\Psi$  op  $\equiv$  let
  pre = precondition-of op
  ; eff = effect-of op
  ; vs = variables-of  $\Psi$ 
  ; D = range-of  $\Psi$ 
  in list-all ( $\lambda(v, a)$ . ListMem v vs) pre
     $\wedge$  list-all ( $\lambda(v, a)$ . (D v  $\neq$  None)  $\wedge$  ListMem a (the (D v))) pre
     $\wedge$  list-all ( $\lambda(v, a)$ . ListMem v vs) eff
     $\wedge$  list-all ( $\lambda(v, a)$ . (D v  $\neq$  None)  $\wedge$  ListMem a (the (D v))) eff
     $\wedge$  list-all ( $\lambda(v, a)$ . list-all ( $\lambda(v', a')$ . v  $\neq$  v'  $\vee$  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  $\Psi$ 
 $\equiv$  let ops = operators-of  $\Psi$ 
  ; vs = variables-of  $\Psi$ 
  ; I = initial-of  $\Psi$ 

```

```

;  $G = \text{goal-of } \Psi$ 
;  $D = \text{range-of } \Psi$ 
in list-all ( $\lambda v. D v \neq \text{None}$ ) vs
 $\wedge$  list-all (is-valid-operator-sas-plus  $\Psi$ ) ops
 $\wedge$  ( $\forall v. I v \neq \text{None} \longleftrightarrow \text{ListMem } v \text{ vs}$ )
 $\wedge$  ( $\forall v. I v \neq \text{None} \longrightarrow \text{ListMem}(\text{the}(I v)) (\text{the}(D v))$ )
 $\wedge$  ( $\forall v. G v \neq \text{None} \longrightarrow \text{ListMem } v (\text{variables-of } \Psi)$ )
 $\wedge$  ( $\forall v. G v \neq \text{None} \longrightarrow \text{ListMem}(\text{the}(G v)) (\text{the}(D v))$ )

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  $\Psi$  v  $\neq$  None  $\wedge$  sas-plus-problem.range-of  $\Psi$  v  $\neq$  Some [])
 $\longleftrightarrow$  ( $\mathcal{R}_+ \Psi v$ )  $\neq$  {}
apply (cases sas-plus-problem.range-of  $\Psi$  v)
by (auto simp add: SAS-Plus-Representation.range-of'-def)

```

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

```

fixes  $\Psi :: ('v, 'd) \text{ sas-plus-problem}$ 
assumes is-valid-operator-sas-plus  $\Psi$  op
shows  $\forall (v, a) \in \text{set}(\text{precondition-of } op). v \in \text{set}((\Psi)_{v+})$ 
and  $\forall (v, a) \in \text{set}(\text{precondition-of } op). (\mathcal{R}_+ \Psi v) \neq \{} \wedge a \in \mathcal{R}_+ \Psi v$ 
and  $\forall (v, a) \in \text{set}(\text{effect-of } op). v \in \text{set}((\Psi)_{v+})$ 
and  $\forall (v, a) \in \text{set}(\text{effect-of } op). (\mathcal{R}_+ \Psi v) \neq \{} \wedge a \in \mathcal{R}_+ \Psi v$ 
and  $\forall (v, a) \in \text{set}(\text{precondition-of } op). \forall (v', a') \in \text{set}(\text{precondition-of } op). v$ 

```

```

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

```

```

and  $\forall (v, a) \in \text{set}(\text{precondition-of } op). (\mathcal{R}_+ \Psi v) \neq \{\} \wedge a \in \mathcal{R}_+ \Psi v$ 
and  $\forall (v, a) \in \text{set}(\text{effect-of } op). v \in \text{set}((\Psi)_{V+})$ 
and  $\forall (v, a) \in \text{set}(\text{effect-of } op). (\mathcal{R}_+ \Psi v) \neq \{\} \wedge a \in \mathcal{R}_+ \Psi v$ 
and  $\forall (v, a) \in \text{set}(\text{precondition-of } op). \forall (v', a') \in \text{set}(\text{precondition-of } op). v \neq v' \vee a = a'$ 
and  $\forall (v, a) \in \text{set}(\text{effect-of } op).$ 
 $\forall (v', a') \in \text{set}(\text{effect-of } op). v \neq v' \vee a = a'$ 
by blast+
qed

```

```

lemma is-valid-problem-sas-plus-then:
fixes  $\Psi::('v,'d)$  sas-plus-problem
assumes is-valid-problem-sas-plus  $\Psi$ 
shows  $\forall v \in \text{set}((\Psi)_{V+}). (\mathcal{R}_+ \Psi v) \neq \{\}$ 
and  $\forall op \in \text{set}((\Psi)_{O+}). \text{is-valid-operator-sas-plus } \Psi \text{ op}$ 
and  $\text{dom } ((\Psi)_{I+}) = \text{set}((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } ((\Psi)_{I+}). \text{the } (((\Psi)_{I+}) v) \in \mathcal{R}_+ \Psi v$ 
and  $\text{dom } ((\Psi)_{G+}) \subseteq \text{set}((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } ((\Psi)_{G+}). \text{the } (((\Psi)_{G+}) v) \in \mathcal{R}_+ \Psi v$ 
proof –
let ?vs = sas-plus-problem.variables-of  $\Psi$ 
and ?ops = sas-plus-problem.operators-of  $\Psi$ 
and ?I = sas-plus-problem.initial-of  $\Psi$ 
and ?G = sas-plus-problem.goal-of  $\Psi$ 
and ?D = sas-plus-problem.range-of  $\Psi$ 
{
  fix v
  have (?D v ≠ None ∧ ?D v ≠ Some [])  $\longleftrightarrow$   $((\mathcal{R}_+ \Psi v) \neq \{\})$ 
    by (cases ?D v; (auto simp: range-of'-def))
  } note nb = this
  have nb1:  $\forall v \in \text{set} \ ?vs. ?D v \neq \text{None}$ 
    and  $\forall op \in \text{set} \ ?ops. \text{is-valid-operator-sas-plus } \Psi \text{ op}$ 
    and  $\forall v. (?I v \neq \text{None}) = (v \in \text{set} \ ?vs)$ 
    and nb2:  $\forall v. ?I v \neq \text{None} \longrightarrow \text{the } (?I v) \in \text{set}(\text{the } (?D v))$ 
    and  $\forall v. ?G v \neq \text{None} \longrightarrow v \in \text{set} \ ?vs$ 
    and nb3:  $\forall v. ?G v \neq \text{None} \longrightarrow \text{the } (?G v) \in \text{set}(\text{the } (?D v))$ 
    using assms
    unfolding SAS-Plus-Representation.is-valid-problem-sas-plus-def Let-def
      list-all-iff ListMem-iff
    by argo+
  then have G3:  $\forall op \in \text{set}((\Psi)_{O+}). \text{is-valid-operator-sas-plus } \Psi \text{ op}$ 
  and G4:  $\text{dom } ((\Psi)_{I+}) = \text{set}((\Psi)_{V+})$ 
  and G5:  $\text{dom } ((\Psi)_{G+}) \subseteq \text{set}((\Psi)_{V+})$ 
  unfolding variables-of-def operators-of-def
  by auto+
  moreover {
    fix v
    assume  $v \in \text{set}((\Psi)_{V+})$ 
  }

```

```

then have ?D v ≠ None
  using nb1
  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 nb2
  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 nb3
  by auto
}
ultimately show ∀ v ∈ set ((Ψ)V+). (R+ Ψ v) ≠ {}

```

```

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

end

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

```

5 SAS+ Semantics

5.1 Serial Execution Semantics

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

```

primrec execute-serial-plan-sas-plus
where execute-serial-plan-sas-plus  $s [] = s$ 
| 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  $\Rightarrow$  ('variable, 'domain) sas-plus-plan  $\Rightarrow$ 
bool
where is-serial-solution-for-problem  $\Psi \psi$ 
 $\equiv$  let
 $I = sas\text{-plus}\text{-problem.initial-of } \Psi$ 
 $; G = sas\text{-plus}\text{-problem.goal-of } \Psi$ 
 $; ops = sas\text{-plus}\text{-problemoperators-of } \Psi$ 
in  $G \subseteq_m execute\text{-serial}\text{-plan}\text{-sas}\text{-plus } I \psi$ 
 $\wedge list\text{-all } (\lambda op. ListMem op ops) \psi$ 

```

```

context
begin

private lemma execute-operator-sas-plus-effect-i:
assumes is-operator-applicable-in  $s op$ 

```

```

and  $\forall (v, a) \in set(\text{effect-of } op)$ .  $\forall (v', a') \in set(\text{effect-of } op)$ .
     $v \neq v' \vee a = a'$ 
and  $(v, a) \in set(\text{effect-of } op)$ 
shows  $(s \gg_+ op) v = \text{Some } a$ 
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  $\forall (v', a') \in set(\text{effect-of } op)$ .  $v' \neq v$ 
shows  $(s \gg_+ op) v = s v$ 
proof -
  let ?effect = effect-of op
  {
    have  $v \notin \text{fst}(\text{set } ?\text{effect})$ 
    using assms(2)
    by fastforce
    then have  $v \notin \text{dom}(\text{map-of } ?\text{effect})$ 
    using dom-map-of-conv-image-fst[of ?effect]
    by argo
    hence  $(s ++ \text{map-of } ?\text{effect}) v = s v$ 
    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:

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

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

```

theorem execute-operator-sas-plus-effect:
assumes is-operator-applicable-in s op
and  $\forall (v, a) \in set(\text{effect-of } op)$ .
     $\forall (v', a') \in set(\text{effect-of } op)$ .  $v \neq v' \vee a = a'$ 

```

```

shows ( $v, a \in \text{set}(\text{effect-of } op)$ )
   $\rightarrow (s \gg_+ op) v = \text{Some } a$ 
and ( $\forall a. (v, a) \notin \text{set}(\text{effect-of } op)$ )
   $\rightarrow (s \gg_+ op) v = s v$ 
proof -
  show ( $v, a \in \text{set}(\text{effect-of } op)$ )
     $\rightarrow (s \gg_+ op) v = \text{Some } a$ 
    using execute-operator-sas-plus-effect-i[OF assms(1, 2)]
    by blast
next
  show ( $\forall a. (v, a) \notin \text{set}(\text{effect-of } op)$ )
     $\rightarrow (s \gg_+ op) v = s v$ 
    using execute-operator-sas-plus-effect-ii[OF assms(1)]
    by blast
qed
end

```

5.2 Parallel Execution Semantics

```

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

definition are-all-operators-applicable-in
  :: ('variable, 'domain) state
   $\Rightarrow$  ('variable, 'domain) sas-plus-operator list
   $\Rightarrow$  bool
where are-all-operators-applicable-in  $s$   $ops$ 
   $\equiv$  list-all (is-operator-applicable-in  $s$ )  $ops$ 

definition are-operator-effects-consistent
  :: ('variable, 'domain) sas-plus-operator
   $\Rightarrow$  ('variable, 'domain) sas-plus-operator
   $\Rightarrow$  bool
where are-operator-effects-consistent  $op$   $op'$ 
   $\equiv$  let
    effect = effect-of  $op$ 
    ; effect' = effect-of  $op'$ 
    in list-all ( $\lambda(v, a). \text{list-all}(\lambda(v', a'). v \neq v' \vee a = a') \text{effect}'$ ) effect

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

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  $\Psi \psi$ 
 $\equiv let$ 
  G = sas-plus-problem.goal-of  $\Psi$ 
  ; I = sas-plus-problem.initial-of  $\Psi$ 
  ; Ops = sas-plus-problem.operators-of  $\Psi$ 

```

in $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi$
 $\wedge \text{list-all } (\lambda \text{ops}. \text{list-all } (\lambda \text{op}. \text{ListMem op } \text{Ops}) \text{ ops}) \psi$

```
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 $\tau \equiv \text{trace-parallel-plan-sas-plus } I \psi$ be a trace of a parallel SAS+ plan ψ with initial state I , then

- the head of the trace $\tau ! 0$ is the initial state of the problem (lemma ??); moreover,
- for all but the last element of the trace—i.e. elements with index $k < \text{length } \tau - 1$ —the parallel operator $\pi ! k$ is executable (lemma ??); and finally,
- for all $k < \text{length } \tau$, the parallel execution of the plan prefix $\text{take } k \psi$ with initial state I equals the k -th element of the trace $\tau ! k$ (lemma ??).

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

```
lemma length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one:
  shows length (trace-parallel-plan-sas-plus I \psi) \leq length \psi + 1
proof (induction \psi arbitrary: I)
  case (Cons a \psi)
  then show ?case
```

```

proof (cases are-all-operators-applicable-in I a  $\wedge$  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) ≤ length ψ + 1
    using Cons.IH[of ?I']
    by blast
    ultimately have length (trace-parallel-plan-sas-plus I (a # ψ)) ≤ 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 ≤ 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-parallel-plan-sas-plus I π) - 1
  shows are-all-operators-applicable-in (trace-parallel-plan-sas-plus I π ! k) (π ! k)
     $\wedge$  are-all-operator-effects-consistent (π ! k)
using assms
proof (induction π arbitrary: I k)

```

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

```

case (Cons a π)
then show ?case
proof (cases are-all-operators-applicable-in I a ∧ are-all-operator-effects-consistent a)
    case True
        have trace-parallel-plan-sas-plus-cons: trace-parallel-plan-sas-plus I (a # π)
            = I # trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) π
        using True
        by simp
    then show ?thesis
    proof (cases k)
        case 0
            have trace-parallel-plan-sas-plus I (a # π) ! 0 = I
            using trace-parallel-plan-sas-plus-cons
            by simp
            moreover have (a # π) ! 0 = a
            by simp
            ultimately show ?thesis
            using True 0
            by presburger
    next
        case (Suc k')
            have trace-parallel-plan-sas-plus I (a # π) ! Suc k'
                = trace-parallel-plan-sas-plus (execute-parallel-operator-sas-plus I a) π ! k'
            using trace-parallel-plan-sas-plus-cons
            by simp
            moreover have (a # π) ! Suc k' = π ! k'
            by simp
            moreover {
                let ?I' = execute-parallel-operator-sas-plus I a
                have length (trace-parallel-plan-sas-plus I (a # π))
                    = 1 + 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) π ! k'
                     (π ! k')
                    ∧ are-all-operator-effects-consistent (π ! 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 $k < \text{length}(\text{trace-parallel-plan-sas-plus } I \psi) - 1$
shows $\text{trace-parallel-plan-sas-plus } I \psi ! \text{Suc } k$
 $= \text{execute-parallel-operator-sas-plus}(\text{trace-parallel-plan-sas-plus } I \psi ! k)(\psi ! k)$

proof –
let $?τ = \text{trace-parallel-plan-sas-plus } I \psi$
let $?τ_k = ?τ ! k$
and $?τ'_k = ?τ ! \text{Suc } k$
— NOTE rewrite the goal using the subplan formulation to be able. This allows us to make the initial state arbitrary.
{
have suc-k-lt-length-τ: $\text{Suc } k < \text{length } ?τ$
 using assms
 by linarith
hence $?τ'_k = \text{execute-parallel-plan-sas-plus } I (\text{take}(\text{Suc } k) \psi)$
 using trace-parallel-plan-sas-plus-prefix[of Suc k]
 by blast
}
note rewrite-goal = this
have execute-parallel-plan-sas-plus I (take (Suc k) ψ)
 $= \text{execute-parallel-operator-sas-plus}(\text{trace-parallel-plan-sas-plus } I \psi ! k)(\psi ! k)$

```

using assms
proof (induction k arbitrary: I  $\psi$ )
  case 0
    obtain ops  $\psi'$  where  $\psi$ -is:  $\psi = \text{ops} \# \psi'$ 
      using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
0.prems]
      by force
    {
      have take (Suc 0)  $\psi = [\psi ! 0]$ 
        using  $\psi$ -is
        by simp
      hence execute-parallel-plan-sas-plus I (take (Suc 0)  $\psi$ )
        = execute-parallel-plan-sas-plus I [ $\psi ! 0$ ]
        by argo
    }
    moreover {
      have trace-parallel-plan-sas-plus I  $\psi ! 0 = I$ 
        using trace-parallel-plan-sas-plus-head-is-initial-state.
      moreover {
        have are-all-operators-applicable-in I ( $\psi ! 0$ )
          and are-all-operator-effects-consistent ( $\psi ! 0$ )
        using trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds[OF
          0.prems] calculation
        by argo+
        then have execute-parallel-plan-sas-plus I [ $\psi ! 0$ ]
          = execute-parallel-operator-sas-plus I ( $\psi ! 0$ )
        by simp
      }
      ultimately have execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus
      I  $\psi ! 0)$ 
        ( $\psi ! 0$ )
        = execute-parallel-plan-sas-plus I [ $\psi ! 0$ ]
        by argo
    }
    ultimately show ?case
      by argo
  next
    case (Suc k)
    obtain ops  $\psi'$  where  $\psi$ -is:  $\psi = \text{ops} \# \psi'$ 
      using plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements[OF
Suc.prems]
      by blast
    let ?I' = execute-parallel-operator-sas-plus I ops
    have execute-parallel-plan-sas-plus I (take (Suc (Suc k))  $\psi$ )
      = execute-parallel-plan-sas-plus ?I' (take (Suc k)  $\psi'$ )
    using Suc.prem  $\psi$ -is
    by fastforce
    moreover {
      thm Suc.IH[of ]
    }
  
```

```

have length (trace-parallel-plan-sas-plus I  $\psi$ )
  = 1 + length (trace-parallel-plan-sas-plus ?I'  $\psi'$ )
  using  $\psi$ -is Suc.prem
  by fastforce
moreover have k < length (trace-parallel-plan-sas-plus ?I'  $\psi')$  - 1
  using Suc.prem calculation
  by fastforce
ultimately have execute-parallel-plan-sas-plus ?I' (take (Suc k)  $\psi')$  =
  execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus ?I'  $\psi'$  ! k)
  ( $\psi'$  ! k)
  using Suc.IH[of ?I'  $\psi'$ ]
  by blast
}
moreover have execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus
?I'  $\psi'$  ! k)
  ( $\psi'$  ! k)
  = execute-parallel-operator-sas-plus (trace-parallel-plan-sas-plus I  $\psi$  ! Suc k)
  ( $\psi$  ! Suc k)
  using Suc.prem  $\psi$ -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  $\subseteq_m$  execute-parallel-plan-sas-plus I  $\psi$ 
   $\longleftrightarrow$  G  $\subseteq_m$  last (trace-parallel-plan-sas-plus I  $\psi$ )
proof -
  let ? $\tau$  = trace-parallel-plan-sas-plus I  $\psi$ 
  show ?thesis
  proof (rule iffI)
    assume G  $\subseteq_m$  execute-parallel-plan-sas-plus I  $\psi$ 
    thus G  $\subseteq_m$  last ? $\tau$ 
    proof (induction  $\psi$  arbitrary: I)
      — NOTE Base case follows from simplification.
      case (Cons ops  $\psi$ )
      show ?case
      proof (cases are-all-operators-applicable-in I ops
         $\wedge$  are-all-operator-effects-consistent ops)
        case True
        let ?s = execute-parallel-operator-sas-plus I ops

```

```

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

```

```

using 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.prem
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.prem 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)ο+)
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-parallel-operator-sas-plus s [op]
proof -
  have are-all-operators-applicable-in s [op]
  unfolding are-all-operators-applicable-in-def
    SAS-Plus-Representation.execute-operator-sas-plus-def
  is-operator-applicable-in-def SAS-Plus-Representation.is-operator-applicable-in-def
    list-all-iff
  using assms(1)
  by fastforce
  moreover have are-all-operator-effects-consistent [op]
  unfolding are-all-operator-effects-consistent-def list-all-iff
  using assms(2)
  by fastforce
  ultimately show ?thesis
  unfolding execute-parallel-operator-sas-plus-def execute-operator-sas-plus-def
  by simp
qed

lemma execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii:
  fixes I :: ('variable, 'domain) state
  assumes  $\forall op \in set \psi$ . are-operator-effects-consistent op op
    and  $G \subseteq_m execute-serial-plan-sas-plus I \psi$ 
  shows  $G \subseteq_m execute-parallel-plan-sas-plus I (embed \psi)$ 
  using assms
proof (induction  $\psi$  arbitrary: I)
  case (Cons op  $\psi$ )
  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  $G \subseteq_m execute-serial-plan-sas-plus ?J \psi$ 
      using Cons.prem(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  $\psi$ )
= execute-parallel-plan-sas-plus ?J' (embed  $\psi$ )
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  $\neg$ is-operator-applicable-in I op
    using calculation
    unfolding are-all-operators-applicable-in-def
      SAS-Plus-Representation.is-operator-applicable-in-def list-all-iff
    by fastforce
  moreover have  $G \subseteq_m I$ 
    using Cons.prems(2) calculation(2)
    unfolding is-operator-applicable-in-def
    by simp
  moreover have execute-parallel-plan-sas-plus I ([op] # embed  $\psi$ ) = I
    using calculation(1)
    by fastforce
  ultimately show ?thesis
    by force
  qed
qed simp

lemma execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iii:
assumes is-valid-problem-sas-plus  $\Psi$ 
and is-serial-solution-for-problem  $\Psi \psi$ 
and op  $\in$  set  $\psi$ 
shows are-operator-effects-consistent op op
proof -
  have op  $\in$  set  $((\Psi)_{O+})$ 
  using assms(2) assms(3)

```

```

unfolding is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
by fastforce
then have is-valid-operator-sas-plus  $\Psi$  op
  using is-valid-problem-sas-plus-then(2) assms(1, 3)
  by auto
thus ?thesis
unfolding are-operator-effects-consistent-def Let-def list-all-iff ListMem-iff
  using is-valid-operator-sas-plus-then(6)
  by fast
qed

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

theorem execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus:
  assumes is-valid-problem-sas-plus  $\Psi$ 
    and is-serial-solution-for-problem  $\Psi \psi$ 
  shows is-parallel-solution-for-problem  $\Psi (embed \psi)$ 
proof -
  let ?ops = sas-plus-problem.operators-of  $\Psi$ 
  and ? $\psi'$  = embed  $\psi$ 
  {
    thm execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii[ $OF$ ]
    have  $(\Psi)_{G+} \subseteq_m execute-serial-plan-sas-plus ((\Psi)_{I+}) \psi$ 
      using assms(2)
    unfolding is-serial-solution-for-problem-def Let-def
      by simp
    moreover have  $\forall op \in set \psi. are-operator-effects-consistent op op$ 
    using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iii[ $OF$  assms]..
    ultimately have  $(\Psi)_{G+} \subseteq_m execute-parallel-plan-sas-plus ((\Psi)_{I+}) ?\psi'$ 
      using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii
      by blast
  }

```

```

}

moreover {
  have  $\forall op \in set \psi. op \in set ((\Psi)_{O+})$ 
  using assms(2)
  unfolding is-serial-solution-for-problem-def Let-def list-all-iff ListMem-iff
  by fastforce
  hence  $\forall ops \in set ?\psi'. \forall op \in set ops. op \in set ((\Psi)_{O+})$ 
  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  $\Psi :: ('v, 'd) sas-plus-problem$ 
  assumes  $\forall ops \in set \pi. \forall op \in set ops. op \in set ((\Psi)_{O+})$ 
  shows  $\forall op \in set (concat \pi). op \in set ((\Psi)_{O+})$ 
proof -
{
  fix op
  assume op ∈ set (concat π)
  moreover have op ∈ ( $\bigcup ops \in set \pi. set ops$ )
    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  $\forall ops \in set \psi. \exists op. ops = [op] \wedge is-valid-operator-sas-plus \Psi op$ 
  and  $G \subseteq_m execute-parallel-plan-sas-plus I \psi$ 
  shows  $G \subseteq_m execute-serial-plan-sas-plus I (concat \psi)$ 
proof -
  show ?thesis
  using assms
  proof (induction  $\psi$  arbitrary: I)
    case (Cons ops ψ)
    obtain op where ops-is: ops = [op] and is-valid-op: is-valid-operator-sas-plus

```

```

 $\Psi \ op$ 
  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  $(\Psi)_{G+} \subseteq_m \text{execute-parallel-plan-sas-plus } ((\Psi)_{I+}) \psi$ 
  using assms(3)
  unfolding is-parallel-solution-for-problem-def
  by fastforce
ultimately have  $(\Psi)_{G+} \subseteq_m \text{execute-serial-plan-sas-plus } ((\Psi)_{I+}) ?\psi'$ 
  using flattening-lemma-ii
  by blast
}
ultimately show is-serial-solution-for-problem  $\Psi ?\psi'$ 
  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  $\Psi$  op
  and  $(v, a) \in \text{set}(\text{precondition-of } \text{op}) \vee (v, a) \in \text{set}(\text{effect-of } \text{op})$ 
  shows  $(\mathcal{R}_+ \Psi v) = \text{set}(\text{the}(\text{sas-plus-problem.range-of } \Psi v))$ 

```

```

proof –

```

```

  consider (A)  $(v, a) \in \text{set}(\text{precondition-of } \text{op})$ 
  | (B)  $(v, a) \in \text{set}(\text{effect-of } \text{op})$ 
  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  $\Psi$ 
   $v \in \text{set}((\Psi)_{V+})$ 
  shows set(the(sas-plus-problem.range-of  $\Psi$  v)) =  $\mathcal{R}_+ \Psi v$ 
proof-
  have  $v \in \text{set}((\Psi)_{V+})$ 
    using assms(2)
    unfolding variables-of-def.
  moreover have  $(\mathcal{R}_+ \Psi v) \neq \{\}$ 
    using assms(1) calculation is-valid-problem-sas-plus-then(1)
    by blast
  moreover have sas-plus-problem.range-of  $\Psi v \neq \text{None}$ 
    and sas-plus-problem.range-of  $\Psi v \neq \text{Some } []$ 
    using calculation(2) range-of-not-empty
    unfolding range-of-def
    by fast+
  ultimately show ?thesis
    unfolding option.case-eq-if range-of'-def
    by force
qed

lemma sublocale-sas-plus-finite-domain-representation-ii:
  fixes  $\Psi$ ::('v,'d) sas-plus-problem
  assumes is-valid-problem-sas-plus  $\Psi$ 
  shows  $\forall v \in \text{set}((\Psi)_{V+}). (\mathcal{R}_+ \Psi v) \neq \{\}$ 
    and  $\forall op \in \text{set}((\Psi)_{O+}). \text{is-valid-operator-sas-plus } \Psi op$ 
    and  $\text{dom}((\Psi)_{I+}) = \text{set}((\Psi)_{V+})$ 
    and  $\forall v \in \text{dom}((\Psi)_{I+}). \text{the } (((\Psi)_{I+}) v) \in \mathcal{R}_+ \Psi v$ 
    and  $\text{dom}((\Psi)_{G+}) \subseteq \text{set}((\Psi)_{V+})$ 
    and  $\forall v \in \text{dom}((\Psi)_{G+}). \text{the } (((\Psi)_{G+}) v) \in \mathcal{R}_+ \Psi v$ 
  using is-valid-problem-sas-plus-then[OF assms]
  by auto

end

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

```

6 SAS+/STRIPS Equivalence

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

6.1 Translation of SAS+ Problems to STRIPS Problems

```

definition possible-assignments-for
:: ('variable, 'domain) sas-plus-problem  $\Rightarrow$  'variable  $\Rightarrow$  ('variable  $\times$  'domain) list
where possible-assignments-for  $\Psi$  v  $\equiv$  [(v, a). a  $\leftarrow$  the (range-of  $\Psi$  v)]
```

```

definition all-possible-assignments-for
:: ('variable, 'domain) sas-plus-problem  $\Rightarrow$  ('variable  $\times$  'domain) list
where all-possible-assignments-for  $\Psi$ 
 $\equiv$  concat [possible-assignments-for  $\Psi$  v. v  $\leftarrow$  variables-of  $\Psi$ ]
```

```

definition state-to-strips-state
:: ('variable, 'domain) sas-plus-problem
 $\Rightarrow$  ('variable, 'domain) state
 $\Rightarrow$  ('variable, 'domain) assignment strips-state
( $\varphi_S - \rightarrow 99$ )
where state-to-strips-state  $\Psi$  s
 $\equiv$  let defined = filter ( $\lambda v. s v \neq \text{None}$ ) (variables-of  $\Psi$ ) in
map-of (map ( $\lambda(v, a). ((v, a), \text{the}(s v) = a)$ )
(concat [possible-assignments-for  $\Psi$  v. v  $\leftarrow$  defined]))
```

```

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

```

definition sas-plus-problem-to-strips-problem
:: ('variable, 'domain) sas-plus-problem  $\Rightarrow$  ('variable, 'domain) assignment strips-problem
( $\varphi - \rightarrow 99$ )
where sas-plus-problem-to-strips-problem  $\Psi$   $\equiv$  let
vs = [as. v  $\leftarrow$  variables-of  $\Psi$ , as  $\leftarrow$  (possible-assignments-for  $\Psi$ ) v]
; ops = map (sasp-op-to-strips  $\Psi$ ) (operators-of  $\Psi$ )
; I = state-to-strips-state  $\Psi$  (initial-of  $\Psi$ )
; G = state-to-strips-state  $\Psi$  (goal-of  $\Psi$ )
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
( $\varphi_P^{-1} \dashv\vdash 99$ )
where sas-plus-parallel-plan-to-strips-parallel-plan  $\Psi \psi$ 
≡ [[sasp-op-to-strips  $\Psi$  op. op ← ops]. ops ←  $\psi$ ]

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

definition strips-op-to-sasp
:: ('variable, 'domain) sas-plus-problem
⇒ ('variable × 'domain) strips-operator
⇒ ('variable, 'domain) sas-plus-operator
( $\varphi_O^{-1} \dashv\vdash 99$ )
where strips-op-to-sasp  $\Psi op$ 
≡ 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
( $\varphi_P^{-1} \dashv\vdash 99$ )
where strips-parallel-plan-to-sas-plus-parallel-plan  $\Pi \pi$ 
≡ [[strips-op-to-sasp  $\Pi$  op. op ← ops]. ops ←  $\pi$ ]

```

To set up the equivalence proof context, we declare a common locale for both the STRIPS and SAS+ formalisms and make it a sublocale of both locale as well as . The declaration itself is omitted for brevity since it basically just joins locales and while renaming the locale parameter to avoid name clashes. The sublocale proofs are shown below.⁵

⁵We 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

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

context
begin

— Set-up simp rules.

lemma[simp]:

$(\varphi \Psi) = (\text{let}$

$vs = [\text{as. } v \leftarrow \text{variables-of } \Psi, \text{as} \leftarrow (\text{possible-assignments-for } \Psi) v]$

$; ops = \text{map } (\text{sasp-op-to-strips } \Psi) (\text{operators-of } \Psi)$

$; I = \text{state-to-strips-state } \Psi (\text{initial-of } \Psi)$

$; G = \text{state-to-strips-state } \Psi (\text{goal-of } \Psi)$

$\text{in STRIPS-Representation.problem-for } vs \text{ ops } I \text{ } G)$

and $(\varphi_S \Psi s)$

$= (\text{let defined} = \text{filter } (\lambda v. s v \neq \text{None}) (\text{variables-of } \Psi) \text{ in}$

$\text{map-of } (\text{map } (\lambda(v, a). ((v, a), \text{the } (s v) = a))$

$(\text{concat } [\text{possible-assignments-for } \Psi v. v \leftarrow \text{defined}])))$

and $(\varphi_O \Psi op)$

$= (\text{let}$

$pre = \text{precondition-of } op$

$; add = \text{effect-of } op$

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

$\text{in STRIPS-Representation.operator-for } pre \text{ add } delete)$

and $(\varphi_P \Psi \psi) = [[\varphi_O \Psi op. op \leftarrow ops]. ops \leftarrow \psi]$

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

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

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

$\text{precondition} = \text{strips-operator.precondition-of } op'$

$; effect = \text{strips-operator.add-effects-of } op'$

$\text{in } () \text{ precondition-of } = \text{precondition, effect-of } = \text{effect } ()$

and $(\varphi_P^{-1} \Psi \pi) = [[\varphi_O^{-1} \Psi op. op \leftarrow ops]. ops \leftarrow \pi]$

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

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

initial-of is ambiguous and will therefore not be bound to either *strips-problem.initial-of* or *sas-plus-problem.initial-of*. Isabelle in fact considers it to be a free variable in this case. We also qualify the parent locales in the sublocale declarations by adding **strips:** and **sas_plus:** before the respective parent locale identifiers.

*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 $\forall v \in \text{set}((\Psi)_{V+}). v \in \text{dom}(\text{sas-plus-problem.range-of } \Psi)$
using assms(1) is-valid-problem-sas-plus-then(1)
unfolding is-valid-problem-sas-plus-def
by (meson domIff list.pred-set)

lemma possible-assignments-for-set-is:
assumes $v \in \text{dom}(\text{sas-plus-problem.range-of } \Psi)$
shows $\text{set}(\text{possible-assignments-for } \Psi v) = \{(v, a) \mid a. a \in \mathcal{R}_+ \Psi v\}$

proof –
have sas-plus-problem.range-of $\Psi v \neq \text{None}$
using assms(1)
by auto
thus ?thesis
unfolding possible-assignments-for-def
by fastforce
qed

lemma all-possible-assignments-for-set-is:
assumes $\forall v \in \text{set}((\Psi)_{V+}). \text{range-of } \Psi v \neq \text{None}$
shows $\text{set}(\text{all-possible-assignments-for } \Psi) = (\bigcup v \in \text{set}((\Psi)_{V+}). \{(v, a) \mid a. a \in \mathcal{R}_+ \Psi v\})$
proof –
let ?vs = variables-of Ψ
have $\text{set}(\text{all-possible-assignments-for } \Psi) = (\bigcup(\text{set}(\lambda v. \text{map}(\lambda(v, a). (v, a))(\text{possible-assignments-for } \Psi v)) \setminus \text{set} ?vs))$
unfolding all-possible-assignments-for-def set-concat
using set-map
by auto
also have ... = $(\bigcup((\lambda v. \text{set}(\text{possible-assignments-for } \Psi v)) \setminus \text{set} ?vs))$
using image-comp set-map
by simp

also have ... = $(\bigcup((\lambda v. \{(v, a) \mid a. a \in \mathcal{R}_+ \Psi v\}) \setminus \text{set} ?vs))$
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  $\forall v \in \text{set } ((\Psi)_{v+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$ 
shows  $\text{set } (\text{concat}$ 

$$[\text{possible-assignments-for } \Psi \ v. v \leftarrow \text{filter } (\lambda v. s v \neq \text{None}) \ (\text{variables-of } \Psi)])$$

 $= (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{v+}) \wedge s v \neq \text{None} \}.$ 

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

proof –
let  $?vs = \text{variables-of } \Psi$ 
let  $?defined = \text{filter } (\lambda v. s v \neq \text{None}) ?vs$ 
let  $?l = \text{concat } [\text{possible-assignments-for } \Psi \ v. v \leftarrow ?defined]$ 
have  $nb: \text{set } ?defined = \{ v \mid v. v \in \text{set } ((\Psi)_{v+}) \wedge s v \neq \text{None} \}$ 
unfolding  $\text{set-filter}$ 
by force
have  $\text{set } ?l = \bigcup (\text{set} \ ' \text{set } (\text{map } (\text{possible-assignments-for } \Psi) ?defined))$ 
unfolding  $\text{set-concat image-Union}$ 
by blast
also have  $\dots = \bigcup (\text{set} \ ' (\text{possible-assignments-for } \Psi) \ ' \text{set } ?defined)$ 
unfolding  $\text{set-map}$ 
by blast
also have  $\dots = (\bigcup v \in \text{set } ?defined. \text{set } (\text{possible-assignments-for } \Psi v))$ 
by blast
also have  $\dots = (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{v+}) \wedge s v \neq \text{None} \}.$ 

$$\text{set } (\text{possible-assignments-for } \Psi v))$$

using  $nb$ 
by argo
finally show  $?thesis$ 
using  $\text{possible-assignments-for-set-is}$ 
is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)
by fastforce
qed

```

lemma *state-to-strips-state-dom-is*:

— NOTE A transformed state is defined on all possible assignments for all variables defined in the original state.

```

assumes is-valid-problem-sas-plus  $\Psi$ 
shows  $\text{dom } (\varphi_S \Psi s)$ 
 $= (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{v+}) \wedge s v \neq \text{None} \}.$ 

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

proof –
let  $?vs = \text{variables-of } \Psi$ 
let  $?l = \text{concat } [\text{possible-assignments-for } \Psi \ v. v \leftarrow \text{filter } (\lambda v. s v \neq \text{None}) ?vs]$ 
have  $nb: \forall v \in \text{set } ((\Psi)_{v+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$ 
using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of assms(1)
by fastforce
have  $\text{dom } (\varphi_S \Psi s) = \text{fst} \ ' \text{set } (\text{map } (\lambda(v, a). ((v, a), \text{the } (s v) = a)) ?l)$ 
unfolding state-to-strips-state-def
SAS-Plus-STRIPS.state-to-strips-state-def
using  $\text{dom-map-of-conv-image-fst}[ \text{of map } (\lambda(v, a). ((v, a), \text{the } (s v) = a)) ?l ]$ 
by presburger
also have  $\dots = \text{fst} \ ' (\lambda(v, a). ((v, a), \text{the } (s v) = a)) \ ' \text{set } ?l$ 

```

```

unfolding set-map
by blast
also have ... = ( $\lambda(v, a). fst((v, a), the(s v) = a))$  ‘set ?l
unfolding image-comp[of  $fst \lambda(v, a). ((v, a), the(s v) = a)$ ] comp-apply[of
 $fst \lambda(v, a). ((v, a), the(s v) = a)$ ] 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  $\Psi$ 
shows  $(v, a) \in \text{dom } (\varphi_S \Psi s) \longleftrightarrow v \in \text{set } ((\Psi)_{V+})$ 
 $\wedge s v \neq \text{None}$ 
 $\wedge a \in \mathcal{R}_+ \Psi v$ 
proof –
let ?vs = variables-of  $\Psi$ 
and ?s' =  $\varphi_S \Psi s$ 
show ?thesis
proof (rule iffI)
assume  $(v, a) \in \text{dom } (\varphi_S \Psi s)$ 
then have  $v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V+}) \wedge s v \neq \text{None} \}$ 
and  $a \in \mathcal{R}_+ \Psi v$ 
unfolding state-to-strips-state-dom-is[OF assms(1)]
by force+
moreover have  $v \in \text{set } ?vs$  and  $s v \neq \text{None}$ 
using calculation(1)
by fastforce+
ultimately show
 $v \in \text{set } ((\Psi)_{V+}) \wedge s v \neq \text{None} \wedge a \in \mathcal{R}_+ \Psi v$ 
by force
next
assume  $v \in \text{set } ((\Psi)_{V+}) \wedge s v \neq \text{None} \wedge a \in \mathcal{R}_+ \Psi v$ 
then have  $v \in \text{set } ((\Psi)_{V+})$ 
and  $s v \neq \text{None}$ 
and a-in-range-of-v:  $a \in \mathcal{R}_+ \Psi v$ 
by simp+
then have  $v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V+}) \wedge s v \neq \text{None} \}$ 
by force
thus  $(v, a) \in \text{dom } (\varphi_S \Psi s)$ 
unfolding state-to-strips-state-dom-is[OF assms(1)]
using a-in-range-of-v
by blast
qed
qed

lemma state-to-strips-state-range-is:
assumes is-valid-problem-sas-plus  $\Psi$ 

```

```

and  $(v, a) \in \text{dom}(\varphi_S \Psi s)$ 
shows  $(\varphi_S \Psi s)(v, a) = \text{Some}(\text{the}(s v) = a)$ 
proof —
  let  $?vs = \text{variables-of } \Psi$ 
  let  $?s' = \varphi_S \Psi s$ 
  and  $?defined = \text{filter}(\lambda v. s v \neq \text{None}) ?vs$ 
  let  $?l = \text{concat}[\text{possible-assignments-for } \Psi v. v \leftarrow ?defined]$ 
  have  $v\text{-in-set-}vs: v \in \text{set } ?vs$ 
    and  $s\text{-of-}v\text{-is-not-None}: s v \neq \text{None}$ 
    and  $a\text{-in-range-of-}v: a \in \mathcal{R}_+ \Psi v$ 
    using  $\text{assms}(2)$ 
    unfolding  $\text{state-to-strips-state-dom-is}[OF \text{ assms}(1)]$ 
    by  $\text{fastforce+}$ 
  moreover {
    have  $\forall v \in \text{set}((\Psi)_+) . v \in \text{dom}(\text{sas-plus-problem.range-of } \Psi)$ 
      using  $\text{assms}(1) \text{ is-valid-problem-sas-plus-then}(1)$ 
      unfolding  $\text{is-valid-problem-sas-plus-def}$ 
      by  $\text{fastforce}$ 
    moreover have  $(v, a) \in \text{set } ?l$ 
      unfolding  $\text{state-to-strips-state-dom-is-i}[OF \text{ calculation}(1)]$ 
      using  $s\text{-of-}v\text{-is-not-None } a\text{-in-range-of-}v \text{ } v\text{-in-set-}vs$ 
      by  $\text{fastforce}$ 
    moreover have  $\text{set } ?l \neq \{\}$ 
      using  $\text{calculation}$ 
      by  $\text{fastforce}$ 
    — TODO slow.
    ultimately have  $(\varphi_S \Psi s)(v, a) = \text{Some}(\text{the}(s v) = a)$ 
      using  $\text{map-of-from-function-graph-is-some-if}[of$ 
         $?l(v, a) \lambda(v, a). \text{the}(s v) = a]$ 
      unfolding  $\text{SAS-Plus-STRIPS.state-to-strips-state-def}$ 
         $\text{state-to-strips-state-def Let-def case-prod-beta'}$ 
      by  $\text{fastforce}$ 
  }
  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 $\text{state-to-strips-state-effect-consistent}$:

```

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

```

and $(\varphi_S \Psi s) (v, a') = \text{Some True}$
shows $(v, a) = (v, a')$
proof –
have $\text{the}(s v) = a$ **and** $\text{the}(s v) = a'$
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 $(\varphi_O \Psi op)$)
 $= (\bigcup (v, a) \in \text{set}(\text{effect-of } op). \{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$
proof –
let ?D = range-of Ψ
and ?effect = effect-of op
let ?delete = $[(v, a'). (v, a) \leftarrow ?effect, a' \leftarrow \text{filter}((\neq) a) (\text{the}(\?D v))]$
 $\{$
fix v a
assume $(v, a) \in \text{set}(\?effect)$
then have $(\mathcal{R}_+ \Psi v) = \text{set}(\text{the}(\?D v))$
using assms
using is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op
by fastforce
hence $\text{set}(\text{filter}((\neq) a) (\text{the}(\?D v))) = \{ a' \in \mathcal{R}_+ \Psi v. a' \neq a \}$
unfolding set-filter
by blast
} note nb = this
 $\{$
— TODO slow.
have $\text{set}(\?delete) = \bigcup (\text{set}^{\cdot}(\lambda(v, a). \text{map}(\text{Pair } v) (\text{filter}((\neq) a) (\text{the}(\?D v)))))$
 $\cdot (\text{set}(\?effect))$
using set-concat
by simp
also have ... $= \bigcup ((\lambda(v, a). \text{Pair } v) \cdot \text{set}(\text{filter}((\neq) a) (\text{the}(\?D v))))$
 $\cdot (\text{set}(\?effect))$
unfolding image-comp[of set] set-map
by auto
— TODO slow.
also have ... $= (\bigcup (v, a) \in \text{set}(\?effect). \text{Pair } v) \cdot \{ a' \in \mathcal{R}_+ \Psi v. a' \neq a \})$
using nb
by fast
finally have $\text{set}(\?delete) = (\bigcup (v, a) \in \text{set}(\?effect).$
 $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$
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  $\Pi$  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  $\Psi$ 
  shows set  $((\varphi \Psi)_V) = (\bigcup v \in \text{set } ((\Psi)_{V+}). \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$ 
  proof -
    let  $\Pi = \varphi \Psi$ 
    and  $?vs = \text{variables-of } \Psi$ 
    {
      have set (strips-problem.variables-of  $\Pi$ )
        = set [as.  $v \leftarrow ?vs$ , as  $\leftarrow \text{possible-assignments-for } \Psi v$ ]
      unfolding sas-plus-problem-to-strips-problem-def
        SAS-Plus-STRIPS.sas-plus-problem-to-strips-problem-def
        by force
      also have ... =  $(\bigcup (\text{set } (\lambda v. \text{possible-assignments-for } \Psi v) \text{ `set } ?vs))$ 
        using set-concat
        by auto
      also have ... =  $(\bigcup ((\text{set } \circ \text{possible-assignments-for } \Psi) \text{ `set } ?vs))$ 
        using image-comp[of set  $\lambda v. \text{possible-assignments-for } \Psi v$  set ?vs]
        by argo
      finally have set (strips-problem.variables-of  $\Pi$ )
        =  $(\bigcup v \in \text{set } ?vs. \text{set } (\text{possible-assignments-for } \Psi v))$ 
      unfolding o-apply
      by blast
    }
    moreover have  $\forall v \in \text{set } ?vs. v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$ 
      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  $\Psi$ 
  shows  $(v, a) \in \text{set } ((\varphi \Psi)_V) \longleftrightarrow v \in \text{set } ((\Psi)_{V+}) \wedge a \in \mathcal{R}_+ \Psi v$ 
  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  $op' \in \text{set } ((\Psi)_{O+})$ 
  and is-valid-operator-sas-plus  $\Psi op'$ 

```

```

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

```

```

using calculation
by blast
moreover have  $(v, a) \in$ 
 $(\bigcup(v, a') \in \text{set}(\text{effect-of } op')). \{ (v, a'')$ 
 $| a''. a'' \in (\mathcal{R}_+ \Psi v) \wedge a'' \neq a' \}$ 
using sasp-op-to-strips-set-delete-effects-is
nb assms(1, 3) v-a-in-delete-effects-of-op
by force
moreover obtain  $a'$  where  $(v, a') \in \text{set}(\text{effect-of } op')$  and  $a \neq a'$ 
using calculation
by blast
moreover have  $(v, a') \in \text{set}(\text{add-effects-of } op)$ 
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  $(v, a) \in \text{set}(\text{effect-of } op')$  and  $(v, a') \in \text{set}(\text{effect-of } op')$ 
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  $(v, a) \in \text{set}(\text{add-effects-of } op)$ 
 $\rightarrow (v, a) \notin \text{set}(\text{delete-effects-of } op)$ 
and  $(v, a) \in \text{set}(\text{delete-effects-of } op)$ 
 $\rightarrow (v, a) \notin \text{set}(\text{add-effects-of } op)$ 
by blast+
qed

lemma is-valid-problem-sas-plus-then-strips-transformation-too-iii:
assumes is-valid-problem-sas-plus  $\Psi$ 
shows list-all (is-valid-operator-strips ( $\varphi \Psi$ ))
  (strips-problem.operators-of ( $\varphi \Psi$ ))
proof -
  let  $\text{?}\Pi = \varphi \Psi$ 
  let  $\text{?vs} = \text{strips-problem.variables-of } \text{?}\Pi$ 
  {
    fix  $op$ 
    assume  $op \in \text{set}(\text{strips-problem.operators-of } \text{?}\Pi)$ 
    — TODO slow.
    then obtain  $op'$ 
      where  $op\text{-is: } op = \varphi_O \Psi op'$ 
        and  $op'\text{-in-operators: } op' \in \text{set}((\Psi)_O)$ 
      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  $\Psi$  op'
  using sublocale-sas-plus-finite-domain-representation-ii(2)[OF assms]
  by blast
moreover {
  fix v a
  assume  $(v, a) \in \text{set}(\text{strips-operator}.precondition-of op)$ 
  — TODO slow.
  then have  $(v, a) \in \text{set}(\text{sas-plus-operator}.precondition-of op')$ 
    using op-is
    unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def
    by force
  moreover have  $v \in \text{set}((\Psi)_{\mathcal{V}_+})$ 
    using is-valid-op' calculation
    using is-valid-operator-sas-plus-then(1)
    by fastforce
  moreover have  $a \in \mathcal{R}_+ \Psi v$ 
    using is-valid-op' calculation(1)
    using is-valid-operator-sas-plus-then(2)
    by fast
  ultimately have  $(v, a) \in \text{set} ?vs$ 
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
    by force
}
moreover {
  fix v a
  assume  $(v, a) \in \text{set}(\text{strips-operator}.add-effects-of op)$ 
  then have  $(v, a) \in \text{set}(\text{effect-of op}')$ 
    using op-is
    unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def
    by force
  then have  $v \in \text{set}((\Psi)_{\mathcal{V}_+}) \text{ and } a \in \mathcal{R}_+ \Psi v$ 
    using is-valid-operator-sas-plus-then is-valid-op'
    by fastforce+
  hence  $(v, a) \in \text{set} ?vs$ 
  using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF assms(1)]
    by force
}
moreover {
  fix v a'
  assume v-a'-in-delete-effects:  $(v, a') \in \text{set}(\text{strips-operator}.delete-effects-of$ 
op)
  moreover have  $\text{set}(\text{strips-operator}.delete-effects-of op)$ 
  =  $(\bigcup(v, a) \in \text{set}(\text{effect-of op}'))$ 
     $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \}$ 

```

```

using sasp-op-to-strips-set-delete-effects-is[OF is-valid-op']
  op-is
  by simp
— TODO slow.

ultimately obtain a
  where (v, a) ∈ 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  $\Psi$ 
  shows  $\forall x. ((\varphi \Psi)_G) x \neq \text{None}$ 
     $\longrightarrow \text{ListMem } x (\text{strips-problem.variables-of } (\varphi \Psi))$ 
proof -
  let ?vs = variables-of  $\Psi$ 
  and ?D = range-of  $\Psi$ 
  and ?G = goal-of  $\Psi$ 
  let ? $\Pi$  =  $\varphi \Psi$ 
  let ?vs' = strips-problem.variables-of ? $\Pi$ 
  and ?G' = strips-problem.goal-of ? $\Pi$ 
  have nb: ?G' =  $\varphi_S \Psi$  ?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 ∈  $\mathcal{R}_+ \Psi$  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 Π_V ; and finally,
- if $(\Pi_G) x$ is defined, then x is in Π_V .

theorem

is-valid-problem-sas-plus-then-strips-transformation-too:

assumes *is-valid-problem-sas-plus* Ψ

shows *is-valid-problem-strips* ($\varphi \Psi$)

proof –

let $\text{?}\Pi = \varphi \Psi$

have *list-all* (*is-valid-operator-strips* ($\varphi \Psi$))

(*strips-problem.operators-of* ($\varphi \Psi$)))

using *is-valid-problem-sas-plus-then-strips-transformation-too-iii*[OF assms].

moreover have $\forall x. (((\varphi \Psi)_I) x \neq \text{None}) =$

ListMem x (*strips-problem.variables-of* ($\varphi \Psi$)))

using *is-valid-problem-sas-plus-then-strips-transformation-too-iv*[OF assms].

moreover have $\forall x. ((\varphi \Psi)_G) x \neq \text{None} \rightarrow$

ListMem x (*strips-problem.variables-of* ($\varphi \Psi$)))

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

(*all-possible-assignments-for* Ψ))

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

proof –

let $\text{?vs} = \text{sas-plus-problem.variables-of } \Psi$

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

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

have *set* $\text{?l} = \text{set}(\text{concat}(\text{map}(\text{filter } \text{?P})(\text{map}(\text{possible-assignments-for } \Psi) \text{?vs})))$

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 $\dots = \text{set}(\text{concat}(\text{map}(\lambda v. \text{filter } \text{?P}(\text{possible-assignments-for } \Psi v)) \text{?vs}))$

unfolding *map-map comp-apply*

by *blast*

also have $\dots = \text{set}(\text{concat}(\text{map}(\lambda v. \text{map}(\text{Pair } v)(\text{filter } (\text{?P} \circ \text{Pair } v)(\text{the}(\text{range-of } \Psi v)))) \text{?vs}))$

unfolding *possible-assignments-for-def filter-map*

by *blast*

also have $\dots = \text{set}(\text{concat}(\text{map}(\lambda v. \text{map}(\text{Pair } v)(\text{filter } (\lambda a. s(v, a) = \text{Some True})))) \text{?vs}))$

```

(the (range-of  $\Psi$  v)))) ?vs))
unfolding comp-apply
by fast
also have ... =  $\bigcup$  (set ‘(( $\lambda v.$  map (Pair v) (filter ( $\lambda a.$  s (v, a) = Some True)
(the (range-of  $\Psi$  v)))) ‘ set ?vs))
unfolding set-concat set-map..
also have ... = ( $\bigcup$   $v \in$  set ?vs. Pair v ‘ set (filter ( $\lambda a.$  s (v, a) = Some True)
(the (range-of  $\Psi$  v))))
unfolding image-comp[of set] comp-apply set-map..
also have ... = ( $\bigcup$   $v \in$  set ?vs. Pair v
‘ { a  $\in$  set (the (range-of  $\Psi$  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  $\Psi$ 
shows dom ( $\varphi_S^{-1}$   $\Psi$  s)
= ( $\bigcup$   $v \in$  set (( $\Psi$ ) $\nu_+$ ).
{ v | a. a  $\in$  ( $\mathcal{R}_+$   $\Psi$  v)  $\wedge$  s (v, a) = Some True })
proof –
let ?vs = variables-of  $\Psi$ 
and ?s' =  $\varphi_S^{-1}$   $\Psi$  s
and ?P = ( $\lambda(v, a).$  s (v, a) = Some True)
let ?l = filter ?P (all-possible-assignments-for  $\Psi$ )
{
have fst ‘ set ?l = fst ‘ ( $\bigcup$   $v \in$  set ?vs. Pair v
‘ { a  $\in$  ( $\mathcal{R}_+$   $\Psi$  v). s (v, a) = Some True })
unfolding set-filter-all-possible-assignments-true-is[OF assms]
by auto
also have ... = ( $\bigcup$   $v \in$  set ?vs. fst ‘ Pair v
‘ { a  $\in$  ( $\mathcal{R}_+$   $\Psi$  v). s (v, a) = Some True })
by blast
also have ... = ( $\bigcup$   $v \in$  set ?vs. ( $\lambda a.$  fst (Pair v a)) ‘
{ a  $\in$  ( $\mathcal{R}_+$   $\Psi$  v). s (v, a) = Some True })
unfolding image-comp[of fst] comp-apply
by blast
finally have fst ‘ set ?l = ( $\bigcup$   $v \in$  set (( $\Psi$ ) $\nu_+$ ).
{ v | a. a  $\in$  ( $\mathcal{R}_+$   $\Psi$  v)  $\wedge$  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  $\Psi$ 
    and  $v \in \text{set}((\Psi)\nu_+)$ 
    and  $a \in \mathcal{R}_+ \Psi v$ 
    and  $(v, a) \in \text{dom } s'$ 
    and  $\forall (v, a) \in \text{dom } s'. \forall (v, a') \in \text{dom } s'. s'(v, a) = \text{Some True} \wedge s'(v, a') = \text{Some True}$ 
       $\longrightarrow (v, a) = (v, a')$ 
    shows  $(\varphi_S^{-1} \Psi s') v = \text{Some } a \longleftrightarrow \text{the}(s'(v, a))$ 
proof -
  let ?vs = variables-of  $\Psi$ 
  and ?D = range-of  $\Psi$ 
  and ?s =  $\varphi_S^{-1} \Psi s'$ 
  let ?as = all-possible-assignments-for  $\Psi$ 
  let ?l = filter  $(\lambda(v, a). s'(v, a) = \text{Some True})$  ?as
  show ?thesis
  proof (rule iffI)
    assume s-of-v-is-Some-a: ?s v = Some a
    {
      have  $(v, a) \in \text{set } ?l$ 
      using s-of-v-is-Some-a
      unfolding SAS-Plus-STRIPS.strips-state-to-state-def
        strips-state-to-state-def
      using map-of-SomeD
      by fast
      hence  $s'(v, a) = \text{Some True}$ 
      unfolding all-possible-assignments-for-set-is set-filter
      by blast
    }
    thus 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:  $s'(v, a) = \text{Some True}$ 
      using assms(4) domIff
      by force
    — TODO slow.
    moreover {
      fix v v' a a'
      assume  $(v, a) \in \text{set } ?l$  and  $(v', a') \in \text{set } ?l$ 
      then have  $v \neq v' \vee a = a'$ 
      using assms(5)
      by fastforce
    }
    moreover {
      have  $\forall v \in \text{set}((\Psi)\nu_+).$  sas-plus-problem.range-of  $\Psi v \neq \text{None}$ 
      using is-valid-problem-sas-plus-then(1) assms(1)
      range-of-not-empty
    }

```

```

by force

moreover have set ?l = Set.filter ( $\lambda(v, a). s'(v, a) = \text{Some True}$ )
  ( $\bigcup v \in \text{set } ((\Psi)v_+). \{ (v, a) \mid a \in \mathcal{R}_+ \Psi v \}$ )
    using all-possible-assignments-for-set-is calculation
    by force
ultimately have (v, a)  $\in$  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.strip-state-to-state-def
    strips-state-to-state-def
  by blast
qed
qed

```

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

```

lemma strips-state-to-state-inverse-is-i:
assumes is-valid-problem-sas-plus  $\Psi$ 
  and v  $\in$  set (( $\Psi$ )v_+)
  and s v  $\neq$  None
  and a  $\in$   $\mathcal{R}_+ \Psi v$ 
shows ( $\varphi_S \Psi s$ ) (v, a) = Some (the (s v) = a)
proof -
  let ?vs = sas-plus-problem.variables-of  $\Psi$ 
  let ?s' =  $\varphi_S \Psi s$ 
  and ?f =  $\lambda(v, a). \text{the}(s v) = a$ 
  and ?l = concat (map (possible-assignments-for  $\Psi$ ) (filter ( $\lambda v. s v \neq \text{None}$ ) ?vs))
  have (v, a)  $\in$  dom ?s'
    using state-to-strips-state-dom-element-iff[
      OF assms(1)] assms(2, 3, 4)
    by presburger
{
  have v  $\in$  { v | v. v  $\in$  set (( $\Psi$ )v_+)  $\wedge$  s v  $\neq$  None }
    using assms(2, 3)
    by blast
moreover have  $\forall v \in \text{set } ((\Psi)v_+). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$ 
  using is-valid-problem-sas-plus-dom-sas-plus-problem-range-of[OF assms(1)].

moreover have set ?l = ( $\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)v_+) \wedge s v \neq \text{None} \}$ .
  { (v, a) | a. a  $\in$   $\mathcal{R}_+ \Psi v$  })
  unfolding state-to-strips-state-dom-is-i[OF calculation(2)]
  by blast
ultimately have (v, a)  $\in$  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 ∈ set ((Ψ)_{V+})
  and s v = Some a
  and a ∈ R_+ Ψ v
  and a' ∈ R_+ Ψ v
  and (φ_S Ψ s) (v, a) = Some True
  and (φ_S Ψ s) (v, a') = Some True
shows a = a'

```

```

proof -
  have  $s \neq None$ 
  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  $\Psi$ 
  and dom  $s \subseteq set((\Psi)_{V+})$ 
  and  $v \in set((\Psi)_{V+})$ 
  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  $\Psi$ 
  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  $\Psi$ )
  and ?As = Set.filter ?P ( $\bigcup v \in set((\Psi)_{V+}). \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \}$ )
  {
    have  $\forall v \in set((\Psi)_{V+}). range-of \Psi v \neq None$ 
    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  $(v, a) \in set ?as$ 
  and  $(v', a') \in set ?as$ 
  then have  $(v, a) \in ?As$  and  $(v', a') \in ?As$ 
  using nb
  by blast+
  then have  $v \text{-in-set-} vs: v \in set ?vs$  and  $v' \text{-in-set-} vs: v' \in set ?vs$ 
  and  $a \text{-in-range-of-} v: a \in \mathcal{R}_+ \Psi v$ 
  and  $a' \text{-in-range-of-} v: a' \in \mathcal{R}_+ \Psi v'$ 
  and  $s' \text{-of-} v \text{-a-} is: ?s'(v, a) = Some True$  and  $s' \text{-of-} v' \text{-a'-} is: ?s'(v', a') = Some True$ 
}

```

```

    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 $\text{dom } s = \text{set } (\Psi_{V+})$ and $\forall v \in \text{dom } s. \text{the } (s v) \in \mathcal{R}_+ \Psi v$ are needed because the transformation functions only take into account variables and domains declared in the problem description. They also sufficiently characterize a state that was transformed from SAS+ to STRIPS.

```

lemma strips-state-to-state-inverse-is:
  assumes is-valid-problem-sas-plus  $\Psi$ 
    and  $\text{dom } s \subseteq \text{set } ((\Psi)_{\mathcal{V}_+})$ 
    and  $\forall v \in \text{dom } s. \text{the } (s v) \in \mathcal{R}_+ \Psi v$ 
  shows  $s = (\varphi_S^{-1} \Psi (\varphi_S \Psi s))$ 
proof -
  let ?vs = variables-of  $\Psi$ 
    and ?D = range-of  $\Psi$ 
  let ?s' =  $\varphi_S \Psi s$ 
  let ?s'' =  $\varphi_S^{-1} \Psi ?s'$ 
  — NOTE Show the thesis by proving that  $s$  and  $?s'$  are mutual submaps.
  {
    fix v
    assume v-in-dom-s:  $v \in \text{dom } s$ 
    then have v-in-set-vs:  $v \in \text{set } ?vs$ 
      using assms(2)
      by auto
    then obtain a
      where the-s-v-is-a:  $s v = \text{Some } a$ 
        and a-in-dom-v:  $a \in \mathcal{R}_+ \Psi v$ 
        using assms(2, 3) v-in-dom-s
        by force
    moreover have ?s'' v = 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 ∈ dom ?s''
    then obtain a
      where a ∈  $\mathcal{R}_+ \Psi v$ 
        and ?s'(v, a) = Some True
        using strips-state-to-state-dom-is[OF assms(1)]
        by blast
    then have (v, a) ∈ 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  $\Psi$ 
  and dom  $s \subseteq \text{set } ((\Psi)_{\mathcal{V}_+})$ 
  and  $\forall v \in \text{dom } s. \text{the } (s v) \in \mathcal{R}_+ \Psi v$ 
  shows  $s \subseteq_m t \longleftrightarrow (\varphi_S \Psi s) \subseteq_m (\varphi_S \Psi t)$ 

proof -
  let ?vs = variables-of  $\Psi$ 
  and ?D = range-of  $\Psi$ 
  and ?s' =  $\varphi_S \Psi s$ 
  and ?t' =  $\varphi_S \Psi t$ 
  show ?thesis
  proof (rule iffI)
    assume s-map-le-t:  $s \subseteq_m t$ 
    {
      fix v a
      assume (v, a) ∈ dom ?s'
      moreover have v ∈ set (( $\Psi$ ) $_{\mathcal{V}_+}$ ) and s v ≠ None and a ∈  $\mathcal{R}_+ \Psi v$ 
        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  $?s' \subseteq_m ?t'$ 
  unfolding map-le-def
  by fast

next
  assume  $s'\text{-map-le-}t': ?s' \subseteq_m ?t'$ 
  {
    fix  $v$ 
    assume  $v\text{-in-dom-}s: v \in \text{dom } s$ 
    moreover obtain  $a$  where  $\text{the-of-}s\text{-of-}v\text{-is-}a: \text{the } (s \ v) = a$ 
      by blast
    moreover have  $v\text{-in-vs}: v \in \text{set } ((\Psi)_{\mathcal{V}_+})$ 
      and  $s\text{-of-}v\text{-is-}None: s \ v \neq \text{None}$ 
      and  $a\text{-in-range-of-}v: a \in \mathcal{R}_+ \Psi \ v$ 
      using assms(2, 3)  $v\text{-in-dom-}s$  calculation
      by blast+
    moreover have  $(v, a) \in \text{dom } ?s'$ 
      using state-to-strips-state-dom-is[OF assms(1)]
        calculation(3, 4, 5)
      by simp
    moreover have  $?s' (v, a) = ?t' (v, a)$ 
      using  $s'\text{-map-le-}t'$  calculation
      unfolding map-le-def
      by blast
    moreover have  $(v, a) \in \text{dom } ?t'$ 
      using calculation
      unfolding domIff
      by argo
    moreover have  $?s' (v, a) = \text{Some } (\text{the } (s \ v) = a)$ 
      and  $?t' (v, a) = \text{Some } (\text{the } (t \ v) = a)$ 
      using state-to-strips-state-range-is[OF assms(1)] calculation
      by fast+
    moreover have  $s \ v = \text{Some } a$ 
      using calculation(2, 4)
      by force
    moreover have  $?s' (v, a) = \text{Some } \text{True}$ 
      using calculation(9, 11)
      by fastforce
    moreover have  $?t' (v, a) = \text{Some } \text{True}$ 
      using calculation(7, 12)
      by argo
    moreover have  $\text{the } (t \ v) = a$ 
      using calculation(10, 13) try0
      by force
    moreover {
      have  $v \in \text{dom } t$ 
        using state-to-strips-state-dom-element-iff[OF assms(1)]
          calculation(8)
      by auto
    }
  }
}

```

```

hence  $t v = \text{Some } a$ 
      using calculation(14)
      by force
}
ultimately have  $s v = t v$ 
      by argo
}
thus  $s \subseteq_m t$ 
      unfolding map-le-def
      by simp
qed
qed

```

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

```

lemma sas-plus-operator-inverse-is:
assumes is-valid-problem-sas-plus  $\Psi$ 
      and  $op \in \text{set } ((\Psi)_{O+})$ 
      shows  $(\varphi_O^{-1} \Psi (\varphi_O \Psi op)) = op$ 
proof —
  let  $?op = \varphi_O^{-1} \Psi (\varphi_O \Psi op)$ 
  have precondition-of  $?op = \text{precondition-of } op$ 
    unfolding SAS-Plus-STRIPS.strip-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  $?op = \text{effect-of } op$ 
    unfolding SAS-Plus-STRIPS.strip-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 $op \in \text{operators-of } \Psi$. If we don't make this assumption, then multiple STRIPS operators of the form $(\text{precondition-of} = [], \text{add-effects-of} = [], \text{delete-effects-of} = [(v, a), ...])$ correspond to one SAS+ operator (since the delete effects are being discarded in the transformation function).

```

lemma strips-operator-inverse-is:
assumes is-valid-problem-sas-plus  $\Psi$ 

```

```

and  $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
shows  $(\varphi_{\mathcal{O}} \Psi (\varphi_{\mathcal{O}}^{-1} \Psi op')) = op'$ 
proof –
  let  $\Pi = \varphi \Psi$ 
  obtain  $op$  where  $op \in \text{set } ((\Psi)_{\mathcal{O}+})$  and  $op' = \varphi_{\mathcal{O}} \Psi op$ 
    using assms
    by auto
  moreover have  $\varphi_{\mathcal{O}}^{-1} \Psi op' = op$ 
    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  $\Psi$ 
  and  $\text{set } ops' \subseteq \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
  and STRIPS-Semantics.are-all-operators-applicable  $(\varphi_S \Psi s) ops'$ 
  and  $op \in \text{set } [\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
  shows map-of  $(\text{precondition-of } op) \subseteq_m (\varphi_S^{-1} \Psi (\varphi_S \Psi s))$ 
proof –
  let  $\Pi = \varphi \Psi$ 
  and  $?s' = \varphi_S \Psi s$ 
  let  $?s = \varphi_S^{-1} \Psi ?s'$ 
  and  $?D = \text{range-of } \Psi$ 
  and  $?ops = [\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
  and  $?pre = \text{precondition-of } op$ 
  have  $nb_1: \forall (v, a) \in \text{dom } ?s'.$ 
     $\forall (v, a') \in \text{dom } ?s'.$ 
     $?s'(v, a) = \text{Some True} \wedge ?s'(v, a') = \text{Some True}$ 
     $\rightarrow (v, a) = (v, a')$ 
  using state-to-strips-state-effect-consistent[OF assms(1)]
  by blast
  {
    fix  $op'$ 
    assume  $op' \in \text{set } ops'$ 
    moreover have  $op' \in \text{set } ((\Pi)_{\mathcal{O}})$ 
      using assms(2) calculation
      by blast
    ultimately have  $\exists op \in \text{set } ((\Psi)_{\mathcal{O}+}). op' = (\varphi_{\mathcal{O}} \Psi op)$ 
      by auto
  }
  note  $nb_2 = this$ 
  {
    fix  $op$ 
    assume  $op \in \text{set } ?ops$ 
    then obtain  $op'$  where  $op' \in \text{set } ops'$  and  $op = \varphi_{\mathcal{O}}^{-1} \Psi op'$ 
      using assms(4)
      by auto
  }

```

```

moreover obtain  $op''$  where  $op'' \in \text{set } ((\Psi)_{\mathcal{O}+})$  and  $op' = \varphi_{\mathcal{O}} \Psi op''$ 
  using  $nb_2$  calculation(1)
  by blast
moreover have  $op = op''$ 
  using sas-plus-operator-inverse-is[ $\text{OF assms}(1)$  calculation(3)] calculation(2,
4)
  by blast
ultimately have  $op \in \text{set } ((\Psi)_{\mathcal{O}+})$ 
  by blast
} note  $nb_3 = \text{this}$ 
{
fix  $op v a$ 
assume  $op \in \text{set } ?ops$ 
  and  $v\text{-}a\text{-in-precondition-of-}op': (v, a) \in \text{set } (\text{precondition-of } op)$ 
moreover obtain  $op'$  where  $op' \in \text{set } ops'$  and  $op = \varphi_{\mathcal{O}}^{-1} \Psi op'$ 
  using calculation(1)
  by auto
moreover have strips-operator.precondition-of  $op' = \text{precondition-of } op$ 
  using calculation(4)
  unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
    strips-op-to-sasp-def
  by simp
ultimately have  $\exists op' \in \text{set } ops'. op = (\varphi_{\mathcal{O}}^{-1} \Psi op')$ 
   $\wedge (v, a) \in \text{set } (\text{strips-operator.precondition-of } op')$ 
  by metis
} note  $nb_4 = \text{this}$ 
{
fix  $op' v a$ 
assume  $op' \in \text{set } ops'$ 
  and  $v\text{-}a\text{-in-precondition-of-}op': (v, a) \in \text{set } (\text{strips-operator.precondition-of } op')$ 
moreover have  $s'\text{-of-}v\text{-}a\text{-is-Some-True}: ?s'(v, a) = \text{Some True}$ 
  using assms(3) calculation(1, 2)
  unfolding are-all-operators-applicable-set
  by blast
moreover {
obtain  $op$  where  $op \in \text{set } ((\Psi)_{\mathcal{O}+})$  and  $op' = \varphi_{\mathcal{O}} \Psi op$ 
  using  $nb_2$  calculation(1)
  by blast
moreover have strips-operator.precondition-of  $op' = \text{precondition-of } op$ 
  using calculation(2)
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
    sasp-op-to-strips-def
  by simp
moreover have  $(v, a) \in \text{set } (\text{precondition-of } op)$ 
  using  $v\text{-}a\text{-in-precondition-of-}op'$  calculation(3)
  by argo
moreover have is-valid-operator-sas-plus  $\Psi op$ 
  using is-valid-problem-sas-plus-then(2) assms(1) calculation(1)

```

```

unfolding is-valid-operator-sas-plus-def
by auto
moreover have  $v \in \text{set } ((\Psi)_{V+})$  and  $a \in \mathcal{R}_+ \Psi v$ 
  using is-valid-operator-sas-plus-then(1,2) calculation(4, 5)
  unfolding is-valid-operator-sas-plus-def
  by fastforce+
moreover have  $v \in \text{dom } ?s$ 
  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  $(v, a) \in \text{dom } ?s'$ 
  using s'-of-v-a-is-Some-True domIff
  by blast
ultimately have  $?s v = \text{Some } a$ 
  using strips-state-to-state-range-is[OF assms(1) - - - nb1]
    s'-of-v-a-is-Some-True
  by simp
}
hence  $?s v = \text{Some } a$ .
} note nb5 = this
{
fix v
assume  $v \in \text{dom } (\text{map-of } ?pre)$ 
then obtain a where map-of ?pre v = Some a
  by fast
moreover have  $(v, a) \in \text{set } ?pre$ 
  using map-of-SomeD calculation
  by fast
moreover {
  have op  $\in \text{set } ((\Psi)_{O+})$ 
    using assms(4) nb3
    by blast
  then have is-valid-operator-sas-plus  $\Psi$  op
    using is-valid-problem-sas-plus-then(2) assms(1)
    unfolding is-valid-operator-sas-plus-def
    by auto
  hence  $\forall (v, a) \in \text{set } ?pre. \forall (v', a') \in \text{set } ?pre. v \neq v' \vee a = a'$ 
    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'  $\in \text{set } ops'$ 
  and  $(v, a) \in \text{set } (\text{strips-operator.precondition-of } op')$ 
  using nb4[OF assms(4) calculation(2)]
  by blast
moreover have ?s v = Some a

```

```

using nb5 calculation(5, 6)
by fast
ultimately have map-of ?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  $\Psi$ 
and set  $ops' \subseteq set ((\varphi \Psi)_{\mathcal{O}})$ 
and  $op \in set [\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
shows  $op \in set ((\Psi)_{\mathcal{O}+}) \wedge (\exists op' \in set ops'. op' = \varphi_{\mathcal{O}} \Psi op)$ 
proof -
let ? $\Pi$  =  $\varphi \Psi$ 
obtain  $op'$  where  $op' \in set ops'$  and  $op = \varphi_{\mathcal{O}}^{-1} \Psi op'$ 
using assms(3)
by auto
moreover have  $op' \in set ((\varphi \Psi)_{\mathcal{O}})$ 
using assms(2) calculation(1)
by blast
moreover obtain  $op''$  where  $op'' \in set ((\Psi)_{\mathcal{O}+})$  and  $op' = \varphi_{\mathcal{O}} \Psi op''$ 
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  $\Psi :: ('variable, 'domain) sas-plus-problem$ 
fixes  $s :: ('variable, 'domain) state$ 
assumes is-valid-problem-sas-plus  $\Psi$ 
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-operator-effects-consistent  $[\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
proof -
let ? $s' = \varphi_s \Psi s$ 
let ? $s = \varphi_s^{-1} \Psi s'$ 
and ? $ops = [\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
and ? $\Pi = \varphi \Psi$ 
have nb:  $\forall (v, a) \in dom s'. v \in dom s'$ .

```

```

 $\forall (v, a') \in \text{dom } ?s'.$ 
 $?s'(v, a) = \text{Some True} \wedge ?s'(v, a') = \text{Some True}$ 
 $\longrightarrow (v, a) = (v, a')$ 
using state-to-strips-state-effect-consistent[OF assms(1)]
by blast
{
  fix  $op_1' op_2'$ 
  assume  $op_1' \in \text{set } ops' \text{ and } op_2' \in \text{set } ops'$ 
  hence STRIPS-Semantics.are-operator-effects-consistent  $op_1' op_2'$ 
    using assms(3)
    unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
      by blast
} note  $nb_1 = this$ 
{
  fix  $op_1 op_1' op_2 op_2'$ 
  assume  $op_1 \text{-in-ops: } op_1 \in \text{set } ?ops$ 
    and  $op_1' \text{-in-ops': } op_1' \in \text{set } ops'$ 
    and  $op_1' \text{-is: } op_1' = \varphi_O \Psi op_1$ 
    and  $is\text{-valid-op}_1: is\text{-valid-operator-sas-plus } \Psi op_1$ 
    and  $op_2 \text{-in-ops: } op_2 \in \text{set } ?ops$ 
    and  $op_2' \text{-in-ops': } op_2' \in \text{set } ops'$ 
    and  $op_2' \text{-is: } op_2' = \varphi_O \Psi op_2$ 
    and  $is\text{-valid-op}_2: is\text{-valid-operator-sas-plus } \Psi op_2$ 
  have  $\forall (v, a) \in \text{set } (\text{add-effects-of } op_1'). \forall (v', a') \in \text{set } (\text{add-effects-of } op_2').$ 
     $v \neq v' \vee a = a'$ 
  proof (rule ccontr)
    assume  $\neg(\forall (v, a) \in \text{set } (\text{add-effects-of } op_1')). \forall (v', a') \in \text{set } (\text{add-effects-of } op_2').$ 
     $v \neq v' \vee a = a'$ 
    then obtain  $v v' a a' \text{ where } (v, a) \in \text{set } (\text{add-effects-of } op_1')$ 
      and  $(v', a') \in \text{set } (\text{add-effects-of } op_2')$ 
      and  $v = v'$ 
      and  $a \neq a'$ 
      by blast
    — TODO slow.
    moreover have  $(v, a) \in \text{set } (\text{effect-of } op_1)$ 
      using  $op_1' \text{-is } op_2' \text{-is calculation}(1, 2)$ 
      unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
        sasp-op-to-strips-def
      by force
    moreover {
      have  $(v', a') \in \text{set } (\text{effect-of } op_2)$ 
      using  $op_2' \text{-is calculation}(2)$ 
      unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
        sasp-op-to-strips-def
      by force
    hence  $a' \in \mathcal{R}_+ \Psi v$ 
      using  $is\text{-valid-operator-sas-plus-then } is\text{-valid-op}_2 \text{ calculation}(3)$ 
      by fastforce
}

```

```

}

moreover have  $(v, a') \in \text{set}(\text{delete-effects-of } op_1')$ 
  using sasp-op-to-strips-set-delete-effects-is
     $op_1'$ -is is-valid-op $_1$  calculation(3, 4, 5, 6)
  by blast
moreover have  $\neg \text{STRIPS-Semantics.are-operator-effects-consistent } op_1'$ 
 $op_2'$ 
  unfolding STRIPS-Semantics.are-operator-effects-consistent-def list-ex-iff

  using calculation(2, 3, 7)
  by meson
ultimately show False
  using assms(3) op_1'-in-ops' op_2'-in-ops'
unfolding STRIPS-Semantics.are-all-operator-effects-consistent-def list-all-iff
  by blast
qed
} note nb $_3 = this$ 
{
fix  $op_1$   $op_2$ 
assume  $op_1\text{-in-ops}: op_1 \in \text{set} \ ?ops$  and  $op_2\text{-in-ops}: op_2 \in \text{set} \ ?ops$ 
moreover have  $op_1\text{-in-operators-of-}\Psi: op_1 \in \text{set} ((\Psi)_{\mathcal{O}_+})$ 
  and  $op_2\text{-in-operators-of-}\Psi: op_2 \in \text{set} ((\Psi)_{\mathcal{O}_+})$ 
  using to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure[OF assms(1, 2)] calculation
  by blast+
moreover have 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  $op_1' op_2'$ 
  where  $op_1\text{-in-ops': } op_1' \in \text{set} \ ops'$ 
    and  $op_1\text{-is: } op_1' = \varphi_O \Psi op_1$ 
    and  $op_2\text{-in-ops': } op_2' \in \text{set} \ ops'$ 
    and  $op_2\text{-is: } op_2' = \varphi_O \Psi op_2$ 
  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  $\forall (v, a) \in \text{set}(\text{add-effects-of } op_1'). \forall (v', a') \in \text{set}(\text{add-effects-of } op_2')$ .
   $v \neq v' \vee a = a'$ 
  using nb $_3$ 
  by auto
hence are-operator-effects-consistent op1 op2
  using op1-is op2-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  $\Psi$ 
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  $?vs' = strips\text{-}problem.variables-of ?Pi$ 
and  $?ops' = strips\text{-}problem.operators-of ?Pi$ 
and  $?vs = variables-of \Psi$ 
and  $?D = range-of \Psi$ 
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  $(\lambda op. map\text{-}of (precondition-of op) \subseteq_m ?s) ?ops$ 
using sas-plus-equivalent-to-strips-i-a-I[OF assms(1) assms(2)] calculation
unfolding list-all-iff
by blast
moreover have list-all  $(\lambda op. list\text{-}all (are-operator-effects-consistent op) ?ops)$ 
?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  $\Psi$ 
and  $\text{dom } s \subseteq \text{set } ((\Psi)_{\mathcal{V}+})$ 
and  $\forall v \in \text{dom } s. \text{the } (s v) \in \mathcal{R}_+ \Psi v$ 
and  $\text{set } ops' \subseteq \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
and are-all-operators-applicable-in  $s [\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops'] \wedge$ 
  are-all-operator-effects-consistent  $[\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
shows STRIPS-Semantics.are-all-operators-applicable  $(\varphi_S \Psi s) ops'$ 
proof -
let ?vs = variables-of  $\Psi$ 
and ?D = range-of  $\Psi$ 
and ?Pi =  $\varphi \Psi$ 
and ?ops =  $[\varphi_{\mathcal{O}}^{-1} \Psi op'. op' \leftarrow ops']$ 
and ?s' =  $\varphi_S \Psi s$ 
— TODO refactor.
{
fix op'
assume  $op' \in \text{set } ops'$ 
moreover obtain op where  $op \in \text{set } ?ops \text{ and } op = \varphi_{\mathcal{O}}^{-1} \Psi op'$ 
  using calculation
  by force
moreover obtain op'' where  $op'' \in \text{set } ((\Psi)_{\mathcal{O}+}) \text{ and } op' = \varphi_{\mathcal{O}} \Psi op''$ 
  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  $\exists op \in \text{set } ?ops. op \in \text{set } ?ops \wedge op = (\varphi_{\mathcal{O}}^{-1} \Psi op')$ 
   $\wedge$  is-valid-operator-sas-plus  $\Psi op$ 
  by blast
} note nb1 = this

```

```

have nb2:  $\forall (v, a) \in \text{dom } ?s'.$ 
 $\forall (v, a') \in \text{dom } ?s'.$ 
 $?s'(v, a) = \text{Some True} \wedge ?s'(v, a') = \text{Some True}$ 
 $\longrightarrow (v, a) = (v, a')$ 
using state-to-strips-state-effect-consistent[OF assms(1), of - - s]
by blast
{
fix op
assume op  $\in$  set ?ops
hence map-of (precondition-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 nb3 = 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-valid-operator-sas-plus  $\Psi$  op
using nb1
by force
moreover have preconditions-are-consistent:
 $\forall (v, a) \in \text{set } (\text{precondition-of op}). \forall (v', a') \in \text{set } (\text{precondition-of op}). v \neq v' \vee a = a'$ 
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-operator.precondition-of op')
moreover have v-a-in-precondition-of-op: (v, a)  $\in$  set (precondition-of op)
using op-is calculation
unfolding SAS-Plus-STRIPS.strips-op-to-sasp-def
strips-op-to-sasp-def
by auto
moreover have map-of (precondition-of op) v = Some a
using map-of-constant-assignments-defined-if[OF
preconditions-are-consistent calculation(2)]
by blast
moreover have s-of-v-is: s v = Some a
using nb3[OF op-in-ops] calculation(3)
unfolding map-le-def
by force
moreover have v  $\in$  set  $((\Psi)v_+)$  and a  $\in \mathcal{R}_+ \Psi v$ 
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 ( $\varphi_S^{-1} \Psi$  ?s') v = Some a
    using strips-state-to-state-inverse-is[OF assms(1, 2, 3)] s-of-v-is
    by argo
    — TODO slow.
  ultimately have ?s' (v, a) = Some True
    using strips-state-to-state-range-is[OF assms(1)] nb2
    by auto
  }
  ultimately have  $\forall (v, a) \in \text{set}(\text{strips-operator.precondition-of } op'). ?s' (v, a)$ 
= Some True
  by fast
}
thus ?thesis
  unfolding are-all-operators-applicable-def is-operator-applicable-in-def
  STRIPS-Representation.is-operator-applicable-in-def list-all-iff
  by simp
qed

```

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

assumes is-valid-problem-sas-plus Ψ

and $\text{dom } s \subseteq \text{set}((\Psi)_{\mathcal{V}_+})$

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

and $\text{set } ops' \subseteq \text{set}((\varphi \Psi)_{\mathcal{O}})$

and are-all-operators-applicable-in $s [\varphi_O^{-1} \Psi op'. op' \leftarrow ops'] \wedge$
are-all-operator-effects-consistent $[\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$

shows STRIPS-Semantics.are-all-operator-effects-consistent ops'

proof –

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

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

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

and $?Pi = \varphi \Psi$

— TODO refactor.

{

fix op'

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

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

using calculation

by force

moreover obtain op'' where $op'' \in \text{set}((\Psi)_{\mathcal{O}_+})$ 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  $\exists op \in \text{set } ?ops. op \in \text{set } ?ops \wedge op' = (\varphi_O \Psi op)$ 
   $\wedge \text{is-valid-operator-sas-plus } \Psi op$ 
  by blast
} note  $nb_1 = \text{this}$ 
{
fix  $op_1' op_2'$ 
assume  $op_1' \in \text{set } ops'$ 
and  $op_2' \in \text{set } ops'$ 
and  $\exists (v, a) \in \text{set } (\text{add-effects-of } op_1'). \exists (v', a') \in \text{set } (\text{delete-effects-of } op_2').$ 
   $(v, a) = (v', a')$ 
moreover obtain  $op_1 op_2$ 
  where  $op_1 \in \text{set } ?ops$ 
    and  $op_1' = \varphi_O \Psi op_1$ 
    and is-valid-operator-sas-plus  $\Psi op_1$ 
    and  $op_2 \in \text{set } ?ops$ 
    and  $op_2' = \varphi_O \Psi op_2$ 
    and is-valid-op2: is-valid-operator-sas-plus  $\Psi op_2$ 
  using  $nb_1$  calculation(1, 2)
  by meson
moreover obtain  $v v' a a'$ 
  where  $(v, a) \in \text{set } (\text{add-effects-of } op_1')$ 
    and  $(v', a') \in \text{set } (\text{delete-effects-of } op_2')$ 
    and  $(v, a) = (v', a')$ 
  using calculation
  by blast
moreover have  $(v, a) \in \text{set } (\text{effect-of } op_1)$ 
  using calculation(5, 10)
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
    sasp-op-to-strips-def
  by fastforce
moreover have  $v = v'$  and  $a = a'$ 
  using calculation(12)
  by simp+
— The next proof block shows that  $(v', a')$  is constructed from an effect  $(v'', a'')$  s.t.  $a' \neq a''$ .
moreover {
  have  $(v', a') \in (\bigcup (v'', a'') \in \text{set } (\text{effect-of } op_2).$ 
     $\{ (v'', a'') \mid a''' . a''' \in (\mathcal{R}_+ \Psi v'') \wedge a''' \neq a'' \})$ 
  using sasp-op-to-strips-set-delete-effects-is
    calculation(8, 11) is-valid-op2
  by blast

```

then obtain $v'' a''$ where $(v'', a'') \in \text{set}(\text{effect-of } op_2)$
 and $(v', a') \in \{ (v'', a'') \mid a''' \cdot a''' \in (\mathcal{R}_+ \Psi v'') \wedge a''' \neq a'' \}$
 by blast
 moreover have $(v', a'') \in \text{set}(\text{effect-of } op_2)$
 using calculation
 by blast
 moreover have $a' \in \mathcal{R}_+ \Psi v''$ and $a' \neq a''$
 using calculation(1, 2)
 by fast+
 ultimately have $\exists a''. (v', a'') \in \text{set}(\text{effect-of } op_2) \wedge a' \in (\mathcal{R}_+ \Psi v')$
 $\wedge a' \neq a''$
 by blast
 }
 moreover obtain a'' where $(v', a'') \in \text{set}(\text{effect-of } op_2)$
 and $a' \in \mathcal{R}_+ \Psi v'$
 and $a' \neq a''$
 using calculation(16)
 by blast
 moreover have $\exists (v, a) \in \text{set}(\text{effect-of } op_1). (\exists (v', a') \in \text{set}(\text{effect-of } op_2).$
 $v = v' \wedge a \neq a')$
 using calculation(13, 14, 15, 17, 19)
 by blast
 moreover have $\neg \text{are-operator-effects-consistent } op_1 \text{ } op_2$
 unfolding $\text{are-operator-effects-consistent-def list-all-iff}$
 using calculation(20)
 by fastforce
 ultimately have $\neg \text{are-all-operator-effects-consistent } ?ops$
 unfolding $\text{are-all-operator-effects-consistent-def list-all-iff}$
 by meson
} note $nb_2 = this$
{
fix $op_1' \text{ } op_2'$
assume $op_1'\text{-in-ops}: op_1' \in \text{set } ops'$ and $op_2'\text{-in-ops}: op_2' \in \text{set } ops'$
have STRIPS-Semantics.are-operator-effects-consistent $op_1' \text{ } op_2'$
proof (rule ccontr)
assume $\neg \text{STRIPS-Semantics.are-operator-effects-consistent } op_1' \text{ } op_2'$
then consider (A) $\exists (v, a) \in \text{set}(\text{add-effects-of } op_1').$
 $\exists (v', a') \in \text{set}(\text{delete-effects-of } op_2'). (v, a) = (v', a')$
| (B) $\exists (v, a) \in \text{set}(\text{add-effects-of } op_2').$
 $\exists (v', a') \in \text{set}(\text{delete-effects-of } op_1'). (v, a) = (v', a')$
unfolding STRIPS-Semantics.are-operator-effects-consistent-def list-ex-iff
by fastforce
thus False
using $nb_2[OP \text{ } op_1'\text{-in-ops } op_2'\text{-in-ops}] \text{ } nb_2[OP \text{ } op_2'\text{-in-ops } op_1'\text{-in-ops}]$
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  $\Psi$ 
and dom  $s \subseteq$  set  $((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } s. \text{the}(s v) \in \mathcal{R}_+ \Psi v$ 
and set  $\text{ops}' \subseteq$  set  $((\varphi \Psi)_O)$ 
and are-all-operators-applicable-in  $s [\varphi_O^{-1} \Psi \text{op}'. \text{op}' \leftarrow \text{ops}'] \wedge$ 
  are-all-operator-effects-consistent  $[\varphi_O^{-1} \Psi \text{op}'. \text{op}' \leftarrow \text{ops}']$ 
shows STRIPS-Semantics.are-all-operators-applicable  $(\varphi_S \Psi s) \text{ops}'$ 
   $\wedge$  STRIPS-Semantics.are-all-operator-effects-consistent  $\text{ops}'$ 
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  $s \subseteq V$ 
and  $\forall op \in \text{set } \text{ops}. \forall (v, a) \in \text{set } (\text{effect-of } op). v \in V$ 
shows dom  $(\text{execute-parallel-operator-sas-plus } s \text{ops}) \subseteq V$ 
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 ? $s' = s ++ \text{map-of } (\text{effect-of } op)$ 
— TODO Wrap IH instantiation in block.
{
have  $\forall (v, a) \in \text{set } (\text{effect-of } op). v \in V$ 
using Cons.prems(2)
by fastforce
moreover have fst ` set  $(\text{effect-of } op) \subseteq V$ 
using calculation
by fastforce
ultimately have dom ? $s' \subseteq V$ 
unfolding dom-map-add dom-map-of-conv-image-fst
using Cons.prems(1)
by blast
}
moreover have  $\forall op \in \text{set } \text{ops}. \forall (v, a) \in \text{set } (\text{effect-of } op). v \in V$ 
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.prem(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.prem(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.prem(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.prem(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.prem(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.prem(2)]
  by meson
}
thus ?case
  unfolding execute-parallel-operator-sas-plus-cons
  by blast
qed
qed

lemma transform-sas-plus-problem-to-strips-problem-operators-valid:
assumes is-valid-problem-sas-plus Ψ
  and op' ∈ set ((φ Ψ)ο)
  obtains op
  where op ∈ set ((Ψ)ο+)
  and op' = (φο Ψ op) is-valid-operator-sas-plus Ψ op
proof -
{
  obtain op where op ∈ set ((Ψ)ο+) and op' = φο Ψ op
  using assms
  by auto
}

```

```

moreover have is-valid-operator-sas-plus  $\Psi$   $op$ 
  using is-valid-problem-sas-plus-then(2)  $assms(1)$   $calculation(1)$ 
  by auto
ultimately have  $\exists op \in set((\Psi)_{\mathcal{O}+}). op' = (\varphi_{\mathcal{O}} \Psi op)$ 
   $\wedge$  is-valid-operator-sas-plus  $\Psi$   $op$ 
  by blast
}
thus ?thesis
  using that
  by blast
qed

lemma sas-plus-equivalent-to-strips-i-a-XI:
  assumes is-valid-problem-sas-plus  $\Psi$ 
  and  $op' \in set((\varphi \Psi)_{\mathcal{O}})$ 
  shows  $(\varphi_S \Psi s) ++ map-of(effect-to-assignments op')$ 
     $= \varphi_S \Psi (s ++ map-of(effect-of(\varphi_O^{-1} \Psi op')))$ 
proof -
  let ? $\Pi = \varphi \Psi$ 
  let ? $vs = variables-of \Psi$ 
  and ? $ops = operators-of \Psi$ 
  and ? $ops' = strips-problemoperators-of \Pi$ 
  let ? $s' = \varphi_S \Psi s$ 
  let ? $t = s' ++ map-of(effect-to-assignments op')$ 
  and ? $t' = \varphi_S \Psi (s ++ map-of(effect-of(\varphi_O^{-1} \Psi op')))$ 
  obtain  $op$  where  $op'-is: op' = (\varphi_O \Psi op)$ 
    and  $op-in-ops: op \in set((\Psi)_{\mathcal{O}+})$ 
    and is-valid-operator-op: is-valid-operator-sas-plus  $\Psi$   $op$ 
    using transfom-sas-plus-problem-to-strips-problem-operators-valid[OF assms]
    by auto
  have  $nb_1: (\varphi_O^{-1} \Psi op') = op$ 
    using sas-plus-operator-inverse-is[OF assms(1)]  $op'-is$   $op-in-ops$ 
    by blast
  — TODO refactor.
{
  have  $dom(map-of(effect-to-assignments op'))$ 
     $= set(strips-operator.add-effects-of op') \cup set(strips-operator.delete-effects-of op')$ 
    unfolding dom-map-of-conv-image-fst
    by force
  — TODO slow.
  also have ...  $= set(effect-of op) \cup set(strips-operator.delete-effects-of op')$ 
    using  $op'-is$ 
    unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def
    by auto
  — TODO slow.
  finally have  $dom(map-of(effect-to-assignments op')) = set(effect-of op)$ 

```

```

 $\cup (\bigcup(v, a) \in \text{set}(\text{effect-of } op). \{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
using sasp-op-to-strips-set-delete-effects-is[OF
    is-valid-operator-op]  $op'$ -is
by argo
} note  $nb_2 = this$ 
have  $nb_3: \text{dom } ?t = \text{dom } ?s' \cup \text{set}(\text{effect-of } op)$ 
 $\cup (\bigcup(v, a) \in \text{set}(\text{effect-of } op). \{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
unfolding  $nb_2$   $\text{dom-map-add}$ 
by blast
— TODO refactor.
have  $nb_4: \text{dom } (s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi op')))$ 
 $= \text{dom } s \cup \text{fst } ' \text{set}(\text{effect-of } op)$ 
unfolding  $\text{dom-map-add dom-map-of-conv-image-fst } nb_1$ 
by fast
{
let  $?u = s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi op'))$ 
have  $\text{dom } ?t' = (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_v)_+ \wedge ?u v \neq \text{None} \}.$ 
     $\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$ 
using state-to-strips-state-dom-is[OF assms(1)]
by presburger
} note  $nb_5 = this$ 
— TODO refactor.
have  $nb_6: \text{set } (\text{add-effects-of } op') = \text{set } (\text{effect-of } op)$ 
using  $op'$ -is
unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
    sasp-op-to-strips-def
by auto
— TODO refactor.
have  $nb_7: \text{set } (\text{delete-effects-of } op') = (\bigcup(v, a) \in \text{set}(\text{effect-of } op).$ 
     $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
using sasp-op-to-strips-set-delete-effects-is[OF
    is-valid-operator-op]  $op'$ -is
by argo
— TODO refactor.
{
let  $?Add = \text{set } (\text{effect-of } op)$ 
let  $?Delete = (\bigcup(v, a) \in \text{set}(\text{effect-of } op).$ 
     $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
have  $\text{dom-add}: \text{dom } (\text{map-of } (\text{map } (\lambda v. (v, \text{True})) (\text{add-effects-of } op'))) = ?Add$ 

unfolding  $\text{dom-map-of-conv-image-fst set-map image-comp comp-apply}$ 
using  $nb_6$ 
by simp
have  $\text{dom-delete}: \text{dom } (\text{map-of } (\text{map } (\lambda v. (v, \text{False})) (\text{delete-effects-of } op'))) =$ 
? $Delete$ 
unfolding  $\text{dom-map-of-conv-image-fst set-map image-comp comp-apply}$ 
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 disjoint (dom (map-of (map (λv. (v, True)) (add-effects-of op'))))
  (dom (map-of (map (λv. (v, False)) (delete-effects-of op'))))
  unfolding disjoint-def Int-def
  using nb7
  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 disjoint (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 nb8 = this
— TODO refactor.
{
  let ?Add = set (effect-of op)
  let ?Delete = (U(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 disjoint-def
    using calculation(1)
    by blast
  moreover have map-of (effect-to-assignments op') (v, a)
  = map-of (map (λv. (v, False)) (delete-effects-of op')) (v, a)
  unfolding effect-to-assignments-simp map-of-append
  using map-add-dom-app-simps(3)[OF calculation(3)]
  by presburger
— TODO slow.
ultimately have map-of (effect-to-assignments op') (v, a) = Some False
  using map-of-defined-if-constructed-from-list-of-constant-assignments[
    of map (λv. (v, False)) (delete-effects-of op') False delete-effects-of op']
    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 ∀(v, a) ∈ set (effect-of op). ∀(v', a') ∈ set (effect-of op). v ≠
v' ∨ a = a'
    using is-valid-operator-sas-plus-then is-valid-operator-op
    unfolding is-valid-operator-sas-plus-def
    by fast
  ultimately have map-of (effect-of op) v = Some a
    using map-of-constant-assignments-defined-if[of effect-of op]
    by presburger
} note nb10 = this
{

```

```

fix v a
assume v-a-in-effect-of-op: (v, a) ∈ set (effect-of op)
  and (s ++ map-of (effect-of (φO-1 Ψ op'))) v ≠ None
moreover have v ∈ set ?vs
  using is-valid-operator-op is-valid-operator-sas-plus-then(3) calculation(1)
    by fastforce
moreover {
  have is-valid-problem-strips ?Π
    using is-valid-problem-sas-plus-then-strips-transformation-too
      assms(1)
    by blast
  thm calculation(1) nb6 assms(2)
  moreover have set (add-effects-of op') ⊆ set ((?Π)V)
    using assms(2) is-valid-problem-strips-operator-variable-sets(2)
      calculation
    by blast
  moreover have (v, a) ∈ set ((?Π)V)
    using v-a-in-effect-of-op nb6 calculation(2)
    by blast
  ultimately have a ∈ R+ Ψ v
    using sas-plus-problem-to-strips-problem-variable-set-element-iff[OF
      assms(1)]
    by fast
}
— TODO slow.
ultimately have (v, a) ∈ dom (φS Ψ (s ++ map-of (effect-of (φO-1 Ψ op'))))

  using state-to-strips-state-dom-is[OF assms(1), of
    s ++ map-of (effect-of (φO-1 Ψ op'))]
  by simp
} note nb11 = this
{
fix v a
assume (v, a) ∈ set (effect-of op)
moreover have v ∈ dom (map-of (effect-of op))
  unfolding dom-map-of-conv-image-fst
  using calculation
  by force
moreover have (s ++ map-of (effect-of (φO-1 Ψ op'))) v = Some a
  unfolding map-add-dom-app-simps(1)[OF calculation(2)] nb1
  using nb10 calculation(1)
  by blast
moreover have (s ++ map-of (effect-of (φO-1 Ψ op'))) v ≠ None
  using calculation(3)
  by auto
moreover have (v, a) ∈ dom (φS Ψ (s ++ map-of (effect-of (φO-1 Ψ op'))))
  using nb11 calculation(1, 4)
  by presburger
ultimately have (φS Ψ (s ++ map-of (effect-of (φO-1 Ψ op')))) (v, a) =

```

Some True

```

using state-to-strips-state-range-is[OF assms(1)]
by simp
} note nb12 = this
{
  fix v a'
  assume (v, a') ∈ dom (map-of (effect-to-assignments op'))
  and (v, a') ∈ (U(v, a) ∈ set (effect-of op).
    { (v, a') | a'. a' ∈ (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' ∈ R+ Ψ v
  and a' ≠ a
  using calculation(2)
  by blast
  moreover have (s ++ map-of (effect-of (φO-1 Ψ op')) v = Some a)
  unfolding map-add-dom-app-simps(1)[OF calculation(3)] nb1
  using nb10 calculation(5)
  by blast
  moreover have (s ++ map-of (effect-of (φO-1 Ψ op')) v ≠ None)
  using calculation(8)
  by auto
  — TODO slow.
  moreover have (v, a' ∈ dom (φS Ψ (s ++ map-of (effect-of (φO-1 Ψ op')))))
  using state-to-strips-state-dom-is[OF assms(1), of
    s ++ map-of (effect-of (φO-1 Ψ op'))] calculation(4, 6, 9)
  by simp
  — TODO slow.
  ultimately have (φS Ψ (s ++ map-of (effect-of (φO-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 (φO-1 Ψ op'))]]
  by simp
} note nb13 = 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  $?t(v, a) = \text{Some } (\text{the } (s v) = a)$ 
  using state-to-strips-state-range-is[OF assms(1)]
  by presburger
} note nb14 = this
{
fix v a
assume  $(v, a) \in \text{dom } ?t$ 
  and  $v\text{-a-not-in: } (v, a) \notin \text{dom } (\text{map-of } (\text{effect-to-assignments } op'))$ 
moreover have  $(v, a) \in \text{dom } ?s'$ 
  using calculation(1, 2)
  unfolding dom-map-add
  by blast
moreover have  $(v, a) \in (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{V+}) \wedge s v \neq \text{None} \}.$ 
 $\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$ 
  using state-to-strips-state-dom-is[OF assms(1)] calculation(3)
  by presburger
moreover have  $v \in \text{set } ((\Psi)_{V+})$  and  $s v \neq \text{None}$  and  $a \in \mathcal{R}_+ \Psi v$ 
  using calculation(4)
  by blast+
— NOTE Hasn't this been proved before?
moreover {
have  $\text{dom } (\text{map-of } (\text{effect-to-assignments } op')) = (\bigcup (v, a) \in \text{set } (\text{effect-of } op).$ 
 $\{ (v, a) \})$ 
 $\cup (\bigcup (v, a) \in \text{set } (\text{effect-of } op).$ 
 $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
  unfolding nb2
  by blast
also have ... =  $(\bigcup (v, a) \in \text{set } (\text{effect-of } op). \{ (v, a) \})$ 
 $\cup \{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
  by blast
finally have  $\text{dom } (\text{map-of } (\text{effect-to-assignments } op'))$ 
=  $(\bigcup (v, a) \in \text{set } (\text{effect-of } op). \{ (v, a) \})$ 
 $\cup \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$ 
  by auto
then have  $(v, a) \notin (\bigcup (v, a) \in \text{set } (\text{effect-of } op).$ 
 $\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$ 
  using v-a-not-in
  by blast
}
— TODO slow.
moreover have  $v \notin \text{dom } (\text{map-of } (\text{effect-of } op))$ 
  using dom-map-of-conv-image-fst calculation
  by fastforce
moreover have  $(s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi op'))) v = s v$ 
  unfolding nb1 map-add-dom-app-simps(3)[OF calculation(9)]
  by simp
— TODO slow.

```

```

moreover have  $(v, a) \in \text{dom } ?t'$ 
  using state-to-strips-state-dom-is[ $\text{OF assms}(1)$ , of
     $s ++ \text{map-of}(\text{effect-of}(\varphi_O^{-1} \Psi op'))$ ] calculation(5, 6, 7, 8, 10)
  by simp
ultimately have  $?t'(v, a) = \text{Some}(\text{the}(s v) = a)$ 
  using state-to-strips-state-range-is[ $\text{OF assms}(1)$ ]
  by presburger
} note nb15 = this
— TODO refactor.

have nb16:  $\text{dom } ?t = (\bigcup v \in \{ v \mid v. v \in \text{set}((\Psi)v_+) \wedge s v \neq \text{None} \}.$ 
   $\{ (v, a) \mid a. a \in (\mathcal{R}_+ \Psi v) \})$ 
   $\cup \text{set}(\text{effect-of } op)$ 
   $\cup (\bigcup (v, a) \in \text{set}(\text{effect-of } op).$ 
   $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$ 
  unfolding dom-map-add nb2
  using state-to-strips-state-dom-is[ $\text{OF assms}(1)$ , of  $s$ ]
  by auto
{
{
  fix  $v a$ 
  assume  $(v, a) \in \text{dom } ?t$ 
  then consider (A)  $(v, a) \in \text{dom}(\varphi_S \Psi s)$ 
  | (B)  $(v, a) \in \text{dom}(\text{map-of}(\text{effect-to-assignments } op'))$ 
  by fast
  hence  $(v, a) \in \text{dom } ?t'$ 
  proof (cases)
    case A
    then have  $v \in \text{set}((\Psi)v_+)$  and  $s v \neq \text{None}$  and  $a \in \mathcal{R}_+ \Psi v$ 
    unfolding state-to-strips-state-dom-element-iff[ $\text{OF assms}(1)$ ]
    by blast+
    thm map-add-None state-to-strips-state-dom-element-iff[ $\text{OF assms}(1)$ ]
    moreover have  $(s ++ \text{map-of}(\text{effect-of}(\varphi_O^{-1} \Psi op'))) v \neq \text{None}$ 
    using calculation(2)
    by simp
    ultimately show ?thesis
    unfolding state-to-strips-state-dom-element-iff[ $\text{OF assms}(1)$ ]
    by blast
  next
    case B
    then have  $(v, a) \in$ 
       $\text{set}(\text{effect-of } op)$ 
       $\cup (\bigcup (v, a) \in \text{set}(\text{effect-of } op). \{ (v, a') \mid a'. a' \in \mathcal{R}_+ \Psi v \wedge a' \neq a \})$ 
      unfolding nb2
      by blast
    then consider (B1)  $(v, a) \in \text{set}(\text{effect-of } op)$ 
    | (B2)  $(v, a) \in (\bigcup (v, a) \in \text{set}(\text{effect-of } op). \{ (v, a') \mid a'. a' \in \mathcal{R}_+ \Psi v \wedge a' \neq a \})$ 
    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 ( $\varphi_O^{-1} \Psi op'$ ))
fix v a
assume v-a-in-dom-t': (v, a) ∈ dom ?t'
thm nb5
then have v-in-vs: v ∈ set (( $\Psi$ )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) ∉
    (U v ∈ { v | v. v ∈ set (( $\Psi$ )V+) ∧ s v ≠ None}. { (v, a) | a. a ∈ R+ Ψ v
})
    ∪ set (effect-of op)
    ∪ (U (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 ( $\varphi_O^{-1} \Psi op'$ )) v = None)
  case True
    then have s v ≠ None
      using u-of-v-is-not-None
      by simp
    then have (v, a) ∈ (U v ∈ { v | v. v ∈ set (( $\Psi$ )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  $v \in \text{dom}(\text{map-of}(\text{effect-of } op))$ 
  using  $u\text{-of-}v\text{-is-not-None } nb_1$ 
  by blast
  then obtain  $a'$  where  $\text{map-of-effect-of-}op\text{-}v\text{-is: map-of}(\text{effect-of } op) v = \text{Some } a'$ 
    by blast
  then have  $v\text{-}a'\text{-in: } (v, a') \in \text{set}(\text{effect-of } op)$ 
    using  $\text{map-of-SomeD}$ 
    by fast
  then show ?thesis
  proof (cases  $a = a'$ )
    case True
    then have  $(v, a) \in \text{set}(\text{effect-of } op)$ 
      using  $v\text{-}a'\text{-in}$ 
      by blast
    then show ?thesis
      using contradiction
      by blast
  next
    case False
    then have  $(v, a) \in (\bigcup_{(v, a) \in \text{set}(\text{effect-of } op)} \{(v, a') \mid a' \in \mathcal{R}_+ \Psi v \wedge a' \neq a\})$ 
      using  $v\text{-}a'\text{-in calculation } a\text{-in-range-of-}v$ 
      by blast
    thus ?thesis
      using contradiction
      by fast
  qed
  qed
}
hence  $(v, a) \in \text{dom } ?t$ 
  by argo
}
moreover have  $\text{dom } ?t \subseteq \text{dom } ?t'$  and  $\text{dom } ?t' \subseteq \text{dom } ?t$ 
  subgoal
    using calculation(1) subrelI[of dom ?t dom ?t']
    by fast
  subgoal
    using calculation(2) subrelI[of dom ?t' dom ?t]
    by argo
  done
  ultimately have  $\text{dom } ?t = \text{dom } ?t'$ 
    by force
}
note  $nb_{17} = this$ 
{
fix  $v a$ 
assume  $v\text{-}a\text{-in-dom-}t: (v, a) \in \text{dom } ?t$ 
hence  $?t(v, a) = ?t'(v, a)$ 
proof (cases  $(v, a) \in \text{dom}(\text{map-of}(\text{effect-to-assignments } op'))$ )

```

```

case True
  — TODO slow.
  — NOTE Split on the (disjunct) domain variable sets of map-of (effect-to-assignments op').
then consider (A1)  $(v, a) \in \text{set}(\text{effect-of } op)$ 
  | (A2)  $(v, a) \in (\bigcup(v, a) \in \text{set}(\text{effect-of } op))$ .
     $\{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \}$ 
using nb2
by fastforce
then show ?thesis
proof (cases)
  case A1
    then have  $?t(v, a) = \text{Some } True$ 
    unfolding map-add-dom-app-simps(1)[OF True]
    using nb9(1)
    by fast
    moreover have  $?t'(v, a) = \text{Some } True$ 
    using nb12[OF A1].
    ultimately show ?thesis..
  next
    case A2
      then have  $?t(v, a) = \text{Some } False$ 
      unfolding map-add-dom-app-simps(1)[OF True]
      using nb9(2)
      by blast
      moreover have  $?t'(v, a) = \text{Some } False$ 
      using nb13[OF True A2].
      ultimately show ?thesis..
    qed
  next
    case False
    moreover have  $?t(v, a) = \text{Some } (\text{the}(s v) = a)$ 
    using nb14[OF v-a-in-dom-t False].
    moreover have  $?t'(v, a) = \text{Some } (\text{the}(s v) = a)$ 
    using nb15[OF v-a-in-dom-t False].
    ultimately show ?thesis
    by argo
  qed
} note nb18 = this
moreover {
  fix v a
  assume  $(v, a) \in \text{dom } ?t'$ 
  hence  $?t(v, a) = ?t'(v, a)$ 
  using nb17 nb18
  by presburger
}
— TODO slow.
ultimately have  $?t \subseteq_m ?t'$  and  $?t' \subseteq_m ?t$ 
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  $\Psi$ 
and  $\forall op' \in \text{set } ops'. op' \in \text{set } ((\varphi \Psi)_O)$ 
shows execute-parallel-operator  $(\varphi_S \Psi s) ops'$ 
 $= \varphi_S \Psi (\text{execute-parallel-operator-sas-plus } s [\varphi_O^{-1} \Psi op'. op' \leftarrow ops'])$ 
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 ?Pi =  $\varphi \Psi$ 
let ?t' =  $(\varphi_S \Psi s) ++ \text{map-of } (\text{effect-to-assignments } op')$ 
and ?t =  $s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi op'))$ 
have nb1:  $?t' = \varphi_S \Psi ?t$ 
using sas-plus-equivalent-to-strips-i-a-XI[OF assms(1)] Cons.prems(2)
by force
{
have  $\forall op' \in \text{set } ops'. op' \in \text{set } (\text{strips-problem.operators-of } ?\Pi)$ 
using Cons.prems(2)
by simp
then have execute-parallel-operator  $(\varphi_S \Psi ?t) ops'$ 
 $= \varphi_S \Psi (\text{execute-parallel-operator-sas-plus } ?t [\varphi_O^{-1} \Psi x. x \leftarrow ops'])$ 
using Cons.IH[OF Cons.prems(1), of ?t]
by fastforce
hence execute-parallel-operator ?t' ops'
 $= \varphi_S \Psi (\text{execute-parallel-operator-sas-plus } ?t [\varphi_O^{-1} \Psi x. x \leftarrow ops'])$ 
using nb1
by argo
}
thus ?case
by simp
qed

```

```

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

```

```

assumes is-valid-problem-sas-plus  $\Psi$ 
and  $\forall op' \in \text{set } ops'. op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
and  $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan}$ 
 $(\text{execute-parallel-operator } (\varphi_S \Psi I) ops') \pi$ 
shows  $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan}$ 
 $(\varphi_S \Psi (\text{execute-parallel-operator-sas-plus } I [\varphi_O^{-1} \Psi op'. op' \leftarrow ops'])) \pi$ 

proof -
  let  $?I' = (\varphi_S \Psi I)$ 
  and  $?G' = \varphi_S \Psi G$ 
  and  $?ops = [\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$ 
  and  $?Pi = \varphi \Psi$ 
  let  $?J = \text{execute-parallel-operator-sas-plus } I ?ops$ 
  {
    fix  $v a$ 
    assume  $(v, a) \in \text{dom } ?G'$ 
    then have  $?G' (v, a) = \text{execute-parallel-plan}$ 
     $(\text{execute-parallel-operator } ?I' ops') \pi (v, a)$ 
    using assms(3)
    unfolding map-le-def
    by auto
    hence  $?G' (v, a) = \text{execute-parallel-plan } (\varphi_S \Psi ?J) \pi (v, a)$ 
    using sas-plus-equivalent-to-strips-i-a-XII[OF assms(1, 2)]
    by simp
  }
  thus ?thesis
    unfolding map-le-def
    by fast
qed

```

— NOTE This is a more abstract formulation of the proposition in *sas-plus-equivalent-to-strips-i* which is better suited for induction proofs. We essentially claim that given a plan the execution in STRIPS semantics of which solves the problem of reaching a transformed goal state $\varphi_S \Psi G$ from a transformed initial state $\varphi_S \Psi I$ —such as the goal and initial state of an induced STRIPS problem for a SAS+ problem—is equivalent to an execution in SAS+ semantics of the transformed plan $\varphi_P^{-1} (\varphi \Psi) \pi$ w.r.t to the original initial state I and original goal state G .

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

```

assumes is-valid-problem-sas-plus  $\Psi$ 
and  $\text{dom } I \subseteq \text{set } ((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } I. \text{the } (I v) \in \mathcal{R}_+ \Psi v$ 
and  $\text{dom } G \subseteq \text{set } ((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } G. \text{the } (G v) \in \mathcal{R}_+ \Psi v$ 
and  $\forall ops' \in \text{set } \pi. \forall op' \in \text{set } ops'. op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
and  $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi I) \pi$ 
shows  $G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\varphi_P^{-1} \Psi \pi)$ 

proof -
  let  $?vs = \text{variables-of } \Psi$ 
  and  $?psi = \varphi_P^{-1} \Psi \pi$ 
  show ?thesis

```

```

using assms
proof (induction  $\pi$  arbitrary:  $I$ )
  case Nil
  then have  $(\varphi_S \Psi G) \subseteq_m (\varphi_S \Psi I)$ 
    by fastforce
  then have  $G \subseteq_m I$ 
    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'  $\pi$ )
  let ?D = range-of  $\Psi$ 
    and ? $\Pi$  =  $\varphi \Psi$ 
    and ? $I'$  =  $\varphi_S \Psi I$ 
    and ? $G'$  =  $\varphi_S \Psi G$ 
  let ?ops = [ $\varphi_O^{-1} \Psi op'. op' \leftarrow ops'$ ]
  let ?J = execute-parallel-operator-sas-plus  $I$  ?ops
    and ? $J'$  = execute-parallel-operator ? $I'$  ?ops'
  have nb1: set ops'  $\subseteq$  set ((? $\Pi$ )O)
    using Cons.prem(6)
    unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def list-all-iff
      ListMem-iff
        by fastforce
  {
    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 auto
    moreover have op'  $\in$  set ((? $\Pi$ )O)
      using nb1 calculation(2)
      by blast
    moreover obtain op'' where op''  $\in$  set (( $\Psi$ )O+) and op' =  $\varphi_O \Psi op''$ 
      using calculation(4)
      by auto
    moreover have op = op''
      using sas-plus-operator-inverse-is[OF assms(1) calculation(5)] calculation(3, 6)
        by presburger
      ultimately have op  $\in$  set (( $\Psi$ )O+)  $\wedge$  ( $\exists$  op'  $\in$  set ops'. op' =  $\varphi_O \Psi op$ )
        by blast
  } note nb2 = this
  {
    fix op v a
    assume op  $\in$  set (( $\Psi$ )O+) and (v, a)  $\in$  set (effect-of op)
    moreover have op  $\in$  set (( $\Psi$ )O+)

```

```

using nb2 calculation(1)
by blast
moreover have is-valid-operator-sas-plus  $\Psi$  op
  using is-valid-problem-sas-plus-then(2) Cons.prems(1) calculation(3)
  by blast
ultimately have  $v \in \text{set } ((\Psi)_{\mathcal{V}_+})$ 
  using is-valid-operator-sas-plus-then(3)
  by fastforce
} note nb3 = this
{
fix op
assume op  $\in$  set ?ops
then have op  $\in$  set  $((\Psi)_{\mathcal{O}_+})$ 
  using nb2
  by blast
then have is-valid-operator-sas-plus  $\Psi$  op
  using is-valid-problem-sas-plus-then(2) Cons.prems(1)
  by blast
hence  $\forall (v, a) \in \text{set } (\text{effect-of } op). v \in \text{set } ((\Psi)_{\mathcal{V}_+})$ 
   $\wedge a \in \mathcal{R}_+ \Psi v$ 
  using is-valid-operator-sas-plus-then(3,4)
  by fast
} note nb4 = this
show ?case
proof (cases STRIPS-Semantics.are-all-operators-applicable ?I' ops'
   $\wedge$  STRIPS-Semantics.are-all-operator-effects-consistent ops')
case True
{
{
have dom I  $\subseteq$  set  $((\Psi)_{\mathcal{V}_+})$ 
  using Cons.prems(2)
  by blast
hence  $(\varphi_S^{-1} \Psi ?I') = I$ 
  using strips-state-to-state-inverse-is[OF
    Cons.prems(1) - Cons.prems(3)]
  by argo
}
then have are-all-operators-applicable-in I ?ops
   $\wedge$  are-all-operator-effects-consistent ?ops
  using sas-plus-equivalent-to-strips-i-a-IV[OF assms(1) nb1, of I] True
  by simp
moreover have  $(\varphi_P^{-1} \Psi (ops' \# \pi)) = ?ops \# (\varphi_P^{-1} \Psi \pi)$ 
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  $(\varphi_P^{-1} \Psi (ops' \# \pi))$ 
  = execute-parallel-plan-sas-plus ?J  $(\varphi_P^{-1} \Psi \pi)$ 

```

```

    by force
} note nb5 = this
— Show the goal using the IH.
{
  have dom-J-subset-eq-vs: dom ?J ⊆ set ((Ψ)_{V+})
    using sas-plus-equivalent-to-strips-i-a-IX[OF Cons.prem(2)] nb2 nb4
    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.prem(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) nb4
      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.prem(2)] calculation(1, 2, 3)
      by fastforce
    {
      fix v
      assume v ∈ dom ?J
      moreover have v ∈ set ((Ψ)_{V+})
        using nb2 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 nb3 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.prem(6)
    by simp
  moreover {
    have ?G' ⊆m execute-parallel-plan ?J' π
      using Cons.prem(7) True
      by auto
    hence (φS Ψ G) ⊆m execute-parallel-plan (φS Ψ ?J) π
  }
}

```

```

        using sas-plus-equivalent-to-strips-i-a-XIII[OF Cons.prems(1)] nb1
        by blast
    }
    ultimately have  $G \subseteq_m \text{execute-parallel-plan-sas-plus } I (\varphi_P^{-1} \Psi (ops' \# \pi))$ 
    using Cons.IH[of ?J, OF Cons.prems(1) - - Cons.prems(4, 5)]
Cons.prems(6) nb5
        by presburger
    }
    thus ?thesis.
next
case False
then have  $?G' \subseteq_m ?I'$ 
using Cons.prems(7)
by force
moreover {
have dom  $I \subseteq \text{set } ?vs$ 
using Cons.prems(2)
by simp
hence  $\neg(\text{are-all-operators-applicable-in } I ?ops$ 
 $\wedge \text{are-all-operator-effects-consistent } ?ops)$ 
using sas-plus-equivalent-to-strips-i-a-VIII[OF Cons.prems(1) -
Cons.prems(3) nb1]
        False
        by force
}
moreover {
have  $(\varphi_P^{-1} \Psi (ops' \# \pi)) = ?ops \# (\varphi_P^{-1} \Psi \pi)$ 
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  $G \subseteq_m \text{execute-parallel-plan-sas-plus } I (?ops \# (\varphi_P^{-1} \Psi \pi))$ 
 $\longleftrightarrow G \subseteq_m I$ 
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 implicitly 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  $\Psi$ 
  and STRIPS-Semantics.is-parallel-solution-for-problem
     $(\varphi \Psi) \pi$ 
  shows goal-of  $\Psi \subseteq_m \text{execute-parallel-plan-sas-plus}$ 
    (sas-plus-problem.initial-of  $\Psi$ )  $(\varphi_P^{-1} \Psi \pi)$ 
  proof -
    let ?vs = variables-of  $\Psi$ 
    and ?I = initial-of  $\Psi$ 
    and ?G = goal-of  $\Psi$ 
    let ?Pi =  $\varphi \Psi$ 
    let ?G' = strips-problem.goal-of ?Pi
    and ?I' = strips-problem.initial-of ?Pi
    let ?psi =  $\varphi_P^{-1} \Psi \pi$ 
    have dom ?I  $\subseteq$  set ?vs
      using is-valid-problem-sas-plus-then(3) assms(1)
      by auto
    moreover have  $\forall v \in \text{dom } ?I. \text{the } (?I v) \in \mathcal{R}_+ \Psi v$ 
      using is-valid-problem-sas-plus-then(4) assms(1) calculation
      by auto
    moreover have dom ?G  $\subseteq$  set ?vs and  $\forall v \in \text{dom } ?G. \text{the } (?G v) \in \mathcal{R}_+ \Psi v$ 
      using is-valid-problem-sas-plus-then(5, 6) assms(1)
      by blast+
    moreover have  $\forall ops' \in \text{set } \pi. \forall op' \in \text{set } ops'. op' \in \text{set } ((?Pi)_O)$ 
      using is-parallel-solution-for-problem-operator-set[OF assms(2)]
      by simp
    moreover {
      have ?G'  $\subseteq_m \text{execute-parallel-plan } ?I' \pi$ 
      using assms(2)
      unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def..
      moreover have ?G' =  $\varphi_S \Psi ?G$  and ?I' =  $\varphi_S \Psi ?I$ 
      by simp+
      ultimately have  $(\varphi_S \Psi ?G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi ?I) \pi$ 
      by simp
    }
    ultimately show ?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  $\Psi$ 
and STRIPS-Semantics.is-parallel-solution-for-problem ( $\varphi \Psi$ )  $\pi$ 
shows list-all (list-all ( $\lambda op. ListMem op (operators-of \Psi)$ )) ( $\varphi_P^{-1} \Psi \pi$ )
proof —
  let  $\text{?}\Pi = \varphi \Psi$ 
  let  $\text{?ops} = operators-of \Psi$ 
  and  $\text{?}\psi = \varphi_P^{-1} \Psi \pi$ 
  have is-valid-problem-strips  $\text{?}\Pi$ 
    using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]
    by simp
  have nb1:  $\forall op' \in set ((\text{?}\Pi)_O). (\exists op \in set \text{?ops}. op' = (\varphi_O \Psi op))$ 
    by auto
  {
    fix  $ops' op' op$ 
    assume  $ops' \in set \pi$  and  $op' \in set ops'$ 
    then have  $op' \in set (strips-problem.operators-of \text{?}\Pi)$ 
      using is-parallel-solution-for-problem-operator-set[OF assms(2)]
      by simp
    then obtain  $op$  where  $op \in set ((\Psi)_O)$  and  $op' = (\varphi_O \Psi op)$ 
      by auto
    then have  $(\varphi_O^{-1} \Psi op') \in set ((\Psi)_O)$ 
      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 $\psi \equiv \varphi_P^{-1} \Psi \pi$ 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.

$$G \subseteq_m execute-parallel-plan-sas-plus I \psi$$

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* $(\varphi \Psi) \pi$
shows *is-parallel-solution-for-problem* $\Psi (\varphi_P^{-1} \Psi \pi)$

proof –

let $?I = \text{initial-of } \Psi$
and $?G = \text{goal-of } \Psi$
and $?ops = \text{operators-of } \Psi$
and $?ψ = \varphi_P^{-1} \Psi \pi$

show $?thesis$
unfolding *is-parallel-solution-for-problem-def Let-def*
proof (*rule conjI*)
show $?G \subseteq_m \text{execute-parallel-plan-sas-plus } ?I ?ψ$
using *sas-plus-equivalent-to-strips-i[OF assms]*.

next
show *list-all (list-all (λop. 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 $\forall op \in \text{set ops}. op \in \text{set } ((\Psi)_{\mathcal{O}+})$
and $op' \in \text{set } [\varphi_O \Psi op. op \leftarrow ops]$
obtains *op where* $op \in \text{set ops}$
and $op' = \varphi_O \Psi op$

proof –

let $?Π = \varphi \Psi$
let $?ops = \text{operators-of } \Psi$
obtain *op where* $op \in \text{set ops}$ **and** $op' = \varphi_O \Psi op$
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 $\forall op \in \text{set ops}. op \in \text{set } ((\Psi)_{\mathcal{O}+})$
and $op' \in \text{set } [\varphi_O \Psi op. op \leftarrow ops]$
shows $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$
and *is-valid-operator-strips* $(\varphi \Psi) op'$

proof –

let $?Π = \varphi \Psi$
let $?ops = \text{operators-of } \Psi$
and $?ops' = \text{strips-problem.operators-of } ?Π$

obtain *op where* $op \in \text{set ops}$ **and** $op' \text{-is: } op' = \varphi_O \Psi op$
using *strips-equivalent-to-sas-plus-i-a-I[OF assms]*.

```

then have nb:  $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
  using assms(2)  $op$ -in  $op'$ -is
  by fastforce
thus  $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
  and is-valid-operator-strips ? $\Pi$   $op'$ 
proof -
  have  $\forall op' \in \text{set } ?ops'. \text{is-valid-operator-strips } ?\Pi op'$ 
  using is-valid-problem-sas-plus-then-strips-transformation-too-iii[OF assms(1)]
  unfolding list-all-iff.
  thus is-valid-operator-strips ? $\Pi$   $op'$ 
    using nb
    by fastforce
qed fastforce
qed

```

```

lemma strips-equivalent-to-sas-plus-i-a-III:
assumes is-valid-problem-sas-plus  $\Psi$ 
  and  $\forall op \in \text{set } ops. op \in \text{set } ((\Psi)_{\mathcal{O}+})$ 
shows execute-parallel-operator  $(\varphi_S \Psi s) [\varphi_O \Psi op. op \leftarrow ops]$ 
   $= (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus } s ops))$ 
proof -
{
  fix  $op s$ 
  assume  $op \in \text{set } ((\Psi)_{\mathcal{O}+})$ 
  moreover have  $(\varphi_O \Psi op) \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
    using calculation
    by simp
  moreover have  $(\varphi_S \Psi s) ++ \text{map-of } (\text{effect-to-assignments } (\varphi_O \Psi op))$ 
   $= (\varphi_S \Psi (s ++ \text{map-of } (\text{effect-of } (\varphi_O^{-1} \Psi (\varphi_O \Psi op)))))$ 
  using sas-plus-equivalent-to-strips-i-a-XI[OF assms(1) calculation(2)]
  by blast
  moreover have  $(\varphi_O^{-1} \Psi (\varphi_O \Psi op)) = op$ 
  using sas-plus-operator-inverse-is[OF assms(1) calculation(1)].
  ultimately have  $(\varphi_S \Psi s) \gg (\varphi_O \Psi op)$ 
   $= (\varphi_S \Psi (s \gg_+ op))$ 
  unfolding execute-operator-def execute-operator-sas-plus-def
  by simp
} note  $nb_1 = \text{this}$ 
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  $\gg_+$  op
let ?s' =  $\varphi_S \Psi s$ 
  and ?ops' =  $[\varphi_O \Psi op. op \leftarrow op \# ops]$ 
let ?t' = ?s'  $\gg (\varphi_O \Psi op)$ 
have execute-parallel-operator ?s' ?ops'
  = execute-parallel-operator ?t'  $[\varphi_O \Psi x. x \leftarrow ops]$ 
  unfolding execute-operator-def
  by simp
moreover have ( $\varphi_S \Psi$  (execute-parallel-operator-sas-plus s (op # ops)))
  = ( $\varphi_S \Psi$  (execute-parallel-operator-sas-plus ?t ops))
  unfolding execute-operator-sas-plus-def
  by simp
moreover {
  have ?t' = ( $\varphi_S \Psi$  ?t)
  using nb1 Cons.prem(2)
  by simp
  hence execute-parallel-operator ?t'  $[\varphi_O \Psi x. x \leftarrow ops]$ 
  = ( $\varphi_S \Psi$  (execute-parallel-operator-sas-plus ?t ops))
  using Cons.IH[of ?t] Cons.prem
  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  $\Psi$ 
and  $\forall op \in set ops. op \in set ((\Psi)_{\mathcal{O}_+})$ 
and are-all-operators-applicable-in I ops
   $\wedge$  are-all-operator-effects-consistent ops
shows STRIPS-Semantics.are-all-operators-applicable ( $\varphi_S \Psi I$ )  $[\varphi_O \Psi op. op \leftarrow ops]$ 
   $\wedge$  STRIPS-Semantics.are-all-operator-effects-consistent  $[\varphi_O \Psi op. op \leftarrow ops]$ 
proof –
let ?vs = variables-of  $\Psi$ 
  and ?ops = operators-of  $\Psi$ 
let ?I' =  $\varphi_S \Psi I$ 
  and ?ops' =  $[\varphi_O \Psi op. op \leftarrow ops]$ 
have nb1:  $\forall op \in set ops. is-operator-applicable-in I op$ 
  using assms(3)
  unfolding are-all-operators-applicable-in-def list-all-iff
  by blast
have nb2:  $\forall op \in set ops. is-valid-operator-sas-plus \Psi op$ 
  using is-valid-problem-sas-plus-then(2) assms(1, 2)
  unfolding is-valid-operator-sas-plus-def
  by auto
have nb3:  $\forall op \in set ops. map-of (precondition-of op) \subseteq_m I$ 
  using nb1

```

```

unfolding is-operator-applicable-in-def list-all-iff
by blast
{
  fix op1 op2
  assume op1 ∈ set ops and op2 ∈ set ops
  hence are-operator-effects-consistent op1 op2
  using assms(3)
  unfolding are-all-operator-effects-consistent-def list-all-iff
  by blast
} note nb4 = this
{
  fix op1 op2
  assume op1 ∈ set ops and op2 ∈ set ops
  hence ∀(v, a) ∈ set (effect-of op1). ∀(v', a') ∈ set (effect-of op2).
    v ≠ v' ∨ a = a'
  using nb4
  unfolding are-operator-effects-consistent-def Let-def list-all-iff
  by presburger
} note nb5 = this
{
  fix op1' op2' I
  assume op1' ∈ set ?ops'
    and op2' ∈ set ?ops'
    and ∃(v, a) ∈ set (add-effects-of op1'). ∃(v', a') ∈ set (delete-effects-of op2').
      (v, a) = (v', a')
  moreover obtain op1 op2
    where op1 ∈ set ops
      and op1' =  $\varphi_O \Psi op_1$ 
      and op2 ∈ set ops
      and op2' =  $\varphi_O \Psi op_2$ 
  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-op2: is-valid-operator-sas-plus  $\Psi op_2$ 
  using calculation(4, 6) nb2
  by blast+
moreover obtain v v' a a'
  where (v, a) ∈ set (add-effects-of op1')
    and (v', a') ∈ set (delete-effects-of op2')
    and (v, a) = (v', a')
  using calculation
  by blast
moreover have (v, a) ∈ set (effect-of op1)
  using calculation(5, 10)
  unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
    sasp-op-to-strips-def Let-def
  by fastforce
moreover have v = v' and a = a'
  using calculation(12)

```

```

by simp+
moreover {
  have  $(v', a') \in (\bigcup(v, a) \in \text{set } (\text{effect-of } op_2))$ .
    {  $(v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a$  }
  using sasp-op-to-strips-set-delete-effects-is
    calculation(7, 9, 11)
  by blast
  then obtain  $v'' a''$  where  $(v'', a'') \in \text{set } (\text{effect-of } op_2)$ 
    and  $(v', a') \in \{ (v'', a'') \mid a'''. a''' \in (\mathcal{R}_+ \Psi v'') \wedge a''' \neq a'' \}$ 
    by blast
  moreover have  $(v', a'') \in \text{set } (\text{effect-of } op_2)$ 
    using calculation
    by blast
  moreover have  $a' \in \mathcal{R}_+ \Psi v''$  and  $a' \neq a''$ 
    using calculation(1, 2)
    by fast+
  ultimately have  $\exists a''. (v', a'') \in \text{set } (\text{effect-of } op_2) \wedge a' \in (\mathcal{R}_+ \Psi v') \wedge a' \neq a''$ 
    by blast
}
moreover obtain  $a''$  where  $a' \in \mathcal{R}_+ \Psi v'$ 
and  $(v', a'') \in \text{set } (\text{effect-of } op_2)$ 
and  $a' \neq a''$ 
using calculation(16)
by blast
moreover have  $\exists (v, a) \in \text{set } (\text{effect-of } op_1). (\exists (v', a') \in \text{set } (\text{effect-of } op_2).$ 
 $v = v' \wedge a \neq a')$ 
using calculation(13, 14, 15, 17, 18, 19)
by blast
— TODO slow.
ultimately have  $\exists op_1 \in \text{set } ops. \exists op_2 \in \text{set } ops. \neg \text{are-operator-effects-consistent}_{op_1 op_2}$ 
  unfolding are-operator-effects-consistent-def list-all-iff
  by fastforce
} note nb6 = this
show ?thesis
proof (rule conjI)
{
  fix  $op'$ 
  assume  $op' \in \text{set } ?ops'$ 
  moreover obtain  $op$  where  $op\text{-in: } op \in \text{set } ops$ 
    and  $op'\text{-is: } op' = \varphi_{\mathcal{O}} \Psi op$ 
    and  $op'\text{-in: } op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$ 
    and  $is\text{-valid-}op': is\text{-valid-operator-strips } (\varphi \Psi) op'$ 
    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  $is\text{-valid-}op: is\text{-valid-operator-sas-plus } \Psi op$ 
    using nb2 calculation(2)..
```

```

{
  fix v a
  assume v-a-in-preconditions': (v, a) ∈ set (strips-operator.precondition-of
op')
  have v-a-in-preconditions: (v, a) ∈ set (precondition-of op)
    using op'-is
    unfolding SAS-Plus-STRIPS.sasp-op-to-strips-def
      sasp-op-to-strips-def Let-def
      using v-a-in-preconditions'
      by force
  moreover have v ∈ set ?vs and a ∈ R+ Ψ v
    using is-valid-operator-sas-plus-then(1,2) is-valid-op calculation(1)
    by fastforce+
  moreover {
    have ∀ (v, a) ∈ set (precondition-of op). ∀ (v', a') ∈ set (precondition-of
op).
      v ≠ v' ∨ a = a'
      using is-valid-operator-sas-plus-then(5) is-valid-op
      by fast
    hence map-of (precondition-of op) v = Some a
      using map-of-constant-assignments-defined-if[OF - v-a-in-preconditions]
      by blast
  }
  moreover have v ∈ dom (map-of (precondition-of op))
    using calculation(4)
    by blast
  moreover have I v = Some a
    using nb3
    unfolding map-le-def
    using op-in calculation(4, 5)
    by metis
  moreover have (v, a) ∈ dom ?I'
    using state-to-strips-state-dom-element-iff[OF assms(1)]
      calculation(2, 3, 6)
    by simp
  ultimately have ?I' (v, a) = 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 nb4
        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  $\Psi$ 
and  $\forall op \in set ops. op \in set ((\Psi)_{\mathcal{O}+})$ 
and  $\neg(\text{are-all-operators-applicable-in } s \text{ ops}$ 
 $\wedge \text{are-all-operator-effects-consistent } ops)$ 
shows  $\neg(\text{STRIPS-Semantics.are-all-operators-applicable } (\varphi_S \Psi s) [\varphi_O \Psi op. op$ 
 $\leftarrow ops]$ 
 $\wedge \text{STRIPS-Semantics.are-all-operator-effects-consistent } [\varphi_O \Psi op. op \leftarrow ops])$ 
proof -
let ?vs = variables-of  $\Psi$ 
and ?ops = operators-of  $\Psi$ 
let ?s' =  $\varphi_S \Psi s$ 
and ?ops' =  $[\varphi_O \Psi op. op \leftarrow ops]$ 
{
fix op
assume op  $\in set ops$ 
hence  $\exists op' \in set ?ops'. op' = \varphi_O \Psi op$ 
by simp
} note nb1 = this
{
fix op
assume op  $\in set ops$ 
then have op  $\in set ((\Psi)_{\mathcal{O}+})$ 
using assms(2)
by blast
then have is-valid-operator-sas-plus  $\Psi op$ 
using is-valid-problem-sas-plus-then(2) assms(1)
unfolding is-valid-operator-sas-plus-def
by auto
hence  $\forall (v, a) \in set (\text{precondition-of } op). \forall (v', a') \in set (\text{precondition-of } op).$ 
 $v \neq v' \vee a = a'$ 
using is-valid-operator-sas-plus-then(5)
unfolding is-valid-operator-sas-plus-def
by fast
} note nb2 = this
{
consider (A)  $\neg\text{are-all-operators-applicable-in } s \text{ ops}$ 
| (B)  $\neg\text{are-all-operator-effects-consistent } ops$ 
using assms(3)
by blast
hence  $\neg\text{STRIPS-Semantics.are-all-operators-applicable } ?s' ?ops'$ 
 $\vee \neg\text{STRIPS-Semantics.are-all-operator-effects-consistent } ?ops'$ 
proof (cases)
case A
then obtain op where op-in:  $op \in set ops$ 
and not-precondition-map-le-s:  $\neg(\text{map-of } (\text{precondition-of } op) \subseteq_m s)$ 

```

```

using A
unfolding are-all-operators-applicable-in-def list-all-iff
  is-operator-applicable-in-def
  by blast
then obtain op' where op'-in: op' ∈ set ?ops' and op'-is: op' = φO Ψ op
  using nb1
  by blast
have ¬are-all-operators-applicable ?s' ?ops'
proof (rule ccontr)
  assume ¬¬are-all-operators-applicable ?s' ?ops'
  then have all-operators-applicable: are-all-operators-applicable ?s' ?ops'
    by simp
  moreover {
    fix v
    assume v ∈ dom (map-of (precondition-of op))
    moreover obtain a where map-of (precondition-of op) v = Some a
      using calculation
      by blast
    moreover have (v, a) ∈ set (precondition-of op)
      using map-of-SomeD[OF calculation(2)].
    moreover have (v, a) ∈ set (strips-operator.precondition-of op')
      using op'-is
      unfolding sasp-op-to-strips-def
        SAS-Plus-STRIPS.sasp-op-to-strips-def
        using calculation(3)
        by auto
    moreover have ?s' (v, a) = Some True
      using all-operators-applicable calculation
      unfolding are-all-operators-applicable-def
        STRIPS-Representation.is-operator-applicable-in-def
          is-operator-applicable-in-def Let-def list-all-iff
        using op'-in
        by fast
    moreover have (v, a) ∈ dom ?s'
      using calculation(5)
      by blast
    moreover have (v, a) ∈ 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 op'1 op'2 where op'1 ∈ set ?ops' and op'1 = φO Ψ op1
    and op'1 ∈ set ?ops' and op'2-is: op'2 = φO Ψ op2
    using nb1[OF calculation(1)] nb1[OF calculation(2)]
    by blast
  moreover have (v, a) ∈ set (add-effects-of op'1)
    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  $a \in \mathcal{R}_+ \Psi v$ 
  using is-valid-operator-sas-plus-then(4) calculation v-a-in
  unfolding is-valid-operator-sas-plus-def
  by fastforce
ultimately have  $(v, a) \in \text{set}(\text{delete-effects-of } op_2')$ 
  using sasp-op-to-strips-set-delete-effects-is[of  $\Psi op_2$ ]
    v'-a'-in v-is a-is
  using op2'-is
  by blast
}
— TODO slow.
ultimately have  $\exists op_1' \in \text{set}(\text{?ops}'). \exists op_2' \in \text{set}(\text{?ops}')$ .
 $\exists (v, a) \in \text{set}(\text{delete-effects-of } op_2'). \exists (v', a') \in \text{set}(\text{add-effects-of } op_1')$ .
 $(v, a) = (v', a')$ 
  by fastforce
}
then have  $\neg \text{STRIPS-Semantics.are-all-operator-effects-consistent } \text{?ops}'$ 
  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  $\Psi$ 
and  $\text{dom } I \subseteq \text{set}((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } I. \text{the}(I v) \in \mathcal{R}_+ \Psi v$ 
and  $\text{dom } G \subseteq \text{set}((\Psi)_{V+})$ 
and  $\forall v \in \text{dom } G. \text{the}(G v) \in \mathcal{R}_+ \Psi v$ 
and  $\forall ops \in \text{set } \psi. \forall op \in \text{set } ops. op \in \text{set}((\Psi)_{O+})$ 
and  $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi$ 
shows  $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi I) (\varphi_P \Psi \psi)$ 
proof –
let  $?Pi = \varphi \Psi$ 
  and  $?G' = \varphi_S \Psi G$ 
show ?thesis
using assms
proof (induction  $\psi$  arbitrary:  $I$ )
  case Nil

```

```

let ?I' =  $\varphi_S \Psi I$ 
have  $G \subseteq_m I$ 
  using Nil
  by simp
moreover have  $?G' \subseteq_m ?I'$ 
  using state-to-strips-state-map-le-iff[OF Nil.prems(1, 4, 5)]
  calculation..
ultimately show ?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 =  $\varphi_P \Psi (ops \# \psi)$ 
let ?I' =  $\varphi_S \Psi I$ 
and ?J' =  $\varphi_S \Psi ?J$ 
and ?ops' = [ $\varphi_O \Psi op. op \leftarrow ops$ ]
{
fix op v a
assume  $op \in set ops$  and  $(v, a) \in set (effect-of op)$ 
moreover have  $op \in set ?ops$ 
  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  $v \in set ((\Psi)_v)$ 
  and  $a \in \mathcal{R}_+ \Psi v$ 
  using is-valid-operator-sas-plus-then(3,4)
  by fastforce+
} note nb1 = this
show ?case
proof (cases are-all-operators-applicable-in I ops
  ∧ are-all-operator-effects-consistent ops)
case True
{
have  $(\varphi_P \Psi (ops \# \psi)) = ?ops' \# (\varphi_P \Psi \psi)$ 
  unfolding sas-plus-parallel-plan-to-strips-parallel-plan-def
    SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def
  by simp
moreover have  $\forall op \in set ops. op \in set ((\Psi)_o)$ 
  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.prem(1) - True]
calculation
  by blast+
ultimately have execute-parallel-plan ?I' ?π
= execute-parallel-plan (execute-parallel-operator ?I' ?ops') (φP 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.prem(2)
  by blast
moreover have ∀ op ∈ set ops. ∀ (v, a) ∈ set (effect-of op).
  v ∈ set ((Ψ)V+)
  using nb1(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 nb2 = this
moreover {
  have dom I ⊆ set (sas-plus-problem.variables-of Ψ)
  using Cons.prem(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.prem(2) calculation
    by blast
    ultimately have the (I v) ∈ set (the (range-of Ψ v))
    using Cons.prem(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:  $\forall v \in \text{dom } ?J. \text{the } (?J v) \in \text{set } (\text{the } (\text{range-of } \Psi v))$ 
    using sas-plus-equivalent-to-strips-i-a-X[of I set ?vs range-of  $\Psi$  ops]
      calculation
    by fast
  moreover {
    fix v
    assume  $v \in \text{dom } ?J$ 
    moreover have  $v \in \text{set } ((\Psi)v_+)$ 
      using nb2 calculation
      by blast
    moreover have  $\text{set } (\text{the } (\text{range-of } \Psi v)) = \mathcal{R}_+ \Psi v$ 
      using set-the-range-of-is-range-of-sas-plus-if[OF assms(1)]
        calculation(2)
      by presburger
    ultimately have  $\text{the } (?J v) \in \mathcal{R}_+ \Psi v$ 
      using nb3
      by blast
  }
  ultimately have  $\forall v \in \text{dom } ?J. \text{the } (?J v) \in \mathcal{R}_+ \Psi v$ 
    by fast
}
moreover have  $\forall \text{ops} \in \text{set } \psi. \forall \text{op} \in \text{set } \text{ops}. \text{op} \in \text{set } ?\text{ops}$ 
  using Cons.prems(6)
  by auto
moreover have  $G \subseteq_m \text{execute-parallel-plan-sas-plus } ?J \psi$ 
  using Cons.prems(7) True
  by simp
ultimately have  $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } ?J' (\varphi_P \Psi \psi)$ 
  using Cons.IH[of ?J, OF Cons.prems(1) -- Cons.prems(4, 5)]
  by fastforce
}
moreover have  $\text{execute-parallel-operator } ?I' ?\text{ops}' = ?J'$ 
using assms(1) strips-equivalent-to-sas-plus-i-a-III[OF assms(1)] Cons.prems(6)
  by auto
ultimately show ?thesis
  by argo
next
case False
then have nb:  $G \subseteq_m I$ 
  using Cons.prems(7)
  by force
moreover {
  have  $? \pi = ?\text{ops}' \# (\varphi_P \Psi \psi)$ 
  unfolding sas-plus-parallel-plan-to-strips-parallel-plan-def
    SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    SAS-Plus-STRIPS.sasp-op-to-strips-def 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.prem(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.prem(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.prem(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))))
(φP Ψ ψ)
proof –
  let ?ops = operators-of Ψ
  let ?Π = φ Ψ
  let ?ops' = strips-problem.operators-of ?Π
  and ?π = φP Ψ ψ
have is-valid-problem-strips ?Π
using is-valid-problem-sas-plus-then-strips-transformation-too[OF assms(1)]
by simp
have nb1: ∀ op ∈ set ?ops. (∃ op' ∈ set ?ops'. op' = (φO Ψ op))
unfolding sas-plus-problem-to-strips-problem-def Let-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' = (φO Ψ op)
    using nb1 calculation(3)
    by auto
  ultimately have (φO 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.

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

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  $\Psi$ 
and is-parallel-solution-for-problem  $\Psi \psi$ 
shows STRIPS-Semantics.is-parallel-solution-for-problem  $(\varphi \Psi) (\varphi_P \Psi \psi)$ 

```

proof –

```

let  $\Pi = \varphi \Psi$ 
let  $I' = \text{strips-problem.initial-of } \Pi$ 
and  $G' = \text{strips-problem.goal-of } \Pi$ 
and  $\text{ops}' = \text{strips-problem.operators-of } \Pi$ 
and  $\pi = \varphi_P \Psi \psi$ 
show ?thesis
unfolding STRIPS-Semantics.is-parallel-solution-for-problem-def

```

proof (rule conjI)

```

show  $G' \subseteq_m \text{execute-parallel-plan } I' \pi$ 
using strips-equivalent-to-sas-plus-i[OF assms]
by simp

```

next

```

show list-all ( $\lambda op. \text{ListMem } op \text{ ?ops}'$ )  $\pi$ 
using strips-equivalent-to-sas-plus-ii[OF assms].

```

qed

qed

lemma *embedded-serial-sas-plus-plan-operator-structure*:

assumes $\text{ops} \in \text{set} (\text{embed } \psi)$

obtains op

where $op \in \text{set } \psi$

and $[\varphi_O \Psi op. op \leftarrow \text{ops}] = [\varphi_O \Psi op]$

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. [φO Ψ op. op ← ops] = [φO Ψ op]
    by auto
}
thus ?thesis
  using that
  by meson
qed

private lemma serial-sas-plus-equivalent-to-serial-strips-i:
  assumes ops ∈ set (φP Ψ (embed ψ))
  obtains op where op ∈ set ψ and ops = [φO Ψ op]
proof -
  let ?ψ' = embed ψ
  {
    have set (φP Ψ (embed ψ)) = { [φO 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 = [φO Ψ op. op ← ops']
      using assms(1) calculation
      by blast
    moreover obtain op where op ∈ set ψ and ops = [φO Ψ op]
      using embedded-serial-sas-plus-plan-operator-structure calculation(2, 3)
      by blast
    ultimately have ∃ op ∈ set ψ. ops = [φO Ψ op]
      by meson
  }
  thus ?thesis
    using that..
qed

private lemma serial-sas-plus-equivalent-to-serial-strips-ii[simp]:
  concat (φP Ψ (embed ψ)) = [φO Ψ op. op ← ψ]
proof -
  let ?ψ' = List-Supplement.embed ψ
  have concat (φP Ψ ?ψ') = map (λop. φO 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 ( $\lambda op. \varphi_O \Psi op$ )  $\psi$ 
unfolding concat-is-inverse-of-embed[of  $\psi$ ]..
finally show concat ( $\varphi_P \Psi (\text{embed } \psi)$ ) = [ $\varphi_O \Psi op. op \leftarrow \psi$ ].
qed

```

Having established the equivalence of parallel STRIPS and SAS+, we can now show the equivalence in the serial case. The proof combines the embedding theorem for serial SAS+ solutions (??), the parallel plan equivalence theorem ??, and the flattening theorem for parallel STRIPS plans (??). More precisely, given a serial SAS+ solution 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

$$\text{concat } \pi = \text{concat} (\varphi_P \Psi (\text{embed } \psi))$$

can be reduced to the intuitive form

$$\pi = [\varphi_O \Psi op. op \leftarrow \psi]$$

which concludes the proof.

theorem

serial-sas-plus-equivalent-to-serial-strips:

assumes *is-valid-problem-sas-plus* Ψ

and *SAS-Plus-Semantics.is-serial-solution-for-problem* $\Psi \psi$

shows *STRIPS-Semantics.is-serial-solution-for-problem* ($\varphi \Psi$) [$\varphi_O \Psi op. op \leftarrow \psi$]

proof –

let $\psi' = \text{embed } \psi$

and $\Pi = \varphi \Psi$

let $\pi' = \varphi_P \Psi \psi'$

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

{

have *SAS-Plus-Semantics.is-parallel-solution-for-problem* $\Psi \psi'$

using *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus*[*OF assms*]
by *simp*

hence *STRIPS-Semantics.is-parallel-solution-for-problem* $\Pi \pi'$

using *strips-equivalent-to-sas-plus*[*OF assms(1)*]
by *simp*

```

}

moreover have ?π = [φO 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 = [φO 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 [φO-1 Π op. op ← ops'] = [φO-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. [φO-1 Π op. op ← ops'] = [φO-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 (φP-1 Π (embed π))
obtains op where op ∈ set π and ops = [φO-1 Π op]
proof -
  let ?π' = embed π
  {
    have set (φP-1 Π (embed π)) = { [φO-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 = [φO-1 Π op. op ←
ops']

```

```

using assms(1) calculation
by blast
moreover obtain op where op ∈ set π and ops = [φO-1 Π op]
  using embedded-serial-strips-plan-operator-structure calculation(2, 3)
  by blast
ultimately have ∃ op ∈ set π. ops = [φO-1 Π op]
  by meson
}
thus ?thesis
  using that..
qed

private lemma serial-strips-equivalent-to-serial-sas-plus-ii[simp]:
  concat (φP-1 Π (embed π)) = [φO-1 Π op. op ← π]
proof -
  let ?π' = List-Supplement.embed π
  have concat (φP-1 Π ?π') = map (λop. φO-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. φO-1 Π op) π
    unfolding concat-is-inverse-of-embed[of π]..
  finally show concat (φP-1 Π (embed π)) = [φO-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 [φO-1 Ψ op. op ←
  π]
proof -
  let ?π' = embed π
  and ?Π = φ Ψ
  let ?ψ' = φP-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
≡ { π. STRIPS-Semantics.is-serial-solution-for-problem Π π ∧ length π = k }

abbreviation bounded-solution-set-strips
:: ('variable × 'domain) strips-problem
⇒ nat
⇒ ('variable × 'domain) strips-plan set
where bounded-solution-set-strips Π N ≡ (⋃ k ∈ {0..N}. bounded-solution-set-strips' Π k)

— Show that plan transformation for all SAS Plus solutions yields a STRIPS so-

lution 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 $P_k \equiv \text{bounded-plan-set}(\text{operators-of } \Psi) k$ with length bound k . The injectiveness of $Sol_k \equiv \text{bounded-solution-set-sas-plus } \Psi k$ —the set of solutions with length bound k —then follows from the subset relation $Sol_k \subseteq P_k$.

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  $?op_1' = \text{strips-operator.precondition-of}$ 
 $?op_2'$ 
    using assms
    by argo
    hence sas-plus-operator.precondition-of  $op_1 = \text{sas-plus-operator.precondition-of}$ 
 $op_2$ 
    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  $?op_1' = \text{strips-operator.add-effects-of}$ 
 $?op_2'$ 
    using assms
    unfolding sasp-op-to-strips-def Let-def
    by argo
    hence sas-plus-operator.effect-of  $op_1 = \text{sas-plus-operator.effect-of}$ 
 $op_2$ 
    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  $\Psi$ 
shows inj-on ( $\lambda\psi. [\varphi_O \Psi op. op \leftarrow \psi]$ ) (bounded-plan-set (sas-plus-problem.operators-of
 $\Psi) k)$ 
proof -
  let  $?ops = \text{sas-plus-problem.operators-of } \Psi$ 

  and  $?var_P = \lambda\psi. [\varphi_O \Psi op. op \leftarrow \psi]$ 
  let  $?P = \text{bounded-plan-set } ?ops$ 
  {
    fix  $\psi_1 \psi_2$ 
    assume  $\psi_1\text{-in: } \psi_1 \in ?P k$ 

```

and $\psi_2\text{-in: } \psi_2 \in ?P k$
and $\varphi_P\text{-of-}\psi_1\text{-is-}\varphi_P\text{-of-}\psi_2: (?\varphi_P \psi_1) = (?\varphi_P \psi_2)$
hence $\psi_1 = \psi_2$
proof (*induction k arbitrary:* $\psi_1 \psi_2$)
 case 0
 then have *length* $\psi_1 = 0$
 and *length* $\psi_2 = 0$
 using $\psi_1\text{-in } \psi_2\text{-in}$
 unfolding *bounded-solution-set-sas-plus'-def*
 by *blast+*
 then show $?case$
 by *blast*
next
 case (*Suc k*)
 moreover have *length* $\psi_1 = Suc k$ **and** *length* $\psi_2 = Suc k$
 using *length-Suc-conv* *Suc(2, 3)*
 unfolding *bounded-solution-set-sas-plus'-def*
 by *blast+*
 moreover obtain $op_1 \psi_1'$ **where** $\psi_1 = op_1 \# \psi_1'$
 and *set* ($op_1 \# \psi_1'$) \subseteq *set* $?ops$
 and *length* $\psi_1' = k$
 using *calculation(5)* *Suc(2)*
 unfolding *length-Suc-conv*
 by *blast*
 moreover obtain $op_2 \psi_2'$ **where** $\psi_2 = op_2 \# \psi_2'$
 and *set* ($op_2 \# \psi_2'$) \subseteq *set* $?ops$
 and *length* $\psi_2' = k$
 using *calculation(6)* *Suc(3)*
 unfolding *length-Suc-conv*
 by *blast*
 moreover have *set* $\psi_1' \subseteq$ *set* $?ops$ **and** *set* $\psi_2' \subseteq$ *set* $?ops$
 using *calculation(8, 11)*
 by *auto+*
 moreover have $\psi_1' \in ?P k$ **and** $\psi_2' \in ?P k$
 using *calculation(9, 12, 13, 14)*
 by *fast+*
 moreover have $?_\varphi \psi_1' = ?_\varphi \psi_2'$
 using *Suc.prefs(3)* *calculation(7, 10)*
 by *fastforce*
 moreover have $\psi_1' = \psi_2'$
 using *Suc.IH[of* $\psi_1' \psi_2'$ *, OF calculation(15, 16, 17)]*
 by *simp*
 moreover have $?_\varphi \psi_1 = (\varphi_O \Psi op_1) \# ?_\varphi \psi_1'$
 and $?_\varphi \psi_2 = (\varphi_O \Psi op_2) \# ?_\varphi \psi_2'$
 using *Suc.prefs(3)* *calculation(7, 10)*
 by *fastforce+*
 moreover have $(\varphi_O \Psi op_1) = (\varphi_O \Psi op_2)$
 using *Suc.prefs(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  $\Psi$ 
shows inj-on ( $\lambda\psi. [\varphi_O \Psi op. op \leftarrow \psi]$ ) (bounded-solution-set-sas-plus'  $\Psi k$ )
proof –
  let ?ops = sas-plus-problem.operators-of  $\Psi$ 
  and ? $\varphi_P$  =  $\lambda\psi. [\varphi_O \Psi op. op \leftarrow \psi]$ 
  {
    fix  $\psi$ 
    assume  $\psi \in$  bounded-solution-set-sas-plus'  $\Psi k$ 
    then have set  $\psi \subseteq$  set ?ops
    and length  $\psi = k$ 
    unfolding bounded-solution-set-sas-plus'-def is-serial-solution-for-problem-def
  Let-def
    list-all-iff ListMem-iff
    by fast+
    hence  $\psi \in$  bounded-plan-set ?ops  $k$ 
    by blast
  }
  hence bounded-solution-set-sas-plus'  $\Psi k \subseteq$  bounded-plan-set ?ops  $k$ 
  by blast
  moreover have inj-on ? $\varphi_P$  (bounded-plan-set ?ops  $k$ )
  using sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a[OF assms(1)].
  ultimately show ?thesis
  using inj-on-subset[of ? $\varphi_P$  bounded-plan-set ?ops  $k$  bounded-solution-set-sas-plus'
   $\Psi k]$ 
  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  $\Psi$ 
shows ( $\lambda\psi. [\varphi_O \Psi op. op \leftarrow \psi]$ ) ‘ (bounded-solution-set-sas-plus'  $\Psi k$ )
  = bounded-solution-set-strips' ( $\varphi \Psi$ )  $k$ 

```

```

proof —
let  $\exists \Pi = \varphi \Psi$ 
    and  $\exists \varphi_P = \lambda \psi. [\varphi_O \Psi op. op \leftarrow \psi]$ 
let  $\exists Sol_k = \text{bounded-solution-set-sas-plus}' \Psi k$ 
    and  $\exists Sol'_k = \text{bounded-solution-set-strips}' \exists \Pi k$ 
{
    assume  $\exists \varphi_P ' Sol_k \neq Sol'_k$ 
    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  $\pi$  where  $\pi \in \varphi_P ' Sol_k$  and  $\pi \notin Sol'_k$ 
            using calculation
            by blast
        moreover obtain  $\psi$  where  $\text{length } \psi = k$ 
            and  $SAS\text{-Plus-Semantics.is-serial-solution-for-problem } \Psi \psi$ 
            and  $\pi = \varphi_P \psi$ 
            using calculation(2)
            unfolding  $\text{bounded-solution-set-sas-plus}'\text{-def}$ 
            by blast
        moreover have  $\text{length } \pi = k$  and  $STRIPS\text{-Semantics.is-serial-solution-for-problem }$ 
 $\exists \Pi \pi$ 
        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  $\text{bounded-solution-set-strips}'\text{-def}$ 
            using calculation(7, 8)
            by simp
        ultimately show  $?thesis$ 
            by fast
    next
        case B
        moreover obtain  $\pi$  where  $\pi \in Sol'_k$  and  $\pi \notin \varphi_P ' Sol_k$ 
            using calculation
            by blast
        moreover have  $STRIPS\text{-Semantics.is-serial-solution-for-problem } \exists \Pi \pi$ 
            and  $\text{length } \pi = k$ 
            using calculation(2)
            unfolding  $\text{bounded-solution-set-strips}'\text{-def}$ 
            by simp+
— Construct the counter example  $\psi \equiv [\varphi_O^{-1} \exists \Pi op. op \leftarrow \pi]$  and show
that  $\psi \in Sol_k$  as well as  $\varphi_P \psi = \pi$  hence  $\pi \in \varphi_P ' Sol_k$ .

```

```

moreover have length [ $\varphi_O^{-1} \Psi op. op \leftarrow \pi$ ] = k
  and SAS-Plus-Semantics.is-serial-solution-for-problem  $\Psi [\varphi_O^{-1} \Psi op. op$ 
 $\leftarrow \pi]$ 
    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  $[\varphi_O^{-1} \Psi op. op \leftarrow \pi] \in ?Sol_k$ 
  unfolding bounded-solution-set-sas-plus'-def
  using calculation(6, 7)
  by blast

moreover {
  have  $\forall op \in set \pi. op \in set ((\exists \Pi)_O)$ 
    using calculation(4)
  unfolding STRIPS-Semantics.is-serial-solution-for-problem-def list-all-iff
ListMem-iff
  by simp
  hence  $?_\varphi_P [\varphi_O^{-1} \Psi op. op \leftarrow \pi] = \pi$ 
    proof (induction  $\pi$ )
      case (Cons op  $\pi$ )
        moreover have  $?_\varphi_P [\varphi_O^{-1} \Psi op. op \leftarrow op \# \pi]$ 
           $= (\varphi_O \Psi (\varphi_O^{-1} \Psi op)) \# ?_\varphi_P [\varphi_O^{-1} \Psi op. op \leftarrow \pi]$ 
          by simp
        moreover have  $op \in set ((\exists \Pi)_O)$ 
          using Cons.prem
          by simp
        moreover have  $(\varphi_O \Psi (\varphi_O^{-1} \Psi op)) = op$ 
          using strips-operator-inverse-is[OF assms(1) calculation(4)].
        moreover have  $?_\varphi_P [\varphi_O^{-1} \Psi op. op \leftarrow \pi] = \pi$ 
          using Cons.IH Cons.prem
          by auto
        ultimately show ?case
          by argo
        qed simp
    }
  moreover have  $\pi \in ?_\varphi_P \setminus ?Sol_k$ 
    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+
      }
    }
  qed
```

```

moreover obtain op  $\pi'$  where  $\pi = op \# \pi'$ 
  using calculation (3)
  unfolding length-Suc-conv
  by fast
moreover have set  $\pi' \subseteq$  set ops and  $op \in$  set ops
  using calculation(2, 4)
  by simp+
moreover have length  $\pi' = k$ 
  using calculation(3, 4)
  by auto
moreover have  $\pi' \in ?P$ 
  using calculation(5, 7)
  by blast
ultimately have  $\pi \in ?P''$ 
  by blast
}
hence  $?P' \subseteq ?P''$ 
  by blast
}
ultimately show ?case
  using rev-finite-subset[of ?P'' ?P]
  by blast
qed force

```

— The set of fixed length SAS+ solutions are subsets of the set of plans with fixed length and therefore also finite.

```

private lemma sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a:
  assumes is-valid-problem-sas-plus  $\Psi$ 
  shows finite (bounded-solution-set-sas-plus'  $\Psi$  k)
proof -
  let  $?Ops = \text{set } ((\Psi)_{\mathcal{O}+})$ 
  let  $?Sol_k = \text{bounded-solution-set-sas-plus}' \Psi k$ 
  and  $?P_k = \{ \pi. \text{set } \pi \subseteq ?Ops \wedge \text{length } \pi = k \}$ 
{
  fix  $\psi$ 
  assume  $\psi \in ?Sol_k$ 
  then have length  $\psi = k$  and set  $\psi \subseteq ?Ops$ 
  unfolding bounded-solution-set-sas-plus'-def
    SAS-Plus-Semantics.is-serial-solution-for-problem-def Let-def list-all-iff 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' ($\varphi \Psi$) k)*
proof —
 let $\text{?}\Pi = \varphi \Psi$
 let $\text{?Ops} = \text{set}((\text{?}\Pi)_{\mathcal{O}})$
 let $\text{?Sol}_k = \text{bounded-solution-set-strips}' \text{?}\Pi k$
 and $\text{?P}_k = \{ \pi. \text{set } \pi \subseteq \text{?Ops} \wedge \text{length } \pi = k \}$
 {
 fix π
 assume $\pi \in \text{?Sol}_k$
 then have $\text{length } \pi = k$ **and** $\text{set } \pi \subseteq \text{?Ops}$
 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 $\pi \in \text{?P}_k$
 by *blast*
 }
 then have $\text{?Sol}_k \subseteq \text{?P}_k$
 by *force*
 thus *?thesis*
 using *bounded-plan-set-finite rev-finite-subset*[*of* $\text{?P}_k \text{?Sol}_k$]
 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 $\text{card}(\text{bounded-solution-set-sas-plus } \Psi N) = \text{card}(\text{bounded-solution-set-strips } (\varphi \Psi) N)$

proof –

let $\varPi = \varphi \Psi$
and $?R = \{0..N\}$

— Due to the disjoint nature of the bounded solution sets for fixed plan length for different lengths, we can sum the individual set cardinality to obtain the cardinality of the overall SAS+ resp. STRIPS solution sets.

have *finite-R*: *finite* $?R$
by *simp*
moreover {
 have $\forall k \in ?R. \text{finite}(\text{bounded-solution-set-sas-plus}' \Psi k)$
 using *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a*[OF
 assms(1)]..
 moreover have $\forall j \in ?R. \forall k \in ?R. j \neq k \rightarrow \text{bounded-solution-set-sas-plus}' \Psi j \cap \text{bounded-solution-set-sas-plus}' \Psi k = \{\}$
 unfold *bounded-solution-set-sas-plus'-def*
 by *blast*
 }
 ultimately have $\text{card}(\text{bounded-solution-set-sas-plus } \Psi N) = (\sum k \in ?R. \text{card}(\text{bounded-solution-set-sas-plus}' \Psi k))$
 using *card-UN-disjoint*
 by *blast*
 }
 moreover {
 have $\forall k \in ?R. \text{finite}(\text{bounded-solution-set-strips}' \varPi k)$
 using *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-b*[OF
 assms(1)]..
 moreover have $\forall j \in ?R. \forall k \in ?R. j \neq k \rightarrow \text{bounded-solution-set-strips}' \varPi j \cap \text{bounded-solution-set-strips}' \varPi k = \{\}$
 unfold *bounded-solution-set-strips'-def*
 by *blast*
 }
 ultimately have $\text{card}(\text{bounded-solution-set-strips } \varPi N) = (\sum k \in ?R. \text{card}(\text{bounded-solution-set-strips}' \varPi k))$
 using *card-UN-disjoint*
 by *blast*
 }
 moreover {
 fix k
 have $\text{card}(\text{bounded-solution-set-sas-plus}' \Psi k) = \text{card}((\lambda\psi. [\varphi_O \Psi \text{op. op} \leftarrow \psi]) \cdot \text{bounded-solution-set-sas-plus}' \Psi k)$

```

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'  $\Psi k$ )
    = card (bounded-solution-set-strips'  $\Pi k$ )
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 ( $\perp$ ) to prevent warnings.
hide-const (open) Orderings.bot-class.bot
no-notation Orderings.bot-class.bot ( $\langle \perp \rangle$ )

— Hide constant and notation for ((-+)) to prevent warnings.
hide-const (open) Transitive-Closure.trancl
unbundle no trancl-syntax

— Hide constant and notation for ((-+)) to prevent warnings.
hide-const (open) Relation.converse
no-notation Relation.converse ( $\langle \langle \text{notation}=\langle \text{postfix } -1 \rangle \rangle^{-1} \rangle$  [1000] 999)

```

7 The Basic SATPlan Encoding

We now move on to the formalization of the basic SATPlan encoding (see ??).

The two major results that we will obtain here are the soundness and completeness result outlined in ?? in ??.

Let in the following $\Phi \equiv \text{encode-to-sat } \Pi t$ 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 $A \equiv \text{encode-state-variable } 0 n b$ with $n \equiv \text{index vs } v$ and $b \equiv I v = \text{Some True}$ for all $v \in \mathcal{V}$. 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 $A \vee \perp$ 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 Λ(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
```

```

 $\equiv \text{let}$ 
 $vs = \text{variables-of } \Pi$ 
 $; G = \text{goal-of } \Pi$ 
 $\text{in } \bigwedge (\text{map } (\lambda v. \text{encode-state-variable } t (\text{index } vs v) (G v)) \vee \perp)$ 
 $(\text{filter } (\lambda v. G v \neq \text{None}) vs))$ 

```

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

$$\text{Atom} (\text{Operator } k (\text{index } ops op)) \rightarrow \text{Atom} (\text{State } k (\text{index } vs v))$$

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

```

definition encode-operator-precondition
:: 'variable strips-problem
⇒ nat
⇒ 'variable strips-operator
⇒ sat-plan-variable formula
where encode-operator-precondition  $\Pi t op \equiv \text{let}$ 
 $vs = \text{variables-of } \Pi$ 
 $; ops = \text{operators-of } \Pi$ 
 $\text{in } \bigwedge (\text{map } (\lambda v.$ 
 $\neg (\text{Atom} (\text{Operator } t (\text{index } ops op))) \vee \text{Atom} (\text{State } t (\text{index } vs v)))$ 
 $(\text{precondition-of } op))$ 

definition encode-all-operator-preconditions
:: 'variable strips-problem
⇒ 'variable strips-operator list
⇒ nat
⇒ sat-plan-variable formula
where encode-all-operator-preconditions  $\Pi ops t \equiv \text{let}$ 
 $l = \text{List.product } [0..<t] ops$ 
 $\text{in foldr } (\wedge) (\text{map } (\lambda(t, op). \text{encode-operator-precondition } \Pi t op) l) (\neg\perp)$ 

```

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.

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

for add effects respectively

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

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

```

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

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 (Λ) (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))
  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 and⁶ 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 evaluates the respective operator encoding at 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

```

⁶Not shown.

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

```

 $\equiv \text{let } ops = \text{operators-of } \Pi$ 
 $; vs = \text{map } (\lambda op. \text{ Operator } i (\text{index } ops \ op)) (\text{remdups } ops)$ 
 $\text{in map } (\lambda v. \text{ case } v \text{ of Operator } - k \Rightarrow ops ! k) (\text{filter } \mathcal{A} \ vs)$ 

```

— We decode maps over range $0, \dots, t - 1$ because the last operator takes effect in t and must therefore have been applied in step $t - 1$.

```

definition decode-plan
:: 'variable strips-problem
⇒ sat-plan-variable valuation
⇒ nat
⇒ 'variable strips-parallel-plan ( $\langle \Phi^{-1} \dashrightarrow 99 \rangle$ )
where decode-plan  $\Pi \mathcal{A} t \equiv \text{map } (\text{decode-plan}' \Pi \mathcal{A}) [0..<t]$ 

```

Similarly to the operator decoding, we can decode a state at time k from a valuation of the SATPlan encoding \mathcal{A} by constructing a map from list of assignments $(v, \mathcal{A} (\text{State } k (\text{index } vs \ v)))$ for all $v \in \mathcal{V}$.

```

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

```

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 ( $\langle \Phi_T \dashrightarrow 99 \rangle$ ) where
encode-transitions  $\Pi t$ 
 $\equiv \text{SAT-Plan-Base.encode-operators } \Pi t \wedge$ 
 $\text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t$ 

```

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

— Setup simp rules.

```

lemma [simp]:
encode-transitions  $\Pi t$ 
 $= \text{SAT-Plan-Base.encode-operators } \Pi t \wedge$ 
 $\text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t$ 
unfolding encode-problem-def encode-initial-state-def encode-transitions-def
 $\text{encode-goal-state-def decode-plan-def decode-state-at-def}$ 
by simp+

```

```

context
begin

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

lemma is-cnf-encode-initial-state:
  assumes is-valid-problem-strips  $\Pi$ 
  shows is-cnf ( $\Phi_I \Pi$ )
proof -
  let  $?I = (\Pi)_I$ 
  and  $?vs = \text{strips-problem.variables-of } \Pi$ 
  let  $?l = \text{map}(\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (?I v) \vee \perp)$ 
     $(\text{filter}(\lambda v. ?I v \neq \text{None}) ?vs)$ 
  {
    fix  $C$ 
    assume c-in-set-l: $C \in \text{set } ?l$ 
    have set  $?l = (\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (?I v) \vee \perp) `$ 
       $\text{set}(\text{filter}(\lambda v. ?I v \neq \text{None}) ?vs)$ 
      using set-map[of  $\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (?I v) \vee \perp$ ]
        filter( $\lambda v. ?I v \neq \text{None}$ ) ?vs]
      by blast
    then have set  $?l = (\lambda v. \text{encode-state-variable } 0 (\text{index } ?vs v) (?I v) \vee \perp) `$ 
      { $v \in \text{set } ?vs. ?I v \neq \text{None}$ }
      using set-filter[of  $\lambda v. ?I v \neq \text{None}$  ?vs]
      by argo
    then obtain  $v$ 
      where c-is:  $C = \text{encode-state-variable } 0 (\text{index } ?vs v) (?I v) \vee \perp$ 
      and v-in-set-vs:  $v \in \text{set } ?vs$ 
      and I-of-v-is-not-None:  $?I v \neq \text{None}$ 
      using c-in-set-l
      by auto
  {
    have  $v \in \text{dom } ?I$ 

```

```

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  $v \in \text{dom } ?G$ 
    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  $\perp$ 
    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  $\Pi k op$ )
proof -
  let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problemoperators-of  $\Pi$ 
  let ?l = map ( $\lambda v. \neg (\text{Atom} (\text{Operator } k (\text{index } ?ops op))) \vee \text{Atom} (\text{State } k (\text{index } ?vs v))$ )
    (precondition-of op)
  {
    have set ?l = ( $\lambda v. \neg (\text{Atom} (\text{Operator } k (\text{index } ?ops op))) \vee \text{Atom} (\text{State } k (\text{index } ?vs v))$ )
      ` set (precondition-of op)
      using set-map
      by force
    then have set ?l = {  $\neg (\text{Atom} (\text{Operator } k (\text{index } ?ops op))) \vee \text{Atom} (\text{State } k (\text{index } ?vs v))$  }
      | v.  $v \in \text{set} (\text{precondition-of } op)$ 
      using setcompr-eq-image[of
         $\lambda v. \neg (\text{Atom} (\text{Operator } k (\text{index } ?ops op))) \vee \text{Atom} (\text{State } k (\text{index } ?vs v))$ 
         $\lambda v. v \in \text{set} (\text{precondition-of } op)$ ]
      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]:

$$\text{set}(\text{map}(\lambda v. A v \vee B v) vs) = \{ A v \vee B v \mid v. v \in \text{set} vs \}$$

proof -
  let ?l =  $\text{map}(\lambda v. A v \vee B v) vs$ 
  have  $\text{set} ?l = (\lambda v. A v \vee B v) ` \text{set} vs$ 
    using set-map
    by force
  thus ?thesis
    using setcompr-eq-image
    by auto
qed

private lemma encode-operator-effects-is-cnf-i:

$$\begin{aligned} & \text{is-cnf}(\bigwedge(\text{map}(\lambda v. \neg(\text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op)))) \\ & \quad \vee \text{Atom}(\text{State}(Suc t) (\text{index}(\text{strips-problem.variables-of } \Pi) v))) (\text{add-effects-of } op))) \end{aligned}$$

proof -
  let ?fs =  $\text{map}(\lambda v. \neg(\text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op))))$ 
  
$$\vee \text{Atom}(\text{State}(Suc t) (\text{index}(\text{strips-problem.variables-of } \Pi) v))) (\text{add-effects-of } op)$$

  {
    fix C
    assume  $C \in \text{set} ?fs$ 
    then obtain v
      where  $v \in \text{set}(\text{add-effects-of } op)$ 
      and  $C = \neg(\text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op)))$ 
         $\vee \text{Atom}(\text{State}(Suc t) (\text{index}(\text{strips-problem.variables-of } \Pi) v))$ 
      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:

$$\text{is-cnf}(\bigwedge(\text{map}(\lambda v. \neg(\text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op))))$$


```

```

 $\vee \neg(\text{Atom}(\text{State}(\text{Suc } t)(\text{index}(\text{strips-problem.variables-of } \Pi) v))) (\text{delete-effects-of } op))$ 
proof –
  let ?fs = map ( $\lambda v. \neg(\text{Atom}(\text{Operator } t(\text{index}(\text{strips-problemoperators-of } \Pi) op)))$ 
   $\vee \neg(\text{Atom}(\text{State}(\text{Suc } t)(\text{index}(\text{strips-problem.variables-of } \Pi) v))) (\text{delete-effects-of } op)$ 
  {
    fix C
    assume C ∈ set ?fs
    then obtain v
      where v ∈ set (delete-effects-of op)
      and C =  $\neg(\text{Atom}(\text{Operator } t(\text{index}(\text{strips-problemoperators-of } \Pi) op)))$ 
       $\vee \neg(\text{Atom}(\text{State}(\text{Suc } t)(\text{index}(\text{strips-problem.variables-of } \Pi) v)))$ 
      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  $\Pi t op$ )
proof –
  let ?ops = strips-problemoperators-of  $\Pi$ 
  and ?vs = strips-problemvariables-of  $\Pi$ 
  let ?fs = map ( $\lambda v. \neg(\text{Atom}(\text{Operator } t(\text{index } ?ops op)))$ 
   $\vee \text{Atom}(\text{State}(\text{Suc } t)(\text{index } ?vs v)))$ 
  (add-effects-of op)
  and ?fs' = map ( $\lambda v. \neg(\text{Atom}(\text{Operator } t(\text{index } ?ops op)))$ 
   $\vee \neg(\text{Atom}(\text{State}(\text{Suc } t)(\text{index } ?vs v)))$ 
  (delete-effects-of op))
  have encode-operator-effect  $\Pi t op = \bigwedge (?fs @ ?fs')$ 
  unfolding encode-operator-effect-def[of  $\Pi t op$ ]
  by metis
  moreover {
    have  $\forall f \in \text{set } ?fs. \text{is-cnf } f \quad \forall f \in \text{set } ?fs'. \text{is-cnf } f$ 
    using encode-operator-effects-is-cnf-i[of  $t \Pi op$ ]
    encode-operator-effects-is-cnf-ii[of  $t \Pi op$ ]
    by (simp+)

    hence  $\forall f \in \text{set } (?fs @ ?fs'). \text{is-cnf } f$ 
    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 (λop. Atom (Operator t (index (strips-problem.operators-of Π) op)))
          (filter (λop. 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 (λop. ListMem v vs) ?ops)
    = { op ∈ set ?ops. ListMem v vs }
    using set-filter
    by force
    then have set (filter (λop. ListMem v vs) ?ops)
    = { op. op ∈ set ?ops ∧ v ∈ set vs }
    using ListMem-iff[of v]
    by blast
  }
  then have set (map (λop. Atom (Operator t (index ?ops op)))
          (filter (λop. ListMem v vs) ?ops))
= (λop. Atom (Operator t (index ?ops op))) ‘{ op ∈ set ?ops. v ∈ set vs }
  using set-map[of λop. Atom (Operator t (index ?ops op))]
  by presburger
  thus ?thesis
  by blast
qed

```

```

lemma is-disj-big-or-if:
  assumes ∀f ∈ set fs. is-lit-plus f
  shows is-disj ∨fs
  using assms
  proof (induction fs)
    case (Cons f fs)
    have is-lit-plus f
    using Cons.prem
    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 = ( $\neg(\text{Atom}(\text{State } t (\text{index } ?vs v)))$ )
and ?B =  $\text{Atom}(\text{State}(\text{Suc } t) (\text{index } ?vs v))$ 
{
  fix f
  assume f ∈ set ?fs

  then obtain op
  where op ∈ set ?ops
    and v ∈ set (delete-effects-of op)
    and f =  $\text{Atom}(\text{Operator } t (\text{index } ?ops op))$ 
    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 =  $\text{Atom}(\text{State } t (\text{index } ?vs v))$ 
  and ?B =  $\neg(\text{Atom}(\text{State}(\text{Suc } t) (\text{index } ?vs v)))$ 
  {
    fix f
    assume f ∈ set ?fs

    then obtain op
    where op ∈ set ?ops
      and v ∈ set (add-effects-of op)
      and f =  $\text{Atom}(\text{Operator } t (\text{index } ?ops op))$ 
      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  $\lambda(k, v).$  encode-positive-transition-frame-axiom
 $\Pi k v$ 
 $\lambda(k, v).$   $(k, v) \in (\{0..<t\} \times \text{set } ?vs)$ ]
by fast
ultimately show ?thesis
by argo
qed

```

```

lemma encode-frame-axioms-is-cnf:
shows is-cnf (encode-all-frame-axioms  $\Pi t$ )
proof -
let ?l = List.product [0..<t] (strips-problem.variables-of  $\Pi$ )
and ?vs = strips-problem.variables-of  $\Pi$ 
let ?A = { encode-negative-transition-frame-axiom  $\Pi k v$ 
|  $k v.$   $(k, v) \in (\{0..<t\} \times \text{set } ?vs)$  }
and ?B = { encode-positive-transition-frame-axiom  $\Pi k v$ 
|  $k v.$   $(k, v) \in (\{0..<t\} \times \text{set } ?vs)$  }
and ?fs = map ( $\lambda(k, v).$  encode-negative-transition-frame-axiom  $\Pi k v)$  ?l
@ (map ( $\lambda(k, v).$  encode-positive-transition-frame-axiom  $\Pi k v)$  ?l)
{
fix f
assume  $f \in \text{set } ?fs$ 

then consider ( $f$ -encodes-negative-frame-axiom)  $f \in ?A$ 
| ( $f$ -encodes-positive-frame-axiom)  $f \in ?B$ 
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  $\Pi$ 
shows is-cnf ( $\Phi \Pi t$ )
proof -
have is-cnf ( $\Phi_I \Pi$ )
using is-cnf-encode-initial-state assms
by auto
moreover have is-cnf (encode-goal-state  $\Pi t$ )
using encode-goal-state-is-cnf[OF assms]
by simp
moreover have is-cnf (encode-operators  $\Pi t \wedge$  encode-all-frame-axioms  $\Pi t$ )

```

```

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  $\mathcal{A} \models \text{SAT-Plan-Base.encode-problem } \Pi t$ 
shows  $\mathcal{A} \models \text{SAT-Plan-Base.encode-initial-state } \Pi$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-goal-state } \Pi t$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-operators } \Pi t$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t$ 
using assms
unfolding SAT-Plan-Base.encode-problem-def
by simp+

lemma cnf-of-encode-problem-structure:
shows cnf(SAT-Plan-Base.encode-initial-state  $\Pi$ )
 $\subseteq$  cnf(SAT-Plan-Base.encode-problem  $\Pi t$ )
and cnf(SAT-Plan-Base.encode-goal-state  $\Pi t$ )
 $\subseteq$  cnf(SAT-Plan-Base.encode-problem  $\Pi t$ )
and cnf(SAT-Plan-Base.encode-operators  $\Pi t$ )
 $\subseteq$  cnf(SAT-Plan-Base.encode-problem  $\Pi t$ )
and cnf(SAT-Plan-Base.encode-all-frame-axioms  $\Pi t$ )
 $\subseteq$  cnf(SAT-Plan-Base.encode-problem  $\Pi t$ )
unfolding SAT-Plan-Base.encode-problem-def
SAT-Plan-Base.encode-problem-def[of  $\Pi t$ ] SAT-Plan-Base.encode-initial-state-def[of
 $\Pi$ ]
SAT-Plan-Base.encode-goal-state-def[of  $\Pi t$ ] SAT-Plan-Base.encode-operators-def
SAT-Plan-Base.encode-all-frame-axioms-def[of  $\Pi t$ ]
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( $\Phi_I \Pi$ ) =  $\bigcup \{ \text{cnf}(\text{encode-state-variable } 0$ 
 $(\text{index}(\text{strips-problem.variables-of } \Pi) v) ((\Pi)_I v))$ 
 $| v. v \in \text{set}(\text{strips-problem.variables-of } \Pi) \wedge ((\Pi)_I v \neq \text{None} ) \}$ 
proof —
let ?vs = strips-problem.variables-of  $\Pi$ 
and ?I = strips-problem.initial-of  $\Pi$ 

```

```

let ?ls = map (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥)
  (filter (λv. ?I v ≠ None) ?vs)
{
  have cnf ` set ?ls = cnf ` (λv. encode-state-variable 0 (index ?vs v) (?I v) ∨
⊥)
    ` set (filter (λv. ?I v ≠ None) ?vs)
  using set-map[of λv. encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥]
  by presburger
also have ... = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v) ∨ ⊥))
  ` set (filter (λv. ?I v ≠ None) ?vs)
  using image-comp
  by blast
also have ... = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v)))
  ` { v ∈ set ?vs. ?I v ≠ None }
  using set-filter[of λv. ?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 λv. cnf (encode-state-variable 0 (index ?vs v) (?I
v))]
  by presburger
}
moreover have cnf (ΦI Π) = ∪ (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 (ΦI Π) = (∪ v ∈ 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 nb1: { v. v ∈ set ?vs ∧ ?I v ≠ None } = set ?vs
    using is-valid-problem-strips-initial-of-dom assms(1)
    by auto

  {
    fix v
    assume v ∈ set ?vs
    then have ?I v ≠ None
  }

```

```

using is-valid-problem-strips-initial-of-dom assms(1)
by auto
then consider (I-v-is-Some-True) ?I v = Some True
| (I-v-is-Some-False) ?I v = Some False
by fastforce
hence cnf (encode-state-variable 0 (index ?vs v) (?I v))
= {{ literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?I v)) } }
unfolding encode-state-variable-def
by (cases, simp+)
} note nb2 = this
{
  have { cnf (encode-state-variable 0 (index ?vs v) (?I v)) | v. v ∈ set ?vs ∧ ?I
v ≠ None }
    = (λv. cnf (encode-state-variable 0 (index ?vs v) (?I v))) ` set ?vs
  using setcompr-eq-image[of λv. cnf (encode-state-variable 0 (index ?vs v) (?I
v))]
    λv. v ∈ set ?vs ∧ ?I v ≠ None] using nb1
  by presburger
  hence { cnf (encode-state-variable 0 (index ?vs v) (?I v)) | v. v ∈ set ?vs ∧ ?I
v ≠ None }
    = (λv. {{ literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?I
v)) }})
      ` set ?vs
  using nb2
  by force
}
thus ?thesis
using cnf-of-encode-initial-state-set-i
by (smt Collect-cong)
qed

```

```

lemma 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 (ΦI Π)
  ∧ C = { (State 0 (index (strips-problem.variables-of Π) v))+ })
and strips-problem.initial-of Π v = Some False → (∃!C. C ∈ cnf (ΦI Π)
  ∧ C = { (State 0 (index (strips-problem.variables-of Π) v))-1 })
proof -
let ?I = (Π)I
let ?vs = strips-problem.variables-of Π
let ?ΦI = ΦI Π
have nb1: cnf (ΦI Π) = ∪ { 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  $v \in \{ v. v \in \text{set } ?vs \wedge ?I v \neq \text{None} \}$ 
  using assms(2)
  by auto
} note nb2 = this
show strips-problem.initial-of  $\Pi$   $v = \text{Some True} \longrightarrow (\exists!C. C \in \text{cnf } (\Phi_I \Pi)$ 
 $\wedge C = \{ (\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \})$ 
and strips-problem.initial-of  $\Pi$   $v = \text{Some False} \longrightarrow (\exists!C. C \in \text{cnf } (\Phi_I \Pi)$ 
 $\wedge C = \{ (\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \})$ 
proof (auto)
  assume i-v-is-some-true: strips-problem.initial-of  $\Pi$   $v = \text{Some True}$ 
  then have {  $(\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$  }
     $\in \text{cnf } (\text{encode-state-variable } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v) (?I$ 
 $v))$ 
    unfolding encode-state-variable-def
    using i-v-is-some-true
    by auto
  thus {  $(\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$  }
     $\in \text{cnf } (\Phi_I \Pi)$ 
    using nb1 nb2
    by auto
next
  assume i-v-is-some-false: strips-problem.initial-of  $\Pi$   $v = \text{Some False}$ 
  then have {  $(\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1}$  }
     $\in \text{cnf } (\text{encode-state-variable } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v) (?I$ 
 $v))$ 
    unfolding encode-state-variable-def
    using i-v-is-some-false
    by auto
  thus {  $(\text{State } 0 (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1}$  }
     $\in \text{cnf } (\Phi_I \Pi)$ 
    using nb1 nb2
    by auto
qed
qed

```

lemma cnf-of-operator-encoding-structure:

$$\begin{aligned} \text{cnf } (\text{encode-operators } \Pi t) &= \text{cnf } (\text{encode-all-operator-preconditions } \Pi \\ &\quad (\text{strips-problemoperators-of } \Pi) t) \\ &\cup \text{cnf } (\text{encode-all-operator-effects } \Pi (\text{strips-problemoperators-of } \Pi) t) \end{aligned}$$

unfolding encode-operators-def
 using cnf.simps(5)
 by metis

corollary cnf-of-operator-precondition-encoding-subset-encoding:

$$\begin{aligned} \text{cnf } (\text{encode-all-operator-preconditions } \Pi (\text{strips-problemoperators-of } \Pi) t) \\ \subseteq \text{cnf } (\Phi \Pi t) \end{aligned}$$

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 ( $\wedge$ ) fs ( $\neg\perp$ )) = ( $\bigcup f \in \text{set } fs. \text{cnf } f$ )
proof (induction fs)
  case (Cons f fs)
    have ih: cnf (foldr ( $\wedge$ ) fs ( $\neg\perp$ )) = ( $\bigcup f \in \text{set } fs. \text{cnf } f$ )
      using Cons.IH
      by blast
    {
      have cnf (foldr ( $\wedge$ ) (f # fs) ( $\neg\perp$ )) = cnf (f  $\wedge$  foldr ( $\wedge$ ) fs ( $\neg\perp$ ))
        by simp
      also have ... = cnf f  $\cup$  cnf (foldr ( $\wedge$ ) fs ( $\neg\perp$ ))
        by force
      finally have cnf (foldr ( $\wedge$ ) (f # fs) ( $\neg\perp$ )) = cnf f  $\cup$  ( $\bigcup f \in \text{set } fs. \text{cnf } f$ )
        using ih
        by argo
    }
    thus ?case
      by auto
  qed simp

```

```

private lemma cnf-of-encode-operator-precondition[simp]:
  cnf (encode-operator-precondition  $\Pi$  t op) = ( $\bigcup v \in \text{set} (\text{precondition-of } op)$ .
     $\{\{(Operator t (\text{index} (\text{strips-problem.operators-of } \Pi) op))^{-1}$ 
    , (State t (\text{index} (\text{strips-problem.variables-of } \Pi) v))^+\}\})
proof -
  let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problem.operators-of  $\Pi$ 
  and ? $\Phi_P$  = encode-operator-precondition  $\Pi$  t op
  let ?fs = map ( $\lambda v. \neg (\text{Atom} (\text{Operator } t (\text{index } ?ops op))) \vee \text{Atom} (\text{State } t (\text{index } ?vs v))$ )
    (precondition-of op)
    and ?A = ( $\lambda v. \neg (\text{Atom} (\text{Operator } t (\text{index } ?ops op))) \vee \text{Atom} (\text{State } t (\text{index } ?vs v))$ )
      ' set (precondition-of op)
  have cnf (encode-operator-precondition  $\Pi$  t op) = cnf ( $\bigwedge ?fs$ )
    unfolding encode-operator-precondition-def
    by presburger
  also have ... =  $\bigcup (\text{cnf}' \text{ set } ?fs)$ 
    using cnf-BigAnd
    by blast
  also have ... =  $\bigcup (\text{cnf}' ?A)$ 
    using set-map[of  $\lambda v. \neg (\text{Atom} (\text{Operator } t (\text{index } ?ops op))) \vee \text{Atom} (\text{State } t (\text{index } ?vs v))$ 
      precondition-of op]

```

```

by argo
also have ... = ( $\bigcup v \in \text{set}(\text{precondition-of } op)$ .
   $\text{cnf}(\neg(\text{Atom}(\text{Operator } t (\text{index } ?ops \ op))) \vee \text{Atom}(\text{State } t (\text{index } ?vs \ v)))$ )
by blast

finally show ?thesis
by auto
qed

lemma cnf-of-encode-all-operator-preconditions-structure[simp]:
   $\text{cnf}(\text{encode-all-operator-preconditions } \Pi (\text{strips-problem.operators-of } \Pi) \ t)$ 
  = ( $\bigcup(t, op) \in (\{.. < t\} \times \text{set}(\text{operators-of } \Pi))$ .
    ( $\bigcup v \in \text{set}(\text{precondition-of } op)$ .
       $\{\{(\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))^{-1}$ 
        ,  $(\text{State } t (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+\}\}))$ )
proof –
  let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problem.operators-of  $\Pi$ 
  let ?l = List.product [0..<t] ?ops
  and ? $\Phi_P$  = encode-all-operator-preconditions  $\Pi (\text{strips-problem.operators-of } \Pi)$ 
  t
  let ?A = set (map ( $\lambda(t, op).$  encode-operator-precondition  $\Pi t op$ ) ?l)
  {
    have set ?l = {0..<t}  $\times$  set (( $\Pi$ ) $_{\mathcal{O}}$ )
    by auto
    then have ?A = ( $\lambda(t, op).$  encode-operator-precondition  $\Pi t op$ ) ‘ ({0..<t}  $\times$ 
      set (( $\Pi$ ) $_{\mathcal{O}}$ ))
    using set-map
    by force
  } note nb = this
  have cnf ? $\Phi_P$  = cnf (foldr ( $\wedge$ ) (map ( $\lambda(t, op).$  encode-operator-precondition  $\Pi t op$ ) ?l) ( $\neg\perp$ ))
    unfolding encode-all-operator-preconditions-def
    by presburger
  also have ... = ( $\bigcup f \in ?A.$  cnf  $f$ )
    by simp

  also have ... = ( $\bigcup(k, op) \in (\{0..<t\} \times \text{set}((\Pi)_{\mathcal{O}}))$ .
     $\text{cnf}(\text{encode-operator-precondition } \Pi k op))$ )
    using nb
    by fastforce

finally show ?thesis
by fastforce
qed

corollary cnf-of-encode-all-operator-preconditions-contains-clause-if:
  fixes  $\Pi$ ::'variable STRIPS-Representation.strips-problem

```

```

assumes is-valid-problem-strips ( $\Pi ::= \text{variable STRIPS-Representation.strips-problem}$ )
  and  $k < t$ 
  and  $op \in \text{set } ((\Pi)_O)$ 
  and  $v \in \text{set } (\text{precondition-of } op)$ 
shows {  $(\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$ 
  ,  $(\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$  }
 $\in \text{cnf} (\text{encode-all-operator-preconditions } \Pi (\text{strips-problem.operators-of } \Pi) t)$ 
proof -
  let ?ops = strips-problem.operators-of  $\Pi$ 
  and ?vs = strips-problem.variables-of  $\Pi$ 
  let ? $\Phi_P$  = encode-all-operator-preconditions  $\Pi$  ?ops t
  and ?C = {  $(\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$ 
    ,  $(\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$  }
  {
    have nb:  $(k, op) \in \{\dots < t\} \times \text{set } ((\Pi)_O)$ 
    using assms(2, 3)
    by blast
    moreover {
      have ?C  $\in (\bigcup_{v \in \text{set } (\text{precondition-of } op)}$ .
        { $\{(\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1},$ 
           $(\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+\}\}}$ 
        using UN-iff[where A=set (precondition-of op)
        and B= $\lambda v. \{(\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1},$ 
           $(\text{State } t (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+\}\}] assms(4)
        by blast
      hence  $\exists x \in \{\dots < t\} \times \text{set } ((\Pi)_O).$ 
        ?C  $\in (\text{case } x \text{ of } (k, op) \Rightarrow \bigcup_{v \in \text{set } (\text{precondition-of } op)}$ .
        { $\{(\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1},$ 
           $(\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+\}\}}$ 
        using nb
        by blast
    }
    ultimately have ?C  $\in (\bigcup (t, op) \in (\{\dots < t\} \times \text{set } ((\Pi)_O))$ .
       $(\bigcup_{v \in \text{set } (\text{precondition-of } op)}$ .
        { $\{(\text{Operator } t (\text{index } ?ops op))^{-1}, (\text{State } t (\text{index } ?vs v))^+\}\})$ 
        by blast
    }
    thus ?thesis
    using cnf-of-encode-all-operator-preconditions-structure[of  $\Pi t$ ]
    by argo
  qed$ 
```

corollary cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem:
 $\text{cnf} (\text{encode-all-operator-effects } \Pi (\text{strips-problem.operators-of } \Pi) t)$
 $\subseteq \text{cnf} (\Phi \Pi t)$
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 ?fs1 = 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))
  and ?fs2 = 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))
  {
    have cnf ‘ set ?fs1 = cnf
    ‘ (λv. ¬(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 ... = (λv. 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 ?fs1 = (λv. { { (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 nb1 = this
  {
    have cnf ‘ set ?fs2 = cnf ‘ (λv. ¬(Atom (Operator t (index (strips-problem.operators-of
Π) op)))
    ∨ ¬(Atom (State (Suc t) (index (strips-problem.variables-of Π) v))))
    ‘ set (delete-effects-of op)
    using set-map
    by force
    also have ... = (λv. cnf (¬(Atom (Operator t (index (strips-problem.operators-of
Π) op)))
    ∨ ¬(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 ?fs2 = (λv. { { (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 nb2 = this
{
have cnf (encode-operator-effect Π t op) = ∪(cnf ` set (?fs1 @ ?fs2))
unfolding encode-operator-effect-def
using cnf-BigAnd[of ?fs1 @ ?fs2]
by meson
also have ... = ∪(cnf ` set ?fs1 ∪ cnf ` set ?fs2)
using set-append[of ?fs1 ?fs2] image-Un[of cnf set ?fs1 set ?fs2]
by argo
also have ... = ∪(cnf ` set ?fs1) ∪ ∪(cnf ` set ?fs2)
using Union-Un-distrib[of cnf ` set ?fs1 cnf ` set ?fs2]
by argo

finally have 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 } })
using nb1 nb2
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)
= (∪(k, op) ∈ ({0..<t} × set ((Π)ο)).
(∪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))+ } }))
∪ (∪(k, op) ∈ ({0..<t} × set ((Π)ο)).
(∪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 ?ΦE = 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).
    (UN 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))⁺ }})
    ∪ (UN 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)₀)
  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 = UN (k, op) ∈ {0..<t} × set ((Π)₀).
    ∪ 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 λv. {{ (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..} × set ((Π)o))
      using assms(2, 3)
      by fastforce

    ultimately have ?C ∈ ?Add
      by blast
  }
  ultimately show ?thesis
    using subset-eq[of ?Add cnf ?ΦE]
    by meson
qed

corollary cnf-of-operator-effect-encoding-contains-delete-effect-clause-if:
  fixes Π:: 'a strips-problem
  assumes is-valid-problem-strips Π
  and k < t
  and op ∈ set ((Π)o)
  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 ?ΦE = 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..} × set ((Π)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 ?ΦE
    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  $(k, op) \in \{0..t\} \times set ?ops$ 
using assms( $2, 3$ )
by force

ultimately have  $?C \in ?Delete$ 
  by fastforce
}

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

private lemma cnf-of-big-or-of-literal-formulas-is[simp]:
assumes  $\forall f \in set fs. is-literal-formula f$ 
shows  $cnf (\bigvee fs) = \{\{ literal-formula-to-literal f \mid f. f \in set fs\}\}$ 
using assms
proof (induction fs)
case (Cons f fs)
{
  have is-literal-formula-f: is-literal-formula f
    using Cons.preds(1)
    by simp
  then have  $cnf f = \{\{ literal-formula-to-literal f \}\}$ 
    using cnf-of-literal-formula
    by blast
} note  $nb_1 = this$ 
{
  have  $\forall f' \in set fs. is-literal-formula f'$ 
    using Cons.preds
    by fastforce
  hence  $cnf (\bigvee fs) = \{\{ literal-formula-to-literal f \mid f. f \in set fs\}\}$ 
    using Cons.IH
    by arg0
} note  $nb_2 = this$ 
{
  have  $cnf (\bigvee(f \# fs)) = (\lambda(g, h). g \cup h)$ 
    ‘  $(\{\{ literal-formula-to-literal f\}\} \times \{\{ literal-formula-to-literal f' \mid f'. f' \in set fs\}\})$ 
    using  $nb_1 nb_2$ 
    by simp
  also have ... =  $\{\{ literal-formula-to-literal f\} \cup \{\{ literal-formula-to-literal f' \mid f'. f' \in set fs\}\}$ 
    by fast
  finally have  $cnf (\bigvee(f \# fs)) = \{\{ literal-formula-to-literal f' \mid f'. f' \in set (f \# fs)\}\}$ 
    by fastforce
}

```

```

thus ?case .
qed simp

private lemma set-filter-op-list-mem-vs[simp]:
  set (filter ( $\lambda op. ListMem v vs$ ) ops) = {  $op. op \in set ops \wedge v \in set vs$  }
  using set-filter[of  $\lambda op. ListMem v vs$ ] ListMem-iff
  by force

private lemma cnf-of-positive-transition-frame-axiom:
  cnf (encode-positive-transition-frame-axiom  $\Pi k v$ )
  = { { (State  $k$  (index (strips-problem.variables-of  $\Pi$ )  $v$ ))+
    , (State ( $Suc k$ ) (index (strips-problem.variables-of  $\Pi$ )  $v$ ))-1 } }
     $\cup$  { (Operator  $k$  (index (strips-problemoperators-of  $\Pi$ )  $op$ ))+
      |  $op. op \in set (\text{strips-problemoperators-of } \Pi) \wedge v \in set (\text{add-effects-of } op)$  }
  }

proof -
  let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problemoperators-of  $\Pi$ 
  let ?adding-operators = filter ( $\lambda op. ListMem v (\text{add-effects-of } op)$ ) ?ops
  let ?fs = map ( $\lambda op. Atom (\text{Operator } k (\text{index } ?ops op))$ ) ?adding-operators
  {
    have set ?fs = ( $\lambda op. Atom (\text{Operator } k (\text{index } ?ops op))$ ) ‘ set ?adding-operators
    using set-map[of  $\lambda op. Atom (\text{Operator } k (\text{index } ?ops op))$ ] ?adding-operators
    by blast

    then have literal-formula-to-literal ‘ set ?fs
    = ( $\lambda op. (\text{Operator } k (\text{index } ?ops op))^{+}$ ) ‘ set ?adding-operators
    using image-comp[of  $\lambda op. Atom (\text{Operator } k (\text{index } ?ops op))$ ] ?adding-operators
    by simp

    also have ... = ( $\lambda op. (\text{Operator } k (\text{index } ?ops op))^{+}$ 
      ‘ {  $op. op \in set ?ops \wedge v \in set (\text{add-effects-of } op)$  })
    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))+ |  $op. op \in set ?ops \wedge v \in set (\text{add-effects-of } op)$  }
    using setcompr-eq-image[of  $\lambda op. (\text{Operator } k (\text{index } ?ops op))^{+}$ 
       $\lambda op. op \in set ?adding-operators$ ]
    by blast

    hence cnf ( $\bigvee ?fs$ ) = { { (Operator  $k$  (index ?ops op))+
      |  $op. op \in set ?ops \wedge v \in set (\text{add-effects-of } op)$  } }
    using cnf-of-big-or-of-literal-formulas-is[of ?fs]
      setcompr-eq-image[of literal-formula-to-literal  $\lambda f. f \in set ?fs$ ]
    by force
  }

```

```

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

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

moreover have cnf (encode-positive-transition-frame-axiom  $\Pi k v$ )
= cnf (( $\text{Atom} (\text{State } k (\text{index } ?vs v)) \vee (\neg(\text{Atom} (\text{State} (\text{Suc } k) (\text{index } ?vs v))) \vee \bigvee ?fs)$ ))
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  $\Pi k v$ )
=  $\{\{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} , (\text{State} (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \} \cup \{ (\text{Operator } k (\text{index } (\text{strips-problemoperators-of } \Pi) op))^+ | op. op \in \text{set } (\text{strips-problemoperators-of } \Pi) \wedge v \in \text{set } (\text{delete-effects-of } op) \}\}$ 
proof –
let ?vs = strips-problem.variables-of  $\Pi$ 
and ?ops = strips-problemoperators-of  $\Pi$ 
let ?deleting-operators = filter ( $\lambda op. \text{ListMem } v (\text{delete-effects-of } op)$ ) ?ops
let ?fs = map ( $\lambda op. \text{Atom} (\text{Operator } k (\text{index } ?ops op))$ ) ?deleting-operators
{
have set ?fs = ( $\lambda op. \text{Atom} (\text{Operator } k (\text{index } ?ops op))$ ) ` set ?deleting-operators
using set-map[of  $\lambda op. \text{Atom} (\text{Operator } k (\text{index } ?ops op))$ ] ?deleting-operators]
by blast

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

```

```

using set-filter-op-list-mem-vs[of v - ?ops]
by auto

finally have literal-formula-to-literal ` set ?fs
= { (Operator k (index ?ops op))^+ | op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of
op) }
using setcompr-eq-image[of λop. (Operator k (index ?ops op))^+
λop. op ∈ set ?deleting-operators]
by blast

hence cnf (∨ ?fs) = { { (Operator k (index ?ops op))^+
| op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) } }
using cnf-of-big-or-of-literal-formulas-is[of ?fs]
setcompr-eq-image[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))^+ } ∪ { (Operator k (index ?ops op))^+
| op. op ∈ set ?ops ∧ v ∈ set (delete-effects-of op) } }
by force

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

moreover have cnf (encode-negative-transition-frame-axiom Π k v)
= cnf ((¬(Atom (State k (index ?vs v))) ∨ (Atom (State (Suc k) (index ?vs
v)) ∨ ∨ ?fs)))
unfoldng 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)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)ο) ∧ v ∈ set (add-effects-of op) } } }
∪ ∪( ∪(k, v) ∈ ({0..<t} × set ((Π)v)).
{ { { (State k (index (strips-problem.variables-of Π) v))^-1

```

```

, (State (Suc k) (index (strips-problem.variables-of  $\Pi$ ) v))+ }
 $\cup \{ (\text{Operator } k (\text{index} (\text{strips-problem.operators-of } \Pi) op))^{+}$ 
| op. op  $\in$  set  $((\Pi)_O)$   $\wedge$  v  $\in$  set (delete-effects-of op) \{\}\})
proof -
let ?vs = strips-problem.variables-of  $\Pi$ 
and ?ops = strips-problem.operators-of  $\Pi$ 
and ? $\Phi_F$  = encode-all-frame-axioms  $\Pi$  t
let ?l = List.product [0..<t] ?vs
let ?fs = map ( $\lambda(k, v).$  encode-negative-transition-frame-axiom  $\Pi k v)$  ?l
@ map ( $\lambda(k, v).$  encode-positive-transition-frame-axiom  $\Pi k v)$  ?l
{
let ?A = { encode-negative-transition-frame-axiom  $\Pi k v$ 
| k v. (k, v)  $\in$  ({0..<t}  $\times$  set  $((\Pi)_V)$ ) }
and ?B = { encode-positive-transition-frame-axiom  $\Pi k v$ 
| k v. (k, v)  $\in$  ({0..<t}  $\times$  set  $((\Pi)_V)$ ) }
have set-l: set ?l = {<t}  $\times$  set  $((\Pi)_V)$ 
using set-product
by force

have set ?fs = ?A  $\cup$  ?B
unfolding set-append set-map
using encode-all-frame-axioms-set
by force
then have cnf ` set ?fs = cnf ` ?A  $\cup$  cnf ` ?B
using image-Un[of cnf ?A ?B]
by argo
moreover {
have ?A = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .
{ encode-negative-transition-frame-axiom  $\Pi k v$  })
by blast
then have cnf ` ?A = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .
{ cnf (encode-negative-transition-frame-axiom  $\Pi k v$  ) })
by blast
hence cnf ` ?A = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .
{{ (State k (index ?vs v))-1
, (State (Suc k) (index ?vs v))+ }
 $\cup \{ (\text{Operator } k (\text{index } ?ops op))^{+}$ 
| op. op  $\in$  set ?ops  $\wedge$  v  $\in$  set (delete-effects-of op)\}\}})
using cnf-of-negative-transition-frame-axiom[of  $\Pi$ ]
by presburger
}
moreover {
have ?B = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .
{ encode-positive-transition-frame-axiom  $\Pi k v$  })
by blast
then have cnf ` ?B = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .
{ cnf (encode-positive-transition-frame-axiom  $\Pi k v$  ) })
by blast
hence cnf ` ?B = ( $\bigcup (k, v) \in ({0..<t} \times \text{set } ((\Pi)_V))$ .

```

```

    \{\{ (State k (index ?vs v))+
        , (State (Suc k) (index ?vs v))-1 \}
    \cup \{(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
= (U(k, v) ∈ ({0..<t} × set ((Π)v)). 
  \{\{ (State k (index ?vs v))+
    , (State (Suc k) (index ?vs v))-1 \}
  \cup \{(Operator k (index ?ops op))+
    | op. op ∈ set ((Π)o) ∧ v ∈ set (add-effects-of op) \}\}\}
  ∪ (U(k, v) ∈ ({0..<t} × set ((Π)v)). 
    \{\{ (State k (index ?vs v))-1
      , (State (Suc k) (index ?vs v))+ \}
    \cup \{(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)
= U((U(k, v) ∈ ({0..<t} × set ((Π)v)). 
  \{\{ (State k (index ?vs v))+
    , (State (Suc k) (index ?vs v))-1 \}
  \cup \{(Operator k (index ?ops op))+
    | op. op ∈ set ((Π)o) ∧ v ∈ set (add-effects-of op) \}\}\}
  ∪ (U(k, v) ∈ ({0..<t} × set ((Π)v)). 
    \{\{ (State k (index ?vs v))-1
      , (State (Suc k) (index ?vs v))+ \}
    \cup \{(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
  (U(k, v) ∈ ({0..<t} × set ((Π)v)). 
    \{\{ (State k (index ?vs v))+
      , (State (Suc k) (index ?vs v))-1 \}
    \cup \{(Operator k (index ?ops op))+
      | op. op ∈ set ((Π)o) ∧ v ∈ set (add-effects-of op) \}\}\}
    (U(k, v) ∈ ({0..<t} × set ((Π)v)). 
      \{\{ (State k (index ?vs v))-1
        , (State (Suc k) (index ?vs v))+ \}
      \cup \{(Operator k (index ?ops op))+
        | op. op ∈ set ((Π)o) ∧ v ∈ set (delete-effects-of op)\}\}\)])
  by argo

```

qed

— A technical lemma used in .

private lemma *cnf-of-encode-goal-state-set-i*:

$$\text{cnf } ((\Phi_G \Pi) t) = \bigcup (\{ \text{cnf} (\text{encode-state-variable } t \\ (\text{index } (\text{strips-problem.variables-of } \Pi) v) (((\Pi)_G) v)) \\ | v. v \in \text{set } ((\Pi)_V) \wedge ((\Pi)_G) v \neq \text{None} \})$$

proof —

```

let ?vs = strips-problem.variables-of  $\Pi$ 
and ?G =  $(\Pi)_G$ 
and ? $\Phi_G$  =  $(\Phi_G \Pi)$   $t$ 
let ?fs = map ( $\lambda v.$  encode-state-variable  $t$  (index ?vs  $v$ ) (?G  $v$ )  $\vee \perp$ )
          (filter ( $\lambda v.$  ?G  $v \neq \text{None}$ ) ?vs)
{
  have cnf ‘ set ?fs = cnf ‘ ( $\lambda v.$  encode-state-variable  $t$  (index ?vs  $v$ ) (?G  $v$ )  $\vee$ 
 $\perp$ )
    ‘ {  $v | v. v \in \text{set } ?vs \wedge ?G v \neq \text{None}$  }
  unfolding set-map
  by force
  also have ... = ( $\lambda v.$  cnf (encode-state-variable  $t$  (index ?vs  $v$ ) (?G  $v$ )  $\vee \perp$ ))
    ‘ {  $v | v. v \in \text{set } ?vs \wedge ?G v \neq \text{None}$  }
  using image-comp[of cnf ( $\lambda v.$  encode-state-variable  $t$  (index ?vs  $v$ ) (?G  $v$ )  $\vee$ 
 $\perp$ )
    ‘ {  $v | v. v \in \text{set } ?vs \wedge ?G v \neq \text{None}$  }]
  by fast
  finally have cnf ‘ set ?fs = { cnf (encode-state-variable  $t$  (index ?vs  $v$ ) (?G  $v$ ))
    |  $v. v \in \text{set } ?vs \wedge ?G v \neq \text{None}$  }
  unfolding setcompr-eq-image[of  $\lambda v.$  cnf (encode-state-variable  $t$  (index ?vs
 $v$ ) (?G  $v$ )  $\vee \perp$ )]
  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) (((Π)_G) v)) }}}
 | $v. v \in \text{set } ((\Pi)_V) \wedge ((\Pi)_G) v \neq \text{None} \})$

proof —

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

```

and ?G = ( $\Pi$ )G
and ? $\Phi_G$  = ( $\Phi_G$   $\Pi$ ) t
{
  fix v
  assume v ∈ { v | v. v ∈ set (( $\Pi$ )V) ∧ ?G v ≠ None }
  then have v ∈ set (( $\Pi$ )V) and G-of-v-is-not-None: ?G v ≠ None
    by fast+
  then consider (A) ?G v = Some True
    | (B) ?G v = Some False
      by fastforce
    hence cnf (encode-state-variable t (index ?vs v) (?G v))
      = {{ literal-formula-to-literal (encode-state-variable t (index ?vs v) (?G v)) }
  })
  unfolding encode-state-variable-def
  by (cases, force+)
} note nb = this
have cnf ? $\Phi_G$  =  $\bigcup$  ({ cnf (encode-state-variable t (index ?vs v) (?G v))
  | v. v ∈ set (( $\Pi$ )V) ∧ ?G v ≠ 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 | v. v ∈ set (( $\Pi$ )V) ∧ (( $\Pi$ )G) v ≠ None })
  using setcompr-eq-image[of
     $\lambda$ v. cnf (encode-state-variable t (index ?vs v) ((( $\Pi$ )G) v))
    λv. v ∈ set (( $\Pi$ )V) ∧ (( $\Pi$ )G) v ≠ None]
  by presburger
also have ... =  $\bigcup$  (( $\lambda$ v. {{ literal-formula-to-literal
    (encode-state-variable t (index ?vs v) (?G v)) }})
  ‘ { v. v ∈ set (( $\Pi$ )V) ∧ (( $\Pi$ )G) v ≠ 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  $\Pi$ :: 'a strips-problem
  assumes is-valid-problem-strips  $\Pi$ 
  and v ∈ dom (( $\Pi$ )G)
  shows (( $\Pi$ )G) v = Some True  $\longrightarrow$  ( $\exists$ !C. C ∈ cnf (( $\Phi_G$   $\Pi$ ) t)
    ∧ C = { (State t (index (strips-problem.variables-of  $\Pi$ ) v))+ })
  and (( $\Pi$ )G) v = Some False  $\longrightarrow$  ( $\exists$ !C. C ∈ cnf (( $\Phi_G$   $\Pi$ ) t)
    ∧ C = { (State t (index (strips-problem.variables-of  $\Pi$ ) v))-1 })
proof –

```

```

let ?vs = strips-problem.variables-of Π
  and ?G = (Π)G
  and ?ΦG = (ΦG Π) t
have nb1: cnf ?ΦG = ∪ { cnf (encode-state-variable t (index ?vs v)
  (?G v)) | v. v ∈ set ((Π)V) ∧ ?G v ≠ None }
  unfolding cnf-of-encode-goal-state-set-i
  by auto
have nb2: v ∈ { v. v ∈ set ((Π)V) ∧ ?G v ≠ None }
  using is-valid-problem-dom-of-goal-state-is assms(1, 2)
  by auto
have nb3: cnf (encode-state-variable t (index (strips-problem.variables-of Π) v)
((Π)G) v)
  ⊆ (∪ { cnf (encode-state-variable t (index ?vs v)
  (?G v)) | v. v ∈ set ((Π)V) ∧ ?G v ≠ None })
  using UN-upper[OF nb2, of λv. cnf (encode-state-variable t (index ?vs v) (?G
v))] nb2
  by blast
show ((Π)G) v = Some True → (∃!C. C ∈ cnf ((ΦG Π) t)
  ∧ C = { (State t (index (strips-problem.variables-of Π) v))+ })
  and ((Π)G) v = Some False → (∃!C. C ∈ cnf ((ΦG Π) t)
  ∧ C = { (State t (index (strips-problem.variables-of Π) v))-1 })
  using nb3
  unfolding nb1 encode-state-variable-def
  by auto+
qed

end

```

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

7.3 Soundness of the Basic SATPlan Algorithm

```

lemma valuation-models-encoding-cnf-formula-equals:
  assumes is-valid-problem-strips Π
  shows A ⊨ Φ Π t = cnf-semantics A (cnf (Φ Π t))
proof -
  let ?Φ = Φ Π t
  {
    have is-cnf ?Φ
      using is-cnf-encode-problem[OF assms].
    hence is-nnf ?Φ
      using is-nnf-cnf
      by blast
  }
  thus ?thesis

```

```

using cnf-semantics[of ?Φ ℬ]
by blast
qed

corollary valuation-models-encoding-cnf-formula-equals-corollary:
assumes is-valid-problem-strips Π
shows ℬ ⊨ (Φ Π t)
  = (forall C ∈ cnf (Φ Π t). ∃ L ∈ C. lit-semantics ℬ L)
using valuation-models-encoding-cnf-formula-equals[OF assms]
unfolding cnf-semantics-def clause-semantics-def encode-problem-def
by presburger

```

— A couple of technical lemmas about *decode-plan*.

```

lemma decode-plan-length:
assumes π = Φ⁻¹ Π ν t
shows length π = t
using assms
unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
by simp

lemma decode-plan'-set-is[simp]:
set (decode-plan' Π ℬ k)
  = { (strips-problem.operators-of Π) ! (index (strips-problem.operators-of Π) op)
    | op. op ∈ set (strips-problem.operators-of Π)
      ∧ ℬ (Operator k (index (strips-problem.operators-of Π) op)) }
proof –
  let ?ops = strips-problem.operators-of Π
  let ?f = λop. Operator k (index ?ops op)
  let ?vs = map ?f ?ops
  {
    have set (filter ℬ ?vs) = set (map ?f (filter (λop. Operator k (index ?ops op)) ?ops))
    unfolding filter-map[of ℬ λop. Operator k (index ?ops op) ?ops]..
    hence set (filter ℬ ?vs) = (λop. Operator k (index ?ops op)) ` { op ∈ set ?ops. ℬ (Operator k (index ?ops op)) }
    unfolding set-map set-filter
    by simp
  }
  have set (decode-plan' Π ℬ k) = (λv. case v of Operator k i ⇒ ?ops ! i)
    ` (λop. Operator k (index ?ops op)) ` { op ∈ set ?ops. ℬ (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. ℬ (Operator k (index ?ops op)) }
    unfolding image-comp comp-apply
    by argo
  also have ... = (λop. ?ops ! (index ?ops op))

```

```

` { op ∈ set ?ops. A (Operator k (index ?ops op)) }
  by force
finally show ?thesis
  by blast
qed

lemma decode-plan-set-is[simp]:
set (Φ⁻¹ Π A t) = (⋃ k ∈ {... { decode-plan' Π A k }})
  unfolding decode-plan-def SAT-Plan-Base.decode-plan-def set-map
  using atLeast-up $t$ 
  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)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)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  $\mathcal{A}$  (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 ( $(\Phi^{-1} \Pi \mathcal{A} t) ! k$ )
proof —
  let  $?ops = \text{strips-problem.operators-of } \Pi$ 
    and  $?_{\pi_k} = (\Phi^{-1} \Pi \mathcal{A} t) ! k$ 
  let  $?f = \lambda op. \text{Operator } k (\text{index } ?ops op)$ 
    and  $?g = \lambda v. \text{case } v \text{ of Operator - } k \Rightarrow ?ops ! k$ 
  let  $?vs = \text{map } ?f (\text{remdups } ?ops)$ 
  have  $nb_1: ?_{\pi_k} = \text{decode-plan}' \Pi \mathcal{A} k$ 
    unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
    using assms
    by fastforce
  {
    have distinct (remdups ?ops)
      by blast
    moreover have inj-on  $?f (\text{set } (\text{remdups } ?ops))$ 
      unfolding inj-on-def
      by fastforce
    ultimately have distinct  $?vs$ 
      using distinct-map
      by blast
  } note  $nb_2 = \text{this}$ 
  {
    have inj-on  $?g (\text{set } ?vs)$ 
      unfolding inj-on-def
      by fastforce
    hence distinct (map ?g ?vs)
      using distinct-map nb2
      by blast
  }
  thus ?thesis
    using distinct-map-filter[of ?g ?vs A]
    unfolding  $nb_1 \text{ decode-plan}'\text{-def Let-def}$ 
    by argo
qed

```

```

lemma decode-state-at-valid-variable:
  fixes  $\Pi :: \text{'a strips-problem}$ 
  assumes  $(\Phi_S^{-1} \Pi \mathcal{A} k) v \neq \text{None}$ 
  shows  $v \in \text{set } ((\Pi)_v)$ 
proof —
  let  $?vs = \text{strips-problem.variables-of } \Pi$ 

```

```

let ?f =  $\lambda v. (v, \mathcal{A} (\text{State } k (\text{index } ?vs v)))$ 
{
  have  $\text{fst} ' \text{set} (\text{map } ?f ?vs) = \text{fst} ' (\lambda v. (v, \mathcal{A} (\text{State } k (\text{index } ?vs v)))) ' \text{set } ?vs$ 
    by force
  also have ... =  $(\lambda v. \text{fst} (v, \mathcal{A} (\text{State } k (\text{index } ?vs v)))) ' \text{set } ?vs$ 
    by blast
  finally have  $\text{fst} ' \text{set} (\text{map } ?f ?vs) = \text{set } ?vs$ 
    by auto
}
moreover have  $\neg v \notin \text{fst} ' \text{set} (\text{map } ?f ?vs)$ 
  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  $\Pi$ :: 'a strips-problem
assumes is-valid-problem-strips  $\Pi$ 
and  $\mathcal{A} \models \Phi \Pi t$ 
and  $k \leq t$ 
and  $v \in \text{set} ((\Pi)_V)$ 
shows  $(\Phi_S^{-1} \Pi \mathcal{A} k) v = \text{Some} (\mathcal{A} (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v)))$ 

```

proof —

```

let ?vs = strips-problem.variables-of  $\Pi$ 
let ?l = map ( $\lambda x. (x, \mathcal{A} (\text{State } k (\text{index } ?vs x)))$ ) ?vs
have set ?vs  $\neq \{\}$ 
  using assms(4)
  by fastforce
then have map-of ?l v = Some ( $\mathcal{A} (\text{State } k (\text{index } ?vs v))$ )
  using map-of-from-function-graph-is-some-if[of ?vs v
     $\lambda v. \mathcal{A} (\text{State } k (\text{index } ?vs v))]$ ] 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  $\Pi$ 
shows dom  $(\Phi_S^{-1} \Pi \mathcal{A} k) = \text{set} ((\Pi)_V)$ 

```

proof —

```

let ?s =  $\Phi_S^{-1} \Pi \mathcal{A} k$ 
and ?vs = strips-problem.variables-of  $\Pi$ 
have dom ?s =  $\text{fst} ' \text{set} (\text{map} (\lambda v. (v, \mathcal{A} (\text{State } k (\text{index } ?vs v)))) ?vs)$ 

```

```

unfolding decode-state-at-def SAT-Plan-Base.decode-state-at-def
using dom-map-of-conv-image-fst[of (map (λv. (v, A (State k (index ?vs v)))))) 
?vs)]
by meson
also have ... = fst ‘(λv. (v, A (State k (index ?vs v))))’ ‘set ((Π)ν)
using set-map[of (λv. (v, A (State k (index ?vs v)))))) ?vs]
by simp
also have ... = (fst o (λv. (v, A (State k (index ?vs v))))) ‘set ((Π)ν)
using image-comp[of fst (λv. (v, A (State k (index ?vs v))))]
by presburger
finally show ?thesis
by force
qed

```

```

lemma decode-state-at-initial-state:
assumes is-valid-problem-strips Π
    and A ⊨ Φ Π t
    shows (ΦS-1 Π A 0) = (Π)I
proof –
    let ?I = (Π)I
    let ?s = ΦS-1 Π A 0
    let ?vs = strips-problem.variables-of Π
    let ?Φ = Φ Π t
    let ?ΦI = ΦI Π
    {
        have is-cnf ?ΦI and cnf ?ΦI ⊆ cnf ?Φ
        subgoal
            using is-cnf-encode-initial-state[OF assms(1)]
            by simp
        subgoal
            using cnf-of-encode-problem-structure(1)
            unfoldng encode-initial-state-def encode-problem-def
            by blast
        done
        then have cnf-semantics A (cnf ?ΦI)
        using cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF
        assms(1)]
            assms(2)
        by blast
        hence ∀ C ∈ cnf ?ΦI. clause-semantics A C
        unfoldng cnf-semantics-def encode-initial-state-def
        by blast
    } note nb1 = this
    {
        {
            fix v
            assume v-in-dom-i: v ∈ dom ?I

```

```

moreover {
  have v-in-variable-set:  $v \in \text{set } ((\Pi)_V)$ 
    using is-valid-problem-strips-initial-of-dom assms(1) v-in-dom-i
    by auto
  hence  $(\Phi_S^{-1} \Pi \mathcal{A} 0) v = \text{Some } (\mathcal{A} (\text{State } 0 (\text{index } ?vs v)))$ 
    using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
      assms(1, 2) - v-in-variable-set]
    by fast
} note nb2 = this
consider (v-initially-true) ?I v = Some True
| (v-initially-false) ?I v = Some False
  using v-in-dom-i
  by fastforce
hence ?I v = ?s v
proof (cases)
  case v-initially-true
  then obtain C
    where  $C \in \text{cnf } ?\Phi_I$ 
      and c-is:  $C = \{ (\text{State } 0 (\text{index } ?vs v))^+ \}$ 
      using cnf-of-encode-initial-state-set v-in-dom-i assms(1)
      by fastforce
    hence  $\mathcal{A} (\text{State } 0 (\text{index } ?vs v)) = \text{True}$ 
      using nb1
      unfolding clause-semantics-def
      by fastforce
    thus ?thesis
      using nb2 v-initially-true
      by presburger
  next
    case v-initially-false
    then obtain C
      where  $C \in \text{cnf } ?\Phi_I$ 
        and c-is:  $C = \{ (\text{State } 0 (\text{index } ?vs v))^{-1} \}$ 
        using cnf-of-encode-initial-state-set assms(1) v-in-dom-i
        by fastforce
      hence  $\mathcal{A} (\text{State } 0 (\text{index } ?vs v)) = \text{False}$ 
        using nb1
        unfolding clause-semantics-def
        by fastforce
      thus ?thesis
        using nb2 v-initially-false
        by presburger
  qed
}
hence ?I  $\subseteq_m$  ?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: ( $\Phi_S^{-1} \amalg \mathcal{A} 0$ ) v = Some ( $\mathcal{A} (\text{State } 0 (\text{index } ?vs v))$ )
  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  $\mathcal{A} ((\text{State } 0 (\text{index } ?vs v))^+)$ 
    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 = {  $(\text{State } 0 (\text{index } ?vs v))^{-1}$  }
    using cnf-of-encode-initial-state-set assms(1) v-in-dom-I
    by fastforce
  hence lit-semantics  $\mathcal{A} ((\text{State } 0 (\text{index } ?vs v))^{-1})$ 
    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  $\mathcal{A}$ -of-s-v: lit-semantics  $\mathcal{A} ((\text{State } 0 (\text{index } ?vs v))^{-1})$ 
    using s-v-is
    by fastforce
  consider (I-v-is-some-true)  $?I v = \text{Some True}$ 
  | (I-v-is-some-false)  $?I v = \text{Some False}$ 
    using v-in-dom-I
    by fastforce
  thus ?thesis
  proof (cases)
    case I-v-is-some-true
    then obtain C
      where C-in-encode-initial-state:  $C \in \text{cnf } ?\Phi_I$ 
      and C-is:  $C = \{ (\text{State } 0 (\text{index } ?vs v))^+ \}$ 
      using cnf-of-encode-initial-state-set assms(1) v-in-dom-I
      by fastforce
    hence lit-semantics  $\mathcal{A} ((\text{State } 0 (\text{index } ?vs v))^+)$ 
    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  $?s \subseteq_m ?I$ 
  using map-le-def
  by blast
} ultimately show ?thesis
  using map-le-antisym
  by blast
qed

lemma decode-state-at-goal-state:
  assumes is-valid-problem-strips  $\Pi$ 
  and  $\mathcal{A} \models \Phi \Pi t$ 
  shows  $(\Pi)_G \subseteq_m \Phi_S^{-1} \Pi \mathcal{A} t$ 
proof -
  let  $?vs = \text{strips-problem.variables-of } \Pi$ 
  and  $?G = (\Pi)_G$ 
  and  $?G' = \Phi_S^{-1} \Pi \mathcal{A} t$ 

```

```

and ?Φ = Φ Π t
and ?ΦG = (ΦG Π) t
{
have is-cnf ?ΦG and cnf ?ΦG ⊆ cnf ?Φ
subgoal
  using encode-goal-state-is-cnf[OF assms(1)]
  by simp
subgoal
  using cnf-of-encode-problem-structure(2)
  unfolding encode-goal-state-def encode-problem-def
  by blast
done
then have cnf-semantics  $\mathcal{A}$  (cnf ?ΦG)
  using cnf-semantics-monotonous-in-cnf-subsets-if is-cnf-encode-problem[OF
assms(1)]
  assms(2)
  by blast
hence  $\forall C \in \text{cnf } ?\Phi_G$ . clause-semantics  $\mathcal{A}$  C
  unfolding cnf-semantics-def encode-initial-state-def
  by blast
} note nb1 = this

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

ultimately have  $(\Phi_S^{-1} \Pi \mathcal{A} t) v = \text{Some } (\mathcal{A} (\text{State } t (\text{index } ?vs v)))$ 
  using map-of-from-function-graph-is-some-if
  by fastforce
} note nb2 = this
{
fix v
assume v-in-dom-G: v ∈ dom ?G
then have v-in-vs: v ∈ set ?vs
  using is-valid-problem-dom-of-goal-state-is assms(1)
  by auto
then have decode-state-at-is:  $(\Phi_S^{-1} \Pi \mathcal{A} t) v = \text{Some } (\mathcal{A} (\text{State } t (\text{index } ?vs$ 
v)))
  using nb2
  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 ⊆ cnf ?Φ_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))+ } ∈ cnf ?Φ_G
    by blast
    then have clause-semantics A { (State t (index ?vs v))+ }
    using nb1
    by blast
    then have lit-semantics A ((State t (index ?vs v))+)
    unfolding clause-semantics-def
    by blast
    hence 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 ⊆ cnf ?Φ_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 } ∈ cnf ?Φ_G
    by blast
    then have clause-semantics A { (State t (index ?vs v))-1 }
    using nb1
    by blast
    then have lit-semantics A ((State t (index ?vs v))-1)
    unfolding clause-semantics-def
    by blast
    hence 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  $\Pi$ 
and  $\mathcal{A} \models \Phi \Pi t$ 
and  $k < t$ 
and  $op \in \text{set}((\Phi^{-1} \Pi \mathcal{A}) ! k)$ 
and  $v \in \text{set}(\text{precondition-of } op)$ 
shows  $\mathcal{A} (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v))$ 

proof —
  let  $?ops = \text{strips-problem.operators-of } \Pi$ 
  and  $?vs = \text{strips-problem.variables-of } \Pi$ 
  let  $?Phi = \Phi \Pi t$ 
  and  $?Phi_O = \text{encode-operators } \Pi t$ 
  and  $?Phi_P = \text{encode-all-operator-preconditions } \Pi ?ops t$ 
  {
    have  $\mathcal{A} (\text{Operator } k (\text{index } ?ops op))$ 
    and  $op \in \text{set}((\Pi)_O)$ 
    using decode-plan-step-element-then[OF assms(3, 4)]
    by blast+
    moreover obtain  $C$ 
      where clause-is-in-operator-encoding:  $C \in \text{cnf } ?Phi_P$ 
      and  $C = \{ (\text{Operator } k (\text{index } ?ops op))^{-1},$ 
         $(\text{State } k (\text{index } ?vs v))^+ \}$ 
      using cnf-of-encode-all-operator-preconditions-contains-clause-if[OF assms(1, 3)
         $\text{calculation}(2) \text{ assms}(5)$ ]
      by blast
      moreover have clause-semantics-A-Phi_P:  $\forall C \in \text{cnf } ?Phi_P. \text{ clause-semantics } \mathcal{A} C$ 
      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  $\Pi$ 
    and  $\mathcal{A} \models \Phi \Pi \theta$ 
  shows cnf  $((\Phi_G \Pi) \theta) \subseteq \text{cnf } (\Phi_I \Pi)$ 
proof -
  let ?vs = strips-problem.variables-of  $\Pi$ 
    and ?I =  $(\Pi)_I$ 
    and ?G =  $(\Pi)_G$ 
    and ? $\Phi_I$  =  $\Phi_I \Pi$ 
    and ? $\Phi_G$  =  $(\Phi_G \Pi) \theta$ 
    and ? $\Phi$  =  $\Phi \Pi \theta$ 
```

— Show that the model of the encoding is also a model of the partial encodings.

```
have  $\mathcal{A}$ -models- $\Phi_I$ :  $\mathcal{A} \models ?\Phi_I$  and  $\mathcal{A}$ -models- $\Phi_G$ :  $\mathcal{A} \models ?\Phi_G$ 
  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- $\Phi_G$ :  $C' \in \text{cnf } ?\Phi_G$ 
  then obtain v
    where v-in-vs:  $v \in \text{set } ?vs$ 
      and G-of-v-is-not-None: ?G v ≠ None
      and C'-is:  $C' = \{ \text{literal-formula-to-literal } (\text{encode-state-variable } 0 \text{ (index } ?vs \text{ v) } (?G v)) \}$ 
    using cnf-of-encode-goal-state-set-ii[OF assms(1)]
    by auto
    obtain C
      where C-in-cnf- $\Phi_I$ :  $C \in \text{cnf } ?\Phi_I$ 
      and C-is:  $C = \{ \text{literal-formula-to-literal } (\text{encode-state-variable } 0 \text{ (index } ?vs \text{ v) } (?I v)) \}$ 
    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 ? $\Phi_I$ 
    using C-in-cnf- $\Phi_I$  C-is
```

```

    by blast
  hence lit-semantics  $\mathcal{A}$  ?L
    using model-then-all-singleton-clauses-modelled[OF
      is-cnf-encode-initial-state[OF assms(1)]-  $\mathcal{A}$ -models- $\Phi_I$ ]
    by blast
  } note lit-semantics- $\mathcal{A}$ -L = this
  {
    let ?L' = literal-formula-to-literal (encode-state-variable 0 (index ?vs v) (?G
      v))
    have { ?L' } ∈ cnf ? $\Phi_G$ 
      using C'-in-cnf- $\Phi_G$  C'-is
      by blast
    hence lit-semantics  $\mathcal{A}$  ?L'
      using model-then-all-singleton-clauses-modelled[OF
        encode-goal-state-is-cnf[OF assms(1)]-  $\mathcal{A}$ -models- $\Phi_G$ ]
      by blast
    } note lit-semantics- $\mathcal{A}$ -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- $\mathcal{A}$ -L lit-semantics- $\mathcal{A}$ -L'
            unfolding encode-state-variable-def
            by (cases, fastforce+)
        qed
      }
      hence C' ∈ cnf ? $\Phi_I$ 
        using C-is C-in-cnf- $\Phi_I$  C'-is C'-in-cnf- $\Phi_G$ 
        by argo
    }
    thus ?thesis
      by blast
  qed

```

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

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

lemma encode-problem-parallel-correct-ii:

```

assumes is-valid-problem-strips  $\Pi$ 
  and  $\mathcal{A} \models \Phi \Pi t$ 
  and  $k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t)$ 
shows are-all-operators-applicable  $(\Phi_S^{-1} \Pi \mathcal{A} k)$ 
 $((\Phi^{-1} \Pi \mathcal{A} t) ! k)$ 
  and are-all-operator-effects-consistent  $((\Phi^{-1} \Pi \mathcal{A} t) ! k)$ 
proof -
let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problem.operators-of  $\Pi$ 
  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  $\Pi$  ?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:  $op \in \text{set } (?\pi ! k)$ 
  and v-in-precondition-set:  $v \in \text{set } (\text{precondition-of } op)$ 
{
have  $\mathcal{A} (\text{Operator } k (\text{index } ?\text{ops } op))$ 
  using decode-plan-step-element-then[OF k-lt-t op-in-kth-of-decoded-plan-set]
  by blast
hence  $\mathcal{A} (\text{State } k (\text{index } ?\text{vs } v))$ 
  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  $op \in \text{set } ((\Pi)_O)$ 
  using decode-plan-step-element-then[OF k-lt-t op-in-kth-of-decoded-plan-set]
  by simp
then have  $v \in \text{set } ((\Pi)_V)$ 
  using is-valid-problem-strips-operator-variable-sets(1) assms(1)
  v-in-precondition-set
  by auto
}
ultimately have  $(\Phi_S^{-1} \Pi \mathcal{A} k) v = \text{Some True}$ 
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
assms(1, 2)]
  by presburger
}
hence are-all-operators-applicable ?s (? $\pi$  ! k)

```

```

using are-all-operators-applicable-set[of ?s ?π ! k]
by blast
} moreover {
{
fix op1 op2
assume op1-in-k-th-of-decoded-plan: op1 ∈ set ((Φ-1 Π A t) ! k)
and op2-in-k-th-of-decoded-plan: op2 ∈ set ((Φ-1 Π A t) ! k)
have op1-in-set-ops: op1 ∈ set ((Π)Ο)
and op2-in-set-ops: op2 ∈ set ((Π)Ο)
and op1-active-at-k: ¬lit-semantics A ((Operator k (index ?ops op1))-1)
and op2-active-at-k: ¬lit-semantics A ((Operator k (index ?ops op2))-1)
subgoal
using decode-plan-step-element-then[OF k-lt-t op1-in-k-th-of-decoded-plan]
by simp
subgoal
using decode-plan-step-element-then[OF k-lt-t op2-in-k-th-of-decoded-plan]
by force
subgoal
using decode-plan-step-element-then[OF k-lt-t op1-in-k-th-of-decoded-plan]
by simp
subgoal
using decode-plan-step-element-then[OF k-lt-t op2-in-k-th-of-decoded-plan]
by simp
done

{
fix v
assume v-in-add-effects-set-of-op1: v ∈ set (add-effects-of op1)
and v-in-delete-effects-set-of-op2: v ∈ set (delete-effects-of op2)
let ?C1 = {(Operator k (index ?ops op1))-1,
(State (Suc k) (index ?vs v))+}
and ?C2 = {(Operator k (index ?ops op2))-1,
(State (Suc k) (index ?vs v))-1}
have ?C1 ∈ cnf ?ΦE and ?C2 ∈ cnf ?ΦE
subgoal
using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[OF
assms(1) k-lt-t op1-in-set-ops v-in-add-effects-set-of-op1]
by blast
subgoal
using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[OF
assms(1) k-lt-t op2-in-set-ops v-in-delete-effects-set-of-op2]
by blast
done
then have ?C1 ∈ cnf ?Φ and ?C2 ∈ cnf ?Φ
using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
by blast+
then have C1-true: clause-semantics A ?C1 and C2-true: clause-semantics
A ?C2
using valuation-models-encoding-cnf-formula-equals[OF assms(1)] assms(2)

```

```

unfolding cnf-semantics-def
  by blast+
have lit-semantics  $\mathcal{A}$  ((State (Suc k) (index ?vs v))+)
and lit-semantics  $\mathcal{A}$  ((State (k + 1) (index ?vs v))-1)
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 auto
} moreover {
  fix v
  assume v-in-delete-effects-set-of-op1: v ∈ set (delete-effects-of op1)
  and v-in-add-effects-set-of-op2: v ∈ set (add-effects-of op2)
let ?C1 = {(Operator k (index ?ops op1))-1, (State (Suc k) (index ?vs v))-1}
  and ?C2 = {(Operator k (index ?ops op2))-1, (State (Suc k) (index ?vs v))+}
have ?C1 ∈ cnf ?ΦE and ?C2 ∈ cnf ?ΦE
  subgoal
    using cnf-of-operator-effect-encoding-contains-delete-effect-clause-if[OF
      assms(1) k-lt-t op1-in-set-ops v-in-delete-effects-set-of-op1]
    by fastforce
  subgoal
    using cnf-of-operator-effect-encoding-contains-add-effect-clause-if[OF
      assms(1) k-lt-t op2-in-set-ops v-in-add-effects-set-of-op2]
    by simp
  done
then have ?C1 ∈ cnf ?Φ and ?C2 ∈ cnf ?Φ
  using cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem
  by blast+
then have C1-true: clause-semantics  $\mathcal{A}$  ?C1 and C2-true: clause-semantics
 $\mathcal{A}$  ?C2
  using valuation-models-encoding-cnf-formula-equals[OF assms(1)] assms(2)
  unfolding cnf-semantics-def
  by blast+
have lit-semantics  $\mathcal{A}$  ((State (Suc k) (index ?vs v))-1)
and lit-semantics  $\mathcal{A}$  ((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)
→ (ΦS-1 Π A (Suc k)) v = Some True
and v ∈ set (delete-effects-of op)
→ (ΦS-1 Π A (Suc k)) v = Some False
proof -
let ?ops = strips-problem.operators-of Π
and ?vs = strips-problem.variables-of Π
let ?ΦF = 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  $v \in \text{set}(\text{add-effects-of } op)$ 
 $\rightarrow (\Phi_S^{-1} \Pi \mathcal{A}(\text{Suc } k)) v = \text{Some True}$ 
and  $v \in \text{set}(\text{delete-effects-of } op)$ 
 $\rightarrow (\Phi_S^{-1} \Pi \mathcal{A}(\text{Suc } k)) v = \text{Some False}$ 
proof (auto)
assume  $v\text{-is-add-effect: } v \in \text{set}(\text{add-effects-of } op)$ 
have  $\mathcal{A}(\text{Operator } k (\text{index } ?ops \text{ } op))$ 
using decode-plan-step-element-then[ $\text{OF } k\text{-lt-}t \text{ assms}(4)$ ]
by blast
moreover {
have  $\{\{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\}\}$ 
 $\in \{\{\{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\}\}$ 
 $| v. v \in \text{set}(\text{add-effects-of } op)\}$ 
using  $v\text{-is-add-effect}$ 
by blast

then have  $\{\{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\}\} \in ?A$ 
using  $k\text{-op-included cnf-of-operator-encoding-structure}$ 
 $\text{UN-iff}[of \{\{( \text{Operator } t (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } t) (\text{index } ?vs \text{ } v))^+\}\}]$ 
 $- \{0..< t\} \times \text{set}((\Pi)_O)]$ 
by blast

then have  $\{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\}$ 
 $\in \bigcup ?A$ 
using  $\text{Union-iff}[of \{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\}]$ 
by blast

moreover have  $\bigcup ?A \subseteq \text{cnf } ?\Phi_F$ 
using  $\text{cnf-of-encode-all-operator-effects-structure}$ 
by blast
ultimately have  $\{( \text{Operator } k (\text{index } ?ops \text{ } op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))^+\} \in \text{cnf } ?\Phi_F$ 
using  $\text{in-mono}[of \bigcup ?A \text{ cnf } ?\Phi_F]$ 
by presburger
}

ultimately have  $\mathcal{A}(\text{State } (\text{Suc } k) (\text{index } ?vs \text{ } v))$ 
using  $\text{cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem}$ 
 $\text{assms}(2)[\text{unfolded valuation-models-encoding-cnf-formula-equals-corollary}[OF$ 
 $\text{assms}(1)]]$ 
unfolding  $\text{Bex-def}$ 
by fastforce
thus  $(\Phi_S^{-1} \Pi \mathcal{A}(\text{Suc } k)) v = \text{Some True}$ 
using  $\text{assms}(1) \text{ assms}(2)$ 
 $\text{decode-state-at-encoding-variables-equals-some-of-valuation-if}$ 

```

`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 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))+1} - {0..} × set ((Π)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 ¬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)V)`
`using v-is-delete-effect is-valid-problem-strips-operator-variable-sets(3) assms(1)`
`op-is-valid`
`by auto`

```

ultimately show ( $\Phi_S^{-1} \Pi \mathcal{A} (\text{Suc } k)$ )  $v = \text{Some False}$ 
  using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF assms(1, 2)]
    by auto
  qed
qed

```

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

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

Now, if the disjunctions are empty—i.e. if no operator which is activated at time step k has either a positive or negative effect—, we have by simplification

$$\begin{aligned}\mathcal{A} \models \neg(\neg a_i \wedge a_{i+1}) &\equiv \mathcal{A} \models a_i \vee \neg a_{i+1} \\ \mathcal{A} \models \neg(a_i \wedge \neg a_{i+1}) &\equiv \mathcal{A} \models \neg a_i \vee a_{i+1}\end{aligned}$$

hence

$$\begin{aligned}\mathcal{A} \models (\neg a_i \vee a_{i+1}) \wedge (a_i \vee \neg a_{i+1}) \\ \rightsquigarrow \mathcal{A} \models \{\{\neg a_i, a_{i+1}\}, \{a_i, \neg a_{i+1}\}\}\end{aligned}$$

The lemma characterizes this simplification.⁸

lemma encode-problem-parallel-correct-iv:

```

fixes  $\Pi$ :: 'a strips-problem
assumes is-valid-problem-strips  $\Pi$ 
and  $\mathcal{A} \models \Phi \Pi t$ 
and  $k < t$ 
and  $v \in \text{set } ((\Pi)_v)$ 
and  $\neg(\exists op \in \text{set } ((\Phi^{-1} \Pi \mathcal{A} t) ! k).$ 
   $v \in \text{set } (\text{add-effects-of } op) \vee v \in \text{set } (\text{delete-effects-of } op))$ 
shows cnf-semantics  $\mathcal{A} \{\{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1}$ 
  ,  $(\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \}\}$ 
and cnf-semantics  $\mathcal{A} \{\{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$ 
  ,  $(\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \}\}$ 
proof —
  let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problem.operators-of  $\Pi$ 
  let ? $\Phi$  =  $\Phi \Pi t$ 
  and ? $\Phi_F$  = encode-all-frame-axioms  $\Pi t$ 
  and ? $\pi_k$  =  $(\Phi^{-1} \Pi \mathcal{A} t) ! k$ 
  and ?A =  $\bigcup (k, v) \in (\{0..<t\} \times \text{set } ?vs).$ 

```

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

```

{ {{ (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 = ∪(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*: $(k, v) \in (\{.. < t\} \times \text{set } ((\Pi)_V))$

using *assms*(3, 4)

by *blast*

have *operator-encoding-subset-encoding*: $\text{cnf } ?\Phi_F \subseteq \text{cnf } ?\Phi$

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 $\text{set } ?\text{ops}$, 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 $\mathcal{A}(\text{Operator } k (\text{index } ?\text{ops } op)) = \perp$ by definition of *decode-plan*; moreover,
- assuming op is not in $\text{set } ?\text{ops}$ —this is implicitly encoded as $\text{Operator } k (\text{length } ?\text{ops})$ and $\mathcal{A}(\text{Operator } k (\text{length } ?\text{ops}))$ may or may not be true—, then it's not in either of the sets.

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

$$\begin{aligned} \forall op. op \in \{ & ((\text{Operator } k (\text{index } ?\text{ops } op))^{+}) \\ & | op. op \in \text{set } ?\text{ops} \wedge v \in \text{set } (\text{add-effects-of } op) \} \\ \longrightarrow & \neg \text{lit-semantics } \mathcal{A} op \end{aligned}$$

and

$$\begin{aligned} \forall op. op \in \{ & ((\text{Operator } k (\text{index } ?\text{ops } op))^{+}) \\ & | op. op \in \text{set } ?\text{ops} \wedge v \in \text{set } (\text{delete-effects-of } op) \} \\ \longrightarrow & \neg \text{lit-semantics } \mathcal{A} op \end{aligned}$$

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 ?πk)
  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 ?πk
          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  $op \notin \{ op \in set ?ops. v \in set (\text{add-effects-of } op) \}$ 
  by blast
moreover have  $?add =$ 
   $(\lambda op. (\text{Operator } k (\text{index } ?ops op))^+)$ 
  ‘ {  $op \in set ?ops. v \in set (\text{add-effects-of } op)$  }
using setcompr-eq-image[of  $\lambda op. (\text{Operator } k (\text{index } ?ops op))^+$ 
   $\lambda op. op \in set ?ops \wedge v \in set (\text{add-effects-of } op)$ ]
  by blast

ultimately have  $(\text{Operator } k (\text{index } ?ops op))^+ \notin ?add$ 
  by force
thus ?thesis using operator-encoding-in-add
  by blast
qed
qed
} moreover {
  fix  $op$ 
assume operator-encoding-in-delete:  $((\text{Operator } k (\text{index } ?ops op))^+) \in ?delete$ 
hence  $\neg \text{lit-semantics } \mathcal{A} ((\text{Operator } k (\text{index } ?ops op))^+)$ 
  proof (cases  $op \in set ?\pi_k$ )
    case True
    then have  $v \notin set (\text{delete-effects-of } op)$ 
      using assms(5)
      by simp
    then have  $(\text{Operator } k (\text{index } ?ops op))^+ \notin ?delete$ 
      by fastforce
    thus ?thesis
      using operator-encoding-in-delete
      by blast
next
  case False
  then show ?thesis
  proof (cases  $op \in set ?ops$ )
    case True
    {
      let  $?A = \{ ?ops ! \text{index } ?ops op \mid op.$ 
         $op \in set ((\Pi)_\mathcal{O}) \wedge \mathcal{A} (\text{Operator } k (\text{index } ?ops op)) \}$ 
      assume lit-semantics  $\mathcal{A} ((\text{Operator } k (\text{index } ?ops op))^+)$ 
      moreover have operator-active-at-k:  $\mathcal{A} (\text{Operator } k (\text{index } ?ops op))$ 
        using calculation
        by auto
      moreover have  $op \in set ((\Pi)_\mathcal{O})$ 
        using True
        by force
      moreover have  $(?ops ! \text{index } ?ops op) \in ?A$ 
        using calculation(2, 3)
        by blast
      ultimately have  $op \in set ?\pi_k$ 
        using decode-plan-step-element-then-i[OF assms(3)]

```

```

        by auto
      hence False
        using False
        by blast
    }
  thus ?thesis
    by blast
next
  case False
  then have 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 A ?C
  = clause-semantics 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 A L
  using nb(2)
  by blast

ultimately have clause-semantics A ?C'
  = clause-semantics 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-A-Φ:cnf-semantics 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)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 ?ΦF and ?C' ∈ cnf ?ΦF
  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 ?ΦF and c'-subset-frame-axiom-encoding: { ?C' } ⊆
cnf ?ΦF
  by blast+
then have { ?C } ⊆ cnf ?Φ and { ?C' } ⊆ cnf ?Φ
  subgoal
    using operator-encoding-subset-encoding
    by fast
  subgoal
    using c'-subset-frame-axiom-encoding operator-encoding-subset-encoding
    by fast
  done

hence cnf-semantics A { ?C } and cnf-semantics A { ?C' }

```

```

using cnf-semantics- $\mathcal{A}$ - $\Phi$  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  $\Pi$ 
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  $\Pi$ 
and ?ops = strips-problemoperators-of  $\Pi$ 
and ? $\pi$  =  $\Phi^{-1} \Pi \mathcal{A} t$ 
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 ? $s_k$  (? $\pi$  !  $k$ )
and ? $\pi'_k$  = ? $\pi$  !  $k$ 
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 ? $\pi$  !  $t$ ] assms(3)
by (argo, fastforce+)
then have operator-preconditions-hold:
are-all-operators-applicable ? $s_k$  ? $\pi_k$   $\wedge$  are-all-operator-effects-consistent ? $\pi'_k$ 
using encode-problem-parallel-correct-ii[OF assms(1, 2, 3)]
by blast

```

— We show the goal in classical fashion by proving that

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

—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-sk' :  $v \in dom ?s'_k$ 
then have sk'-not-none: ? $s'_k$  v ≠ None
by blast
hence ? $s'_k$  v = ? $t'_k$  v
proof (cases  $\exists op \in set ?\pi_k. v \in set (add-effects-of op) \vee v \in set (delete-effects-of op)$ )

```

```

case True
then obtain op
  where op-in-πk: op ∈ set  $\pi_k$ 
  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  $\exists s_k' v = \text{Some True}$ 
        using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)
op-in-πk]
          v-is-add-effect
          by blast
        moreover have are-all-operators-applicable  $(\Phi_S^{-1} \amalg \mathcal{A} k) ((\Phi^{-1} \amalg \mathcal{A}$ 
t) ! k)
          and are-all-operator-effects-consistent  $((\Phi^{-1} \amalg \mathcal{A} t) ! k)$ 
          using operator-preconditions-hold v-is-add-effect
          by blast+
        moreover have  $\exists t_k' v = \text{Some True}$ 
          using execute-parallel-operator-positive-effect-if[of
             $\Phi_S^{-1} \amalg \mathcal{A} k (\Phi^{-1} \amalg \mathcal{A} t) ! k$  op-in-πk
            v-is-add-effect calculation(2, 3)
          by blast
        ultimately show ?thesis
          by argo
next
  case v-is-delete-effect
  then have  $\exists s_k' v = \text{Some False}$ 
    using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)
op-in-πk]
      v-is-delete-effect
      by blast
    moreover have are-all-operators-applicable  $(\Phi_S^{-1} \amalg \mathcal{A} k) ((\Phi^{-1} \amalg \mathcal{A}$ 
t) ! k)
      and are-all-operator-effects-consistent  $((\Phi^{-1} \amalg \mathcal{A} t) ! k)$ 
      using operator-preconditions-hold
      by blast+
    moreover have  $\exists t_k' v = \text{Some False}$ 
      using execute-parallel-operator-effect(2) op-in-πk
      v-is-delete-effect calculation(2, 3)
      by fast
    moreover have  $\exists t_k' v = \text{Some False}$ 
      by (meson execute-parallel-operator-negative-effect-if op-in-πk operator-preconditions-hold v-is-delete-effect)
    ultimately show ?thesis
      by argo

```

```

qed
next
case False

then have  $?t_k' v = ?s_k v$ 
using execute-parallel-operator-no-effect-if
by fastforce
moreover {
have v-in-set-vs:  $v \in \text{set } ((\Pi)_V)$ 
using decode-state-at-valid-variable[ $\text{OF } s_k' \text{-not-none}$ ].
then have state-propagation-positive:
cnf-semantics  $\mathcal{A} \{ \{( \text{State } k \text{ (index } ?vs \text{ v))}^{-1}$ 
 $, ( \text{State } (\text{Suc } k) \text{ (index } ?vs \text{ v))}^+ \} \}$ 
and state-propagation-negative:
cnf-semantics  $\mathcal{A} \{ \{( \text{State } k \text{ (index } ?vs \text{ v))}^+$ 
 $, ( \text{State } (\text{Suc } k) \text{ (index } ?vs \text{ v))}^{-1} \} \}$ 
using encode-problem-parallel-correct-iv[ $\text{OF assms}(1, 2) \text{ k-lt-t - False}$ ]
by fastforce+
consider ( $s_k' \text{-v-positive}$ )  $?s_k' v = \text{Some True}$ 
| ( $s_k' \text{-v-negative}$ )  $?s_k' v = \text{Some False}$ 
using  $s_k' \text{-not-none}$ 
by fastforce
hence  $?s_k' v = ?s_k v$ 
proof (cases)
case  $s_k' \text{-v-positive}$ 
then have lit-semantics  $\mathcal{A} ((\text{State } (\text{Suc } k) \text{ (index } ?vs \text{ v))}^+)$ 
using decode-state-at-encoding-variables-equals-some-of-valuation-if[ $\text{OF assms}(1, 2) \text{ suc-k-lte-t v-in-set-vs}$ ]
by fastforce

then have lit-semantics  $\mathcal{A} ((\text{State } k \text{ (index } ?vs \text{ v))}^+)$ 
using state-propagation-negative
unfolding cnf-semantics-def clause-semantics-def
by fastforce
then show ?thesis
using decode-state-at-encoding-variables-equals-some-of-valuation-if[ $\text{OF assms}(1, 2) \text{ k-lte-t v-in-set-vs} ] s_k' \text{-v-positive}$ 
by fastforce
next
case  $s_k' \text{-v-negative}$ 
then have  $\neg \text{lit-semantics } \mathcal{A} ((\text{State } (\text{Suc } k) \text{ (index } ?vs \text{ v))}^+)$ 
using decode-state-at-encoding-variables-equals-some-of-valuation-if[
 $\text{OF assms}(1, 2) \text{ suc-k-lte-t v-in-set-vs}$ ]
by fastforce

then have  $\neg \text{lit-semantics } \mathcal{A} ((\text{State } k \text{ (index } ?vs \text{ v))}^+)$ 
using state-propagation-positive
unfolding cnf-semantics-def clause-semantics-def
by fastforce

```

```

then show ?thesis
using decode-state-at-encoding-variables-equals-some-of-valuation-if[OF
    assms(1, 2) k-lte-t v-in-set-vs]  $s_k'$ -v-negative
    by fastforce
    qed
}
ultimately show ?thesis
    by argo
    qed
}
hence  $?s_k' \subseteq_m ?t_k'$ 
    using map-le-def
    by blast
}
moreover {
{
    fix v
    assume  $v \in \text{dom } ?t_k'$ 
    then have  $t_k'$ -not-none:  $?t_k' v \neq \text{None}$ 
        by blast
    {
    {
        assume contradiction:  $v \notin \text{set } ((\Pi)_V)$ 
        then have  $(\Phi_S^{-1} \Pi \mathcal{A} k) v = \text{None}$ 
            using decode-state-at-valid-variable
            by fastforce
        then obtain op
            where op-in:  $op \in \text{set } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$ 
            and v-is-or:  $v \in \text{set } (\text{add-effects-of } op)$ 
                 $\vee v \in \text{set } (\text{delete-effects-of } op)$ 
            using execute-parallel-operators-strips-none-if-contraposition[OF
                 $t_k'$ -not-none]
            by blast
        have op-in:  $op \in \text{set } ((\Pi)_O)$ 
            using op-in decode-plan-step-element-then(1) k-lt-t
            by blast
        consider (A)  $v \in \text{set } (\text{add-effects-of } op)$ 
         $| (B) v \in \text{set } (\text{delete-effects-of } op)$ 
        using v-is-or
        by blast
    hence False
    proof (cases)
        case A
        then have v ∈ set  $((\Pi)_V)$ 
            using is-valid-problem-strips-operator-variable-sets(2)[OF
                assms(1)] op-in A
            by blast
        thus False
            using contradiction
    }
}

```

```

    by blast
next
  case B
    then have  $v \in \text{set } ((\Pi)_V)$ 
      using is-valid-problem-strips-operator-variable-sets(3)[OF assms(1)] op-in B
      by blast
    thus False
      using contradiction
      by blast
    qed
  }
hence v-in-set-vs:  $v \in \text{set } ((\Pi)_V)$ 
  by blast
hence  $?t_k' v = ?s_k' v$ 
proof (cases ( $\exists op \in \text{set } ?\pi_k. v \in \text{set } (\text{add-effects-of } op) \vee v \in \text{set } (\text{delete-effects-of } op))$ )
  case True
    then obtain op
      where op-in-set- $\pi_k$ :  $op \in \text{set } ?\pi_k$ 
      and v-options:  $v \in \text{set } (\text{add-effects-of } op) \vee v \in \text{set } (\text{delete-effects-of } op)$ 
      by blast
    then have  $op \in \text{set } ((\Pi)_O)$ 
      using decode-plan-step-element-then[OF k-lt-t]
      by blast
    then consider (v-is-add-effect)  $v \in \text{set } (\text{add-effects-of } op)$ 
      | (v-is-delete-effect)  $v \in \text{set } (\text{delete-effects-of } op)$ 
      using v-options
      by blast
    thus ?thesis
      proof (cases)
        case v-is-add-effect
          then have  $?t_k' v = \text{Some True}$ 
            using execute-parallel-operator-positive-effect-if[OF -- op-in-set- $\pi_k$ ]
            operator-preconditions-hold
            by blast
          moreover have  $?s_k' v = \text{Some True}$ 
            using encode-problem-parallel-correct-iii(1)[OF assms(1, 2, 3)
op-in-set- $\pi_k$ ]
            v-is-add-effect
            by blast
          ultimately show ?thesis
            by argo
      next
        case v-is-delete-effect
          then have  $?t_k' v = \text{Some False}$ 
            using execute-parallel-operator-negative-effect-if[OF -- op-in-set- $\pi_k$ ]
            operator-preconditions-hold

```

```

    by blast
  moreover have ?sk' v = Some False
    using encode-problem-parallel-correct-iii(2)[OF assms(1, 2, 3)
op-in-set-πk]
      v-is-delete-effect
      by blast
      ultimately show ?thesis
        by argo
      qed
    next
      case False
        have state-propagation-positive:
          cnf-semantics A {{(State k (index ?vs v))-1, (State (Suc k) (index ?vs
v))+}}
        and state-propagation-negative:
          cnf-semantics A {{(State k (index ?vs v))+, (State (Suc k) (index ?vs
v))-1}}
        using encode-problem-parallel-correct-iv[OF assms(1, 2) k-lt-t v-in-set-vs
          False]
        by blast+
      {
        have all-op-in-set-πk-have-no-effect:
          ∀ op ∈ set ?πk. v ∉ set (add-effects-of op) ∧ v ∉ set (delete-effects-of
op)
        using False
        by blast
        then have ?tk' v = ?sk v
        using execute-parallel-operator-no-effect-if[OF all-op-in-set-πk-have-no-effect]
        by blast
      } note tk'-equals-sk = this
      {
        have ?sk v ≠ None
        using tk'-not-none tk'-equals-sk
        by argo
        then consider (sk-v-is-some-true) ?sk v = Some True
          | (sk-v-is-some-false) ?sk v = Some False
          by fastforce
      }
      then show ?thesis
    proof (cases)
      case sk-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] sk-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 ?sk' 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 tk'-equals-sk
  by simp
next
case sk-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] sk-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 ?sk' 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 tk'-equals-sk
  by simp
qed
qed
}
hence ?tk' ⊆m ?sk'
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
= ΦS-1 Π A k
using assms
proof -
let ?I = (Π)I

```

```

and  $\pi = \Phi^{-1} \Pi \mathcal{A} t$ 
let  $\tau = \text{trace-parallel-plan-strips } I \pi$ 
show  $\text{thesis}$ 
using assms
proof (induction k)
case 0
hence  $\tau ! 0 = I$ 
using trace-parallel-plan-strips-head-is-initial-state
by blast
moreover have  $\Phi_S^{-1} \Pi \mathcal{A} 0 = I$ 
using decode-state-at-initial-state[OF assms(1, 2)]
by simp
ultimately show ?case
by simp
next
case (Suc k)
let  $\tau_k = \text{trace-parallel-plan-strips } I \pi ! k$ 
and  $s_k = \Phi_S^{-1} \Pi \mathcal{A} k$ 
have  $k < \text{length } \tau - 1 \text{ and } k < \text{length } \tau$ 
?
using Suc.prems(3)
by linarith+
— Use the induction hypothesis to obtain the proposition for the previous step
k. Then, show that applying the k-th parallel operator in the plan  $\pi$  on either the
state obtained from the trace or decoded from the model yields the same successor
state.
{
have  $\tau ! k = \text{execute-parallel-plan } I (\text{take } k \pi)$ 
using trace-parallel-plan-prefix k-lt-length-τ
by blast
hence  $\tau_k = s_k$ 
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 ?τk (?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  $k < \text{length } \pi$ 
using Suc.prems(3)
unfolding Suc-eq-plus1
by linarith
hence  $\Phi_S^{-1} \Pi \mathcal{A} (\text{Suc } k)$ 
= execute-parallel-operator ?sk (?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  $\Pi$ 
  and  $\mathcal{A} \models \Phi \Pi t$ 
shows length (map (decode-state-at  $\Pi$   $\mathcal{A}$ )
  [0.. $< \text{Suc}(\text{length } (\Phi^{-1} \Pi \mathcal{A} t))$ ])
= length (trace-parallel-plan-strips (( $\Pi$ )I) ( $\Phi^{-1} \Pi \mathcal{A} t$ ))
proof -
let ?I = ( $\Pi$ )I
and ? $\pi$  =  $\Phi^{-1} \Pi \mathcal{A} t$ 
let ? $\sigma$  = map (decode-state-at  $\Pi$   $\mathcal{A}$ ) [0.. $< \text{Suc}(\text{length } ?\pi)$ ]
  and ? $\tau$  = trace-parallel-plan-strips ?I ? $\pi$ 
let ?l = length ? $\tau$ 
let ?k = ?l - 1
show ?thesis
proof (rule ccontr)
assume length- $\sigma$ -neq-length- $\tau$ : length ? $\sigma$  ≠ length ? $\tau$ 
{
have length ? $\sigma$  = length ? $\pi$  + 1
  by fastforce
moreover have length ? $\tau$  ≤ length ? $\pi$  + 1
  using length-trace-parallel-plan-strips-lte-length-plan-plus-one
  by blast
moreover have length ? $\tau$  < length ? $\pi$  + 1
  using length- $\sigma$ -neq-length- $\tau$  calculation
  by linarith
} note nb1 = this
{
have 0 < length ? $\tau$ 
  using trace-parallel-plan-strips-not-nil..
then have length ? $\tau$  - 1 < length ? $\pi$ 
  using nb1
  by linarith
} note nb2 = this
{
obtain k' where length ? $\tau$  = Suc k'
  using less-imp-Suc-add[OF length-trace-parallel-plan-gt-0]
  by blast
hence ?k < length ? $\pi$ 
  using nb2
  by blast
} note nb3 = this

```

```

{
  have ?τ ! ?k = execute-parallel-plan ?I (take ?k ?π)
    using trace-parallel-plan-plan-prefix[of ?k]
      length-trace-minus-one-lt-length-trace
    by blast
  thm encode-problem-parallel-correct-vi[OF assms(1, 2)] nb3
  moreover have (ΦS-1 Π A ?k) = ?τ ! ?k
    using encode-problem-parallel-correct-vi[OF assms(1, 2)
      length-trace-minus-one-lt-length-trace]..
  ultimately have (ΦS-1 Π A ?k) = execute-parallel-plan ?I (take ?k ?π)
    by argo
} note nb4 = this
{
  have are-all-operators-applicable (ΦS-1 Π A ?k) (?π ! ?k)
    and are-all-operator-effects-consistent (?π ! ?k)
    using encode-problem-parallel-correct-ii(1, 2)[OF assms(1, 2)] nb3
    by blast+
  — Unsure why calculation(1, 2) is needed for this proof step. Should just
  require the default proof.
  moreover have ¬are-all-operators-applicable (ΦS-1 Π A ?k) (?π ! ?k)
    and ¬are-all-operator-effects-consistent (?π ! ?k)
    using length-trace-parallel-plan-strips-lt-length-plan-plus-one-then[OF nb1]
      calculation(1, 2)
    unfolding nb3 nb4
    by blast+
  ultimately have False
    by blast
}
thus False.
qed
qed

lemma encode-problem-parallel-correct-x:
  assumes is-valid-problem-strips Π
    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 (( $\Pi$ )I) ( $\Phi^{-1} \Pi \mathcal{A} t$ ) ! k
  =  $\Phi_S^{-1} \Pi \mathcal{A} k$ 
  using encode-problem-parallel-correct-vi[OF assms]
  by blast
moreover {
  have length ? $\tau$  ≤ length ? $\pi$  + 1
    using length-trace-parallel-plan-strips-lte-length-plan-plus-one
    by blast
  then have k < length ? $\pi$  + 1
    using k-lt-length- $\tau$ 
    by linarith
  then have k < Suc (length ? $\pi$ ) - 0
    using simp
  hence ? $\sigma$  ! k =  $\Phi_S^{-1} \Pi \mathcal{A} k$ 
    using nth-map-upd[of k Suc (length ? $\pi$ ) 0]
    by auto
}
ultimately have ? $\tau$  ! k = ? $\sigma$  ! k
  by argo
}
ultimately have ? $\tau$  = ? $\sigma$ 
  using list-eq-iff-nth-eq[of ? $\tau$  ? $\sigma$ ]
  by blast
}
thus ?thesis
  by argo
qed

lemma encode-problem-parallel-correct-xi:
fixes  $\Pi$ :: 'a strips-problem
assumes is-valid-problem-strips  $\Pi$ 
and  $\mathcal{A} \models \Phi \Pi t$ 
and ops ∈ set ( $\Phi^{-1} \Pi \mathcal{A} t$ )
and op ∈ set ops
shows op ∈ set (( $\Pi$ )O)
proof -
let ? $\pi$  =  $\Phi^{-1} \Pi \mathcal{A} t$ 
have length ? $\pi$  = t
  using decode-plan-length
  by force
moreover obtain k where k < length ? $\pi$  and ops = ? $\pi$  ! k
  using in-set-conv-nth[of ops ? $\pi$ ] 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

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

decoded from the model \mathcal{A} . Let

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

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 $G \subseteq_m \text{last } \sigma$ by lemma ??, i.e. the last state reached by plan execution (and moreover the last state decoded from the model), satisfies the goal state G defined by the problem. By lemma ??, we can conclude that π is a solution for I and G .

Moreover, we show that all operators op in all parallel operators $ops \in \text{set } \pi$ are also contained in \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  $\Pi$ 
    and  $\mathcal{A} \models \Phi \Pi t$ 
shows is-parallel-solution-for-problem  $\Pi (\Phi^{-1} \Pi \mathcal{A} t)$ 
proof -
  let  $?ops = \text{strips-problem.operators-of } \Pi$ 
    and  $?I = (\Pi)_I$ 
    and  $?G = (\Pi)_G$ 
    and  $?pi = \Phi^{-1} \Pi \mathcal{A} t$ 
  let  $?sigma = \text{map } (\lambda k. \Phi_S^{-1} \Pi \mathcal{A} k) [0..<\text{Suc } (\text{length } ?pi)]$ 
    and  $?tau = \text{trace-parallel-plan-strips } ?I ?pi$ 
  {
    have  $?sigma = ?tau$ 
      using encode-problem-parallel-correct-x[OF assms].
    moreover {
      have  $\text{length } ?pi = t$ 
        using decode-plan-length
        by auto
      then have  $?G \subseteq_m \text{last } ?sigma$ 
        using decode-state-at-goal-state[OF assms]
        by simp
  }

```

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

```

}
ultimately have  $((\Pi)_G) \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t)$ 
  using execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace
  by auto
}
moreover have  $\forall ops \in \text{set } ?\pi. \forall op \in \text{set } ops. op \in \text{set } ((\Pi)_O)$ 
  using encode-problem-parallel-correct-xi[OF assms(1, 2)]
  by auto
ultimately show ?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 ($\langle \mathcal{A}_0 \rangle$)
where empty-valuation $\equiv (\lambda \cdot. \text{False})$

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

— Since the trace may be shorter than the plan length even though the last trace element subsumes the goal state—namely in case plan execution is impossible due to violation of the execution condition but the reached state serendipitously subsumes the goal state—, we also have to repeat the valuation for all time steps $k' \in \{\text{length } \tau.. \text{length } \pi + 1\}$ for all $v \in \mathcal{V}$ (see \mathcal{A}_2).

definition valuation-for-state-variables :: 'variable strips-problem
 \Rightarrow 'variable strips-operator list list
 \Rightarrow 'variable strips-state list
 \Rightarrow sat-plan-variable valuation
where valuation-for-state-variables $\Pi \pi \tau \equiv \text{let}$
 $t' = \text{length } \tau$
 $; \tau_\Omega = \tau ! (t' - 1)$
 $; vs = \text{variables-of } \Pi$
 $; V_1 = \{ \text{State } k (\text{index vs } v) \mid k v. k \in \{0..<t'\} \wedge v \in \text{set } vs \}$
 $; V_2 = \{ \text{State } k (\text{index vs } v) \mid k v. k \in \{t'..(\text{length } \pi + 1)\} \wedge v \in \text{set } vs \}$
 $; \mathcal{A}_1 = \text{foldr}$
 $(\lambda(k, v). \mathcal{A}. \text{valuation-for-state } (\text{variables-of } \Pi) (\tau ! k) k v \mathcal{A})$

```

(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 A_0 as the valuation that is false for every argument ensures this implicitly.

```

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

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

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

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

or all $v \in \mathcal{V}$ where $s \equiv \tau ! k$. ¹⁰

Similarly to \mathcal{A}_V , we obtain an operator valuation \mathcal{A}_O by defining

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

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

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  $\Pi$ 
  ; ops = operators-of  $\Pi$ 
  ;  $\tau$  = trace-parallel-plan-strips (initial-of  $\Pi$ )  $\pi$ 
  ; t = length  $\pi$ 
  ; t' = length  $\tau$ 
  ;  $\mathcal{A}_V$  = 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 ∈ {0..<t + 1} ∧ v ∈ set vs }
  ; Op = { Operator k (index ops op)
    | k op. k ∈ {0..<t} ∧ op ∈ 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  $\Pi 0 = \neg\perp \wedge \neg\perp$ 
  unfolding encode-operators-def encode-all-operator-effects-def
    encode-all-operator-preconditions-def
  by simp
  moreover have encode-all-frame-axioms  $\Pi 0 = \neg\perp$ 
  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
```

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

```
lemma model-initial-state-set-valuations:
  assumes is-valid-problem-strips  $\Pi$ 
  shows set (map ( $\lambda v.$  case  $((\Pi)_I)$  v of Some b
```

```

 $\Rightarrow \mathcal{A}_0(\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v) := b)$ 
 $| - \Rightarrow \mathcal{A}_0)$ 
 $(\text{strips-problem.variables-of } \Pi))$ 
 $= \{ \mathcal{A}_0(\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v) := \text{the } (((\Pi)_I) v))$ 
 $| v. v \in \text{set } ((\Pi)_V) \}$ 
proof –
let  $?I = (\Pi)_I$ 
and  $?vs = \text{strips-problem.variables-of } \Pi$ 
let  $?f = \lambda v. \text{case } ((\Pi)_I) v \text{ of Some } b$ 
 $\Rightarrow \mathcal{A}_0(\text{State } 0 (\text{index } ?vs v) := b) | - \Rightarrow \mathcal{A}_0$ 
and  $?g = \lambda v. \mathcal{A}_0(\text{State } 0 (\text{index } ?vs v) := \text{the } (?I v))$ 
let  $?As = \text{map } ?f ?vs$ 
have  $nb_1 : \text{dom } ?I = \text{set } ((\Pi)_V)$ 
using is-valid-problem-strips-initial-of-dom assms
by fastforce
{
{
fix  $v$ 
assume  $v \in \text{dom } ?I$ 
hence  $?f v = ?g v$ 
using  $nb_1$ 
by fastforce
}
hence  $?f ` \text{set } ((\Pi)_V) = ?g ` \text{set } ((\Pi)_V)$ 
using  $nb_1$ 
by force
}
then have  $\text{set } ?As = ?g ` \text{set } ((\Pi)_V)$ 
unfolding set-map
by simp
thus thesis
by blast
qed

```

lemma *valuation-of-state-variable-implies-lit-semantics-if*:

assumes $v \in \text{dom } S$

and $\mathcal{A}(\text{State } k (\text{index } vs v)) = \text{the } (S v)$

shows lit-semantics $\mathcal{A}(\text{literal-formula-to-literal } (\text{encode-state-variable } k (\text{index } vs v) (S v)))$

proof –
let $?L = \text{literal-formula-to-literal } (\text{encode-state-variable } k (\text{index } vs v) (S v))$
consider $(\text{True}) S v = \text{Some True}$
 $| (\text{False}) S v = \text{Some False}$
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 \mathcal{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 |= ΦI Π
proof –

```

```

let ?vs = strips-problem.variables-of Π
  and ?I = (Π)I
  and ?G = (Π)G
  and ?ΦI = ΦI Π
let ?τ = trace-parallel-plan-strips ?I π
{
fix C
assume C ∈ cnf ?ΦI
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 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 $\Pi :: 'a \text{ strips-problem}$

assumes *is-valid-problem-strips* Π

and $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$

and $\forall v k. k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$

$\rightarrow (\mathcal{A} (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v))$

$\leftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True})$

and $\forall v l. l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \wedge l < \text{length } \pi + 1$

$\rightarrow \mathcal{A} (\text{State } l (\text{index} (\text{strips-problem.variables-of } \Pi) v))$

$= \mathcal{A} (\text{State } (\text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$

$(\text{index} (\text{strips-problem.variables-of } \Pi) v))$

shows $\mathcal{A} \models (\Phi_G \Pi) (\text{length } \pi)$

proof —

let $?vs = \text{strips-problem.variables-of } \Pi$

and $?I = (\Pi)_I$

and $?G = (\Pi)_G$

and $?Phi_I = \Phi_I \Pi$

and $?t = \text{length } \pi$

and $?Phi_G = (\Phi_G \Pi) (\text{length } \pi)$

let $?tau = \text{trace-parallel-plan-strips } ?I \pi$

let $?t' = \text{length } ?tau$

{

fix v

assume *G-of-v-is-not-None*: $?G v \neq \text{None}$

have $?G \subseteq_m \text{last } ?tau$

using *execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace assms(2)*

by *blast*

also have $\dots = ?tau ! (?t' - 1)$

using *last-conv-nth[OF trace-parallel-plan-strips-not-nil]*.

finally have $?G \subseteq_m ?tau ! (?t' - 1)$

by *argo*

hence $(?tau ! (?t' - 1)) v = ?G v$

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 A (State ?t (index ?vs v)) ↔ ?G v = Some True
    proof (cases ?t' = ?t + 1)
      case True
        moreover have ?t < ?t'
          using calculation
          by fastforce
        moreover have A (State ?t (index ?vs v)) ↔ (?τ ! ?t) v = Some True
          using assms(3) calculation(2)
          by blast
        ultimately show ?thesis
          using nb1[OF G-of-v-is-not-None]
          by force
      next
      case False
      {
        have ?t' < ?t + 1
        using length-trace-parallel-plan-strips-lte-length-plan-plus-one False
        le-neq-implies-less
        by blast
        moreover have A (State ?t (index ?vs v)) = A (State (?t' - 1) (index
?vs v))
          using assms(4) calculation
          by simp
        moreover have ?t' - 1 < ?t'
        using trace-parallel-plan-strips-not-nil length-greater-0-conv[of ?τ]
        less-diff-conv2[of 1 ?t' ?t]
        by force
        moreover have A (State (?t' - 1) (index ?vs v)) ↔ (?τ ! (?t' - 1)) v
= Some True
          using assms(3) calculation(3)
          by blast
        ultimately have A (State ?t (index ?vs v)) ↔ (?τ ! (?t' - 1)) v =
Some True
          by blast
      }
      thus ?thesis
        using nb1[OF G-of-v-is-not-None]
        by presburger
      qed
    } note nb2 = this
    {
      fix C
      assume C-in-cnf-of-Φ_G: C ∈ cnf ?Φ_G
      moreover obtain v

```

```

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

```

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

```

lemma  $\text{encode-problem-parallel-complete-iii-a}$ :
fixes  $\Pi : 'a \text{ strips-problem}$ 
assumes  $\text{is-valid-problem-strips } \Pi$ 
and  $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$ 
and  $C \in \text{cnf} (\text{encode-all-operator-preconditions } \Pi (\text{strips-problem.operators-of } \Pi) (\text{length } \pi))$ 
and  $\forall k \text{ op. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1$ 
 $\longrightarrow \mathcal{A} (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op} \in \text{set } (\pi ! k))$ 
and  $\forall l \text{ op. } l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \wedge l < \text{length } \pi$ 
 $\longrightarrow \neg \mathcal{A} (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op}))$ 
and  $\forall v \text{ k. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
 $\longrightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))$ 
 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True})$ 
shows  $\text{clause-semantics } \mathcal{A} C$ 
proof –
let  $?ops = \text{strips-problem.operators-of } \Pi$ 
and  $?vs = \text{strips-problem.variables-of } \Pi$ 
and  $?t = \text{length } \pi$ 
let  $?τ = \text{trace-parallel-plan-strips } ((\Pi)_I) \pi$ 

obtain  $k \text{ op}$ 
where  $k\text{-and-op-are}$ :  $(k, \text{op}) \in (\{0..<?t\} \times \text{set } ((\Pi)_O))$ 
and  $C \in (\bigcup v \in \text{set } (\text{precondition-of op}). \{\{ (\text{Operator } k (\text{index } ?ops \text{ op}))^{-1}$ 
 $, (\text{State } k (\text{index } ?vs v))^+ \}\})$ 

```

```

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 \mathcal{A} here since it's not needed.

```

lemma encode-problem-parallel-complete-iii-b:
  fixes  $\Pi$ ::'a strips-problem
  assumes is-valid-problem-strips  $\Pi$ 
  and  $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$ 
  and  $C \in \text{cnf} (\text{encode-all-operator-effects } \Pi (\text{strips-problem.operators-of } \Pi) (\text{length } \pi))$ 
  and  $\forall k \text{ op. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1$ 
     $\rightarrow \mathcal{A} (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op} \in \text{set } (\pi ! k))$ 
  and  $\forall l \text{ op. } l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \wedge l < \text{length } \pi$ 
     $\rightarrow \neg \mathcal{A} (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op}))$ 
  and  $\forall v \text{ k. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
     $\rightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$ 
     $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True})$ 
  shows clause-semantics  $\mathcal{A} C$ 
  proof –
    let ?ops = strips-problem.operators-of  $\Pi$ 
    and ?vs = strips-problem.variables-of  $\Pi$ 
    and ?t = length  $\pi$ 
    let ?τ = trace-parallel-plan-strips  $((\Pi)_I) \pi$ 
    let ?A =  $(\bigcup (k, \text{op}) \in \{0..<?t\} \times \text{set } ((\Pi)_O))$ .
       $\bigcup v \in \text{set } (\text{add-effects-of } \text{op}).$ 
       $\{\{ (\text{Operator } k (\text{index } ?ops \text{ op}))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ v}))^+ \}\}$ 
    and ?B =  $(\bigcup (k, \text{op}) \in \{0..<?t\} \times \text{set } ((\Pi)_O))$ .
       $\bigcup v \in \text{set } (\text{delete-effects-of } \text{op}).$ 
       $\{\{ (\text{Operator } k (\text{index } ?ops \text{ op}))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ v}))^{-1} \}\}$ 
    consider (C-in-A)  $C \in ?A$ 
    | (C-in-B)  $C \in ?B$ 
    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:  $(k, \text{op}) \in \{0..<?t\} \times \text{set}((\Pi)_O)$ 
        and  $C \in (\bigcup v \in \text{set } (\text{add-effects-of } \text{op}))$ .
           $\{\{ (\text{Operator } k (\text{index } ?ops \text{ op}))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ v}))^+ \}\}$ 
        by blast
      then obtain v where v-in-add-effects-of-op:  $v \in \text{set } (\text{add-effects-of } \text{op})$ 
        and C-is:  $C = \{ (\text{Operator } k (\text{index } ?ops \text{ op}))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs \text{ v}))^+ \}$ 
        by blast
      thus ?thesis
  
```

```

proof (cases  $k < \text{length } ?\tau - 1$ )
  case  $k\text{-lt-length-}\tau\text{-minus-one}$ : True
  thus ?thesis
    proof (cases  $op \in \text{set } (\pi ! k)$ )
      case True
      {
        then have are-all-operators-applicable ( $??\tau ! k$ ) ( $\pi ! k$ )
        and are-all-operator-effects-consistent ( $\pi ! k$ )
        using trace-parallel-plan-strips-operator-preconditions  $k\text{-lt-length-}\tau\text{-minus-one}$ 
        by blast+
        hence execute-parallel-operator ( $??\tau ! k$ ) ( $\pi ! k$ )  $v = \text{Some True}$ 
        using execute-parallel-operator-positive-effect-if[
           $OF \dashv \text{True } v\text{-in-add-effects-of-}op$ , of  $??\tau ! k$ ]
        by blast
      }
      then have  $\tau\text{-Suc-}k\text{-is-Some-True}$ : ( $??\tau ! \text{Suc } k$ )  $v = \text{Some True}$ 
      using trace-parallel-plan-step-effect-is[ $OF k\text{-lt-length-}\tau\text{-minus-one}$ ]
      by argo
      have  $\mathcal{A}(\text{State } (\text{Suc } k) (\text{index } ?vs v))$ 
      using assms(6)  $k\text{-lt-length-}\tau\text{-minus-one } \tau\text{-Suc-}k\text{-is-Some-True}$ 
      by fastforce
      thus ?thesis
      using C-is
      unfolding clause-semantics-def
      by fastforce
    next
      case False
      then have  $\neg\mathcal{A}(\text{Operator } k (\text{index } ?ops op))$ 
      using assms(4)  $k\text{-lt-length-}\tau\text{-minus-one}$ 
      by blast
      thus ?thesis
      using C-is
      unfolding clause-semantics-def
      by force
    qed
  next
    case False
    then have  $k \geq \text{length } ?\tau - 1$  and  $k < ?t$ 
    using k-and-op-are
    by auto
    then have  $\neg\mathcal{A}(\text{Operator } k (\text{index } ?ops op))$ 
    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*: $(k, op) \in \{0..<?t\} \times \text{set } ((\Pi)_O)$

and $C \in (\bigcup_{v \in \text{set } (\text{delete-effects-of } op)} \{\})$.

$\{\{ (\text{Operator } k (\text{index } ?ops op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^{-1} \}\})$

by *blast*

then obtain *v* **where** *v-in-delete-effects-of-op*: $v \in \text{set } (\text{delete-effects-of } op)$

and *C-is*: $C = \{ (\text{Operator } k (\text{index } ?ops op))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs$

$v))^{-1} \}$

by *blast*

thus *?thesis*

proof (*cases* $k < \text{length } ?\tau - 1$)

case *k-lt-length-τ-minus-one*: *True*

thus *?thesis*

proof (*cases* *op* $\in \text{set } (\pi ! k)$)

case *True*

 {

then have *are-all-operators-applicable* $(?\tau ! k) (\pi ! k)$

and *are-all-operator-effects-consistent* $(\pi ! k)$

using *trace-parallel-plan-strips-operator-preconditions k-lt-length-τ-minus-one*

by *blast+*

hence *execute-parallel-operator* $(?\tau ! k) (\pi ! k) v = \text{Some False}$

using *execute-parallel-operator-negative-effect-if[*

OF $-$ *True* *v-in-delete-effects-of-op*, *of* $?\tau ! k]$

by *blast*

 }

then have *τ-Suc-k-is-Some-True*: $(?\tau ! \text{Suc } k) v = \text{Some False}$

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

by *argo*

have $\neg A (\text{State } (\text{Suc } k) (\text{index } ?vs v))$

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 $\neg A (\text{Operator } k (\text{index } ?ops op))$

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  $k \geq \text{length } ?\tau - 1$  and  $k < ?t$ 
  using k-and-op-are
  by auto
then have  $\neg \mathcal{A} (\text{Operator } k (\text{index } ?\text{ops } op))$ 
  using assms(5)
  by blast
thus ?thesis
  using C-is
  unfolding clause-semantics-def
  by fastforce
qed
qed
qed

```

lemma encode-problem-parallel-complete-iii:

fixes Π ::'a strips-problem

assumes is-valid-problem-strips Π

and $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$

and $\forall k \text{ op. } k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \rightarrow \mathcal{A} (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op} \in \text{set } (\pi ! k))$

and $\forall l \text{ op. } l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \wedge l < \text{length } \pi \rightarrow \neg \mathcal{A} (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } \Pi) \text{ op}))$

and $\forall v k. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \rightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)) \longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True})$

shows $\mathcal{A} \models \text{encode-operators } \Pi (\text{length } \pi)$

proof –

let $?t = \text{length } \pi$

and $?ops = \text{strips-problem.operators-of } \Pi$

let $?F_O = \text{encode-operators } \Pi ?t$

and $?F_P = \text{encode-all-operator-preconditions } \Pi ?ops ?t$

and $?F_E = \text{encode-all-operator-effects } \Pi ?ops ?t$

{

fix C

assume $C \in \text{cnf } ?F_O$

then consider (C-in-precondition-encoding) $C \in \text{cnf } ?F_P$

| (C-in-effect-encoding) $C \in \text{cnf } ?F_E$

using cnf-of-operator-encoding-structure

by blast

hence clause-semantics $\mathcal{A} C$

proof (cases)

case C-in-precondition-encoding

thus ?thesis

using encode-problem-parallel-complete-iii-a[$\text{OF assms}(1, 2) - \text{assms}(3, 4, 5)$]

by blast

next

```

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

lemma encode-problem-parallel-complete-iv-a:
fixes  $\Pi :: 'a \text{ strips-problem}$ 
assumes STRIPS-Semantics.is-parallel-solution-for-problem  $\Pi \pi$ 
and  $\forall k \text{ op}. k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) = 1$ 
   $\rightarrow \mathcal{A} (\text{Operator } k (\text{index} (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op} \in \text{set} (\pi ! k))$ 
and  $\forall v k. k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
   $\rightarrow (\mathcal{A} (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v)))$ 
   $\leftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$ 
and  $\forall v l. l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \wedge l < \text{length } \pi + 1$ 
   $\rightarrow \mathcal{A} (\text{State } l (\text{index} (\text{strips-problem.variables-of } \Pi) v))$ 
   $= \mathcal{A} (\text{State}$ 
     $(\text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$ 
     $(\text{index} (\text{strips-problem.variables-of } \Pi) v))$ 
and  $C \in \bigcup (\bigcup (k, v) \in \{0..<\text{length } \pi\} \times \text{set } ((\Pi)_V).$ 
   $\{\{\{ (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v))^+$ 
   $, (\text{State } (\text{Suc } k) (\text{index} (\text{strips-problem.variables-of } \Pi) v))^{-1} \}$ 
   $\cup \{ (\text{Operator } k (\text{index} (\text{strips-problem.operators-of } \Pi) \text{ op}))^+$ 
     $| \text{op}. \text{op} \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{add-effects-of } \text{op}) \}\}\}$ 
shows clause-semantics  $\mathcal{A} C$ 
proof -
let ?vs = strips-problem.variables-of  $\Pi$ 
and ?ops = strips-problem.operators-of  $\Pi$ 
and ?t = length  $\pi$ 
let ? $\tau$  = trace-parallel-plan-strips  $((\Pi)_I) \pi$ 
let ?A =  $(\bigcup (k, v) \in \{0..<?t\} \times \text{set } ?vs.$ 
   $\{\{\{ (\text{State } k (\text{index } ?vs v))^+, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^{-1} \}$ 
   $\cup \{ (\text{Operator } k (\text{index } ?ops \text{ op}))^+ | \text{op}. \text{op} \in \text{set } ?ops \wedge v \in \text{set } (\text{add-effects-of } \text{op}) \}\}\}$ 
}

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) ∈ {0..<?t} × set ?vs
  and C' ∈ {{ (State k (index ?vs v))+, (State (Suc k) (index ?vs v))-1 }
    ∪ { (Operator k (index ?ops op))+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of
      op) } } }
    using UN-E
    by blast
  hence ∃ k v.
    k ∈ {0..<?t}
    ∧ v ∈ set ?vs
    ∧ C = { (State k (index ?vs v))+, (State (Suc k) (index ?vs v))-1 }
      ∪ { (Operator k (index ?ops op))+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of
        op) } }
    using C-in-C'
    by blast
  }
then obtain k v
  where k-in: k ∈ {0..<?t}
  and v-in-vs: v ∈ set ?vs
  and C-is: C = { (State k (index ?vs v))+, (State (Suc k) (index ?vs v))-1 }
    ∪ { (Operator k (index ?ops op))+ | op. op ∈ set ?ops ∧ v ∈ set (add-effects-of
      op) } }
    by blast
  show ?thesis
  proof (cases k < length ?τ - 1)
    case k-lt-length-τ-minus-one: True
    then have k-lt-t: k < ?t
      using k-in
      by force
    have all-operators-applicable: are-all-operators-applicable (?τ ! k) (π ! k)
      and all-operator-effects-consistent: are-all-operator-effects-consistent (π ! k)
    using trace-parallel-plan-strips-operator-preconditions[OF k-lt-length-τ-minus-one]
      by simp+
    then consider (A) ∃ op ∈ set (π ! k). v ∈ set (add-effects-of op)
      | (B) ∃ op ∈ set (π ! k). v ∈ set (delete-effects-of op)
      | (C) ∀ 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-πk: op ∈ set (π ! k)
        and v-is-add-effect: v ∈ set (add-effects-of op)
        using A
        by blast
      moreover {
        have (π ! k) ∈ set π
        using k-lt-t
      }
  
```

```

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

```

```

then have  $\mathcal{A} (\text{State } k (\text{index } ?vs v))$ 
  using assms(3) k-lt-length- $\tau$ -minus-one
  by force
thus  $?thesis$ 
  using C-is
  unfolding clause-semantics-def
  by fastforce
next
  case False
  {
    have  $(? \tau ! \text{Suc } k) = \text{execute-parallel-operator} (? \tau ! k) (\pi ! k)$ 
      using trace-parallel-plan-step-effect-is[OF k-lt-length- $\tau$ -minus-one].
    then have  $(? \tau ! \text{Suc } k) v = (? \tau ! k) v$ 
      using execute-parallel-operator-no-effect-if C
      by fastforce
    hence  $(? \tau ! \text{Suc } k) v \neq \text{Some True}$ 
      using False
      by arg0
  }
  then have  $\neg \mathcal{A} (\text{State } (\text{Suc } k) (\text{index } ?vs v))$ 
    using assms(3) k-lt-length- $\tau$ -minus-one
    by auto
thus  $?thesis$ 
  using C-is
  unfolding clause-semantics-def
  by fastforce
qed
qed
next
  case k-gte-length- $\tau$ -minus-one: False
show  $?thesis$ 
  proof (cases  $\mathcal{A} (\text{State} (\text{length } ?\tau - 1) (\text{index } ?vs v)))$ 
    case True
    {
      have  $\mathcal{A} (\text{State } k (\text{index } ?vs v)) = \mathcal{A} (\text{State} (\text{length } ?\tau - 1) (\text{index } ?vs v))$ 
        proof (cases  $k = \text{length } ?\tau - 1$ )
          case False
          then have  $\text{length } ?\tau \leq k$  and  $k < ?t + 1$ 
            using k-gte-length- $\tau$ -minus-one k-in
            by fastforce+
          thus  $?thesis$ 
            using assms(4)
            by blast
        qed blast
        hence  $\mathcal{A} (\text{State } k (\text{index } ?vs v))$ 
          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 $\Pi \pi$

and $\forall k op. k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1$

$\rightarrow A (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op)) = (op \in \text{set } (\pi ! k))$

and $\forall v k. k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$

$\rightarrow (A (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))$

$\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True})$

and $\forall v l. l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \wedge l < \text{length } \pi + 1$

$\rightarrow A (\text{State } l (\text{index } (\text{strips-problem.variables-of } \Pi) v))$

$= A (\text{State } (\text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$

$(\text{index } (\text{strips-problem.variables-of } \Pi) v))$

and $C \in \bigcup (\bigcup (k, v) \in \{0..<\text{length } \pi\} \times \text{set } ((\Pi)_V).$

$\{\{\{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1}$

$, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \}$

$\cup \{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^+$

$| op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{delete-effects-of } op) \}\}\}$

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 = ( $\bigcup_{(k, v) \in \{0..<?t\} \times \text{set } ?vs}$ .
   $\{\{(State k (index ?vs v))^{-1}, (State (\text{Suc } k) (index ?vs v))^+ \}$ 
   $\cup \{(Operator k (index ?ops op))^+ | op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{delete-effects-of } op)\}\})$ 
 $\{$ 
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' ∈  $\{\{(State k (index ?vs v))^{-1}, (State (\text{Suc } k) (index ?vs v))^+ \}$ 
   $\cup \{(Operator k (index ?ops op))^+ | op. op \in \text{set } ?ops \wedge v \in \text{set } (\text{delete-effects-of } op)\}\}$ 
  using UN-E
  by fastforce
  hence  $\exists k. v.$ 
   $k \in \{0..<?t\}$ 
   $\wedge v \in \text{set } ?vs$ 
   $\wedge C = \{(State k (index ?vs v))^{-1}, (State (\text{Suc } k) (index ?vs v))^+ \}$ 
   $\cup \{(Operator k (index ?ops op))^+ | op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{delete-effects-of } op)\}$ 
  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 =  $\{(State k (index ?vs v))^{-1}, (State (\text{Suc } k) (index ?vs v))^+ \}$ 
   $\cup \{(Operator k (index ?ops op))^+ | op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{delete-effects-of } op)\}$ 
  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)  $\exists op \in \text{set } (\pi ! k). v \in \text{set } (\text{delete-effects-of } op)$ 
     $| (B) \exists op \in \text{set } (\pi ! k). v \in \text{set } (\text{add-effects-of } op)$ 
     $| (C) \forall op \in \text{set } (\pi ! k). v \notin \text{set } (\text{add-effects-of } op) \wedge v \notin \text{set } (\text{delete-effects-of } op)$ 

```

```

 $op)$ 
  by blast
  thus  $?thesis$ 
  proof (cases)
    case A
    moreover obtain  $op$ 
      where  $op\text{-in-}\pi_k$ :  $op \in set(\pi ! k)$ 
      and  $v\text{-is-delete-effect}$ :  $v \in set(delete\text{-effects-of } op)$ 
      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-πk]
      by auto
    }
    ultimately have (Operator k (index ?ops op))+
     $\in \{ (\text{Operator } k (\text{index } ?ops op))^+ \mid op. op \in set ?ops \wedge v \in set (delete\text{-effects-of } op) \}$ 
    using v-is-delete-effect
    by blast
    then have (Operator k (index ?ops op))+  $\in C$ 
    using C-is
    by auto
    moreover have  $\mathcal{A} (\text{Operator } k (\text{index } ?ops op))$ 
    using assms(2) k-lt-length-τ-minus-one op-in-πk
    by blast
    ultimately show  $?thesis$ 
    unfolding clause-semantics-def
    by force
  next
    case B
    then obtain  $op$ 
      where  $op\text{-in-}\pi_k$ :  $op \in set(\pi ! k)$ 
      and  $v\text{-is-add-effect}$ :  $v \in set(add\text{-effects-of } op)..\.$ 
    then have  $\neg(\exists op \in set(\pi ! k). v \in set(delete\text{-effects-of } op))$ 
    using all-operator-effects-consistent are-all-operator-effects-consistent-set
    by fast
    then have execute-parallel-operator (?τ ! k) (π ! k)  $v = \text{Some True}$ 
    using execute-parallel-operator-positive-effect-if[OF all-operators-applicable all-operator-effects-consistent op-in-πk v-is-add-effect]
    by blast
    moreover have (?τ ! Suc k)  $v = \text{execute-parallel-operator} (?\tau ! k) (\pi ! k)$ 
     $v$ 
    using trace-parallel-plan-step-effect-is[OF k-lt-length-τ-minus-one]
    by simp

```

```

ultimately have  $\mathcal{A}(\text{State } (\text{Suc } k) \text{ (index } ?vs\ v))$ 
  using  $\text{assms}(3)$   $k\text{-lt-length-}\tau\text{-minus-one}$ 
  by  $\text{simp}$ 
thus  $?thesis$ 
  using  $C\text{-is}$ 
  unfolding  $\text{clause-semantics-def}$ 
  by  $\text{simp}$ 
next
  case  $C$ 
  show  $?thesis$ 
    — We split on cases for  $(?\tau ! k) v = \text{Some True}$  here to avoid having to
    proof  $(?\tau ! k) v \neq \text{None}$ .
    proof (cases  $(?\tau ! k) v = \text{Some True}$ )
      case  $\text{True}$ 
      {
        have  $(?\tau ! \text{Suc } k) = \text{execute-parallel-operator } (?\tau ! k) (\pi ! k)$ 
          using  $\text{trace-parallel-plan-step-effect-is}[\text{OF } k\text{-lt-length-}\tau\text{-minus-one}]$ .
        then have  $(?\tau ! \text{Suc } k) v = (?\tau ! k) v$ 
          using  $\text{execute-parallel-operator-no-effect-if } C$ 
          by  $\text{fastforce}$ 
        then have  $(?\tau ! \text{Suc } k) v = \text{Some True}$ 
          using  $\text{True}$ 
          by  $\text{arg0}$ 
        hence  $\mathcal{A}(\text{State } (\text{Suc } k) \text{ (index } ?vs\ v))$ 
          using  $\text{assms}(3)$   $k\text{-lt-length-}\tau\text{-minus-one}$ 
          by  $\text{fastforce}$ 
      }
      thus  $?thesis$ 
        using  $C\text{-is}$ 
        unfolding  $\text{clause-semantics-def}$ 
        by  $\text{fastforce}$ 
    next
      case  $\text{False}$ 
      then have  $\neg\mathcal{A}(\text{State } k \text{ (index } ?vs\ v))$ 
        using  $\text{assms}(3)$   $k\text{-lt-length-}\tau\text{-minus-one}$ 
        by  $\text{simp}$ 
      thus  $?thesis$ 
        using  $C\text{-is}$ 
        unfolding  $\text{clause-semantics-def}$ 
        by  $\text{fastforce}$ 
      qed
    qed
  next
  case  $k\text{-gte-length-}\tau\text{-minus-one: False}$ 
  show  $?thesis$ 
  proof (cases  $\mathcal{A}(\text{State } (\text{length } ?\tau - 1) \text{ (index } ?vs\ v))$ )
    case  $\text{True}$ 
    {
      have  $\text{length } ?\tau \leq \text{Suc } k$  and  $\text{Suc } k < ?t + 1$ 

```

```

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 $\forall k \text{ op. } k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1 \rightarrow A (\text{Operator } k (\text{index} (\text{strips-problem.operators-of } \Pi) \text{ op})) = (\text{op} \in \text{set} (\pi ! k))$

and $\forall v \text{ k. } k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \rightarrow (A (\text{State } k (\text{index} (\text{strips-problem.variables-of } \Pi) v)))$

```

 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$ 
and  $\forall v l. l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \wedge l < \text{length } \pi + 1$ 
 $\rightarrow \mathcal{A} (\text{State } l (\text{index } (\text{strips-problem.variables-of } \Pi) v))$ 
 $= \mathcal{A} (\text{State}$ 
 $(\text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$ 
 $(\text{index } (\text{strips-problem.variables-of } \Pi) v))$ 
shows  $\mathcal{A} \models \text{encode-all-frame-axioms } \Pi (\text{length } \pi)$ 
proof -
let  $?F = \text{encode-all-frame-axioms } \Pi (\text{length } \pi)$ 
let  $?vs = \text{strips-problem.variables-of } \Pi$ 
and  $?ops = \text{strips-problemoperators-of } \Pi$ 
and  $?t = \text{length } \pi$ 
let  $?A = \bigcup (\bigcup (k, v) \in \{0..<?t\} \times \text{set } ((\Pi)_V).$ 
 $\{\{\{ (\text{State } k (\text{index } ?vs v))^+, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^{-1} \}$ 
 $\cup \{ (\text{Operator } k (\text{index } ?ops op))^+$ 
 $| op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{add-effects-of } op) \}\}\}$ 
and  $?B = \bigcup (\bigcup (k, v) \in \{0..<?t\} \times \text{set } ((\Pi)_V).$ 
 $\{\{\{ (\text{State } k (\text{index } ?vs v))^{-1}, (\text{State } (\text{Suc } k) (\text{index } ?vs v))^+ \}$ 
 $\cup \{ (\text{Operator } k (\text{index } ?ops op))^+$ 
 $| op. op \in \text{set } ((\Pi)_O) \wedge v \in \text{set } (\text{delete-effects-of } op) \}\}\}$ 

have  $\text{cnf-}\Phi_F\text{-is-}A\text{-union-}B$ :  $\text{cnf } ?F = ?A \cup ?B$ 
using  $\text{cnf-of-encode-all-frame-axioms-structure}$ 
by (simp add: cnf-of-encode-all-frame-axioms-structure)
{
  fix  $C$ 
  assume  $C \in \text{cnf } ?F$ 
  then consider ( $C\text{-in-}A$ )  $C \in ?A$ 
  | ( $C\text{-in-}B$ )  $C \in ?B$ 
  using  $\text{Un-iff}[of C ?A ?B] \text{ cnf-}\Phi_F\text{-is-}A\text{-union-}B$ 
  by argo
  hence clause-semantics  $\mathcal{A} C$ 
  proof (cases)
    case  $C\text{-in-}A$ 
    then show  $?thesis$ 
    using  $\text{encode-problem-parallel-complete-iv-a}[OF \text{assms}(2, 3, 4, 5) C\text{-in-}A]$ 
    by blast
  next
    case  $C\text{-in-}B$ 
    then show  $?thesis$ 
    using  $\text{encode-problem-parallel-complete-iv-b}[OF \text{assms}(2, 3, 4, 5) C\text{-in-}B]$ 
    by blast
  qed
}
thus  $?thesis$ 
using  $\text{encode-frame-axioms-is-cnf is-nnf-cnf cnf-semantics}$ 
unfolding  $\text{cnf-semantics-def}$ 
by blast
qed

```

```

lemma valuation-for-operator-variables-is:
  fixes  $\Pi :: 'a \text{ strips-problem}$ 
  assumes is-parallel-solution-for-problem  $\Pi \pi$ 
    and  $k < \text{length}(\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1$ 
    and  $op \in \text{set } ((\Pi)_O)$ 
  shows valuation-for-operator-variables  $\Pi \pi (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
     $(\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))$ 
     $= (\text{op} \in \text{set } (\pi ! k))$ 
  proof -
    let ?ops = strips-problem.operators-of  $\Pi$ 
    and ? $\tau$  = trace-parallel-plan-strips  $((\Pi)_I) \pi$ 
    let ?v = Operator k (index ?ops op)
    and ?Op = { Operator k (index ?ops op)
      |  $k \text{ op. } k \in \{0..<\text{length } ?\tau - 1\} \wedge op \in \text{set } ((\Pi)_O) \}$ 
    let ?l = concat (map ( $\lambda k. \text{map } (\text{Pair } k) (\pi ! k)$ )  $[0..<\text{length } ?\tau - 1]$ )
    and ?f =  $\lambda x. \text{Operator } (\text{fst } x) (\text{index } ?ops (\text{snd } x))$ 
    — show that our operator construction function is injective on set (concat (map
    ( $\lambda k. \text{map } (\text{Pair } k) (\pi ! k)$ )  $[0..<\text{length } ?\tau - 1]$ )).
    have k-in:  $k \in \{0..<\text{length } ?\tau - 1\}$ 
    using assms(2)
    by fastforce
  {
    {
      fix k k' op op'
      assume k-op-in:  $(k, op) \in \text{set } ?l$  and k'-op'-in:  $(k', op') \in \text{set } ?l$ 
      have Operator k (index ?ops op) = Operator k' (index ?ops op')  $\longleftrightarrow (k, op) = (k', op')$ 
      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:  $k' < \text{length } ?\tau - 1$ 
          using k'-op'-in
          by fastforce
        }
        have op-in:  $op \in \text{set } (\pi ! k)$ 
        using k-op-in
        by force
        then have op'-in:  $op' \in \text{set } (\pi ! k)$ 
        using k'-op'-in k-is-k'
        by auto
      {
        have length-tau-gt-1:  $\text{length } ?\tau > 1$ 
      }
    }
  }

```

```

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 =  $\bigcup((\lambda(k, op). \{ (k, op) \})$ 
  ‘ { (k, op). k ∈ {0.. $<\text{length } \pi - 1$ } ∧ op ∈ set ( $\pi ! k$ ) }
    using setcompr-eq-image[of  $\lambda(k, op). \{ (k, op) \}$  -]
    by auto
} note set-l-is = this
{
  have Operator k (index ?ops op) ∈ ?Op
    using assms(3) k-in
    by blast

hence valuation-for-operator-variables II π ?τ ?v
  = foldr ( $\lambda(k, op)$  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 ( $\pi ! k$ ))
  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 =  $\lambda\_. \text{True}$ 
      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 ( $\lambda(k, op)$  A. A(Operator k (index ?ops op) := True))
        = ( $\lambda x$  A. A(Operator (fst x) (index ?ops (snd x)) := True))
        by fastforce
      moreover have foldr ( $\lambda x$  A. A(Operator (fst x) (index ?ops (snd x)) := ?g x))
        ?l A₀ (Operator (fst (k, op)) (index ?ops (snd (k, op)))) = True
        unfolding foldr-fun-upd[OF inj-on-f-set-l k-op-in]..
      ultimately have valuation-for-operator-variables II π ?τ ?v = True
        using nb
        by argo
    }
    thus ?thesis
      using True
      by blast
  next
    case False
    {
      have (k, op) ∉ set ?l
        using False set-l-is
        by fast
      moreover {

```

```

fix k' op'
assume (k', op') ∈ set ?l
  and ?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)₁) 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)₁) 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 ?Aπ ?v = ?AO ?v
      unfolding valuation-for-plan-def override-on-def Let-def
      by force
  }
  then show ?thesis
    using valuation-for-operator-variables-is[OF assms(1, 2) True]
    by blast
next

  case False
  {
    {
      — We have  $\neg \text{index } ?\text{ops } op < \text{length } ?\text{ops}$  due to the assumption that  $\neg op \in \text{set } ?\text{ops}$ . Hence  $\neg k \in \{0..<?t\}$  and therefore  $?v \notin ?Op$ .
      have ?Op = ( $\lambda(k, op)$ . Operator k (index ?ops op)) ‘( $\{0..<?t\} \times \text{set } ?\text{ops}$ )
        by fast
      moreover have  $\neg \text{index } ?\text{ops } op < \text{length } ?\text{ops}$ 
        using False
        by simp
      ultimately have ?v  $\notin ?Op$ 
        by fastforce
    }
    moreover have ?v  $\notin ?V$ 
    by force

    ultimately have ?Aπ ?v = A0 ?v
      unfolding valuation-for-plan-def override-on-def
      by metis
    hence  $\neg ?A_\pi ?v$ 
      unfolding empty-valuation-def
      by blast
}

```

```

}
moreover have  $(\pi ! k) \in \text{set } \pi$ 
  using k-lt-length-π
  by simp
moreover have  $op \notin \text{set } (\pi ! k)$ 
  using is-parallel-solution-for-problem-operator-set[OF assms(1) calculation(2)] False
  by blast
ultimately show ?thesis
  by blast
qed
qed

```

```

lemma encode-problem-parallel-complete-vi-b:
fixes  $\Pi :: 'a \text{ strips-problem}$ 
assumes is-parallel-solution-for-problem  $\Pi \pi$ 
  and  $l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1$ 
  and  $l < \text{length } \pi$ 
shows  $\neg \text{valuation-for-plan } \Pi \pi (\text{Operator } l (\text{index } (\text{strips-problem.operators-of } \Pi) op))$ 
proof -
let ?vs = strips-problem.variables-of  $\Pi$ 
  and ?ops = strips-problem.operators-of  $\Pi$ 
  and ?t = length  $\pi$ 
  and ?τ = trace-parallel-plan-strips  $((\Pi)_I) \pi$ 
let ?Aπ = valuation-for-plan  $\Pi \pi$ 
  and ?AO = valuation-for-operator-variables  $\Pi \pi ?\tau$ 
  and ?Op = { Operator k (index ?ops op) | k op.  $k \in \{0..<?t\} \wedge op \in \text{set } ?ops$  }
  and ?Op' = { Operator k (index ?ops op) | k op.  $k \in \{0..<\text{length } ?\tau - 1\} \wedge op \in \text{set } ?ops$  }
  and ?V = { State k (index ?vs v) | k v.  $k \in \{0..<?t + 1\} \wedge v \in \text{set } ?vs$  }
  and ?v = Operator l (index ?ops op)
show ?thesis
proof (cases op ∈ set  $((\Pi)_O))$ 
  case True
  {
    {
      have ?v ∈ ?Op
        using assms(3) True
        by auto
      hence ?Aπ ?v = ?AO ?v
        unfolding valuation-for-plan-def override-on-def Let-def
        by simp
    }
    moreover {

```

```

have  $l \notin \{0..<\text{length } ?\tau - 1\}$ 
  using assms(2)
  by simp
then have  $?v \notin ?Op'$ 
  by blast
hence  $?A_O ?v = A_0 ?v$ 
  unfolding valuation-for-operator-variables-def override-on-def
  by meson
}
ultimately have  $\neg ?A_\pi ?v$ 
  unfolding empty-valuation-def
  by blast
}
then show ?thesis
  by blast
next

case False
{
{
— We have  $\neg \text{index } ?ops op < \text{length } ?ops$  due to the assumption that  $\neg op \in \text{set } ?ops$ . Hence  $\neg k \in \{0..<?t\}$  and therefore  $?v \notin ?Op$ .
have  $?Op = (\lambda(k, op). \text{Operator } k (\text{index } ?ops op))`(\{0..<?t\} \times \text{set } ?ops)$ 
  by fast
moreover have  $\neg \text{index } ?ops op < \text{length } ?ops$ 
  using False
  by simp
ultimately have  $?v \notin ?Op$ 
  by fastforce
}
moreover have  $?v \notin ?V$ 
  by force

ultimately have  $?A_\pi ?v = A_0 ?v$ 
  unfolding valuation-for-plan-def override-on-def
  by metis
hence  $\neg ?A_\pi ?v$ 
  unfolding empty-valuation-def
  by blast
}
thus ?thesis
  by blast
qed
qed

```

— As a corollary from lemmas and we obtain the result that the constructed valuation $A \equiv \text{valuation-for-plan } \Pi \pi$ evaluates SATPlan operator variables as false if they are not contained in any operator set $\pi ! k$ for any time point $k < \text{length } \pi$.

corollary *encode-problem-parallel-complete-vi-d*:

```

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

and $v \in \text{set } vs$

shows $\text{foldr}(\lambda(k, v). A. \text{valuation-for-state } vs(s k) k v A) (\text{List.product } ks vs)$

A_0

$(\text{State } k (\text{index } vs v))$

$\longleftrightarrow (s k) v = \text{Some True}$

proof —

let $?v = \text{State } k (\text{index } vs v)$

and $?ps = \text{List.product } ks vs$

let $?A = \text{foldr}(\lambda(k, v). A. \text{valuation-for-state } vs(s k) k v A) ?ps A_0$

and $?f = \lambda x. \text{State}(\text{fst } x) (\text{index } vs (\text{snd } x))$

and $?g = \lambda x. (s(\text{fst } x)) (\text{snd } x) = \text{Some True}$

have $nb_1: (k, v) \in \text{set } ?ps$

using $\text{assms}(1, 2)$ *set-product*

by *simp*

moreover {

{

fix $x y$

assume $x\text{-in-}ps: x \in \text{set } ?ps$ **and** $y\text{-in-}ps: y \in \text{set } ?ps$

and $\neg(?f x = ?f y \rightarrow x = y)$

then have $f\text{-}x\text{-is-}f\text{-}y: ?f x = ?f y$ **and** $x\text{-is-not-}y: x \neq y$

by *blast+*

then obtain $k' k'' v' v''$

where $x\text{-is: } x = (k', v')$

and $y\text{-is: } y = (k'', v'')$

by *fastforce*

then consider $(A) k' \neq k''$

| $(B) v' \neq v''$

using $x\text{-is-not-}y$

by *blast*

hence *False*

proof (*cases*)

case *A*

then have $?f x \neq ?f y$

using $x\text{-is } y\text{-is}$

by *simp*

thus *?thesis*

using $f\text{-}x\text{-is-}f\text{-}y$

by *argo*

next

case *B*

have $v' \in \text{set } vs$ **and** $v'' \in \text{set } vs$

```

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 ?Aπ = valuation-for-plan Π π

```

```

and  $\mathcal{A}_V = \text{valuation-for-state-variables } \Pi \pi \ ?\tau$ 
and  $\mathcal{A}_O = \text{valuation-for-state-variables } \Pi \pi \ ?\tau$ 
and  $\mathcal{A}_1 = \text{foldr}$ 
 $(\lambda(k, v) \mathcal{A}. \text{valuation-for-state } ?vs (\ ?\tau ! k) k v \mathcal{A})$ 
 $(\text{List.product } [0..<?t'] ?vs) \mathcal{A}_0$ 
and  $?Op = \{ \text{Operator } k (\text{index } ?ops op) \mid k \text{ op. } k \in \{0..<?t\} \wedge op \in \text{set } ((\Pi)_\mathcal{O}) \}$ 
}
and  $?Op' = \{ \text{Operator } k (\text{index } ?ops op) \mid k \text{ op. } k \in \{0..<?t' - 1\} \wedge op \in \text{set } ((\Pi)_\mathcal{O}) \}$ 
and  $?V = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{0..<?t + 1\} \wedge v \in \text{set } ((\Pi)_\mathcal{V}) \}$ 
and  $?V_1 = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{0..<?t'\} \wedge v \in \text{set } ((\Pi)_\mathcal{V}) \}$ 
and  $?V_2 = \{ \text{State } k (\text{index } ?vs v) \mid k \text{ v. } k \in \{?t'..(?t + 1)\} \wedge v \in \text{set } ((\Pi)_\mathcal{V}) \}$ 
}
and  $?v = \text{State } k (\text{index } ?vs v)$ 
have  $v\text{-notin-}Op: ?v \notin ?Op$ 
by blast
have  $k\text{-lte-length-}\pi\text{-plus-one}: k < \text{length } \pi + 1$ 
using less-le-trans length-trace-parallel-plan-strips-lte-length-plan-plus-one assms(3)
by blast
show ?thesis
proof (cases  $v \in \text{set } ((\Pi)_\mathcal{V})$ )
case True
{
{
have  $?v \in ?V ?v \notin ?Op$ 
using k-lte-length- $\pi$ -plus-one True
by force+
hence  $?A_\pi ?v = ?A_V ?v$ 
unfolding valuation-for-plan-def override-on-def Let-def
by simp
}
moreover {
have  $?v \in ?V_1 ?v \notin ?V_2$ 
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  $?A_\pi ?v = ?A_1 ?v$ 
by blast
}
moreover have  $k \in \text{set } [0..<?t']$ 
using assms(3)
by simp
moreover have  $v \in \text{set } (\text{strips-problem.variables-of } \Pi)$ 
using True
by simp
}

```

```

ultimately show ?thesis
  using valuation-for-state-variables-is[of k [0..<?t']]
  by fastforce
next
  case False
  {
    {
      have  $\neg \text{index } ?vs v < \text{length } ?vs$ 
        using False index-less-size-conv
        by simp
      hence  $?v \notin ?V$ 
        by fastforce
    }
    then have  $\neg ?A_\pi ?v$ 
      using v-notin-Op
    unfolding valuation-for-plan-def override-on-def empty-valuation-def Let-def
      variables-of-def operators-of-def
      by presburger
    }
    moreover have  $\neg(\tau ! k) v = \text{Some True}$ 
      using trace-parallel-plan-strips-none-if[of  $\Pi \pi k v$ ] 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  $\Pi :: \text{'a strips-problem}$ 
  assumes is-valid-problem-strips  $\Pi$ 
    and is-parallel-solution-for-problem  $\Pi \pi$ 
    and  $l \geq \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
    and  $l < \text{length } \pi + 1$ 
  shows valuation-for-plan  $\Pi \pi (\text{State } l (\text{index } (\text{strips-problem.variables-of } \Pi) v))$ 
     $= \text{valuation-for-plan } \Pi \pi$ 
     $(\text{State } (\text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$ 
     $(\text{index } (\text{strips-problem.variables-of } \Pi) v))$ 

```

proof –

```

let  $?vs = \text{strips-problem.variables-of } \Pi$ 
and  $?ops = \text{strips-problem.operators-of } \Pi$ 
and  $?t = \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$ 
let  $?t' = \text{length } \pi$ 
and  $?t' = \text{length } ?t$ 
let  $?t_\Omega = ?t ! (?t' - 1)$ 
and  $?A_\pi = \text{valuation-for-plan } \Pi \pi$ 

```

```

and ? $\mathcal{A}_V$  = valuation-for-state-variables  $\Pi \pi \ ?\tau$ 
and ? $\mathcal{A}_O$  = valuation-for-state-variables  $\Pi \pi \ ?\tau$ 
let ? $\mathcal{A}_2$  = foldr
  ( $\lambda(k, v) \mathcal{A}.$  valuation-for-state (strips-problem.variables-of  $\Pi$ )  $??\tau_\Omega k v \mathcal{A}$ )
  ( $List.product [?t'..<length \pi + 2] ?vs$ )
   $\mathcal{A}_0$ 
and ? $Op$  = { Operator  $k$  (index ?ops op) |  $k$  op.  $k \in \{0..<?t\} \wedge op \in set ((\Pi)_\mathcal{O})$  }
and ? $Op'$  = { Operator  $k$  (index ?ops op) |  $k$  op.  $k \in \{0..<?t' - 1\} \wedge op \in set ((\Pi)_\mathcal{O})$  }
and ? $V$  = { State  $k$  (index ?vs v) |  $k$  v.  $k \in \{0..<?t + 1\} \wedge v \in set ((\Pi)_V)$  }
and ? $V_1$  = { State  $k$  (index ?vs v) |  $k$  v.  $k \in \{0..<?t'\} \wedge v \in set ((\Pi)_V)$  }
and ? $V_2$  = { State  $k$  (index ?vs v) |  $k$  v.  $k \in \{?t'..(?t + 1)\} \wedge v \in set ((\Pi)_V)$  }
}
and ? $v$  = State  $l$  (index ?vs v)
have v-notin- $Op$ : ? $v \notin ?Op$ 
  by blast
show ?thesis
  proof (cases  $v \in set ((\Pi)_V)$ )
    case True
    {
      {
        have ? $v \in ?V ?v \notin ?Op$ 
          using assms(4) True
          by force+
        hence ? $\mathcal{A}_\pi ?v = ?\mathcal{A}_V ?v$ 
          unfolding valuation-for-plan-def override-on-def Let-def
          by simp
      }
      moreover {
        have ? $v \notin ?V_1 ?v \in ?V_2$ 
          using assms(3, 4) True
          by force+
        hence ? $\mathcal{A}_V ?v = ?\mathcal{A}_2 ?v$ 
          unfolding valuation-for-state-variables-def override-on-def Let-def
          by auto
      }
      ultimately have ? $\mathcal{A}_\pi ?v = ?\mathcal{A}_2 ?v$ 
        by blast
    } note nb = this
    moreover
    {
      have  $l \in set [?t'..<?t + 2]$ 
        using assms(3, 4)
        by auto
    }
  }

```

```

hence ?A2 ?v  $\longleftrightarrow$  ?τΩ v = Some True
  using valuation-for-state-variables-is[of l [?t'..<?t + 2]] True nb
  by fastforce
}
ultimately have ?Aπ ?v  $\longleftrightarrow$  ?τΩ 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 ¬?Aπ ?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 $\tau \equiv \text{trace-parallel-plan-strips } I \ \pi$ be the trace of the plan π , $t \equiv \text{length } \pi$, and $t' \equiv \text{length } \tau$.

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

1. for all k and for all op with $k < t' - 1$

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

2. for all l and for all op with $t' - 1 \leq l$ and $l < \text{length } \pi$ we require

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

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

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

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

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

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 $k < t' - 1$ 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 $\mathcal{A}(\text{State } k (\text{index}(\Pi_V) v))$ for the SAT plan variable of every state variable v at time point k if and only if $(\tau ! k) v = \text{Some True}$ for the corresponding state $\tau ! k$ at time k (and $\neg \mathcal{A}(\text{State } k (\text{index}(\Pi_V) v))$ otherwise).

The second respectively fourth condition cover early plan termination by negating operator activation and propagating the last reached state. Note that in the state propagation constraint, the index is incremented by one compared to the similar constraint for operators, since operator activations are always followed by at least one successor state. Hence the last state in the trace has index $\text{length}(\text{trace-parallel-plan-strips } (\Pi_I) \pi) - 1$ and the remaining states take up the indexes to $\text{length } \pi + 1$.

value stop

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

— To show completeness—i.e. every valid parallel plan π corresponds to a model for the SATPlan encoding $\Phi \Pi$ ($\text{length } \pi$)—, we simply split the conjunction defined by the encoding into partial encodings and show that the model satisfies each of them.

theorem

```

encode-problem-parallel-complete:
assumes is-valid-problem-strips  $\Pi$ 
    and is-parallel-solution-for-problem  $\Pi \pi$ 
shows valuation-for-plan  $\Pi \pi \models \Phi \Pi (\text{length } \pi)$ 

proof —
  let  $?t = \text{length } \pi$ 
  and  $?I = (\Pi)_I$ 
  and  $?G = (\Pi)_G$ 
  and  $?A = \text{valuation-for-plan } \Pi \pi$ 
  have  $nb: ?G \subseteq_m \text{execute-parallel-plan } ?I \pi$ 
    using assms(2)
    unfolding is-parallel-solution-for-problem-def
    by force
  have  $?A \models \Phi_I \Pi$ 
    using encode-problem-parallel-complete-i[OF assms(1) nb]
      encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
    by presburger
  moreover have  $?A \models (\Phi_G \Pi) ?t$ 
    using encode-problem-parallel-complete-ii[OF assms(1) nb]
      encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
      encode-problem-parallel-complete-vi-f[OF assms(1, 2)]
    by presburger
  moreover have  $?A \models \text{encode-operators } \Pi ?t$ 
    using encode-problem-parallel-complete-iii[OF assms(1) nb]
      encode-problem-parallel-complete-vi-a[OF assms(2)]
      encode-problem-parallel-complete-vi-b[OF assms(2)]
      encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
    by presburger
  moreover have  $?A \models \text{encode-all-frame-axioms } \Pi ?t$ 
    using encode-problem-parallel-complete-iv[OF assms(1, 2)]
      encode-problem-parallel-complete-vi-a[OF assms(2)]
      encode-problem-parallel-complete-vi-c[OF assms(1, 2)]
      encode-problem-parallel-complete-vi-f[OF assms(1, 2)]
    by presburger
  ultimately show  $?thesis$ 
    unfolding encode-problem-def SAT-Plan-Base.encode-problem-def
      encode-initial-state-def encode-goal-state-def
    by auto
qed

end

theory SAT-Plan-Extensions
imports SAT-Plan-Base

```

begin

8 Serializable SATPlan Encodings

A SATPlan encoding with exclusion of operator interference (see definition ??) can be defined by extending the basic SATPlan encoding with clauses

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

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 op1 op2
≡ let ops = operators-of Π in
    ¬(Atom (Operator k (index ops op1)))
    ∨ ¬(Atom (Operator k (index ops op2)))

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(op1, op2). index ops op1 ≠ index ops op2
        ∧ are-operators-interfering op1 op2) (List.product ops ops)
    in foldr (Λ) [encode-interfering-operator-pair-exclusion Π k op1 op2.
        (op1, op2) ← 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* Πt to the basic SATPlan encoding.

```
definition encode-problem-with-operator-interference-exclusion
:: 'variable strips-problem ⇒ nat ⇒ sat-plan-variable formula
(Φ∨ - -> 52)
where encode-problem-with-operator-interference-exclusion Π t
≡ encode-initial-state Π
    ∧ (encode-operators Π t
    ∧ (encode-all-frame-axioms Π t
    ∧ (encode-interfering-operator-exclusion Π t
    ∧ (encode-goal-state Π t))))
```

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

```

lemma cnf-of-encode-interfering-operator-pair-exclusion-is-i[simp]:
  cnf (encode-interfering-operator-pair-exclusion Π k op1 op2) = {{}
    (Operator k (index (strips-problem.operators-of Π) op1))-1
    , (Operator k (index (strips-problem.operators-of Π) op2))-1 {}}

proof -
  let ?ops = strips-problem.operators-of Π
  have cnf (encode-interfering-operator-pair-exclusion Π k op1 op2)
    = cnf (¬(Atom (Operator k (index ?ops op1))) ∨ ¬(Atom (Operator k (index
    ?ops op2))))
  unfolding encode-interfering-operator-pair-exclusion-def
  by metis
  also have ... = { C ∪ D | C D.
    C ∈ cnf (¬(Atom (Operator k (index ?ops op1))))
    ∧ D ∈ cnf (¬(Atom (Operator k (index ?ops op2)))) }
  by simp
  finally show ?thesis
  by auto
qed

lemma cnf-of-encode-interfering-operator-exclusion-is-ii[simp]:
  set [encode-interfering-operator-pair-exclusion Π k op1 op2.
  (op1, op2) ← filter (λ(op1, op2).
    index (strips-problem.operators-of Π) op1 ≠ index (strips-problem.operators-of
  Π) op2
    ∧ are-operators-interfering op1 op2)
    (List.product (strips-problem.operators-of Π) (strips-problem.operators-of
  Π))
    , k ← [0..<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 }.
    (λk. encode-interfering-operator-pair-exclusion Π k op1 op2) ` {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 set ?fs = ⋃(set
    ` (λ(op1, op2). map (λk. encode-interfering-operator-pair-exclusion Π k op1 op2)
  [0..<t])`)

```

```

‘(set (filter (λ(op1, op2). index ?ops op1 ≠ index ?ops op2 ∧ are-operators-interfering
op1 op2)
(List.product ?ops ?ops))))
unfolding set-concat set-map
by blast
— TODO slow.
also have ... = ∪ ((λ(op1, op2).
set (map (λk. encode-interfering-operator-pair-exclusion Π k op1 op2) [0..<t]))
‘(set (filter (λ(op1, op2). index ?ops op1 ≠ index ?ops op2 ∧ are-operators-interfering
op1 op2)
(List.product ?ops ?ops))))
unfolding image-comp[of
set λ(op1, op2). map (λk. encode-interfering-operator-pair-exclusion Π k op1
op2) [0..<t]]
comp-apply
by fast
also have ... = ∪ ((λ(op1, op2).
(λk. encode-interfering-operator-pair-exclusion Π k op1 op2) ‘{0..<t})
‘(set (filter (λ(op1, op2). index ?ops op1 ≠ index ?ops op2 ∧ are-operators-interfering
op1 op2)
(List.product ?ops ?ops))))
unfolding set-map[of - [0..<t]] atLeastLessThan-up[of 0 t]
by blast
also have ... = ∪ ((λ(op1, op2).
(λk. encode-interfering-operator-pair-exclusion Π k op1 op2) ‘{0..<t})
‘(Set.filter (λ(op1, op2). index ?ops op1 ≠ index ?ops op2 ∧ are-operators-interfering
op1 op2)
(set (List.product ?ops ?ops))))
unfolding set-filter[of λ(op1, op2). are-operators-interfering op1 op2 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 op1 op2.
(op1, op2) ← filter (λ(op1, op2).
index (strips-problem.operators-of Π) op1 ≠ index (strips-problem.operators-of
Π) op2
∧ are-operators-interfering op1 op2)
(List.product (strips-problem.operators-of Π) (strips-problem.operators-of
Π))
, k ← [0..<t]]]

```

```

= ( $\bigcup$  ( $op_1, op_2$ )
 $\in \{ (op_1, op_2) \in set(strips\text{-}problem.\text{operators}\text{-}of \Pi) \times set(strips\text{-}problem.\text{operators}\text{-}of \Pi).$ 
 $index(strips\text{-}problem.\text{operators}\text{-}of \Pi) op_1 \neq index(strips\text{-}problem.\text{operators}\text{-}of \Pi) op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2 \}.$ 
 $\{\{\{ (Operator k (index(strips\text{-}problem.\text{operators}\text{-}of \Pi) op_1))^{-1}$ 
 $, (Operator k (index(strips\text{-}problem.\text{operators}\text{-}of \Pi) op_2))^{-1} \} \} \mid k. k \in$ 
 $\{0..<t\}\})$ 
proof –
  let ?ops = strips\text{-}problem.\text{operators}\text{-}of \Pi
  let ?interfering = filter ( $\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2$ ) (List.product ?ops ?ops)
  let ?fs = [encode-interfering-operator-pair-exclusion \Pi k op_1 op_2.
    ( $op_1, op_2) \leftarrow ?interfering, k \leftarrow [0..<t]$ ]
  have cnf ‘set ?fs = cnf ‘( $\bigcup (op_1, op_2) \in \{ (op_1, op_2).$ 
 $(op_1, op_2) \in set(operators\text{-}of \Pi) \times set(operators\text{-}of \Pi) \wedge index ?ops op_1 \neq$ 
 $index ?ops op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2 \}.$ 
 $(\lambda k. encode\text{-}interfering\text{-}operator\text{-}pair\text{-}exclusion \Pi k op_1 op_2) ‘\{0..<t\}$ )
unfolding cnf-of-encode-interfering-operator-exclusion-is-ii
by blast
also have ... = ( $\bigcup (op_1, op_2) \in \{ (op_1, op_2).$ 
 $(op_1, op_2) \in set(operators\text{-}of \Pi) \times set(operators\text{-}of \Pi) \wedge index ?ops op_1 \neq$ 
 $index ?ops op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2 \}.$ 
 $(\lambda k. cnf (encode\text{-}interfering\text{-}operator\text{-}pair\text{-}exclusion \Pi k op_1 op_2)) ‘\{0..<t\}$ )
unfolding image-Un image-comp comp-apply
by blast
also have ... = ( $\bigcup (op_1, op_2) \in \{ (op_1, op_2).$ 
 $(op_1, op_2) \in set(operators\text{-}of \Pi) \times set(operators\text{-}of \Pi) \wedge index ?ops op_1 \neq$ 
 $index ?ops op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2 \}.$ 
 $(\lambda k. \{\{ (Operator k (index ?ops op_1))^{-1}, (Operator k (index ?ops op_2))^{-1} \} \})$ 
‘ $\{0..<t\}$ )
by simp
also have ... = ( $\bigcup (op_1, op_2) \in \{ (op_1, op_2).$ 
 $(op_1, op_2) \in set(operators\text{-}of \Pi) \times set(operators\text{-}of \Pi) \wedge index ?ops op_1 \neq$ 
 $index ?ops op_2$ 
 $\wedge are\text{-}operators\text{-}interfering op_1 op_2 \}.$ 
 $(\lambda k. \{\{ (Operator k (index ?ops op_1))^{-1}, (Operator k (index ?ops op_2))^{-1} \} \})$ 
‘ $\{ k \mid k. k \in \{0..<t\}\}$ )
by blast
— TODO slow.
finally show ?thesis
unfolding operators-of-def setcompr-eq-image[ $of - \lambda k. k \in \{0..<t\}$ ]
by force
qed

```

```

lemma cnf-of-encode-interfering-operator-exclusion-is:
  cnf (encode-interfering-operator-exclusion  $\Pi$   $t$ ) =  $\bigcup (\bigcup (op_1, op_2)$ 
     $\in \{ (op_1, op_2) \in set (operators-of \Pi) \times set (operators-of \Pi).$ 
     $index (strips-problem.operators-of \Pi) op_1 \neq index (strips-problem.operators-of$ 
 $\Pi) op_2$ 
     $\wedge are-operators-interfering op_1 op_2 \}.$ 
   $\{ \{ (Operator k (index (strips-problem.operators-of \Pi) op_1))^{-1}$ 
     $, (Operator k (index (strips-problem.operators-of \Pi) op_2))^{-1} \} \} \mid k. k \in$ 
   $\{0..<t\}\}$ 
proof -
  let ?ops = strips-problem.operators-of  $\Pi$ 
  let ?interfering = filter ( $\lambda(op_1, op_2). index ?ops op_1 \neq index ?ops op_2$ 
     $\wedge are-operators-interfering op_1 op_2$ ) (List.product ?ops ?ops)
  let ?fs = [encode-interfering-operator-pair-exclusion  $\Pi$   $k op_1 op_2.$ 
     $(op_1, op_2) \leftarrow ?interfering, k \leftarrow [0..<t]$ ]
  have cnf (encode-interfering-operator-exclusion  $\Pi$   $t$ ) = cnf (foldr ( $\wedge$ ) ?fs ( $\neg\perp$ ))
    unfolding encode-interfering-operator-exclusion-def
    by metis
  also have ... =  $\bigcup (cnf ` set ?fs)$ 
    unfolding cnf-foldr-and[of ?fs]..
  finally show ?thesis
    unfolding cnf-of-encode-interfering-operator-exclusion-is-iii[of  $\Pi t$ ]
    by blast
qed

```

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

```

fixes  $\Pi :: 'variable strips-problem$ 
assumes  $k < t$ 
and  $op_1 \in set (strips-problem.operators-of \Pi)$  and  $op_2 \in set (strips-problem.operators-of$ 
 $\Pi)$ 
and  $index (strips-problem.operators-of \Pi) op_1 \neq index (strips-problem.operators-of$ 
 $\Pi) op_2$ 
and  $are-operators-interfering op_1 op_2$ 
shows  $\{ (Operator k (index (strips-problem.operators-of \Pi) op_1))^{-1}$ 
   $, (Operator k (index (strips-problem.operators-of \Pi) op_2))^{-1} \}$ 
   $\in cnf (encode-interfering-operator-exclusion \Pi t)$ 
proof -
  let ?ops = strips-problem.operators-of  $\Pi$ 
  and ? $\Phi_X$  = encode-interfering-operator-exclusion  $\Pi t$ 
  let ?Ops =  $\{ (op_1, op_2) \in set (operators-of \Pi) \times set (operators-of \Pi).$ 
     $index ?ops op_1 \neq index ?ops op_2 \wedge are-operators-interfering op_1 op_2 \}$ 
  and ?f =  $\lambda(op_1, op_2). \{ \{ (Operator k (index ?ops op_1))^{-1}, (Operator k (index$ 
 $?ops op_2))^{-1} \} \}$ 
  |  $k. k \in \{0..<t\}\}$ 
  let ?A =  $(\bigcup (op_1, op_2) \in ?Ops. ?f (op_1, op_2))$ 
  let ?B =  $\bigcup ?A$ 
  and ?C =  $\{ (Operator k (index ?ops op_1))^{-1}, (Operator k (index ?ops op_2))^{-1}$ 
}

```

```

{
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 op1 op2
  where (op1, op2) ∈ set (operators-of Π) × set (operators-of Π)
  and are-operators-interfering op1 op2
  and f ∈ (λk. encode-interfering-operator-pair-exclusion Π k op1 op2) ` {0..<t}
  by fast
then obtain k where f = encode-interfering-operator-pair-exclusion Π k op1
  op2

```

```

    by blast
  then have  $f = \neg(\text{Atom} (\text{Operator } k (\text{index } ?ops op_1))) \vee \neg(\text{Atom} (\text{Operator } k (\text{index } ?ops op_2)))$ 
    unfolding encode-interfering-operator-pair-exclusion-def
    by metis
  hence  $\text{is-cnf } f$ 
    by force
  }
thus  $?thesis$ 
  unfolding encode-interfering-operator-exclusion-def
  using is-cnf-foldr-and-if[of ?fs]
  by meson
qed

lemma is-cnf-encode-problem-with-operator-interference-exclusion:
assumes is-valid-problem-strips  $\Pi$ 
shows is-cnf  $(\Phi_{\forall} \Pi t)$ 
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  $(\Phi_I \Pi) \subseteq \text{cnf} (\Phi_{\forall} \Pi t)$ 
and cnf  $((\Phi_G \Pi) t) \subseteq \text{cnf} (\Phi_{\forall} \Pi t)$ 
and cnf  $(\text{encode-operators } \Pi t) \subseteq \text{cnf} (\Phi_{\forall} \Pi t)$ 
and cnf  $(\text{encode-all-frame-axioms } \Pi t) \subseteq \text{cnf} (\Phi_{\forall} \Pi t)$ 
and cnf  $(\text{encode-interfering-operator-exclusion } \Pi t) \subseteq \text{cnf} (\Phi_{\forall} \Pi t)$ 
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  $\mathcal{A} \models \Phi_{\forall} \Pi t$ 
shows  $\mathcal{A} \models \text{SAT-Plan-Base.encode-initial-state } \Pi$ 
and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-operators } \Pi t$ 
and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t$ 
and  $\mathcal{A} \models \text{encode-interfering-operator-exclusion } \Pi t$ 
and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-goal-state } \Pi t$ 
using assms
unfolding encode-problem-with-operator-interference-exclusion-def encode-problem-def
SAT-Plan-Base.encode-problem-def
by simp+

```

Just as for the basic SATPlan encoding we defined local context for the SATPlan encoding with interfering operator exclusion. We omit this here

since it is basically identical to the one shown in the basic SATPlan theory replacing only the definitions of and 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  $\Pi$ 
  and  $\mathcal{A} \models \Phi_{\forall} \Pi t$ 
  and  $k < t$ 
  and  $ops \in set(subseqs((\Phi^{-1} \Pi \mathcal{A} t) ! k))$ 
  shows are-all-operators-non-interfering ops
  proof -
    let ?ops = strips-problem.operators-of  $\Pi$ 
    and ? $\pi$  =  $\Phi^{-1} \Pi \mathcal{A} t$ 
    and ? $\Phi_X$  = encode-interfering-operator-exclusion  $\Pi t$ 
    let ? $\pi_k$  =  $(\Phi^{-1} \Pi \mathcal{A} t) ! k$ 

    {
      fix C
      assume C-in:  $C \in cnf ?\Phi_X$ 
      have cnf-semantics  $\mathcal{A} (cnf ?\Phi_X)$ 
      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 nb1 = this
    {
      fix op1 op2
      assume op1 ∈ set ? $\pi_k$  and op2 ∈ set ? $\pi_k$ 
        and index-op1-is-not-index-op2: index ?ops op1 ≠ index ?ops op2
      moreover have op1-in: op1 ∈ set ?ops and  $\mathcal{A}$ -models-op1: $\mathcal{A}$  (Operator k (index ?ops op1))
    }
```

```

and  $op_2\text{-in: } op_2 \in \text{set } ?ops$  and  $\mathcal{A}\text{-models-}op_2: \mathcal{A}$  ( $Operator\ k\ (\text{index } ?ops\ op_2)$ )
  using  $decode\text{-plan}\text{-step}\text{-element}\text{-then}[OF\ assms(3)]$  calculation
  unfolding  $decode\text{-plan}\text{-def}$ 
  by  $blast+$ 
  moreover {
    let  $?C = \{ (Operator\ k\ (\text{index } ?ops\ op_1))^{-1}, (Operator\ k\ (\text{index } ?ops\ op_2))^{-1}$ 
  }
  assume  $are\text{-operators}\text{-interfering } op_1\ op_2$ 
  moreover have  $?C \in cnf\ ?\Phi_X$ 
    using  $cnf\text{-of-}encode\text{-interfering}\text{-operator}\text{-exclusion}\text{-contains}\text{-clause}\text{-if}[OF\ assms(3)\ op_1\text{-in } op_2\text{-in } index\text{-}op_1\text{-is-not-index-}op_2]$  calculation
    by  $blast$ 
  moreover have  $\neg clause\text{-semantics } \mathcal{A}\ ?C$ 
    using  $\mathcal{A}\text{-models-}op_1\ \mathcal{A}\text{-models-}op_2$ 
    unfolding  $clause\text{-semantics}\text{-def}$ 
    by  $auto$ 
  ultimately have  $False$ 
    using  $nb_1$ 
    by  $blast$ 
  }
  ultimately have  $\neg are\text{-operators}\text{-interfering } op_1\ op_2$ 
  by  $blast$ 
}
note  $nb_3 = this$ 
show  $?thesis$ 
  using  $assms$ 
  proof (induction  $ops$ )
    case ( $Cons\ op_1\ ops$ )
      have  $are\text{-all}\text{-operators}\text{-non-interfering } ops$ 
      using  $Cons.IH[OF\ Cons.\text{prems}(1, 2, 3)\ Cons\text{-in}\text{-subseqsD}[OF\ Cons.\text{prems}(4)]]$ 
      by  $blast$ 
    moreover {
      fix  $op_2$ 
      assume  $op_2\text{-in-}ops: op_2 \in \text{set } ops$ 
      moreover have  $op_1\text{-in-}\pi_k: op_1 \in \text{set } ?\pi_k$  and  $op_2\text{-in-}\pi_k: op_2 \in \text{set } ?\pi_k$ 
        using  $element\text{-of-}subseqs\text{-then-}subset[OF\ Cons.\text{prems}(4)]$  calculation(1)
        by  $auto+$ 
      moreover
      {
        have  $distinct\ (op_1 \# ops)$ 
        using  $subseqs\text{-distinctD}[OF\ Cons.\text{prems}(4)]$ 
           $decode\text{-plan}\text{-step}\text{-distinct}[OF\ Cons.\text{prems}(3)]$ 
        unfolding  $decode\text{-plan}\text{-def}$ 
        by  $blast$ 
      moreover have  $op_1 \in \text{set } ?ops$  and  $op_2 \in \text{set } ?ops$ 
        using  $decode\text{-plan}\text{-step}\text{-element}\text{-then}(1)[OF\ Cons.\text{prems}(3)]$   $op_1\text{-in-}\pi_k$ 
         $op_2\text{-in-}\pi_k$ 
        unfolding  $decode\text{-plan}\text{-def}$ 
        by  $force+$ 
    
```

```

moreover have  $op_1 \neq op_2$ 
  using  $op_2\text{-in-}ops\ calculation(1)$ 
  by fastforce
ultimately have  $\text{index } ?ops\ op_1 \neq \text{index } ?ops\ op_2$ 
  using  $\text{index}\text{-eq}\text{-index}\text{-conv}$ 
  by auto
}
ultimately have  $\neg\text{are-operators-interfering } op_1\ op_2$ 
  using  $nb_3$ 
  by blast
}
ultimately show  $?case$ 
  using list-all-iff
  by auto
qed simp
qed

theorem encode-problem-serializable-sound:
assumes is-valid-problem-strips  $\Pi$ 
  and  $\mathcal{A} \models \Phi_{\forall} \Pi t$ 
shows is-parallel-solution-for-problem  $\Pi (\Phi^{-1} \Pi \mathcal{A} t)$ 
  and  $\forall k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t). \text{are-all-operators-non-interfering } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$ 
proof -
{
  have  $\mathcal{A} \models \text{SAT-Plan-Base.encode-initial-state } \Pi$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-operators } \Pi t$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-all-frame-axioms } \Pi t$ 
    and  $\mathcal{A} \models \text{SAT-Plan-Base.encode-goal-state } \Pi t$ 
    using assms(2)
    unfolding encode-problem-with-operator-interference-exclusion-def
    by simp+
  then have  $\mathcal{A} \models \text{SAT-Plan-Base.encode-problem } \Pi t$ 
    unfolding SAT-Plan-Base.encode-problem-def
    by simp
}
thus is-parallel-solution-for-problem  $\Pi (\Phi^{-1} \Pi \mathcal{A} t)$ 
  using encode-problem-parallel-sound assms(1, 2)
  unfolding decode-plan-def
  by blast
next
let  $?{\pi} = \Phi^{-1} \Pi \mathcal{A} t$ 
{
  fix  $k$ 
  assume  $k < t$ 
  moreover have  $?{\pi} ! k \in \text{set } (\text{subseqs } (?{\pi} ! k))$ 
    using subseqs-refl
    by blast
  ultimately have are-all-operators-non-interfering  $(?{\pi} ! k)$ 
}

```

```

using encode-problem-serializable-sound-i[OF assms]
unfolding SAT-Plan-Base.decode-plan-def decode-plan-def
by blast
}
moreover have 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 ?ΦX = 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 = { (op1, op2). (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..<length π}}}
let ?A = ∪(?f ` ?Ops)
let ?B = ∪?A
have nb1: ∀ ops ∈ set π. ∀ op ∈ set ops. op ∈ set (operators-of Π)
  using is-parallel-solution-for-problem-operator-set[OF assms(2)]
  unfolding operators-of-def
  by blast
}

fix k op
assume k < length π and op ∈ set (π ! k)
hence lit-semantics ?A ((Operator k (index ?ops op))+) = (k < length ?τ - 1)
  using encode-problem-parallel-complete-vi-a[OF assms(2)]
    encode-problem-parallel-complete-vi-b[OF assms(2)] initial-of-def
  by(cases k < length ?τ - 1; simp)
} note nb2 = this
{
fix k op1 op2

```

```

assume  $k < \text{length } \pi$ 
and  $\text{op}_1 \in \text{set}(\pi ! k)$ 
and  $\text{index } ?\text{ops } \text{op}_1 \neq \text{index } ?\text{ops } \text{op}_2$ 
and  $\text{are-operators-interfering } \text{op}_1 \text{ op}_2$ 
moreover have  $\text{are-all-operators-non-interfering } (\pi ! k)$ 
using  $\text{assms}(3)$   $\text{calculation}(1)$ 
by  $\text{blast}$ 
moreover have  $\text{op}_1 \neq \text{op}_2$ 
using  $\text{calculation}(3)$ 
by  $\text{blast}$ 
ultimately have  $\text{op}_2 \notin \text{set}(\pi ! k)$ 
using  $\text{are-all-operators-non-interfering-set-contains-no-distinct-interfering-operator-pairs}$ 
assms}(3)
by  $\text{blast}$ 
} note  $\text{nb}_3 = \text{this}$ 
{
fix  $C$ 
assume  $C \in \text{cnf } ?\Phi_X$ 
then have  $C \in ?B$ 
using  $\text{cnf-of-encode-interfering-operator-exclusion-is}[\text{of } \Pi \text{ length } \pi]$ 
by  $\text{arg0}$ 
then obtain  $C'$  where  $C' \in ?A$  and  $C\text{-in: } C \in C'$ 
using  $\text{Union-iff}[\text{of } C ?A]$ 
by  $\text{meson}$ 
then obtain  $\text{op}_1 \text{ op}_2$  where  $(\text{op}_1, \text{op}_2) \in \text{set}(\text{operators-of } \Pi) \times \text{set}(\text{operators-of } \Pi)$ 
and  $\text{index-}\text{op}_1\text{-is-not-index-}\text{op}_2: \text{index } ?\text{ops } \text{op}_1 \neq \text{index } ?\text{ops } \text{op}_2$ 
and  $\text{are-operators-interfering-}\text{op}_1\text{-}\text{op}_2: \text{are-operators-interfering } \text{op}_1 \text{ op}_2$ 
and  $C'\text{-in: } C' \in \{\{(Operator } k (\text{index } ?\text{ops } \text{op}_1))^{-1}, (\text{Operator } k (\text{index } ?\text{ops } \text{op}_2))^{-1}\} \}$ 
|  $k, k \in \{0..<\text{length } \pi\}\}$ 
using  $\text{UN-iff}[\text{of } C' ?f ?\text{Ops}]$ 
by  $\text{blast}$ 
then obtain  $k$  where  $k \in \{0..<\text{length } \pi\}$ 
and  $C\text{-is: } C = \{( \text{Operator } k (\text{index } ?\text{ops } \text{op}_1))^{-1}, (\text{Operator } k (\text{index } ?\text{ops } \text{op}_2))^{-1}\}$ 
using  $C\text{-in } C'\text{-in}$ 
by  $\text{blast}$ 
then have  $k\text{-lt-length-}\pi: k < \text{length } \pi$ 
by  $\text{simp}$ 
consider (A)  $\text{op}_1 \in \text{set}(\pi ! k)$ 
| (B)  $\text{op}_2 \in \text{set}(\pi ! k)$ 
| (C)  $\neg \text{op}_1 \in \text{set}(\pi ! k) \vee \neg \text{op}_2 \in \text{set}(\pi ! k)$ 
by  $\text{linarith}$ 
hence  $\text{clause-semantics } ?\mathcal{A} C$ 
proof (cases)
case A
moreover have  $\text{op}_2 \notin \text{set}(\pi ! k)$ 
using  $\text{nb}_3 k\text{-lt-length-}\pi$   $\text{calculation }$   $\text{index-}\text{op}_1\text{-is-not-index-}\text{op}_2$   $\text{are-operators-interfering-}\text{op}_1\text{-}\text{op}_2$ 

```

```

    by blast
  moreover have  $\neg \mathcal{A} (\text{Operator } k (\text{index } ?ops op_2))$ 
    using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length- $\pi$ ]
      calculation(2)
    by blast
  ultimately show ?thesis
    using C-is
    unfolding clause-semantics-def
    by force
next
  case B
  moreover have  $op_1 \notin set (\pi ! k)$ 
  using nb3 k-lt-length- $\pi$  calculation index-op1-is-not-index-op2 are-operators-interfering-op1-op2
    by blast
  moreover have  $\neg \mathcal{A} (\text{Operator } k (\text{index } ?ops op_1))$ 
    using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length- $\pi$ ]
      calculation(2)
    by blast
  ultimately show ?thesis
    using C-is
    unfolding clause-semantics-def
    by force
next
  case C
  then show ?thesis
    proof (rule disjE)
      assume  $op_1 \notin set (\pi ! k)$ 
      then have  $\neg \mathcal{A} (\text{Operator } k (\text{index } ?ops op_1))$ 
        using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length- $\pi$ ]
          by blast
      thus clause-semantics (valuation-for-plan  $\Pi \pi$ ) C
        using C-is
        unfolding clause-semantics-def
        by force
    next
      assume  $op_2 \notin set (\pi ! k)$ 
      then have  $\neg \mathcal{A} (\text{Operator } k (\text{index } ?ops op_2))$ 
        using encode-problem-parallel-complete-vi-d[OF assms(2) k-lt-length- $\pi$ ]
          by blast
      thus clause-semantics (valuation-for-plan  $\Pi \pi$ ) C
        using C-is
        unfolding clause-semantics-def
        by force
    qed
  qed
}
then have cnf-semantics ?A (cnf ? $\Phi_X$ )
  unfolding cnf-semantics-def..
thus ?thesis

```

```

using cnf-semantics[OF is-nnf-cnf[OF is-cnf-encode-interfering-operator-exclusion]]
  by fast
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:

$$\{ (Operator\ k\ (index\ ops\ op_1))^{-1}, (Operator\ k\ (index\ ops\ op_2))^{-1} \}$$

where $ops \equiv \Pi_{\mathcal{O}}$. 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 $\mathcal{A} \equiv valuation-for-plan \Pi \pi$ it holds that

$$\mathcal{A} \models \neg Atom\ (Operator\ k\ (index\ ops\ op_2))$$

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  $\Pi$ 
    and is-parallel-solution-for-problem  $\Pi \pi$ 
    and  $\forall k < length \pi. are-all-operators-non-interfering (\pi ! k)$ 
  shows valuation-for-plan  $\Pi \pi \models \Phi_{\forall} \Pi (length \pi)$ 
proof -
  let ? $\mathcal{A}$  = valuation-for-plan  $\Pi \pi$ 
    and ? $\Phi_X$  = encode-interfering-operator-exclusion  $\Pi (length \pi)$ 
  have ? $\mathcal{A}$   $\models SAT\text{-}Plan\text{-}Base.encode\text{-}problem \Pi (length \pi)$ 
    using assms(1, 2) encode-problem-parallel-complete
    by auto
  moreover have ? $\mathcal{A}$   $\models \Phi_X$ 
  using encode-problem-with-operator-interference-exclusion-complete-i[OF assms].
  ultimately show ?thesis
  unfolding encode-problem-with-operator-interference-exclusion-def encode-problem-def
    SAT-Plan-Base.encode-problem-def
    by force
qed
value stop

```

```

lemma encode-problem-forall-step-decoded-plan-is-serializable-i:
  assumes is-valid-problem-strips  $\Pi$ 
    and  $\mathcal{A} \models \Phi_{\vee} \Pi t$ 
  shows  $(\Pi)_G \subseteq_m \text{execute-serial-plan } ((\Pi)_I) (\text{concat } (\Phi^{-1} \Pi \mathcal{A} t))$ 
proof -
  let  $?G = (\Pi)_G$ 
  and  $?I = (\Pi)_I$ 
  and  $?{\pi} = \Phi^{-1} \Pi \mathcal{A} t$ 
  let  $?{\pi}' = \text{concat } (\Phi^{-1} \Pi \mathcal{A} t)$ 
  and  $?{\tau} = \text{trace-parallel-plan-strips } ?I ?{\pi}$ 
  and  $?{\sigma} = \text{map } (\text{decode-state-at } \Pi \mathcal{A}) [0..<\text{Suc } (\text{length } ?{\pi})]$ 
  {
    fix  $k$ 
    assume  $k < \text{length } ?{\pi}$ 
    moreover have  $\mathcal{A} \models \text{SAT-Plan-Base.encode-problem } \Pi t$ 
      using assms(2)
      unfolding encode-problem-with-operator-interference-exclusion-def
        encode-problem-def SAT-Plan-Base.encode-problem-def
      by simp
    moreover have  $\text{length } ?{\sigma} = \text{length } ?{\tau}$ 
      using encode-problem-parallel-correct-vii assms(1) calculation
      unfolding decode-state-at-def decode-plan-def initial-of-def
      by fast
    ultimately have  $k < \text{length } ?{\tau} - 1$  and  $k < t$ 
      unfolding decode-plan-def SAT-Plan-Base.decode-plan-def
      by force+
  } note nb = this
  {
    have  $?G \subseteq_m \text{execute-parallel-plan } ?I ?{\pi}$ 
      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  $?G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } ?I ?{\pi})$ 
      using execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace
      by fast
  }
  moreover {
    fix  $k$ 
    assume  $k < \text{length } ?{\pi}$ 
    moreover have  $k < \text{length } ?{\tau} - 1$  and  $k < t$ 
      using nb calculation
      by blast+
    moreover have are-all-operators-applicable  $(?{\tau} ! k) (?{\pi} ! k)$ 
      and are-all-operator-effects-consistent  $(?{\pi} ! k)$ 
      using trace-parallel-plan-strips-operator-preconditions calculation(2)
      by blast+
    moreover have are-all-operators-non-interfering  $(?{\pi} ! k)$ 
  }

```

```

using encode-problem-serializable-sound(2)[OF assms(1, 2)] k-lt-length-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'

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

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

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

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

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

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

theorem *serializable-encoding-decoded-plan-is-serializable*:

assumes *is-valid-problem-strips* Π

and $\mathcal{A} \models \Phi_{\forall} \Pi t$

shows *is-serial-solution-for-problem* $\Pi (concat (\Phi^{-1} \Pi \mathcal{A} t))$

using *encode-problem-forall-step-decoded-plan-is-serializable-i* [OF assms]

encode-problem-forall-step-decoded-plan-is-serializable-ii

unfolding *is-serial-solution-for-problem-def* *goal-of-def*

initial-of-def *decode-plan-def*

by *blast*

end

theory *SAT-Solve-SAS-Plus*

imports *SAS-Plus-STRIPS*

SAT-Plan-Extensions

begin

¹²These 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.

9 SAT-Solving of SAS+ Problems

```

lemma sas-plus-problem-has-serial-solution-iff-i:
  assumes is-valid-problem-sas-plus  $\Psi$ 
    and  $\mathcal{A} \models \Phi_{\forall} (\varphi \Psi) t$ 
  shows is-serial-solution-for-problem  $\Psi [\varphi_O^{-1} \Psi op. op \leftarrow concat (\Phi^{-1} (\varphi \Psi) \mathcal{A} t)]$ 
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  $\Psi$ 
    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  $\mathcal{A} = valuation-for-plan \Pi \pi$ 
  let  $t = length \psi$ 

  have nb:  $length \psi = length \pi$ 
  unfolding SAS-Plus-STRIPS.sas-plus-parallel-plan-to-strips-parallel-plan-def
    sasp-op-to-strips-def
    sas-plus-parallel-plan-to-strips-parallel-plan-def
    by (induction  $\psi$ ; auto)
  have is-valid-problem-strips  $\Pi$ 
    using assms(1) is-valid-problem-sas-plus-then-strips-transformation-too
    by blast
  moreover have STRIPS-Semantics.is-parallel-solution-for-problem  $\Pi \pi$ 
    using execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus[OF assms(1,2)]

```

```

strips-equivalent-to-sas-plus[OF assms(1)]
by blast
moreover {
fix k
assume k < 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 = [[φO 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' = [φO 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 $\Phi_{\forall} \varphi \Psi t$ exist.

theorem sas-plus-problem-has-serial-solution-iff:

```

assumes is-valid-problem-sas-plus  $\Psi$ 
shows  $(\exists \psi. \text{is-serial-solution-for-problem } \Psi \psi) \longleftrightarrow (\exists \mathcal{A} t. \mathcal{A} \models \Phi_{\forall} (\varphi \Psi) t)$ 
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  $\equiv (\text{SAS-Plus-Representation.sas-plus-operator.precondition-of} = []$ ,
 $\text{SAS-Plus-Representation.sas-plus-operator.effect-of} = [])$ 

```

```

lemma sasp-exec-noops: execute-serial-plan-sas-plus  $s$  (replicate n empty-sasp-action)
 $= s$ 
by (induction n arbitrary:
  (auto simp: empty-sasp-action-def STRIPS-Representation.is-operator-applicable-in-def
   execute-operator-def))

```

definition

```

prob-with-noop  $\Pi \equiv$ 
  (SAS-Plus-Representation.sas-plus-problem.variables-of = SAS-Plus-Representation.sas-plus-problem.variables-of  $\Pi$ ,
   SAS-Plus-Representation.sas-plus-problemoperators-of = empty-sasp-action
  # SAS-Plus-Representation.sas-plus-problemoperators-of  $\Pi$ ,
   SAS-Plus-Representation.sas-plus-problem.initial-of = SAS-Plus-Representation.sas-plus-problem.initial-of  $\Pi$ ,
   SAS-Plus-Representation.sas-plus-problem.goal-of = SAS-Plus-Representation.sas-plus-problem.goal-of  $\Pi$ ,
   SAS-Plus-Representation.sas-plus-problem.range-of = SAS-Plus-Representation.sas-plus-problem.range-of  $\Pi$ )

```

```

lemma sasp-noops-in-noop-problem: set (replicate n empty-sasp-action)  $\subseteq$  set (SAS-Plus-Representation.sas-plus-problem (prob-with-noop  $\Pi$ ))
by (induction n) (auto simp: prob-with-noop-def)

```

lemma *noops-complete*:

```

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

```

```

definition rem-noops  $\equiv \text{filter } (\lambda op. op \neq \text{empty-sasp-action})$ 

```

```

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

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

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

lemma sas-plus-problem-has-serial-solution-iff-i':
  assumes is-valid-problem-sas-plus  $\Psi$ 
  and  $\mathcal{A} \models \Phi_{\forall} (\varphi (\text{prob-with-noop } \Psi)) t$ 
  shows SAS-Plus-Semantics.is-serial-solution-for-problem  $\Psi$ 
    (rem-noops
      (map ( $\lambda op. \varphi_O^{-1} (\text{prob-with-noop } \Psi) op$ )
        (concat ( $\Phi^{-1} (\varphi (\text{prob-with-noop } \Psi)) \mathcal{A} t$ ))))
  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  $\Psi$ 
  and SAS-Plus-Semantics.is-serial-solution-for-problem  $\Psi \psi$ 
  and length  $\psi \leq h$ 
  shows  $\exists \mathcal{A}. (\mathcal{A} \models \Phi_{\forall} (\varphi (\text{prob-with-noop } \Psi)) h)$ 
  using assms
  by(fastforce
    intro!: assms noops-valid noops-complete
    sas-plus-problem-has-serial-solution-iff-ii
    [where  $\psi = (\text{replicate } (h - \text{length } \psi) \text{ empty-sasp-action}) @ \psi$ ])
  end

theory AST-SAS-Plus-Equivalence
  imports AI-Planning-Languages-Semantics.SASP-Semantics SAS-Plus-Semantics
  List-Index.List-Index
  begin

```

11 Proving Equivalence of SAS+ representation and Fast-Downward's Multi-Valued Problem Representation

11.1 Translating Fast-Downward's representation to SAS+

```

type-synonym nat-sas-plus-problem = (nat, nat) sas-plus-problem
type-synonym nat-sas-plus-operator = (nat, nat) sas-plus-operator
type-synonym nat-sas-plus-plan = (nat, nat) sas-plus-plan
type-synonym nat-sas-plus-state = (nat, nat) state

definition is-standard-effect :: ast-effect  $\Rightarrow$  bool
where is-standard-effect  $\equiv \lambda(\text{pre}, \text{-}, \text{-}, \text{-}). \text{pre} = []$ 

definition is-standard-operator :: ast-operator  $\Rightarrow$  bool
where is-standard-operator  $\equiv \lambda(\text{-}, \text{-}, \text{effects}, \text{-}). \text{list-all is-standard-effect effects}$ 

fun rem-effect-implicit-pres:: ast-effect  $\Rightarrow$  ast-effect where
  rem-effect-implicit-pres (preconds, v, implicit-pre, eff) = (preconds, v, None, eff)

fun rem-implicit-pres :: ast-operator  $\Rightarrow$  ast-operator where
  rem-implicit-pres (name, preconds, effects, cost) =
    (name, (implicit-pres effects) @ preconds, map rem-effect-implicit-pres effects, cost)

fun rem-implicit-pres-ops :: ast-problem  $\Rightarrow$  ast-problem where
  rem-implicit-pres-ops (vars, init, goal, ops) = (vars, init, goal, map rem-implicit-pres ops)

definition consistent-map-lists xs1 xs2  $\equiv (\forall (x1, x2) \in \text{set xs1}. \forall (y1, y2) \in \text{set xs2}. x1 = y1 \longrightarrow x1 = y2)$ 

lemma map-add-comm:  $(\bigwedge x. x \in \text{dom m1} \wedge x \in \text{dom m2} \implies m1 x = m2 x) \implies$ 
   $m1 ++ m2 = m2 ++ m1$ 
by (fastforce simp add: map-add-def split: option.splits)

lemma first-map-add-submap:  $(\bigwedge x. x \in \text{dom m1} \wedge x \in \text{dom m2} \implies m1 x = m2 x) \implies$ 
   $m1 ++ m2 \subseteq_m x \implies m1 \subseteq_m x$ 
using map-add-le-mapE map-add-comm
by force

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

```

```

lemma consistent-map-lists:
   $\llbracket \text{distinct } (\text{map} \text{ } \text{fst} \text{ } (xs1 @ xs2)); x \in \text{dom } (\text{map-of } xs1) \cap \text{dom } (\text{map-of } xs2) \rrbracket \implies$ 
     $(\text{map-of } xs1) \ x = (\text{map-of } xs2) \ x$ 
apply(induction xs1)
  apply (simp-all add: consistent-map-lists-def image-def)
  using map-of-SomeD
  by fastforce

lemma subsuming-states-append:
   $\text{distinct } (\text{map} \text{ } \text{fst} \text{ } (xs @ ys)) \implies$ 
     $(\text{map-of } (xs @ ys)) \subseteq_m s \longleftrightarrow ((\text{map-of } ys) \subseteq_m s \wedge (\text{map-of } xs) \subseteq_m s)$ 
  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-pres-op op  $\equiv$  (case op of (name, pres, effs, cost)  $\Rightarrow$ 
     $\text{distinct } (\text{map} \text{ } \text{fst} \text{ } (\text{pres} @ (\text{implicit-pres } effs)))$ 
     $\wedge \text{consistent-map-lists } \text{pres } (\text{implicit-pres } effs))$ 

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

lemma consistent-pres-op-then': consistent-pres-op op  $\implies$  consistent-pres-op' op
  by(auto simp add: consistent-pres-op'-def consistent-pres-op-def)

lemma rem-implicit-pres-ops-valid-states:
  ast-problem.valid-states (rem-implicit-pres-ops prob) = ast-problem.valid-states
  prob
  apply(cases prob)
  by(auto simp add: ast-problem.valid-states-def ast-problem.Dom-def
    ast-problem.numVars-def ast-problem.astDom-def
    ast-problem.range-of-var-def ast-problem.numVals-def)

lemma rem-implicit-pres-ops-lookup-op-None:
  ast-problem.lookup-operator (vars, init, goal, ops) name = None  $\longleftrightarrow$ 
  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)  $\implies$ 
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name
  =
  Some (rem-implicit-pres (n,p,vp,e))
  by (induction ops) (fastforce simp: ast-problem.lookup-operator-def ast-problem.astδ-def) +

```

```

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

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

lemma rem-implicit-pres-ops-lookup-op-Some-2':
  ast-problem.lookup-operator (rem-implicit-pres-ops (vars, init, goal, ops)) name
= Some op
 $\implies \exists op'. \text{ast-problem.lookup-operator}(\text{vars}, \text{init}, \text{goal}, \text{ops}) \text{ name} = \text{Some } op'$ 
 $\wedge$ 
  (op = rem-implicit-pres op')
by (induction ops) (auto simp: ast-problem.lookup-operator-def 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)
 $\implies \exists op'. \text{ast-problem.lookup-operator prob name} = \text{Some } op' \wedge$ 
  ((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)
 $\wedge$  ps ⊆m s)
by (auto simp add: ast-problem.subsuming-states-def)

lemma rem-implicit-pres-ops-enabled-1:
   $\llbracket (\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}) \implies \text{consistent-pres-op } op);$ 
   $\text{ast-problem.enabled prob name } s \rrbracket \implies$ 
   $\text{ast-problem.enabled (rem-implicit-pres-ops prob) name } s$ 
by (fastforce simp: ast-problem.enabled-def rem-implicit-pres-ops-valid-states 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-ind: lookup-operator π = Some op  $\implies$  op ∈ set astδ

```

```

by(auto simp: find-Some-if in-set-conv-nth lookup-operator-def)

end

lemma rem-implicit-pres-ops-enabled-2:
assumes "(\ $\bigwedge op. op \in set (ast\text{-}problem.ast\delta prob) \Rightarrow$  consistent-pres-op op)
shows ast\text{-}problem.enabled (rem-implicit-pres-ops prob) name s  $\Rightarrow$ 
ast\text{-}problem.enabled prob name s
using assms[OF ast\text{-}problem.lookup-Some-in\delta, unfolded consistent-pres-op-def]
apply(auto simp: subsuming-states-append rem-implicit-pres-ops-valid-states sub-
suming-states-def'
ast\text{-}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:
( $\bigwedge op. op \in set (ast\text{-}problem.ast\delta prob) \Rightarrow$  consistent-pres-op op)  $\Rightarrow$ 
ast\text{-}problem.enabled (rem-implicit-pres-ops prob) name s = ast\text{-}problem.enabled
prob name s
using rem-implicit-pres-ops-enabled-1 rem-implicit-pres-ops-enabled-2
by blast

context ast\text{-}problem
begin

lemma std-eff-enabled[simp]:
is-standard-operator (name, pres, effs, layer)  $\Rightarrow$  s \in valid-states  $\Rightarrow$  (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)  $\Rightarrow$ 
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:
 $\llbracket (\bigwedge op. op \in set (ast\text{-}problem.ast\delta (a,b,c,d)) \Rightarrow$  is-standard-operator op);
op \in set (ast\text{-}problem.ast\delta (rem-implicit-pres-ops (a,b,c,d)))  $\rrbracket$ 
 $\Rightarrow$  is-standard-operator op
by (induction d) (fastforce simp add: ast\text{-}problem.ast\delta-def image-def dest!: is-standard-operator-rem-implicit)+

lemma is-standard-operator-rem-implicit-pres-ops':
 $\llbracket op \in set (ast\text{-}problem.ast\delta (rem-implicit-pres-ops prob));$ 
 $(\bigwedge op. op \in set (ast\text{-}problem.ast\delta prob) \Rightarrow$  is-standard-operator op)  $\rrbracket$ 

```

```

 $\implies \text{is-standard-operator } op$ 
apply(cases prob)
using is-standard-operator-rem-implicit-pres-ops
by blast

lemma in-rem-implicit-pres- $\delta$ :
 $op \in \text{set } (\text{ast-problem}.ast\delta \text{ prob}) \implies$ 
 $\text{rem-implicit-pres } op \in \text{set } (\text{ast-problem}.ast\delta (\text{rem-implicit-pres-ops prob}))$ 
by(auto simp add: ast-problem.ast $\delta$ -def)

lemma rem-implicit-pres-ops-execute:
assumes
 $(\bigwedge op. op \in \text{set } (\text{ast-problem}.ast\delta \text{ prob}) \implies \text{is-standard-operator } op) \text{ and}$ 
 $s \in \text{ast-problem.valid-states prob}$ 
shows ast-problem.execute (rem-implicit-pres-ops prob) name s = ast-problem.execute
prob name s
proof-
have (n,ps,es,c)  $\in \text{set } (\text{ast-problem}.ast\delta \text{ prob}) \implies$ 
 $(\text{filter } (\text{ast-problem.eff-enabled prob } s) \text{ es}) = es \text{ for } n \text{ ps } es \text{ c}$ 
using assms(2)
by (auto simp add: ast-problem.std-eff-enabled dest!: assms(1))
moreover have (n,ps,es,c)  $\in \text{set } (\text{ast-problem}.ast\delta \text{ prob}) \implies$ 
 $(\text{filter } (\text{ast-problem.eff-enabled } (\text{rem-implicit-pres-ops prob}) \text{ s}) \text{ (map rem-effect-implicit-pres}$ 
 $es)) = \text{map rem-effect-implicit-pres es for } n \text{ ps } es \text{ c}$ 
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- $\delta$ )
moreover have map-of (map (( $\lambda(-,x,-,v). (x,v)$ ) o rem-effect-implicit-pres) effs)
=
map-of (map ( $\lambda(-,x,-,v). (x,v)$ ) 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-in $\delta$ )
qed

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

```

```

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

lemma rem-implicit-pres-ops-goal[simp]: ast-problem.G (rem-implicit-pres-ops prob)
= ast-problem.G prob
apply(cases prob)
using rem-implicit-pres-ops-valid-states
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
    ( $\bigwedge op. op \in set (ast\text{-problem}.ast\delta prob) \implies$  consistent-pres-op op)
    ( $\bigwedge op. op \in set (ast\text{-problem}.ast\delta prob) \implies$  is-standard-operator op)
shows ast-problem.valid-plan (rem-implicit-pres-ops prob)  $\pi s =$  ast-problem.valid-plan
prob  $\pi s$ 
using wf-ast-problem.I-valid[OF assms(1)] rem-implicit-pres-ops-path-to[OF assms]
by (simp add: ast-problem.valid-plan-def rem-implicit-pres-ops-goal rem-implicit-pres-ops-init)

lemma rem-implicit-pres-ops-numVars[simp]:
    ast-problem.numVars (rem-implicit-pres-ops prob) = ast-problem.numVars prob
by (cases prob) (simp add: ast-problem.numVars-def ast-problem.astDom-def)

lemma rem-implicit-pres-ops-numVals[simp]:
    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:
     $(x, a) \in set (implicit\text{-pres} effs) \implies (\exists epres v vp. (epres, x, vp, v) \in set effs \wedge vp =$ 
Some a)
by (induction effs) (fastforce simp: implicit-pres-def image-def split: if-splits) +

lemma pair4-eqD:  $(a1, a2, a3, a4) = (b1, b2, b3, b4) \implies a3 = b3$ 
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 (exists vp. (epres,x,vp,v) ∈ set effs)  $\longleftrightarrow$  (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))
   $\implies$  (exists 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 ( $\bigwedge op. op \in set (ast\text{-}problem.ast\delta prob) \implies consistent\text{-}pres\text{-}op op$ )
    ast\text{-}problem.well-formed prob
  shows ast\text{-}problem.well-formed (rem-implicit-pres-ops prob)
proof-
  have map fst (ast\text{-}problem.ast\delta (rem-implicit-pres-ops prob)) = map fst (ast\text{-}problem.ast\delta prob)
    by (cases prob) (auto simp: ast\text{-}problem.ast\delta-def)
    thus ?thesis
      using assms
    by(auto simp add: ast\text{-}problem.well-formed-def rem-implicit-pres-ops-wf-partial-state
      simp del: rem-implicit-pres.simps
      dest!: rem-implicit-pres-ops-in\delta D
      intro!: rem-implicit-pres-wf-operator)
  qed

definition is-standard-effect'
  :: ast\text{-}effect  $\Rightarrow$  bool
  where is-standard-effect'  $\equiv$   $\lambda(pre, -, vpre, -). pre = [] \wedge vpre = None$ 

definition is-standard-operator'
  :: ast\text{-}operator  $\Rightarrow$  bool
  where is-standard-operator'  $\equiv$   $\lambda(-, -, effects, -). list\text{-}all is\text{-}standard\text{-}effect' effects$ 

lemma rem-implicit-pres-is-standard-operator':
  is-standard-operator (n,p,es,c)  $\implies$  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':
  ( $\bigwedge op. op \in set (ast\text{-}problem.ast\delta (vs, I, G, ops)) \implies is\text{-}standard\text{-}operator op$ )
   $\implies$ 
   $\pi \in set (ast\text{-}problem.ast\delta (rem-implicit-pres-ops (vs, I, G, ops))) \implies is\text{-}standard\text{-}operator'$ 
   $\pi$ 
  by (cases ops) (auto simp: ast\text{-}problem.ast\delta-def dest!: rem-implicit-pres-is-standard-operator')

locale abs-ast-prob = wf-ast\text{-}problem +
  assumes no-cond-efs:  $\forall \pi \in set ast\delta. is\text{-}standard\text{-}operator' \pi$ 

context ast\text{-}problem
begin

definition abs-ast-variable-section = [0..<(length astDom)]
definition abs-range-map
  :: (nat  $\rightarrow$  nat list)
  where abs-range-map  $\equiv$ 

```

```

map-of (zip abs-ast-variable-section
        (map ((λvals. [0..<length vals]) o snd o snd)
              astDom))

end

context abs-ast-prob
begin

lemma is-valid-vars-1: astDom ≠ []  $\implies$  abs-ast-variable-section ≠ []
  by (simp add: abs-ast-variable-section-def)

end

lemma upt-eq-Nil-conv'[simp]: ([] = [i..<j]) = (j = 0 ∨ j ≤ i)
  by (induct j) simp-all

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

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

lemma in-set-zip-lengthE:
  (x,y) ∈ set(zip [0..<length xs] xs)  $\implies$  ([ x < length xs; xs ! x = y ]  $\implies$  R)  $\implies$ 
  R
  by (induction xs rule: rev-induct) (auto simp add: nth-append map-add-Some-iff)

context abs-ast-prob
begin

lemma is-valid-vars-2:
  shows list-all (λv. abs-range-map v ≠ 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  $\longleftrightarrow$  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  $\implies$ 
    (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  $\longleftrightarrow$  v < length astDom
  by (simp add: abs-ast-variable-section-def)

lemma [simp]: v < length astDom  $\implies$  (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  $\implies$  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  $\implies$  x < length (snd (snd (astDom
! v)))  $\implies$ 
  x ∈ set (the (abs-range-map v))
  using abs-range-map-Some
  by (auto simp add: )

lemma [intro!]: x < length astDom  $\implies$  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  $\implies$  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  $\implies$  (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
    numVals-def)

```

```

list.pred-set
  numVars-def
  split: prod.splits)

lemma abs-valid-operator-1[intro!]:
  wf-operator op  $\implies$  list-all ( $\lambda(v, a)$ . 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)  $\implies$  wf-partial-state
pres
  by (simp add: wf-operator-def)

lemma abs-valid-operator-2[intro!]:
  wf-operator op  $\implies$ 
  list-all ( $\lambda(v, a)$ . ( $\exists y$ . abs-range-map v = Some y)  $\wedge$  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)  $\implies$ 
  ( $\llbracket$ distinct (map ( $\lambda(-, v, -, -)$ . v) effs);
 $\wedge$ epres x vp v. (epres,x,vp,v) $\in$ set effs  $\implies$  wf-partial-state epres;
 $\wedge$ epres x vp v.(epres,x,vp,v) $\in$ set effs  $\implies$  x < numVars;
 $\wedge$ epres x vp v. (epres,x,vp,v) $\in$ set effs  $\implies$  v < numVals x;
 $\wedge$ epres x vp v. (epres,x,vp,v) $\in$ set effs  $\implies$ 
  case vp of None  $\Rightarrow$  True | Some v  $\Rightarrow$  v<numVals x $\rrbracket$ 
 $\implies$  P)
 $\implies$  P
unfolding wf-operator-def
by (auto split: prod.splits)

lemma abs-valid-operator-3':
  wf-operator (name, pre, eff, cost)  $\implies$ 
  list-all ( $\lambda(v, a)$ . ListMem v abs-ast-variable-section) (map ( $\lambda(-, v, -, a)$ . (v, a))
eff)
  by (fastforce simp add: list.pred-set ListMem-iff abs-ast-variable-section-def im-
age-def numVars-def
elim!: wf-operator-effE split: prod.splits)

lemma abs-valid-operator-3[intro!]:
  wf-operator op  $\implies$ 
  list-all ( $\lambda(v, a)$ . 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)  $\implies$  wf-partial-state (map
 $(\lambda(-, v, -, a). (v, a)) \text{ eff}$ )
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)  $\implies$ 
list-all ( $\lambda(v, a).$  (abs-range-map  $v \neq \text{None}$ )  $\wedge$  ListMem a (the (abs-range-map
 $v))) (map ( $\lambda(-, v, -, a).$  (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  $\implies$ 
list-all ( $\lambda(v, a).$  ( $\exists y.$  abs-range-map  $v = \text{Some } y$ )  $\wedge$  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  $\implies$ 
list-all ( $\lambda(v, a).$  list-all ( $\lambda(v', a').$   $v \neq v' \vee a = a'$ ) 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-5':
wf-operator (name, pre, eff, cost)  $\implies$ 
list-all ( $\lambda(v, a).$  list-all ( $\lambda(v', a').$   $v \neq v' \vee a = a'$ ) pre) pre
apply(drule wf-operator-preD)
by (intro consistent-list-set)

lemma abs-valid-operator-5[intro!]:
wf-operator op  $\implies$ 
list-all ( $\lambda(v, a).$  list-all ( $\lambda(v', a').$   $v \neq v' \vee a = a'$ ) (precondition-of (abs-ast-operator
op)))
(b 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)  $\implies$ 
list-all ( $\lambda(v, a).$  list-all ( $\lambda(v', a').$   $v \neq v' \vee a = a'$ ) 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 ( $\lambda(v, a).$  list-all ( $\lambda(v', a').$   $v \neq v' \vee a = a'$ ) (map ( $\lambda(-, v, -, a).$ 
(v, a)) eff))$$ 
```

```

 $(map (\lambda(-, v, -, a). (v, a)) \text{eff})$ 
proof-
have *:  $map \text{fst} (map (\lambda(-, v, -, a). (v, a)) \text{eff}) = (map (\lambda(-, v, -, -). v) \text{eff})$ 
  by (induction eff) auto
show ?thesis
  using assms
  apply(elim wf-operator-effE)
  apply(intro consistent-list-set-2)
  by (subst *)
qed

lemma abs-valid-operator-6[intro!]:
  wf-operator op  $\implies$ 
    list-all  $(\lambda(v, a). list-all (\lambda(v', a'). v \neq v' \vee a = a') (effect-of (abs-ast-operator op)))$ 
    (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  $\leftarrow$  ast $\delta$ ]

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

end

context abs-ast-prob
begin

lemma [simp]: op  $\in$  set ast $\delta$   $\implies$  (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  $\Rightarrow$  nat-sas-plus-plan
  where abs-ast-plan  $\pi s$ 
     $\equiv$  map (abs-ast-operator o the o lookup-operator)  $\pi s$ 

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

lemma [simp]: enabled  $\pi s \Rightarrow$  lookup-operator  $\pi = \text{Some} (\text{name}, \text{pres}, \text{effs}, \text{layer})$ 
 $\Rightarrow$ 
  is-standard-operator' (name, pres, effs, layer)  $\Rightarrow$ 
    (filter (eff-enabled s) effs) = effs
  by (auto simp add: enabled-def is-standard-operator'-def eff-enabled-def is-standard-effect'-def
    filter-id-conv list.pred-set)

lemma effs-eq-abs-effs: (effect-of (abs-ast-operator (name, pres, effs, layer))) =
  (map ( $\lambda(-, x, -v). (x, v)$ ) effs)
  by (auto simp add: abs-ast-operator-def
    split: option.splits prod.splits)

lemma exect-eq-abs-execute:
   $\llbracket$ enabled  $\pi s; \text{lookup-operator } \pi = \text{Some} (\text{name}, \text{preconds}, \text{effs}, \text{layer});$ 
    is-standard-operator'(name, preconds, effs, layer) $\rrbracket \Rightarrow$ 
  execute  $\pi s = (\text{execute-operator-sas-plus } s ((\text{abs-ast-operator o the o lookup-operator})$ 
 $\pi))$ 
  using effs-eq-abs-effs
  by (auto simp add: execute-def execute-operator-sas-plus-def)

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

lemma path-to-then-exec-serial:  $\forall \pi \in \text{set } \pi s. \text{lookup-operator } \pi \neq \text{None} \Rightarrow$ 
  path-to  $s \pi s s' \Rightarrow$ 

```

```

 $s' \subseteq_m \text{execute-serial-plan-sas-plus } s (\text{abs-ast-plan } \pi s)$ 
proof(induction  $\pi s$  arbitrary:  $s s'$ )
  case ( $\text{Cons } a \pi s$ )
  then show ?case
    by (force simp:  $\text{exec-eq-abs-execute abs-ast-plan-def lookup-}Some\text{-in}\delta$  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:
   $(None = \text{map-of } xys x) = (x \notin \text{fst } (\text{set } xys))$ 
  by (induct xys) simp-all

lemma [simp]:  $I = \text{abs-ast-initial-state}$ 
  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]:  $\forall \pi \in \text{set } \pi s.$   $\text{lookup-operator } \pi \neq None \implies$ 
   $op \in \text{set } (\text{abs-ast-plan } \pi s) \implies op \in \text{set } \text{abs-ast-operator-section}$ 
  by (induction  $\pi s$ ) (auto simp: abs-ast-plan-def abs-ast-operator-section-def lookup- $Some\text{-in}\delta$ )
end

context ast-problem
begin

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

lemma valid-plan-then-lookup- $Some$ : valid-plan  $\pi s \implies (\forall \pi \in \text{set } \pi s.$ 
   $\text{lookup-operator } \pi \neq None)$ 
  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  $\pi s$ 
  shows is-serial-solution-for-problem abs-prob (abs-ast-plan  $\pi s$ )
  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+ representation to Fast-Downward's

```
context ast-problem
begin
```

```
definition lookup-action:: nat-sas-plus-operator  $\Rightarrow$  ast-operator option where
  lookup-action op  $\equiv$ 
    find ( $\lambda(-, pres, effs, -)$ . precondition-of op = pres  $\wedge$ 
          map ( $\lambda(v,a)$ . ( $\emptyset, v, \text{None}, a$ ) (effect-of op) = effs))
    ast $\delta$ 
```

```
end
```

```
context abs-ast-prob
begin
```

```
lemma find-Some: find P xs = Some x  $\implies$  x  $\in$  set xs  $\wedge$  P x
  by (auto simp add: find-Some-iff)
```

```
lemma distinct-find: distinct (map f xs)  $\implies$  x  $\in$  set xs  $\implies$  find ( $\lambda x'. f x' = f x$ )
  xs = Some x
  by (induction xs) (auto simp: image-def)
```

```
lemma lookup-operator-find: lookup-operator nme = find ( $\lambda op. fst op = nme$ ) ast $\delta$ 
  by (auto simp: lookup-operator-def intro!: arg-cong[where f = ( $\lambda x. find x$ ) ast $\delta$ ])
```

```
lemma lookup-operator-works-1: lookup-action op = Some  $\pi'$   $\implies$  lookup-operator
  (fst  $\pi')$  = Some  $\pi'$ 
  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')
   $\implies$  pres = pres'
  by (auto simp: lookup-action-def abs-ast-operator-def dest!: find-Some)
```

```
lemma [simp]: is-standard-operator' (name, pres, effs, layer)  $\implies$ 
  map ( $\lambda(v,a)$ . ( $\emptyset, v, \text{None}, a$ ) (effect-of (abs-ast-operator (name, pres, effs,
  layer)))) = effs
  by (induction effs) (auto simp: is-standard-operator'-def abs-ast-operator-def
  is-standard-effect'-def)
```

```
lemma lookup-operator-works-3:
  is-standard-operator' (name, pres, effs, layer)  $\implies$  (name, pres, effs, layer)  $\in$  set
  ast $\delta$   $\implies$ 
```

```

lookup-action (abs-ast-operator (name, pres, effs, layer)) = Some (name', pres',  

effs', layer')  

   $\implies \text{effs} = \text{effs}'$   

by(auto simp: is-standard-operator'-def lookup-action-def dest!: find-Some)

```

lemma *mem-find-Some*: $x \in \text{set } xs \implies P x \implies \exists x'. \text{find } P xs = \text{Some } x'$
by (*induction xs*) *auto*

lemma [*simp*]: *precondition-of* (*abs-ast-operator* (*x1*, *a*, *aa*, *b*)) = *a*
by(*simp add: abs-ast-operator-def*)

lemma *std-lookup-action*: *is-standard-operator'* *ast-op* $\implies \text{ast-op} \in \text{set } \text{ast}\delta \implies$
 $\exists \text{ast-op}'. \text{lookup-action} (\text{abs-ast-operator } \text{ast-op}) = \text{Some } \text{ast-op}'$
unfolding *lookup-action-def*
apply(*intro mem-find-Some*)
by (*auto split: prod.splits simp: o-def*)

lemma *is-applicable-then-enabled-1*:
 $\text{ast-op} \in \text{set } \text{ast}\delta \implies$
 $\exists \text{ast-op}'. \text{lookup-operator} ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) \text{ ast-op})$
= *Some ast-op'*
using *lookup-operator-works-1 std-lookup-action no-cond-effs*
by *auto*

lemma *lookup-action-Some-in-δ*: *lookup-action op = Some ast-op* $\implies \text{ast-op} \in \text{set } \text{ast}\delta$
using *lookup-operator-works-1 lookup-Some-inδ by fastforce*

lemma *lookup-operator-eq-name*: *lookup-operator name = Some (name', pres, effs, layer)* $\implies \text{name} = \text{name}'$
using *lookup-operator-wf(2)*
by *fastforce*

lemma *eq-name-eq-pres*: $(\text{name}, \text{pres}, \text{effs}, \text{layer}) \in \text{set } \text{ast}\delta \implies (\text{name}, \text{pres}', \text{effs}', \text{layer}') \in \text{set } \text{ast}\delta$
 $\implies \text{pres} = \text{pres}'$
using *eq-key-imp-eq-value[OF wf-operators(1)]*
by *auto*

lemma *eq-name-eq-effs*:
 $\text{name} = \text{name}' \implies (\text{name}, \text{pres}, \text{effs}, \text{layer}) \in \text{set } \text{ast}\delta \implies (\text{name}', \text{pres}', \text{effs}', \text{layer}') \in \text{set } \text{ast}\delta$
 $\implies \text{effs} = \text{effs}'$
using *eq-key-imp-eq-value[OF wf-operators(1)]*
by *auto*

lemma *is-applicable-then-subsumes*:
 $s \in \text{valid-states} \implies$

```

SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator (name,
pres, effs, layer))  $\implies$ 
  s  $\in$  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'*:

$$\llbracket s \in \text{valid-states} ; \text{is-standard-operator}' (\text{name}, \text{pres}, \text{effs}, \text{layer}) ; (\text{name}, \text{pres}, \text{effs}, \text{layer}) \in \text{set ast}\delta ;$$

$$\text{lookup-operator } ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) (\text{name}, \text{pres}, \text{effs}, \text{layer})) = \text{Some } (\text{name}', \text{pres}', \text{effs}', \text{layer}') \rrbracket$$

$$\implies \text{pres} = \text{pres}'$$

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-in δ])

lemma *is-applicable-then-enabled-2*:

$$\llbracket s \in \text{valid-states} ; \text{ast-op} \in \text{set ast}\delta ;$$

$$\text{SAS-Plus-Representation.is-operator-applicable-in } s (\text{abs-ast-operator ast-op}) ;$$

$$\text{lookup-operator } ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) \text{ ast-op}) = \text{Some } (\text{name}, \text{pres}, \text{effs}, \text{layer}) \rrbracket$$

$$\implies s \in \text{subsuming-states} (\text{map-of pres})$$

apply (*cases ast-op*)

using *eq-name-eq-pres' is-applicable-then-subsumes no-cond-effs*

by *fastforce*

lemma *is-applicable-then-enabled-3*:

$$\llbracket s \in \text{valid-states} ;$$

$$\text{lookup-operator } ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) \text{ ast-op}) = \text{Some } (\text{name}, \text{pres}, \text{effs}, \text{layer}) \rrbracket$$

$$\implies s \in \text{subsuming-states} (\text{map-of } (\text{implicit-pres effs}))$$

apply (*cases ast-op*)

using *no-cond-effs*

by (*auto dest!: std-then-implici-effs std-lookup-action lookup-Some-ind*

simp: subsuming-states-def)

lemma *is-applicable-then-enabled*:

$$\llbracket s \in \text{valid-states} ; \text{ast-op} \in \text{set ast}\delta ;$$

$$\text{SAS-Plus-Representation.is-operator-applicable-in } s (\text{abs-ast-operator ast-op}) \rrbracket$$

$$\implies \text{enabled } ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) \text{ ast-op}) \text{ s}$$

using *is-applicable-then-enabled-1 is-applicable-then-enabled-2 is-applicable-then-enabled-3*

by (*simp add: enabled-def split: option.splits*)

lemma *eq-name-eq-effs'*:

assumes *lookup-operator ((fst o the o lookup-action o abs-ast-operator) (name, pres, effs, layer)) = Some (name', pres', effs', layer')*

is-standard-operator' (name, pres, effs, layer) (name, pres, effs, layer) \in set ast δ

```

 $s \in valid-states$ 
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[ $OF lookup-operator-eq-name lookup-action-Some-in-\delta$ 
      lookup-Some-in $\delta$ ])

lemma std-eff-enabled'[simp]:
  is-standard-operator' (name, pres, effs, layer)  $\implies s \in valid-states \implies (\text{filter}$ 
  (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:
   $\llbracket s \in valid-states; ast\text{-}op \in set ast\delta;$ 
  SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator ast\text{-}op)  $\rrbracket$ 
 $\implies$ 
  execute ((fst o the o lookup-action o abs-ast-operator) ast\text{-}op) s =
  execute-operator-sas-plus s (abs-ast-operator ast\text{-}op)
using no-cond-effs
by(cases ast\text{-}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  $\wedge$ 
     sat-preconds-as (execute-operator-sas-plus s op) ops)

lemma exec-serial-then-path-to':
   $\llbracket s \in valid-states;$ 
   $\forall op \in set ops. \exists ast\text{-}op \in set ast\delta. op = abs-ast-operator ast\text{-}op;$ 
  (sat-preconds-as s ops)  $\implies$ 
  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-condless: execute-serial-plan-sas-plus s (rem-condless-ops s ops)
= execute-serial-plan-sas-plus s ops
by (induction ops arbitrary: s) auto

lemma rem-condless-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)  $\implies$  x ∈ set ops
by (induction ops arbitrary: s) auto

lemma exec-serial-then-path-to:
[ $s \in \text{valid-states}$ ;
  $\forall op \in \text{set ops}. \exists ast\text{-}op \in \text{set ast}\delta. op = \text{abs-ast-operator } ast\text{-}op$ ]  $\implies$ 
 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-condless-sat
by (fastforce dest!: set-rem-condlessD
 intro!: exec-serial-then-path-to'
 [where s = s and ops = rem-condless-ops s ops,
 unfolded exec-rem-condless])

lemma is-serial-solution-then-abstracted:
 is-serial-solution-for-problem abs-prob ops
  $\implies \forall op \in \text{set ops}. \exists ast\text{-}op \in \text{set ast}\delta. op = \text{abs-ast-operator } ast\text{-}op$ 
by(auto simp: is-serial-solution-for-problem-def abs-prob-def Let-def list.pred-set
 ListMem-iff abs-ast-operator-section-def
 split: if-splits)

lemma lookup-operator-works-1': lookup-action op = Some  $\pi' \implies \exists op. \text{lookup-operator } (\text{fst } \pi') = op$ 
using lookup-operator-works-1 by auto

lemma is-serial-sol-then-valid-plan-1:
 [is-serial-solution-for-problem abs-prob ops;
  $\pi \in \text{set } ((\text{map } (\text{fst } o \text{ the } o \text{ lookup-action}) o \text{ rem-condless-ops } I) \text{ ops})$ ]  $\implies$ 
 lookup-operator  $\pi \neq \text{None}$ 
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:

```

```

 $\llbracket \text{is-serial-solution-for-problem abs-prob ops} \rrbracket \implies$ 
 $(\exists s' \in G. \text{path-to } I ((\text{map } (\text{fst } o \text{ the } o \text{ lookup-action}) o \text{ rem-condless-ops } I) \text{ ops})$ 
 $s')$ 
using  $I$ -valid
by (fastforce intro: path-to-pres-valid exec-serial-then-path-to
      intro!: bexI[where  $x = \text{execute-serial-plan-sas-plus } I \text{ ops}$ ]
      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  $\equiv$  (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:
 $\llbracket \text{is-serial-solution-for-problem abs-prob ops} \rrbracket \implies$ 
 $\text{valid-plan } (\text{decode-abs-plan } \text{ops})$ 
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:
   $\mathcal{A} \models \Phi_{\forall} (\varphi (\text{prob-with-noop abs-prob})) t \implies$ 
    valid-plan
    (decode-abs-plan
      (rem-noops
        (map (\lambda op.  $\varphi_O^{-1}$  (\text{prob-with-noop abs-prob}) op)
          (concat ( $\Phi^{-1}$  (\varphi (\text{prob-with-noop abs-prob}))  $\mathcal{A} t))))))
  by (fastforce intro!: is-serial-sol-then-valid-plan abs-prob-valid
    sas-plus-problem-has-serial-solution-iff-i')

lemma length-abs-ast-plan: length  $\pi s = \text{length} (\text{abs-ast-plan } \pi s)$ 
  by (auto simp: abs-ast-plan-def)

theorem valid-plan-then-is-serial-sol-encoded:
  valid-plan  $\pi s \implies \text{length } \pi s \leq h \implies \exists \mathcal{A}. \mathcal{A} \models \Phi_{\forall} (\varphi (\text{prob-with-noop abs-prob})) h$ 
  apply(subst (asm) length-abs-ast-plan)
  by (fastforce intro!: sas-plus-problem-has-serial-solution-iff-ii' abs-prob-valid
    valid-plan-then-is-serial-sol)
end$ 
```

12 DIMACS-like semantics for CNF formulae

We now push the SAT encoding towards a lower-level representation by replacing the atoms which have variable IDs and time steps into natural numbers.

```

lemma gtD: ((l::nat) < n)  $\implies$  ( $\exists m. n = \text{Suc } m \wedge l \leq m$ )
  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  $\equiv$ 
    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)  $\longleftrightarrow$  t < h  $\wedge$  k < n-ops

```

```

| valid-state-var (State t k)  $\longleftrightarrow$  t < h

lemma State-works:
  valid-state-var (State t k)  $\implies$ 
    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)  $\implies$ 
    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  $\implies$ 
    dimacs-to-var (var-to-dimacs sv) = sv
  apply(cases sv)
  using State-works Operator-works
  by auto

end

lemma changing-atoms-works:
  ( $\bigwedge x. P x \implies (f o g) x = x$ )  $\implies (\forall x \in atoms \phi. P x) \implies M \models \phi \longleftrightarrow M o f$ 
   $\models map-formula g \phi$ 
  by (induction phi) auto

lemma changing-atoms-works':
   $M o g \models \phi \longleftrightarrow M \models map-formula g \phi$ 
  by (induction phi) auto

context cnf-to-dimacs
begin

lemma sat-plan-to-dimacs:
  ( $\bigwedge sv. sv \in atoms \text{ sat-plan-formula} \implies \text{valid-state-var } sv$ )  $\implies$ 
    M  $\models \text{sat-plan-formula}$ 
     $\longleftrightarrow M o \text{dimacs-to-var} \models map-formula \text{ var-to-dimacs sat-plan-formula}$ 
  by(auto intro!: changing-atoms-works[where P = valid-state-var] simp: sat-plan-to-dimacs-works)

lemma dimacs-to-sat-plan:
   $M o \text{var-to-dimacs} \models \text{sat-plan-formula}$ 
   $\longleftrightarrow M \models map-formula \text{ var-to-dimacs sat-plan-formula}$ 
  using changing-atoms-works' .

end

```

```

locale sat-solve-sasp = abs-ast-prob  $\Pi + \text{cnf-to-dimacs }$  Suc h Suc (length ast $\delta$ )
  for  $\Pi$  h
begin

lemma encode-initial-state-valid:
  sv  $\in$  atoms (encode-initial-state Prob)  $\implies$  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 ( $\varphi$  (prob-with-noop abs-prob))) = Suc
  (length ast $\delta$ )
  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  $\implies$  op  $\in$  set (operators-of ( $\varphi$  (prob-with-noop
  abs-prob)))  $\implies$ 
  sv  $\in$  atoms
  ( $\bigwedge$ (map ( $\lambda v.$ 
     $\neg$ (Atom (Operator t (index (operators-of ( $\varphi$  (prob-with-noop abs-prob)))) op)))
   $\vee$  Atom (State (Suc t) (index vs v)))
  asses))  $\implies$ 
  valid-state-var sv
  using length-operators
  by (induction asses) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)

lemma encode-operator-effect-valid-2: t < h  $\implies$  op  $\in$  set (operators-of ( $\varphi$  (prob-with-noop
  abs-prob)))  $\implies$ 
  sv  $\in$  atoms
  ( $\bigwedge$ (map ( $\lambda v.$ 
     $\neg$ (Atom (Operator t (index (operators-of ( $\varphi$  (prob-with-noop abs-prob)))) op)))
   $\vee$   $\neg$  (Atom (State (Suc t) (index vs v))))
  asses))  $\implies$ 
  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 ( $\bigwedge$  (as1 @ as2)) = atoms ( $\bigwedge$  as1)  $\cup$  atoms
  ( $\bigwedge$  as2)
  by (induction as1) auto

context sat-solve-sasp
begin

lemma encode-operator-effect-valid:

```

```

 $sv \in atoms (\text{encode-operator-effect} (\varphi (\text{prob-with-noop abs-prob})) t op) \implies$ 
 $t < h \implies op \in set (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \implies$ 
 $\text{valid-state-var } sv$ 
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 ( $\wedge$ ) as ( $\neg \perp$ ) = ( $\wedge$  as)
by (induction as) auto

context sat-solve-sasp
begin

lemma encode-all-operator-effects-valid:
 $t < Suc h \implies$ 
 $sv \in atoms (\text{encode-all-operator-effects} (\varphi (\text{prob-with-noop abs-prob})) (\text{operators-of}$ 
 $(\varphi (\text{prob-with-noop abs-prob}))) t) \implies$ 
 $\text{valid-state-var } sv$ 
unfolding encode-all-operator-effects-def foldr-And
by (force simp add: encode-operator-effect-valid)

lemma encode-operator-precondition-valid-1:
 $t < h \implies op \in set (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \implies$ 
 $sv \in atoms$ 
 $(\bigwedge (\text{map} (\lambda v.$ 
 $\neg (\text{Atom} (\text{Operator} t (\text{index} (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob})))$ 
 $op))) \vee \text{Atom} (\text{State} t (f v)))$ 
 $\text{asses})) \implies$ 
 $\text{valid-state-var } sv$ 
using length-operators
by (induction asses) (auto simp: simp add: cnf-to-dimacs.valid-state-var.simps)

lemma encode-operator-precondition-valid:
 $sv \in atoms (\text{encode-operator-precondition} (\varphi (\text{prob-with-noop abs-prob})) t op) \implies$ 
 $t < h \implies op \in set (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \implies$ 
 $\text{valid-state-var } sv$ 
by (force simp: encode-operator-precondition-def Let-def
      intro!: encode-operator-precondition-valid-1)

lemma encode-all-operator-preconditions-valid:
 $t < Suc h \implies$ 
 $sv \in atoms (\text{encode-all-operator-preconditions} (\varphi (\text{prob-with-noop abs-prob}))$ 
 $(\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) t) \implies$ 
 $\text{valid-state-var } sv$ 
unfolding encode-all-operator-preconditions-def foldr-And
by (force simp add: encode-operator-precondition-valid)

```

```

lemma encode-operators-valid:
  sv ∈ atoms (encode-operators (φ (prob-with-noop abs-prob)) t) ⇒ t < Suc h
  ⇒
    valid-state-var sv
  unfolding encode-operators-def Let-def
  by (force simp add: encode-all-operator-preconditions-valid encode-all-operator-effects-valid)

lemma encode-negative-transition-frame-axiom':
  t < h ⇒
  set deleting-operators ⊆ set (operators-of (φ (prob-with-noop abs-prob))) ⇒
  sv ∈ atoms
  (¬(Atom (State t v-idx))
   ∨ (Atom (State (Suc t) v-idx)
   ∨ √ (map (λop. Atom (Operator t (index (operators-of (φ (prob-with-noop
   abs-prob)) op))) deleting-operators))) ⇒
   valid-state-var sv
  by (induction deleting-operators) (auto simp: length-operators[symmetric] cnf-to-dimacs.valid-state-var.simps)

lemma encode-negative-transition-frame-axiom-valid:
  sv ∈ atoms (encode-negative-transition-frame-axiom (φ (prob-with-noop abs-prob))
  t v) ⇒ t < h ⇒
  valid-state-var sv
  unfolding encode-negative-transition-frame-axiom-def Let-def
  apply(intro encode-negative-transition-frame-axiom'[of t])
  by auto

lemma encode-positive-transition-frame-axiom-valid:
  sv ∈ atoms (encode-positive-transition-frame-axiom (φ (prob-with-noop abs-prob))
  t v) ⇒ t < h ⇒
  valid-state-var sv
  unfolding encode-positive-transition-frame-axiom-def Let-def
  apply(intro encode-negative-transition-frame-axiom'[of t])
  by auto

lemma encode-all-frame-axioms-valid:
  sv ∈ atoms (encode-all-frame-axioms (φ (prob-with-noop abs-prob)) t) ⇒ t <
  Suc h ⇒
  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 op1 op2) ⇒ t < Suc h ⇒
    op1 ∈ set (operators-of (φ (prob-with-noop abs-prob))) ⇒ op2 ∈ 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 πs ⇒ length πs ≤ h ⇒
  ∃ M. M ⊨ map-formula var-to-dimacs (Φ∀ (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:
  A ⊨ map-formula var-to-dimacs (Φ∀ (φ (prob-with-noop abs-prob)) t) ⇒
  valid-plan
  (decode-abs-plan
  (rem-noops
  (map (λop. φO-1 (prob-with-noop abs-prob) op)
  (concat (Φ-1 (φ (prob-with-noop abs-prob)) (A o var-to-dimacs) t))))))
using changing-atoms-works'
by (fastforce intro!: is-serial-sol-then-valid-plan-encoded)

end

```

12.1 Going from Formulae to DIMACS-like CNF

We now represent the CNF formulae into a very low-level representation that is reminiscent to the DIMACS representation, where a CNF formula is a list of list of integers.

```

fun disj-to-dimacs::nat formula  $\Rightarrow$  int list where
  disj-to-dimacs ( $\varphi_1 \vee \varphi_2$ ) = disj-to-dimacs  $\varphi_1$  @ disj-to-dimacs  $\varphi_2$ 
  | disj-to-dimacs  $\perp$  = []
  | disj-to-dimacs (Not  $\perp$ ) = [-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  $\varphi_1$  @ cnf-to-dimacs  $\varphi_2$ 
  | cnf-to-dimacs d = [disj-to-dimacs d]

definition dimacs-lit-to-var l  $\equiv$  nat (abs l)

definition find-max (xs::nat list)  $\equiv$  (fold max xs 1)

lemma find-max-works:
x  $\in$  set xs  $\Rightarrow$  x  $\leq$  find-max xs (is ?P  $\Rightarrow$  ?Q)
proof-
  have x  $\in$  set xs  $\Rightarrow$  (x::nat)  $\leq$  (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  $\Rightarrow$  ?Q
  by(auto simp add: find-max-def max-def)
qed

fun formula-vars where
formula-vars ( $\perp$ ) = [] |
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  $\in$  atoms (f::nat formula)  $\Rightarrow$  v  $\leq$  find-max (formula-vars f)
using find-max-works
by(simp add: atoms-formula-vars)

definition dimacs-max-var cs  $\equiv$  find-max (map (find-max o (map (nat o abs))) cs)

```

```

lemma fold-max-ge:  $b \leq a \implies (b::nat) \leq \text{fold } (\lambda x m. \text{ if } m \leq x \text{ then } x \text{ else } m) ys$ 
 $a$ 
by (induction ys arbitrary: a b) auto

lemma find-max-append:  $\text{find-max} (xs @ ys) = \max (\text{find-max} xs) (\text{find-max} ys)$ 
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  $\Rightarrow$  int list list  $\Rightarrow$  bool where
dimacs-model ls cs  $\equiv$  ( $\forall c \in \text{set} cs. (\exists l \in \text{set} ls. l \in \text{set} c)) \wedge$ 
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)  $\cup$ 
set (model-to-dimacs-model M vs')
by (induction vs) auto

lemma upt-append-sing:  $xs @ [x] = [a..<n-vars] \implies a < n-vars \implies (xs = [a..<n-vars$ 
– 1]  $\wedge x = n-vars - 1 \wedge n-vars > 0)$ 
by (induction n-vars) auto

lemma upt-eqD:  $upt a b = upt a b' \implies (b = b' \vee b' \leq a \vee b \leq a)$ 
by (induction b) (auto dest!: upt-append-sing split: if-splits)

lemma pos-in-model:  $M n \implies 0 < n \implies n < n-vars \implies \text{int } n \in \text{set} (\text{model-to-dimacs-model}$ 
M [1..<n-vars])
by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append
)

lemma neg-in-model:  $\neg M n \implies 0 < n \implies n < n-vars \implies -(\text{int } n) \in \text{set}$ 
(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:  $0 < n \implies n < n-vars \implies \text{int } n \in \text{set} (\text{model-to-dimacs-model}$ 
M [1..<n-vars])  $\vee -(\text{int } n) \in \text{set} (\text{model-to-dimacs-model } M [1..<n-vars])$ 
using pos-in-model neg-in-model
by metis

lemma model-to-dimacs-model-all-vars:
( $\forall v \in \text{atoms } f. 0 < v \wedge v < n-vars \implies \text{is-cnf } f \implies M \models f \implies$ 
 $(\forall n < n-vars. 0 < n \longrightarrow (\text{int } n \in \text{set} (\text{model-to-dimacs-model } M [(1::nat)..<n-vars]))$ 

```

```

 $\vee$ 
 $-(int n) \in set (model\text{-}to\text{-}dimacs\text{-}model M [(1::nat)..<n\text{-}vars]))$ 
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  $\wedge$  f2)) = set (cnf-to-dimacs f1)  $\cup$  set (cnf-to-dimacs f2)
by auto

lemma one-always-in:
 $1 < n\text{-}vars \implies 1 \in set (model\text{-}to\text{-}dimacs\text{-}model M ([1..<n\text{-}vars])) \vee -1 \in set (model\text{-}to\text{-}dimacs\text{-}model M ([1..<n\text{-}vars]))$ 
by (induction n-vars) (auto simp add: less-Suc-eq model-to-dimacs-model-append)

lemma [simp]: (disj-to-dimacs (f1  $\vee$  f2)) = (disj-to-dimacs f1) @ (disj-to-dimacs f2)
by auto

lemma [simp]: (atoms (f1  $\vee$  f2)) = atoms f1  $\cup$  atoms f2
by auto

lemma isdisj-disjD: (is-disj (f1  $\vee$  f2))  $\implies$  is-disj f1  $\wedge$  is-disj f2
by (cases f1; auto)

lemma disj-to-dimacs-sound:
 $1 < n\text{-}vars \implies (\forall v \in atoms f. 0 < v \wedge v < n\text{-}vars) \implies is\text{-}disj f \implies M \models f$ 
 $\implies \exists l \in set (model\text{-}to\text{-}dimacs\text{-}model M [(1::nat)..<n\text{-}vars]). l \in set (disj\text{-}to\text{-}dimacs f)$ 
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  $\vee$  f2)  $\implies$  ( $\bigwedge f. f1 \vee f2 = f \implies is\text{-}disj f \implies P$ )  $\implies$ 
P
by auto

lemma cnf-to-dimacs-disj: is-disj f  $\implies$  cnf-to-dimacs f = [disj-to-dimacs f]
by (induction f) auto

lemma model-to-dimacs-model-all-clauses:
 $1 < n\text{-}vars \implies (\forall v \in atoms f. 0 < v \wedge v < n\text{-}vars) \implies is\text{-}cnf f \implies M \models f \implies$ 
 $c \in set (cnf\text{-}to\text{-}dimacs f) \implies \exists l \in set (model\text{-}to\text{-}dimacs\text{-}model M [(1::nat)..<n\text{-}vars]). l \in set c$ 
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 upto-eq-Cons-conv:
  (x#xs = [i..<j]) = (i < j  $\wedge$  i = x  $\wedge$  [i+1..<j] = xs)
  using upto-eq-Cons-conv
  by metis

lemma model-to-dimacs-model-append':
  (model-to-dimacs-model M (vs @ vs')) = (model-to-dimacs-model M vs) @ (model-to-dimacs-model M vs')
  by (induction vs) auto

lemma model-to-dimacs-neg-nin:
  n-vars  $\leq$  x  $\implies$  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  $\implies$  - 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  $\implies$  ( $\bigwedge$ n. 0  $\neq$  (int n)  $\implies$  z = int n  $\implies$  P)  $\implies$  ( $\bigwedge$ n. 0  $\neq$  - (int n)  $\implies$ 
  z = - (int n)  $\implies$  P)  $\implies$  P
  by (metis (full-types) int-cases2)

lemma model-to-dimacs-model-distinct:
  1 < n-vars  $\implies$  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  $\implies$  ( $\forall v \in atoms f$ . 0 < v  $\wedge$  v < n-vars)  $\implies$  is-cnf f  $\implies$  M  $\models f$   $\implies$ 
  dimacs-model (model-to-dimacs-model M [(1::nat)..<n-vars]) (cnf-to-dimacs
f)
  unfold 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-vars  $\implies$  ( $\forall v \in atoms f$ . 0 < v  $\wedge$  v < n-vars)  $\implies$  is-cnf f  $\implies$  M  $\models f$   $\implies$ 
   $\exists M\text{-dimacs. dimacs-model } M\text{-dimacs} (cnf-to-dimacs } f)$ 
  using model-to-dimacs-model-sound
  by metis

```

```

definition dimacs-to-atom ::int  $\Rightarrow$  nat formula where
  dimacs-to-atom  $l \equiv$  if ( $l < 0$ ) then Not (Atom (nat (abs  $l$ ))) else Atom (nat (abs  $l$ ))

definition dimacs-to-disj::int list  $\Rightarrow$  nat formula where
  dimacs-to-disj  $f \equiv \bigvee (\text{map dimacs-to-atom } f)$ 

definition dimacs-to-cnf::int list list  $\Rightarrow$  nat formula where
  dimacs-to-cnf  $f \equiv \bigwedge \text{map dimacs-to-disj } f$ 

definition dimacs-model-to-abs dimacs-M  $M \equiv$ 
  fold ( $\lambda l M$ . if ( $l > 0$ ) then  $M((\text{nat (abs } l)) := \text{True})$  else  $M((\text{nat (abs } l)) := \text{False}))$ 
  dimacs-M  $M$ 

lemma dimacs-model-to-abs-atom:
   $0 < x \implies \text{int } x \in \text{set dimacs-M} \implies \text{distinct } (\text{map dimacs-lit-to-var dimacs-M})$ 
   $\implies \text{dimacs-model-to-abs dimacs-M } x$ 
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':
   $0 < x \implies -(\text{int } x) \in \text{set dimacs-M} \implies \text{distinct } (\text{map dimacs-lit-to-var dimacs-M}) \implies \neg \text{dimacs-model-to-abs dimacs-M } x$ 
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:
   $(\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-disj } f \implies \text{distinct } (\text{map dimacs-lit-to-var dimacs-M})$ 
   $\implies \text{dimacs-model dimacs-M (cnf-to-dimacs } f) \implies \text{dimacs-model-to-abs dimacs-M } (\lambda \_. \text{False}) \models f$ 
  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:
   $(\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnff } f \implies \text{distinct } (\text{map dimacs-lit-to-var dimacs-M})$ 
   $\implies \text{dimacs-model dimacs-M (cnf-to-dimacs } f) \implies \text{dimacs-model-to-abs dimacs-M } (\lambda \_. \text{False}) \models f$ 
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 ==> 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 ==> length πs ≤ h ==>
let cnf-formula = (map-formula var-to-dimacs
(Φ ∀ (φ (prob-with-noop abs-prob)) h))
in
∃ dimacs-M. dimacs-model dimacs-M (cnf-to-dimacs cnf-formula)
unfolding Let-def
by (fastforce simp: var-to-dimacs-pos)

```

```

dest!: planning-by-cnf-dimacs-complete
intro: model-to-dimacs-model-sound-max-var map-is-cnf
is-cnf-encode-problem-with-operator-interference-exclusion
is-valid-problem-sas-plus-then-strips-transformation-too
noops-valid abs-prob-valid)

lemma planning-dimacs-sound:
let cnf-formula =
(map-formula var-to-dimacs
( $\Phi_{\forall} (\varphi (\text{prob-with-noop abs-prob})) h$ )
in
dimacs-model dimacs-M (cnf-to-dimacs cnf-formula)  $\implies$ 
valid-plan
(decode-abs-plan
(rem-noops
(map ( $\lambda op. \varphi_O^{-1} (\text{prob-with-noop abs-prob}) op$ )
(concat
( $\Phi^{-1} (\varphi (\text{prob-with-noop abs-prob})) ((\text{dimacs-model-to-abs dimacs-M}$ 
( $\lambda -. \text{False})) o \text{var-to-dimacs} h)))))$ 
by(fastforce simp: var-to-dimacs-pos Let-def
intro: planning-by-cnf-dimacs-sound model-to-dimacs-model-complete-max-var
map-is-cnf is-cnf-encode-problem-with-operator-interference-exclusion
is-valid-problem-sas-plus-then-strips-transformation-too abs-prob-valid
noops-valid)

end

```

13 Code Generation

We now generate SML code equivalent to the functions that encode a problem as a CNF formula and that decode the model of the given encodings into a plan.

definition

```

SASP-to-DIMACS h prob  $\equiv$ 
cnf-to-dimacs
(map-formula
(cnf-to-dimacs.var-to-dimacs (Suc h) (Suc (length (ast-problem.astδ prob))))
( $\Phi_{\forall} (\varphi (\text{prob-with-noop (ast-problem.abs-prob prob})) h$ )

```

lemma *planning-dimacs-complete-code:*

```

[ast-problem.well-formed prob;
 $\forall \pi \in \text{set (ast-problem.ast}\delta \text{ prob). is-standard-operator}' \pi;$ 
ast-problem.valid-plan prob πs;
length πs  $\leq h$ ]  $\implies$ 
let cnf-formula = (SASP-to-DIMACS h prob) in
   $\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}$ 
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':
   $\llbracket \text{ast-problem.well-formed prob} ;$ 
     $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}) \implies \text{consistent-pres-op } op) ;$ 
     $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}) \implies \text{is-standard-operator } op) ;$ 
     $\text{ast-problem.valid-plan prob } \pi s ;$ 
     $\text{length } \pi s \leq h \rrbracket \implies$ 
  let cnf-formula = (SASP-to-DIMACS' h prob) in
     $\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}$ 
  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-indD)

```

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 ( $\forall op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}). \text{consistent-pres-op } op$ ) then
      if ( $\forall op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}). \text{is-standard-operator } op$ ) 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:

```

 $\llbracket \text{ast-problem.valid-plan prob } \pi s ; \text{length } \pi s \leq h ;$ 
  encode h prob = Inl cnf-formula  $\rrbracket \implies$ 
  ( $\exists \text{dimacs-M. dimacs-model dimacs-M cnf-formula}$ )
  unfolding encode-def
  by (auto split: if-splits simp: list.pred-set
        intro: planning-dimacs-complete-code'[unfolded Let-def])

```

lemma encode-complete:

```

encode h prob = Inr err  $\implies$ 
 $\neg(\text{ast-problem.well-formed prob} \wedge (\forall op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}). \text{consis-}$ 

```

```

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(clar 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  $\implies \exists p. p \in \text{set}(\text{inorder } T) \wedge p = (x, y)$ 
  by (induction T) (auto split: if-splits)

lemma map-of-lookup: sorted1 (inorder T)  $\implies$  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: ( $\bigwedge x. m1 x = m2 x$ )  $\implies m1 \subseteq_m s \longleftrightarrow m2 \subseteq_m s$ 
  by presburger

lemma match-pres-submap:
  match-pres (inorder (M.tree-map-of' empty pres)) s  $\longleftrightarrow$  Map.map-of pres  $\subseteq_m s$ 
  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  $\longleftrightarrow$ 
  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  $\equiv$ 
  (ast-problem.decode-abs-plan prob
    (rem-noops
      (map (λop.  $\varphi_O^{-1}$  (prob-with-noop (ast-problem.abs-prob prob)) op)
        (concat
          ( $\Phi^{-1}$  (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:
   $\llbracket \text{ast-problem.well-formed prob} ;$ 
   $\forall \pi \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}). \text{is-standard-operator}' \pi \rrbracket \implies$ 
  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':
   $\llbracket \text{ast-problem.well-formed prob} ;$ 
   $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}) \implies \text{consistent-pres-op } op) ;$ 
   $\forall \pi \in \text{set}(\text{ast-problem.ast}\delta \text{ prob}). \text{is-standard-operator } \pi \rrbracket \implies$ 
  let
    cnf-formula = (SASP-to-DIMACS' h prob);
    decoded-plan = decode-DIMACS-model' dimacs-M h prob
  in
    (dimacs-model dimacs-M cnf-formula → ast-problem.valid-plan prob decoded-plan)
  unfolding SASP-to-DIMACS'-def decode-DIMACS-model'-def
  apply(subst rem-implicit-pres-ops-valid-plan[symmetric])
  by(fastforce simp only: rem-implicit-pres-ops-valid-plan wf-ast-problem-def
      intro!: rem-implicit-pres-is-standard-operator'
      rem-implicit-pres-ops-well-formed
      rev-iffD2[OF - rem-implicit-pres-ops-valid-plan]
      planning-dimacs-sound-code wf-ast-problem.intro
      dest!: rem-implicit-pres-ops-indD)+
```

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)  $\wedge$  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  $\longleftrightarrow$ 
    (let tls = list-to-rbt ls in
     ( $\forall c \in set cs.$  size (inter-rbt (tls) (list-to-rbt c))  $\neq 0$ )  $\wedge$ 
      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  $\equiv$ 
  if ast-problem.well-formed prob then
    if ( $\forall op \in set (ast\text{-problem}.ast\delta prob).$  consistent-pres-op op) then
      if ( $\forall op \in set (ast\text{-problem}.ast\delta prob).$  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  $\implies$ 
    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  $\implies$ 
     $\neg (ast\text{-problem.well-formed prob} \wedge$ 
    ( $\forall op \in set (ast\text{-problem}.ast\delta prob).$  consistent-pres-op op)  $\wedge$ 
    ( $\forall \pi \in set (ast\text{-problem}.ast\delta prob).$  is-standard-operator \pi)  $\wedge$ 
    dimacs-model M (SASP-to-DIMACS' h prob))

```

```

unfolding decode-def
by (auto split: if-splits simp: list.pred-set)

lemma [code]:
  ‹ListMem x xs ↔ List.member xs x›
  by (simp add: ListMem-iff)

lemmas [code] = 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
  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
lemmas [code] = cnf-to-dimacs.var-to-dimacs.simps

definition nat-opt-of-integer :: integer ⇒ nat option where
  nat-opt-of-integer i = (if (i ≥ 0) then Some (nat-of-integer i) else None)

definition max-var :: int list ⇒ int where
  max-var xs ≡ fold (λ(x:int) (y:int). if abs x ≥ abs y then (abs x) else y) xs
  (0:int)

export-code encode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode
  String.implode max-var concat char-of-nat Int.nat integer-of-int length int-of-integer
  in SML module-name exported file-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>.