Verified SAT-Based AI Planning

Mohammad Abdulaziz and Friedrich Kurz*

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

Contents

*Author names are alphabetically ordered.

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

Propositional-Proof-Systems.CNF
begin

1 State-Variable Representation

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

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

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

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

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

end

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

2 STRIPS Representation

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

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

record $('variable)$ *strips-operator* =
precondition-of :: *'variable list*
add-effects-of :: *'variable list*
delete-effects-of :: *'variable list*

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

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

abbreviation *operator-for*
:: *'variable list* ⇒ *'variable list* ⇒ *'variable list* ⇒ *'variable strips-operator*
where *operator-for pre add delete* ≡ (|
precondition-of = *pre*
, *add-effects-of* = *add*
, *delete-effects-of* = *delete* |)

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

definition *to-effect*
:: *'variable strips-operator* ⇒ (*'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* (($-_{\mathcal{V}}$) [*1000*] *999*)
operators-of :: *'variable strips-operator list* (($-_{\mathcal{O}}$) [*1000*] *999*)
initial-of :: *'variable strips-state* (($-_{I}$) [*1000*] *999*)
goal-of :: *'variable strips-state* (($-_{G}$) [*1000*] *999*)

value *stop*

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

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

end

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

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

3 STRIPS Semantics

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

3.1 Serial Plan Execution Semantics

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

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

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

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

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

definition *is-serial-solution-for-problem*
  where *is-serial-solution-for-problem* Π π
    ≡ (*goal-of* Π) $\subseteq_m$ *execute-serial-plan* (*initial-of* Π) π
      ∧ *list-all* (λ*op*. *ListMem op* (*operators-of* Π)) π

lemma *is-valid-problem-strips-initial-of-dom*:
  fixes Π:: *′a strips-problem*
  assumes *is-valid-problem-strips* Π
  shows *dom* $((\Pi)_I)$ = *set* $((\Pi)_\mathcal{V})$
  ⟨*proof*⟩

lemma *is-valid-problem-dom-of-goal-state-is*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
shows *dom* $((\Pi)_G) \subseteq$ *set* $((\Pi)_{\mathcal{V}})$
⟨*proof*⟩

lemma *is-valid-problem-strips-operator-variable-sets*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and *op* $\in$ *set* $((\Pi)_{\mathcal{O}})$
shows *set* (*precondition-of op*) $\subseteq$ *set* $((\Pi)_{\mathcal{V}})$
and *set* (*add-effects-of op*) $\subseteq$ *set* $((\Pi)_{\mathcal{V}})$
and *set* (*delete-effects-of op*) $\subseteq$ *set* $((\Pi)_{\mathcal{V}})$
and *disjnt* (*set* (*add-effects-of op*)) (*set* (*delete-effects-of op*))
⟨*proof*⟩

lemma *effect-to-assignments-i*:
assumes *as* = *effect-to-assignments op*
shows *as* = (*map* (λv. (v, *True*)) (*add-effects-of op*)
@ *map* (λv. (v, *False*)) (*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* = *effect-to-assignments op*
obtains as_1 as_2
where $as = as_1$ @ as_2
and as_1 = *map* (λv. (v, *True*)) (*add-effects-of op*)
and as_2 = *map* (λv. (v, *False*)) (*delete-effects-of op*)
⟨*proof*⟩
lemma *effect-to-assignments-iii-a*:
fixes v
assumes $v \in$ *set* (*add-effects-of op*)
and *as* = *effect-to-assignments op*
obtains a **where** $a \in$ *set as* $a = (v, \textit{True})$
⟨*proof*⟩

lemma *effect-to-assignments-iii-b*:
— NOTE This proof is symmetrical to the one above.
fixes v
assumes $v \in$ *set* (*delete-effects-of op*)
and *as* = *effect-to-assignments op*
obtains a **where** $a \in$ *set as* $a = (v, \textit{False})$
⟨*proof*⟩

lemma *effect--strips-i*:
fixes *op*

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

lemma *effect--strips-ii*:
fixes *op*
assumes $e = ConjunctiveEffect\ (es_1 \ @ \ es_2)$
and $es_1 = map\ (\lambda v.\ (v,\ True))\ (\textit{add-effects-of op})$
and $es_2 = map\ (\lambda v.\ (v,\ False))\ (\textit{delete-effects-of op})$
shows $\forall\, v \in set\ (\textit{add-effects-of op}).\ (\exists\, e' \in set\ es_1.\ e' = (v,\ True))$
and $\forall\, v \in set\ (\textit{delete-effects-of op}).\ (\exists\, e' \in set\ es_2.\ e' = (v,\ False))$
⟨*proof*⟩

lemma *map-of-constant-assignments-dom*:
— NOTE ancillary lemma used in the proof below.
assumes $m = \textit{map-of}\ (map\ (\lambda v.\ (v,\ d))\ vs)$
shows $dom\ m = set\ vs$
⟨*proof*⟩

lemma *effect--strips-iii-a*:
assumes $s' = (s \gg op)$
shows $\bigwedge v.\ v \in set\ (\textit{add-effects-of op}) \Longrightarrow s'\ v = Some\ True$
⟨*proof*⟩

lemma *effect--strips-iii-b*:
assumes $s' = (s \gg op)$
shows $\bigwedge v.\ v \in set\ (\textit{delete-effects-of op}) \wedge v \notin set\ (\textit{add-effects-of op}) \Longrightarrow s'\ v =$
$Some\ False$
⟨*proof*⟩

lemma *effect--strips-iii-c*:
assumes $s' = (s \gg op)$
shows $\bigwedge v.\ v \notin set\ (\textit{add-effects-of op}) \wedge v \notin set\ (\textit{delete-effects-of op}) \Longrightarrow s'\ v =$
$s\ v$
⟨*proof*⟩

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

- every add effect is satisfied in s' (sublemma); and,

[3] Lemmas `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
 $\bigwedge v.$
 $v \in set\ (add\text{-}effects\text{-}of\ op)$
 $\Longrightarrow s'\ v = Some\ True$
 and $\bigwedge v.$
 $v \notin set\ (add\text{-}effects\text{-}of\ op) \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)$
 $\Longrightarrow s'\ v = Some\ False$
 and $\bigwedge v.$
 $v \notin set\ (add\text{-}effects\text{-}of\ op) \wedge v \notin set\ (delete\text{-}effects\text{-}of\ op)$
 $\Longrightarrow s'\ v = s\ v$
$\langle proof \rangle$

3.2 Parallel Plan Semantics

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

definition *are-operator-effects-consistent* $op_1\ op_2 \equiv let$
 $add_1 = add\text{-}effects\text{-}of\ op_1$
 $;\ add_2 = add\text{-}effects\text{-}of\ op_2$
 $;\ del_1 = delete\text{-}effects\text{-}of\ op_1$
 $;\ del_2 = delete\text{-}effects\text{-}of\ op_2$
 $in\ \neg list\text{-}ex\ (\lambda v.\ list\text{-}ex\ ((=)\ v)\ del_2)\ add_1 \wedge \neg list\text{-}ex\ (\lambda v.\ list\text{-}ex\ ((=)\ v)\ add_2)$
del_1

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

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

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

fun *execute-parallel-plan*

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

definition *are-operators-interfering* op_1 op_2
≡ *list-ex* (λ*v*. *list-ex* ((=) *v*) (*delete-effects-of* op_1)) (*precondition-of* op_2)
∨ *list-ex* (λ*v*. *list-ex* ((=) *v*) (*precondition-of* op_1)) (*delete-effects-of* op_2)

primrec *are-all-operators-non-interfering*
:: *'variable strips-operator list* ⇒ *bool*
where *are-all-operators-non-interfering* [] = *True*
| *are-all-operators-non-interfering* (*op* # *ops*)
= (*list-all* (λ*op'*. ¬*are-operators-interfering op op'*) *ops*
∧ *are-all-operators-non-interfering ops*)

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

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

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

definition *is-parallel-solution-for-problem*
where *is-parallel-solution-for-problem* Π π
≡ (*strips-problem.goal-of* Π) $\subseteq_m$ *execute-parallel-plan*
(*strips-problem.initial-of* Π) π
∧ *list-all* (λ*ops*. *list-all* (λ*op*.
ListMem op (*strips-problem.operators-of* Π)) *ops*) π

lemma *are-all-operators-applicable-set*:
are-all-operators-applicable s ops
⟷ (∀ *op* ∈ *set ops*. ∀ *v* ∈ *set* (*precondition-of op*). *s v* = *Some True*)

⟨*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 $op_1 \in set\ ops$
    and $op_2 \in set\ ops$
  shows *are-operator-effects-consistent* $op_1\ op_2$
    $= (set\ (add\text{-}effects\text{-}of\ op_1) \cap set\ (delete\text{-}effects\text{-}of\ op_2) = \{\}$
    $\wedge\ set\ (delete\text{-}effects\text{-}of\ op_1) \cap set\ (add\text{-}effects\text{-}of\ op_2) = \{\})$
  ⟨*proof*⟩

lemma *are-all-operator-effects-consistent-set*:
  are-all-operator-effects-consistent ops
    $\longleftrightarrow (\forall\ op_1 \in set\ ops.\ \forall\ op_2 \in set\ ops.$
      $(set\ (add\text{-}effects\text{-}of\ op_1) \cap set\ (delete\text{-}effects\text{-}of\ op_2) = \{\})$
        $\wedge\ (set\ (delete\text{-}effects\text{-}of\ op_1) \cap set\ (add\text{-}effects\text{-}of\ op_2) = \{\}))$
  ⟨*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* $op_1\ op_2$
  shows *are-operators-interfering* $op_2\ op_1$
  ⟨*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* $op_1\ op_2$
    and $op_1 \neq op_2$
  shows $op_1 \notin set\ ops \vee op_2 \notin set\ ops$
  ⟨*proof*⟩

lemma *execute-parallel-plan-precondition-cons-i*:
  fixes *s* :: (*'variable*, *bool*) *state*
  assumes ¬*are-operators-interfering op op'*

and *is-operator-applicable-in s op*
and *is-operator-applicable-in s op'*
shows *is-operator-applicable-in* (*s* ++ *map-of* (*effect-to-assignments op*)) *op'*
⟨*proof*⟩
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* (*λv.* (*v*, *True*)) (*add-effects-of op*) @ *map* (*λv.* (*v*, *False*)) (*delete-effects-of op*)
⟨*proof*⟩

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

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

and *effect-to-assignments op* = *map*
(λ*v*. (*v*, *if ListMem v* (*delete-effects-of op*) *then False else True*))
(*add-effects-of op* @ *delete-effects-of op*)
⟨*proof*⟩

corollary *map-of-effect-to-assignments-is-none-if*:
assumes ¬*v* ∈ *set* (*add-effects-of op*)
and ¬*v* ∈ *set* (*delete-effects-of op*)
shows *map-of* (*effect-to-assignments op*) *v* = *None*
⟨*proof*⟩

lemma *execute-parallel-operator-positive-effect-if-i*:
assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and *op* ∈ *set ops*
and *v* ∈ *set* (*add-effects-of op*)
shows *map-of* (*effect-to-assignments op*) *v* = *Some True*
⟨*proof*⟩

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

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

lemma *execute-parallel-operator-negative-effect-if*:
assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and *op* ∈ *set ops*
and *v* ∈ *set* (*delete-effects-of op*)
shows *execute-parallel-operator s ops v* = *Some False*
⟨*proof*⟩

lemma *execute-parallel-operator-no-effect-if*:
assumes ∀ *op* ∈ *set ops*. ¬*v* ∈ *set* (*add-effects-of op*) ∧ ¬*v* ∈ *set* (*delete-effects-of op*)
shows *execute-parallel-operator s ops v* = *s v*
⟨*proof*⟩

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* ∈ *set π*
and *op* ∈ *set ops*
shows *op* ∈ *set* $((\Pi)_{\mathcal{O}})$
⟨*proof*⟩

lemma *trace-parallel-plan-strips-not-nil*: *trace-parallel-plan-strips I π* ≠ []
⟨*proof*⟩

corollary *length-trace-parallel-plan-gt-0*[*simp*]: *0* < *length* (*trace-parallel-plan-strips I π*)
⟨*proof*⟩

corollary *length-trace-minus-one-lt-length-trace*[*simp*]:
length (*trace-parallel-plan-strips I π*) − *1* < *length* (*trace-parallel-plan-strips I π*)
⟨*proof*⟩

lemma *trace-parallel-plan-strips-head-is-initial-state*:
trace-parallel-plan-strips I π ! *0* = *I*
⟨*proof*⟩

lemma *trace-parallel-plan-strips-length-gt-one-if*:
assumes *k* < *length* (*trace-parallel-plan-strips I π*) − *1*
shows *1* < *length* (*trace-parallel-plan-strips I π*)
⟨*proof*⟩
lemma *trace-parallel-plan-strips-last-cons-then*:
last (*s* # *trace-parallel-plan-strips s' π*) = *last* (*trace-parallel-plan-strips s' π*)
⟨*proof*⟩

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

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

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

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

assumes $k <$ *length* (*trace-parallel-plan-strips* I π) $-$ *1*
shows *are-all-operators-applicable* (*trace-parallel-plan-strips* I π ! k) (π ! k)
$\wedge$ *are-all-operator-effects-consistent* (π ! k)
⟨*proof*⟩

Another interesting property that we verify below is that elements of the trace store the result of plan prefix execution. This means that for an index k with
$k <$ *length* (*trace-parallel-plan-strips* I π), the k-th element of the trace is state reached by executing the plan prefix *take* k π consisting of the first k parallel operators of π.

lemma *trace-parallel-plan-plan-prefix*:
assumes $k <$ *length* (*trace-parallel-plan-strips* I π)
shows *trace-parallel-plan-strips* I π ! k = *execute-parallel-plan* I (*take* k π)
⟨*proof*⟩

lemma *length-trace-parallel-plan-strips-lte-length-plan-plus-one*:
shows *length* (*trace-parallel-plan-strips* I π) $\leq$ *length* π + *1*
⟨*proof*⟩
lemma *plan-is-at-least-singleton-plan-if-trace-has-at-least-two-elements*:
assumes $k <$ *length* (*trace-parallel-plan-strips* I π) $-$ *1*
obtains *ops* π' **where** π = *ops* # π'
⟨*proof*⟩
corollary *length-trace-parallel-plan-strips-lt-length-plan-plus-one-then*:
assumes *length* (*trace-parallel-plan-strips* I π) < *length* π + *1*
shows $\neg$*are-all-operators-applicable*
(*execute-parallel-plan* I (*take* (*length* (*trace-parallel-plan-strips* I π) $-$ *1*) π))
(π ! (*length* (*trace-parallel-plan-strips* I π) $-$ *1*))
$\vee$ $\neg$*are-all-operator-effects-consistent* (π ! (*length* (*trace-parallel-plan-strips* I π)
$-$ *1*))
⟨*proof*⟩

lemma *trace-parallel-plan-step-effect-is*:
assumes $k <$ *length* (*trace-parallel-plan-strips* I π) $-$ *1*
shows *trace-parallel-plan-strips* I π ! *Suc* k
= *execute-parallel-operator* (*trace-parallel-plan-strips* I π ! k) (π ! k)
⟨*proof*⟩

lemma *trace-parallel-plan-strips-none-if*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* Π π
and $k <$ *length* (*trace-parallel-plan-strips* $((\Pi)_I)$ π)
shows (*trace-parallel-plan-strips* $((\Pi)_I)$ π ! k) v = *None* $\longleftrightarrow$ $v \notin$ *set* $((\Pi)_{\mathcal{V}})$
⟨*proof*⟩

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

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

lemma *execute-parallel-plan-reaches-goal-iff-goal-is-last-element-of-trace*:
$G \subseteq_m$ *execute-parallel-plan* $I\ \pi$
$\longleftrightarrow G \subseteq_m$ *last* (*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.

— Given that non interference implies that operator execution order can be switched arbitrarily, it stands to reason that parallel operator execution can be serialized if non interference is mandated in addition to the regular parallel execution condition (applicability and effect consistency). This is in fact true as we show in the lemma below [5]

lemma *execute-parallel-operator-equals-execute-sequential-strips-if*:
fixes s :: (*'variable*, *bool*) *state*
assumes *are-all-operators-applicable s ops*
and *are-all-operator-effects-consistent ops*
and *are-all-operators-non-interfering ops*
shows *execute-parallel-operator s ops* = *execute-serial-plan s ops*
⟨*proof*⟩

lemma *execute-serial-plan-split-i*:
assumes *are-all-operators-applicable* s (*op* # π)
and *are-all-operators-non-interfering* (*op* # π)
shows *are-all-operators-applicable* ($s \gg$ *op*) π
⟨*proof*⟩
lemma *execute-serial-plan-split*:
fixes s :: (*'variable*, *bool*) *state*
assumes *are-all-operators-applicable* $s\ \pi_1$
and *are-all-operators-non-interfering* π_1
shows *execute-serial-plan* s (π_1 @ π_2)
= *execute-serial-plan* (*execute-serial-plan* $s\ \pi_1$) π_2

[5] In the source literatur it is required that $\text{app}_S(s)$ is defined which requires that $\text{app}_o(s)$ is defined for every $o \in S$. This again means that the preconditions hold in s and the set of effects is consistent which translates to the execution condition in *execute-parallel-operator*. [3, Lemma 2.11., p.1037]

Also, the proposition [3, Lemma 2.11., p.1037] is in fact proposed to be true for any total ordering of the operator set but we only proof it for the implicit total ordering induced by the specific order in the operator list of the problem statement.

$\langle proof \rangle$

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

lemma *execute-serial-plan-is-execute-parallel-plan-ii*:
fixes I :: *'variable strips-state*
assumes $\forall\, op \in set\ \pi.\ are\text{-}operator\text{-}effects\text{-}consistent\ op\ op$
and $G \subseteq_m execute\text{-}serial\text{-}plan\ I\ \pi$
shows $G \subseteq_m execute\text{-}parallel\text{-}plan\ I\ (embed\ \pi)$
$\langle proof \rangle$

lemma *embedding-lemma-iii*:
fixes Π:: *'a strips-problem*
assumes $\forall\, op \in set\ \pi.\ op \in set\ ((\Pi)_{\mathcal{O}})$
shows $\forall\, ops \in set\ (embed\ \pi).\ \forall\, op \in set\ ops.\ op \in set\ ((\Pi)_{\mathcal{O}})$
$\langle proof \rangle$

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

$$G \subseteq_m execute\text{-}serial\text{-}plan\ ((\Pi)_I)\ \pi \\ \quad \Longrightarrow G \subseteq_m execute\text{-}serial\text{-}plan\ ((\Pi)_I)\ (embed\ \pi)$$

using lemma ; and moreover by showing that

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

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

theorem *embedding-lemma*:
assumes *is-valid-problem-strips* Π
and *is-serial-solution-for-problem* $\Pi\ \pi$
shows *is-parallel-solution-for-problem* $\Pi\ (embed\ \pi)$
$\langle proof \rangle$

lemma *flattening-lemma-i*:
fixes Π:: *'a strips-problem*
assumes $\forall\, ops \in set\ \pi.\ \forall\, op \in set\ ops.\ op \in set\ ((\Pi)_{\mathcal{O}})$
shows $\forall\, op \in set\ (concat\ \pi).\ op \in set\ ((\Pi)_{\mathcal{O}})$
$\langle proof \rangle$

lemma *flattening-lemma-ii*:
fixes I :: *'variable strips-state*
assumes $\forall$ *ops* $\in$ *set* π. $\exists$ *op*. *ops* $= [op] \wedge$ *is-valid-operator-strips* Π *op*
and $G \subseteq_m$ *execute-parallel-plan* I π
shows $G \subseteq_m$ *execute-serial-plan* I (*concat* π)
⟨*proof*⟩

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* $\in$ *set* π. $\exists$ *op*. *ops* $= [op]$
and *is-parallel-solution-for-problem* Π π
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 $k <$ *length* π, 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

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

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

theorem *execute-parallel-plan-is-execute-sequential-plan-if*:
fixes I :: (*'variable*, *bool*) *state*
assumes *is-valid-problem* Π
and $G \subseteq_m$ *last* (*trace-parallel-plan-strips* I π)
and $\forall k <$ *length* π.
are-all-operators-applicable (*trace-parallel-plan-strips* I π ! k) (π ! k)
$\wedge$ *are-all-operator-effects-consistent* (π ! k)
$\wedge$ *are-all-operators-non-interfering* (π ! k)
shows $G \subseteq_m$ *execute-serial-plan* I (*concat* π)
⟨*proof*⟩

3.4 Auxiliary lemmas about STRIPS

lemma *set-to-precondition-of-op-is*[*simp*]: *set* (*to-precondition op*)

= { (*v*, *True*) | *v*. *v* ∈ *set* (*precondition-of op*) }
⟨*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* ((-$_{\mathcal{V}+}$) [*1000*] *999*)
operators-of :: (*'variable*, *'domain*) *sas-plus-operator list* ((-$_{\mathcal{O}+}$) [*1000*] *999*)
initial-of :: (*'variable*, *'domain*) *state* ((-$_{I+}$) [*1000*] *999*)
goal-of :: (*'variable*, *'domain*) *state* ((-$_{G+}$) [*1000*] *999*)
range-of :: *'variable* ⇀ *'domain list*

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

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

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

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

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

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

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

definition *is-valid-operator-sas-plus*
:: ('variable, 'domain) sas-plus-problem ⇒ ('variable, 'domain) sas-plus-operator ⇒ bool
where *is-valid-operator-sas-plus* Ψ *op* ≡ *let*
pre = *precondition-of op*
; *eff* = *effect-of op*
; *vs* = *variables-of* Ψ
; *D* = *range-of* Ψ
in list-all (λ(*v*, *a*). *ListMem v vs*) *pre*
∧ *list-all* (λ(*v*, *a*). (*D v* ≠ *None*) ∧ *ListMem a* (*the* (*D v*))) *pre*
∧ *list-all* (λ(*v*, *a*). *ListMem v vs*) *eff*
∧ *list-all* (λ(*v*, *a*). (*D v* ≠ *None*) ∧ *ListMem a* (*the* (*D v*))) *eff*
∧ *list-all* (λ(*v*, *a*). *list-all* (λ(*v*′, *a*′). *v* ≠ *v*′ ∨ *a* = *a*′) *pre*) *pre*
∧ *list-all* (λ(*v*, *a*). *list-all* (λ(*v*′, *a*′). *v* ≠ *v*′ ∨ *a* = *a*′) *eff*) *eff*

definition *is-valid-problem-sas-plus* Ψ
≡ *let ops* = *operators-of* Ψ
; *vs* = *variables-of* Ψ
; *I* = *initial-of* Ψ
; *G* = *goal-of* Ψ
; *D* = *range-of* Ψ
in list-all (λ*v*. *D v* ≠ *None*) *vs*
∧ *list-all* (*is-valid-operator-sas-plus* Ψ) *ops*
∧ (∀ *v*. *I v* ≠ *None* ⟷ *ListMem v vs*)
∧ (∀ *v*. *I v* ≠ *None* ⟶ *ListMem* (*the* (*I v*)) (*the* (*D v*)))
∧ (∀ *v*. *G v* ≠ *None* ⟶ *ListMem v* (*variables-of* Ψ))
∧ (∀ *v*. *G v* ≠ *None* ⟶ *ListMem* (*the* (*G v*)) (*the* (*D v*)))

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

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

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

automatically substitute definitions).
lemma[*simp*]:
  $is\text{-}operator\text{-}applicable\text{-}in\ s\ op = (map\text{-}of\ (precondition\text{-}of\ op) \subseteq_m s)$
  $s \gg_+ op = s ++ map\text{-}of\ (effect\text{-}of\ op)$
  ⟨*proof*⟩

lemma *range-of-not-empty*:
  $(sas\text{-}plus\text{-}problem.range\text{-}of\ \Psi\ v \neq None \wedge sas\text{-}plus\text{-}problem.range\text{-}of\ \Psi\ v \neq Some\ [])$
    $\longleftrightarrow (\mathcal{R}_+\ \Psi\ v) \neq \{\}$
  ⟨*proof*⟩

lemma *is-valid-operator-sas-plus-then*:
  fixes Ψ::('v,'d) *sas-plus-problem*
  assumes *is-valid-operator-sas-plus* Ψ *op*
  shows $\forall (v, a) \in set\ (precondition\text{-}of\ op).\ v \in set\ ((\Psi)_{\mathcal{V}+})$
    and $\forall (v, a) \in set\ (precondition\text{-}of\ op).\ (\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
    and $\forall (v, a) \in set\ (effect\text{-}of\ op).\ v \in set\ ((\Psi)_{\mathcal{V}+})$
    and $\forall (v, a) \in set\ (effect\text{-}of\ op).\ (\mathcal{R}_+\ \Psi\ v) \neq \{\} \wedge a \in \mathcal{R}_+\ \Psi\ v$
    and $\forall (v, a) \in set\ (precondition\text{-}of\ op).\ \forall (v', a') \in set\ (precondition\text{-}of\ op).\ v \neq v' \vee a = a'$
    and $\forall (v, a) \in set\ (effect\text{-}of\ op).$
      $\forall (v', a') \in set\ (effect\text{-}of\ op).\ v \neq v' \vee a = a'$
⟨*proof*⟩

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

end

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

5 SAS+ Semantics

5.1 Serial Execution Semantics

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

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

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

definition *is-serial-solution-for-problem*
  :: ('*variable*, '*domain*) *sas-plus-problem* ⇒ ('*variable*, '*domain*) *sas-plus-plan* ⇒ *bool*
  where *is-serial-solution-for-problem* Ψ ψ
    ≡ *let*
        I = *sas-plus-problem.initial-of* Ψ
        ; *G* = *sas-plus-problem.goal-of* Ψ
        ; *ops* = *sas-plus-problem.operators-of* Ψ
      in G $\subseteq_m$ *execute-serial-plan-sas-plus I* ψ
        ∧ *list-all* (λ*op. ListMem op ops*) ψ

context
begin

private lemma *execute-operator-sas-plus-effect-i*:
  assumes *is-operator-applicable-in s op*
    and ∀ (*v*, *a*) ∈ *set* (*effect-of op*). ∀ (*v*′, *a*′) ∈ *set* (*effect-of op*).
      v ≠ *v*′ ∨ *a* = *a*′
    and(*v*, *a*) ∈ *set* (*effect-of op*)
  shows (*s* $\gg_+$ *op*) *v* = *Some a*
⟨*proof*⟩ **lemma** *execute-operator-sas-plus-effect-ii*:
  assumes *is-operator-applicable-in s op*
    and ∀ (*v*′, *a*′) ∈ *set* (*effect-of op*). *v*′ ≠ *v*
  shows (*s* $\gg_+$ *op*) *v* = *s v*
⟨*proof*⟩

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*′ ≡ *s* $\gg_+$ *op* has the following properties:

- *s*′ *v* = *Some a* if (*v*, *a*) exist in *set* (*effect-of op*); and,

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

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

theorem *execute-operator-sas-plus-effect*:
  assumes *is-operator-applicable-in s op*
    and $\forall (v,\ a) \in$ *set* (*effect-of op*).
      $\forall (v',\ a') \in$ *set* (*effect-of op*). $v \neq v' \lor a = a'$
  shows $(v,\ a) \in$ *set* (*effect-of op*)
      $\longrightarrow (s \gg_+ op)\ v = Some\ a$
    and ($\forall a.\ (v,\ a) \notin$ *set* (*effect-of op*))
      $\longrightarrow (s \gg_+ op)\ v = s\ v$
⟨*proof*⟩

end

5.2 Parallel Execution Semantics

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

type-synonym (*'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 ($\lambda(v,\ a)$. *list-all* ($\lambda(v',\ a')$. $v \neq v' \lor a = a'$) *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*

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

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

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

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

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

1. starting from the initial state Ψ, SAS+ parallel plan execution reaches a state which satisfies the described goal state Ψ_{G+}; and,
2. all parallel operators *ops* in the plan ψ only consist of operators that are specified in the problem description.

definition *is-parallel-solution-for-problem*
:: ('*variable*, '*domain*) *sas-plus-problem*
$\Rightarrow$ ('*variable*, '*domain*) *sas-plus-parallel-plan*
$\Rightarrow$ *bool*
where *is-parallel-solution-for-problem* Ψ ψ
$\equiv$ *let*
G = *sas-plus-problem.goal-of* Ψ
; I = *sas-plus-problem.initial-of* Ψ

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

context
begin

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

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

lemma *trace-parallel-plan-sas-plus-head-is-initial-state*:
trace-parallel-plan-sas-plus I ψ ! *0* = *I*
⟨*proof*⟩

lemma *trace-parallel-plan-sas-plus-length-gt-one-if*:
assumes $k <$ *length* (*trace-parallel-plan-sas-plus I* ψ) − *1*
shows *1* < *length* (*trace-parallel-plan-sas-plus I* ψ)
⟨*proof*⟩

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

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

lemma *trace-parallel-plan-sas-plus-step-implies-operator-execution-condition-holds*:
assumes $k <$ *length* (*trace-parallel-plan-sas-plus I* π) − *1*
shows *are-all-operators-applicable-in* (*trace-parallel-plan-sas-plus I* π ! k) (π ! k)

$\wedge$ *are-all-operator-effects-consistent* $(\pi\ !\ k)$
⟨*proof*⟩

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

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

⟨*proof*⟩

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

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

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

fixes Ψ :: $('v, 'd)$ *sas-plus-problem*
assumes *is-parallel-solution-for-problem* Ψ ψ
shows $\forall$ *ops* $\in$ *set* ψ. $\forall$ *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 ∀ *op* ∈ *set* ψ. *are-operator-effects-consistent op op*
and $G \subseteq_m$ *execute-serial-plan-sas-plus I* ψ
shows $G \subseteq_m$ *execute-parallel-plan-sas-plus I* (*embed* ψ)
⟨*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* Ψ ψ
and *op* ∈ *set* ψ
shows *are-operator-effects-consistent op op*
⟨*proof*⟩

lemma *execute-serial-plan-sas-plus-is-execute-parallel-plan-sas-plus-iv*:
fixes Ψ :: (*'v*, *'d*) *sas-plus-problem*
assumes ∀ *op* ∈ *set* ψ. *op* ∈ *set* $((\Psi)_{\mathcal{O}+})$
shows ∀ *ops* ∈ *set* (*embed* ψ). ∀ *op* ∈ *set ops*. *op* ∈ *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* Ψ ψ
shows *is-parallel-solution-for-problem* Ψ (*embed* ψ)
⟨*proof*⟩

lemma *flattening-lemma-i*:
fixes Ψ :: (*'v*, *'d*) *sas-plus-problem*
assumes ∀ *ops* ∈ *set* π. ∀ *op* ∈ *set ops*. *op* ∈ *set* $((\Psi)_{\mathcal{O}+})$
shows ∀ *op* ∈ *set* (*concat* π). *op* ∈ *set* $((\Psi)_{\mathcal{O}+})$
⟨*proof*⟩

lemma *flattening-lemma-ii*:
fixes *I* :: (*'variable*, *'domain*) *state*
assumes ∀ *ops* ∈ *set* ψ. ∃ *op*. *ops* = [*op*] ∧ *is-valid-operator-sas-plus* Ψ *op*
and $G \subseteq_m$ *execute-parallel-plan-sas-plus I* ψ
shows $G \subseteq_m$ *execute-serial-plan-sas-plus I* (*concat* ψ)
⟨*proof*⟩

lemma *flattening-lemma*:
assumes *is-valid-problem-sas-plus* Ψ

and ∀ *ops* ∈ *set* ψ. ∃ *op*. *ops* = [*op*]
and *is-parallel-solution-for-problem* Ψ ψ
shows *is-serial-solution-for-problem* Ψ (*concat* ψ)
⟨*proof*⟩
end

5.4 Auxiliary lemmata on SAS+

context
begin

— Relate the locale definition *range-of* with its corresponding implementation for valid operators and given an effect (v, a).
lemma *is-valid-operator-sas-plus-then-range-of-sas-plus-op-is-set-range-of-op*:
assumes *is-valid-operator-sas-plus* Ψ *op*
and $(v, a) \in$ *set* (*precondition-of op*) ∨ $(v, a) \in$ *set* (*effect-of op*)
shows $(\mathcal{R}_+ \ \Psi \ v)$ = *set* (*the* (*sas-plus-problem.range-of* Ψ v))
⟨*proof*⟩

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

lemma *sublocale-sas-plus-finite-domain-representation-ii*:
fixes Ψ::$('v,'d)$ *sas-plus-problem*
assumes *is-valid-problem-sas-plus* Ψ
shows ∀ $v \in$ *set* $((\Psi)_{\mathcal{V}+})$. $(\mathcal{R}_+ \ \Psi \ v) \neq \{\}$
and ∀ *op* ∈ *set* $((\Psi)_{\mathcal{O}+})$. *is-valid-operator-sas-plus* Ψ *op*
and *dom* $((\Psi)_{I+})$ = *set* $((\Psi)_{\mathcal{V}+})$
and ∀ $v \in$ *dom* $((\Psi)_{I+})$. *the* $(((\Psi)_{I+}) \ v) \in \mathcal{R}_+ \ \Psi \ v$
and *dom* $((\Psi)_{G+}) \subseteq$ *set* $((\Psi)_{\mathcal{V}+})$
and ∀ $v \in$ *dom* $((\Psi)_{G+})$. *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 ⇒ 'variable ⇒ ('variable × 'domain) list

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

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

definition state-to-strips-state
  :: ('variable, 'domain) sas-plus-problem
    ⇒ ('variable, 'domain) state
    ⇒ ('variable, 'domain) assignment strips-state
  (φ_S - - 99)
  where state-to-strips-state Ψ s
    ≡ let defined = filter (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]))

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

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

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

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

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

definition *strips-op-to-sasp*
:: ('*variable*, '*domain*) *sas-plus-problem*
⇒ ('*variable* × '*domain*) *strips-operator*
⇒ ('*variable*, '*domain*) *sas-plus-operator*
(${\varphi_O}^{-1}$ - - *99*)
where *strips-op-to-sasp* Ψ *op*
≡ *let*
precondition = *strips-operator.precondition-of op*
; *effect* = *strips-operator.add-effects-of op*
in ⦇ *precondition-of* = *precondition*, *effect-of* = *effect* ⦈

definition *strips-parallel-plan-to-sas-plus-parallel-plan*
:: ('*variable*, '*domain*) *sas-plus-problem*
⇒ ('*variable* × '*domain*) *strips-parallel-plan*
⇒ ('*variable*, '*domain*) *sas-plus-parallel-plan*
(${\varphi_P}^{-1}$ - - *99*)
where *strips-parallel-plan-to-sas-plus-parallel-plan* Π π
≡ [[*strips-op-to-sasp* Π *op*. *op* ← *ops*]. *ops* ← π]

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

[6] We append a suffix identifying the respective formalism to the the parameter names passed to the parameter names in the locale. This is necessary to avoid ambiguous names in the sublocale declarations. For example, without addition of suffixes the type for

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

context
begin

— Set-up simp rules.
lemma[*simp*]:
$(\varphi\ \Psi) = ($*let*
vs $=$ [*as*. $v \leftarrow$ *variables-of* Ψ, *as* $\leftarrow$ (*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 $(\varphi_S\ \Psi\ s)$
$=$ (*let defined* $=$ *filter* ($\lambda v.\ s\ v \neq$ *None*) (*variables-of* Ψ) *in*
map-of (*map* ($\lambda(v, a)$. ((v, a), *the* (s v) $= a$))
(*concat* [*possible-assignments-for* Ψ v. $v \leftarrow$ *defined*])))
and $(\varphi_O\ \Psi$ *op*$)$
$=$ (*let*
pre $=$ *precondition-of op*
; *add* $=$ *effect-of op*
; *delete* $=$ [(v, a'). (v, a) $\leftarrow$ *effect-of op*, $a' \leftarrow$ *filter* (($\neq$) a) (*the* (*range-of* Ψ
v))]
in STRIPS-Representation.operator-for pre add delete)
and $(\varphi_P\ \Psi\ \psi) = [[\varphi_O\ \Psi$ *op*. *op* $\leftarrow$ *ops*]. *ops* $\leftarrow \psi]$
and $({\varphi_S}^{-1}\ \Psi\ s')=$ *map-of* (*filter* ($\lambda(v, a)$. s' (v, a) $=$ *Some True*)
(*all-possible-assignments-for* Ψ))
and $({\varphi_O}^{-1}\ \Psi$ *op*$') = ($*let*
precondition $=$ *strips-operator.precondition-of op*$'$
; *effect* $=$ *strips-operator.add-effects-of op*$'$
in ⦇ *precondition-of* $=$ *precondition*, *effect-of* $=$ *effect* ⦈)
and $({\varphi_P}^{-1}\ \Psi\ \pi) = [[{\varphi_O}^{-1}\ \Psi$ *op*. *op* $\leftarrow$ *ops*]. *ops* $\leftarrow \pi]$
⟨*proof*⟩

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

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

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

---

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

$\langle proof \rangle$

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

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

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 $dom\ (\varphi_S\ \ \Psi\ \ s)$
 $= (\bigcup v \in \{\ v\ \mid\ v.\ v \in set\ ((\Psi)_{\mathcal{V}+}) \wedge s\ v \neq None\ \}.$
 $\{\ (v,\ a)\ \mid\ a.\ a \in \mathcal{R}_+\ \ \Psi\ \ v\ \})$
$\langle proof \rangle$

corollary *state-to-strips-state-dom-element-iff*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows $(v,\ a) \in dom\ (\varphi_S\ \ \Psi\ \ s) \longleftrightarrow v \in set\ ((\Psi)_{\mathcal{V}+})$
 $\wedge\ s\ v \neq None$
 $\wedge\ a \in \mathcal{R}_+\ \ \Psi\ \ v$
$\langle proof \rangle$

lemma *state-to-strips-state-range-is*:
 assumes *is-valid-problem-sas-plus* Ψ
 and $(v,\ a) \in dom\ (\varphi_S\ \ \Psi\ \ s)$
 shows $(\varphi_S\ \ \Psi\ \ s)\ (v,\ a) = Some\ (the\ (s\ v) = a)$
$\langle proof \rangle$
lemma *state-to-strips-state-effect-consistent*:
 assumes *is-valid-problem-sas-plus* Ψ
 and $(v,\ a) \in dom\ (\varphi_S\ \ \Psi\ \ s)$
 and $(v,\ a') \in dom\ (\varphi_S\ \ \Psi\ \ s)$
 and $(\varphi_S\ \ \Psi\ \ s)\ (v,\ a) = Some\ True$
 and $(\varphi_S\ \ \Psi\ \ s)\ (v,\ a') = Some\ True$
 shows $(v,\ a) = (v,\ a')$
$\langle proof \rangle$

lemma *sasp-op-to-strips-set-delete-effects-is*:
 assumes *is-valid-operator-sas-plus* Ψ *op*

shows *set* (*strips-operator.delete-effects-of* (φ_O Ψ *op*))
  = ($\bigcup$(*v*, *a*) ∈ *set* (*effect-of op*). { (*v*, *a*′) | *a*′. *a*′ ∈ ($\mathcal{R}_+$ Ψ *v*) ∧ *a*′ ≠ *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 *set* ((φ Ψ)$_{\mathcal{V}}$) = ($\bigcup$ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$). { (*v*, *a*) | *a*. *a* ∈ $\mathcal{R}_+$ Ψ *v* })
⟨*proof*⟩

corollary *sas-plus-problem-to-strips-problem-variable-set-element-iff*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows (*v*, *a*) ∈ *set* ((φ Ψ)$_{\mathcal{V}}$) ⟷ *v* ∈ *set* ((Ψ)$_{\mathcal{V}+}$) ∧ *a* ∈ $\mathcal{R}_+$ Ψ *v*
 ⟨*proof*⟩

lemma *sasp-op-to-strips-effect-consistent*:
 assumes *op* = φ_O Ψ *op*′
  and *op*′ ∈ *set* ((Ψ)$_{\mathcal{O}+}$)
  and *is-valid-operator-sas-plus* Ψ *op*′
 shows (*v*, *a*) ∈ *set* (*add-effects-of op*) ⟶ (*v*, *a*) ∉ *set* (*delete-effects-of op*)
  and (*v*, *a*) ∈ *set* (*delete-effects-of op*) ⟶ (*v*, *a*) ∉ *set* (*add-effects-of op*)
⟨*proof*⟩

lemma *is-valid-problem-sas-plus-then-strips-transformation-too-iii*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows *list-all* (*is-valid-operator-strips* (φ Ψ))
  (*strips-problem.operators-of* (φ Ψ))
⟨*proof*⟩

lemma *is-valid-problem-sas-plus-then-strips-transformation-too-iv*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows ∀ *x*. ((φ Ψ)$_I$) *x* ≠ *None*
  ⟷ *ListMem x* (*strips-problem.variables-of* (φ Ψ))
⟨*proof*⟩ **lemma** *is-valid-problem-sas-plus-then-strips-transformation-too-v*:
 assumes *is-valid-problem-sas-plus* Ψ
 shows ∀ *x*. ((φ Ψ)$_G$) *x* ≠ *None*
  ⟶ *ListMem x* (*strips-problem.variables-of* (φ Ψ))
⟨*proof*⟩

We now show that given Ψ is a valid SASPlus problem, then Π ≡ φ Ψ is a valid STRIPS problem as well. The proof unfolds the definition of *is-valid-problem-strips* and then shows each of the conjuncts for Π. These are:

- Π has at least one variable;
- Π has at least one operator;
- all operators are valid STRIPS operators;

- Π_I is defined for all variables in $\Pi_{\mathcal{V}}$; and finally,

- if (Π_G) x is defined, then x is in $\Pi_{\mathcal{V}}$.

theorem
is-valid-problem-sas-plus-then-strips-transformation-too:
assumes *is-valid-problem-sas-plus* Ψ
shows *is-valid-problem-strips* $(\varphi\ \Psi)$
$\langle proof \rangle$

lemma *set-filter-all-possible-assignments-true-is*:
assumes *is-valid-problem-sas-plus* Ψ
shows $set\ (filter\ (\lambda(v,\ a).\ s\ (v,\ a) = Some\ True)$
$(all\text{-}possible\text{-}assignments\text{-}for\ \Psi))$
$= (\bigcup v \in set\ ((\Psi)_{\mathcal{V}+}).\ Pair\ v\ \text{‘}\ \{\ a \in \mathcal{R}_+\ \Psi\ v.\ s\ (v,\ a) = Some\ True\ \})$
$\langle proof \rangle$

lemma *strips-state-to-state-dom-is*:
assumes *is-valid-problem-sas-plus* Ψ
shows $dom\ ({\varphi_S}^{-1}\ \Psi\ s)$
$= (\bigcup v \in set\ ((\Psi)_{\mathcal{V}+}).$
$\{\ v \mid a.\ a \in (\mathcal{R}_+\ \Psi\ v) \wedge s\ (v,\ a) = Some\ True\ \})$
$\langle proof \rangle$

lemma *strips-state-to-state-range-is*:
assumes *is-valid-problem-sas-plus* Ψ
and $v \in set\ ((\Psi)_{\mathcal{V}+})$
and $a \in \mathcal{R}_+\ \Psi\ v$
and $(v,\ a) \in dom\ s'$
and $\forall (v,\ a) \in dom\ s'.\ \forall (v,\ a') \in dom\ s'.\ s'\ (v,\ a) = Some\ True \wedge s'\ (v,\ a') = Some\ True$
$\longrightarrow (v,\ a) = (v,\ a')$
shows $({\varphi_S}^{-1}\ \Psi\ s')\ v = Some\ a \longleftrightarrow the\ (s'\ (v,\ a))$
$\langle proof \rangle$
lemma *strips-state-to-state-inverse-is-i*:
assumes *is-valid-problem-sas-plus* Ψ
and $v \in set\ ((\Psi)_{\mathcal{V}+})$
and $s\ v \neq None$
and $a \in \mathcal{R}_+\ \Psi\ v$
shows $(\varphi_S\ \Psi\ s)\ (v,\ a) = Some\ (the\ (s\ v) = a)$
$\langle proof \rangle$

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

⟨*proof*⟩

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

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

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

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

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

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

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 $\forall\, v \in dom\ s.\ the\ (s\ v) \in \mathcal{R}_+\ \Psi\ v$
and *set ops'* $\subseteq$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *are-all-operators-applicable-in s* $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops'] \wedge$
are-all-operator-effects-consistent $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
shows *STRIPS-Semantics.are-all-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 $\forall\, v \in$ *dom s*. *the* $(s\ v) \in \mathcal{R}_+\ \Psi\ v$
and *set ops'* $\subseteq$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *are-all-operators-applicable-in s* $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops'] \wedge$
are-all-operator-effects-consistent $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
shows *STRIPS-Semantics.are-all-operator-effects-consistent ops'*
⟨*proof*⟩

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 $\forall\, v \in$ *dom s*. *the* $(s\ v) \in \mathcal{R}_+\ \Psi\ v$
and *set ops'* $\subseteq$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *are-all-operators-applicable-in s* $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops'] \wedge$
are-all-operator-effects-consistent $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$
shows *STRIPS-Semantics.are-all-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 $\forall\, op \in$ *set ops*. $\forall\,(v,\ a) \in$ *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 *dom s* $\subseteq V$
and $V \subseteq$ *dom D*
and $\forall\, v \in$ *dom s*. *the* $(s\ v) \in$ *set* (*the* $(D\ v)$)
and $\forall\, op \in$ *set ops*. $\forall\,(v,\ a) \in$ *set* (*effect-of op*). $v \in V \wedge a \in$ *set* (*the* $(D\ v)$)
shows $\forall\, v \in$ *dom* (*execute-parallel-operator-sas-plus s ops*).
the (*execute-parallel-operator-sas-plus s ops v*) $\in$ *set* (*the* $(D\ v)$)
⟨*proof*⟩

lemma *transfom-sas-plus-problem-to-strips-problem-operators-valid*:
assumes *is-valid-problem-sas-plus* Ψ
and $op' \in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
obtains *op*
where $op \in$ *set* $((\Psi)_{\mathcal{O}+})$
and $op' = (\varphi_O\ \Psi\ op)$ *is-valid-operator-sas-plus* Ψ *op*
⟨*proof*⟩

lemma *sas-plus-equivalent-to-strips-i-a-XI*:
assumes *is-valid-problem-sas-plus* Ψ
and $op' \in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
shows $(\varphi_S\ \Psi\ s)$ ++ *map-of* (*effect-to-assignments op'*)
$= \varphi_S\ \Psi\ (s$ ++ *map-of* (*effect-of* $(\varphi_O{}^{-1}\ \Psi\ op')$))
⟨*proof*⟩

lemma *sas-plus-equivalent-to-strips-i-a-XII*:
  assumes *is-valid-problem-sas-plus* Ψ
    and $\forall$ *op'* $\in$ *set ops'*. *op'* $\in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
  shows *execute-parallel-operator* $(\varphi_S\ \Psi\ s)$ *ops'*
    $= \varphi_S\ \Psi$ (*execute-parallel-operator-sas-plus s* $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$)
⟨*proof*⟩

lemma *sas-plus-equivalent-to-strips-i-a-XIII*:
  assumes *is-valid-problem-sas-plus* Ψ
    and $\forall$ *op'* $\in$ *set ops'*. *op'* $\in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
    and $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan*
      (*execute-parallel-operator* $(\varphi_S\ \Psi\ I)$ *ops'*) π
  shows $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan*
    $(\varphi_S\ \Psi$ (*execute-parallel-operator-sas-plus I* $[\varphi_O{}^{-1}\ \Psi\ op'.\ op' \leftarrow ops']$)) π
⟨*proof*⟩
lemma *sas-plus-equivalent-to-strips-i-a*:
  assumes *is-valid-problem-sas-plus* Ψ
    and *dom I* $\subseteq$ *set* $((\Psi)_{\mathcal{V}+})$
    and $\forall v \in$ *dom I*. *the* $(I\ v) \in \mathcal{R}_+\ \Psi\ v$
    and *dom G* $\subseteq$ *set* $((\Psi)_{\mathcal{V}+})$
    and $\forall v \in$ *dom G*. *the* $(G\ v) \in \mathcal{R}_+\ \Psi\ v$
    and $\forall$ *ops'* $\in$ *set* π. $\forall$ *op'* $\in$ *set ops'*. *op'* $\in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
    and $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan* $(\varphi_S\ \Psi\ I)\ \pi$
  shows $G \subseteq_m$ *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*
    (*sas-plus-problem.initial-of* Ψ) $(\varphi_P{}^{-1}\ \Psi\ \pi)$
⟨*proof*⟩
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* (λ*op*. *ListMem op* (*operators-of* Ψ))) $(\varphi_P{}^{-1}\ \Psi\ \pi)$
⟨*proof*⟩

We now show that for a parallel solution π of Π the SAS+ plan $\psi \equiv \varphi_P{}^{-1}$ $\Psi\ \pi$ yielded by the STRIPS to SAS+ plan transformation is a solution for Ψ. The proof uses the definition of parallel STRIPS solutions and shows that the execution of ψ on the initial state of the SAS+ problem yields a state satisfying the problem's goal state, i.e.

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

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

theorem
sas-plus-equivalent-to-strips:
assumes *is-valid-problem-sas-plus* Ψ
and *STRIPS-Semantics.is-parallel-solution-for-problem* $(\varphi\ \Psi)\ \pi$
shows *is-parallel-solution-for-problem* $\Psi\ ({\varphi_P}^{-1}\ \Psi\ \pi)$
⟨*proof*⟩ **lemma** *strips-equivalent-to-sas-plus-i-a-I*:
assumes *is-valid-problem-sas-plus* Ψ
and $\forall$ *op* $\in$ *set ops. op* $\in$ *set* $((\Psi)_{\mathcal{O}+})$
and *op'* $\in$ *set* $[\varphi_O\ \Psi$ *op. op* $\leftarrow$ *ops*]
obtains *op* **where** *op* $\in$ *set ops*
and *op'* $= \varphi_O\ \Psi$ *op*
⟨*proof*⟩ **corollary** *strips-equivalent-to-sas-plus-i-a-II*:
assumes*is-valid-problem-sas-plus* Ψ
and $\forall$ *op* $\in$ *set ops. op* $\in$ *set* $((\Psi)_{\mathcal{O}+})$
and *op'* $\in$ *set* $[\varphi_O\ \Psi$ *op. op* $\leftarrow$ *ops*]
shows *op'* $\in$ *set* $((\varphi\ \Psi)_{\mathcal{O}})$
and *is-valid-operator-strips* $(\varphi\ \Psi)$ *op'*
⟨*proof*⟩

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

lemma *strips-equivalent-to-sas-plus-i-a*:
assumes *is-valid-problem-sas-plus* Ψ
and *dom I* $\subseteq$ *set* $((\Psi)_{\mathcal{V}+})$
and $\forall$ *v* $\in$ *dom I. the* (*I v*) $\in \mathcal{R}_+\ \Psi\ v$
and *dom G* $\subseteq$ *set* $((\Psi)_{\mathcal{V}+})$

and $\forall\, v \in dom\ G.\ the\ (G\ v) \in \mathcal{R}_+\ \Psi\ v$
and $\forall\, ops \in set\ \psi.\ \forall\, op \in set\ ops.\ op \in set\ ((\Psi)_{\mathcal{O}+})$
and $G \subseteq_m$ *execute-parallel-plan-sas-plus* $I\ \psi$
shows $(\varphi_S\ \Psi\ G) \subseteq_m$ *execute-parallel-plan* $(\varphi_S\ \Psi\ I)\ (\varphi_P\ \Psi\ \psi)$
⟨*proof*⟩

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

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* (*list-all* (λ*op. ListMem op* (*strips-problem.operators-of* $(\varphi\ \Psi))))$
$(\varphi_P\ \Psi\ \psi)$
⟨*proof*⟩

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.

$G' \subseteq_m$ *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 $ops \in set\ (embed\ \psi)$
obtains *op*
where $op \in set\ \psi$
and $[\varphi_O\ \Psi\ op.\ op \leftarrow ops] = [\varphi_O\ \Psi\ op]$
⟨*proof*⟩ **lemma** *serial-sas-plus-equivalent-to-serial-strips-i*:
assumes $ops \in set\ (\varphi_P\ \Psi\ (embed\ \psi))$
obtains *op* **where** $op \in set\ \psi$ **and** $ops = [\varphi_O\ \Psi\ op]$
⟨*proof*⟩ **lemma** *serial-sas-plus-equivalent-to-serial-strips-ii*[*simp*]:
$concat\ (\varphi_P\ \Psi\ (embed\ \psi)) = [\varphi_O\ \Psi\ op.\ op \leftarrow \psi]$

⟨*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, $\pi \equiv \varphi_P\ \Psi$ *List-Supplement.embed* ψ is a parallel solution for the corresponding STRIPS problem $\varphi\ \Psi$. Moreover, since *List-Supplement.embed* ψ is a plan consisting of singleton parallel operators, the same is true for π. Hence, the flattening lemma applies and *concat* π is a serial solution for $\varphi\ \Psi$. Since *concat* moreover can be shown to be the inverse of *List-Supplement.embed*, the term

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

can be reduced to the intuitive form

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

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* $(\varphi\ \Psi)\ [\varphi_O\ \Psi\ \textit{op}.\ \textit{op} \leftarrow \psi]$
⟨*proof*⟩

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

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* $\Psi\ [{\varphi_O}^{-1}\ \Psi\ op.\ op \leftarrow \pi]$
$\langle proof \rangle$

6.2 Equivalence of SAS+ and STRIPS

— Define the sets of plans with upper length bound as well as the sets of solutions with upper length bound for SAS problems and induced STRIPS problems.
We keep this polymorphic by not specifying concrete types so it applies to both STRIPS and SAS+ plans.
abbreviation *bounded-plan-set*
where *bounded-plan-set ops k* $\equiv \{\ \pi.\ set\ \pi \subseteq set\ ops \wedge length\ \pi = k\ \}$

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

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

definition *bounded-solution-set-strips'*
:: *('variable × 'domain) strips-problem*
⇒ *nat*
⇒ *('variable × 'domain) strips-plan set*
where *bounded-solution-set-strips'* $\Pi\ k$
$\equiv \{\ \pi.\ STRIPS\text{-}Semantics.is\text{-}serial\text{-}solution\text{-}for\text{-}problem\ \Pi\ \pi \wedge length\ \pi = k\ \}$

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

— 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 $\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi]$ on the set of plans $P_k \equiv bounded\text{-}plan\text{-}set\ (operators\text{-}of\ \Psi)\ k$ with length bound k. The

injectiveness of $Sol_k \equiv$ *bounded-solution-set-sas-plus* Ψ k—the set of solutions with length bound k–then follows from the subset relation $Sol_k \subseteq P_k$.

lemma *sasp-op-to-strips-injective*:
assumes $(\varphi_O\ \Psi\ op_1) = (\varphi_O\ \Psi\ op_2)$
shows $op_1 = op_2$
⟨*proof*⟩

lemma *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-a*:
assumes *is-valid-problem-sas-plus* Ψ
shows *inj-on* $(\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi])$ (*bounded-plan-set* (*sas-plus-problem.operators-of* Ψ) 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* $(\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi])$ (*bounded-solution-set-sas-plus'* Ψ k)
⟨*proof*⟩ **lemma** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-i-c*:
assumes *is-valid-problem-sas-plus* Ψ
shows $(\lambda\psi.\ [\varphi_O\ \Psi\ op.\ op \leftarrow \psi])$ ‘ (*bounded-solution-set-sas-plus'* Ψ k)
= *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 *card* (*bounded-solution-set-sas-plus'* Ψ k) $\leq$ *card* (*bounded-solution-set-strips'* $(\varphi\ \Psi)$ k)
⟨*proof*⟩

lemma *bounded-plan-set-finite*:
shows *finite* $\{\ \pi.\ set\ \pi \subseteq set\ ops \land length\ \pi = k\ \}$
⟨*proof*⟩ **lemma** *sas-plus-formalism-and-induced-strips-formalism-are-equally-expressive-ii-a*:
assumes *is-valid-problem-sas-plus* Ψ
shows *finite* (*bounded-solution-set-sas-plus'* Ψ 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 disjunctiveness of the solution sets for either problem to be able to equivalently transform the set cardinality over the union of sets of solutions with bounded lengths into a sum over the cardinality of the sets of solutions with bounded length. Moreover, since we know that for each SAS+ solution with a given length an equivalent STRIPS solution exists in the solution set of the transformed problem with the same length, both sets must have the same cardinality.

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

STRIPS problem over all length up to a given upper bound N.

theorem
  assumes *is-valid-problem-sas-plus* Ψ
  shows *card* (*bounded-solution-set-sas-plus* Ψ N)
    = *card* (*bounded-solution-set-strips* (φ Ψ) N)
  ⟨*proof*⟩

end

end

theory *SAT-Plan-Base*
  imports *List−Index.List-Index*
    Propositional-Proof-Systems.Formulas
    STRIPS-Semantics
    Map-Supplement List-Supplement
    CNF-Semantics-Supplement CNF-Supplement
begin

— Hide constant and notation for ($\bot$) to prevent warnings.
hide-const (**open**) *Orderings.bot-class.bot*
no-notation *Orderings.bot-class.bot* ($\bot$)

— Hide constant and notation for ((-$^+$)) to prevent warnings.
hide-const (**open**) *Transitive-Closure.trancl*
no-notation *Transitive-Closure.trancl* ((-$^+$) [*1000*] *999*)

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

7 The Basic SATPlan Encoding

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

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

Let in the following $\Phi \equiv$ *encode-to-sat* Π t denote the SATPlan encoding for a STRIPS problem Π and makespan t. Let $k < t$ and $I \equiv (\Pi)_I$ be the initial state of Π, $G \equiv (\Pi)_G$ be its goal state, $\mathcal{V} \equiv (\Pi)_\mathcal{V}$ its variable set, and $\mathcal{O} \equiv (\Pi)_\mathcal{O}$ its operator set.

7.1 Encoding Function Definitions

Since the SATPlan encoding uses propositional variables for both operators and state variables of the problem as well as time points, we define a datatype using separate constructors —*State k n* for state variables resp. *Operator k n* for operator activation—to facilitate case distinction. The natural number values store the time index resp. the indexes of the variable or operator within their lists in the problem representation.

datatype *sat-plan-variable* =
  State nat nat
  | *Operator nat nat*

A SATPlan formula is a regular propositional formula over SATPlan variables. We add a type synonym to improve readability.

type-synonym *sat-plan-formula* = *sat-plan-variable formula*

We now continue with the concrete definitions used in the implementation of the SATPlan encoding. State variables are encoded as literals over SATPlan variables using the *State* constructor of .

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

The initial state encoding (definition **??**) is a conjunction of state variable encodings $A \equiv$ *encode-state-variable 0 n b* with $n \equiv$ *index vs v* and $b \equiv I\ v =$ *Some True* for all $v \in \mathcal{V}$. As we can see below, the same function but substituting the initial state with the goal state and zero with the makespan t produces the goal state encoding (**??**). Note that both functions construct a conjunction of clauses $A \vee \bot$ for which it is easy to show that we can normalize to conjunctive normal form (CNF).

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

definition *encode-goal-state*
  :: *'variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula* (Φ_G - *99*)
  where *encode-goal-state* Π *t*
    ≡ *let*
     vs = *variables-of* Π
     ; *G* = *goal-of* Π
    in ⋀(*map* (λ*v*. *encode-state-variable t* (*index vs v*) (*G v*) ∨ ⊥)

$(\mathit{filter}\ (\lambda v.\ G\ v \neq \mathit{None})\ \mathit{vs}))$

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

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

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

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

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

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

$$\mathit{Atom}\ (\mathit{Operator}\ k\ m) \longrightarrow \mathit{Atom}\ (\mathit{State}\ (\mathit{Suc}\ k)\ n)$$

for add effects respectively

$$\mathit{Atom}\ (\mathit{Operator}\ k\ m) \longrightarrow \neg \mathit{Atom}\ (\mathit{State}\ (\mathit{Suc}\ k)\ n)$$

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*

$\Rightarrow$ *sat-plan-variable formula*
where *encode-operator-effect* Π *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* (*Suc t*) (*index vs v*)))
(*add-effects-of op*)
@ *map* (λv.
$\neg$(*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*
$\Rightarrow$ *'variable strips-operator list*
$\Rightarrow$ *nat*
$\Rightarrow$ *sat-plan-variable formula*
where *encode-all-operator-effects* Π *ops t*
$\equiv$ *let l* = *List.product* [*0..<t*] *ops*
in foldr ($\wedge$) (*map* (λ(*t*, *op*). *encode-operator-effect* Π *t op*) *l*) ($\neg\bot$)

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

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

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

definition *encode-positive-transition-frame-axiom*
:: *'variable strips-problem*
$\Rightarrow$ *nat*
$\Rightarrow$ *'variable*

⇒ *sat-plan-variable formula*
where *encode-positive-transition-frame-axiom* Π *t v*
≡ *let vs* = *variables-of* Π
; *ops* = *operators-of* Π
; *adding-operators* = *filter* (λ*op. ListMem v* (*add-effects-of op*)) *ops*
in (*Atom* (*State t* (*index vs v*))
∨ (¬(*Atom* (*State* (*Suc t*) (*index vs v*)))
∨ ⋁(*map* (λ*op. Atom* (*Operator t* (*index ops op*))) *adding-operators*)))

definition *encode-all-frame-axioms*
:: *′variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula*
where *encode-all-frame-axioms* Π *t*
≡ *let l* = *List.product* [*0*..<*t*] (*variables-of* Π)
in ⋀(*map* (λ(*k*, *v*). *encode-negative-transition-frame-axiom* Π *k v*) *l*
@ *map* (λ(*k*, *v*). *encode-positive-transition-frame-axiom* Π *k v*) *l*)

Finally, the basic SATPlan encoding is the conjunction of the initial state, goal state, operator and frame axiom encoding for all time steps. The functions and [7] take care of mapping the operator precondition, effect and frame axiom encoding over all possible combinations of time point and operators resp. time points, variables, and operators.

definition *encode-problem* (Φ - - *99*)
where *encode-problem* Π *t*
≡ *encode-initial-state* Π
∧ (*encode-operators* Π *t*
∧ (*encode-all-frame-axioms* Π *t*
∧ (*encode-goal-state* Π *t*)))

7.2 Decoding Function Definitions

Decoding plans from a valuation $\mathcal{A}$ of a SATPlan encoding entails extracting all activated operators for all time points except the last one. We implement this by mapping over all $k < t$ and extracting activated operators—i.e. operators for which the model valuates the respective operator encoding at time k to true—into a parallel operator (see definition **??**). [8]

— Note that due to the implementation based on lists, we have to address the problem of duplicate operator declarations in the operator list of the problem. Since *index op* = *index op′* for equal operators, the parallel operator obtained from will contain duplicates in case the problem's operator list does. We therefore remove duplicates first using *remdups ops* and then filter out activated operators.
definition *decode-plan′*
:: *′variable strips-problem*
⇒ *sat-plan-variable valuation*
⇒ *nat*
⇒ *′variable strips-operator list*

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

where *decode-plan′* Π $\mathcal{A}$ *i*
  ≡ *let ops* = *operators-of* Π
    ; *vs* = *map* (λ*op*. *Operator i* (*index ops op*)) (*remdups ops*)
   in map (λ*v*. *case v of Operator* - *k* ⇒ *ops* ! *k*) (*filter* $\mathcal{A}$ *vs*)

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

definition *decode-plan*
 :: *′variable strips-problem*
  ⇒ *sat-plan-variable valuation*
  ⇒ *nat*
  ⇒ *′variable strips-parallel-plan* (Φ^{-1} - - - *99*)
 where *decode-plan* Π $\mathcal{A}$ *t* ≡ *map* (*decode-plan′* Π $\mathcal{A}$) [*0*..<*t*]

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

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

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

definition *encode-transitions* ::*′variable strips-problem* ⇒ *nat* ⇒ *sat-plan-variable formula* (Φ_T - - *99*) **where**
 encode-transitions Π *t*
   ≡ *SAT-Plan-Base.encode-operators* Π *t* ∧
    SAT-Plan-Base.encode-all-frame-axioms Π *t*

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

— Setup simp rules.
lemma [*simp*]:
 encode-transitions Π *t*
  = *SAT-Plan-Base.encode-operators* Π *t* ∧
   SAT-Plan-Base.encode-all-frame-axioms Π *t*
 ⟨*proof*⟩

context

begin

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

lemma *is-cnf-encode-initial-state*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (Φ_I Π)
⟨*proof*⟩

lemma *encode-goal-state-is-cnf*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (*encode-goal-state* Π *t*)
⟨*proof*⟩ **lemma** *encode-operator-precondition-is-cnf*:
is-cnf (*encode-operator-precondition* Π *k op*)
⟨*proof*⟩ **lemma** *set-map-operator-precondition*[*simp*]:
set (*map* (λ(*k*, *op*). *encode-operator-precondition* Π *k op*) (*List.product* [*0*..<*t*] *ops*))
= { *encode-operator-precondition* Π *k op* | *k op*. (*k*, *op*) ∈ ({*0*..<*t*} × *set ops*) }
⟨*proof*⟩ **lemma** *is-cnf-encode-all-operator-preconditions*:
is-cnf (*encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩ **lemma** *set-map-or*[*simp*]:
set (*map* (λ*v*. *A v* ∨ *B v*) *vs*) = { *A v* ∨ *B v* | *v*. *v* ∈ *set vs* }
⟨*proof*⟩ **lemma** *encode-operator-effects-is-cnf-i*:
is-cnf (⋀(*map* (λ*v*. (¬ (*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))))
∨ *Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))) (*add-effects-of op*)))
⟨*proof*⟩ **lemma** *encode-operator-effects-is-cnf-ii*:
is-cnf (⋀(*map* (λ*v*. ¬(*Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
∨ ¬(*Atom* (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*)))) (*delete-effects-of op*)))
⟨*proof*⟩ **lemma** *encode-operator-effect-is-cnf*:
shows *is-cnf* (*encode-operator-effect* Π *t op*)
⟨*proof*⟩ **lemma** *set-map-encode-operator-effect*[*simp*]:
set (*map* (λ(*t*, *op*). *encode-operator-effect* Π *t op*) (*List.product* [*0*..<*t*]
(*strips-problem.operators-of* Π)))
= { *encode-operator-effect* Π *k op*
| *k op*. (*k*, *op*) ∈ ({*0*..<*t*} × *set* (*strips-problem.operators-of* Π)) }
⟨*proof*⟩ **lemma** *encode-all-operator-effects-is-cnf*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩

lemma *encode-operators-is-cnf*:
  assumes *is-valid-problem-strips* Π
  shows *is-cnf* (*encode-operators* Π *t*)
  ⟨*proof*⟩ **lemma** *set-map-to-operator-atom*[*simp*]:
  set (*map* (λ*op*. *Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*)))
      (*filter* (λ*op*. *ListMem v vs*) (*strips-problem.operators-of* Π)))
    = { *Atom* (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))
      | *op*. *op* ∈ *set* (*strips-problem.operators-of* Π) ∧ *v* ∈ *set vs* }
⟨*proof*⟩

lemma *is-disj-big-or-if*:
  assumes ∀ *f* ∈ *set fs*. *is-lit-plus f*
  shows *is-disj* ⋁*fs*
  ⟨*proof*⟩

lemma *is-cnf-encode-negative-transition-frame-axiom*:
  shows *is-cnf* (*encode-negative-transition-frame-axiom* Π *t v*)
⟨*proof*⟩

lemma *is-cnf-encode-positive-transition-frame-axiom*:
  shows *is-cnf* (*encode-positive-transition-frame-axiom* Π *t v*)
⟨*proof*⟩ **lemma** *encode-all-frame-axioms-set*[*simp*]:
  set (*map* (λ(*k*, *v*). *encode-negative-transition-frame-axiom* Π *k v*)
        (*List.product* [*0*..<*t*] (*strips-problem.variables-of* Π))
      @ (*map* (λ(*k*, *v*). *encode-positive-transition-frame-axiom* Π *k v*)
        (*List.product* [*0*..<*t*] (*strips-problem.variables-of* Π))))
    = { *encode-negative-transition-frame-axiom* Π *k v*
        | *k v*. (*k*, *v*) ∈ ({*0*..<*t*} × *set* (*strips-problem.variables-of* Π)) }
      ∪ { *encode-positive-transition-frame-axiom* Π *k v*
        | *k v*. (*k*, *v*) ∈ ({*0*..<*t*} × *set* (*strips-problem.variables-of* Π)) }
⟨*proof*⟩

lemma *encode-frame-axioms-is-cnf*:
  shows *is-cnf* (*encode-all-frame-axioms* Π *t*)
⟨*proof*⟩

lemma *is-cnf-encode-problem*:
  assumes *is-valid-problem-strips* Π
  shows *is-cnf* (Φ Π *t*)
⟨*proof*⟩

lemma *encode-problem-has-model-then-also-partial-encodings*:
  assumes 𝒜 ⊨ *SAT-Plan-Base.encode-problem* Π *t*
  shows 𝒜 ⊨ *SAT-Plan-Base.encode-initial-state* Π
    and 𝒜 ⊨ *SAT-Plan-Base.encode-goal-state* Π *t*
    and 𝒜 ⊨ *SAT-Plan-Base.encode-operators* Π *t*
    and 𝒜 ⊨ *SAT-Plan-Base.encode-all-frame-axioms* Π *t*

⟨*proof*⟩

lemma *cnf-of-encode-problem-structure*:
shows *cnf* (*SAT-Plan-Base.encode-initial-state* Π)
⊆ *cnf* (*SAT-Plan-Base.encode-problem* Π *t*)
and *cnf* (*SAT-Plan-Base.encode-goal-state* Π *t*)
⊆ *cnf* (*SAT-Plan-Base.encode-problem* Π *t*)
and *cnf* (*SAT-Plan-Base.encode-operators* Π *t*)
⊆ *cnf* (*SAT-Plan-Base.encode-problem* Π *t*)
and *cnf* (*SAT-Plan-Base.encode-all-frame-axioms* Π *t*)
⊆ *cnf* (*SAT-Plan-Base.encode-problem* Π *t*)
⟨*proof*⟩ **lemma** *cnf-of-encode-initial-state-set-i*:
shows *cnf* (Φ_I Π) = ⋃ { *cnf* (*encode-state-variable 0*
(*index* (*strips-problem.variables-of* Π) *v*) ((($\Pi)_I$) *v*))
| *v*. *v* ∈ *set* (*strips-problem.variables-of* Π) ∧ (($\Pi)_I$) *v* ≠ *None* }
⟨*proof*⟩

corollary *cnf-of-encode-initial-state-set-ii*:
assumes *is-valid-problem-strips* Π
shows *cnf* (Φ_I Π) = (⋃*v* ∈ *set* (*strips-problem.variables-of* Π). {{
literal-formula-to-literal (*encode-state-variable 0* (*index* (*strips-problem.variables-of* Π) *v*)
(*strips-problem.initial-of* Π *v*)) }})
⟨*proof*⟩

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

lemma *cnf-of-operator-encoding-structure*:
cnf (*encode-operators* Π *t*) = *cnf* (*encode-all-operator-preconditions* Π
(*strips-problem.operators-of* Π) *t*)
∪ *cnf* (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩

corollary *cnf-of-operator-precondition-encoding-subset-encoding*:
cnf (*encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*)
⊆ *cnf* (Φ Π *t*)
⟨*proof*⟩

lemma *cnf-foldr-and*[*simp*]:
cnf (*foldr* (∧) *fs* (¬⊥)) = (⋃*f* ∈ *set fs*. *cnf f*)

⟨*proof*⟩ **lemma** *cnf-of-encode-operator-precondition*[*simp*]:
cnf (*encode-operator-precondition* Π *t op*) = (⋃ *v* ∈ *set* (*precondition-of op*).
{{(*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State t* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$}})
⟨*proof*⟩

lemma *cnf-of-encode-all-operator-preconditions-structure*[*simp*]:
cnf (*encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*)
= (⋃ (*t*, *op*) ∈ ({..<*t*} × *set* (*operators-of* Π)).
(⋃ *v* ∈ *set* (*precondition-of op*).
{{(*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State t* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$}}))
⟨*proof*⟩

corollary *cnf-of-encode-all-operator-preconditions-contains-clause-if*:
fixes Π::*'variable STRIPS-Representation.strips-problem*
assumes *is-valid-problem-strips* (Π::*'variable STRIPS-Representation.strips-problem*)
and $k < t$
and *op* ∈ *set* ((Π)$_{\mathcal{O}}$)
and *v* ∈ *set* (*precondition-of op*)
shows { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
∈ *cnf* (*encode-all-operator-preconditions* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩

corollary *cnf-of-encode-all-operator-effects-subset-cnf-of-encode-problem*:
cnf (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⊆ *cnf* (Φ Π *t*)
⟨*proof*⟩ **lemma** *cnf-of-encode-operator-effect-structure*[*simp*]:
cnf (*encode-operator-effect* Π *t op*)
= (⋃ *v* ∈ *set* (*add-effects-of op*). {{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }})
∪ (⋃ *v* ∈ *set* (*delete-effects-of op*).
{{ (*Operator t* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc t*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }})
⟨*proof*⟩

lemma *cnf-of-encode-all-operator-effects-structure*:
cnf (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
= (⋃ (*k*, *op*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{O}}$)).
(⋃ *v* ∈ *set* (*add-effects-of op*).
{{ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }}))
∪ (⋃ (*k*, *op*) ∈ ({*0*..<*t*} × *set* ((Π)$_{\mathcal{O}}$)).
(⋃ *v* ∈ *set* (*delete-effects-of op*).
{{ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }}))

⟨*proof*⟩

corollary *cnf-of-operator-effect-encoding-contains-add-effect-clause-if*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and $k < t$
and $op \in set ((\Pi)_{\mathcal{O}})$
and $v \in set$ (*add-effects-of op*)
shows { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
∈ *cnf* (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩

corollary *cnf-of-operator-effect-encoding-contains-delete-effect-clause-if*:
fixes Π:: *'a strips-problem*
assumes *is-valid-problem-strips* Π
and $k < t$
and $op \in set ((\Pi)_{\mathcal{O}})$
and $v \in set$ (*delete-effects-of op*)
shows { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }
∈ *cnf* (*encode-all-operator-effects* Π (*strips-problem.operators-of* Π) *t*)
⟨*proof*⟩ **lemma** *cnf-of-big-or-of-literal-formulas-is*[*simp*]:
assumes ∀ *f* ∈ *set fs. is-literal-formula f*
shows *cnf* (⋁*fs*) = {{ *literal-formula-to-literal f* | *f*. *f* ∈ *set fs* }}
⟨*proof*⟩ **lemma** *set-filter-op-list-mem-vs*[*simp*]:
set (*filter* (λ*op. ListMem v vs*) *ops*) = { *op. op* ∈ *set ops* ∧ *v* ∈ *set vs* }
⟨*proof*⟩ **lemma** *cnf-of-positive-transition-frame-axiom*:
cnf (*encode-positive-transition-frame-axiom* Π *k v*)
= {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }
∪ { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{+}$
| *op. op* ∈ *set* (*strips-problem.operators-of* Π) ∧ *v* ∈ *set* (*add-effects-of op*)
}}
⟨*proof*⟩ **lemma** *cnf-of-negative-transition-frame-axiom*:
cnf (*encode-negative-transition-frame-axiom* Π *k v*)
= {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }
∪ { (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{+}$
| *op. op* ∈ *set* (*strips-problem.operators-of* Π) ∧ *v* ∈ *set* (*delete-effects-of op*)
}}
⟨*proof*⟩

lemma *cnf-of-encode-all-frame-axioms-structure*:
cnf (*encode-all-frame-axioms* Π *t*)
= ⋃(⋃(*k*, *v*) ∈ ({*0*..<*t*} × *set* $((\Pi)_{\mathcal{V}})$).
{{{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }
∪ {(*Operator k* (*index* (*strips-problem.operators-of* Π) *op*))$^{+}$

$\mid$ *op. op* $\in$ *set* $((\Pi)_{\mathcal{O}}) \wedge v \in$ *set* (*add-effects-of op*) $\}\}\})$
$\cup \bigcup(\bigcup(k, v) \in (\{0..<t\} \times set\ ((\Pi)_{\mathcal{V}}))$.
$\{\{\{$ (*State k* (*index* (*strips-problem.variables-of* Π) *v*)$)^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*)$)^{+}$ $\}$
$\cup$ $\{$ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*)$)^{+}$
$\mid$ *op. op* $\in$ *set* $((\Pi)_{\mathcal{O}}) \wedge v \in$ *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*
(*index* (*strips-problem.variables-of* Π) *v*) $(((\Pi)_G)\ v))$
$\mid$ *v. v* $\in$ *set* $((\Pi)_{\mathcal{V}}) \wedge ((\Pi)_G)\ v \neq$ *None* $\})$
⟨*proof*⟩

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

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

lemma *decode-plan-length*:
assumes $\pi = \Phi^{-1}$ Π ν t
shows *length* $\pi = t$
⟨*proof*⟩

lemma *decode-plan′-set-is*[*simp*]:
set (*decode-plan′* Π $\mathcal{A}$ k)
= { (*strips-problem.operators-of* Π) ! (*index* (*strips-problem.operators-of* Π) *op*)
| *op*. *op* $\in$ *set* (*strips-problem.operators-of* Π)
$\wedge$ $\mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π) *op*)) }
⟨*proof*⟩

lemma *decode-plan-set-is*[*simp*]:
set (Φ^{-1} Π $\mathcal{A}$ t) = ($\bigcup k \in \{..<t\}$. { *decode-plan′* Π $\mathcal{A}$ k })
⟨*proof*⟩

lemma *decode-plan-step-element-then-i*:
assumes $k < t$
shows *set* ((Φ^{-1} Π $\mathcal{A}$ t) ! k)
= { (*strips-problem.operators-of* Π) ! (*index* (*strips-problem.operators-of* Π) *op*)
| *op*. *op* $\in$ *set* ((Π)$_{\mathcal{O}}$) $\wedge$ $\mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π)
op)) }
⟨*proof*⟩

lemma *decode-plan-step-element-then*:
fixes Π::′*a strips-problem*
assumes $k < t$
and *op* $\in$ *set* ((Φ^{-1} Π $\mathcal{A}$ t) ! k)
shows *op* $\in$ *set* ((Π)$_{\mathcal{O}}$)
and $\mathcal{A}$ (*Operator* k (*index* (*strips-problem.operators-of* Π) *op*))
⟨*proof*⟩

lemma *decode-plan-step-distinct*:
assumes $k < t$
shows *distinct* ((Φ^{-1} Π $\mathcal{A}$ t) ! k)
⟨*proof*⟩

lemma *decode-state-at-valid-variable*:
fixes Π :: ′*a strips-problem*
assumes (${\Phi_S}^{-1}$ Π $\mathcal{A}$ k) $v \neq$ *None*
shows $v \in$ *set* ((Π)$_{\mathcal{V}}$)
⟨*proof*⟩

lemma *decode-state-at-encoding-variables-equals-some-of-valuation-if*:
fixes Π:: ′*a strips-problem*
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi$ Π t
and $k \leq t$
and $v \in$ *set* ((Π)$_{\mathcal{V}}$)
shows (${\Phi_S}^{-1}$ Π $\mathcal{A}$ k) v

$= Some\ (\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v)))$
⟨*proof*⟩

lemma *decode-state-at-dom*:
assumes *is-valid-problem-strips* Π
shows $dom\ (\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ k) = set\ ((\Pi)_{\mathcal{V}})$
⟨*proof*⟩

lemma *decode-state-at-initial-state*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
shows $(\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ 0) = (\Pi)_I$
⟨*proof*⟩

lemma *decode-state-at-goal-state*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
shows $(\Pi)_G \subseteq_m \Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ t$
⟨*proof*⟩
lemma *decode-state-at-preconditions*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
and $k < t$
and $op \in set\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
and $v \in set\ (precondition\text{-}of\ op)$
shows $\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))$
⟨*proof*⟩

lemma *encode-problem-parallel-correct-i*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ 0$
shows $cnf\ ((\Phi_G\ \Pi)\ 0) \subseteq cnf\ (\Phi_I\ \Pi)$
⟨*proof*⟩
lemma *encode-problem-parallel-correct-ii*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
and $k < length\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
shows $are\text{-}all\text{-}operators\text{-}applicable\ (\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ k)$
$((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
and $are\text{-}all\text{-}operator\text{-}effects\text{-}consistent\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
⟨*proof*⟩
lemma *encode-problem-parallel-correct-iii*:
assumes *is-valid-problem-strips* Π
and $\mathcal{A} \models \Phi\ \Pi\ t$
and $k < length\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
and $op \in set\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k)$
shows $v \in set\ (add\text{-}effects\text{-}of\ op)$
$\longrightarrow (\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ (Suc\ k))\ v = Some\ True$

and $v \in set$ (*delete-effects-of op*)
$\longrightarrow (\Phi_S{}^{-1}\ \Pi\ \mathcal{A}\ (Suc\ k))\ v = Some\ False$
⟨*proof*⟩
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 $v \in set\ ((\Pi)_{\mathcal{V}})$
and $\neg(\exists\ op \in set\ ((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)\ !\ k).$
$v \in set$ (*add-effects-of op*) $\vee$ $v \in set$ (*delete-effects-of op*))
shows *cnf-semantics* $\mathcal{A}$ {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$ }}
and *cnf-semantics* $\mathcal{A}$ {{ (*State k* (*index* (*strips-problem.variables-of* Π) *v*))$^{+}$
, (*State* (*Suc k*) (*index* (*strips-problem.variables-of* Π) *v*))$^{-1}$ }}
⟨*proof*⟩

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

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

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

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

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 $ops \in set\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
and $op \in set\ ops$
shows $op \in set\ ((\Pi)_{\mathcal{O}})$
⟨*proof*⟩

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

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

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

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

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

be the trace of π. Theorem **??** first establishes the equality $\sigma = \tau$ of the decoded state sequence and the trace of π. We can then derive that $G \subseteq_m last\ \sigma$ by lemma **??**, i.e. the last state reached by plan execution (and moreover the last state decoded from the model), satisfies the goal state G defined by the problem. By lemma **??**, we can conclude that π is a solution for I and G.

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

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

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

value *stop*

7.4 Completeness

definition *empty-valuation* :: *sat-plan-variable valuation* ($\mathcal{A}_0$)

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

where *empty-valuation* ≡ (λ-. *False*)

abbreviation *valuation-for-state*
:: *'variable list*
⇒*'variable strips-state*
⇒ *nat*
⇒ *'variable*
⇒ *sat-plan-variable valuation*
⇒ *sat-plan-variable valuation*
where *valuation-for-state vs s k v* $\mathcal{A}$
≡ $\mathcal{A}$(*State k* (*index vs v*) := (*s v* = *Some True*))

— Since the trace may be shorter than the plan length even though the last trace element subsumes the goal state—namely in case plan execution is impossible due to violation of the execution condition but the reached state serendipitously subsumes the goal state—, we also have to repeat the valuation for all time steps $k' \in \{length\ \tau..length\ \pi + 1\}$ for all $v \in \mathcal{V}$ (see $\mathcal{A}_2$).
definition *valuation-for-state-variables*
:: *'variable strips-problem*
⇒ *'variable strips-operator list list*
⇒ *'variable strips-state list*
⇒ *sat-plan-variable valuation*
where *valuation-for-state-variables* Π π τ ≡ *let*
t' = *length* τ
; $\tau_\Omega = \tau\ !\ (t' - 1)$
; *vs* = *variables-of* Π
; V_1 = { *State k* (*index vs v*) | *k v*. $k \in \{0..<t'\} \wedge v \in$ *set vs* }
; V_2 = { *State k* (*index vs v*) | *k v*. $k \in \{t'..(length\ \pi + 1)\} \wedge v \in$ *set vs* }
; $\mathcal{A}_1$ = *foldr*
(λ(*k*, *v*) $\mathcal{A}$. *valuation-for-state* (*variables-of* Π) (τ ! *k*) *k v* $\mathcal{A}$)
(*List.product* [*0*..<t'] *vs*)
$\mathcal{A}_0$
; $\mathcal{A}_2$ = *foldr*
(λ(*k*, *v*) $\mathcal{A}$. *valuation-for-state* (*variables-of* Π) τ_Ω *k v* $\mathcal{A}$)
(*List.product* [t'..<*length* π + *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 $k' \in \{length\ \tau..length\ \pi + 1\}$ since no more operators are executed after the trace ends anyway. The definition of $\mathcal{A}_0$ as the valuation that is false for every argument ensures this implicitely.
definition *valuation-for-operator-variables*
:: *'variable strips-problem*
⇒ *'variable strips-operator list list*
⇒ *'variable strips-state list*
⇒ *sat-plan-variable valuation*
where *valuation-for-operator-variables* Π π τ ≡ *let*
ops = *operators-of* Π

; $Op = \{$ *Operator* k (*index ops op*) $\mid k$ *op*. $k \in \{0..<\mathit{length}\ \tau - 1\} \wedge op \in$ *set ops* $\}$
in override-on
$\mathcal{A}_0$
(*foldr*
($\lambda(k,\ op)\ \mathcal{A}.\ \mathcal{A}(\mathit{Operator}\ k\ (\mathit{index\ ops\ op}) := \mathit{True})$)
(*concat* (*map* ($\lambda k.$ *map* (*Pair* k) ($\pi\ !\ k$)) $[0..<\mathit{length}\ \tau - 1]$))
$\mathcal{A}_0$)
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 $\tau \equiv$ *trace-parallel-plan-strips* $I\ \pi$ with

$\tau\ !\ k =$ *execute-parallel-plan* I (*take* $k\ \pi$)

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

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

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

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

for all operators $op \in \mathcal{O}$ s.t. $op \in$ *set* $(\pi\ !\ k)$ for all $k <$ *length* $\tau - 1$.
The overall valuation for the plan execution $\mathcal{A}$ can now be constructed by combining the state variable valuation $\mathcal{A}_V$ and operator valuation $\mathcal{A}_O$.

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

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

$\mid k\ op.\ k \in \{0..<t\} \wedge op \in set\ ops\ \}$
$in\ override\text{-}on\ (override\text{-}on\ \mathcal{A}_0\ \mathcal{A}_V\ \ V)\ \mathcal{A}_O\ \ Op$

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

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

lemma *empty-valution-is-False*[*simp*]: $\mathcal{A}_0\ \ v = False$
⟨*proof*⟩

lemma *model-initial-state-set-valuations*:
assumes *is-valid-problem-strips* Π
shows $set\ (map\ (\lambda v.\ case\ ((\Pi)_I)\ v\ of\ Some\ b$
$\Rightarrow \mathcal{A}_0(State\ 0\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v) := b)$
$\mid \text{-} \Rightarrow \mathcal{A}_0)$
$(strips\text{-}problem.variables\text{-}of\ \Pi))$
$= \{\ \mathcal{A}_0(State\ 0\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v) := the\ (((\Pi)_I)\ v))$
$\mid v.\ v \in set\ ((\Pi)_{\mathcal{V}})\ \}$
⟨*proof*⟩

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

lemma *foldr-fun-upd*:
assumes *inj-on f* (*set xs*)
and $x \in set\ xs$
shows $foldr\ (\lambda x\ \mathcal{A}.\ \mathcal{A}(f\ x := g\ x))\ xs\ \mathcal{A}\ (f\ x) = g\ x$
⟨*proof*⟩

lemma *foldr-fun-no-upd*:
assumes *inj-on f* (*set xs*)
and $y \notin f\ `\ set\ xs$
shows $foldr\ (\lambda x\ \mathcal{A}.\ \mathcal{A}(f\ x := g\ x))\ xs\ \mathcal{A}\ y = \mathcal{A}\ y$
⟨*proof*⟩
lemma *encode-problem-parallel-complete-i*:
fixes Π::'*a strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m execute\text{-}parallel\text{-}plan\ ((\Pi)_I)\ \pi$

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

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

lemma *encode-problem-parallel-complete-iii-b*:
fixes Π::*'a strips-problem*
assumes *is-valid-problem-strips* Π
and $(\Pi)_G \subseteq_m$ *execute-parallel-plan* $((\Pi)_I)$ π
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* $((\Pi)_I)$ π) − *1*
$\longrightarrow$ $\mathcal{A}$ (*Operator k* (*index* (*strips-problem.operators-of* Π) *op*)) = (*op* $\in$ *set* (π ! *k*))
and $\forall$ *l op. l* $\geq$ *length* (*trace-parallel-plan-strips* $((\Pi)_I)$ π) − *1* $\wedge$ *l* $<$ *length* π
$\longrightarrow$ $\neg\mathcal{A}$ (*Operator l* (*index* (*strips-problem.operators-of* Π) *op*))

and $\forall v\ k.\ k < \textit{length}\ (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi)$
$\longrightarrow\ (\mathcal{A}\ (\textit{State}\ k\ (\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))$
$\longleftrightarrow (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi\ !\ k)\ v = \textit{Some True})$
shows $\textit{clause-semantics}\ \mathcal{A}\ C$
⟨*proof*⟩

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

lemma *encode-problem-parallel-complete-iv-a*:
fixes Π :: '*a strips-problem*
assumes *STRIPS-Semantics.is-parallel-solution-for-problem* $\Pi\ \pi$
and $\forall k\ op.\ k < \textit{length}\ (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi) - 1$
$\longrightarrow \mathcal{A}\ (\textit{Operator}\ k\ (\textit{index}\ (\textit{strips-problem.operators-of}\ \Pi)\ op)) = (op \in \textit{set}\ (\pi\ !\ k))$
and $\forall v\ k.\ k < \textit{length}\ (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi)$
$\longrightarrow\ (\mathcal{A}\ (\textit{State}\ k\ (\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))$
$\longleftrightarrow (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi\ !\ k)\ v = \textit{Some True})$
and $\forall v\ l.\ l \geq \textit{length}\ (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi) \wedge l < \textit{length}\ \pi + 1$
$\longrightarrow \mathcal{A}\ (\textit{State}\ l\ (\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))$
$= \mathcal{A}\ (\textit{State}$
$(\textit{length}\ (\textit{trace-parallel-plan-strips}\ ((\Pi)_I)\ \pi) - 1)$
$(\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))$
and $C \in \bigcup\ (\bigcup(k,\ v) \in \{0..<\textit{length}\ \pi\} \times \textit{set}\ ((\Pi)_{\mathcal{V}}).$
$\{\{\{\ (\textit{State}\ k\ (\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))^{+}$
$,\ (\textit{State}\ (\textit{Suc}\ k)\ (\textit{index}\ (\textit{strips-problem.variables-of}\ \Pi)\ v))^{-1}\ \}$
$\cup\ \{\ (\textit{Operator}\ k\ (\textit{index}\ (\textit{strips-problem.operators-of}\ \Pi)\ op))^{+}$
$|op.\ op \in \textit{set}\ ((\Pi)_{\mathcal{O}}) \wedge v \in \textit{set}\ (\textit{add-effects-of}\ op)\ \}\}\})$
shows $\textit{clause-semantics}\ \mathcal{A}\ C$
⟨*proof*⟩

lemma *encode-problem-parallel-complete-iv-b*:
fixes Π :: '*a strips-problem*
assumes *is-parallel-solution-for-problem* $\Pi\ \pi$

and $\forall k\ op.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1$
$\longrightarrow \mathcal{A}\ (Operator\ k\ (index\ (strips\text{-}problem.operators\text{-}of\ \Pi)\ op)) = (op \in set\ (\pi\ !\ k))$
and $\forall v\ k.\ k < length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi)$
$\longrightarrow (\mathcal{A}\ (State\ k\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))$
$\longleftrightarrow (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi\ !\ k)\ v = Some\ True)$
and $\forall v\ l.\ l \geq length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) \wedge l < length\ \pi + 1$
$\longrightarrow \mathcal{A}\ (State\ l\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))$
$= \mathcal{A}\ (State$
$(length\ (trace\text{-}parallel\text{-}plan\text{-}strips\ ((\Pi)_I)\ \pi) - 1)$
$(index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))$
and $C \in \bigcup\ (\bigcup(k,\ v) \in \{0..{<}length\ \pi\} \times set\ ((\Pi)_{\mathcal{V}}).$
$\{\{\{\ (State\ k\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^{-1}$
$,\ (State\ (Suc\ k)\ (index\ (strips\text{-}problem.variables\text{-}of\ \Pi)\ v))^{+}\ \}$
$\cup \{\ (Operator\ k\ (index\ (strips\text{-}problem.operators\text{-}of\ \Pi)\ op))^{+}$
$|op.\ op \in set\ ((\Pi)_{\mathcal{O}}) \wedge v \in set\ (delete\text{-}effects\text{-}of\ op)\ \}\}\})$
shows $clause\text{-}semantics\ \mathcal{A}\ C$
⟨*proof*⟩

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

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

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

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

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

lemma *list-product-is-nil-iff*: *List.product xs ys* = [] $\longleftrightarrow$ *xs* = [] $\vee$ *ys* = []
⟨*proof*⟩
lemma *valuation-for-state-variables-is*:
assumes $k \in$ *set ks*
and $v \in$ *set vs*
shows *foldr* ($\lambda(k, v)$ $\mathcal{A}$. *valuation-for-state vs* (*s k*) *k v* $\mathcal{A}$) (*List.product ks vs*) $\mathcal{A}_0$
(*State k* (*index vs v*))
$\longleftrightarrow$ (*s k*) *v* = *Some True*
⟨*proof*⟩

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

⟨*proof*⟩

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

Let now $\tau \equiv$ *trace-parallel-plan-strips* I π be the trace of the plan π, $t \equiv$ *length* π, and $t' \equiv$ *length* τ.
Any model of the SATPlan encoding $\mathcal{A}$ must satisfy the following properties: [11]

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

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

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

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

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

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

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

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

Condition “1.” states that the model must reflect operator activation for all operators in the parallel operator lists $\pi\ !\ k$ of the plan π for each time step $k < t' - (1{::}'a)$ s.t. there is a successor state in the trace. Moreover “3.” requires that the model is consistent with the states reached during plan execution (i.e. the elements $\tau\ !\ k$ for $k < t'$ of the trace τ). Meaning that $\mathcal{A}$

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

($State\ k\ (index\ (\Pi_{\mathcal{V}})\ v)$) for the SAT plan variable of every state variable v at time point k if and only if $(\tau\ !\ k)\ v = Some\ True$ for the corresponding state $\tau\ !\ k$ at time k (and $\neg\ \mathcal{A}\ (State\ k\ (index\ (\Pi_{\mathcal{V}})\ v))$ otherwise).

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

value *stop*

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

theorem
encode-problem-parallel-complete:
assumes *is-valid-problem-strips* Π
and *is-parallel-solution-for-problem* $\Pi\ \pi$
shows *valuation-for-plan* $\Pi\ \pi \models \Phi\ \Pi$ (*length* π)
⟨*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

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

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

definition *encode-interfering-operator-pair-exclusion*
:: *'variable strips-problem*
⇒ *nat*
⇒ *'variable strips-operator*

$\Rightarrow$ *'variable strips-operator*
$\Rightarrow$ *sat-plan-variable formula*
where *encode-interfering-operator-pair-exclusion* Π k op_1 op_2
$\equiv$ *let ops = operators-of* Π *in*
$\neg$(*Atom* (*Operator* k (*index ops* op_1)))
$\vee$ $\neg$(*Atom* (*Operator* k (*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 = operators-of Π
; *interfering = filter* (λ(op_1, op_2). *index ops* op_1 $\neq$ *index ops* op_2
$\wedge$ *are-operators-interfering* op_1 op_2) (*List.product ops ops*)
in foldr ($\wedge$) [*encode-interfering-operator-pair-exclusion* Π k op_1 op_2.
(op_1, op_2) $\leftarrow$ *interfering*, k $\leftarrow$ [*0..*<*t*]] ($\neg\bot$)

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

— NOTE This is the quadratic size encoding for the $\forall$-step semantics as defined in [3, 3.2.1, p.1045]. This encoding ensures that decoded plans are sequentializable by simply excluding the simultaneous execution of operators with potential interference at any timepoint. Note that this yields a $\forall$-step plan for which parallel operator execution at any time step may be sequentialised in any order (due to non-interference).

definition *encode-problem-with-operator-interference-exclusion*
:: *'variable strips-problem* $\Rightarrow$ *nat* $\Rightarrow$ *sat-plan-variable formula*
($\Phi_\forall$ - - *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 proof the sublocale proposition for strips in order to gain access to definitions and lemmas.

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

lemma *cnf-of-encode-interfering-operator-exclusion-is-ii*[*simp*]:
set [*encode-interfering-operator-pair-exclusion* Π k op_1 op_2.

$(op_1,\ op_2) \leftarrow$ *filter* $(\lambda(op_1,\ op_2).$
index (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
$\wedge$ *are-operators-interfering* op_1 op_2)
(*List.product* (*strips-problem.operators-of* Π) (*strips-problem.operators-of* Π))
, $k \leftarrow [0..<t]]$
$= (\bigcup (op_1,\ op_2)$
$\in \{\ (op_1,\ op_2) \in$ *set* (*operators-of* Π) $\times$ *set* (*operators-of* Π).
index (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
$\wedge$ *are-operators-interfering* op_1 op_2 }.
($\lambda k.$ *encode-interfering-operator-pair-exclusion* Π k op_1 op_2) ‘ $\{0..<t\}$)
⟨*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 op_1 op_2.
$(op_1,\ op_2) \leftarrow$ *filter* $(\lambda(op_1,\ op_2).$
index (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
$\wedge$ *are-operators-interfering* op_1 op_2)
(*List.product* (*strips-problem.operators-of* Π) (*strips-problem.operators-of* Π))
, $k \leftarrow [0..<t]]$
$= (\bigcup (op_1,\ op_2)$
$\in \{\ (op_1,\ op_2) \in$ *set* (*strips-problem.operators-of* Π) $\times$ *set* (*strips-problem.operators-of* Π).
index (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
$\wedge$ *are-operators-interfering* op_1 op_2 }.
$\{\{\{$ (*Operator* k (*index* (*strips-problem.operators-of* Π) op_1))$^{-1}$
, (*Operator* k (*index* (*strips-problem.operators-of* Π) op_2))$^{-1}$ $\}\}$ | $k.$ $k \in \{0..<t\}\}$)
⟨*proof*⟩

lemma *cnf-of-encode-interfering-operator-exclusion-is*:
cnf (*encode-interfering-operator-exclusion* Π t) $= \bigcup (\bigcup (op_1,\ op_2)$
$\in \{\ (op_1,\ op_2) \in$ *set* (*operators-of* Π) $\times$ *set* (*operators-of* Π).
index (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
$\wedge$ *are-operators-interfering* op_1 op_2 }.
$\{\{\{$ (*Operator* k (*index* (*strips-problem.operators-of* Π) op_1))$^{-1}$
, (*Operator* k (*index* (*strips-problem.operators-of* Π) op_2))$^{-1}$ $\}\}$ | $k.$ $k \in \{0..<t\}\}$)
⟨*proof*⟩

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

fixes Π :: *'variable strips-problem*
assumes $k < t$
and $op_1 \in set$ (*strips-problem.operators-of* Π) **and** $op_2 \in set$ (*strips-problem.operators-of* Π)
and *index* (*strips-problem.operators-of* Π) $op_1 \neq$ *index* (*strips-problem.operators-of* Π) op_2
and *are-operators-interfering* op_1 op_2
shows { (*Operator* k (*index* (*strips-problem.operators-of* Π) op_1))$^{-1}$
, (*Operator* k (*index* (*strips-problem.operators-of* Π) op_2))$^{-1}$}
$\in$ *cnf* (*encode-interfering-operator-exclusion* Π t)
⟨*proof*⟩

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

fixes Π :: *'variable strips-problem*
shows *is-cnf* (*encode-interfering-operator-exclusion* Π t)
⟨*proof*⟩

lemma *is-cnf-encode-problem-with-operator-interference-exclusion*:
assumes *is-valid-problem-strips* Π
shows *is-cnf* ($\Phi_\forall$ Π t)
⟨*proof*⟩

lemma *cnf-of-encode-problem-with-operator-interference-exclusion-structure*:
shows *cnf* (Φ_I Π) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* ((Φ_G Π) t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-operators* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-all-frame-axioms* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
and *cnf* (*encode-interfering-operator-exclusion* Π t) $\subseteq$ *cnf* ($\Phi_\forall$ Π t)
⟨*proof*⟩

lemma *encode-problem-with-operator-interference-exclusion-has-model-then-also-partial-encodings*:
assumes $\mathcal{A} \models \Phi_\forall$ Π t
shows $\mathcal{A} \models$ *SAT-Plan-Base.encode-initial-state* Π
and $\mathcal{A} \models$ *SAT-Plan-Base.encode-operators* Π t
and $\mathcal{A} \models$ *SAT-Plan-Base.encode-all-frame-axioms* Π t
and $\mathcal{A} \models$ *encode-interfering-operator-exclusion* Π t
and $\mathcal{A} \models$ *SAT-Plan-Base.encode-goal-state* Π t
⟨*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 and . The sublocale proof is shown below. It confirms that the new encoding again a CNF as required by locale .

8.1 Soundness

The Proof of soundness for the SATPlan encoding with interfering operator exclusion follows directly from the proof of soundness of the basic SATPlan encoding. By looking at the structure of the new encoding which simply extends the basic SATPlan encoding with a conjunct, any model for encoding with exclusion of operator interference also models the basic SATPlan encoding and the soundness of the new encoding therefore follows from theorem **??**.

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

— NOTE We use the *subseq* formulation in the fourth assumption to be able to instantiate the induction hypothesis on the subseq *ops* given the induction premise $op \mathbin{\#} ops \in set\ (subseqs\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t\ !\ k))$. We do not use subsets in the assumption since we would otherwise lose the distinctness property which can be infered from $ops \in set\ (subseqs\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t\ !\ k))$ using lemma *subseqs-distinctD*.

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

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 $\forall k < length\ (\Phi^{-1}\ \Pi\ \mathcal{A}\ t).$ *are-all-operators-non-interfering* $((\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$
$!\ k)$
⟨*proof*⟩

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 $\forall k < length\ \pi.$ *are-all-operators-non-interfering* $(\pi\ !\ k)$
  shows *valuation-for-plan* $\Pi\ \pi \models$ *encode-interfering-operator-exclusion* $\Pi\ (length$
$\pi)$
⟨*proof*⟩

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

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

where $\mathit{ops} \equiv \Pi_{\mathcal{O}}$. Now, consider an operator op_1 that is contained in the k-th plan step $\pi\ !\ k$ (symmetrically for op_2). Since π is a serializable solution, there can be no interference between op_1 and op_2 at time k. Hence op_2 cannot be in $\pi\ !\ k$ This entails that for $\mathcal{A} \equiv$ *valuation-for-plan* $\Pi\ \pi$ it holds that

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

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

Furthermore, if neither is present, than $\mathcal{A}$ will evaluate both atoms to false and the clause therefore evaluates to true as well.

It follows from this that each clause in the extension of the SATPlan encoding evaluates to true for $\mathcal{A}$. The other parts of the encoding evaluate to true as per the completeness of the basic SATPlan encoding (theorem **??**).

theorem *encode-problem-serializable-complete*:
  assumes *is-valid-problem-strips* Π
    and *is-parallel-solution-for-problem* $\Pi\ \pi$
    and $\forall\, k <$ *length* π. *are-all-operators-non-interfering* $(\pi\ !\ k)$
  shows *valuation-for-plan* $\Pi\ \pi \models \Phi_\forall\ \Pi$ (*length* π)
⟨*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-serial-plan* $((\Pi)_I)$ (*concat* $(\Phi^{-1}\ \Pi\ \mathcal{A}\ t)$)
⟨*proof*⟩

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

  fixes Π :: *'variable strips-problem*
  shows *list-all* (λ*op*. *ListMem op* (*strips-problem.operators-of* Π))
    (*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* π is a serial solution for Π. To this end, we have to show that

- the state reached by serial execution of π' subsumes G, and

- all operators in π' are operators contained in $\mathcal{O}$.

While the proof of the latter step is rather straight forward, the proof for the former requires a bit more work. We use the previously established theorem on serial and parallel STRIPS equivalence (theorem **??**) to show the serializability of π and therefore have to show that G is subsumed by the last state of the trace of π'

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

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

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

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

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

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

theorem *serializable-encoding-decoded-plan-is-serializable*:
  assumes *is-valid-problem-strips* Π
    and $\mathcal{A} \models \Phi_\forall$ Π t
  shows *is-serial-solution-for-problem* Π (*concat* (Φ^{-1} Π $\mathcal{A}$ 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 $\mathcal{A} \models \Phi_\forall$ (φ Ψ) t
  shows *is-serial-solution-for-problem* Ψ [${\varphi_O}^{-1}$ Ψ *op*. *op* ← *concat* (Φ^{-1} (φ Ψ) $\mathcal{A}$ t)]
⟨*proof*⟩

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

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

assumes *is-valid-problem-sas-plus* Ψ
and *is-serial-solution-for-problem* Ψ ψ
and h = *length* ψ
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 $(\exists \psi.$ *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* ≡ (|*SAS-Plus-Representation.sas-plus-operator.precondition-of* = [],
SAS-Plus-Representation.sas-plus-operator.effect-of = []|)

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

definition
prob-with-noop Π ≡
(|*SAS-Plus-Representation.sas-plus-problem.variables-of* = *SAS-Plus-Representation.sas-plus-problem.varia*
Π,
SAS-Plus-Representation.sas-plus-problem.operators-of = *empty-sasp-action*
SAS-Plus-Representation.sas-plus-problem.operators-of Π,
SAS-Plus-Representation.sas-plus-problem.initial-of = *SAS-Plus-Representation.sas-plus-problem.initial-o*
Π,
SAS-Plus-Representation.sas-plus-problem.goal-of = *SAS-Plus-Representation.sas-plus-problem.goal-of*
Π,
SAS-Plus-Representation.sas-plus-problem.range-of = *SAS-Plus-Representation.sas-plus-problem.range-of*
Π|)

lemma *sasp-noops-in-noop-problem*: *set* (*replicate n empty-sasp-action*) ⊆ *set* (*SAS-Plus-Representation.sas-p*
(*prob-with-noop* Π))
⟨*proof*⟩

lemma *noops-complete*:
SAS-Plus-Semantics.is-serial-solution-for-problem Ψ π $\Longrightarrow$
SAS-Plus-Semantics.is-serial-solution-for-problem (*prob-with-noop* Ψ) ((*replicate n empty-sasp-action*) @ π)
⟨*proof*⟩

definition *rem-noops* ≡ *filter* (λ*op*. *op* ≠ *empty-sasp-action*)

lemma *sasp-filter-empty-action*:
execute-serial-plan-sas-plus s (*rem-noops* πs) = *execute-serial-plan-sas-plus s* πs
⟨*proof*⟩

lemma *noops-sound*:
SAS-Plus-Semantics.is-serial-solution-for-problem (*prob-with-noop* Ψ) πs $\Longrightarrow$
SAS-Plus-Semantics.is-serial-solution-for-problem Ψ (*rem-noops* πs)
⟨*proof*⟩

lemma *noops-valid*: *is-valid-problem-sas-plus* Ψ $\Longrightarrow$ *is-valid-problem-sas-plus* (*prob-with-noop* Ψ)
⟨*proof*⟩

lemma *sas-plus-problem-has-serial-solution-iff-i'*:
assumes *is-valid-problem-sas-plus* Ψ
and $\mathcal{A} \models \Phi_\forall$ (φ (*prob-with-noop* Ψ)) *t*
shows *SAS-Plus-Semantics.is-serial-solution-for-problem* Ψ
(*rem-noops*
(*map* (λ*op*. ${\varphi_O}^{-1}$ (*prob-with-noop* Ψ) *op*)
(*concat* (Φ^{-1} (φ (*prob-with-noop* Ψ)) $\mathcal{A}$ *t*))))
⟨*proof*⟩

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

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

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

11.1 Translating Fast-Downward's represnetation to SAS+

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

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

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

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

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

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

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

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

lemma *first-map-add-submap*: (⋀*x*. *x* ∈ *dom m1* ∧ *x* ∈ *dom m2* ⟹ *m1 x* = *m2 x*) ⟹
  $m1 \mathbin{++} m2 \subseteq_m x \Longrightarrow m1 \subseteq_m x$
 ⟨*proof*⟩

lemma *subsuming-states-map-add*:
 (⋀*x*. *x* ∈ *dom m1* ∩ *dom m2* ⟹ *m1 x* = *m2 x*) ⟹
 $m1 \mathbin{++} m2 \subseteq_m s \longleftrightarrow (m1 \subseteq_m s \wedge m2 \subseteq_m s)$
 ⟨*proof*⟩

lemma *consistent-map-lists*:

$[\![$*distinct* (*map fst* (*xs1* @ *xs2*)); $x \in$ *dom* (*map-of xs1*) $\cap$ *dom* (*map-of xs2*)$]\!] \Longrightarrow$

(*map-of xs1*) x = (*map-of xs2*) x
⟨*proof*⟩

lemma *subsuming-states-append*:
distinct (*map fst* (*xs* @ *ys*)) $\Longrightarrow$
(*map-of* (*xs* @ *ys*)) $\subseteq_m$ *s* $\longleftrightarrow$ ((*map-of ys*) $\subseteq_m$ *s* $\wedge$ (*map-of xs*) $\subseteq_m$ *s*)
⟨*proof*⟩

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

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

lemma *consistent-pres-op-then′*: *consistent-pres-op op* $\Longrightarrow$ *consistent-pres-op′ op*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-valid-states*:
ast-problem.valid-states (*rem-implicit-pres-ops prob*) = *ast-problem.valid-states prob*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-lookup-op-None*:
ast-problem.lookup-operator (*vars*, *init*, *goal*, *ops*) *name* = *None* $\longleftrightarrow$
ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name* = *None*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-lookup-op-Some-1*:
ast-problem.lookup-operator (*vars*, *init*, *goal*, *ops*) *name* = *Some* (*n*,*p*,*vp*,*e*) $\Longrightarrow$
ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name* =
Some (*rem-implicit-pres* (*n*,*p*,*vp*,*e*))
⟨*proof*⟩

lemma *rem-implicit-pres-ops-lookup-op-Some-1′*:
ast-problem.lookup-operator prob name = *Some* (*n*,*p*,*vp*,*e*) $\Longrightarrow$
ast-problem.lookup-operator (*rem-implicit-pres-ops prob*) *name* =
Some (*rem-implicit-pres* (*n*,*p*,*vp*,*e*))
⟨*proof*⟩

lemma *implicit-pres-empty*: *implicit-pres* (*map rem-effect-implicit-pres effs*) = []
⟨*proof*⟩

lemma *rem-implicit-pres-ops-lookup-op-Some-2*:
ast-problem.lookup-operator (*rem-implicit-pres-ops* (*vars*, *init*, *goal*, *ops*)) *name* = *Some op*
$\Longrightarrow$ $\exists$ *op'*. *ast-problem.lookup-operator* (*vars*, *init*, *goal*, *ops*) *name* = *Some op'* $\wedge$
(*op* = *rem-implicit-pres op'*)
⟨*proof*⟩

lemma *rem-implicit-pres-ops-lookup-op-Some-2'*:
ast-problem.lookup-operator (*rem-implicit-pres-ops prob*) *name* = *Some* (*n*,*p*,*e*,*c*)
$\Longrightarrow$ $\exists$ *op'*. *ast-problem.lookup-operator prob name* = *Some op'* $\wedge$
((*n*,*p*,*e*,*c*) = *rem-implicit-pres op'*)
⟨*proof*⟩

lemma *subsuming-states-def'*:
s $\in$ *ast-problem.subsuming-states prob ps* = (*s* $\in$ (*ast-problem.valid-states prob*) $\wedge$ *ps* $\subseteq_m$ *s*)
⟨*proof*⟩

lemma *rem-implicit-pres-ops-enabled-1*:
⟦($\bigwedge$*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *consistent-pres-op op*);
ast-problem.enabled prob name s⟧ $\Longrightarrow$
ast-problem.enabled (*rem-implicit-pres-ops prob*) *name s*
⟨*proof*⟩

context *ast-problem*
begin

lemma *lookup-Some-inδ*: *lookup-operator* π = *Some op* $\Longrightarrow$ *op*$\in$*set astδ*
⟨*proof*⟩

end

lemma *rem-implicit-pres-ops-enabled-2*:
assumes ($\bigwedge$*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *consistent-pres-op op*)
shows *ast-problem.enabled* (*rem-implicit-pres-ops prob*) *name s* $\Longrightarrow$
ast-problem.enabled prob name s
⟨*proof*⟩

lemma *rem-implicit-pres-ops-enabled*:
($\bigwedge$*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *consistent-pres-op op*) $\Longrightarrow$
ast-problem.enabled (*rem-implicit-pres-ops prob*) *name s* = *ast-problem.enabled prob name s*
⟨*proof*⟩

context *ast-problem*
begin

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

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

end

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

lemma *is-standard-operator-rem-implicit-pres-ops*:
⟦(⋀*op*. *op* $\in$ *set* (*ast-problem.astδ* (*a*,*b*,*c*,*d*)) $\Longrightarrow$ *is-standard-operator op*);
op $\in$ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops* (*a*,*b*,*c*,*d*)))⟧
$\Longrightarrow$ *is-standard-operator op*
⟨*proof*⟩

lemma *is-standard-operator-rem-implicit-pres-ops'*:
⟦*op* $\in$ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*));
(⋀*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *is-standard-operator op*)⟧
$\Longrightarrow$ *is-standard-operator op*
⟨*proof*⟩

lemma *in-rem-implicit-pres-δ*:
op $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$
rem-implicit-pres op $\in$ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*))
⟨*proof*⟩

lemma *rem-implicit-pres-ops-execute*:
assumes
(⋀*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *is-standard-operator op*) **and**
s $\in$ *ast-problem.valid-states prob*
shows *ast-problem.execute* (*rem-implicit-pres-ops prob*) *name s* = *ast-problem.execute prob name s*
⟨*proof*⟩

lemma *rem-implicit-pres-ops-path-to*:
wf-ast-problem prob $\Longrightarrow$
(⋀*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *consistent-pres-op op*) $\Longrightarrow$
(⋀*op*. *op* $\in$ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *is-standard-operator op*) $\Longrightarrow$
s $\in$ *ast-problem.valid-states prob* $\Longrightarrow$
ast-problem.path-to (*rem-implicit-pres-ops prob*) *s* πs s' = *ast-problem.path-to prob s* πs s'
⟨*proof*⟩

lemma *rem-implicit-pres-ops-astG*[*simp*]: *ast-problem.astG* (*rem-implicit-pres-ops prob*) =
ast-problem.astG 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* ∈ *set* (*ast-problem.astδ prob*) ⟹ *consistent-pres-op op*)
($\bigwedge$*op. op* ∈ *set* (*ast-problem.astδ prob*) ⟹ *is-standard-operator op*)
shows *ast-problem.valid-plan* (*rem-implicit-pres-ops prob*) *πs* = *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*) ∈ *set* (*implicit-pres effs*) ⟹ (∃ *epres v vp.* (*epres*,*x*,*vp*,*v*)∈ *set effs* ∧ *vp* = *Some a*)
⟨*proof*⟩

lemma *pair4-eqD*: (*a1*,*a2*,*a3*,*a4*) = (*b1*,*b2*,*b3*,*b4*) ⟹ *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-inδD*: *op* ∈ *set* (*ast-problem.astδ* (*rem-implicit-pres-ops prob*))
$\Longrightarrow$ (∃ *op′*. *op′* ∈ *set* (*ast-problem.astδ prob*) ∧ *op* = *rem-implicit-pres op′*)
⟨*proof*⟩

lemma *rem-implicit-pres-ops-well-formed*:
assumes (⋀*op*. *op* ∈ *set* (*ast-problem.astδ prob*) $\Longrightarrow$ *consistent-pres-op op*)
ast-problem.well-formed prob
shows *ast-problem.well-formed* (*rem-implicit-pres-ops prob*)
⟨*proof*⟩

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

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

lemma *rem-implicit-pres-is-standard-operator′*:
is-standard-operator (*n*,*p*,*es*,*c*) $\Longrightarrow$ *is-standard-operator′* (*rem-implicit-pres* (*n*,*p*,*es*,*c*))
⟨*proof*⟩

lemma *rem-implicit-pres-ops-is-standard-operator′*:
(⋀*op*. *op* ∈ *set* (*ast-problem.astδ* (*vs*, *I*, *G*, *ops*)) $\Longrightarrow$ *is-standard-operator op*)
$\Longrightarrow$
π∈*set* (*ast-problem.astδ* (*rem-implicit-pres-ops* (*vs*, *I*, *G*, *ops*))) $\Longrightarrow$ *is-standard-operator′* π
⟨*proof*⟩

locale *abs-ast-prob* = *wf-ast-problem* +
assumes *no-cond-effs*: ∀ π∈*set astδ*. *is-standard-operator′* π

context *ast-problem*
begin

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

definition *abs-range-map*
:: (*nat* ⇀ *nat list*)
where *abs-range-map* ≡
map-of (*zip abs-ast-variable-section*
(*map* ((λ*vals*. [*0*..<*length vals*]) *o snd o snd*)
astDom))

end

context *abs-ast-prob*
begin

lemma *is-valid-vars-1*: *astDom* ≠ [] ⟹ *abs-ast-variable-section* ≠ []
⟨*proof*⟩

end

lemma *upt-eq-Nil-conv′*[*simp*]: ([] = [*i*..<*j*]) = (*j* = *0* ∨ *j* ≤ *i*)
⟨*proof*⟩

lemma *map-of-zip-map-Some*:
v < *length xs*
⟹ (*map-of* (*zip* [*0*..<*length xs*] (*map f xs*)) *v*) = *Some* (*f* (*xs* ! *v*))
⟨*proof*⟩

lemma *map-of-zip-Some*:
v < *length xs*
⟹ (*map-of* (*zip* [*0*..<*length xs*] *xs*) *v*) = *Some* (*xs* ! *v*)
⟨*proof*⟩

lemma *in-set-zip-lengthE*:
(*x*,*y*) ∈ *set*(*zip* [*0*..<*length xs*] *xs*) ⟹ (⟦ *x* < *length xs*; *xs* ! *x* =*y* ⟧ ⟹ *R*) ⟹
R
⟨*proof*⟩

context *abs-ast-prob*
begin

lemma *is-valid-vars-2*:
shows *list-all* (λ*v*. *abs-range-map v* ≠ *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* ≡ *map-of* (*zip* [*0*..<*length astI*] *astI*)

end

context *abs-ast-prob*
begin

lemma *valid-abs-init-1*: *abs-ast-initial-state v* ≠ *None* ⟷ *v* ∈ *set abs-ast-variable-section*
⟨*proof*⟩

lemma *abs-range-map-Some*:
shows *v* ∈ *set abs-ast-variable-section* ⟹

(*abs-range-map v*) = *Some* [*0*..<*length* (*snd* (*snd* (*astDom* ! *v*)))]
⟨*proof*⟩

lemma *in-abs-v-sec-length*: *v* ∈ *set abs-ast-variable-section* ⟷ *v* < *length astDom*
⟨*proof*⟩

lemma [*simp*]: *v* < *length astDom* ⟹ (*abs-ast-initial-state v*) = *Some* (*astI* ! *v*)
⟨*proof*⟩

lemma [*simp*]: *v* < *length astDom* ⟹ *astI* ! *v* < *length* (*snd* (*snd* (*astDom* ! *v*)))
⟨*proof*⟩

lemma [*intro*!]: *v* ∈ *set abs-ast-variable-section* ⟹ *x* < *length* (*snd* (*snd* (*astDom* ! *v*))) ⟹
x ∈ *set* (*the* (*abs-range-map v*))
⟨*proof*⟩

lemma [*intro*!]: *x*<*length astDom* ⟹ *astI* ! *x* < *length* (*snd* (*snd* (*astDom* ! *x*)))
⟨*proof*⟩

lemma [*simp*]: *abs-ast-initial-state v* = *Some a* ⟹ *a* < *length* (*snd* (*snd* (*astDom* ! *v*)))
⟨*proof*⟩

lemma *valid-abs-init-2*:
abs-ast-initial-state v ≠ *None* ⟹ (*the* (*abs-ast-initial-state v*)) ∈ *set* (*the* (*abs-range-map v*))
⟨*proof*⟩

end

context *ast-problem*
begin

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

end

context *abs-ast-prob*
begin

lemma [*simp*]: *wf-partial-state s* ⟹ (*v*, *a*) ∈ *set s* ⟹ *v* ∈ *set abs-ast-variable-section*
⟨*proof*⟩

lemma *valid-abs-goal-1*: *abs-ast-goal v* ≠ *None* ⟹ *v* ∈ *set abs-ast-variable-section*
⟨*proof*⟩

lemma *in-abs-rangeI*: *wf-partial-state s* $\Longrightarrow$ (*v*, *a*) $\in$ *set s* $\Longrightarrow$ (*a* $\in$ *set* (*the* (*abs-range-map v*)))
⟨*proof*⟩

lemma *valid-abs-goal-2*:
abs-ast-goal v $\neq$ *None* $\Longrightarrow$ (*the* (*abs-ast-goal v*)) $\in$ *set* (*the* (*abs-range-map v*))
⟨*proof*⟩

end

context *ast-problem*
begin

definition *abs-ast-operator*
:: *ast-operator* $\Rightarrow$ *nat-sas-plus-operator*
where *abs-ast-operator* $\equiv$ λ(*name*, *preconditions*, *effects*, *cost*).
(| *precondition-of* = *preconditions*,
effect-of = [(*v*, *x*). (-, *v*, -, *x*) $\leftarrow$ *effects*] |)

end

context *abs-ast-prob*
begin

lemma *abs-rangeI*: *wf-partial-state s* $\Longrightarrow$ (*v*, *a*) $\in$ *set s* $\Longrightarrow$ (*abs-range-map v* $\neq$ *None*)
⟨*proof*⟩

lemma *abs-valid-operator-1*[*intro*!]:
wf-operator op $\Longrightarrow$ *list-all* (λ(*v*, *a*). *ListMem v abs-ast-variable-section*)
(*precondition-of* (*abs-ast-operator op*))
⟨*proof*⟩

lemma *wf-operator-preD*: *wf-operator* (*name*, *pres*, *effs*, *cost*) $\Longrightarrow$ *wf-partial-state pres*
⟨*proof*⟩

lemma *abs-valid-operator-2*[*intro*!]:
wf-operator op $\Longrightarrow$
list-all (λ(*v*, *a*). ($\exists$ *y*. *abs-range-map v* = *Some y*) $\wedge$ *ListMem a* (*the* (*abs-range-map v*)))
(*precondition-of* (*abs-ast-operator op*))
⟨*proof*⟩

lemma *wf-operator-effE*: *wf-operator* (*name*, *pres*, *effs*, *cost*) $\Longrightarrow$
([[*distinct* (*map* (λ(-, *v*, -, -). *v*) *effs*);
$\bigwedge$*epres x vp v*. (*epres*,*x*,*vp*,*v*)$\in$*set effs* $\Longrightarrow$ *wf-partial-state epres*;
$\bigwedge$*epres x vp v*.(*epres*,*x*,*vp*,*v*)$\in$*set effs* $\Longrightarrow$ *x* < *numVars*;
$\bigwedge$*epres x vp v*. (*epres*,*x*,*vp*,*v*)$\in$*set effs* $\Longrightarrow$ *v* < *numVals x*;

$\bigwedge$*epres x vp v.* (*epres*,*x*,*vp*,*v*)∈*set effs* ⟹
case vp of None ⇒ *True* | *Some v* ⇒ *v*<*numVals x*⟧
⟹ *P*)
⟹ *P*
⟨*proof*⟩

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

lemma *abs-valid-operator-3*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). *ListMem v abs-ast-variable-section*) (*effect-of* (*abs-ast-operator op*))
⟨*proof*⟩

lemma *wf-abs-eff*: *wf-operator* (*name*, *pre*, *eff*, *cost*) ⟹ *wf-partial-state* (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
⟨*proof*⟩

lemma *abs-valid-operator-4'*:
wf-operator (*name*, *pre*, *eff*, *cost*) ⟹
list-all (λ(*v*, *a*). (*abs-range-map v* ≠ *None*) ∧ *ListMem a* (*the* (*abs-range-map v*))) (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
⟨*proof*⟩

lemma *abs-valid-operator-4*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). (∃ *y*. *abs-range-map v* = *Some y*) ∧ *ListMem a* (*the* (*abs-range-map v*)))
(*effect-of* (*abs-ast-operator op*))
⟨*proof*⟩

lemma *consistent-list-set*: *wf-partial-state s* ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) *s*) *s*
⟨*proof*⟩

lemma *abs-valid-operator-5'*:
wf-operator (*name*, *pre*, *eff*, *cost*) ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) *pre*) *pre*
⟨*proof*⟩

lemma *abs-valid-operator-5*[*intro*!]:
wf-operator op ⟹
list-all (λ(*v*, *a*). *list-all* (λ(*v'*, *a'*). *v* ≠ *v'* ∨ *a* = *a'*) (*precondition-of* (*abs-ast-operator op*)))
(*precondition-of* (*abs-ast-operator op*))

⟨*proof*⟩

lemma *consistent-list-set-2*: *distinct* (*map fst s*) $\Longrightarrow$
list-all ($\lambda(v, a)$. *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* ($\lambda(v, a)$. *list-all* ($\lambda(v', a')$. $v \neq v' \vee a = a'$) (*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*))
(*map* (λ(-, *v*, -, *a*). (*v*, *a*)) *eff*)
⟨*proof*⟩

lemma *abs-valid-operator-6*[*intro*!]:
wf-operator op $\Longrightarrow$
list-all ($\lambda(v, a)$. *list-all* ($\lambda(v', a')$. $v \neq v' \vee a = a'$) (*effect-of* (*abs-ast-operator op*)))
(*effect-of* (*abs-ast-operator op*))
⟨*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*δ $\Longrightarrow$ (*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-Semantics.plan* ⇒ *nat-sas-plus-plan*
where *abs-ast-plan πs*
≡ *map* (*abs-ast-operator o the o lookup-operator*) *πs*

lemma *std-then-implici-effs*[*simp*]: *is-standard-operator′* (*name*, *pres*, *effs*, *layer*) ⟹ *implicit-pres effs* = []
⟨*proof*⟩

lemma [*simp*]: *enabled π s* ⟹ *lookup-operator π* = *Some* (*name*, *pres*, *effs*, *layer*) ⟹
is-standard-operator′ (*name*, *pres*, *effs*, *layer*) ⟹
(*filter* (*eff-enabled s*) *effs*) = *effs*
⟨*proof*⟩

lemma *effs-eq-abs-effs*: (*effect-of* (*abs-ast-operator* (*name*, *pres*, *effs*, *layer*))) =
(*map* (λ(-,*x*,-,*v*). (*x*,*v*)) *effs*)
⟨*proof*⟩

lemma *exect-eq-abs-execute*:
⟦*enabled π s*; *lookup-operator π* = *Some* (*name*, *preconds*, *effs*, *layer*);
is-standard-operator′(*name*, *preconds*, *effs*, *layer*)⟧ ⟹
execute π s = (*execute-operator-sas-plus s* ((*abs-ast-operator o the o lookup-operator*) *π*))
⟨*proof*⟩

lemma *enabled-then-sas-applicable*:
enabled π s ⟹ *SAS-Plus-Representation.is-operator-applicable-in s* ((*abs-ast-operator o the o lookup-operator*) *π*)
⟨*proof*⟩

lemma *path-to-then-exec-serial*: ∀ *π*∈*set πs*. *lookup-operator π* ≠ *None* ⟹
path-to s πs s′ ⟹
s′ $\subseteq_m$ *execute-serial-plan-sas-plus s* (*abs-ast-plan πs*)
⟨*proof*⟩

lemma *map-of-eq-None-iff*:
(*None* = *map-of xys x*) = (*x* ∉ *fst* ‘ (*set xys*))
⟨*proof*⟩

lemma [*simp*]: *I* = *abs-ast-initial-state*
⟨*proof*⟩

lemma [*simp*]: ∀ *π* ∈ *set πs*. *lookup-operator π* ≠ *None* ⟹

op∈*set* (*abs-ast-plan* πs) $\Longrightarrow$ *op* ∈ *set abs-ast-operator-section*
⟨*proof*⟩

end

context *ast-problem*
begin

lemma *path-to-then-lookup-Some*: (∃ s'∈*G*. *path-to* *s* πs s') $\Longrightarrow$ (∀ π ∈ *set* πs. *lookup-operator* $\pi \neq$ *None*)
⟨*proof*⟩

lemma *valid-plan-then-lookup-Some*: *valid-plan* πs $\Longrightarrow$ (∀ π ∈ *set* πs. *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+ represnetation to Fast-Downward's

context *ast-problem*
begin

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

end

context *abs-ast-prob*
begin

lemma *find-Some*: *find P xs* = *Some x* $\Longrightarrow$ *x* ∈ *set xs* ∧ *P x*
⟨*proof*⟩

lemma *distinct-find*: *distinct* (*map f xs*) $\Longrightarrow$ *x* ∈ *set xs* $\Longrightarrow$ *find* (λx'. *f* x' = *f x*) *xs* = *Some x*

⟨*proof*⟩

lemma *lookup-operator-find*: *lookup-operator nme* = *find* (λ*op*. *fst op* = *nme*) *astδ*
⟨*proof*⟩

lemma *lookup-operator-works-1*: *lookup-action op* = *Some* π′ ⟹ *lookup-operator* (*fst* π′) = *Some* π′
⟨*proof*⟩

lemma *lookup-operator-works-2*:
lookup-action (*abs-ast-operator* (*name*, *pres*, *effs*, *layer*)) = *Some* (*name*′, *pres*′, *effs*′, *layer*′)
⟹ *pres* = *pres*′
⟨*proof*⟩

lemma [*simp*]: *is-standard-operator*′ (*name*, *pres*, *effs*, *layer*) ⟹
map (λ(*v*,*a*). ([], *v*, *None*, *a*)) (*effect-of* (*abs-ast-operator* (*name*, *pres*, *effs*, *layer*))) = *effs*
⟨*proof*⟩

lemma *lookup-operator-works-3*:
is-standard-operator′ (*name*, *pres*, *effs*, *layer*) ⟹ (*name*, *pres*, *effs*, *layer*) ∈ *set astδ* ⟹
lookup-action (*abs-ast-operator* (*name*, *pres*, *effs*, *layer*)) = *Some* (*name*′, *pres*′, *effs*′, *layer*′)
⟹ *effs* = *effs*′
⟨*proof*⟩

lemma *mem-find-Some*: *x* ∈ *set xs* ⟹ *P x* ⟹ ∃ *x*′. *find P xs* = *Some x*′
⟨*proof*⟩

lemma [*simp*]: *precondition-of* (*abs-ast-operator* (*x1*, *a*, *aa*, *b*)) = *a*
⟨*proof*⟩

lemma *std-lookup-action*: *is-standard-operator*′ *ast-op* ⟹ *ast-op* ∈ *set astδ* ⟹
∃ *ast-op*′. *lookup-action* (*abs-ast-operator ast-op*) = *Some ast-op*′
⟨*proof*⟩

lemma *is-applicable-then-enabled-1*:
ast-op ∈ *set astδ* ⟹
∃ *ast-op*′. *lookup-operator* ((*fst o the o lookup-action o abs-ast-operator*) *ast-op*) = *Some ast-op*′
⟨*proof*⟩

lemma *lookup-action-Some-in-δ*: *lookup-action op* = *Some ast-op* ⟹ *ast-op* ∈ *set astδ*
⟨*proof*⟩

lemma *lookup-operator-eq-name*: *lookup-operator name* = *Some* (*name′*, *pres*, *effs*, *layer*) ⟹ *name* = *name′*
⟨*proof*⟩

lemma *eq-name-eq-pres*: (*name*, *pres*, *effs*, *layer*) ∈ *set astδ* ⟹ (*name*, *pres′*, *effs′*, *layer′*) ∈ *set astδ*
⟹ *pres* = *pres′*
⟨*proof*⟩

lemma *eq-name-eq-effs*:
name = *name′* ⟹ (*name*, *pres*, *effs*, *layer*) ∈ *set astδ* ⟹ (*name′*, *pres′*, *effs′*, *layer′*) ∈ *set astδ*
⟹ *effs* = *effs′*
⟨*proof*⟩

lemma *is-applicable-then-subsumes*:
s ∈ *valid-states* ⟹
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator* (*name*, *pres*, *effs*, *layer*)) ⟹
s ∈ *subsuming-states* (*map-of pres*)
⟨*proof*⟩

lemma *eq-name-eq-pres′*:
⟦*s* ∈ *valid-states* ; *is-standard-operator′* (*name*, *pres*, *effs*, *layer*); (*name*, *pres*, *effs*, *layer*) ∈ *set astδ* ;
lookup-operator ((*fst o the o lookup-action o abs-ast-operator*) (*name*, *pres*, *effs*, *layer*)) = *Some* (*name′*, *pres′*, *effs′*, *layer′*)⟧
⟹ *pres* = *pres′*
⟨*proof*⟩

lemma *is-applicable-then-enabled-2*:
⟦*s* ∈ *valid-states* ; *ast-op* ∈ *set astδ* ;
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator ast-op*);
lookup-operator ((*fst o the o lookup-action o abs-ast-operator*) *ast-op*) = *Some* (*name*, *pres*, *effs*, *layer*)⟧
⟹ *s*∈*subsuming-states* (*map-of pres*)
⟨*proof*⟩

lemma *is-applicable-then-enabled-3*:
⟦*s* ∈ *valid-states*;
lookup-operator ((*fst o the o lookup-action o abs-ast-operator*) *ast-op*) = *Some* (*name*, *pres*, *effs*, *layer*)⟧
⟹ *s*∈*subsuming-states* (*map-of* (*implicit-pres effs*))
⟨*proof*⟩

lemma *is-applicable-then-enabled*:
⟦*s* ∈ *valid-states*; *ast-op* ∈ *set astδ*;
SAS-Plus-Representation.is-operator-applicable-in s (*abs-ast-operator ast-op*)⟧
⟹ *enabled* ((*fst o the o lookup-action o abs-ast-operator*) *ast-op*) *s*

⟨*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-consdless*: *execute-serial-plan-sas-plus s* (*rem-condless-ops s ops*) = *execute-serial-plan-sas-plus s ops*
⟨*proof*⟩

lemma *rem-conless-sat*: *sat-preconds-as s* (*rem-condless-ops s ops*)
⟨*proof*⟩

lemma *set-rem-condlessD*: $x \in$ *set* (*rem-condless-ops s ops*) $\Longrightarrow x \in$ *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

context *abs-ast-prob*
begin

theorem *is-serial-sol-then-valid-plan*:
$[\![$*is-serial-solution-for-problem abs-prob ops*$]\!] \Longrightarrow$
valid-plan (*decode-abs-plan ops*)
$\langle proof \rangle$

end

end

$\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle proof \rangle\langle p$

theory *Solve-SASP*
imports *AST-SAS-Plus-Equivalence SAT-Solve-SAS-Plus*
HOL−Data-Structures.RBT-Map HOL−Library.Code-Target-Nat HOL.String
AI-Planning-Languages-Semantics.SASP-Checker Set2-Join-RBT
begin

11.3 SAT encoding works for Fast-Downward's representation

context *abs-ast-prob*
begin

theorem *is-serial-sol-then-valid-plan-encoded*:
$\mathcal{A} \models \Phi_\forall$ (φ (*prob-with-noop abs-prob*)) $t \Longrightarrow$
valid-plan
(*decode-abs-plan*
(*rem-noops*
(*map* ($\lambda op.$ ${\varphi_O}^{-1}$ (*prob-with-noop abs-prob*) *op*)
(*concat* (Φ^{-1} (φ (*prob-with-noop abs-prob*)) $\mathcal{A}$ t)))))
$\langle proof \rangle$

lemma *length-abs-ast-plan*: *length* πs = *length* (*abs-ast-plan* πs)
$\langle proof \rangle$

theorem *valid-plan-then-is-serial-sol-encoded*:
valid-plan $\pi s \Longrightarrow$ *length* $\pi s \leq h \Longrightarrow \exists \mathcal{A}.\ \mathcal{A} \models \Phi_\forall$ (φ (*prob-with-noop abs-prob*))
h
$\langle proof \rangle$
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) ⟹ (∃ m. n = Suc m ∧ l ≤ m)
  ⟨proof⟩

locale cnf-to-dimacs =
  fixes h :: nat and n-ops :: nat
begin

fun var-to-dimacs where
  var-to-dimacs (Operator t k) = 1 + t + k ∗ h
| var-to-dimacs (State t k) = 1 + n-ops ∗ h + t + k ∗ (h)

definition dimacs-to-var where
  dimacs-to-var v ≡
    if v < 1 + n-ops ∗ h then
      Operator ((v − 1) mod (h)) ((v − 1) div (h))
    else
      (let k = ((v − 1) − n-ops ∗ h) in
        State (k mod (h)) (k div (h)))

fun valid-state-var where
  valid-state-var (Operator t k) ⟷ t < h ∧ k < n-ops
| valid-state-var (State t k) ⟷ t < h

lemma State-works:
valid-state-var (State t k) ⟹
      dimacs-to-var (var-to-dimacs (State t k)) =
        (State t k)
  ⟨proof⟩

lemma Operator-works:
   valid-state-var (Operator t k) ⟹
      dimacs-to-var (var-to-dimacs (Operator t k)) =
        (Operator t k)
  ⟨proof⟩

lemma sat-plan-to-dimacs-works:
  valid-state-var sv ⟹
     dimacs-to-var (var-to-dimacs sv) = sv
  ⟨proof⟩

end

lemma changing-atoms-works:
```

$(\bigwedge x.\ P\ x \Longrightarrow (f\ o\ g)\ x = x) \Longrightarrow (\forall x{\in}atoms\ phi.\ P\ x) \Longrightarrow M \models phi \longleftrightarrow M\ o\ f \models map\text{-}formula\ g\ phi$
⟨*proof*⟩

lemma *changing-atoms-works'*:
$M\ o\ g \models phi \longleftrightarrow M \models map\text{-}formula\ g\ phi$
⟨*proof*⟩

context *cnf-to-dimacs*
begin

lemma *sat-plan-to-dimacs*:
$(\bigwedge sv.\ sv{\in}atoms\ sat\text{-}plan\text{-}formula \Longrightarrow valid\text{-}state\text{-}var\ sv) \Longrightarrow$
$M \models sat\text{-}plan\text{-}formula$
$\longleftrightarrow M\ o\ dimacs\text{-}to\text{-}var \models map\text{-}formula\ var\text{-}to\text{-}dimacs\ sat\text{-}plan\text{-}formula$
⟨*proof*⟩

lemma *dimacs-to-sat-plan*:
$M\ o\ var\text{-}to\text{-}dimacs \models sat\text{-}plan\text{-}formula$
$\longleftrightarrow M \models map\text{-}formula\ var\text{-}to\text{-}dimacs\ sat\text{-}plan\text{-}formula$
⟨*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*:
$sv \in atoms\ (encode\text{-}initial\text{-}state\ Prob) \Longrightarrow valid\text{-}state\text{-}var\ sv$
⟨*proof*⟩

lemma *length-operators*: $length\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))) = Suc\ (length\ ast\delta)$
⟨*proof*⟩

lemma *encode-operator-effect-valid-1*: $t < h \Longrightarrow op \in set\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))) \Longrightarrow$
$sv \in atoms$
$(\bigwedge(map\ (\lambda v.$
$\neg(Atom\ (Operator\ t\ (index\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob)))\ op)))$
$\lor\ Atom\ (State\ (Suc\ t)\ (index\ vs\ v)))$
$asses)) \Longrightarrow$
$valid\text{-}state\text{-}var\ sv$
⟨*proof*⟩

lemma *encode-operator-effect-valid-2*: $t < h \Longrightarrow op \in set\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop$

abs-prob))) $\Longrightarrow$
sv $\in$ *atoms*
($\bigwedge$(*map* (λ*v*.
$\neg$(*Atom* (*Operator t* (*index* (*operators-of* (φ (*prob-with-noop abs-prob*)))
op)))
$\vee$ $\neg$ (*Atom* (*State* (*Suc t*) (*index vs v*))))
asses)) $\Longrightarrow$
valid-state-var sv
⟨*proof*⟩

end

lemma *atoms-And-append*: *atoms* ($\bigwedge$ (*as1* @ *as2*)) = *atoms* ($\bigwedge$ *as1*) $\cup$ *atoms*
($\bigwedge$ *as2*)
⟨*proof*⟩

context *sat-solve-sasp*
begin

lemma *encode-operator-effect-valid*:
sv $\in$ *atoms* (*encode-operator-effect* (φ (*prob-with-noop abs-prob*)) *t op*) $\Longrightarrow$
t < *h* $\Longrightarrow$ *op* $\in$ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) $\Longrightarrow$
valid-state-var sv
⟨*proof*⟩

end

lemma *foldr-And*: *foldr* ($\wedge$) *as* ($\neg$ $\bot$) = ($\bigwedge$ *as*)
⟨*proof*⟩

context *sat-solve-sasp*
begin

lemma *encode-all-operator-effects-valid*:
t < *Suc h* $\Longrightarrow$
sv $\in$ *atoms* (*encode-all-operator-effects* (φ (*prob-with-noop abs-prob*)) (*operators-of*
(φ (*prob-with-noop abs-prob*))) *t*) $\Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-operator-precondition-valid-1*:
t < *h* $\Longrightarrow$ *op* $\in$ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) $\Longrightarrow$
sv $\in$ *atoms*
($\bigwedge$(*map* (λ*v*.
$\neg$ (*Atom* (*Operator t* (*index* (*operators-of* (φ (*prob-with-noop abs-prob*)))
op))) $\vee$ *Atom* (*State t* (*f v*)))
asses)) $\Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-operator-precondition-valid*:
$sv \in atoms\ (encode\text{-}operator\text{-}precondition\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t\ op) \Longrightarrow$

$t < h \Longrightarrow op \in set\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))) \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-all-operator-preconditions-valid*:
$t < Suc\ h \Longrightarrow$
$sv \in atoms\ (encode\text{-}all\text{-}operator\text{-}preconditions\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob)))\ t) \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-operators-valid*:
$sv \in atoms\ (encode\text{-}operators\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t) \Longrightarrow t < Suc\ h \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-negative-transition-frame-axiom′*:
$t < h \Longrightarrow$
$set\ deleting\text{-}operators \subseteq set\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))) \Longrightarrow$
$sv \in atoms$
$(\neg(Atom\ (State\ t\ v\text{-}idx))$
$\lor (Atom\ (State\ (Suc\ t)\ v\text{-}idx)$
$\lor \bigvee (map\ (\lambda op.\ Atom\ (Operator\ t\ (index\ (operators\text{-}of\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob)))\ op)))$
$deleting\text{-}operators))) \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-negative-transition-frame-axiom-valid*:
$sv \in atoms\ (encode\text{-}negative\text{-}transition\text{-}frame\text{-}axiom\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t\ v) \Longrightarrow t < h \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-positive-transition-frame-axiom-valid*:
$sv \in atoms\ (encode\text{-}positive\text{-}transition\text{-}frame\text{-}axiom\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t\ v) \Longrightarrow t < h \Longrightarrow$
valid-state-var sv
⟨*proof*⟩

lemma *encode-all-frame-axioms-valid*:
$sv \in atoms\ (encode\text{-}all\text{-}frame\text{-}axioms\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ t) \Longrightarrow t < Suc\ h \Longrightarrow$
valid-state-var sv

⟨proof⟩

lemma *encode-goal-state-valid*:
sv ∈ *atoms* (*encode-goal-state Prob t*) ⟹ *t* < *Suc h* ⟹ *valid-state-var sv*
⟨proof⟩

lemma *encode-problem-valid*:
sv ∈ *atoms* (*encode-problem* (φ (*prob-with-noop abs-prob*)) *h*) ⟹ *valid-state-var sv*
⟨proof⟩

lemma *encode-interfering-operator-pair-exclusion-valid*:
sv ∈ *atoms* (*encode-interfering-operator-pair-exclusion* (φ (*prob-with-noop abs-prob*)) *t* op_1 op_2) ⟹ *t* < *Suc h* ⟹
op_1 ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹ op_2 ∈ *set* (*operators-of* (φ (*prob-with-noop abs-prob*))) ⟹
valid-state-var sv
⟨proof⟩

lemma *encode-interfering-operator-exclusion-valid*:
sv ∈ *atoms* (*encode-interfering-operator-exclusion* (φ (*prob-with-noop abs-prob*)) *t*) ⟹ *t* < *Suc h* ⟹
valid-state-var sv
⟨proof⟩

lemma *encode-problem-with-operator-interference-exclusion-valid*:
sv ∈ *atoms* (*encode-problem-with-operator-interference-exclusion* (φ (*prob-with-noop abs-prob*)) *h*) ⟹ *valid-state-var sv*
⟨proof⟩

lemma *planning-by-cnf-dimacs-complete*:
valid-plan πs ⟹ *length* $\pi s \leq h$ ⟹
∃ *M*. *M* ⊨ *map-formula var-to-dimacs* ($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *h*)
⟨proof⟩

lemma *planning-by-cnf-dimacs-sound*:
$\mathcal{A}$ ⊨ *map-formula var-to-dimacs* ($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *t*) ⟹
valid-plan
(*decode-abs-plan*
(*rem-noops*
(*map* ($\lambda op.$ ${\varphi_O}^{-1}$ (*prob-with-noop abs-prob*) *op*)
(*concat* (Φ^{-1} (φ (*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* $\bot$ = []
| *disj-to-dimacs* (*Not* $\bot$) = [$-$*1*::*int*,*1*::*int*]
| *disj-to-dimacs* (*Atom v*) = [*int v*]
| *disj-to-dimacs* (*Not* (*Atom v*)) = [$-$(*int v*)]

fun *cnf-to-dimacs*::*nat formula* $\Rightarrow$ *int list list* **where**
  cnf-to-dimacs $(\varphi_1 \wedge \varphi_2)$ = *cnf-to-dimacs* φ_1 @ *cnf-to-dimacs* φ_2
| *cnf-to-dimacs d* = [*disj-to-dimacs d*]

definition *dimacs-lit-to-var l* $\equiv$ *nat* (*abs l*)

definition *find-max* (*xs*::*nat list*)$\equiv$ (*fold max xs 1*)

lemma *find-max-works*:
$x \in$ *set xs* $\Longrightarrow x \leq$ *find-max xs* (**is** *?P* $\Longrightarrow$ *?Q*)
⟨*proof*⟩

fun *formula-vars* **where**
formula-vars ($\bot$) = [] |
formula-vars (*Atom k*) = [*k*] |
formula-vars (*Not F*) = *formula-vars F* |
formula-vars (*And F G*) = *formula-vars F* @ *formula-vars G* |
formula-vars (*Imp F G*) = *formula-vars F* @ *formula-vars G* |
formula-vars (*Or F G*) = *formula-vars F* @ *formula-vars G*

lemma *atoms-formula-vars*: *atoms f* = *set* (*formula-vars f*)
  ⟨*proof*⟩

lemma *max-var*: $v \in$ *atoms* (*f*::*nat formula*) $\Longrightarrow v \leq$ *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*: $b \leq a \Longrightarrow$ (*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 ≡ (∀ *c*∈*set cs*. (∃ *l*∈*set ls*. *l* ∈ *set c*)) ∧
distinct (*map dimacs-lit-to-var ls*)

fun *model-to-dimacs-model* **where**
model-to-dimacs-model M (*v*#*vs*) = (*if M v then int v else* − (*int v*)) # (*model-to-dimacs-model M vs*)
| *model-to-dimacs-model* - [] = []

lemma *model-to-dimacs-model-append*:
set (*model-to-dimacs-model M* (*vs* @ *vs′*)) = *set* (*model-to-dimacs-model M vs*) ∪ *set* (*model-to-dimacs-model M vs′*)
⟨*proof*⟩

lemma *upt-append-sing*: *xs* @ [*x*] = [*a*..<*n-vars*] ⟹ *a* < *n-vars* ⟹ (*xs* = [*a*..<*n-vars* − *1*] ∧ *x* = *n-vars*−*1* ∧ *n-vars* > *0*)
⟨*proof*⟩

lemma *upt-eqD*: *upt a b* = *upt a b′* ⟹ (*b* = *b′* ∨ *b′* ≤ *a* ∨ *b* ≤ *a*)
⟨*proof*⟩

lemma *pos-in-model*: *M n* ⟹ *0* < *n* ⟹ *n* < *n-vars* ⟹ *int n* ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*])
⟨*proof*⟩

lemma *neg-in-model*: ¬ *M n* ⟹ *0* < *n* ⟹ *n* < *n-vars* ⟹ − (*int n*) ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*])
⟨*proof*⟩

lemma *in-model*: *0* < *n* ⟹ *n* < *n-vars* ⟹ *int n* ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*]) ∨ − (*int n*) ∈ *set* (*model-to-dimacs-model M* [*1*..<*n-vars*])
⟨*proof*⟩

lemma *model-to-dimacs-model-all-vars*:
(∀ *v*∈*atoms f*. *0* < *v* ∧ *v* < *n-vars*) ⟹ *is-cnf f* ⟹ *M* ⊨ *f* ⟹
(∀ *n*<*n-vars*. *0* < *n* ⟶ (*int n* ∈ *set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]) ∨
−(*int n*) ∈ *set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*])))
⟨*proof*⟩

lemma *cnf-And*: *set* (*cnf-to-dimacs* (*f1* ∧ *f2*)) = *set* (*cnf-to-dimacs f1*) ∪ *set* (*cnf-to-dimacs f2*)
⟨*proof*⟩

lemma *one-always-in*:
1 < *n-vars* ⟹ *1* ∈ *set* (*model-to-dimacs-model M* ([*1*..<*n-vars*])) ∨ − *1* ∈ *set* (*model-to-dimacs-model M* ([*1*..<*n-vars*]))
⟨*proof*⟩

lemma [*simp*]: (*disj-to-dimacs* (*f1* ∨ *f2*)) = (*disj-to-dimacs f1*) @ (*disj-to-dimacs*

f2)
⟨*proof*⟩

lemma [*simp*]: (*atoms* (*f1* ∨ *f2*)) = *atoms f1* ∪ *atoms f2*
⟨*proof*⟩

lemma *isdisj-disjD*: (*is-disj* (*f1* ∨ *f2*)) ⟹ *is-disj f1* ∧ *is-disj f2*
⟨*proof*⟩

lemma *disj-to-dimacs-sound*:
1 < *n-vars* ⟹ (∀ *v*∈*atoms f*. *0* < *v* ∧ *v* < *n-vars*) ⟹ *is-disj f* ⟹ *M* ⊨ *f*
⟹ ∃ *l*∈*set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]). *l* ∈ *set* (*disj-to-dimacs f*)
⟨*proof*⟩

lemma *is-cnf-disj*: *is-cnf* (*f1* ∨ *f2*) ⟹ (⋀*f*. *f1* ∨ *f2* = *f* ⟹ *is-disj f* ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *cnf-to-dimacs-disj*: *is-disj f* ⟹ *cnf-to-dimacs f* = [*disj-to-dimacs f*]
⟨*proof*⟩

lemma *model-to-dimacs-model-all-clauses*:
1 < *n-vars* ⟹ (∀ *v*∈*atoms f*. *0* < *v* ∧ *v* < *n-vars*) ⟹ *is-cnf f* ⟹ *M* ⊨ *f* ⟹
c∈*set* (*cnf-to-dimacs f*) ⟹ ∃ *l*∈*set* (*model-to-dimacs-model M* [(*1*::*nat*)..<*n-vars*]). *l* ∈ *set c*
⟨*proof*⟩

lemma *upt-eq-Cons-conv*:
(*x*#*xs* = [*i*..<*j*]) = (*i* < *j* ∧ *i* = *x* ∧ [*i*+*1*..<*j*] = *xs*)
⟨*proof*⟩

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′*)
⟨*proof*⟩

lemma *model-to-dimacs-neg-nin*:
n-vars ≤ *x* ⟹ *int x* ∉ *set* (*model-to-dimacs-model M* [*a*..<*n-vars*])
⟨*proof*⟩

lemma *model-to-dimacs-pos-nin*:
n-vars ≤ *x* ⟹ − *int x* ∉ *set* (*model-to-dimacs-model M* [*a*..<*n-vars*])
⟨*proof*⟩

lemma *int-cases2′*:
z ≠ *0* ⟹ (⋀*n*. *0* ≠ (*int n*) ⟹ *z* = *int n* ⟹ *P*) ⟹ (⋀*n*. *0* ≠ − (*int n*) ⟹ *z* = − (*int n*) ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *model-to-dimacs-model-distinct*:
$1 < n\text{-}vars \Longrightarrow distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var\ (model\text{-}to\text{-}dimacs\text{-}model\ M\ [1..<n\text{-}vars]))$
⟨*proof*⟩

lemma *model-to-dimacs-model-sound*:
$1 < n\text{-}vars \Longrightarrow (\forall v \in atoms\ f.\ 0 < v \land v < n\text{-}vars) \Longrightarrow is\text{-}cnf\ f \Longrightarrow M \models f \Longrightarrow dimacs\text{-}model\ (model\text{-}to\text{-}dimacs\text{-}model\ M\ [(1::nat)..<n\text{-}vars])\ (cnf\text{-}to\text{-}dimacs\ f)$
⟨*proof*⟩

lemma *model-to-dimacs-model-sound-exists*:
$1 < n\text{-}vars \Longrightarrow (\forall v \in atoms\ f.\ 0 < v \land v < n\text{-}vars) \Longrightarrow is\text{-}cnf\ f \Longrightarrow M \models f \Longrightarrow \exists M\text{-}dimacs.\ dimacs\text{-}model\ M\text{-}dimacs\ (cnf\text{-}to\text{-}dimacs\ f)$
⟨*proof*⟩

definition *dimacs-to-atom* ::*int* ⇒ *nat formula* **where**
$dimacs\text{-}to\text{-}atom\ l \equiv if\ (l < 0)\ then\ Not\ (Atom\ (nat\ (abs\ l)))\ else\ Atom\ (nat\ (abs\ l))$

definition *dimacs-to-disj*::*int list* ⇒ *nat formula* **where**
$dimacs\text{-}to\text{-}disj\ f \equiv \bigvee\ (map\ dimacs\text{-}to\text{-}atom\ f)$

definition *dimacs-to-cnf*::*int list list* ⇒ *nat formula* **where**
$dimacs\text{-}to\text{-}cnf\ f \equiv \bigwedge map\ dimacs\text{-}to\text{-}disj\ f$

definition *dimacs-model-to-abs dimacs-M M* ≡
$fold\ (\lambda l\ M.\ if\ (l > 0)\ then\ M((nat\ (abs\ l)) := True)\ else\ M((nat\ (abs\ l)) := False))\ dimacs\text{-}M\ M$

lemma *dimacs-model-to-abs-atom*:
$0 < x \Longrightarrow int\ x \in set\ dimacs\text{-}M \Longrightarrow distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var\ \ dimacs\text{-}M) \Longrightarrow dimacs\text{-}model\text{-}to\text{-}abs\ dimacs\text{-}M\ M\ x$
⟨*proof*⟩

lemma *dimacs-model-to-abs-atom'*:
$0 < x \Longrightarrow -(int\ x) \in set\ dimacs\text{-}M \Longrightarrow distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var\ \ dimacs\text{-}M) \Longrightarrow \neg\ dimacs\text{-}model\text{-}to\text{-}abs\ dimacs\text{-}M\ M\ x$
⟨*proof*⟩

lemma *model-to-dimacs-model-complete-disj*:
$(\forall v \in atoms\ f.\ 0 < v \land v < n\text{-}vars) \Longrightarrow is\text{-}disj\ f \Longrightarrow distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var\ dimacs\text{-}M) \Longrightarrow dimacs\text{-}model\ dimacs\text{-}M\ (cnf\text{-}to\text{-}dimacs\ f) \Longrightarrow dimacs\text{-}model\text{-}to\text{-}abs\ dimacs\text{-}M\ (\lambda\text{-}.\ False) \models f$
⟨*proof*⟩

lemma *model-to-dimacs-model-complete*:
$(\forall v \in atoms\ f.\ 0 < v \land v < n\text{-}vars) \Longrightarrow is\text{-}cnf\ f \Longrightarrow distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var$

dimacs-M)
  ⟹ *dimacs-model dimacs-M* (*cnf-to-dimacs f*) ⟹ *dimacs-model-to-abs dimacs-M* (λ-. *False*) ⊨ *f*
⟨*proof*⟩

lemma *model-to-dimacs-model-complete-max-var*:
 (∀ *v*∈*atoms f*. *0* < *v*) ⟹ *is-cnf f* ⟹
  dimacs-model dimacs-M (*cnf-to-dimacs f*) ⟹
   dimacs-model-to-abs dimacs-M (λ-. *False*) ⊨ *f*
 ⟨*proof*⟩

lemma *model-to-dimacs-model-sound-max-var*:
 (∀ *v*∈*atoms f*. *0* < *v*) ⟹ *is-cnf f* ⟹ *M* ⊨ *f* ⟹
   dimacs-model (*model-to-dimacs-model M* [(*1*::*nat*)..<(*find-max* (*formula-vars f*) + *2*)])
     (*cnf-to-dimacs f*)
 ⟨*proof*⟩

context *sat-solve-sasp*
begin

lemma [*simp*]: *var-to-dimacs sv* > *0*
 ⟨*proof*⟩

lemma *var-to-dimacs-pos*:
 v ∈ *atoms* (*map-formula var-to-dimacs f*) ⟹ *0* < *v*
 ⟨*proof*⟩

lemma *map-is-disj*: *is-disj f* ⟹ *is-disj* (*map-formula F f*)
 ⟨*proof*⟩

lemma *map-is-cnf*: *is-cnf f* ⟹ *is-cnf* (*map-formula F f*)
 ⟨*proof*⟩

lemma *planning-dimacs-complete*:
 valid-plan π*s* ⟹ *length* π*s* ≤ *h* ⟹
  let cnf-formula = (*map-formula var-to-dimacs*
     ($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *h*))
  in
   ∃ *dimacs-M*. *dimacs-model dimacs-M* (*cnf-to-dimacs cnf-formula*)
 ⟨*proof*⟩

lemma *planning-dimacs-sound*:
 let cnf-formula =
   (*map-formula var-to-dimacs*
     ($\Phi_\forall$ (φ (*prob-with-noop abs-prob*)) *h*))
  in
   dimacs-model dimacs-M (*cnf-to-dimacs cnf-formula*) ⟹
  valid-plan

$(decode\text{-}abs\text{-}plan$
$(rem\text{-}noops$
$(map\ (\lambda op.\ \varphi_O{}^{-1}\ (prob\text{-}with\text{-}noop\ abs\text{-}prob)\ op)$
$(concat$
$(\Phi^{-1}\ (\varphi\ (prob\text{-}with\text{-}noop\ abs\text{-}prob))\ ((dimacs\text{-}model\text{-}to\text{-}abs\ dimacs\text{-}M$
$(\lambda\text{-}.\ False))\ o\ var\text{-}to\text{-}dimacs)\ h)))))$
$\langle proof \rangle$

end

13 Code Generation

We now generate SML code equivalent to the functions that encode a problem as a CNF formula and that decode the model of the given encodings into a plan.

lemma [*code*]:
$dimacs\text{-}model\ ls\ cs \equiv (list\text{-}all\ (\lambda c.\ list\text{-}ex\ (\lambda l.\ ListMem\ l\ c\ )\ ls)\ cs) \wedge$
$distinct\ (map\ dimacs\text{-}lit\text{-}to\text{-}var\ ls)$
$\langle proof \rangle$

definition
$SASP\text{-}to\text{-}DIMACS\ h\ prob \equiv$
$cnf\text{-}to\text{-}dimacs$
$(map\text{-}formula$
$(cnf\text{-}to\text{-}dimacs.var\text{-}to\text{-}dimacs\ (Suc\ h)\ (Suc\ (length\ (ast\text{-}problem.ast\delta\ prob))))$
$(\Phi_\forall\ (\varphi\ (prob\text{-}with\text{-}noop\ (ast\text{-}problem.abs\text{-}prob\ prob)))\ h))$

lemma *planning-dimacs-complete-code*:
$[\![ast\text{-}problem.well\text{-}formed\ prob;$
$\forall \pi \in set\ (ast\text{-}problem.ast\delta\ prob).\ is\text{-}standard\text{-}operator'\ \pi;$
$ast\text{-}problem.valid\text{-}plan\ prob\ \pi s;$
$length\ \pi s \leq h]\!] \Longrightarrow$
$let\ cnf\text{-}formula = (SASP\text{-}to\text{-}DIMACS\ h\ prob)\ in$
$\exists\, dimacs\text{-}M.\ dimacs\text{-}model\ dimacs\text{-}M\ cnf\text{-}formula$
$\langle proof \rangle$

definition $SASP\text{-}to\text{-}DIMACS'\ h\ prob \equiv SASP\text{-}to\text{-}DIMACS\ h\ (rem\text{-}implicit\text{-}pres\text{-}ops\ prob)$

lemma *planning-dimacs-complete-code'*:
$[\![ast\text{-}problem.well\text{-}formed\ prob;$
$(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow consistent\text{-}pres\text{-}op\ op);$
$(\bigwedge op.\ op \in set\ (ast\text{-}problem.ast\delta\ prob) \Longrightarrow is\text{-}standard\text{-}operator\ op);$
$ast\text{-}problem.valid\text{-}plan\ prob\ \pi s;$
$length\ \pi s \leq h]\!] \Longrightarrow$
$let\ cnf\text{-}formula = (SASP\text{-}to\text{-}DIMACS'\ h\ prob)\ in$
$\exists\, dimacs\text{-}M.\ dimacs\text{-}model\ dimacs\text{-}M\ cnf\text{-}formula$
$\langle proof \rangle$

A function that does the checks required by the completeness theorem above, and returns appropriate error messages if any of the checks fail.

definition
encode h prob ≡
if ast-problem.well-formed prob then
if (∀ *op* ∈ *set* (*ast-problem.astδ prob*). *consistent-pres-op op*) *then*
if (∀ *op* ∈ *set* (*ast-problem.astδ prob*). *is-standard-operator op*) *then*
Inl (*SASP-to-DIMACS′ h prob*)
else
Inr (*STR ″Error*: *Conditional effects*!″)
else
Inr (*STR ″Error*: *Preconditions inconsistent″*)
else
Inr (*STR ″Error*: *Problem malformed*!″)

lemma *encode-sound*:
⟦*ast-problem.valid-plan prob πs*; *length πs* ≤ *h*;
encode h prob = *Inl cnf-formula*⟧ ⟹
(∃ *dimacs-M*. *dimacs-model dimacs-M cnf-formula*)
⟨*proof*⟩

lemma *encode-complete*:
encode h prob = *Inr err* ⟹
¬(*ast-problem.well-formed prob* ∧ (∀ *op* ∈ *set* (*ast-problem.astδ prob*). *consistent-pres-op op*) ∧
(∀ *op* ∈ *set* (*ast-problem.astδ prob*). *is-standard-operator op*))
⟨*proof*⟩

definition *match-pre* **where**
match-pre ≡ λ(*x*,*v*) *s*. *s x* = *Some v*

definition *match-pres* **where**
match-pres pres s ≡ ∀ *pre*∈*set pres*. *match-pre pre s*

lemma *match-pres-distinct*:
distinct (*map fst pres*) ⟹ *match-pres pres s* ⟷ *Map.map-of pres* $\subseteq_m$ *s*
⟨*proof*⟩

fun *tree-map-of* **where**
tree-map-of updatea T [] = *T*
| *tree-map-of updatea T* ((*v*,*a*)#*m*) = *updatea v a* (*tree-map-of updatea T m*)

context *Map*
begin

abbreviation *tree-map-of′* ≡ *tree-map-of update*

lemma *tree-map-of-invar*: *invar T* ⟹ *invar* (*tree-map-of′ T pres*)
⟨*proof*⟩

lemma *tree-map-of-works*: *lookup* (*tree-map-of′ empty pres*) *x* = *map-of pres x*
⟨*proof*⟩

lemma *tree-map-of-dom*: *dom* (*lookup* (*tree-map-of′ empty pres*)) = *dom* (*map-of pres*)
⟨*proof*⟩
end

lemma *distinct-if-sorted*: *sorted xs* ⟹ *distinct xs*
⟨*proof*⟩

context *Map-by-Ordered*
begin

lemma *tree-map-of-distinct*: *distinct* (*map fst* (*inorder* (*tree-map-of′ empty pres*)))
⟨*proof*⟩

end

lemma *set-tree-intorder*: *set-tree t* = *set* (*inorder t*)
⟨*proof*⟩

lemma *map-of-eq*:
map-of xs = *Map.map-of xs*
⟨*proof*⟩

lemma *lookup-someD*: *lookup T x* = *Some y* ⟹ ∃ *p*. *p* ∈ *set* (*inorder T*) ∧ *p* = (*x*, *y*)
⟨*proof*⟩

lemma *map-of-lookup*: *sorted1* (*inorder T*) ⟹ *Map.map-of* (*inorder T*) = *lookup T*
⟨*proof*⟩

lemma *map-le-cong*: (⋀*x*. *m1 x* = *m2 x*) ⟹ *m1* $\subseteq_m$ *s* ⟷ *m2* $\subseteq_m$ *s*
⟨*proof*⟩

lemma *match-pres-submap*:
match-pres (*inorder* (*M.tree-map-of′ empty pres*)) *s* ⟷ *Map.map-of pres* $\subseteq_m$ *s*
⟨*proof*⟩

lemma [*code*]:
SAS-Plus-Representation.is-operator-applicable-in s op ⟷
match-pres (*inorder* (*M.tree-map-of′ empty* (*SAS-Plus-Representation.precondition-of op*))) *s*
⟨*proof*⟩

definition *decode-DIMACS-model dimacs-M h prob* ≡

(*ast-problem.decode-abs-plan prob*
  (*rem-noops*
    (*map* (λ*op*. $\varphi_O{}^{-1}$ (*prob-with-noop* (*ast-problem.abs-prob prob*)) *op*)
      (*concat*
        (Φ^{-1} (φ (*prob-with-noop* (*ast-problem.abs-prob prob*)))
          ((*dimacs-model-to-abs dimacs-M* (λ-. *False*)) *o*
            (*cnf-to-dimacs.var-to-dimacs* (*Suc h*)
              (*Suc* (*length* (*ast-problem.ast*δ *prob*)))))
          h)))))

lemma *planning-dimacs-sound-code*:
⟦*ast-problem.well-formed prob*;
  $\forall\,\pi\in$*set* (*ast-problem.ast*δ *prob*). *is-standard-operator*$'$ π⟧ $\Longrightarrow$
  let
    cnf-formula = (*SASP-to-DIMACS h prob*);
    decoded-plan = *decode-DIMACS-model dimacs-M h prob*
  in
    (*dimacs-model dimacs-M cnf-formula* $\longrightarrow$ *ast-problem.valid-plan prob decoded-plan*)
⟨*proof*⟩

definition
decode-DIMACS-model$'$ *dimacs-M h prob* $\equiv$
  decode-DIMACS-model dimacs-M h (*rem-implicit-pres-ops prob*)

lemma *planning-dimacs-sound-code*$'$:
⟦*ast-problem.well-formed prob*;
  ($\bigwedge$*op*. *op* $\in$ *set* (*ast-problem.ast*δ *prob*) $\Longrightarrow$ *consistent-pres-op op*);
  $\forall\,\pi\in$*set* (*ast-problem.ast*δ *prob*). *is-standard-operator* π⟧ $\Longrightarrow$
  let
    cnf-formula = (*SASP-to-DIMACS*$'$ *h prob*);
    decoded-plan = *decode-DIMACS-model*$'$ *dimacs-M h prob*
  in
    (*dimacs-model dimacs-M cnf-formula* $\longrightarrow$ *ast-problem.valid-plan prob decoded-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* $\Rightarrow$ *int rbt* **where**
  list-to-rbt [] = *Leaf*
| *list-to-rbt* (*x#xs*) = *insert-rbt x* (*list-to-rbt xs*)

lemma *inv-list-to-rbt*: *invc* (*list-to-rbt xs*) $\wedge$ *invh* (*list-to-rbt xs*)
⟨*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 ⟷
(*let tls* = *list-to-rbt ls in*
(∀ *c*∈*set cs*. *size* (*inter-rbt* (*tls*) (*list-to-rbt c*)) ≠ *0*) ∧
distinct (*map dimacs-lit-to-var ls*))
⟨*proof*⟩

definition
decode M h prob ≡
if ast-problem.well-formed prob then
if (∀ *op*∈*set* (*ast-problem.astδ prob*). *consistent-pres-op op*) *then*
if (∀ *op*∈*set* (*ast-problem.astδ prob*). *is-standard-operator op*) *then*
if (*dimacs-model M* (*SASP-to-DIMACS′ h prob*)) *then*
Inl (*decode-DIMACS-model′ M h prob*)
else Inr (*STR ′′Error*: *Model does not solve the problem*!′′)
else
Inr (*STR ′′Error*: *Conditional effects*!′′)
else
Inr (*STR ′′Error*: *Preconditions inconsistent′′*)
else
Inr (*STR ′′Error*: *Problem malformed*!′′)

lemma *decode-sound*:
decode M h prob = *Inl plan* ⟹
ast-problem.valid-plan prob plan
⟨*proof*⟩

lemma *decode-complete*:
decode M h prob = *Inr err* ⟹
¬ (*ast-problem.well-formed prob* ∧
(∀ *op* ∈ *set* (*ast-problem.astδ prob*). *consistent-pres-op op*) ∧
(∀ *π*∈*set* (*ast-problem.astδ prob*). *is-standard-operator π*) ∧
dimacs-model M (*SASP-to-DIMACS′ h prob*))
⟨*proof*⟩

lemma [*code*]:
ListMem x′ []= *False*
ListMem x′ (*x*#*xs*) = (*x′* = *x* ∨ *ListMem x′ xs*)
⟨*proof*⟩

lemmas [*code*] = *SASP-to-DIMACS-def ast-problem.abs-prob-def*
ast-problem.abs-ast-variable-section-def ast-problem.abs-ast-operator-section-def
ast-problem.abs-ast-initial-state-def ast-problem.abs-range-map-def
ast-problem.abs-ast-goal-def cnf-to-dimacs.var-to-dimacs.simps

ast-problem.astδ-def ast-problem.astDom-def ast-problem.abs-ast-operator-def ast-problem.astI-def ast-problem.astG-def ast-problem.lookup-action-def ast-problem.I-def execute-operator-sas-plus-def ast-problem.decode-abs-plan-def

definition *nat-opt-of-integer* :: *integer* ⇒ *nat option* **where**
nat-opt-of-integer i = (*if* (*i* ≥ *0*) *then Some* (*nat-of-integer i*) *else None*)

definition *max-var* :: *int list* ⇒ *int* **where**
max-var xs ≡ *fold* (λ(*x*::*int*) (*y*::*int*). *if abs x* ≥ *abs y then* (*abs x*) *else y*) *xs* (*0*::*int*)

export-code *encode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode String.implode max-var concat char-of-nat Int.nat integer-of-int length int-of-integer*
in *SML* **module-name** *exported* **file-prefix** *SASP-to-DIMACS*

export-code *decode nat-of-integer integer-of-nat nat-opt-of-integer Inl Inr String.explode String.implode max-var concat char-of-nat Int.nat integer-of-int length int-of-integer*
in *SML* **module-name** *exported* **file-prefix** *decode-DIMACS-model*

end

References

[1] M. Abdulaziz and F. Kurz. Formally verified sat-based ai planning, 2020.

[2] H. A. Kautz and B. Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

[3] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.