

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	8
3.3	Serializable Parallel Plans	16
3.4	Auxiliary lemmas about STRIPS	18
4	SAS+ Representation	18
5	SAS+ Semantics	21
5.1	Serial Execution Semantics	21
5.2	Parallel Execution Semantics	22
5.3	Serializable Parallel Plans	26
5.4	Auxiliary lemmata on SAS+	27
6	SAS+/STRIPS Equivalence	28
6.1	Translation of SAS+ Problems to STRIPS Problems	28
6.2	Equivalence of SAS+ and STRIPS	41

*Author names are alphabetically ordered.

7	The Basic SATPlan Encoding	44
7.1	Encoding Function Definitions	44
7.2	Decoding Function Definitions	48
7.3	Soundness of the Basic SATPlan Algorithm	55
7.4	Completeness	59
8	Serializable SATPlan Encodings	67
8.1	Soundness	71
8.2	Completeness	71
9	SAT-Solving of SAS+ Problems	73
10	Adding Noop actions to the SAS+ problem	74
11	Proving Equivalence of SAS+ representation and Fast-Downward's Multi-Valued Problem Representation	76
11.1	Translating Fast-Downward's represnetation to SAS+	76
11.2	Translating SAS+ represnetation to Fast-Downward's	88
11.3	SAT encoding works for Fast-Downward's representation	93
12	DIMACS-like semantics for CNF formulae	94
12.1	Going from Formualae to DIMACS-like CNF	99
13	Code Generation	104

```

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 ($\lambda v. (v, \text{True})$) (*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 ($\langle(-V)\rangle$ [1000] 999)
 operators-of :: 'variable strips-operator list ($\langle(-O)\rangle$ [1000] 999)
 initial-of :: 'variable strips-state ($\langle(-I)\rangle$ [1000] 999)
 goal-of :: 'variable strips-state ($\langle(-G)\rangle$ [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** $\langle\gg\rangle$ 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>
```

lemma *is-valid-problem-dom-of-goal-state-is*:

fixes Π :: 'a *strips-problem*
assumes *is-valid-problem-strips* Π
shows $\text{dom } ((\Pi)_G) \subseteq \text{set } ((\Pi)_V)$
(*proof*)

lemma *is-valid-problem-strips-operator-variable-sets*:

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

lemma *effect-to-assignments-i*:

assumes $as = \text{effect-to-assignments } op$
shows $as = (\text{map } (\lambda v. (v, \text{True})) (\text{add-effects-of } op))$
 $\quad @ \text{map } (\lambda v. (v, \text{False})) (\text{delete-effects-of } op)$
(*proof*)

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 $as = \text{effect-to-assignments } op$
obtains $as_1 as_2$
where $as = as_1 @ as_2$
and $as_1 = \text{map } (\lambda v. (v, \text{True})) (\text{add-effects-of } op)$
and $as_2 = \text{map } (\lambda v. (v, \text{False})) (\text{delete-effects-of } op)$
(*proof*)

lemma *effect-to-assignments-iii-a*:

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

lemma *effect-to-assignments-iii-b*:

— NOTE This proof is symmetrical to the one above.

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

lemma *effect--strips-i*:

fixes op

assumes $e = \text{effect--strips } op$
obtains $es_1 \ es_2$
where $e = (es_1 \ @ \ es_2)$
and $es_1 = \text{map } (\lambda v. (v, \text{True})) \ (\text{add-effects-of } op)$
and $es_2 = \text{map } (\lambda v. (v, \text{False})) \ (\text{delete-effects-of } op)$
 $\langle \text{proof} \rangle$

lemma *effect--strips-ii:*

fixes op
assumes $e = \text{ConjunctiveEffect } (es_1 \ @ \ es_2)$
and $es_1 = \text{map } (\lambda v. (v, \text{True})) \ (\text{add-effects-of } op)$
and $es_2 = \text{map } (\lambda v. (v, \text{False})) \ (\text{delete-effects-of } op)$
shows $\forall v \in \text{set } (\text{add-effects-of } op). (\exists e' \in \text{set } es_1. e' = (v, \text{True}))$
and $\forall v \in \text{set } (\text{delete-effects-of } op). (\exists e' \in \text{set } es_2. e' = (v, \text{False}))$
 $\langle \text{proof} \rangle$

lemma *map-of-constant-assignments-dom:*

— NOTE ancillary lemma used in the proof below.
assumes $m = \text{map-of } (\text{map } (\lambda v. (v, d)) \ vs)$
shows $\text{dom } m = \text{set } vs$
 $\langle \text{proof} \rangle$

lemma *effect--strips-iii-a:*

assumes $s' = (s \gg op)$
shows $\bigwedge v. v \in \text{set } (\text{add-effects-of } op) \implies s' v = \text{Some True}$
 $\langle \text{proof} \rangle$

lemma *effect--strips-iii-b:*

assumes $s' = (s \gg op)$
shows $\bigwedge v. v \in \text{set } (\text{delete-effects-of } op) \wedge v \notin \text{set } (\text{add-effects-of } op) \implies s' v = \text{Some False}$
 $\langle \text{proof} \rangle$

lemma *effect--strips-iii-c:*

assumes $s' = (s \gg op)$
shows $\bigwedge v. v \notin \text{set } (\text{add-effects-of } op) \wedge v \notin \text{set } (\text{delete-effects-of } op) \implies s' v = s v$
 $\langle \text{proof} \rangle$

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,

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

- 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

$\bigwedge v.$

$v \in \text{set } (add\text{-effects-of } op)$

$\implies s' v = \text{Some True}$

and $\bigwedge v.$

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

$\implies s' v = \text{Some False}$

and $\bigwedge v.$

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

$\implies s' v = s v$

<proof>

3.2 Parallel Plan Semantics

definition *are-all-operators-applicable* $s ops$

$\equiv \text{list-all } (\lambda op. \text{is-operator-applicable-in } s op) ops$

definition *are-operator-effects-consistent* $op_1 op_2 \equiv \text{let}$

$add_1 = \text{add-effects-of } op_1$

$; add_2 = \text{add-effects-of } op_2$

$; del_1 = \text{delete-effects-of } op_1$

$; del_2 = \text{delete-effects-of } op_2$

$\text{in } \neg \text{list-ex } (\lambda v. \text{list-ex } ((=) v) del_2) add_1 \wedge \neg \text{list-ex } (\lambda v. \text{list-ex } ((=) v) add_2) del_1$

definition *are-all-operator-effects-consistent* $ops \equiv$

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

definition *execute-parallel-operator*

$:: 'variable \text{ strips-state}$

$\Rightarrow 'variable \text{ strips-operator list}$

$\Rightarrow 'variable \text{ strips-state}$

where *execute-parallel-operator* $s ops$

$\equiv \text{foldl } (++) s (\text{map } (\text{map-of } \circ \text{effect-to-assignments}) ops)$

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

fun *execute-parallel-plan*

```

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

```

definition *are-operators-interfering* $op_1 op_2$
 $\equiv list-ex (\lambda v. list-ex ((=) v) (delete-effects-of op_1)) (precondition-of op_2)$
 $\vee list-ex (\lambda v. list-ex ((=) v) (precondition-of op_1)) (delete-effects-of op_2)$

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 ($\lambda op'. \neg are-operators-interfering op op'$) ops
 $\wedge are-all-operators-non-interfering ops$)

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 \Rightarrow 'variable strips-parallel-plan \Rightarrow '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* $\Pi \pi$
 $\equiv (strips-problem.goal-of \Pi) \subseteq_m execute-parallel-plan$
 $(strips-problem.initial-of \Pi) \pi$
 $\wedge list-all (\lambda ops. list-all (\lambda op.$
 $ListMem op (strips-problem.operators-of \Pi)) ops) \pi$

lemma *are-all-operators-applicable-set:*
are-all-operators-applicable s ops
 $\longleftrightarrow (\forall op \in set ops. \forall v \in set (precondition-of op). s v = Some True)$

<proof>

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>

lemma *are-operator-effects-consistent-set:*
assumes $op_1 \in \text{set ops}$
and $op_2 \in \text{set ops}$
shows *are-operator-effects-consistent op₁ op₂*
 $= (\text{set (add-effects-of } op_1) \cap \text{set (delete-effects-of } op_2) = \{\})$
 $\wedge \text{set (delete-effects-of } op_1) \cap \text{set (add-effects-of } op_2) = \{\})$
<proof>

lemma *are-all-operator-effects-consistent-set:*
are-all-operator-effects-consistent ops
 $\longleftrightarrow (\forall op_1 \in \text{set ops. } \forall op_2 \in \text{set ops.}$
 $\text{set (add-effects-of } op_1) \cap \text{set (delete-effects-of } op_2) = \{\})$
 $\wedge \text{set (delete-effects-of } op_1) \cap \text{set (add-effects-of } op_2) = \{\})$
<proof>

lemma *are-all-effects-consistent-tail:*
assumes *are-all-operator-effects-consistent (op # ops)*
shows *are-all-operator-effects-consistent ops*
<proof>

lemma *are-all-operators-non-interfering-tail:*
assumes *are-all-operators-non-interfering (op # ops)*
shows *are-all-operators-non-interfering ops*
<proof>

lemma *are-operators-interfering-symmetric:*
assumes *are-operators-interfering op₁ op₂*
shows *are-operators-interfering op₂ op₁*
<proof>

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 $op_1 \neq op_2$
shows $op_1 \notin \text{set ops} \vee op_2 \notin \text{set ops}$
<proof>

lemma *execute-parallel-plan-precondition-cons-i:*
fixes $s :: ('variable, bool) \text{state}$
assumes $\neg \text{are-operators-interfering } op \ op'$

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*⟩

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
 ⟨*proof*⟩

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

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*⟩

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*⟩

lemma *effect-to-assignments-simp[simp]*: *effect-to-assignments* op
 = *map* ($\lambda v. (v, True)$) (*add-effects-of* op) @ *map* ($\lambda v. (v, False)$) (*delete-effects-of*
 op)
 ⟨*proof*⟩

lemma *effect-to-assignments-set-is[simp]*:
set (*effect-to-assignments* op) = { $((v, a), True) \mid v a. (v, a) \in \text{set } (add-effects-of$
 $op)$ }
 ∪ { $((v, a), False) \mid v a. (v, a) \in \text{set } (delete-effects-of$ $op)$ }
 ⟨*proof*⟩

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*
 ($\lambda v. (v, \text{if } ListMem\ v\ (add-effects-of\ op)\ \text{then } True\ \text{else } False)$)
 (*add-effects-of* op @ *delete-effects-of* op)

and *effect-to-assignments op = map*
 $(\lambda v. (v, \text{if ListMem } v \text{ (delete-effects-of op) then False else True}))$
 $(\text{add-effects-of op @ delete-effects-of op})$
 $\langle \text{proof} \rangle$

corollary *map-of-effect-to-assignments-is-none-if:*

assumes $\neg v \in \text{set (add-effects-of op)}$
and $\neg v \in \text{set (delete-effects-of op)}$
shows *map-of (effect-to-assignments op) v = None*
 $\langle \text{proof} \rangle$

lemma *execute-parallel-operator-positive-effect-if-i:*

assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and $op \in \text{set ops}$
and $v \in \text{set (add-effects-of op)}$
shows *map-of (effect-to-assignments op) v = Some True*
 $\langle \text{proof} \rangle$

lemma *execute-parallel-operator-positive-effect-if:*

fixes *ops*
assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and $op \in \text{set ops}$
and $v \in \text{set (add-effects-of op)}$
shows *execute-parallel-operator s ops v = Some True*
 $\langle \text{proof} \rangle$

lemma *execute-parallel-operator-negative-effect-if-i:*

assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and $op \in \text{set ops}$
and $v \in \text{set (delete-effects-of op)}$
shows *map-of (effect-to-assignments op) v = Some False*
 $\langle \text{proof} \rangle$

lemma *execute-parallel-operator-negative-effect-if:*

assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and $op \in \text{set ops}$
and $v \in \text{set (delete-effects-of op)}$
shows *execute-parallel-operator s ops v = Some False*
 $\langle \text{proof} \rangle$

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

assumes $\forall op \in \text{set ops. } \neg v \in \text{set (add-effects-of op)} \wedge \neg v \in \text{set (delete-effects-of op)}$
shows *execute-parallel-operator s ops v = s v*
 $\langle \text{proof} \rangle$

corollary *execute-parallel-operators-strips-none-if:*

assumes $\forall op \in set\ ops. \neg v \in set\ (add\ effects\ of\ op) \wedge \neg v \in set\ (delete\ effects\ of\ op)$

and $s\ v = None$

shows $execute\ parallel\ operator\ s\ ops\ v = None$

<proof>

corollary *execute-parallel-operators-strips-none-if-contraposition:*

assumes $\neg execute\ parallel\ operator\ s\ ops\ v = None$

shows $(\exists op \in set\ ops. v \in set\ (add\ effects\ of\ op) \vee v \in set\ (delete\ effects\ of\ op))$

$\vee s\ v \neq None$

<proof>

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

$$s' \equiv execute\ parallel\ operator\ s\ ops$$

will evaluate *v* to true, that is

$$execute\ parallel\ operator\ s\ ops\ v = Some\ True$$

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

$$execute\ parallel\ operator\ s\ ops\ v = Some\ False$$

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

$$execute\ parallel\ operator\ s\ ops\ v = s\ v$$

theorem *execute-parallel-operator-effect:*

assumes *are-all-operators-applicable* $s\ ops$

and *are-all-operator-effects-consistent* ops

shows $op \in set\ ops \wedge v \in set\ (add\ effects\ of\ op)$

$\longrightarrow execute\ parallel\ operator\ s\ ops\ v = Some\ True$

and $op \in set\ ops \wedge v \in set\ (delete\ effects\ of\ op)$

$\longrightarrow execute\ parallel\ operator\ s\ ops\ v = Some\ False$

and $(\forall op \in set\ ops.$

$v \notin set\ (add\ effects\ of\ op) \wedge v \notin set\ (delete\ effects\ of\ op))$

$\longrightarrow execute\ parallel\ operator\ s\ ops\ v = s\ v$

<proof>

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

fixes Π :: 'a strips-problem

assumes *is-parallel-solution-for-problem* Π π

and $ops \in set\ \pi$

and $op \in set\ ops$

shows $op \in set\ ((\Pi)_O)$

<proof>

lemma *trace-parallel-plan-strips-not-nil:* *trace-parallel-plan-strips* I $\pi \neq []$

<proof>

corollary *length-trace-parallel-plan-gt-0[simp]:* $0 < length\ (trace-parallel-plan-strips\ I\ \pi)$

<proof>

corollary *length-trace-minus-one-lt-length-trace[simp]:*

$length\ (trace-parallel-plan-strips\ I\ \pi) - 1 < length\ (trace-parallel-plan-strips\ I\ \pi)$

<proof>

lemma *trace-parallel-plan-strips-head-is-initial-state:*

trace-parallel-plan-strips I $\pi ! 0 = I$

<proof>

lemma *trace-parallel-plan-strips-length-gt-one-if:*

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

shows $1 < length\ (trace-parallel-plan-strips\ I\ \pi)$

<proof>

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

$last\ (s \# trace-parallel-plan-strips\ s'\ \pi) = last\ (trace-parallel-plan-strips\ s'\ \pi)$

<proof>

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

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

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

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

assumes $k < \text{length} (\text{trace-parallel-plan-strips } I \pi) - 1$
shows $\text{are-all-operators-applicable} (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k)$
 $\wedge \text{are-all-operator-effects-consistent} (\pi ! k)$
 $\langle \text{proof} \rangle$

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-plan-prefix:*

assumes $k < \text{length} (\text{trace-parallel-plan-strips } I \pi)$
shows $\text{trace-parallel-plan-strips } I \pi ! k = \text{execute-parallel-plan } I (\text{take } k \pi)$
 $\langle \text{proof} \rangle$

lemma *length-trace-parallel-plan-strips-lte-length-plan-plus-one:*

shows $\text{length} (\text{trace-parallel-plan-strips } I \pi) \leq \text{length } \pi + 1$
 $\langle \text{proof} \rangle$

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

assumes $k < \text{length} (\text{trace-parallel-plan-strips } I \pi) - 1$
obtains $\text{ops } \pi'$ **where** $\pi = \text{ops } \# \pi'$
 $\langle \text{proof} \rangle$

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

assumes $\text{length} (\text{trace-parallel-plan-strips } I \pi) < \text{length } \pi + 1$
shows $\neg \text{are-all-operators-applicable}$
 $(\text{execute-parallel-plan } I (\text{take} (\text{length} (\text{trace-parallel-plan-strips } I \pi) - 1) \pi))$
 $(\pi ! (\text{length} (\text{trace-parallel-plan-strips } I \pi) - 1))$
 $\vee \neg \text{are-all-operator-effects-consistent} (\pi ! (\text{length} (\text{trace-parallel-plan-strips } I \pi)$
 $- 1))$
 $\langle \text{proof} \rangle$

lemma *trace-parallel-plan-step-effect-is:*

assumes $k < \text{length} (\text{trace-parallel-plan-strips } I \pi) - 1$
shows $\text{trace-parallel-plan-strips } I \pi ! \text{Suc } k$
 $= \text{execute-parallel-operator} (\text{trace-parallel-plan-strips } I \pi ! k) (\pi ! k)$
 $\langle \text{proof} \rangle$

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

fixes Π :: 'a *strips-problem*
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* $\Pi \pi$
and $k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$
shows $(\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{None} \longleftrightarrow v \notin \text{set } ((\Pi)_V)$
 $\langle \text{proof} \rangle$

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

π —and that the goal state is subsumed by the last element of the trace of π with initial state I .

lemma *execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace:*

$G \subseteq_m \text{execute-parallel-plan } I \ \pi$
 $\longleftrightarrow G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } I \ \pi)$
<proof>

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 $s :: ('variable, bool) \text{ state}$
assumes *are-all-operators-applicable* $s \ ops$
and *are-all-operator-effects-consistent* ops
and *are-all-operators-non-interfering* ops
shows $\text{execute-parallel-operator } s \ ops = \text{execute-serial-plan } s \ ops$
<proof>

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

assumes *are-all-operators-applicable* $s \ (op \ \# \ \pi)$
and *are-all-operators-non-interfering* $(op \ \# \ \pi)$
shows *are-all-operators-applicable* $(s \ \gg \ op) \ \pi$
<proof>

lemma *execute-serial-plan-split:*

fixes $s :: ('variable, bool) \text{ state}$
assumes *are-all-operators-applicable* $s \ \pi_1$
and *are-all-operators-non-interfering* π_1
shows $\text{execute-serial-plan } s \ (\pi_1 \ @ \ \pi_2)$
 $= \text{execute-serial-plan } (\text{execute-serial-plan } s \ \pi_1) \ \pi_2$
<proof>

lemma *embedding-lemma-i:*

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

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

fixes $I :: 'variable \text{ strips-state}$
assumes $\forall op \in \text{set } \pi. \text{are-operator-effects-consistent } op \ op$
and $G \subseteq_m \text{execute-serial-plan } I \ \pi$

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

lemma *embedding-lemma-iii:*

fixes Π :: 'a *strips-problem*

assumes $\forall op \in \text{set } \pi. op \in \text{set } ((\Pi)_O)$

shows $\forall ops \in \text{set } (\text{embed } \pi). \forall op \in \text{set } ops. op \in \text{set } ((\Pi)_O)$

$\langle \text{proof} \rangle$

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

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

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* Π

and *is-serial-solution-for-problem* $\Pi \pi$

shows *is-parallel-solution-for-problem* $\Pi \text{ (embed } \pi)$

$\langle \text{proof} \rangle$

lemma *flattening-lemma-i:*

fixes Π :: 'a *strips-problem*

assumes $\forall ops \in \text{set } \pi. \forall op \in \text{set } ops. op \in \text{set } ((\Pi)_O)$

shows $\forall op \in \text{set } (\text{concat } \pi). op \in \text{set } ((\Pi)_O)$

$\langle \text{proof} \rangle$

lemma *flattening-lemma-ii:*

fixes I :: 'variable *strips-state*

assumes $\forall ops \in \text{set } \pi. \exists op. ops = [op] \wedge \text{is-valid-operator-strips } \Pi op$

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

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

$\langle \text{proof} \rangle$

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 $\forall ops \in \text{set } \pi. \exists op. ops = [op]$

and *is-parallel-solution-for-problem* $\Pi \pi$

shows *is-serial-solution-for-problem* Π (*concat* π)
 ⟨*proof*⟩

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 $k < \text{length } \pi$, the flattening of the parallel plan *concat* π is a serial solution for the initial and goal state of the problem. To wit, by lemma ?? we have

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

so the second assumption entails that 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 $I :: ('variable, bool) \text{ state}$
assumes *is-valid-problem* Π
and $G \subseteq_m \text{last } (\text{trace-parallel-plan-strips } I \pi)$
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 $G \subseteq_m \text{execute-serial-plan } I (\text{concat } \pi)$
 ⟨*proof*⟩

3.4 Auxiliary lemmas about STRIPS

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

end

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

4 SAS+ Representation

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

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

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

record ('variable, 'domain) sas-plus-problem =
 variables-of :: 'variable list (<(-v+)> [1000] 999)
 operators-of :: ('variable, 'domain) sas-plus-operator list (<(-o+)> [1000] 999)
 initial-of :: ('variable, 'domain) state (<(-I+)> [1000] 999)
 goal-of :: ('variable, 'domain) state (<(-G+)> [1000] 999)
 range-of :: 'variable \rightarrow 'domain list

definition range-of':: ('variable, 'domain) sas-plus-problem \Rightarrow 'variable \Rightarrow 'domain set (<R+ - -> 52)

where

range-of' Ψ v \equiv

(case sas-plus-problem.range-of Ψ 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 (<Q>)

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 Ψ op \equiv let

pre = precondition-of op

; eff = effect-of op

; vs = variables-of Ψ

; D = range-of Ψ

in list-all ($\lambda(v, a).$ ListMem v vs) pre

$$\begin{aligned}
& \wedge \text{list-all } (\lambda(v, a). (D v \neq \text{None}) \wedge \text{ListMem } a \text{ (the } (D v))) \text{ pre} \\
& \wedge \text{list-all } (\lambda(v, a). \text{ListMem } v \text{ vs}) \text{ eff} \\
& \wedge \text{list-all } (\lambda(v, a). (D v \neq \text{None}) \wedge \text{ListMem } a \text{ (the } (D v))) \text{ eff} \\
& \wedge \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') \text{ pre}) \text{ pre} \\
& \wedge \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') \text{ eff}) \text{ eff}
\end{aligned}$$

definition *is-valid-problem-sas-plus* Ψ

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

definition *is-operator-applicable-in*

$$\begin{aligned}
& :: ('variable, 'domain) \text{ state} \\
& \Rightarrow ('variable, 'domain) \text{ sas-plus-operator} \\
& \Rightarrow \text{bool} \\
& \text{where } \text{is-operator-applicable-in } s \text{ op} \\
& \quad \equiv \text{map-of } (\text{precondition-of } \text{op}) \subseteq_m s
\end{aligned}$$

definition *execute-operator-sas-plus*

$$\begin{aligned}
& :: ('variable, 'domain) \text{ state} \\
& \Rightarrow ('variable, 'domain) \text{ sas-plus-operator} \\
& \Rightarrow ('variable, 'domain) \text{ state } (\mathbf{infixl} \langle \gg_+ \rangle 52) \\
& \text{where } \text{execute-operator-sas-plus } s \text{ op} \equiv s ++ \text{map-of } (\text{effect-of } \text{op})
\end{aligned}$$

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

lemma_[simp]:

$$\begin{aligned}
& \text{is-operator-applicable-in } s \text{ op} = (\text{map-of } (\text{precondition-of } \text{op}) \subseteq_m s) \\
& s \gg_+ \text{op} = s ++ \text{map-of } (\text{effect-of } \text{op}) \\
& \langle \text{proof} \rangle
\end{aligned}$$

lemma *range-of-not-empty*:

$$\begin{aligned}
& (\text{sas-plus-problem.range-of } \Psi v \neq \text{None} \wedge \text{sas-plus-problem.range-of } \Psi v \neq \text{Some } \square) \\
& \longleftrightarrow (\mathcal{R}_+ \Psi v) \neq \{\} \\
& \langle \text{proof} \rangle
\end{aligned}$$

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

$$\begin{aligned}
& \text{fixes } \Psi :: ('v, 'd) \text{ sas-plus-problem} \\
& \text{assumes } \text{is-valid-operator-sas-plus } \Psi \text{ op}
\end{aligned}$$

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

lemma *is-valid-problem-sas-plus-then:*

fixes $\Psi :: ('v, 'd) \text{ sas-plus-problem}$
assumes *is-valid-problem-sas-plus* Ψ
shows $\forall v \in \text{set } ((\Psi)_{\mathcal{V}+}). (\mathcal{R}_+ \Psi v) \neq \{\}$
and $\forall op \in \text{set } ((\Psi)_{\mathcal{O}+}). \text{is-valid-operator-sas-plus } \Psi \text{ } op$
and $\text{dom } ((\Psi)_{\mathcal{I}+}) = \text{set } ((\Psi)_{\mathcal{V}+})$
and $\forall v \in \text{dom } ((\Psi)_{\mathcal{I}+}). \text{the } (((\Psi)_{\mathcal{I}+}) v) \in \mathcal{R}_+ \Psi v$
and $\text{dom } ((\Psi)_{\mathcal{G}+}) \subseteq \text{set } ((\Psi)_{\mathcal{V}+})$
and $\forall v \in \text{dom } ((\Psi)_{\mathcal{G}+}). \text{the } (((\Psi)_{\mathcal{G}+}) v) \in \mathcal{R}_+ \Psi v$
 $\langle \text{proof} \rangle$

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* $s \ [] = s$
 $| \text{execute-serial-plan-sas-plus } s \ (op \ \# \ ops)$
 $= (\text{if } \text{is-operator-applicable-in } s \ op$
 $\text{then } \text{execute-serial-plan-sas-plus } (\text{execute-operator-sas-plus } s \ op) \ ops$
 $\text{else } s)$

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

definition *is-serial-solution-for-problem*

$:: ('variable, 'domain) \text{ sas-plus-problem} \Rightarrow ('variable, 'domain) \text{ sas-plus-plan} \Rightarrow$
 bool

where *is-serial-solution-for-problem* $\Psi \ \psi$
 \equiv *let*
 $I = \text{sas-plus-problem.initial-of } \Psi$
 $; G = \text{sas-plus-problem.goal-of } \Psi$
 $; ops = \text{sas-plus-problem.operators-of } \Psi$
in $G \subseteq_m \text{execute-serial-plan-sas-plus } I \ \psi$
 $\wedge \text{list-all } (\lambda op. \text{ListMem } op \ ops) \ \psi$

context
begin

private lemma *execute-operator-sas-plus-effect-i:*
assumes *is-operator-applicable-in* $s \ op$
and $\forall (v, a) \in \text{set } (\text{effect-of } op). \forall (v', a') \in \text{set } (\text{effect-of } op).$
 $v \neq v' \vee a = a'$
and $(v, a) \in \text{set } (\text{effect-of } op)$
shows $(s \ggg_+ op) \ v = \text{Some } a$
 $\langle \text{proof} \rangle$ **lemma** *execute-operator-sas-plus-effect-ii:*
assumes *is-operator-applicable-in* $s \ op$
and $\forall (v', a') \in \text{set } (\text{effect-of } op). v' \neq v$
shows $(s \ggg_+ op) \ v = s \ v$
 $\langle \text{proof} \rangle$

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 $s' \equiv s \ggg_+ op$ has the following properties:

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

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 $\forall (v, a) \in \text{set } (\text{effect-of } op).$
 $\forall (v', a') \in \text{set } (\text{effect-of } op). v \neq v' \vee a = a'$
shows $(v, a) \in \text{set } (\text{effect-of } op)$
 $\longrightarrow (s \ggg_+ op) \ v = \text{Some } a$
and $(\forall a. (v, a) \notin \text{set } (\text{effect-of } op))$
 $\longrightarrow (s \ggg_+ op) \ v = s \ v$
 $\langle \text{proof} \rangle$

end

5.2 Parallel Execution Semantics

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

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

definition *are-all-operators-applicable-in*

:: ('variable, 'domain) state
 ⇒ ('variable, 'domain) sas-plus-operator list
 ⇒ bool

where *are-all-operators-applicable-in s ops*
 ≡ list-all (is-operator-applicable-in s) ops

definition *are-operator-effects-consistent*

:: ('variable, 'domain) sas-plus-operator
 ⇒ ('variable, 'domain) sas-plus-operator
 ⇒ bool

where *are-operator-effects-consistent op op'*
 ≡ let
 effect = effect-of op
 ; effect' = effect-of op'
 in list-all (λ(v, a). list-all (λ(v', a'). v ≠ v' ∨ a = a') effect') effect

definition *are-all-operator-effects-consistent*

:: ('variable, 'domain) sas-plus-operator list
 ⇒ bool

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

definition *execute-parallel-operator-sas-plus*

:: ('variable, 'domain) state
 ⇒ ('variable, 'domain) sas-plus-operator list
 ⇒ ('variable, 'domain) state

where *execute-parallel-operator-sas-plus s ops*
 ≡ foldl (++) s (map (map-of ∘ effect-of) ops)

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

fun *execute-parallel-plan-sas-plus*

:: ('variable, 'domain) state
 ⇒ ('variable, 'domain) sas-plus-parallel-plan
 ⇒ ('variable, 'domain) state

where *execute-parallel-plan-sas-plus s [] = s*
 | *execute-parallel-plan-sas-plus s (ops # opss) = (if*
 are-all-operators-applicable-in s ops
 ∧ *are-all-operator-effects-consistent ops*
 then *execute-parallel-plan-sas-plus*
 (*execute-parallel-operator-sas-plus s ops*) *opss*
 else *s*)

fun *trace-parallel-plan-sas-plus*

$:: ('variable, 'domain) state$
 $\Rightarrow ('variable, 'domain) sas-plus-parallel-plan$
 $\Rightarrow ('variable, 'domain) state list$
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* $\Psi \psi$
 $\equiv let$
 $G = sas-plus-problem.goal-of \Psi$
 $; I = sas-plus-problem.initial-of \Psi$
 $; Ops = sas-plus-problem.operators-of \Psi$
 $in G \subseteq_m execute-parallel-plan-sas-plus I \psi$
 $\wedge list-all (\lambda ops. list-all (\lambda op. ListMem op Ops) ops) \psi$

context

begin

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

$execute-parallel-operator-sas-plus s (op \# ops)$
 $= execute-parallel-operator-sas-plus (s ++ map-of (effect-of op)) ops$
<proof>

The following lemmas show the properties of SAS+ parallel plan execution traces. The results are analogous to those for STRIPS. So, let $\tau \equiv trace-parallel-plan-sas-plus I \psi$ be a trace of a parallel SAS+ plan 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 $k < length \tau - 1$ —the parallel operator $\tau ! k$ 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:*
 $\text{trace-parallel-plan-sas-plus } I \ \psi ! 0 = I$
 ⟨proof⟩

lemma *trace-parallel-plan-sas-plus-length-gt-one-if:*
assumes $k < \text{length } (\text{trace-parallel-plan-sas-plus } I \ \psi) - 1$
shows $1 < \text{length } (\text{trace-parallel-plan-sas-plus } I \ \psi)$
 ⟨proof⟩

lemma *length-trace-parallel-plan-sas-plus-lte-length-plan-plus-one:*
shows $\text{length } (\text{trace-parallel-plan-sas-plus } I \ \psi) \leq \text{length } \psi + 1$
 ⟨proof⟩

lemma *plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements:*
assumes $k < \text{length } (\text{trace-parallel-plan-sas-plus } I \ \psi) - 1$
obtains $\text{ops } \psi'$ **where** $\psi = \text{ops } \# \ \psi'$
 ⟨proof⟩

lemma *trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds:*
assumes $k < \text{length } (\text{trace-parallel-plan-sas-plus } I \ \pi) - 1$
shows $\text{are-all-operators-applicable-in } (\text{trace-parallel-plan-sas-plus } I \ \pi ! k) (\pi ! k)$
 $\wedge \text{are-all-operator-effects-consistent } (\pi ! k)$
 ⟨proof⟩

lemma *trace-parallel-plan-sas-plus-prefix:*
assumes $k < \text{length } (\text{trace-parallel-plan-sas-plus } I \ \psi)$
shows $\text{trace-parallel-plan-sas-plus } I \ \psi ! k = \text{execute-parallel-plan-sas-plus } I \ (\text{take } k \ \psi)$
 ⟨proof⟩

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⟩

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

lemma *execute-parallel-plan-sas-plus-reaches-goal-iff-goal-is-last-element-of-trace:*
 $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \ \psi$
 $\longleftrightarrow G \subseteq_m \text{last } (\text{trace-parallel-plan-sas-plus } I \ \psi)$
 ⟨proof⟩

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

fixes $\Psi :: ('v, 'd) \text{ sas-plus-problem}$
assumes *is-parallel-solution-for-problem* $\Psi \ \psi$
shows $\forall ops \in set \ \psi. \forall op \in set \ ops. op \in set \ ((\Psi)_{\mathcal{O}+})$
<proof>

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>

lemma *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-ii*:

fixes $I :: ('variable, 'domain) \ state$
assumes $\forall op \in set \ \psi. \text{ are-operator-effects-consistent } op \ op$
and $G \subseteq_m execute-serial-plan-sas-plus \ I \ \psi$
shows $G \subseteq_m execute-parallel-plan-sas-plus \ I \ (embed \ \psi)$
<proof>

lemma *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iii*:

assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* $\Psi \ \psi$
and $op \in set \ \psi$
shows *are-operator-effects-consistent* $op \ op$
<proof>

lemma *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iv*:

fixes $\Psi :: ('v, 'd) \text{ sas-plus-problem}$
assumes $\forall op \in set \ \psi. op \in set \ ((\Psi)_{\mathcal{O}+})$

shows $\forall ops \in set (embed \ \psi). \forall op \in set \ ops. op \in set ((\Psi)_{\mathcal{O}+})$
 <proof>

theorem *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus:*

assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* $\Psi \ \psi$
shows *is-parallel-solution-for-problem* $\Psi (embed \ \psi)$
 <proof>

lemma *flattening-lemma-i:*

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

lemma *flattening-lemma-ii:*

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

lemma *flattening-lemma:*

assumes *is-valid-problem-sas-plus* Ψ
and $\forall ops \in set \ \psi. \exists op. ops = [op]$
and *is-parallel-solution-for-problem* $\Psi \ \psi$
shows *is-serial-solution-for-problem* $\Psi (concat \ \psi)$
 <proof>
 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 $(v, a) \in set (precondition-of \ op) \vee (v, a) \in set (effect-of \ op)$
shows $(\mathcal{R}_+ \ \Psi \ v) = set (the (sas-plus-problem.range-of \ \Psi \ v))$
 <proof>

lemma *set-the-range-of-is-range-of-sas-plus-if:*

fixes $\Psi :: ('v, 'd) \text{ sas-plus-problem}$
assumes *is-valid-problem-sas-plus* Ψ
 $v \in set ((\Psi)_{\mathcal{V}+})$
shows $set (the (sas-plus-problem.range-of \ \Psi \ v)) = \mathcal{R}_+ \ \Psi \ v$
 <proof>

lemma *sublocale-sas-plus-finite-domain-representation-ii:*
fixes $\Psi :: ('v, 'd)$ *sas-plus-problem*
assumes *is-valid-problem-sas-plus* Ψ
shows $\forall v \in \text{set } ((\Psi)_{\mathcal{V}_+}). (\mathcal{R}_+ \Psi v) \neq \{\}$
and $\forall op \in \text{set } ((\Psi)_{\mathcal{O}_+}). \text{is-valid-operator-sas-plus } \Psi \text{ } op$
and $\text{dom } ((\Psi)_{I_+}) = \text{set } ((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } ((\Psi)_{I_+}). \text{the } (((\Psi)_{I_+}) v) \in \mathcal{R}_+ \Psi v$
and $\text{dom } ((\Psi)_{G_+}) \subseteq \text{set } ((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } ((\Psi)_{G_+}). \text{the } (((\Psi)_{G_+}) v) \in \mathcal{R}_+ \Psi v$
<proof>

end

end

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

6 SAS+/STRIPS Equivalence

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

6.1 Translation of SAS+ Problems to STRIPS Problems

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

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

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

definition *sasp-op-to-strips*

$:: ('variable, 'domain) \text{ sas-plus-problem}$
 $\Rightarrow ('variable, 'domain) \text{ sas-plus-operator}$
 $\Rightarrow ('variable, 'domain) \text{ assignment strips-operator}$
 $(\langle \varphi_O - \rangle 99)$

where *sasp-op-to-strips* Ψ *op* \equiv *let*

pre = *precondition-of op*

; *add* = *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))]

in STRIPS-Representation.operator-for pre add delete

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

$:: ('variable, 'domain) \text{ sas-plus-problem} \Rightarrow ('variable, 'domain) \text{ assignment strips-problem}$

$(\langle \varphi - \rangle 99)$

where *sas-plus-problem-to-strips-problem* $\Psi \equiv$ *let*

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

; *ops* = *map (sasp-op-to-strips Ψ) (operators-of Ψ)*

; *I* = *state-to-strips-state Ψ (initial-of Ψ)*

; *G* = *state-to-strips-state Ψ (goal-of Ψ)*

in STRIPS-Representation.problem-for vs ops I G

definition *sas-plus-parallel-plan-to-strips-parallel-plan*

$:: ('variable, 'domain) \text{ sas-plus-problem}$

$\Rightarrow ('variable, 'domain) \text{ sas-plus-parallel-plan}$

$\Rightarrow ('variable \times 'domain) \text{ strips-parallel-plan}$

$(\langle \varphi_P - \rangle 99)$

where *sas-plus-parallel-plan-to-strips-parallel-plan* Ψ ψ

$\equiv [[\text{sasp-op-to-strips } \Psi \text{ op. op} \leftarrow \text{ops}]. \text{ops} \leftarrow \psi]$

definition *strips-state-to-state*

$:: ('variable, 'domain) \text{ sas-plus-problem}$

$\Rightarrow ('variable, 'domain) \text{ assignment strips-state}$

$\Rightarrow ('variable, 'domain) \text{ state}$

$(\langle \varphi_S^{-1} - \rangle 99)$

where *strips-state-to-state* Ψ *s*

$\equiv \text{map-of } (\text{filter } (\lambda(v, a). s(v, a) = \text{Some True}) (\text{all-possible-assignments-for}$

$\Psi))$

definition *strips-op-to-sasp*

$:: ('variable, 'domain) \text{ sas-plus-problem}$

$\Rightarrow ('variable \times 'domain) \text{ strips-operator}$

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

$(\langle \varphi_O^{-1} - \rangle 99)$

where *strips-op-to-sasp* Ψ *op*

$\equiv \text{let}$

```

precondition = strips-operator.precondition-of op
; effect = strips-operator.add-effects-of op
in (| precondition-of = precondition, effect-of = effect |)

```

definition *strips-parallel-plan-to-sas-plus-parallel-plan*
:: ('variable, 'domain) sas-plus-problem
⇒ ('variable × 'domain) strips-parallel-plan
⇒ ('variable, 'domain) sas-plus-parallel-plan
(⟨φ_P⁻¹ - -⟩ 99)
where *strips-parallel-plan-to-sas-plus-parallel-plan* Π π
≡ [[*strips-op-to-sasp* Π *op*. *op* ← *ops*]. *ops* ← π]

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

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

context
begin

— Set-up simp rules.

lemma[*simp*]:

```

(φ Ψ) = (let
  vs = [as. v ← variables-of Ψ, as ← (possible-assignments-for Ψ) v]
  ; ops = map (sasp-op-to-strips Ψ) (operators-of Ψ)
  ; I = state-to-strips-state Ψ (initial-of Ψ)
  ; G = state-to-strips-state Ψ (goal-of Ψ)
  in STRIPS-Representation.problem-for vs ops I G)
and (φS Ψ s)
= (let defined = filter (λv. s v ≠ None) (variables-of Ψ) in
  map-of (map (λ(v, a). ((v, a), the (s v) = a))
    (concat [possible-assignments-for Ψ v. v ← defined])))
and (φO Ψ op)
= (let
  pre = precondition-of op
  ; add = effect-of op
  ; delete = [(v, a'). (v, a) ← effect-of op, a' ← filter ((≠) a) (the (range-of Ψ
v))]
  in STRIPS-Representation.operator-for pre add delete)

```

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

and $(\varphi_P \Psi \psi) = [[\varphi_O \Psi op. op \leftarrow ops]. ops \leftarrow \psi]$
and $(\varphi_S^{-1} \Psi s') = \text{map-of } (\text{filter } (\lambda(v, a). s'(v, a) = \text{Some True}))$
(all-possible-assignments-for Ψ)
and $(\varphi_O^{-1} \Psi op') = (\text{let}$
 $\text{precondition} = \text{strips-operator.precondition-of } op'$
 $;$ $\text{effect} = \text{strips-operator.add-effects-of } op'$
 $\text{in } (| \text{precondition-of} = \text{precondition}, \text{effect-of} = \text{effect} |))$
and $(\varphi_P^{-1} \Psi \pi) = [[\varphi_O^{-1} \Psi op. op \leftarrow ops]. ops \leftarrow \pi]$
 $\langle \text{proof} \rangle$

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

lemma *is-valid-problem-sas-plus-dom-sas-plus-problem-range-of*:
assumes *is-valid-problem-sas-plus Ψ*
shows $\forall v \in \text{set } ((\Psi)_{\mathcal{V}_+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$
 $\langle \text{proof} \rangle$

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

lemma *all-possible-assignments-for-set-is*:
assumes $\forall v \in \text{set } ((\Psi)_{\mathcal{V}_+}). \text{range-of } \Psi v \neq \text{None}$
shows $\text{set } (\text{all-possible-assignments-for } \Psi)$
 $= (\bigcup v \in \text{set } ((\Psi)_{\mathcal{V}_+}). \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$
 $\langle \text{proof} \rangle$

lemma *state-to-strips-state-dom-is-i[simp]*:
assumes $\forall v \in \text{set } ((\Psi)_{\mathcal{V}_+}). v \in \text{dom } (\text{sas-plus-problem.range-of } \Psi)$
shows $\text{set } (\text{concat}$
 $[\text{possible-assignments-for } \Psi v. v \leftarrow \text{filter } (\lambda v. s v \neq \text{None}) (\text{variables-of } \Psi)])$
 $= (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{\mathcal{V}_+}) \wedge s v \neq \text{None} \}.$
 $\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$
 $\langle \text{proof} \rangle$

lemma *state-to-strips-state-dom-is*:
— NOTE A transformed state is defined on all possible assignments for all variables defined in the original state.
assumes *is-valid-problem-sas-plus Ψ*
shows $\text{dom } (\varphi_S \Psi s)$
 $= (\bigcup v \in \{ v \mid v. v \in \text{set } ((\Psi)_{\mathcal{V}_+}) \wedge s v \neq \text{None} \}.$
 $\{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$
 $\langle \text{proof} \rangle$

corollary *state-to-strips-state-dom-element-iff*:
assumes *is-valid-problem-sas-plus Ψ*
shows $(v, a) \in \text{dom } (\varphi_S \Psi s) \longleftrightarrow v \in \text{set } ((\Psi)_{\mathcal{V}_+})$

$\wedge s v \neq \text{None}$
 $\wedge a \in \mathcal{R}_+ \Psi v$
 <proof>

lemma *state-to-strips-state-range-is*:
assumes *is-valid-problem-sas-plus* Ψ
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>

lemma *state-to-strips-state-effect-consistent*:
assumes *is-valid-problem-sas-plus* Ψ
and $(v, a) \in \text{dom} (\varphi_S \Psi s)$
and $(v, a') \in \text{dom} (\varphi_S \Psi s)$
and $(\varphi_S \Psi s) (v, a) = \text{Some True}$
and $(\varphi_S \Psi s) (v, a') = \text{Some True}$
shows $(v, a) = (v, a')$
 <proof>

lemma *sasp-op-to-strips-set-delete-effects-is*:
assumes *is-valid-operator-sas-plus* Ψ *op*
shows $\text{set} (\text{strips-operator.delete-effects-of } (\varphi_O \Psi \text{op}))$
 $= (\bigcup (v, a) \in \text{set} (\text{effect-of } \text{op}). \{ (v, a') \mid a'. a' \in (\mathcal{R}_+ \Psi v) \wedge a' \neq a \})$
 <proof>

lemma *sas-plus-problem-to-strips-problem-variable-set-is*:
 — The variable set of Π is the set of all possible assignments that are possible using the variables of \mathcal{V} and the corresponding domains.
assumes *is-valid-problem-sas-plus* Ψ
shows $\text{set} ((\varphi \Psi)_{\mathcal{V}}) = (\bigcup v \in \text{set} ((\Psi)_{\mathcal{V}_+}). \{ (v, a) \mid a. a \in \mathcal{R}_+ \Psi v \})$
 <proof>

corollary *sas-plus-problem-to-strips-problem-variable-set-element-iff*:
assumes *is-valid-problem-sas-plus* Ψ
shows $(v, a) \in \text{set} ((\varphi \Psi)_{\mathcal{V}}) \iff v \in \text{set} ((\Psi)_{\mathcal{V}_+}) \wedge a \in \mathcal{R}_+ \Psi v$
 <proof>

lemma *sasp-op-to-strips-effect-consistent*:
assumes $\text{op} = \varphi_O \Psi \text{op}'$
and $\text{op}' \in \text{set} ((\Psi)_{\mathcal{O}_+})$
and *is-valid-operator-sas-plus* Ψ *op'*
shows $(v, a) \in \text{set} (\text{add-effects-of } \text{op}) \implies (v, a) \notin \text{set} (\text{delete-effects-of } \text{op})$
and $(v, a) \in \text{set} (\text{delete-effects-of } \text{op}) \implies (v, a) \notin \text{set} (\text{add-effects-of } \text{op})$
 <proof>

lemma *is-valid-problem-sas-plus-then-strips-transformation-too-iii*:
assumes *is-valid-problem-sas-plus* Ψ
shows *list-all (is-valid-operator-strips* $(\varphi \Psi)$
(strips-problem.operators-of $(\varphi \Psi)$)

<proof>

lemma *is-valid-problem-sas-plus-then-strips-transformation-too-iv:*

assumes *is-valid-problem-sas-plus* Ψ

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

$\longleftrightarrow \text{ListMem } x \text{ (strips-problem.variables-of } (\varphi \Psi))$

<proof> **lemma** *is-valid-problem-sas-plus-then-strips-transformation-too-v:*

assumes *is-valid-problem-sas-plus* Ψ

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

$\longrightarrow \text{ListMem } x \text{ (strips-problem.variables-of } (\varphi \Psi))$

<proof>

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>

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))

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

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

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

$= (\bigcup v \in \text{set } ((\Psi)_{V+}).$

$\{ v \mid a. a \in (\mathcal{R}_+ \Psi v) \wedge s(v, a) = \text{Some True} \})$

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

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

and $a \in \mathcal{R}_+ \Psi v$
and $(v, a) \in \text{dom } s'$
and $\forall (v, a) \in \text{dom } s'. \forall (v, a') \in \text{dom } s'. s'(v, a) = \text{Some True} \wedge s'(v, a') = \text{Some True}$
 $\rightarrow (v, a) = (v, a')$
shows $(\varphi_S^{-1} \Psi s') v = \text{Some } a \iff \text{the } (s'(v, a))$
 $\langle \text{proof} \rangle$

lemma *strips-state-to-state-inverse-is-i:*
assumes *is-valid-problem-sas-plus* Ψ
and $v \in \text{set } ((\Psi)_{\mathcal{V}_+})$
and $s v \neq \text{None}$
and $a \in \mathcal{R}_+ \Psi v$
shows $(\varphi_S \Psi s) (v, a) = \text{Some } (\text{the } (s v) = a)$
 $\langle \text{proof} \rangle$

corollary *strips-state-to-state-inverse-is-ii:*
assumes *is-valid-problem-sas-plus* Ψ
and $v \in \text{set } ((\Psi)_{\mathcal{V}_+})$
and $s v = \text{Some } a$
and $a \in \mathcal{R}_+ \Psi v$
and $a' \in \mathcal{R}_+ \Psi v$
and $a' \neq a$
shows $(\varphi_S \Psi s) (v, a') = \text{Some False}$
 $\langle \text{proof} \rangle$

corollary *strips-state-to-state-inverse-is-iii:*
assumes *is-valid-problem-sas-plus* Ψ
and $v \in \text{set } ((\Psi)_{\mathcal{V}_+})$
and $s v = \text{Some } a$
and $a \in \mathcal{R}_+ \Psi v$
and $a' \in \mathcal{R}_+ \Psi v$
and $(\varphi_S \Psi s) (v, a) = \text{Some True}$
and $(\varphi_S \Psi s) (v, a') = \text{Some True}$
shows $a = a'$
 $\langle \text{proof} \rangle$

lemma *strips-state-to-state-inverse-is-iv:*
assumes *is-valid-problem-sas-plus* Ψ
and $\text{dom } s \subseteq \text{set } ((\Psi)_{\mathcal{V}_+})$
and $v \in \text{set } ((\Psi)_{\mathcal{V}_+})$
and $s v = \text{Some } a$
and $a \in \mathcal{R}_+ \Psi v$
shows $(\varphi_S^{-1} \Psi (\varphi_S \Psi s)) v = \text{Some } a$
 $\langle \text{proof} \rangle$

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

shows $s = (\varphi_S^{-1} \Psi (\varphi_S \Psi s))$
 ⟨proof⟩
lemma *state-to-strips-state-map-le-iff*:
assumes *is-valid-problem-sas-plus* Ψ
and $\text{dom } s \subseteq \text{set } ((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } s. \text{ the } (s \ v) \in \mathcal{R}_+ \Psi \ v$
shows $s \subseteq_m t \iff (\varphi_S \Psi s) \subseteq_m (\varphi_S \Psi t)$
 ⟨proof⟩

lemma *sas-plus-operator-inverse-is*:
assumes *is-valid-problem-sas-plus* Ψ
and $op \in \text{set } ((\Psi)_{\mathcal{O}_+})$
shows $(\varphi_O^{-1} \Psi (\varphi_O \Psi op)) = op$
 ⟨proof⟩
lemma *strips-operator-inverse-is*:
assumes *is-valid-problem-sas-plus* Ψ
and $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$
shows $(\varphi_O \Psi (\varphi_O^{-1} \Psi op')) = op'$
 ⟨proof⟩

lemma *sas-plus-equivalent-to-strips-i-a-I*:
assumes *is-valid-problem-sas-plus* Ψ
and $\text{set } ops' \subseteq \text{set } ((\varphi \Psi)_{\mathcal{O}})$
and *STRIPS-Semantics.are-all-operators-applicable* $(\varphi_S \Psi s) ops'$
and $op \in \text{set } [\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$
shows *map-of (precondition-of op)* $\subseteq_m (\varphi_S^{-1} \Psi (\varphi_S \Psi s))$
 ⟨proof⟩

lemma *to-sas-plus-list-of-transformed-sas-plus-problem-operators-structure*:
assumes *is-valid-problem-sas-plus* Ψ
and $\text{set } ops' \subseteq \text{set } ((\varphi \Psi)_{\mathcal{O}})$
and $op \in \text{set } [\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$
shows $op \in \text{set } ((\Psi)_{\mathcal{O}_+}) \wedge (\exists op' \in \text{set } ops'. op' = \varphi_O \Psi op)$
 ⟨proof⟩

lemma *sas-plus-equivalent-to-strips-i-a-II*:
fixes $\Psi :: ('variable, 'domain) \text{ sas-plus-problem}$
fixes $s :: ('variable, 'domain) \text{ state}$
assumes *is-valid-problem-sas-plus* Ψ
and $\text{set } ops' \subseteq \text{set } ((\varphi \Psi)_{\mathcal{O}})$
and *STRIPS-Semantics.are-all-operators-applicable* $(\varphi_s \Psi s) ops'$
and *STRIPS-Semantics.are-all-operator-effects-consistent* ops'
shows *are-all-operator-effects-consistent* $[\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$
 ⟨proof⟩

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

assumes *is-valid-problem-sas-plus* Ψ
and *set ops'* \subseteq *set* $((\varphi \Psi)_O)$
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* $(\varphi_S^{-1} \Psi (\varphi_S \Psi s))$ $[\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$
 $ops'] \wedge$
are-all-operator-effects-consistent $[\varphi_O^{-1} \Psi op'. op' \leftarrow ops']$
<proof>

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

assumes *is-valid-problem-sas-plus* Ψ
and *dom s* \subseteq *set* $((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } s.$ *the* $(s v) \in \mathcal{R}_+ \Psi v$
and *set ops'* \subseteq *set* $((\varphi \Psi)_O)$
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-operators-applicable* $(\varphi_S \Psi s)$ *ops'*
<proof>

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

assumes *is-valid-problem-sas-plus* Ψ
and *dom s* \subseteq *set* $((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } s.$ *the* $(s v) \in \mathcal{R}_+ \Psi v$
and *set ops'* \subseteq *set* $((\varphi \Psi)_O)$
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>

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

assumes *is-valid-problem-sas-plus* Ψ
and *dom s* \subseteq *set* $((\Psi)_{\mathcal{V}_+})$
and $\forall v \in \text{dom } s.$ *the* $(s v) \in \mathcal{R}_+ \Psi v$
and *set ops'* \subseteq *set* $((\varphi \Psi)_O)$
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-operators-applicable* $(\varphi_S \Psi s)$ *ops'*
 \wedge *STRIPS-Semantics.are-all-operator-effects-consistent* *ops'*
<proof>

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

assumes *dom s* $\subseteq V$
and $\forall op \in \text{set ops.}$ $\forall (v, a) \in \text{set (effect-of op).}$ $v \in V$
shows *dom (execute-parallel-operator-sas-plus s ops)* $\subseteq V$
<proof>

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

assumes $\text{dom } s \subseteq V$

and $V \subseteq \text{dom } D$

and $\forall v \in \text{dom } s. \text{the } (s \ v) \in \text{set } (\text{the } (D \ v))$

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

shows $\forall v \in \text{dom } (\text{execute-parallel-operator-sas-plus } s \ \text{ops}).$

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

<proof>

lemma *transform-sas-plus-problem-to-strips-problem-operators-valid*:

assumes *is-valid-problem-sas-plus* Ψ

and $\text{op}' \in \text{set } ((\varphi \ \Psi)_{\mathcal{O}})$

obtains op

where $\text{op} \in \text{set } ((\Psi)_{\mathcal{O}_+})$

and $\text{op}' = (\varphi_{\mathcal{O}} \ \Psi \ \text{op})$ *is-valid-operator-sas-plus* $\Psi \ \text{op}$

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

and $\text{op}' \in \text{set } ((\varphi \ \Psi)_{\mathcal{O}})$

shows $(\varphi_S \ \Psi \ s) \ ++ \ \text{map-of } (\text{effect-to-assignments } \text{op}')$

$= \varphi_S \ \Psi \ (s \ ++ \ \text{map-of } (\text{effect-of } (\varphi_{\mathcal{O}}^{-1} \ \Psi \ \text{op}')))$

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

and $\forall \text{op}' \in \text{set ops}'. \ \text{op}' \in \text{set } ((\varphi \ \Psi)_{\mathcal{O}})$

shows *execute-parallel-operator* $(\varphi_S \ \Psi \ s) \ \text{ops}'$

$= \varphi_S \ \Psi \ (\text{execute-parallel-operator-sas-plus } s \ [\varphi_{\mathcal{O}}^{-1} \ \Psi \ \text{op}'. \ \text{op}' \leftarrow \text{ops}'])$

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

and $\forall \text{op}' \in \text{set ops}'. \ \text{op}' \in \text{set } ((\varphi \ \Psi)_{\mathcal{O}})$

and $(\varphi_S \ \Psi \ G) \subseteq_m \text{execute-parallel-plan}$

$(\text{execute-parallel-operator } (\varphi_S \ \Psi \ I) \ \text{ops}') \ \pi$

shows $(\varphi_S \ \Psi \ G) \subseteq_m \text{execute-parallel-plan}$

$(\varphi_S \ \Psi \ (\text{execute-parallel-operator-sas-plus } I \ [\varphi_{\mathcal{O}}^{-1} \ \Psi \ \text{op}'. \ \text{op}' \leftarrow \text{ops}'])) \ \pi$

<proof>

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

assumes *is-valid-problem-sas-plus* Ψ

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

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

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

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

and $\forall \text{ops}' \in \text{set } \pi. \forall \text{op}' \in \text{set ops}'. \ \text{op}' \in \text{set } ((\varphi \ \Psi)_{\mathcal{O}})$

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>

lemma *sas-plus-equivalent-to-strips-i*:
assumes *is-valid-problem-sas-plus* Ψ
and *STRIPS-Semantics.is-parallel-solution-for-problem*
 $(\varphi \Psi) \pi$
shows *goal-of* $\Psi \subseteq_m$ *execute-parallel-plan-sas-plus*
 $(\text{sas-plus-problem.initial-of } \Psi) (\varphi_P^{-1} \Psi \pi)$
 $\langle \text{proof} \rangle$

lemma *sas-plus-equivalent-to-strips-ii*:
assumes *is-valid-problem-sas-plus* Ψ
and *STRIPS-Semantics.is-parallel-solution-for-problem* $(\varphi \Psi) \pi$
shows *list-all (list-all ($\lambda op. \text{ListMem } op (\text{operators-of } \Psi))$)* $(\varphi_P^{-1} \Psi \pi)$
 $\langle \text{proof} \rangle$

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.

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

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* $\Psi (\varphi_P^{-1} \Psi \pi)$

$\langle \text{proof} \rangle$ **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 $op' \in \text{set } [\varphi_{\mathcal{O}} \Psi op. op \leftarrow ops]$
obtains *op where* $op \in \text{set ops}$
and $op' = \varphi_{\mathcal{O}} \Psi op$

$\langle \text{proof} \rangle$ **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 $op' \in \text{set } [\varphi_{\mathcal{O}} \Psi op. op \leftarrow ops]$
shows $op' \in \text{set } ((\varphi \Psi)_{\mathcal{O}})$
and *is-valid-operator-strips* $(\varphi \Psi) op'$

$\langle \text{proof} \rangle$

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

assumes *is-valid-problem-sas-plus* Ψ
and $\forall op \in \text{set ops. } op \in \text{set } ((\Psi)_{\mathcal{O}+})$
shows *execute-parallel-operator* $(\varphi_S \Psi s) [\varphi_{\mathcal{O}} \Psi op. op \leftarrow ops]$
 $= (\varphi_S \Psi (\text{execute-parallel-operator-sas-plus } s ops))$

$\langle \text{proof} \rangle$ **lemma** *strips-equivalent-to-sas-plus-i-a-IV*:

assumes *is-valid-problem-sas-plus* Ψ
and $\forall op \in \text{set ops. } op \in \text{set } ((\Psi)_{\mathcal{O}+})$
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* $[\varphi_O \Psi op. op \leftarrow ops]$
 $\langle \text{proof} \rangle$ **lemma** *strips-equivalent-to-sas-plus-i-a-V*:
assumes *is-valid-problem-sas-plus* Ψ
and $\forall op \in \text{set ops. } op \in \text{set } ((\Psi)_{\mathcal{O}+})$
and $\neg(\text{are-all-operators-applicable-in } s \text{ ops})$
 \wedge *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 *STRIPS-Semantics.are-all-operator-effects-consistent* $[\varphi_O \Psi op. op \leftarrow ops]$
 $\langle \text{proof} \rangle$

lemma *strips-equivalent-to-sas-plus-i-a*:
assumes *is-valid-problem-sas-plus* Ψ
and $\text{dom } I \subseteq \text{set } ((\Psi)_{\mathcal{V}+})$
and $\forall v \in \text{dom } I. \text{ the } (I v) \in \mathcal{R}_+ \Psi v$
and $\text{dom } G \subseteq \text{set } ((\Psi)_{\mathcal{V}+})$
and $\forall v \in \text{dom } G. \text{ the } (G v) \in \mathcal{R}_+ \Psi v$
and $\forall ops \in \text{set } \psi. \forall op \in \text{set ops. } op \in \text{set } ((\Psi)_{\mathcal{O}+})$
and $G \subseteq_m \text{execute-parallel-plan-sas-plus } I \psi$
shows $(\varphi_S \Psi G) \subseteq_m \text{execute-parallel-plan } (\varphi_S \Psi I) (\varphi_P \Psi \psi)$
 $\langle \text{proof} \rangle$

lemma *strips-equivalent-to-sas-plus-i*:
assumes *is-valid-problem-sas-plus* Ψ
and *is-parallel-solution-for-problem* $\Psi \psi$
shows $(\text{strips-problem.goal-of } (\varphi \Psi)) \subseteq_m \text{execute-parallel-plan}$
 $(\text{strips-problem.initial-of } (\varphi \Psi)) (\varphi_P \Psi \psi)$
 $\langle \text{proof} \rangle$

lemma *strips-equivalent-to-sas-plus-ii*:
assumes *is-valid-problem-sas-plus* Ψ
and *is-parallel-solution-for-problem* $\Psi \psi$
shows *list-all* $(\text{list-all } (\lambda op. \text{ListMem } op (\text{strips-problem.operators-of } (\varphi \Psi))))$
 $(\varphi_P \Psi \psi)$
 $\langle \text{proof} \rangle$

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* Ψ

and *is-parallel-solution-for-problem* $\Psi \psi$

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

<proof>

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

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

obtains op

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

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

<proof> **lemma** *serial-sas-plus-equivalent-to-serial-strips-i:*

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

obtains op **where** $op \in \text{set } \psi$ **and** $ops = [\varphi_O \Psi \text{ op}]$

<proof> **lemma** *serial-sas-plus-equivalent-to-serial-strips-ii[simp]:*

$\text{concat } (\varphi_P \Psi (\text{embed } \psi)) = [\varphi_O \Psi \text{ op. } op \leftarrow \psi]$

<proof>

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, $\pi \equiv \varphi_P \Psi \text{ List-Supplement.embed } \psi$ 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

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

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* $(\varphi \Psi) [\varphi_O \Psi \text{ op. op} \leftarrow \psi]$
 $\langle \text{proof} \rangle$

lemma *embedded-serial-strips-plan-operator-structure:*
assumes $\text{ops}' \in \text{set} (\text{embed } \pi)$
obtains op
where $\text{op} \in \text{set } \pi$ **and** $[\varphi_O^{-1} \Pi \text{ op. op} \leftarrow \text{ops}'] = [\varphi_O^{-1} \Pi \text{ op}]$
 $\langle \text{proof} \rangle$ **lemma** *serial-strips-equivalent-to-serial-sas-plus-i:*
assumes $\text{ops} \in \text{set} (\varphi_P^{-1} \Pi (\text{embed } \pi))$
obtains op **where** $\text{op} \in \text{set } \pi$ **and** $\text{ops} = [\varphi_O^{-1} \Pi \text{ op}]$
 $\langle \text{proof} \rangle$ **lemma** *serial-strips-equivalent-to-serial-sas-plus-ii[simp]:*
 $\text{concat} (\varphi_P^{-1} \Pi (\text{embed } \pi)) = [\varphi_O^{-1} \Pi \text{ op. op} \leftarrow \pi]$
 $\langle \text{proof} \rangle$

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* $(\varphi \Psi) \pi$
shows *SAS-Plus-Semantics.is-serial-solution-for-problem* $\Psi [\varphi_O^{-1} \Psi \text{ op. op} \leftarrow \pi]$
 $\langle \text{proof} \rangle$

6.2 Equivalence of SAS+ and STRIPS

abbreviation *bounded-plan-set*
where *bounded-plan-set ops k* $\equiv \{ \pi. \text{set } \pi \subseteq \text{set ops} \wedge \text{length } \pi = k \}$

definition *bounded-solution-set-sas-plus'*
 $:: ('variable, 'domain) \text{ sas-plus-problem}$
 $\Rightarrow \text{nat}$
 $\Rightarrow ('variable, 'domain) \text{ sas-plus-plan set}$
where *bounded-solution-set-sas-plus' Ψk*
 $\equiv \{ \psi. \text{is-serial-solution-for-problem } \Psi \psi \wedge \text{length } \psi = k \}$

abbreviation *bounded-solution-set-sas-plus*
 $:: ('variable, 'domain) \text{ sas-plus-problem}$
 $\Rightarrow \text{nat}$
 $\Rightarrow ('variable, 'domain) \text{ sas-plus-plan set}$
where *bounded-solution-set-sas-plus ΨN*
 $\equiv (\bigcup k \in \{0..N\}. \text{bounded-solution-set-sas-plus}' \Psi k)$

definition *bounded-solution-set-strips'*

$:: ('variable \times 'domain) \text{ strips-problem}$

$\Rightarrow nat$

$\Rightarrow ('variable \times 'domain) \text{ strips-plan set}$

where *bounded-solution-set-strips'* Πk

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

abbreviation *bounded-solution-set-strips*

$:: ('variable \times 'domain) \text{ strips-problem}$

$\Rightarrow nat$

$\Rightarrow ('variable \times 'domain) \text{ strips-plan set}$

where *bounded-solution-set-strips* $\Pi N \equiv (\bigcup k \in \{0..N\}. \text{bounded-solution-set-strips}' \Pi k)$

— Show that plan transformation for all SAS Plus solutions yields a STRIPS solution for the induced STRIPS problem with same length.

We first show injectiveness of plan transformation $\lambda\psi. [\varphi_O \Psi \text{ op. } \text{op} \leftarrow \psi]$ on the set of plans $P_k \equiv \text{bounded-plan-set (operators-of } \Psi) k$ 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 $(\varphi_O \Psi \text{ op}_1) = (\varphi_O \Psi \text{ op}_2)$

shows $\text{op}_1 = \text{op}_2$

<proof>

lemma *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a*:

assumes *is-valid-problem-sas-plus* Ψ

shows *inj-on* $(\lambda\psi. [\varphi_O \Psi \text{ op. } \text{op} \leftarrow \psi]) (\text{bounded-plan-set (sas-plus-problem.operators-of } \Psi) k)$

<proof> **corollary** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-b*:

assumes *is-valid-problem-sas-plus* Ψ

shows *inj-on* $(\lambda\psi. [\varphi_O \Psi \text{ op. } \text{op} \leftarrow \psi]) (\text{bounded-solution-set-sas-plus}' \Psi k)$

<proof> **lemma** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c*:

assumes *is-valid-problem-sas-plus* Ψ

shows $(\lambda\psi. [\varphi_O \Psi \text{ op. } \text{op} \leftarrow \psi]) ' (\text{bounded-solution-set-sas-plus}' \Psi k)$

$= \text{bounded-solution-set-strips}' (\varphi \Psi) k$

<proof> **lemma** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-d*:

assumes *is-valid-problem-sas-plus* Ψ

shows $\text{card} (\text{bounded-solution-set-sas-plus}' \Psi k) \leq \text{card} (\text{bounded-solution-set-strips}' (\varphi \Psi) k)$

<proof>

lemma *bounded-plan-set-finite*:

shows *finite* $\{ \pi. \text{set } \pi \subseteq \text{set ops} \wedge \text{length } \pi = k \}$

<proof> **lemma** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a*:

assumes *is-valid-problem-sas-plus* Ψ

shows *finite* $(\text{bounded-solution-set-sas-plus}' \Psi k)$

<proof> **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>

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

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

theorem

assumes *is-valid-problem-sas-plus Ψ*
shows *card (bounded-solution-set-sas-plus ΨN)*
= card (bounded-solution-set-strips ($\varphi \Psi$) N)
<proof>

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* \Rightarrow *nat* \Rightarrow *bool option* \Rightarrow *sat-plan-variable formula*
where *encode-state-variable* $t \ k \ v \equiv$ *case* v of
 Some True \Rightarrow *Atom* (*State* $t \ k$)
 | *Some False* \Rightarrow \neg (*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 \Rightarrow sat-plan-variable formula ($\langle \Phi_I \rightarrow 99 \rangle$)
where *encode-initial-state* Π
 \equiv let $I =$ initial-of Π
; $vs =$ variables-of Π
in \bigwedge (map (λv . encode-state-variable 0 (index vs v) ($I v$) $\vee \perp$)
(filter (λv . $I v \neq$ None) vs))

definition *encode-goal-state*

$::$ 'variable strips-problem \Rightarrow nat \Rightarrow sat-plan-variable formula ($\langle \Phi_G \rightarrow 99 \rangle$)
where *encode-goal-state* Πt
 \equiv let
 $vs =$ variables-of Π
; $G =$ goal-of Π
in \bigwedge (map (λv . encode-state-variable t (index vs v) ($G v$) $\vee \perp$)
(filter (λv . $G v \neq$ None) vs))

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

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

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

definition *encode-operator-precondition*

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

definition *encode-all-operator-preconditions*

$::$ 'variable strips-problem
 \Rightarrow 'variable strips-operator list
 \Rightarrow nat
 \Rightarrow sat-plan-variable formula
where *encode-all-operator-preconditions* $\Pi ops t \equiv$ let
 $l = List.product [0..<t] ops$
in foldr (\bigwedge) (map ($\lambda(t, op)$. *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.

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

for add effects respectively

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

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

definition *encode-operator-effect*

```

:: 'variable strips-problem
  => nat
  => 'variable strips-operator
  => sat-plan-variable formula
where encode-operator-effect  $\Pi$  t op
   $\equiv$  let
    vs = variables-of  $\Pi$ 
    ; ops = operators-of  $\Pi$ 
  in  $\bigwedge$ (map ( $\lambda v$ .
     $\neg$ (Atom (Operator t (index ops op)))
     $\vee$  Atom (State (Suc t) (index vs v)))
    (add-effects-of op)
    @ map ( $\lambda v$ .
     $\neg$ (Atom (Operator t (index ops op)))
     $\vee$   $\neg$  (Atom (State (Suc t) (index vs v))))
    (delete-effects-of op))

```

definition *encode-all-operator-effects*

```

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

```

definition *encode-operators*

```

:: 'variable strips-problem => nat => sat-plan-variable formula
where encode-operators  $\Pi$  t
   $\equiv$  let ops = operators-of  $\Pi$ 
  in encode-all-operator-preconditions  $\Pi$  ops t  $\wedge$  encode-all-operator-effects  $\Pi$ 
  ops t

```

Definitions ?? and ?? similarly encode the negative resp. positive transition frame axioms as disjunctions.

definition *encode-negative-transition-frame-axiom*

$::$ *'variable strips-problem*
 \Rightarrow *nat*
 \Rightarrow *'variable*
 \Rightarrow *sat-plan-variable formula*
where *encode-negative-transition-frame-axiom* Π t v
 \equiv *let* $vs = \text{variables-of } \Pi$
 $\quad ; ops = \text{operators-of } \Pi$
 $\quad ; \text{deleting-operators} = \text{filter } (\lambda op. \text{ListMem } v (\text{delete-effects-of } op)) ops$
 $\text{in } \neg(\text{Atom } (\text{State } t (\text{index } vs v)))$
 $\quad \vee (\text{Atom } (\text{State } (\text{Suc } t) (\text{index } vs v)))$
 $\quad \vee \bigvee (\text{map } (\lambda op. \text{Atom } (\text{Operator } t (\text{index } ops op))) \text{deleting-operators}))$

definition *encode-positive-transition-frame-axiom*
 $::$ *'variable strips-problem*
 \Rightarrow *nat*
 \Rightarrow *'variable*
 \Rightarrow *sat-plan-variable formula*
where *encode-positive-transition-frame-axiom* Π t v
 \equiv *let* $vs = \text{variables-of } \Pi$
 $\quad ; ops = \text{operators-of } \Pi$
 $\quad ; \text{adding-operators} = \text{filter } (\lambda op. \text{ListMem } v (\text{add-effects-of } op)) ops$
 $\text{in } (\text{Atom } (\text{State } t (\text{index } vs v)))$
 $\quad \vee (\neg(\text{Atom } (\text{State } (\text{Suc } t) (\text{index } vs v))))$
 $\quad \vee \bigvee (\text{map } (\lambda op. \text{Atom } (\text{Operator } t (\text{index } ops op))) \text{adding-operators}))$

definition *encode-all-frame-axioms*
 $::$ *'variable strips-problem* \Rightarrow *nat* \Rightarrow *sat-plan-variable formula*
where *encode-all-frame-axioms* Π t
 \equiv *let* $l = \text{List.product } [0..<t]$ (*variables-of* Π)
 $\text{in } \bigwedge (\text{map } (\lambda(k, v). \text{encode-negative-transition-frame-axiom } \Pi k v) l$
 $\quad @ \text{map } (\lambda(k, v). \text{encode-positive-transition-frame-axiom } \Pi k v) l)$

Finally, the basic SATPlan encoding is the conjunction of the initial state, goal state, operator and frame axiom encoding for all time steps. The functions and ⁶ take care of mapping the operator precondition, effect and frame axiom encoding over all possible combinations of time point and operators resp. time points, variables, and operators.

definition *encode-problem* ($\langle \Phi \rightarrow 99$)
where *encode-problem* Π t
 \equiv *encode-initial-state* Π
 $\quad \wedge (\text{encode-operators } \Pi t)$
 $\quad \wedge (\text{encode-all-frame-axioms } \Pi t)$
 $\quad \wedge (\text{encode-goal-state } \Pi t))$

⁶Not shown.

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
 \Rightarrow sat-plan-variable valuation
 \Rightarrow nat
 \Rightarrow 'variable strips-operator list
where *decode-plan'* Π \mathcal{A} i
 \equiv let ops = operators-of Π
; vs = map ($\lambda op.$ Operator i (index ops op)) (remdups ops)
in map ($\lambda v.$ case v of Operator - $k \Rightarrow ops ! k$) (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
 \Rightarrow sat-plan-variable valuation
 \Rightarrow nat
 \Rightarrow 'variable strips-parallel-plan ($\langle \Phi^{-1} - - \rightarrow 99$)
where *decode-plan* Π \mathcal{A} $t \equiv$ map (*decode-plan'* Π \mathcal{A}) $[0..<t]$

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

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

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

definition *encode-transitions* $::$ 'variable strips-problem \Rightarrow nat \Rightarrow sat-plan-variable formula ($\langle \Phi_T - - \rightarrow 99$) **where**

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

$encode-transitions \Pi t$
 $\equiv SAT-Plan-Base.encode-operators \Pi t \wedge$
 $SAT-Plan-Base.encode-all-frame-axioms \Pi t$

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

— Setup simp rules.

lemma $[simp]$:
 $encode-transitions \Pi t$
 $= SAT-Plan-Base.encode-operators \Pi t \wedge$
 $SAT-Plan-Base.encode-all-frame-axioms \Pi t$
 $\langle proof \rangle$

context
begin

lemma $encode-state-variable-is-lit-plus-if$:
assumes $is-valid-problem-strips \Pi$
and $v \in dom s$
shows $is-lit-plus (encode-state-variable k (index (strips-problem.variables-of \Pi) v) (s v))$
 $\langle proof \rangle$

lemma $is-cnf-encode-initial-state$:
assumes $is-valid-problem-strips \Pi$
shows $is-cnf (\Phi_I \Pi)$
 $\langle proof \rangle$

lemma $encode-goal-state-is-cnf$:
assumes $is-valid-problem-strips \Pi$
shows $is-cnf (encode-goal-state \Pi t)$
 $\langle proof \rangle$ **lemma** $encode-operator-precondition-is-cnf$:
 $is-cnf (encode-operator-precondition \Pi k op)$
 $\langle proof \rangle$ **lemma** $set-map-operator-precondition[simp]$:
 $set (map (\lambda(k, op). encode-operator-precondition \Pi k op) (List.product [0..<t] ops))$
 $= \{ encode-operator-precondition \Pi k op \mid k op. (k, op) \in (\{0..<t\} \times set ops) \}$
 $\langle proof \rangle$ **lemma** $is-cnf-encode-all-operator-preconditions$:
 $is-cnf (encode-all-operator-preconditions \Pi (strips-problem.operators-of \Pi) t)$
 $\langle proof \rangle$ **lemma** $set-map-or[simp]$:
 $set (map (\lambda v. A v \vee B v) vs) = \{ A v \vee B v \mid v. v \in set vs \}$
 $\langle proof \rangle$ **lemma** $encode-operator-effects-is-cnf-i$:
 $is-cnf (\wedge (map (\lambda v. (\neg (Atom (Operator t (index (strips-problem.operators-of \Pi) op))))$
 $\vee Atom (State (Suc t) (index (strips-problem.variables-of \Pi) v))) (add-effects-of op)))$
 $\langle proof \rangle$ **lemma** $encode-operator-effects-is-cnf-ii$:
 $is-cnf (\wedge (map (\lambda v. \neg (Atom (Operator t (index (strips-problem.operators-of \Pi)$

$op)))$
 $\vee \neg(\text{Atom}(\text{State}(\text{Suc } t) (\text{index}(\text{strips-problem.variables-of } \Pi) v)))) (\text{delete-effects-of } op)))$
 $\langle \text{proof} \rangle$ **lemma** *encode-operator-effect-is-cnf*:
shows *is-cnf* (*encode-operator-effect* Π t op)
 $\langle \text{proof} \rangle$ **lemma** *set-map-encode-operator-effect[simp]*:
 $\text{set}(\text{map}(\lambda(t, op). \text{encode-operator-effect } \Pi t op) (\text{List.product } [0..<t] (\text{strips-problem.operators-of } \Pi))))$
 $= \{ \text{encode-operator-effect } \Pi k op \mid k op. (k, op) \in (\{0..<t\} \times \text{set}(\text{strips-problem.operators-of } \Pi)) \}$
 $\langle \text{proof} \rangle$ **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)
 $\langle \text{proof} \rangle$

lemma *encode-operators-is-cnf*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (*encode-operators* Π t)
 $\langle \text{proof} \rangle$ **lemma** *set-map-to-operator-atom[simp]*:
 $\text{set}(\text{map}(\lambda op. \text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op))) (\text{filter}(\lambda op. \text{ListMem } v vs) (\text{strips-problem.operators-of } \Pi))))$
 $= \{ \text{Atom}(\text{Operator } t (\text{index}(\text{strips-problem.operators-of } \Pi) op)) \mid op. op \in \text{set}(\text{strips-problem.operators-of } \Pi) \wedge v \in \text{set } vs \}$
 $\langle \text{proof} \rangle$

lemma *is-disj-big-or-if*:
assumes $\forall f \in \text{set } fs. \text{is-lit-plus } f$
shows *is-disj* $\bigvee fs$
 $\langle \text{proof} \rangle$

lemma *is-cnf-encode-negative-transition-frame-axiom*:
shows *is-cnf* (*encode-negative-transition-frame-axiom* Π t v)
 $\langle \text{proof} \rangle$

lemma *is-cnf-encode-positive-transition-frame-axiom*:
shows *is-cnf* (*encode-positive-transition-frame-axiom* Π t v)
 $\langle \text{proof} \rangle$ **lemma** *encode-all-frame-axioms-set[simp]*:
 $\text{set}(\text{map}(\lambda(k, v). \text{encode-negative-transition-frame-axiom } \Pi k v) (\text{List.product } [0..<t] (\text{strips-problem.variables-of } \Pi)))$
 $\quad @ (\text{map}(\lambda(k, v). \text{encode-positive-transition-frame-axiom } \Pi k v) (\text{List.product } [0..<t] (\text{strips-problem.variables-of } \Pi))))$
 $= \{ \text{encode-negative-transition-frame-axiom } \Pi k v \mid k v. (k, v) \in (\{0..<t\} \times \text{set}(\text{strips-problem.variables-of } \Pi)) \}$
 $\cup \{ \text{encode-positive-transition-frame-axiom } \Pi k v \mid k v. (k, v) \in (\{0..<t\} \times \text{set}(\text{strips-problem.variables-of } \Pi)) \}$
 $\langle \text{proof} \rangle$

lemma *encode-frame-axioms-is-cnf*:
shows *is-cnf* (*encode-all-frame-axioms* Π t)
 \langle *proof* \rangle

lemma *is-cnf-encode-problem*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (Φ Π t)
 \langle *proof* \rangle

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
 \langle *proof* \rangle

lemma *cnf-of-encode-problem-structure*:
shows *cnf* (*SAT-Plan-Base.encode-initial-state* Π)
 \subseteq *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-goal-state* Π t)
 \subseteq *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-operators* Π t)
 \subseteq *cnf* (*SAT-Plan-Base.encode-problem* Π t)
and *cnf* (*SAT-Plan-Base.encode-all-frame-axioms* Π t)
 \subseteq *cnf* (*SAT-Plan-Base.encode-problem* Π t)
 \langle *proof* \rangle **lemma** *cnf-of-encode-initial-state-set-i*:
shows *cnf* (Φ_I Π) = $\bigcup \{ \text{cnf} (\text{encode-state-variable } 0$
 $(\text{index} (\text{strips-problem.variables-of } \Pi) v) ((\Pi)_I v))$
 $\mid v. v \in \text{set} (\text{strips-problem.variables-of } \Pi) \wedge ((\Pi)_I v) \neq \text{None} \}$
 \langle *proof* \rangle

corollary *cnf-of-encode-initial-state-set-ii*:
assumes *is-valid-problem-strips* Π
shows *cnf* (Φ_I Π) = $(\bigcup v \in \text{set} (\text{strips-problem.variables-of } \Pi). \{ \{$
 $\text{literal-formula-to-literal} (\text{encode-state-variable } 0 (\text{index} (\text{strips-problem.variables-of}$
 $\Pi) v)$
 $(\text{strips-problem.initial-of } \Pi v) \} \})$
 \langle *proof* \rangle

lemma *cnf-of-encode-initial-state-set*:
assumes *is-valid-problem-strips* Π
and $v \in \text{dom} (\text{strips-problem.initial-of } \Pi)$
shows *strips-problem.initial-of* Π $v = \text{Some True} \longrightarrow (\exists! C. C \in \text{cnf} (\Phi_I \Pi)$
 $\wedge C = \{ (\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v))^+ \}$)
and *strips-problem.initial-of* Π $v = \text{Some False} \longrightarrow (\exists! C. C \in \text{cnf} (\Phi_I \Pi)$
 $\wedge C = \{ (\text{State } 0 (\text{index} (\text{strips-problem.variables-of } \Pi) v))^{-1} \}$)
 \langle *proof* \rangle

lemma *cnf-of-operator-encoding-structure*:

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

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

$$\begin{aligned} \text{cnf } (\text{encode-all-operator-preconditions } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t) \\ \subseteq \text{cnf } (\Phi \ \Pi \ t) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *cnf-foldr-and[simp]*:

$$\begin{aligned} \text{cnf } (\text{foldr } (\wedge) \ fs \ (\neg\perp)) &= (\bigcup f \in \text{set } fs. \ \text{cnf } f) \\ \langle \text{proof} \rangle \text{ lemma } \text{cnf-of-encode-operator-precondition[simp]}: \\ \text{cnf } (\text{encode-operator-precondition } \Pi \ t \ op) &= (\bigcup v \in \text{set } (\text{precondition-of } op). \\ &\quad \{ \{ (\text{Operator } t \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))^{-1} \\ &\quad \quad , \ (\text{State } t \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+ \} \} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *cnf-of-encode-all-operator-preconditions-structure[simp]*:

$$\begin{aligned} \text{cnf } (\text{encode-all-operator-preconditions } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t) \\ = (\bigcup (t, \ op) \in (\{..<t\} \times \text{set } (\text{operators-of } \Pi))). \\ (\bigcup v \in \text{set } (\text{precondition-of } op). \\ \{ \{ (\text{Operator } t \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))^{-1} \\ \quad , \ (\text{State } t \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+ \} \} \\ \langle \text{proof} \rangle \end{aligned}$$

corollary *cnf-of-encode-all-operator-preconditions-contains-clause-if*:

$$\begin{aligned} \text{fixes } \Pi::\text{'variable STRIPS-Representation.strips-problem} \\ \text{assumes } \text{is-valid-problem-strips } (\Pi::\text{'variable STRIPS-Representation.strips-problem}) \\ \text{and } k < t \\ \text{and } op \in \text{set } ((\Pi)_{\mathcal{O}}) \\ \text{and } v \in \text{set } (\text{precondition-of } op) \\ \text{shows } \{ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))^{-1} \\ \quad , \ (\text{State } k \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+ \} \\ \in \text{cnf } (\text{encode-all-operator-preconditions } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t) \\ \langle \text{proof} \rangle \end{aligned}$$

corollary *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*:

$$\begin{aligned} \text{cnf } (\text{encode-all-operator-effects } \Pi \ (\text{strips-problem.operators-of } \Pi) \ t) \\ \subseteq \text{cnf } (\Phi \ \Pi \ t) \\ \langle \text{proof} \rangle \text{ lemma } \text{cnf-of-encode-operator-effect-structure[simp]}: \\ \text{cnf } (\text{encode-operator-effect } \Pi \ t \ op) \\ = (\bigcup v \in \text{set } (\text{add-effects-of } op). \{ \{ (\text{Operator } t \ (\text{index } (\text{strips-problem.operators-of} \\ \Pi) \ op))^{-1} \\ \quad , \ (\text{State } (\text{Suc } t) \ (\text{index } (\text{strips-problem.variables-of } \Pi) \ v))^+ \} \} \end{aligned}$$

$\cup (\bigcup v \in \text{set } (\text{delete-effects-of } op).$
 $\{\{ (\text{Operator } t (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$
 $, (\text{State } (\text{Suc } t) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \}\})$
 <proof>

lemma *cnf-of-encode-all-operator-effects-structure:*

$\text{cnf } (\text{encode-all-operator-effects } \Pi (\text{strips-problem.operators-of } \Pi) t)$
 $= (\bigcup (k, op) \in (\{0..<t\} \times \text{set } ((\Pi)_O)).$
 $(\bigcup v \in \text{set } (\text{add-effects-of } op).$
 $\{\{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \}\}))$
 $\cup (\bigcup (k, op) \in (\{0..<t\} \times \text{set } ((\Pi)_O)).$
 $(\bigcup v \in \text{set } (\text{delete-effects-of } op).$
 $\{\{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \}\}))$
 <proof>

corollary *cnf-of-operator-effect-encoding-contains-add-effect-clause-if:*

fixes $\Pi::$ 'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $k < t$
and $op \in \text{set } ((\Pi)_O)$
and $v \in \text{set } (\text{add-effects-of } op)$
shows $\{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \}$
 $\in \text{cnf } (\text{encode-all-operator-effects } \Pi (\text{strips-problem.operators-of } \Pi) t)$
 <proof>

corollary *cnf-of-operator-effect-encoding-contains-delete-effect-clause-if:*

fixes $\Pi::$ 'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $k < t$
and $op \in \text{set } ((\Pi)_O)$
and $v \in \text{set } (\text{delete-effects-of } op)$
shows $\{ (\text{Operator } k (\text{index } (\text{strips-problem.operators-of } \Pi) op))^{-1}$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \}$
 $\in \text{cnf } (\text{encode-all-operator-effects } \Pi (\text{strips-problem.operators-of } \Pi) t)$
 <proof>

lemma *cnf-of-big-or-of-literal-formulas-is[simp]:*

assumes $\forall f \in \text{set } fs. \text{is-literal-formula } f$
shows $\text{cnf } (\bigvee fs) = \{\{ \text{literal-formula-to-literal } f \mid f. f \in \text{set } fs \}\}$

<proof> **lemma** *set-filter-op-list-mem-vs[simp]:*

$\text{set } (\text{filter } (\lambda op. \text{ListMem } v \text{ us}) ops) = \{ op. op \in \text{set } ops \wedge v \in \text{set } us \}$

<proof> **lemma** *cnf-of-positive-transition-frame-axiom:*

$\text{cnf } (\text{encode-positive-transition-frame-axiom } \Pi k v)$

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

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

$\mid op. op \in \text{set } (\text{strips-problem.operators-of } \Pi) \wedge v \in \text{set } (\text{add-effects-of } op)$

}}

⟨proof⟩ **lemma** *cnf-of-negative-transition-frame-axiom*:
 $cnf\ (encode-negative-transition-frame-axiom\ \Pi\ k\ v)$
 $= \{ \{ (State\ k\ (index\ (strips-problem.variables-of\ \Pi)\ v))^{-1}$
 $\quad ,\ (State\ (Suc\ k)\ (index\ (strips-problem.variables-of\ \Pi)\ v))^+ \}$
 $\cup \{ (Operator\ k\ (index\ (strips-problem.operators-of\ \Pi)\ op))^+$
 $\quad | op.\ op \in set\ (strips-problem.operators-of\ \Pi) \wedge v \in set\ (delete-effects-of\ op)$
 $\} \}$
 ⟨proof⟩

lemma *cnf-of-encode-all-frame-axioms-structure*:
 $cnf\ (encode-all-frame-axioms\ \Pi\ t)$
 $= \bigcup (\bigcup (k, v) \in (\{0..<t\} \times set\ ((\Pi)_V)).$
 $\quad \{ \{ (State\ k\ (index\ (strips-problem.variables-of\ \Pi)\ v))^+$
 $\quad ,\ (State\ (Suc\ k)\ (index\ (strips-problem.variables-of\ \Pi)\ v))^{-1} \}$
 $\quad \cup \{ (Operator\ k\ (index\ (strips-problem.operators-of\ \Pi)\ op))^+$
 $\quad \quad | op.\ op \in set\ ((\Pi)_O) \wedge v \in set\ (add-effects-of\ op) \} \}$
 $\cup \bigcup (\bigcup (k, v) \in (\{0..<t\} \times set\ ((\Pi)_V)).$
 $\quad \{ \{ (State\ k\ (index\ (strips-problem.variables-of\ \Pi)\ v))^{-1}$
 $\quad ,\ (State\ (Suc\ k)\ (index\ (strips-problem.variables-of\ \Pi)\ v))^+ \}$
 $\quad \cup \{ (Operator\ k\ (index\ (strips-problem.operators-of\ \Pi)\ op))^+$
 $\quad \quad | op.\ op \in set\ ((\Pi)_O) \wedge v \in set\ (delete-effects-of\ op) \} \}$

⟨proof⟩ **lemma** *cnf-of-encode-goal-state-set-i*:
 $cnf\ ((\Phi_G\ \Pi)\ t) = \bigcup (\{ cnf\ (encode-state-variable\ t$
 $\quad (index\ (strips-problem.variables-of\ \Pi)\ v)\ ((\Pi)_G)\ v)$
 $\quad | v.\ v \in set\ ((\Pi)_V) \wedge ((\Pi)_G)\ v \neq None \}$
 ⟨proof⟩

corollary *cnf-of-encode-goal-state-set-ii*:
assumes *is-valid-problem-strips* Π
shows $cnf\ ((\Phi_G\ \Pi)\ t) = \bigcup (\{ \{ \{ literal-formula-to-literal$
 $\quad (encode-state-variable\ t\ (index\ (strips-problem.variables-of\ \Pi)\ v)\ ((\Pi)_G)\ v)$
 $\} \}$
 $\quad | v.\ v \in set\ ((\Pi)_V) \wedge ((\Pi)_G)\ v \neq None \}$
 ⟨proof⟩

lemma *cnf-of-encode-goal-state-set*:
fixes Π :: 'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $v \in dom\ ((\Pi)_G)$
shows $((\Pi)_G)\ v = Some\ True \longrightarrow (\exists! C.\ C \in cnf\ ((\Phi_G\ \Pi)\ t)$
 $\quad \wedge C = \{ (State\ t\ (index\ (strips-problem.variables-of\ \Pi)\ v))^+ \}$
and $((\Pi)_G)\ v = Some\ False \longrightarrow (\exists! C.\ C \in cnf\ ((\Phi_G\ \Pi)\ t)$
 $\quad \wedge C = \{ (State\ t\ (index\ (strips-problem.variables-of\ \Pi)\ v))^{-1} \}$
 ⟨proof⟩

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 $\mathcal{A} \models \Phi \Pi t = cnf\text{-}semantics \mathcal{A} (cnf(\Phi \Pi t))$

<proof>

corollary *valuation-models-encoding-cnf-formula-equals-corollary*:

assumes *is-valid-problem-strips* Π

shows $\mathcal{A} \models (\Phi \Pi t)$

$= (\forall C \in cnf(\Phi \Pi t). \exists L \in C. lit\text{-}semantics \mathcal{A} L)$

<proof>

lemma *decode-plan-length*:

assumes $\pi = \Phi^{-1} \Pi \nu t$

shows $length \pi = t$

<proof>

lemma *decode-plan'-set-is[simp]*:

set (decode-plan' $\Pi \mathcal{A} k$)

$= \{ (strips\text{-}problem.operators\text{-}of \Pi) ! (index (strips\text{-}problem.operators\text{-}of \Pi) op)$

$| op. op \in set (strips\text{-}problem.operators\text{-}of \Pi)$

$\wedge \mathcal{A} (Operator k (index (strips\text{-}problem.operators\text{-}of \Pi) op)) \}$

<proof>

lemma *decode-plan-set-is[simp]*:

set ($\Phi^{-1} \Pi \mathcal{A} t$) = ($\bigcup k \in \{..<t\}. \{ decode\text{-}plan' \Pi \mathcal{A} k \}$)

<proof>

lemma *decode-plan-step-element-then-i*:

assumes $k < t$

shows *set ($(\Phi^{-1} \Pi \mathcal{A} t) ! k$)*

$= \{ (strips\text{-}problem.operators\text{-}of \Pi) ! (index (strips\text{-}problem.operators\text{-}of \Pi) op)$

$| op. op \in set ((\Pi)_O) \wedge \mathcal{A} (Operator k (index (strips\text{-}problem.operators\text{-}of \Pi)$

$op)) \}$

<proof>

lemma *decode-plan-step-element-then*:

fixes $\Pi::'a \text{ strips-problem}$

assumes $k < t$

and $op \in set ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$

shows $op \in set ((\Pi)_O)$

and $\mathcal{A} (Operator k (index (strips\text{-}problem.operators\text{-}of \Pi) op))$

<proof>

lemma *decode-plan-step-distinct*:

assumes $k < t$

shows *distinct* $((\Phi^{-1} \Pi \mathcal{A} t) ! k)$
 $\langle proof \rangle$

lemma *decode-state-at-valid-variable:*

fixes $\Pi :: 'a \text{ strips-problem}$
assumes $(\Phi_S^{-1} \Pi \mathcal{A} k) v \neq \text{None}$
shows $v \in \text{set } ((\Pi)_V)$

$\langle proof \rangle$

lemma *decode-state-at-encoding-variables-equals-some-of-valuation-if:*

fixes $\Pi :: 'a \text{ strips-problem}$
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $k \leq t$
and $v \in \text{set } ((\Pi)_V)$
shows $(\Phi_S^{-1} \Pi \mathcal{A} k) v$
 $= \text{Some } (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$

$\langle proof \rangle$

lemma *decode-state-at-dom:*

assumes *is-valid-problem-strips* Π
shows $\text{dom } (\Phi_S^{-1} \Pi \mathcal{A} k) = \text{set } ((\Pi)_V)$

$\langle proof \rangle$

lemma *decode-state-at-initial-state:*

assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
shows $(\Phi_S^{-1} \Pi \mathcal{A} 0) = (\Pi)_I$

$\langle proof \rangle$

lemma *decode-state-at-goal-state:*

assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
shows $(\Pi)_G \subseteq_m \Phi_S^{-1} \Pi \mathcal{A} t$

$\langle proof \rangle$

lemma *decode-state-at-preconditions:*

assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $k < t$
and $op \in \text{set } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$
and $v \in \text{set } (\text{precondition-of } op)$
shows $\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))$

$\langle proof \rangle$

lemma *encode-problem-parallel-correct-i:*

assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi 0$
shows $\text{cnf } ((\Phi_G \Pi) 0) \subseteq \text{cnf } (\Phi_I \Pi)$

$\langle proof \rangle$

lemma *encode-problem-parallel-correct-ii:*
assumes *is-valid-problem-strips* Π
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* $((\Phi^{-1} \Pi \mathcal{A} t) ! k)$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-iii:*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $k < \text{length} (\Phi^{-1} \Pi \mathcal{A} t)$
and $op \in \text{set} ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$
shows $v \in \text{set} (\text{add-effects-of } op)$
 $\rightarrow (\Phi_S^{-1} \Pi \mathcal{A} (\text{Suc } k)) v = \text{Some True}$
and $v \in \text{set} (\text{delete-effects-of } op)$
 $\rightarrow (\Phi_S^{-1} \Pi \mathcal{A} (\text{Suc } k)) v = \text{Some False}$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-iv:*
fixes Π :: *'a strips-problem*
assumes *is-valid-problem-strips* Π
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-semantic* $\mathcal{A} \{ \{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1}$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+ \} \}$
and *cnf-semantic* $\mathcal{A} \{ \{ (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v))^+$
 $, (\text{State } (\text{Suc } k) (\text{index } (\text{strips-problem.variables-of } \Pi) v))^{-1} \} \}$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-v:*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $k < \text{length} (\Phi^{-1} \Pi \mathcal{A} t)$
shows $(\Phi_S^{-1} \Pi \mathcal{A} (\text{Suc } k)) = \text{execute-parallel-operator } (\Phi_S^{-1} \Pi \mathcal{A} k) ((\Phi^{-1} \Pi$
 $\mathcal{A} t) ! k)$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-vi:*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $k < \text{length} (\text{trace-parallel-plan-strips } ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t))$
shows *trace-parallel-plan-strips* $((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t) ! k$
 $= \Phi_S^{-1} \Pi \mathcal{A} k$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-vii:*

assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
shows $\text{length } (\text{map } (\text{decode-state-at } \Pi \mathcal{A})$
 $\quad [0..<\text{Suc } (\text{length } (\Phi^{-1} \Pi \mathcal{A} t))])$
 $\quad = \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t))$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-x*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
shows $\text{map } (\text{decode-state-at } \Pi \mathcal{A})$
 $\quad [0..<\text{Suc } (\text{length } (\Phi^{-1} \Pi \mathcal{A} t))]$
 $\quad = \text{trace-parallel-plan-strips } ((\Pi)_I) (\Phi^{-1} \Pi \mathcal{A} t)$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-correct-xi*:
fixes Π :: 'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
and $\text{ops} \in \text{set } (\Phi^{-1} \Pi \mathcal{A} t)$
and $\text{op} \in \text{set } \text{ops}$
shows $\text{op} \in \text{set } ((\Pi)_O)$
 $\langle \text{proof} \rangle$

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 $\text{ops} \in \text{set } \pi$ 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 .⁸

theorem *encode-problem-parallel-sound*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi \Pi t$
shows *is-parallel-solution-for-problem* $\Pi (\Phi^{-1} \Pi \mathcal{A} t)$
<proof>

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 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* $\Pi \pi \tau \equiv \text{let}$
 $t' = \text{length } \tau$
 $;$ $\tau_\Omega = \tau ! (t' - 1)$
 $;$ $vs = \text{variables-of } \Pi$
 $;$ $V_1 = \{ \text{State } k \text{ (index vs } v) \mid k \ v. \ k \in \{0..<t'\} \wedge v \in \text{set } vs \}$
 $;$ $V_2 = \{ \text{State } k \text{ (index vs } v) \mid k \ v. \ k \in \{t'..(\text{length } \pi + 1)\} \wedge v \in \text{set } vs \}$
 $;$ $\mathcal{A}_1 = \text{foldr}$
 $(\lambda(k, v) \ \mathcal{A}. \text{valuation-for-state (variables-of } \Pi) (\tau ! k) \ k \ v \ \mathcal{A})$
 $(\text{List.product } [0..<t'] \ vs)$
 \mathcal{A}_0
 $;$ $\mathcal{A}_2 = \text{foldr}$
 $(\lambda(k, v) \ \mathcal{A}. \text{valuation-for-state (variables-of } \Pi) \ \tau_\Omega \ k \ v \ \mathcal{A})$

⁸This lemma is used in the proof but not shown.

(*List.product* [*t'..<length* $\pi + 2$] *vs*)
 \mathcal{A}_0
in override-on (*override-on* \mathcal{A}_0 \mathcal{A}_1 V_1) \mathcal{A}_2 V_2

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

definition *valuation-for-operator-variables*

:: 'variable strips-problem
 \Rightarrow 'variable strips-operator list list
 \Rightarrow 'variable strips-state list
 \Rightarrow sat-plan-variable valuation
where *valuation-for-operator-variables* Π π $\tau \equiv$ *let*
ops = operators-of Π
; Op = { Operator k (index ops op) | k op. k \in {0.. $\text{length } \tau - 1$ } \wedge op \in
set ops }
in override-on
 \mathcal{A}_0
(foldr
($\lambda(k, op)$ \mathcal{A} . $\mathcal{A}(\text{Operator } k (\text{index ops op}) := \text{True})$)
(concat (map (λk . map (Pair k) ($\pi ! k$)) [0.. $\text{length } \tau - 1$]))
 \mathcal{A}_0)
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

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

or all $v \in \mathcal{V}$ where $s \equiv \tau ! k$.⁹

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

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

for all operators $op \in \mathcal{O}$ s.t. $op \in \text{set } (\pi ! k)$ for all $k < \text{length } \tau - 1$.

The overall valuation for the plan execution \mathcal{A} can now be constructed by combining the state variable valuation \mathcal{A}_V and operator valuation \mathcal{A}_O .

definition *valuation-for-plan*

⁹It is helpful to remember at this point, that the trace elements of a solution contain the states reached by plan prefix execution (lemma ??).

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

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

lemma *model-of-encode-problem-makespan-zero-iff*:
 $\mathcal{A} \models \Phi \Pi \ 0 \iff \mathcal{A} \models \Phi_I \Pi \wedge (\Phi_G \Pi) \ 0$
<proof>

lemma *empty-valuation-is-False[simp]*: $\mathcal{A}_0 \ v =$ *False*
<proof>

lemma *model-initial-state-set-valuations*:
assumes *is-valid-problem-strips* Π
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>

lemma *valuation-of-state-variable-implies-lit-antics-if*:
assumes $v \in$ *dom* S
and $\mathcal{A}(\text{State } k \ (\text{index } vs \ v)) =$ *the* $(S \ v)$
shows *lit-antics* \mathcal{A} (*literal-formula-to-literal (encode-state-variable* k (*index* $vs \ v)$ $(S \ v)))$
<proof>

lemma *foldr-fun-upd*:

assumes *inj-on* f (*set* xs)
and $x \in \text{set } xs$
shows $\text{foldr } (\lambda x \mathcal{A}. \mathcal{A}(f x := g x)) xs \mathcal{A} (f x) = g x$
 $\langle \text{proof} \rangle$

lemma *foldr-fun-no-upd*:

assumes *inj-on* f (*set* xs)
and $y \notin f \text{ ` set } xs$
shows $\text{foldr } (\lambda x \mathcal{A}. \mathcal{A}(f x := g x)) xs \mathcal{A} y = \mathcal{A} y$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-complete-i*:

fixes Π ::'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$
 $\forall v k. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$
 $\longrightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$
 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$
 $\wedge (\neg \mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$
 $\longleftrightarrow ((\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v \neq \text{Some True})$
shows $\mathcal{A} \models \Phi_I \Pi$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-complete-ii*:

fixes Π ::'a *strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m \text{execute-parallel-plan } ((\Pi)_I) \pi$
and $\forall v k. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$
 $\longrightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$
 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$
and $\forall v l. l \geq \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) \wedge l < \text{length } \pi + 1$
 $\longrightarrow \mathcal{A} (\text{State } l (\text{index } (\text{strips-problem.variables-of } \Pi) v))$
 $= \mathcal{A} (\text{State } (\text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi) - 1)$
 $(\text{index } (\text{strips-problem.variables-of } \Pi) v))$
shows $\mathcal{A} \models (\Phi_G \Pi)(\text{length } \pi)$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-complete-iii-a*:

fixes Π ::'a *strips-problem*
assumes *is-valid-problem-strips* Π
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 $\forall v k. k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi)$
 $\longrightarrow (\mathcal{A} (\text{State } k (\text{index } (\text{strips-problem.variables-of } \Pi) v)))$
 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \pi ! k) v = \text{Some True}$

shows *clause-semantics* $\mathcal{A} C$
 ⟨*proof*⟩

lemma *encode-problem-parallel-complete-iii-b:*

fixes $\Pi :: 'a$ *strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m$ *execute-parallel-plan* $((\Pi)_I) \pi$
and $C \in$ *cnf* (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π)
 (*length* π))
and $\forall k$ *op.* $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $- 1$
 $\longrightarrow \mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π) *op*)) = (*op* \in *set*
 ($\pi ! k$))
and $\forall l$ *op.* $l \geq$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $- 1 \wedge l <$ *length* π
 $\longrightarrow \neg \mathcal{A}$ (*Operator* l (*index* (*strips-problem.operators-of* Π) *op*))
and $\forall v$ $k.$ $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$)
 $\longrightarrow (\mathcal{A}$ (*State* k (*index* (*strips-problem.variables-of* Π) v))
 \longleftrightarrow (*trace-parallel-plan-strips* $((\Pi)_I) \pi ! k$) $v =$ *Some True*)
shows *clause-semantics* $\mathcal{A} C$
 ⟨*proof*⟩

lemma *encode-problem-parallel-complete-iii:*

fixes $\Pi :: 'a$ *strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m$ *execute-parallel-plan* $((\Pi)_I) \pi$
and $\forall k$ *op.* $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $- 1$
 $\longrightarrow \mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π) *op*)) = (*op* \in *set*
 ($\pi ! k$))
and $\forall l$ *op.* $l \geq$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $- 1 \wedge l <$ *length* π
 $\longrightarrow \neg \mathcal{A}$ (*Operator* l (*index* (*strips-problem.operators-of* Π) *op*))
and $\forall v$ $k.$ $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$)
 $\longrightarrow (\mathcal{A}$ (*State* k (*index* (*strips-problem.variables-of* Π) v))
 \longleftrightarrow (*trace-parallel-plan-strips* $((\Pi)_I) \pi ! k$) $v =$ *Some True*)
shows $\mathcal{A} \models$ *encode-operators* Π (*length* π)
 ⟨*proof*⟩

lemma *encode-problem-parallel-complete-iv-a:*

fixes $\Pi :: 'a$ *strips-problem*
assumes *STRIPS-Semantics.is-parallel-solution-for-problem* $\Pi \pi$
and $\forall k$ *op.* $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $- 1$
 $\longrightarrow \mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π) *op*)) = (*op* \in *set*
 ($\pi ! k$))
and $\forall v$ $k.$ $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$)
 $\longrightarrow (\mathcal{A}$ (*State* k (*index* (*strips-problem.variables-of* Π) v))
 \longleftrightarrow (*trace-parallel-plan-strips* $((\Pi)_I) \pi ! k$) $v =$ *Some True*)
and $\forall v$ $l.$ $l \geq$ *length* (*trace-parallel-plan-strips* $((\Pi)_I) \pi$) $\wedge l <$ *length* $\pi + 1$
 $\longrightarrow \mathcal{A}$ (*State* l (*index* (*strips-problem.variables-of* Π) v))
 = \mathcal{A} (*State*

$(length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1)$
 $(index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
and $C \in \bigcup (\bigcup (k, v) \in \{0..<length\ \pi\} \times set\ ((\Pi)_V).$
 $\{\{\{ (State\ k\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))^+$
 $, (State\ (Suc\ k)\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))^{-1}\}$
 $\cup \{ (Operator\ k\ (index\ (strips\text{-}problem.\text{operators-of}\ \Pi)\ op))^+$
 $| op.\ op \in set\ ((\Pi)_O) \wedge v \in set\ (add\text{-}effects\text{-}of\ op)\ \}\}\}$
shows *clause-antics* $\mathcal{A}\ C$
 $\langle proof \rangle$

lemma *encode-problem-parallel-complete-iv-b:*

fixes $\Pi :: 'a\ strips\text{-}problem$
assumes *is-parallel-solution-for-problem* $\Pi\ \pi$
and $\forall k\ op.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1$
 $\longrightarrow \mathcal{A}\ (Operator\ k\ (index\ (strips\text{-}problem.\text{operators-of}\ \Pi)\ op)) = (op \in set$
 $(\pi!\ k))$
and $\forall v\ k.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi)$
 $\longrightarrow (\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
 $\longleftrightarrow (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi!\ k)\ v = Some\ True)$
and $\forall v\ l.\ l \geq length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) \wedge l < length\ \pi + 1$
 $\longrightarrow \mathcal{A}\ (State\ l\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
 $= \mathcal{A}\ (State$
 $(length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1)$
 $(index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
and $C \in \bigcup (\bigcup (k, v) \in \{0..<length\ \pi\} \times set\ ((\Pi)_V).$
 $\{\{\{ (State\ k\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))^{-1}$
 $, (State\ (Suc\ k)\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))^+\}$
 $\cup \{ (Operator\ k\ (index\ (strips\text{-}problem.\text{operators-of}\ \Pi)\ op))^+$
 $| op.\ op \in set\ ((\Pi)_O) \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}\}\}$
shows *clause-antics* $\mathcal{A}\ C$
 $\langle proof \rangle$

lemma *encode-problem-parallel-complete-iv:*

fixes $\Pi :: 'a\ strips\text{-}problem$
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* $\Pi\ \pi$
and $\forall k\ op.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1$
 $\longrightarrow \mathcal{A}\ (Operator\ k\ (index\ (strips\text{-}problem.\text{operators-of}\ \Pi)\ op)) = (op \in set$
 $(\pi!\ k))$
and $\forall v\ k.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi)$
 $\longrightarrow (\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
 $\longleftrightarrow (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi!\ k)\ v = Some\ True)$
and $\forall v\ l.\ l \geq length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) \wedge l < length\ \pi + 1$
 $\longrightarrow \mathcal{A}\ (State\ l\ (index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$
 $= \mathcal{A}\ (State$
 $(length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1)$
 $(index\ (strips\text{-}problem.\text{variables-of}\ \Pi)\ v))$

shows $\mathcal{A} \models \text{encode-all-frame-axioms } \Pi \ (\text{length } \pi)$
 ⟨proof⟩

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* $\Pi \ \pi \ (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi)$
 (*Operator* $k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op)$)
 = ($op \in \text{set } (\pi ! k)$)
 ⟨proof⟩

lemma *encode-problem-parallel-complete-vi-a:*

fixes $\Pi :: 'a \ \text{strips-problem}$
assumes *is-parallel-solution-for-problem* $\Pi \ \pi$
and $k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi) - 1$
shows *valuation-for-plan* $\Pi \ \pi \ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))$
 = ($op \in \text{set } (\pi ! k)$)
 ⟨proof⟩

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⟩

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

fixes $\Pi :: 'variable \ \text{strips-problem}$
assumes *is-parallel-solution-for-problem* $\Pi \ \pi$
and $k < \text{length } \pi$
and $op \notin \text{set } (\pi ! k)$
shows $\neg \text{valuation-for-plan } \Pi \ \pi \ (\text{Operator } k \ (\text{index } (\text{strips-problem.operators-of } \Pi) \ op))$
 ⟨proof⟩

lemma *list-product-is-nil-iff:* $\text{List.product } xs \ ys = [] \longleftrightarrow xs = [] \vee ys = []$

⟨proof⟩

lemma *valuation-for-state-variables-is:*

assumes $k \in \text{set } ks$
and $v \in \text{set } vs$

shows $\text{foldr } (\lambda(k, v) \mathcal{A}. \text{valuation-for-state } vs (s \ k) \ k \ v \ \mathcal{A}) \ (\text{List.product } ks \ vs)$
 \mathcal{A}_0
 (State k (index $vs \ v$))
 $\longleftrightarrow (s \ k) \ v = \text{Some True}$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-complete-vi-c*:
fixes $\Pi :: 'a \ \text{strips-problem}$
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* $\Pi \ \pi$
and $k < \text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi)$
shows *valuation-for-plan* $\Pi \ \pi$ (State k (index (strips-problem.variables-of Π) v))
 $\longleftrightarrow (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi \ ! \ k) \ v = \text{Some True}$
 $\langle \text{proof} \rangle$

lemma *encode-problem-parallel-complete-vi-f*:
fixes $\Pi :: 'a \ \text{strips-problem}$
assumes *is-valid-problem-strips* Π
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* $\Pi \ \pi$ (State l (index (strips-problem.variables-of Π) v))
 $= \text{valuation-for-plan } \Pi \ \pi$
 (State ($\text{length } (\text{trace-parallel-plan-strips } ((\Pi)_I) \ \pi) - 1$)
 (index (strips-problem.variables-of Π) v))
 $\langle \text{proof} \rangle$

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:
¹⁰

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

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

¹⁰Cf. [3, Theorem 3.1, p. 1044] for the construction of \mathcal{A} .

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

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

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 $\mathcal{A} (\text{State } k \text{ (index } (\Pi_{\gamma}) v))$ 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 $\neg \mathcal{A} (\text{State } k \text{ (index } (\Pi_{\gamma}) v))$ otherwise).

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

value *stop*

— To show completeness—i.e. every valid parallel plan 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* Π

and *is-parallel-solution-for-problem* $\Pi \pi$

shows *valuation-for-plan* $\Pi \pi \models \Phi \Pi (\text{length } \pi)$

<proof>

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

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

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

definition *encode-interfering-operator-pair-exclusion*
 $::$ 'variable strips-problem
 \Rightarrow nat
 \Rightarrow 'variable strips-operator
 \Rightarrow 'variable strips-operator
 \Rightarrow sat-plan-variable formula
where *encode-interfering-operator-pair-exclusion* Π k op_1 op_2
 \equiv let $ops = \text{operators-of } \Pi$ in
 $\neg(\text{Atom } (\text{Operator } k \text{ (index ops } op_1)))$
 $\vee \neg(\text{Atom } (\text{Operator } k \text{ (index ops } op_2)))$

definition *encode-interfering-operator-exclusion*
 $::$ 'variable strips-problem \Rightarrow nat \Rightarrow sat-plan-variable formula
where *encode-interfering-operator-exclusion* Π $t \equiv$ let
 $ops = \text{operators-of } \Pi$
 $;$ *interfering* = filter ($\lambda(op_1, op_2). \text{index ops } op_1 \neq \text{index ops } op_2$
 $\wedge \text{are-operators-interfering } op_1 \text{ } op_2$) (*List.product ops ops*)
in foldr (\wedge) [*encode-interfering-operator-pair-exclusion* Π k op_1 op_2 .
 $(op_1, op_2) \leftarrow \text{interfering}, k \leftarrow [0..<t]$] ($\neg \perp$)

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

definition *encode-problem-with-operator-interference-exclusion*
 $::$ 'variable strips-problem \Rightarrow nat \Rightarrow sat-plan-variable formula
 $(\langle \Phi_{\forall} - \rangle 52)$
where *encode-problem-with-operator-interference-exclusion* Π t
 \equiv *encode-initial-state* Π
 \wedge (*encode-operators* Π t
 \wedge (*encode-all-frame-axioms* Π t
 \wedge (*encode-interfering-operator-exclusion* Π t
 \wedge (*encode-goal-state* Π t))))

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

lemma *cnf-of-encode-interfering-operator-pair-exclusion-is-i[simp]*:
 $\text{cnf } (\text{encode-interfering-operator-pair-exclusion } \Pi$ k op_1 $op_2) = \{ \{$
 $(\text{Operator } k \text{ (index (strips-problem.operators-of } \Pi) op_1))^{-1}$

, (Operator k (index (strips-problem.operators-of Π) op₂))⁻¹ } }
 ⟨proof⟩

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⟩

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]]
 = (⋃ (op₁, op₂)
 ∈ { (op₁, op₂) ∈ set (strips-problem.operators-of Π) × set (strips-problem.operators-of
 Π).
 index (strips-problem.operators-of Π) op₁ ≠ index (strips-problem.operators-of
 Π) op₂
 ∧ are-operators-interfering op₁ op₂ }.
 { { { (Operator k (index (strips-problem.operators-of Π) op₁))⁻¹
 , (Operator k (index (strips-problem.operators-of Π) op₂))⁻¹ } } | k. k ∈
 {0..<t} } })
 ⟨proof⟩

lemma *cnf-of-encode-interfering-operator-exclusion-is*:

cnf (encode-interfering-operator-exclusion Π t) = ⋃ (⋃ (op₁, op₂)
 ∈ { (op₁, op₂) ∈ set (operators-of Π) × set (operators-of Π).
 index (strips-problem.operators-of Π) op₁ ≠ index (strips-problem.operators-of
 Π) op₂
 ∧ are-operators-interfering op₁ op₂ }.

$\{\{\{ (Operator\ k\ (index\ (strips\ problem.\ operators\ of\ \Pi)\ op_1))^{-1}$
 $, (Operator\ k\ (index\ (strips\ problem.\ operators\ of\ \Pi)\ op_2))^{-1} \}\} \mid k.\ k \in$
 $\{0..<t\}\}\}$
 <proof>

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>

lemma *is-cnf-encode-interfering-operator-exclusion:*

fixes $\Pi :: 'variable\ strips\ problem$
shows *is-cnf* $(encode\ interfering\ operator\ exclusion\ \Pi\ t)$
 <proof>

lemma *is-cnf-encode-problem-with-operator-interference-exclusion:*

assumes *is-valid-problem-strips* Π
shows *is-cnf* $(\Phi_{\forall}\ \Pi\ t)$
 <proof>

lemma *cnf-of-encode-problem-with-operator-interference-exclusion-structure:*

shows $cnf\ (\Phi_I\ \Pi) \subseteq cnf\ (\Phi_{\forall}\ \Pi\ t)$
and $cnf\ ((\Phi_G\ \Pi)\ t) \subseteq cnf\ (\Phi_{\forall}\ \Pi\ t)$
and $cnf\ (encode\ operators\ \Pi\ t) \subseteq cnf\ (\Phi_{\forall}\ \Pi\ t)$
and $cnf\ (encode\ all\ frame\ axioms\ \Pi\ t) \subseteq cnf\ (\Phi_{\forall}\ \Pi\ t)$
and $cnf\ (encode\ interfering\ operator\ exclusion\ \Pi\ t) \subseteq cnf\ (\Phi_{\forall}\ \Pi\ t)$
 <proof>

lemma *encode-problem-with-operator-interference-exclusion-has-model-then-also-partial-encodings:*

assumes $\mathcal{A} \models \Phi_{\forall}\ \Pi\ t$
shows $\mathcal{A} \models SAT\ Plan\ Base.\ encode\ initial\ state\ \Pi$
and $\mathcal{A} \models SAT\ Plan\ Base.\ encode\ operators\ \Pi\ t$
and $\mathcal{A} \models SAT\ Plan\ Base.\ encode\ all\ frame\ axioms\ \Pi\ t$
and $\mathcal{A} \models encode\ interfering\ operator\ exclusion\ \Pi\ t$
and $\mathcal{A} \models SAT\ Plan\ Base.\ encode\ goal\ state\ \Pi\ t$
 <proof>

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 Φ and \mathcal{A} . The sublocale proof is shown below. It confirms that the new encoding again a CNF as required by locale CNF .

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 SATPlan_sound .

Moreover, since we additionally added interfering operator exclusion clauses at every timestep, the decoded parallel plan cannot contain any interfering operators in any parallel operator (making it serializable).

lemma *encode-problem-serializable-sound-i:*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi_{\forall} \Pi t$
and $k < t$
and $ops \in \text{set } (\text{subseqs } ((\Phi^{-1} \Pi \mathcal{A} t) ! k))$
shows *are-all-operators-non-interfering ops*
 $\langle \text{proof} \rangle$

theorem *encode-problem-serializable-sound:*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi_{\forall} \Pi t$
shows *is-parallel-solution-for-problem* $\Pi (\Phi^{-1} \Pi \mathcal{A} t)$
and $\forall k < \text{length } (\Phi^{-1} \Pi \mathcal{A} t). \text{ are-all-operators-non-interfering } ((\Phi^{-1} \Pi \mathcal{A} t) ! k)$
 $\langle \text{proof} \rangle$

8.2 Completeness

lemma *encode-problem-with-operator-interference-exclusion-complete-i:*
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* $\Pi \pi$
and $\forall k < \text{length } \pi. \text{ are-all-operators-non-interfering } (\pi ! k)$
shows *valuation-for-plan* $\Pi \pi \models \text{encode-interfering-operator-exclusion } \Pi (\text{length } \pi)$
 $\langle \text{proof} \rangle$

Similar to the soundness proof, we may reuse the previously established facts about the valuation for the completeness proof of the basic SATPlan encoding (SATPlan_complete). 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 $\mathcal{A} \equiv valuation\text{-for-plan}\ \Pi\ \pi$ 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* Π

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>

value *stop*

lemma *encode-problem-forall-step-decoded-plan-is-serializable-i:*

assumes *is-valid-problem-strips* Π

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

shows $(\Pi)_G \subseteq_m execute\text{-serial-plan}\ ((\Pi)_I)\ (concat\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t))$

<proof>

lemma *encode-problem-forall-step-decoded-plan-is-serializable-ii:*

fixes $\Pi :: 'variable\ strips\text{-problem}$

shows *list-all* $(\lambda op. ListMem\ op\ (strips\text{-problem}.operators\text{-of}\ \Pi))$

$(concat\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t))$

<proof>

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'

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

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

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

. ¹¹ Note that the parallel operator execution condition is implicit in the existence of the parallel trace for π with

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

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* Π (*concat* ($\Phi^{-1} \Pi \mathcal{A} t$))

<proof>

end

theory *SAT-Solve-SAS-Plus*

imports *SAS-Plus-STRIPS*

SAT-Plan-Extensions

begin

9 SAT-Solving of SAS+ Problems

lemma *sas-plus-problem-has-serial-solution-iff-i:*

assumes *is-valid-problem-sas-plus* Ψ

and $\mathcal{A} \models \Phi_{\forall} (\varphi \Psi) t$

shows *is-serial-solution-for-problem* Ψ [*$\varphi_{\mathcal{O}}^{-1} \Psi$ op. op \leftarrow concat ($\Phi^{-1} (\varphi \Psi) \mathcal{A} t$)*]

<proof>

¹¹These 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.

lemma *sas-plus-problem-has-serial-solution-iff-ii:*

assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* Ψ ψ
and $h = \text{length } \psi$
shows $\exists \mathcal{A}. (\mathcal{A} \models \Phi_{\forall} (\varphi \Psi) h)$

<proof>

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* Ψ
shows $(\exists \psi. \text{is-serial-solution-for-problem } \Psi \psi) \longleftrightarrow (\exists \mathcal{A} t. \mathcal{A} \models \Phi_{\forall} (\varphi \Psi) t)$

<proof>

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 $(\downarrow \text{SAS-Plus-Representation.sas-plus-operator.precondition-of} = [],$

$$\text{SAS-Plus-Representation.sas-plus-operator.effect-of} = [])$$

lemma *sasp-exec-noops: execute-serial-plan-sas-plus* s (*replicate* n *empty-sasp-action*)
 $= s$

<proof>

definition

prob-with-noop $\Pi \equiv$

$(\downarrow \text{SAS-Plus-Representation.sas-plus-problem.variables-of} = \text{SAS-Plus-Representation.sas-plus-problem.variables-of } \Pi,$

$\Pi,$

$\text{SAS-Plus-Representation.sas-plus-problem.operators-of} = \text{empty-sasp-action}$

$\# \text{SAS-Plus-Representation.sas-plus-problem.operators-of } \Pi,$

$\text{SAS-Plus-Representation.sas-plus-problem.initial-of} = \text{SAS-Plus-Representation.sas-plus-problem.initial-of } \Pi,$

$\Pi,$

$\text{SAS-Plus-Representation.sas-plus-problem.goal-of} = \text{SAS-Plus-Representation.sas-plus-problem.goal-of } \Pi,$

$\Pi,$

$SAS\text{-Plus-Representation.sas-plus-problem.range-of} = SAS\text{-Plus-Representation.sas-plus-problem.range-of}$
 Π)

lemma *sasp-noops-in-noop-problem*: $set (replicate\ n\ empty\ sasp\ action) \subseteq set (SAS\text{-Plus-Representation.sas-plus-problem.range-of}$
 $(prob\text{-with-noop}\ \Pi))$
 $\langle proof \rangle$

lemma *noops-complete*:
 $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ \Psi\ \pi \implies$
 $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ (prob\text{-with-noop}\ \Psi)\ ((replicate\ n\ empty\ sasp\ action)\ @\ \pi)$
 $\langle proof \rangle$

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

lemma *sasp-filter-empty-action*:
 $execute\ serial\ plan\ sas\ plus\ s\ (rem\ noops\ \pi\ s) = execute\ serial\ plan\ sas\ plus\ s\ \pi\ s$
 $\langle proof \rangle$

lemma *noops-sound*:
 $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ (prob\text{-with-noop}\ \Psi)\ \pi\ s \implies$
 $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ \Psi\ (rem\ noops\ \pi\ s)$
 $\langle proof \rangle$

lemma *noops-valid*: $is\ valid\ problem\ sas\ plus\ \Psi \implies is\ valid\ problem\ sas\ plus\ (prob\text{-with-noop}\ \Psi)$
 $\langle proof \rangle$

lemma *sas-plus-problem-has-serial-solution-iff-i'*:
assumes $is\ valid\ problem\ sas\ plus\ \Psi$
and $\mathcal{A} \models \Phi_{\forall} (\varphi (prob\text{-with-noop}\ \Psi))\ t$
shows $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ \Psi$
 $(rem\ noops$
 $(map\ (\lambda op.\ \varphi_O^{-1}\ (prob\text{-with-noop}\ \Psi)\ op)$
 $(concat\ (\Phi^{-1}\ (\varphi (prob\text{-with-noop}\ \Psi))\ \mathcal{A}\ t))))$
 $\langle proof \rangle$

lemma *sas-plus-problem-has-serial-solution-iff-ii'*:
assumes $is\ valid\ problem\ sas\ plus\ \Psi$
and $SAS\text{-Plus-Semantics.is-serial-solution-for-problem}\ \Psi\ \psi$
and $length\ \psi \leq h$
shows $\exists \mathcal{A}.\ (\mathcal{A} \models \Phi_{\forall} (\varphi (prob\text{-with-noop}\ \Psi))\ h)$
 $\langle proof \rangle$
end

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

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

11.1 Translating Fast-Downward's representation to SAS+

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

definition *is-standard-effect* :: *ast-effect* ⇒ *bool*
where *is-standard-effect* ≡ λ(*pre*, -, -, -). *pre* = []

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

fun *rem-effect-implicit-pres*:: *ast-effect* ⇒ *ast-effect* **where**
rem-effect-implicit-pres (*preconds*, *v*, *implicit-pre*, *eff*) = (*preconds*, *v*, *None*, *eff*)

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

fun *rem-implicit-pres-ops* :: *ast-problem* ⇒ *ast-problem* **where**
rem-implicit-pres-ops (*vars*, *init*, *goal*, *ops*) = (*vars*, *init*, *goal*, *map rem-implicit-pres ops*)

definition *consistent-map-lists* *xs1 xs2* ≡ (∀ (*x1*, *x2*) ∈ *set xs1*. ∀ (*y1*, *y2*) ∈ *set xs2*.
x1 = *y1* → *x1* = *y2*)

lemma *map-add-comm*: (∧*x*. *x* ∈ *dom m1* ∧ *x* ∈ *dom m2* ⇒ *m1 x* = *m2 x*) ⇒
m1 ++ *m2* = *m2* ++ *m1*
⟨*proof*⟩

lemma *first-map-add-submap*: (∧*x*. *x* ∈ *dom m1* ∧ *x* ∈ *dom m2* ⇒ *m1 x* = *m2 x*) ⇒
m1 ++ *m2* ⊆_{*m*} *x* ⇒ *m1* ⊆_{*m*} *x*
⟨*proof*⟩

lemma *subsuming-states-map-add*:
(∧*x*. *x* ∈ *dom m1* ∩ *dom m2* ⇒ *m1 x* = *m2 x*) ⇒
m1 ++ *m2* ⊆_{*m*} *s* ↔ (*m1* ⊆_{*m*} *s* ∧ *m2* ⊆_{*m*} *s*)
⟨*proof*⟩

lemma *consistent-map-lists*:

$\llbracket \text{distinct} (\text{map fst} (xs1 @ xs2)); x \in \text{dom} (\text{map-of } xs1) \cap \text{dom} (\text{map-of } xs2) \rrbracket \implies$

$(\text{map-of } xs1) x = (\text{map-of } xs2) x$
 $\langle \text{proof} \rangle$

lemma *subsuming-states-append*:

$\text{distinct} (\text{map fst} (xs @ ys)) \implies$
 $(\text{map-of} (xs @ ys)) \subseteq_m s \iff ((\text{map-of } ys) \subseteq_m s \wedge (\text{map-of } xs) \subseteq_m s)$
 $\langle \text{proof} \rangle$

definition *consistent-pres-op where*

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

definition *consistent-pres-op' where*

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

lemma *consistent-pres-op-then'*: $\text{consistent-pres-op } op \implies \text{consistent-pres-op}' op$

$\langle \text{proof} \rangle$

lemma *rem-implicit-pres-ops-valid-states*:

$\text{ast-problem.valid-states} (\text{rem-implicit-pres-ops } prob) = \text{ast-problem.valid-states } prob$
 $\langle \text{proof} \rangle$

lemma *rem-implicit-pres-ops-lookup-op-None*:

$\text{ast-problem.lookup-operator} (vars, init, goal, ops) name = None \iff$
 $\text{ast-problem.lookup-operator} (\text{rem-implicit-pres-ops} (vars, init, goal, ops)) name$
 $= None$
 $\langle \text{proof} \rangle$

lemma *rem-implicit-pres-ops-lookup-op-Some-1*:

$\text{ast-problem.lookup-operator} (vars, init, goal, ops) name = \text{Some } (n,p,vp,e) \implies$
 $\text{ast-problem.lookup-operator} (\text{rem-implicit-pres-ops} (vars, init, goal, ops)) name =$
 $=$
 $\text{Some } (\text{rem-implicit-pres } (n,p,vp,e))$
 $\langle \text{proof} \rangle$

lemma *rem-implicit-pres-ops-lookup-op-Some-1'*:

$\text{ast-problem.lookup-operator } prob name = \text{Some } (n,p,vp,e) \implies$
 $\text{ast-problem.lookup-operator} (\text{rem-implicit-pres-ops } prob) name =$
 $\text{Some } (\text{rem-implicit-pres } (n,p,vp,e))$
 $\langle \text{proof} \rangle$

lemma *implicit-pres-empty*: $\text{implicit-pres} (\text{map } \text{rem-effect-implicit-pres } effs) = []$

$\langle \text{proof} \rangle$

lemma *rem-implicit-pres-ops-lookup-op-Some-2*:
 $ast\text{-}problem.lookup\text{-}operator (rem\text{-}implicit\text{-}pres\text{-}ops (vars, init, goal, ops)) name = Some\ op$
 $\implies \exists op'. ast\text{-}problem.lookup\text{-}operator (vars, init, goal, ops) name = Some\ op'$
 \wedge
 $(op = rem\text{-}implicit\text{-}pres\ op')$
 $\langle proof \rangle$

lemma *rem-implicit-pres-ops-lookup-op-Some-2'*:
 $ast\text{-}problem.lookup\text{-}operator (rem\text{-}implicit\text{-}pres\text{-}ops\ prob) name = Some\ (n,p,e,c)$
 $\implies \exists op'. ast\text{-}problem.lookup\text{-}operator\ prob\ name = Some\ op' \wedge$
 $((n,p,e,c) = rem\text{-}implicit\text{-}pres\ op')$
 $\langle proof \rangle$

lemma *subsuming-states-def'*:
 $s \in ast\text{-}problem.subsuming\text{-}states\ prob\ ps = (s \in (ast\text{-}problem.valid\text{-}states\ prob))$
 $\wedge ps \subseteq_m s)$
 $\langle proof \rangle$

lemma *rem-implicit-pres-ops-enabled-1*:
 $\llbracket (\wedge op. op \in set (ast\text{-}problem.ast\delta\ prob) \implies consistent\text{-}pres\text{-}op\ op);$
 $ast\text{-}problem.enabled\ prob\ name\ s \rrbracket \implies$
 $ast\text{-}problem.enabled (rem\text{-}implicit\text{-}pres\text{-}ops\ prob) name\ s$
 $\langle proof \rangle$

context *ast-problem*
begin

lemma *lookup-Some-in δ* : $lookup\text{-}operator\ \pi = Some\ op \implies op \in set\ ast\delta$
 $\langle proof \rangle$

end

lemma *rem-implicit-pres-ops-enabled-2*:
assumes $(\wedge op. op \in set (ast\text{-}problem.ast\delta\ prob) \implies consistent\text{-}pres\text{-}op\ op)$
shows $ast\text{-}problem.enabled (rem\text{-}implicit\text{-}pres\text{-}ops\ prob) name\ s \implies$
 $ast\text{-}problem.enabled\ prob\ name\ s$
 $\langle proof \rangle$

lemma *rem-implicit-pres-ops-enabled*:
 $(\wedge op. op \in set (ast\text{-}problem.ast\delta\ prob) \implies consistent\text{-}pres\text{-}op\ op) \implies$
 $ast\text{-}problem.enabled (rem\text{-}implicit\text{-}pres\text{-}ops\ prob) name\ s = ast\text{-}problem.enabled$
 $prob\ name\ s$
 $\langle proof \rangle$

context *ast-problem*
begin

lemma *std-eff-enabled[simp]*:

is-standard-operator (*name*, *pres*, *effs*, *layer*) $\implies s \in \text{valid-states} \implies (\text{filter}(\text{eff-enabled } s) \text{ effs}) = \text{effs}$
 ⟨proof⟩

end

lemma *is-standard-operator-rem-implicit*: *is-standard-operator* (*n,p,vp,v*) \implies
is-standard-operator (*rem-implicit-pres* (*n,p,vp,v*))
 ⟨proof⟩

lemma *is-standard-operator-rem-implicit-pres-ops*:
 $\llbracket (\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta(a,b,c,d)) \implies \text{is-standard-operator } op);$
 $op \in \text{set}(\text{ast-problem.ast}\delta(\text{rem-implicit-pres-ops}(a,b,c,d))) \rrbracket$
 $\implies \text{is-standard-operator } op$
 ⟨proof⟩

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

lemma *in-rem-implicit-pres-δ*:
 $op \in \text{set}(\text{ast-problem.ast}\delta prob) \implies$
 $\text{rem-implicit-pres } op \in \text{set}(\text{ast-problem.ast}\delta(\text{rem-implicit-pres-ops } prob))$
 ⟨proof⟩

lemma *rem-implicit-pres-ops-execute*:
assumes
 $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta prob) \implies \text{is-standard-operator } op)$ **and**
 $s \in \text{ast-problem.valid-states } prob$
shows $\text{ast-problem.execute}(\text{rem-implicit-pres-ops } prob) \text{ name } s = \text{ast-problem.execute}$
 $prob \text{ name } s$
 ⟨proof⟩

lemma *rem-implicit-pres-ops-path-to*:
 $\text{wf-ast-problem } prob \implies$
 $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta prob) \implies \text{consistent-pres-op } op) \implies$
 $(\bigwedge op. op \in \text{set}(\text{ast-problem.ast}\delta prob) \implies \text{is-standard-operator } op) \implies$
 $s \in \text{ast-problem.valid-states } prob \implies$
 $\text{ast-problem.path-to}(\text{rem-implicit-pres-ops } prob) s \pi s s' = \text{ast-problem.path-to}$
 $prob s \pi s s'$
 ⟨proof⟩

lemma *rem-implicit-pres-ops-astG[simp]*: $\text{ast-problem.ast}G(\text{rem-implicit-pres-ops}$
 $prob) =$
 $\text{ast-problem.ast}G prob$
 ⟨proof⟩

lemma *rem-implicit-pres-ops-goal*[simp]: *ast-problem.G* (*rem-implicit-pres-ops prob*)
= *ast-problem.G prob*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-astI*[simp]:
ast-problem.astI (*rem-implicit-pres-ops prob*) = *ast-problem.astI prob*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-init*[simp]: *ast-problem.I* (*rem-implicit-pres-ops prob*)
= *ast-problem.I prob*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-valid-plan*:
assumes *wf-ast-problem prob*
 $(\bigwedge op. op \in set (ast-problem.ast\delta prob) \implies consistent-pres-op op)$
 $(\bigwedge op. op \in set (ast-problem.ast\delta prob) \implies is-standard-operator op)$
shows *ast-problem.valid-plan* (*rem-implicit-pres-ops prob*) $\pi s = ast-problem.valid-plan$
prob πs
⟨*proof*⟩

lemma *rem-implicit-pres-ops-numVars*[simp]:
ast-problem.numVars (*rem-implicit-pres-ops prob*) = *ast-problem.numVars prob*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-numVals*[simp]:
ast-problem.numVals (*rem-implicit-pres-ops prob*) $x = ast-problem.numVals prob$
 x
⟨*proof*⟩

lemma *in-implicit-pres*:
 $(x, a) \in set (implicit-pres effs) \implies (\exists epres\ v\ vp. (epres,x,vp,v) \in set\ effs \wedge vp =$
Some a)
⟨*proof*⟩

lemma *pair4-eqD*: $(a1,a2,a3,a4) = (b1,b2,b3,b4) \implies a3 = b3$
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

lemma *rem-implicit-pres-ops-in δ D*: $op \in \text{set } (ast\text{-problem}.ast\delta (rem\text{-implicit}\text{-pres}\text{-ops } prob))$
 $\implies (\exists op'. op' \in \text{set } (ast\text{-problem}.ast\delta prob) \wedge op = rem\text{-implicit}\text{-pres } op')$
 $\langle proof \rangle$

lemma *rem-implicit-pres-ops-well-formed*:
assumes $(\bigwedge op. op \in \text{set } (ast\text{-problem}.ast\delta prob) \implies consistent\text{-pres}\text{-op } op)$
 $ast\text{-problem}.well\text{-formed } prob$
shows $ast\text{-problem}.well\text{-formed } (rem\text{-implicit}\text{-pres}\text{-ops } prob)$
 $\langle proof \rangle$

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

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

lemma *rem-implicit-pres-is-standard-operator'*:
 $is\text{-standard}\text{-operator } (n,p,es,c) \implies is\text{-standard}\text{-operator}' (rem\text{-implicit}\text{-pres } (n,p,es,c))$
 $\langle proof \rangle$

lemma *rem-implicit-pres-ops-is-standard-operator'*:
 $(\bigwedge op. op \in \text{set } (ast\text{-problem}.ast\delta (vs, I, G, ops))) \implies is\text{-standard}\text{-operator } op$
 \implies
 $\pi \in \text{set } (ast\text{-problem}.ast\delta (rem\text{-implicit}\text{-pres}\text{-ops } (vs, I, G, ops))) \implies is\text{-standard}\text{-operator}' \pi$
 $\langle proof \rangle$

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

context *ast-problem*
begin

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

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

end

context *abs-ast-prob*
begin

lemma *is-valid-vars-1*: $astDom \neq [] \implies abs-ast-variable-section \neq []$
<proof>

end

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

lemma *map-of-zip-map-Some*:
 $v < length\ xs$
 $\implies (map-of\ (zip\ [0..<length\ xs])\ (map\ f\ xs))\ v = Some\ (f\ (xs\ !\ v))$
<proof>

lemma *map-of-zip-Some*:
 $v < length\ xs$
 $\implies (map-of\ (zip\ [0..<length\ xs]\ xs)\ v) = Some\ (xs\ !\ v)$
<proof>

lemma *in-set-zip-lengthE*:
 $(x,y) \in set(zip\ [0..<length\ xs]\ xs) \implies (\llbracket x < length\ xs; xs\ !\ x = y \rrbracket \implies R) \implies R$
<proof>

context *abs-ast-prob*
begin

lemma *is-valid-vars-2*:
shows $list-all\ (\lambda v. abs-range-map\ v \neq None)\ abs-ast-variable-section$
<proof>
end

context *ast-problem*
begin

definition *abs-ast-initial-state*
 $:: nat-sas-plus-state$
where $abs-ast-initial-state \equiv map-of\ (zip\ [0..<length\ astI]\ astI)$

end

context *abs-ast-prob*
begin

lemma *valid-abs-init-1*: $abs-ast-initial-state\ v \neq None \longleftrightarrow v \in set\ abs-ast-variable-section$
<proof>

lemma *abs-range-map-Some*:
shows $v \in set\ abs-ast-variable-section \implies$

$(\text{abs-range-map } v) = \text{Some } [0..<\text{length } (\text{snd } (\text{snd } (\text{astDom } ! v)))]$
 <proof>

lemma *in-abs-v-sec-length*: $v \in \text{set abs-ast-variable-section} \longleftrightarrow v < \text{length astDom}$
 <proof>

lemma [*simp*]: $v < \text{length astDom} \implies (\text{abs-ast-initial-state } v) = \text{Some } (\text{astI } ! v)$
 <proof>

lemma [*simp*]: $v < \text{length astDom} \implies \text{astI } ! v < \text{length } (\text{snd } (\text{snd } (\text{astDom } ! v)))$
 <proof>

lemma [*intro!*]: $v \in \text{set abs-ast-variable-section} \implies x < \text{length } (\text{snd } (\text{snd } (\text{astDom } ! v))) \implies$
 $x \in \text{set } (\text{the } (\text{abs-range-map } v))$
 <proof>

lemma [*intro!*]: $x < \text{length astDom} \implies \text{astI } ! x < \text{length } (\text{snd } (\text{snd } (\text{astDom } ! x)))$
 <proof>

lemma [*simp*]: $\text{abs-ast-initial-state } v = \text{Some } a \implies a < \text{length } (\text{snd } (\text{snd } (\text{astDom } ! v)))$
 <proof>

lemma *valid-abs-init-2*:
 $\text{abs-ast-initial-state } v \neq \text{None} \implies (\text{the } (\text{abs-ast-initial-state } v)) \in \text{set } (\text{the } (\text{abs-range-map } v))$
 <proof>

end

context *ast-problem*
begin

definition *abs-ast-goal*
 :: *nat-sas-plus-state*
 where $\text{abs-ast-goal} \equiv \text{map-of astG}$

end

context *abs-ast-prob*
begin

lemma [*simp*]: $\text{wf-partial-state } s \implies (v, a) \in \text{set } s \implies v \in \text{set abs-ast-variable-section}$
 <proof>

lemma *valid-abs-goal-1*: $\text{abs-ast-goal } v \neq \text{None} \implies v \in \text{set abs-ast-variable-section}$
 <proof>

lemma *in-abs-rangeI*: $wf\text{-}partial\text{-}state\ s \implies (v, a) \in set\ s \implies (a \in set\ (the\ (abs\text{-}range\text{-}map\ v)))$

<proof>

lemma *valid-abs-goal-2*:

$abs\text{-}ast\text{-}goal\ v \neq None \implies (the\ (abs\text{-}ast\text{-}goal\ v)) \in set\ (the\ (abs\text{-}range\text{-}map\ v))$

<proof>

end

context *ast-problem*

begin

definition *abs-ast-operator*

$::\ ast\text{-}operator \Rightarrow nat\text{-}sas\text{-}plus\text{-}operator$

where $abs\text{-}ast\text{-}operator \equiv \lambda(name, preconditions, effects, cost).$

$(\lfloor\ precondition\text{-}of = preconditions,$

$effect\text{-}of = [(v, x). (-, v, -, x) \leftarrow effects] \rfloor)$

end

context *abs-ast-prob*

begin

lemma *abs-rangeI*: $wf\text{-}partial\text{-}state\ s \implies (v, a) \in set\ s \implies (abs\text{-}range\text{-}map\ v \neq None)$

<proof>

lemma *abs-valid-operator-1* [*intro!*]:

$wf\text{-}operator\ op \implies list\text{-}all\ (\lambda(v, a). ListMem\ v\ abs\text{-}ast\text{-}variable\text{-}section)$

$(precondition\text{-}of\ (abs\text{-}ast\text{-}operator\ op))$

<proof>

lemma *wf-operator-preD*: $wf\text{-}operator\ (name, pres, effs, cost) \implies wf\text{-}partial\text{-}state\ pres$

<proof>

lemma *abs-valid-operator-2* [*intro!*]:

$wf\text{-}operator\ op \implies$

$list\text{-}all\ (\lambda(v, a). (\exists y. abs\text{-}range\text{-}map\ v = Some\ y) \wedge ListMem\ a\ (the\ (abs\text{-}range\text{-}map\ v)))$

$(precondition\text{-}of\ (abs\text{-}ast\text{-}operator\ op))$

<proof>

lemma *wf-operator-effE*: $wf\text{-}operator\ (name, pres, effs, cost) \implies$

$(\llbracket distinct\ (map\ (\lambda(-, v, -, -). v)\ effs);$

$\bigwedge epres\ x\ vp\ v. (epres, x, vp, v) \in set\ effs \implies wf\text{-}partial\text{-}state\ epres;$

$\bigwedge epres\ x\ vp\ v. (epres, x, vp, v) \in set\ effs \implies x < numVars;$

$\bigwedge epres\ x\ vp\ v. (epres, x, vp, v) \in set\ effs \implies v < numVals\ x;$

$$\begin{aligned} & \bigwedge \text{epres } x \text{ vp } v. (\text{epres}, x, \text{vp}, v) \in \text{set } \text{effs} \implies \\ & \quad \text{case vp of None} \implies \text{True} \mid \text{Some } v \implies v < \text{numVals } x \\ & \implies P \\ & \implies P \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *abs-valid-operator-3'*:

$$\begin{aligned} & \text{wf-operator } (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies \\ & \quad \text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{map } (\lambda(-, v, -, a). (v, a)) \\ & \text{eff}) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *abs-valid-operator-3[intro!]*:

$$\begin{aligned} & \text{wf-operator } \text{op} \implies \\ & \quad \text{list-all } (\lambda(v, a). \text{ListMem } v \text{ abs-ast-variable-section}) (\text{effect-of } (\text{abs-ast-operator} \\ & \text{op})) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *wf-abs-eff*: $\text{wf-operator } (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies \text{wf-partial-state } (\text{map } (\lambda(-, v, -, a). (v, a)) \text{eff})$

$\langle \text{proof} \rangle$

lemma *abs-valid-operator-4'*:

$$\begin{aligned} & \text{wf-operator } (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies \\ & \quad \text{list-all } (\lambda(v, a). (\text{abs-range-map } v \neq \text{None}) \wedge \text{ListMem } a (\text{the } (\text{abs-range-map} \\ & v))) (\text{map } (\lambda(-, v, -, a). (v, a)) \text{eff}) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *abs-valid-operator-4[intro!]*:

$$\begin{aligned} & \text{wf-operator } \text{op} \implies \\ & \quad \text{list-all } (\lambda(v, a). (\exists y. \text{abs-range-map } v = \text{Some } y) \wedge \text{ListMem } a (\text{the } (\text{abs-range-map} \\ & v))) \\ & \quad (\text{effect-of } (\text{abs-ast-operator } \text{op})) \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *consistent-list-set*: $\text{wf-partial-state } s \implies$

$$\text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') s) s$$

$\langle \text{proof} \rangle$

lemma *abs-valid-operator-5'*:

$$\begin{aligned} & \text{wf-operator } (\text{name}, \text{pre}, \text{eff}, \text{cost}) \implies \\ & \quad \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') \text{pre}) \text{pre} \\ \langle \text{proof} \rangle \end{aligned}$$

lemma *abs-valid-operator-5[intro!]*:

$$\begin{aligned} & \text{wf-operator } \text{op} \implies \\ & \quad \text{list-all } (\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') (\text{precondition-of } (\text{abs-ast-operator} \\ & \text{op}))) \\ & \quad (\text{precondition-of } (\text{abs-ast-operator } \text{op})) \end{aligned}$$

<proof>

lemma *consistent-list-set-2*: *distinct (map fst s) \implies*
list-all ($\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') s$) s
<proof>

lemma *abs-valid-operator-6'*:

assumes *wf-operator (name, pre, eff, cost)*

shows *list-all ($\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') (\text{map } (\lambda(-, v, -, a). (v, a)) \text{ eff}))$*

(map ($\lambda(-, v, -, a). (v, a)) \text{ eff})$

<proof>

lemma *abs-valid-operator-6[intro!]*:

wf-operator op \implies

list-all ($\lambda(v, a). \text{list-all } (\lambda(v', a'). v \neq v' \vee a = a') (\text{effect-of } (\text{abs-ast-operator } op)))$

(effect-of (abs-ast-operator op))

<proof>

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 δ]*

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)*

<proof>

lemma *abs-ast-operator-section-valid*:

list-all (is-valid-operator-sas-plus abs-prob) abs-ast-operator-section

<proof>

lemma *abs-prob-valid: is-valid-problem-sas-plus abs-prob*
<proof>

definition *abs-ast-plan*
 $:: SASP\text{-Semantics.plan} \Rightarrow nat\text{-sas-plus-plan}$
where *abs-ast-plan* πs
 $\equiv map (abs\text{-ast-operator } o \text{ the } o \text{ lookup-operator}) \pi s$

lemma *std-then-implici-effs[simp]: is-standard-operator' (name, pres, effs, layer)*
 $\Rightarrow implicit\text{-pres } effs = []$
<proof>

lemma *[simp]: enabled $\pi s \Rightarrow lookup\text{-operator } \pi = Some (name, pres, effs, layer)$*
 \Rightarrow
is-standard-operator' (name, pres, effs, layer) \Rightarrow
(filter (eff-enabled s) effs) = effs
<proof>

lemma *effs-eq-abs-effs: (effect-of (abs-ast-operator (name, pres, effs, layer))) =*
(map ($\lambda(-,x,-,v). (x,v)$) effs)
<proof>

lemma *exec-eq-abs-execute:*
 $\llbracket enabled \pi s; lookup\text{-operator } \pi = Some (name, preconds, effs, layer);$
 $is\text{-standard-operator}'(name, preconds, effs, layer) \rrbracket \Rightarrow$
 $execute \pi s = (execute\text{-operator-sas-plus } s ((abs\text{-ast-operator } o \text{ the } o \text{ lookup-operator})$
 $\pi))$
<proof>

lemma *enabled-then-sas-applicable:*
 $enabled \pi s \Rightarrow SAS\text{-Plus-Representation.is-operator-applicable-in } s ((abs\text{-ast-operator}$
 $o \text{ the } o \text{ lookup-operator}) \pi)$
<proof>

lemma *path-to-then-exec-serial: $\forall \pi \in set \pi s. lookup\text{-operator } \pi \neq None \Rightarrow$*
path-to s $\pi s s' \Rightarrow$
 $s' \subseteq_m execute\text{-serial-plan-sas-plus } s (abs\text{-ast-plan } \pi s)$
<proof>

lemma *map-of-eq-None-iff:*
 $(None = map\text{-of } xys x) = (x \notin fst \text{ ' (set } xys))$
<proof>

lemma *[simp]: I = abs-ast-initial-state*
<proof>

lemma *[simp]: $\forall \pi \in set \pi s. lookup\text{-operator } \pi \neq None \Rightarrow$*

$op \in set (abs-ast-plan \pi s) \implies op \in set\ abs-ast-operator-section$
 <proof>

end

context *ast-problem*
begin

lemma *path-to-then-lookup-Some*: $(\exists s' \in G. path-to\ s\ \pi s\ s') \implies (\forall \pi \in set\ \pi s. lookup-operator\ \pi \neq None)$
 <proof>

lemma *valid-plan-then-lookup-Some*: $valid-plan\ \pi s \implies (\forall \pi \in set\ \pi s. lookup-operator\ \pi \neq None)$
 <proof>

end

context *abs-ast-prob*
begin

theorem *valid-plan-then-is-serial-sol*:
assumes *valid-plan* πs
shows *is-serial-solution-for-problem* *abs-prob* (*abs-ast-plan* πs)
 <proof>

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 $op \equiv$
 $find\ (\lambda(-, pres, effs, -). precondition-of\ op = pres \wedge$
 $map\ (\lambda(v, a). ([], v, None, a))\ (effect-of\ op) = effs)$
 $ast\delta$

end

context *abs-ast-prob*
begin

lemma *find-Some*: $find\ P\ xs = Some\ x \implies x \in set\ xs \wedge P\ x$
 <proof>

lemma *distinct-find*: $distinct\ (map\ f\ xs) \implies x \in set\ xs \implies find\ (\lambda x'. f\ x' = f\ x)$
 $xs = Some\ x$

$\langle \text{proof} \rangle$

lemma *lookup-operator-find*: $\text{lookup-operator } nme = \text{find } (\lambda op. \text{fst } op = nme) \text{ ast}\delta$
 $\langle \text{proof} \rangle$

lemma *lookup-operator-works-1*: $\text{lookup-action } op = \text{Some } \pi' \implies \text{lookup-operator } (\text{fst } \pi') = \text{Some } \pi'$
 $\langle \text{proof} \rangle$

lemma *lookup-operator-works-2*:
 $\text{lookup-action } (\text{abs-ast-operator } (name, pres, effs, layer)) = \text{Some } (name', pres', effs', layer')$
 $\implies pres = pres'$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $\text{is-standard-operator}' (name, pres, effs, layer) \implies$
 $\text{map } (\lambda(v,a). (\square, v, \text{None}, a)) (\text{effect-of } (\text{abs-ast-operator } (name, pres, effs, layer))) = effs$
 $\langle \text{proof} \rangle$

lemma *lookup-operator-works-3*:
 $\text{is-standard-operator}' (name, pres, effs, layer) \implies (name, pres, effs, layer) \in \text{set } \text{ast}\delta \implies$
 $\text{lookup-action } (\text{abs-ast-operator } (name, pres, effs, layer)) = \text{Some } (name', pres', effs', layer')$
 $\implies effs = effs'$
 $\langle \text{proof} \rangle$

lemma *mem-find-Some*: $x \in \text{set } xs \implies P x \implies \exists x'. \text{find } P xs = \text{Some } x'$
 $\langle \text{proof} \rangle$

lemma [*simp*]: $\text{precondition-of } (\text{abs-ast-operator } (x1, a, aa, b)) = a$
 $\langle \text{proof} \rangle$

lemma *std-lookup-action*: $\text{is-standard-operator}' \text{ ast-op} \implies \text{ast-op} \in \text{set } \text{ast}\delta \implies$
 $\exists \text{ast-op}'. \text{lookup-action } (\text{abs-ast-operator } \text{ast-op}) = \text{Some } \text{ast-op}'$
 $\langle \text{proof} \rangle$

lemma *is-applicable-then-enabled-1*:
 $\text{ast-op} \in \text{set } \text{ast}\delta \implies$
 $\exists \text{ast-op}'. \text{lookup-operator } ((\text{fst } o \text{ the } o \text{ lookup-action } o \text{ abs-ast-operator}) \text{ast-op})$
 $= \text{Some } \text{ast-op}'$
 $\langle \text{proof} \rangle$

lemma *lookup-action-Some-in- δ* : $\text{lookup-action } op = \text{Some } \text{ast-op} \implies \text{ast-op} \in \text{set } \text{ast}\delta$
 $\langle \text{proof} \rangle$

lemma *lookup-operator-eq-name*: $\text{lookup-operator } name = \text{Some } (name', pres, effs, layer) \implies name = name'$
 ⟨proof⟩

lemma *eq-name-eq-pres*: $(name, pres, effs, layer) \in \text{set } ast\delta \implies (name, pres', effs', layer') \in \text{set } ast\delta$
 $\implies pres = pres'$
 ⟨proof⟩

lemma *eq-name-eq-effs*:
 $name = name' \implies (name, pres, effs, layer) \in \text{set } ast\delta \implies (name', pres', effs', layer') \in \text{set } ast\delta$
 $\implies effs = effs'$
 ⟨proof⟩

lemma *is-applicable-then-subsumes*:
 $s \in \text{valid-states} \implies$
 $SAS\text{-Plus-Representation.is-operator-applicable-in } s (\text{abs-ast-operator } (name, pres, effs, layer)) \implies$
 $s \in \text{subsuming-states } (\text{map-of } pres)$
 ⟨proof⟩

lemma *eq-name-eq-pres'*:
 $\llbracket s \in \text{valid-states} ; \text{is-standard-operator}' (name, pres, effs, layer); (name, pres, effs, layer) \in \text{set } ast\delta ;$
 $\text{lookup-operator } ((fst \text{ o the o lookup-action o abs-ast-operator}) (name, pres, effs, layer)) = \text{Some } (name', pres', effs', layer') \rrbracket$
 $\implies pres = pres'$
 ⟨proof⟩

lemma *is-applicable-then-enabled-2*:
 $\llbracket s \in \text{valid-states} ; ast\text{-op} \in \text{set } ast\delta ;$
 $SAS\text{-Plus-Representation.is-operator-applicable-in } s (\text{abs-ast-operator } ast\text{-op});$
 $\text{lookup-operator } ((fst \text{ o the o lookup-action o abs-ast-operator}) ast\text{-op}) = \text{Some } (name, pres, effs, layer) \rrbracket$
 $\implies s \in \text{subsuming-states } (\text{map-of } pres)$
 ⟨proof⟩

lemma *is-applicable-then-enabled-3*:
 $\llbracket s \in \text{valid-states};$
 $\text{lookup-operator } ((fst \text{ o the o lookup-action o abs-ast-operator}) ast\text{-op}) = \text{Some } (name, pres, effs, layer) \rrbracket$
 $\implies s \in \text{subsuming-states } (\text{map-of } (\text{implicit-pres } effs))$
 ⟨proof⟩

lemma *is-applicable-then-enabled*:
 $\llbracket s \in \text{valid-states}; ast\text{-op} \in \text{set } ast\delta;$
 $SAS\text{-Plus-Representation.is-operator-applicable-in } s (\text{abs-ast-operator } ast\text{-op}) \rrbracket$
 $\implies \text{enabled } ((fst \text{ o the o lookup-action o abs-ast-operator}) ast\text{-op}) s$

<proof>

lemma *eq-name-eq-effs'*:

assumes *lookup-operator* ((fst o the o *lookup-action* o *abs-ast-operator*) (name, pres, effs, layer)) =

Some (name', pres', effs', layer')

is-standard-operator' (name, pres, effs, layer) (name, pres, effs, layer) ∈ set astδ

s ∈ *valid-states*

shows effs = effs'

<proof>

lemma *std-eff-enabled'[simp]*:

is-standard-operator' (name, pres, effs, layer) ⇒ s ∈ *valid-states* ⇒ (filter (*eff-enabled* s) effs) = effs

<proof>

lemma *execute-abs*:

[[s ∈ *valid-states*; ast-op ∈ set astδ;

SAS-Plus-Representation.is-operator-applicable-in s (abs-ast-operator ast-op)]]

⇒

execute ((fst o the o *lookup-action* o *abs-ast-operator*) ast-op) s =

execute-operator-sas-plus s (abs-ast-operator ast-op)

<proof>

fun *sat-preconds-as* **where**

sat-preconds-as s [] = True

| *sat-preconds-as* s (op#ops) =

(*SAS-Plus-Representation.is-operator-applicable-in* s op ∧

sat-preconds-as (execute-operator-sas-plus s op) ops)

lemma *exec-serial-then-path-to'*:

[[s ∈ *valid-states*;

∀ op ∈ set ops. ∃ ast-op ∈ set astδ. op = abs-ast-operator ast-op;

(*sat-preconds-as* s ops)]] ⇒

path-to s (map (fst o the o *lookup-action*) ops) (*execute-serial-plan-sas-plus* s ops)

<proof>

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
<proof>

lemma *rem-condless-sat: sat-preconds-as s (rem-condless-ops s ops)*
<proof>

lemma *set-rem-condlessD: x ∈ set (rem-condless-ops s ops) ⇒ x ∈ set ops*
<proof>

lemma *exec-serial-then-path-to:*
[[s ∈ valid-states;
∀ op ∈ set ops. ∃ ast-op ∈ set astδ. op = abs-ast-operator ast-op]] ⇒
path-to s (((map (fst o the o lookup-action)) o rem-condless-ops s) ops)
(execute-serial-plan-sas-plus s ops)
<proof>

lemma *is-serial-solution-then-abstracted:*
is-serial-solution-for-problem abs-prob ops
⇒ ∀ op ∈ set ops. ∃ ast-op ∈ set astδ. op = abs-ast-operator ast-op
<proof>

lemma *lookup-operator-works-1': lookup-action op = Some π' ⇒ ∃ op. lookup-operator*
(fst π') = op
<proof>

lemma *is-serial-sol-then-valid-plan-1:*
[[is-serial-solution-for-problem abs-prob ops;
π ∈ set ((map (fst o the o lookup-action) o rem-condless-ops I) ops)] ⇒
lookup-operator π ≠ None
<proof>

lemma *is-serial-sol-then-valid-plan-2:*
[[is-serial-solution-for-problem abs-prob ops]] ⇒
(∃ s' ∈ G. path-to I ((map (fst o the o lookup-action) o rem-condless-ops I) ops)
s')
<proof>

end

context *ast-problem*

begin

definition *decode-abs-plan ≡ (map (fst o the o lookup-action) o rem-condless-ops*
I)

end

12 DIMACS-like semantics for CNF formulae

We now push the SAT encoding towards a lower-level representation by replacing the atoms which have variable IDs and time steps into natural numbers.

lemma *gtD*: $((l::nat) < n) \implies (\exists m. n = Suc\ m \wedge l \leq m)$
<proof>

locale *cnf-to-dimacs* =
fixes $h :: nat$ **and** $n\text{-ops} :: nat$
begin

fun *var-to-dimacs* **where**
var-to-dimacs (*Operator* $t\ k$) = $1 + t + k * h$
| *var-to-dimacs* (*State* $t\ k$) = $1 + n\text{-ops} * h + t + k * (h)$

definition *dimacs-to-var* **where**
dimacs-to-var $v \equiv$
if $v < 1 + n\text{-ops} * h$ *then*
Operator $((v - 1) \bmod (h)) ((v - 1) \text{div} (h))$
else
(let $k = ((v - 1) - n\text{-ops} * h)$ *in*
State $(k \bmod (h)) (k \text{div} (h))$)

fun *valid-state-var* **where**
valid-state-var (*Operator* $t\ k$) $\longleftrightarrow t < h \wedge k < n\text{-ops}$
| *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* $t\ k$)
<proof>

lemma *Operator-works*:
valid-state-var (*Operator* $t\ k$) \implies
dimacs-to-var (*var-to-dimacs* (*Operator* $t\ k$)) =
(*Operator* $t\ k$)
<proof>

lemma *sat-plan-to-dimacs-works*:
valid-state-var $sv \implies$
dimacs-to-var (*var-to-dimacs* sv) = sv
<proof>

end

lemma *changing-atoms-works*:

$(\bigwedge x. P x \implies (f \circ g) x = x) \implies (\forall x \in \text{atoms } \text{phi}. P x) \implies M \models \text{phi} \longleftrightarrow M \circ f \models \text{map-formula } g \text{ phi}$
 <proof>

lemma *changing-atoms-works'*:

$M \circ g \models \text{phi} \longleftrightarrow M \models \text{map-formula } g \text{ phi}$
 <proof>

context *cnf-to-dimacs*

begin

lemma *sat-plan-to-dimacs*:

$(\bigwedge sv. sv \in \text{atoms } \text{sat-plan-formula} \implies \text{valid-state-var } sv) \implies$
 $M \models \text{sat-plan-formula}$
 $\longleftrightarrow M \circ \text{dimacs-to-var} \models \text{map-formula } \text{var-to-dimacs } \text{sat-plan-formula}$
 <proof>

lemma *dimacs-to-sat-plan*:

$M \circ \text{var-to-dimacs} \models \text{sat-plan-formula}$
 $\longleftrightarrow M \models \text{map-formula } \text{var-to-dimacs } \text{sat-plan-formula}$
 <proof>

end

locale *sat-solve-sasp* = *abs-ast-prob* Π + *cnf-to-dimacs* *Suc* *h* *Suc* (*length ast* δ)

for Π *h*

begin

lemma *encode-initial-state-valid*:

$sv \in \text{atoms } (\text{encode-initial-state } \text{Prob}) \implies \text{valid-state-var } sv$
 <proof>

lemma *length-operators*: $\text{length } (\text{operators-of } (\varphi (\text{prob-with-noop } \text{abs-prob}))) = \text{Suc } (\text{length } \text{ast}\delta)$

<proof>

lemma *encode-operator-effect-valid-1*: $t < h \implies op \in \text{set } (\text{operators-of } (\varphi (\text{prob-with-noop } \text{abs-prob}))) \implies$

$sv \in \text{atoms}$
 $(\bigwedge (\text{map } (\lambda v.$
 $\neg(\text{Atom } (\text{Operator } t (\text{index } (\text{operators-of } (\varphi (\text{prob-with-noop } \text{abs-prob}))))$
 $op))))$
 $\vee \text{Atom } (\text{State } (\text{Suc } t) (\text{index } vs v)))$
 $\text{asses})) \implies$
 $\text{valid-state-var } sv$
 <proof>

lemma *encode-operator-effect-valid-2*: $t < h \implies op \in \text{set } (\text{operators-of } (\varphi (\text{prob-with-noop } \text{abs-prob})))$

$abs-prob))) \implies$
 $sv \in atoms$
 $(\bigwedge(\text{map } (\lambda v.$
 $\neg(Atom (Operator t (index (operators-of (\varphi (prob-with-noop abs-prob))))$
 $op)))$
 $\vee \neg (Atom (State (Suc t) (index vs v))))$
 $asses)) \implies$
 $valid-state-var sv$
 $\langle proof \rangle$

end

lemma *atoms-And-append*: $atoms (\bigwedge (as1 @ as2)) = atoms (\bigwedge as1) \cup atoms$
 $(\bigwedge as2)$
 $\langle proof \rangle$

context *sat-solve-sasp*
begin

lemma *encode-operator-effect-valid*:
 $sv \in atoms (encode-operator-effect (\varphi (prob-with-noop abs-prob)) t op) \implies$
 $t < h \implies op \in set (operators-of (\varphi (prob-with-noop abs-prob))) \implies$
 $valid-state-var sv$
 $\langle proof \rangle$

end

lemma *foldr-And*: $foldr (\bigwedge) as (\neg \perp) = (\bigwedge as)$
 $\langle proof \rangle$

context *sat-solve-sasp*
begin

lemma *encode-all-operator-effects-valid*:
 $t < Suc h \implies$
 $sv \in atoms (encode-all-operator-effects (\varphi (prob-with-noop abs-prob)) (operators-of$
 $(\varphi (prob-with-noop abs-prob))) t) \implies$
 $valid-state-var sv$
 $\langle proof \rangle$

lemma *encode-operator-precondition-valid-1*:
 $t < h \implies op \in set (operators-of (\varphi (prob-with-noop abs-prob))) \implies$
 $sv \in atoms$
 $(\bigwedge(\text{map } (\lambda v.$
 $\neg (Atom (Operator t (index (operators-of (\varphi (prob-with-noop abs-prob))))$
 $op))) \vee Atom (State t (f v)))$
 $asses)) \implies$
 $valid-state-var sv$
 $\langle proof \rangle$

lemma *encode-operator-precondition-valid:*

$sv \in \text{atoms} (\text{encode-operator-precondition} (\varphi (\text{prob-with-noop abs-prob})) t \text{ op}) \implies$
 $t < h \implies \text{op} \in \text{set} (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \implies$
valid-state-var sv
 ⟨proof⟩

lemma *encode-all-operator-preconditions-valid:*

$t < \text{Suc } h \implies$
 $sv \in \text{atoms} (\text{encode-all-operator-preconditions} (\varphi (\text{prob-with-noop abs-prob}))$
 $(\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) t) \implies$
valid-state-var sv
 ⟨proof⟩

lemma *encode-operators-valid:*

$sv \in \text{atoms} (\text{encode-operators} (\varphi (\text{prob-with-noop abs-prob})) t) \implies t < \text{Suc } h$
 \implies
valid-state-var sv
 ⟨proof⟩

lemma *encode-negative-transition-frame-axiom':*

$t < h \implies$
 $\text{set deleting-operators} \subseteq \text{set} (\text{operators-of} (\varphi (\text{prob-with-noop abs-prob}))) \implies$
 $sv \in \text{atoms}$
 $(\neg (\text{Atom} (\text{State } t \text{ v-idx}))$
 $\vee (\text{Atom} (\text{State} (\text{Suc } t) \text{ v-idx}))$
 $\vee \bigvee (\text{map} (\lambda \text{op. Atom} (\text{Operator } t (\text{index} (\text{operators-of} (\varphi (\text{prob-with-noop}$
 $\text{abs-prob}))) \text{ op})))$
 $\text{deleting-operators})) \implies$
valid-state-var sv
 ⟨proof⟩

lemma *encode-negative-transition-frame-axiom-valid:*

$sv \in \text{atoms} (\text{encode-negative-transition-frame-axiom} (\varphi (\text{prob-with-noop abs-prob}))$
 $t \text{ v}) \implies t < h \implies$
valid-state-var sv
 ⟨proof⟩

lemma *encode-positive-transition-frame-axiom-valid:*

$sv \in \text{atoms} (\text{encode-positive-transition-frame-axiom} (\varphi (\text{prob-with-noop abs-prob}))$
 $t \text{ v}) \implies t < h \implies$
valid-state-var sv
 ⟨proof⟩

lemma *encode-all-frame-axioms-valid:*

$sv \in \text{atoms} (\text{encode-all-frame-axioms} (\varphi (\text{prob-with-noop abs-prob})) t) \implies t <$
 $\text{Suc } h \implies$
valid-state-var sv

<proof>

lemma *encode-goal-state-valid:*

$sv \in atoms (encode-goal-state Prob t) \implies t < Suc h \implies valid-state-var sv$
<proof>

lemma *encode-problem-valid:*

$sv \in atoms (encode-problem (\varphi (prob-with-noop abs-prob)) h) \implies valid-state-var sv$
<proof>

lemma *encode-interfering-operator-pair-exclusion-valid:*

$sv \in atoms (encode-interfering-operator-pair-exclusion (\varphi (prob-with-noop abs-prob)))$
 $t op_1 op_2 \implies t < Suc h \implies$
 $op_1 \in set (operators-of (\varphi (prob-with-noop abs-prob))) \implies op_2 \in set$
 $(operators-of (\varphi (prob-with-noop abs-prob))) \implies$
 $valid-state-var sv$
<proof>

lemma *encode-interfering-operator-exclusion-valid:*

$sv \in atoms (encode-interfering-operator-exclusion (\varphi (prob-with-noop abs-prob))$
 $t) \implies t < Suc h \implies$
 $valid-state-var sv$
<proof>

lemma *encode-problem-with-operator-interference-exclusion-valid:*

$sv \in atoms (encode-problem-with-operator-interference-exclusion (\varphi (prob-with-noop$
 $abs-prob)) h) \implies valid-state-var sv$
<proof>

lemma *planning-by-cnf-dimacs-complete:*

$valid-plan \pi s \implies length \pi s \leq h \implies$
 $\exists M. M \models map-formula var-to-dimacs (\Phi_{\forall} (\varphi (prob-with-noop abs-prob)) h)$
<proof>

lemma *planning-by-cnf-dimacs-sound:*

$\mathcal{A} \models map-formula var-to-dimacs (\Phi_{\forall} (\varphi (prob-with-noop abs-prob)) t) \implies$
 $valid-plan$
 $(decode-abs-plan$
 $(rem-noops$
 $(map (\lambda op. \varphi_O^{-1} (prob-with-noop abs-prob) op)$
 $(concat (\Phi^{-1} (\varphi (prob-with-noop abs-prob)) (\mathcal{A} o var-to-dimacs) t))))))$
<proof>

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* φ_1 @ *disj-to-dimacs* φ_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* φ_1 @ *cnf-to-dimacs* φ_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*:

$x \in set\ xs \implies x \leq find-max\ xs$ (**is** $?P \implies ?Q$)
<proof>

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)

<proof>

lemma *max-var*: $v \in atoms\ (f::nat\ formula) \implies v \leq find-max\ (formula-vars\ f)$

<proof>

definition *dimacs-max-var* $cs \equiv find-max\ (map\ (find-max\ o\ (map\ (nat\ o\ abs)))\ cs)$

lemma *fold-max-ge*: $b \leq a \implies (b::nat) \leq fold\ (\lambda x\ m.\ if\ m \leq x\ then\ x\ else\ m)\ ys\ a$

<proof>

lemma *find-max-append*: $find-max\ (xs\ @\ ys) = max\ (find-max\ xs)\ (find-max\ ys)$

<proof>

definition *dimacs-model*::*int list* \Rightarrow *int list list* \Rightarrow *bool* **where**

$dimacs-model\ ls\ cs \equiv (\forall c \in set\ cs. (\exists l \in set\ ls. l \in set\ c)) \wedge$
 $distinct\ (map\ dimacs-lit-to-var\ ls)$

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*:

$set\ (model-to-dimacs-model\ M\ (vs\ @\ vs')) = set\ (model-to-dimacs-model\ M\ vs) \cup$
 $set\ (model-to-dimacs-model\ M\ vs')$
 $\langle proof \rangle$

lemma *upt-append-sing*: $xs\ @\ [x] = [a..<n-vars] \implies a < n-vars \implies (xs = [a..<n-vars$
 $- 1] \wedge x = n-vars - 1 \wedge n-vars > 0)$
 $\langle proof \rangle$

lemma *upt-eqD*: $upt\ a\ b = upt\ a\ b' \implies (b = b' \vee b' \leq a \vee b \leq a)$
 $\langle proof \rangle$

lemma *pos-in-model*: $M\ n \implies 0 < n \implies n < n-vars \implies int\ n \in set\ (model-to-dimacs-model\ M\ [1..<n-vars])$
 $\langle proof \rangle$

lemma *neg-in-model*: $\neg\ M\ n \implies 0 < n \implies n < n-vars \implies -\ (int\ n) \in set\ (model-to-dimacs-model\ M\ [1..<n-vars])$
 $\langle proof \rangle$

lemma *in-model*: $0 < n \implies n < n-vars \implies int\ n \in set\ (model-to-dimacs-model\ M\ [1..<n-vars]) \vee -\ (int\ n) \in set\ (model-to-dimacs-model\ M\ [1..<n-vars])$
 $\langle proof \rangle$

lemma *model-to-dimacs-model-all-vars*:

$(\forall v \in atoms\ f. 0 < v \wedge v < n-vars) \implies is-cnf\ f \implies M \models f \implies$
 $(\forall n < n-vars. 0 < n \implies (int\ n \in set\ (model-to-dimacs-model\ M\ [(1::nat)..<n-vars])$
 \vee
 $- (int\ n) \in set\ (model-to-dimacs-model\ M\ [(1::nat)..<n-vars])))$
 $\langle proof \rangle$

lemma *cnf-And*: $set\ (cnf-to-dimacs\ (f1 \wedge f2)) = set\ (cnf-to-dimacs\ f1) \cup set\ (cnf-to-dimacs\ f2)$
 $\langle proof \rangle$

lemma *one-always-in*:

$1 < n-vars \implies 1 \in set\ (model-to-dimacs-model\ M\ ([1..<n-vars])) \vee - 1 \in set\ (model-to-dimacs-model\ M\ ([1..<n-vars]))$
 $\langle proof \rangle$

lemma [*simp*]: $(disj-to-dimacs\ (f1 \vee f2)) = (disj-to-dimacs\ f1) @ (disj-to-dimacs\ f2)$

$f2$)
 $\langle proof \rangle$

lemma $[simp]$: $(atoms (f1 \vee f2)) = atoms f1 \cup atoms f2$
 $\langle proof \rangle$

lemma $isdisj-disjD$: $(is-disj (f1 \vee f2)) \implies is-disj f1 \wedge is-disj f2$
 $\langle proof \rangle$

lemma $disj-to-dimacs-sound$:
 $1 < n-vars \implies (\forall v \in atoms f. 0 < v \wedge v < n-vars) \implies is-disj f \implies M \models f$
 $\implies \exists l \in set (model-to-dimacs-model M [(1::nat)..<n-vars]). l \in set (disj-to-dimacs f)$
 $\langle proof \rangle$

lemma $is-cnf-disj$: $is-cnf (f1 \vee f2) \implies (\bigwedge f. f1 \vee f2 = f \implies is-disj f \implies P) \implies P$
 $\langle proof \rangle$

lemma $cnf-to-dimacs-disj$: $is-disj f \implies cnf-to-dimacs f = [disj-to-dimacs f]$
 $\langle proof \rangle$

lemma $model-to-dimacs-model-all-clauses$:
 $1 < n-vars \implies (\forall v \in atoms f. 0 < v \wedge v < n-vars) \implies is-cnf f \implies M \models f \implies$
 $c \in set (cnf-to-dimacs f) \implies \exists l \in set (model-to-dimacs-model M [(1::nat)..<n-vars]).$
 $l \in set c$
 $\langle proof \rangle$

lemma $upt-eq-Cons-conv$:
 $(x \# xs = [i..<j]) = (i < j \wedge i = x \wedge [i+1..<j] = xs)$
 $\langle proof \rangle$

lemma $model-to-dimacs-model-append'$:
 $(model-to-dimacs-model M (vs @ vs')) = (model-to-dimacs-model M vs) @ (model-to-dimacs-model M vs')$
 $\langle proof \rangle$

lemma $model-to-dimacs-neg-nin$:
 $n-vars \leq x \implies int x \notin set (model-to-dimacs-model M [a..<n-vars])$
 $\langle proof \rangle$

lemma $model-to-dimacs-pos-nin$:
 $n-vars \leq x \implies - int x \notin set (model-to-dimacs-model M [a..<n-vars])$
 $\langle proof \rangle$

lemma $int-cases2'$:
 $z \neq 0 \implies (\bigwedge n. 0 \neq (int n) \implies z = int n \implies P) \implies (\bigwedge n. 0 \neq - (int n) \implies$
 $z = - (int n) \implies P) \implies P$
 $\langle proof \rangle$

lemma *model-to-dimacs-model-distinct*:

$1 < n\text{-vars} \implies \text{distinct} (\text{map dimacs-lit-to-var} (\text{model-to-dimacs-model } M [1..<n\text{-vars}])))$
(proof)

lemma *model-to-dimacs-model-sound*:

$1 < n\text{-vars} \implies (\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnf } f \implies M \models f \implies$
 $\text{dimacs-model} (\text{model-to-dimacs-model } M [(1::\text{nat})..<n\text{-vars}]) (\text{cnf-to-dimacs } f)$
(proof)

lemma *model-to-dimacs-model-sound-exists*:

$1 < n\text{-vars} \implies (\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnf } f \implies M \models f \implies$
 $\exists M\text{-dimacs}. \text{dimacs-model } M\text{-dimacs} (\text{cnf-to-dimacs } f)$
(proof)

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 ($\lambda l M. \text{if } (l > 0) \text{ then } M((\text{nat } (\text{abs } l)) := \text{True}) \text{ else } M((\text{nat } (\text{abs } l)) := \text{False}))$
dimacs-M M

lemma *dimacs-model-to-abs-atom*:

$0 < x \implies \text{int } x \in \text{set dimacs-M} \implies \text{distinct} (\text{map dimacs-lit-to-var dimacs-M})$
 $\implies \text{dimacs-model-to-abs dimacs-M } M x$
(proof)

lemma *dimacs-model-to-abs-atom'*:

$0 < x \implies \neg(\text{int } x) \in \text{set dimacs-M} \implies \text{distinct} (\text{map dimacs-lit-to-var dimacs-M})$
 $\implies \neg \text{dimacs-model-to-abs dimacs-M } M x$
(proof)

lemma *model-to-dimacs-model-complete-disj*:

$(\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-disj } f \implies \text{distinct} (\text{map dimacs-lit-to-var dimacs-M})$
 $\implies \text{dimacs-model dimacs-M} (\text{cnf-to-dimacs } f) \implies \text{dimacs-model-to-abs dimacs-M } (\lambda-. \text{False}) \models f$
(proof)

lemma *model-to-dimacs-model-complete*:

$(\forall v \in \text{atoms } f. 0 < v \wedge v < n\text{-vars}) \implies \text{is-cnf } f \implies \text{distinct} (\text{map dimacs-lit-to-var}$

dimacs-M)
 $\implies \text{dimacs-model dimacs-M (cnf-to-dimacs f)} \implies \text{dimacs-model-to-abs dimacs-M } (\lambda\cdot. \text{False}) \models f$
 <proof>

lemma *model-to-dimacs-model-complete-max-var*:
 $(\forall v \in \text{atoms } f. 0 < v) \implies \text{is-cnf } f \implies$
 $\text{dimacs-model dimacs-M (cnf-to-dimacs f)} \implies$
 $\text{dimacs-model-to-abs dimacs-M } (\lambda\cdot. \text{False}) \models f$
 <proof>

lemma *model-to-dimacs-model-sound-max-var*:
 $(\forall v \in \text{atoms } f. 0 < v) \implies \text{is-cnf } f \implies M \models f \implies$
 $\text{dimacs-model (model-to-dimacs-model M [(1::nat)..<(find-max (formula-vars f) + 2)])}$
 $(\text{cnf-to-dimacs } f)$
 <proof>

context *sat-solve-sasp*
begin

lemma [*simp*]: *var-to-dimacs* $sv > 0$
 <proof>

lemma *var-to-dimacs-pos*:
 $v \in \text{atoms (map-formula var-to-dimacs f)} \implies 0 < v$
 <proof>

lemma *map-is-disj*: $\text{is-disj } f \implies \text{is-disj (map-formula F f)}$
 <proof>

lemma *map-is-cnf*: $\text{is-cnf } f \implies \text{is-cnf (map-formula F f)}$
 <proof>

lemma *planning-dimacs-complete*:
 $\text{valid-plan } \pi s \implies \text{length } \pi s \leq h \implies$
 $\text{let cnf-formula} = (\text{map-formula var-to-dimacs}$
 $\quad (\Phi_{\forall} (\varphi (\text{prob-with-noop abs-prob})) h))$
 in
 $\exists \text{dimacs-M. dimacs-model dimacs-M (cnf-to-dimacs cnf-formula)}$
 <proof>

lemma *planning-dimacs-sound*:
 $\text{let cnf-formula} =$
 $(\text{map-formula var-to-dimacs}$
 $\quad (\Phi_{\forall} (\varphi (\text{prob-with-noop abs-prob})) h))$
 in
 $\text{dimacs-model dimacs-M (cnf-to-dimacs cnf-formula)} \implies$
 valid-plan

```

      (decode-abs-plan
       (rem-noops
        (map ( $\lambda op.$   $\varphi_O^{-1}$  (prob-with-noop abs-prob) op)
         (concat
          ( $\Phi^{-1}$  ( $\varphi$  (prob-with-noop abs-prob)) ((dimacs-model-to-abs dimacs-M
            ( $\lambda.$  False)) o var-to-dimacs) h))))))
      ⟨proof⟩

```

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.ast $\delta$  prob))))
   ( $\Phi_{\forall}$  ( $\varphi$  (prob-with-noop (ast-problem.abs-prob prob)) h)))

```

lemma *planning-dimacs-complete-code*:

```

[[ast-problem.well-formed prob;
  $\forall \pi \in \text{set } (ast-problem.ast\delta \text{ prob}). \text{is-standard-operator } \pi$ ;
 ast-problem.valid-plan prob  $\pi s$ ;
 length  $\pi s \leq h$ ]  $\implies$ 
let cnf-formula = (SASP-to-DIMACS h prob) in
   $\exists$  dimacs-M. dimacs-model dimacs-M cnf-formula
⟨proof⟩

```

definition *SASP-to-DIMACS' h prob* \equiv *SASP-to-DIMACS h (rem-implicit-pres-ops prob)*

lemma *planning-dimacs-complete-code'*:

```

[[ast-problem.well-formed prob;
 ( $\bigwedge op. op \in \text{set } (ast-problem.ast\delta \text{ prob}) \implies \text{consistent-pres-op } op$ );
 ( $\bigwedge op. op \in \text{set } (ast-problem.ast\delta \text{ prob}) \implies \text{is-standard-operator } op$ );
 ast-problem.valid-plan prob  $\pi s$ ;
 length  $\pi s \leq h$ ]  $\implies$ 
let cnf-formula = (SASP-to-DIMACS' h prob) in
   $\exists$  dimacs-M. dimacs-model dimacs-M cnf-formula
⟨proof⟩

```

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 \equiv
 if *ast-problem.well-formed prob* then
 if $(\forall op \in \text{set } (ast\text{-problem.}ast\delta \text{ prob}). \text{consistent-pres-op } op)$ then
 if $(\forall op \in \text{set } (ast\text{-problem.}ast\delta \text{ prob}). \text{is-standard-operator } op)$ 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*:
 $\llbracket ast\text{-problem.valid-plan } prob \ \pi s; \text{length } \pi s \leq h; \text{encode } h \text{ prob} = \text{Inl } cnf\text{-formula} \rrbracket \implies$
 $(\exists \text{dimacs-M. dimacs-model } \text{dimacs-M } cnf\text{-formula})$
 <proof>

lemma *encode-complete*:
 $\text{encode } h \text{ prob} = \text{Inr } err \implies$
 $\neg(ast\text{-problem.well-formed } prob \wedge (\forall op \in \text{set } (ast\text{-problem.}ast\delta \text{ prob}). \text{consistent-pres-op } op) \wedge$
 $(\forall op \in \text{set } (ast\text{-problem.}ast\delta \text{ prob}). \text{is-standard-operator } op))$
 <proof>

definition *match-pre where*
 $\text{match-pre} \equiv \lambda(x,v) \ s. \ s \ x = \text{Some } v$

definition *match-pres where*
 $\text{match-pres } pres \ s \equiv \forall pre \in \text{set } pres. \ \text{match-pre } pre \ s$

lemma *match-pres-distinct*:
 $\text{distinct } (\text{map } fst \ pres) \implies \text{match-pres } pres \ s \longleftrightarrow \text{Map.map-of } pres \subseteq_m \ s$
 <proof>

fun *tree-map-of where*
 $\text{tree-map-of } updatea \ T \ [] = T$
 $| \text{tree-map-of } updatea \ T \ ((v,a)\#m) = updatea \ v \ a \ (\text{tree-map-of } updatea \ T \ m)$

context *Map*
begin

abbreviation *tree-map-of'* $\equiv \text{tree-map-of } update$

lemma *tree-map-of-invar*: $\text{invar } T \implies \text{invar } (\text{tree-map-of}' \ T \ pres)$
 <proof>

lemma *tree-map-of-works*: $\text{lookup } (\text{tree-map-of}' \ \text{empty } pres) \ x = \text{map-of } pres \ x$
 <proof>

lemma *tree-map-of-dom*: $\text{dom} (\text{lookup} (\text{tree-map-of}' \text{ empty pres})) = \text{dom} (\text{map-of pres})$
 ⟨proof⟩
end

lemma *distinct-if-sorted*: $\text{sorted } xs \implies \text{distinct } xs$
 ⟨proof⟩

context *Map-by-Ordered*
begin

lemma *tree-map-of-distinct*: $\text{distinct} (\text{map fst} (\text{inorder} (\text{tree-map-of}' \text{ empty pres})))$
 ⟨proof⟩

end

lemma *set-tree-intorder*: $\text{set-tree } t = \text{set} (\text{inorder } t)$
 ⟨proof⟩

lemma *map-of-eq*:
 $\text{map-of } xs = \text{Map.map-of } xs$
 ⟨proof⟩

lemma *lookup-someD*: $\text{lookup } T x = \text{Some } y \implies \exists p. p \in \text{set} (\text{inorder } T) \wedge p = (x, y)$
 ⟨proof⟩

lemma *map-of-lookup*: $\text{sorted1} (\text{inorder } T) \implies \text{Map.map-of} (\text{inorder } T) = \text{lookup } T$
 ⟨proof⟩

lemma *map-le-cong*: $(\bigwedge x. m1 x = m2 x) \implies m1 \subseteq_m s \longleftrightarrow m2 \subseteq_m s$
 ⟨proof⟩

lemma *match-pres-submap*:
 $\text{match-pres} (\text{inorder} (M.\text{tree-map-of}' \text{ empty pres})) s \longleftrightarrow \text{Map.map-of pres} \subseteq_m s$
 ⟨proof⟩

lemma [code]:
 $\text{SAS-Plus-Representation.is-operator-applicable-in } s \text{ op} \longleftrightarrow$
 $\text{match-pres} (\text{inorder} (M.\text{tree-map-of}' \text{ empty} (\text{SAS-Plus-Representation.precondition-of op}))) s$
 ⟨proof⟩

definition *decode-DIMACS-model* $\text{dimacs-M } h \text{ prob} \equiv$
 $(\text{ast-problem.decode-abs-plan prob}$
 $(\text{rem-noops}$
 $(\text{map} (\lambda op. \varphi_O^{-1} (\text{prob-with-noop} (\text{ast-problem.abs-prob prob})) op)) \text{ op})$

```

(concat
  (Φ-1 (φ (prob-with-noop (ast-problem.abs-prob prob))))
  ((dimacs-model-to-abs dimacs-M (λ-. False)) o
   (cnf-to-dimacs.var-to-dimacs (Suc h)
    (Suc (length (ast-problem.astδ prob))))
   h))))

```

lemma *planning-dimacs-sound-code*:

```

[[ast-problem.well-formed prob;
  ∀ π ∈ set (ast-problem.astδ prob). is-standard-operator' π]] ⇒
let
  cnf-formula = (SASP-to-DIMACS h prob);
  decoded-plan = decode-DIMACS-model dimacs-M h prob
in
  (dimacs-model dimacs-M cnf-formula → ast-problem.valid-plan prob de-
   coded-plan)
  ⟨proof⟩

```

definition

```

decode-DIMACS-model' dimacs-M h prob ≡
  decode-DIMACS-model dimacs-M h (rem-implicit-pres-ops prob)

```

lemma *planning-dimacs-sound-code'*:

```

[[ast-problem.well-formed prob;
  (∧ op. op ∈ set (ast-problem.astδ prob) ⇒ consistent-pres-op op);
  ∀ π ∈ set (ast-problem.astδ prob). is-standard-operator π]] ⇒
let
  cnf-formula = (SASP-to-DIMACS' h prob);
  decoded-plan = decode-DIMACS-model' dimacs-M h prob
in
  (dimacs-model dimacs-M cnf-formula → ast-problem.valid-plan prob de-
   coded-plan)
  ⟨proof⟩

```

Checking if the model satisfies the formula takes the longest time in the decoding function. We reimplement that part using red black trees, which makes it 10 times faster, on average!

fun *list-to-rbt* :: *int list* ⇒ *int rbt* **where**

```

  list-to-rbt [] = Leaf
| list-to-rbt (x#xs) = insert-rbt x (list-to-rbt xs)

```

lemma *inv-list-to-rbt*: *invc (list-to-rbt xs) ∧ invh (list-to-rbt xs)*
 ⟨proof⟩

lemma *Tree2-list-to-rbt*: *Tree2.bst (list-to-rbt xs)*
 ⟨proof⟩

lemma *set-list-to-rbt*: *Tree2.set-tree (list-to-rbt xs) = set xs*
 ⟨proof⟩

The following

lemma *dimacs-model-code*[code]:

dimacs-model *ls cs* \longleftrightarrow
 (let *tls* = *list-to-rbt* *ls* in
 ($\forall c \in \text{set } cs. \text{size } (\text{inter-rbt } (tls) (\text{list-to-rbt } c)) \neq 0$) \wedge
distinct (*map dimacs-lit-to-var* *ls*))
 ⟨*proof*⟩

definition

decode *M h prob* \equiv
 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
 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*
 ⟨*proof*⟩

lemma *decode-complete*:

decode *M h prob* = *Inr err* \implies
 $\neg (\text{ast-problem.well-formed } \text{prob} \wedge$
 $(\forall op \in \text{set } (\text{ast-problem.ast}\delta \text{ prob}). \text{consistent-pres-op } op) \wedge$
 $(\forall \pi \in \text{set } (\text{ast-problem.ast}\delta \text{ prob}). \text{is-standard-operator } \pi) \wedge$
dimacs-model *M* (*SASP-to-DIMACS'* *h prob*))
 ⟨*proof*⟩

lemma [code]:

⟨*ListMem* *x xs* \longleftrightarrow *List.member* *xs x*⟩
 ⟨*proof*⟩

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.ast δ -*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* \Rightarrow *nat option* **where**
nat-opt-of-integer *i* = (if (*i* \geq 0) then *Some* (*nat-of-integer* *i*) else *None*)

definition *max-var* :: *int list* \Rightarrow *int* **where**
max-var *xs* \equiv *fold* ($\lambda(x::int) (y::int). \text{if } \text{abs } x \geq \text{abs } y \text{ then } (\text{abs } x) \text{ else } y$) *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>.