# 9

# Translation

The purpose of the translation component is to transform the input planning task, specified in the (first-order) PDDL formalism [38], into a fully instantiated multi-valued representation based on the SAS$^+$ formalism [10, 72].

PDDL and multi-valued planning tasks are introduced in Sect. 9.1, followed by an overview of the translation algorithm in Sect. 9.2. Translation is performed in four stages: *normalization* (Sect. 9.3), *invariant synthesis* (Sect. 9.4), *grounding* (Sect. 9.5), and *multi-valued planning task generation* (Sect. 9.6). The chapter ends with some notes on the performance of the translation component (Sect. 9.7).

## 9.1 PDDL and Multi-valued Planning Tasks

PDDL is the language in which the standard benchmarks discussed in Part I are usually expressed. In particular, the planning tasks of the international planning competitions are expressed in PDDL, so a planning system must be able to deal with this language in order to participate.

Like most current planning systems, Fast Downward is limited in scope to the *non-numerical* fragment of PDDL2.2, or what is called "level 1" of the PDDL language [42]. In other words, it does not accept PDDL tasks involving numerical state variables (introduced in PDDL level 2) or "durative actions", which allow specifying tasks that can only be solved by concurrent plans (introduced in PDDL level 3 and refined in PDDL level 4).

On the other hand, Fast Downward *can* deal with all "purely logical" aspects of the language, including arbitrary first-order formulae in action conditions and goals, universal and conditional effects, and derived predicates (axioms) introduced in PDDL2.2 [38]. To make these notions somewhat more precise, we now formally introduce the class of PDDL tasks which the planner can handle.

M. Helmert: Understanding Planning Tasks, LNAI 4929, pp. 171–206, 2008.

**Definition 9.1.1.** ***PDDL Tasks***
*A (non-numeric, non-temporal)* ***PDDL task*** *is given by a 5-tuple* $\Pi = \langle \mathcal{L}, \chi_0, \chi_\star, \mathcal{A}, \mathcal{O} \rangle$ *with the following components:*

- $\mathcal{L}$ *is a finite first-order language, consisting of constant symbols (****objects****), relation symbols (****predicates****) and variable symbols. Predicates are partitioned into* ***fluent predicates*** *(affected by operators) and* ***derived predicates*** *(computed by evaluating axioms).*
- $\chi_0$ *is a conjunction of ground atoms over objects and fluent predicates called the* ***initial state****.*
- $\chi_\star$ *is a closed first-order formula over* $\mathcal{L}$ *called the* ***goal formula****.*
- $\mathcal{A}$ *is a set of* ***schematic axioms****, which are pairs* $\langle \varphi, \psi \rangle$ *such that* $\varphi$ *is an atom over* $\mathcal{L}$ *whose predicate symbol is a derived predicate and* $\psi$ *is a formula over* $\mathcal{L}$ *with* $\mathit{free}(\psi) \subseteq \mathit{free}(\varphi)$*. We write the axiom* $\langle \varphi, \psi \rangle$ *as* $\varphi \leftarrow \psi$ *and call* $\varphi$ *the* ***head*** *and* $\psi$ *the* ***body*** *of the axiom.*
  *We require that* $\mathcal{A}$ *is stratifiable, i. e., there exists a total preorder* $\preceq$ *on the set of derived predicates such that for each axiom where* $Q$ *occurs in the head, we must have* $P \preceq Q$ *for all derived predicates* $P$ *occurring in the body, and* $P \prec Q$ *for all derived predicates* $P$ *occurring in a negative literal in the translation of the body to negation normal form. Intuitively,* $P \prec Q$ *means that the truth value of atoms over* $P$ *must be computed before the truth value of atoms over* $Q$*.*
- $\mathcal{O}$ *is a finite set of* ***schematic operators*** *over* $\mathcal{L}$*. A schematic operator* $\langle \chi, e \rangle$ *consists of a first-order formula* $\chi$ *over* $\mathcal{L}$ *called its* ***precondition*** *and its* ***effect*** $e$*. Effects are recursively defined by finite application of the following rules:*
  - *A literal* $l$ *over* $\mathcal{L}$*, excluding derived predicates, is an effect called a* ***simple effect****.*
  - *If* $e_1, \ldots, e_n$ *are effects, then* $e_1 \wedge \cdots \wedge e_n$ *is an effect called a* ***conjunctive effect****.*
  - *If* $\chi$ *is a first-order formula over* $\mathcal{L}$ *and* $e$ *is an effect, then* $\chi \rhd e$ *is an effect called a* ***conditional effect****.*
  - *If* $v_1, \ldots, v_k$ *are variable symbols in* $\mathcal{L}$ *and* $e$ *is an effect, then* $\forall v_1 \ldots v_k : e$ *is an effect called a* ***universally quantified effect*** *or* ***universal effect****.*

  *Free variables of an effect are defined recursively as in first-order logic, where the set of free variables of a conditional effect is defined as* $\mathit{free}(\chi \rhd e) = \mathit{free}(\chi) \cup \mathit{free}(e)$*.*
  *The set of free variables of a schematic operator is defined as* $\mathit{free}(\langle \chi, e \rangle) = \mathit{free}(\chi) \cup \mathit{free}(e)$*. Free variables are also referred to as the* ***parameters*** *of the schematic operator.*

We assume that the reader is already familiar with PDDL semantics and point to the language definition [38, 42] for more information. Our definition allows general first-order conditions as well as (possibly nested) conditional and quantified effects and axioms. The stratifiability condition for axioms

ensures that the interpretation of derived predicates is well-defined. Without this condition, there could be rules of the form "$P(x)$ is true whenever $P(x)$ is false."

Apart from syntactic differences, there are three aspects of non-numerical, non-temporal PDDL2.2 not captured by our definition:

- There are no operator names. The translator deals with operator names in such a way that the translated operators are referred to by the same name as its PDDL2.2 counterpart, so that the plans generated by the planner need not undergo any form of post-processing. This is all fairly simple, and we will not discuss this matter further.
- There is no distinction between domain constants and objects of the problem instance, or indeed between the domain and problem instance specification in general. At the level of individual problem instances at which the translator works, there is no need for such a distinction.
- There are no types. The translator compiles away types into unary predicates straight away, so a PDDL specification stating that `a` is an object of type `vehicle` is treated equivalently to an untyped specification stating that `(vehicle a)` is true in the initial state. Types occurring in quantified conditions or effects are translated accordingly; e. g. a precondition `(exists (?v - vehicle) (empty ?v))` is translated to $\exists v$ : $\texttt{vehicle}(v) \wedge \texttt{empty}(v)$, and an effect `(forall (?v - vehicle) (empty ?v))` is translated to $\forall v : (\texttt{vehicle}(v) \rhd \texttt{empty}(v))$.

With PDDL as a starting point, let us now introduce the kinds of planning tasks we want the translator to generate. These are based on the SAS$^+$ planning model [10, 72], extended to allow for derived predicates and conditional effects.

The definition will exhibit a number of similarities, but also a few differences between PDDL tasks and our planning model. Most notably, PDDL tasks use first-order concepts such as schematic operators whose variables can be instantiated in many different ways, while our formalism is grounded. Moreover, our formalism only allows simple conjunctions in goals, axioms and operators, and conditional effects cannot be nested. The former difference necessitates operator instantiation as part of the translation process, while the others require normalization of conditions.

**Definition 9.1.2.** ***Multi-valued Planning Tasks (MPTs)***
*A* ***multi-valued planning task (MPT)*** *is given by a 5-tuple* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ *with the following components:*

- $\mathcal{V}$ *is a finite set of* ***state variables****, each with an associated finite domain* $\mathcal{D}_v$*. State variables are partitioned into* ***fluents*** *(affected by operators) and* ***derived variables*** *(computed by evaluating axioms). The domains of derived variables must contain the* ***default value*** $\bot$*.*
  *A* ***partial variable assignment*** *or* ***partial state*** *over* $\mathcal{V}$ *is a function* $s$ *on some subset of* $\mathcal{V}$ *such that* $s(v) \in \mathcal{D}_v$ *wherever* $s(v)$ *is defined. A*

*partial state is called an* ***extended state*** *if it is defined for all variables in* $\mathcal{V}$ *and a* ***reduced state*** *or* ***state*** *if it is defined for all fluents in* $\mathcal{V}$*. In the context of partial variable assignments, we write* $v = d$ *for the variable-value pairing* $\langle v, d\rangle$ *or* $v \mapsto d$*.*

- $s_0$ *is a state over* $\mathcal{V}$ *called the* ***initial state****.*
- $s_\star$ *is a partial variable assignment over* $\mathcal{V}$ *called the* ***goal****.*
- $\mathcal{A}$ *is a finite set of (MPT)* ***axioms*** *over* $\mathcal{V}$*. Axioms are triples of the form* $\langle cond, v, d\rangle$*, where cond is a partial variable assignment called the* ***condition*** *or* ***body*** *of the axiom,* $v$ *is a derived variable called the* ***affected variable****, and* $d \in \mathcal{D}_v$ *is called the* ***derived value*** *for* $v$*. The pair* $\langle v, d\rangle$ *is called the* ***head*** *of the axiom and can be written as* $v := d$*.*
  *The axiom set* $\mathcal{A}$ *is partitioned into a totally ordered set of* ***axiom layers*** $\mathcal{A}_1 \prec \cdots \prec \mathcal{A}_k$ *such that within the same layer, each affected variable may only be associated with a single value in axiom heads and bodies. In other words, within the same layer, axioms with the same affected variable but different derived values are forbidden, and if a variable appears in an axiom head, then it may not appear with a different value in a body. This is called the* ***layering property****.*
- $\mathcal{O}$ *is a finite set of (MPT)* ***operators*** *over* $\mathcal{V}$*. An operator* $\langle pre, eff\rangle$ *consists of a partial variable assignment pre over* $\mathcal{V}$ *called its* ***precondition****, and a finite set of* ***effects*** *eff. Effects are triples* $\langle cond, v, d\rangle$*, where cond is a (possibly empty) partial variable assignment called the* ***effect condition****,* $v$ *is a fluent called the* ***affected variable****, and* $d \in \mathcal{D}_v$ *is called the* ***new value*** *for* $v$*.*

*For axioms and effects, we also use the notation* $cond \to v := d$ *in place of* $\langle cond, v, d\rangle$*.*

To provide a formal semantics for MPT planning, we first need to formalize the semantics of axioms.

**Definition 9.1.3.** ***Extended States Defined by a State***
*Let* $s$ *be a state of an MPT* $\Pi$ *with axioms* $\mathcal{A}$*, layered as* $\mathcal{A}_1 \prec \cdots \prec \mathcal{A}_k$*. The* ***extended state defined by*** $s$*, written as* $\mathcal{A}(s)$*, is the result* $s'$ *of the following algorithm:*

**algorithm** evaluate-axioms($\mathcal{A}_1$, …, $\mathcal{A}_k$, $s$):
  **for each** variable $v$:
    $$s'(v) := \begin{cases} s(v) & \text{if } v \text{ is a fluent variable} \\ \bot & \text{if } v \text{ is a derived variable} \end{cases}$$
  **for** $i \in \{1, \ldots, k\}$:
    **while** there exists an axiom $(cond \to v := d) \in \mathcal{A}_i$
      with $cond \subseteq s'$ **and** $s'(v) \neq d$:
      Choose such an axiom $cond \to v := d$.
      $s'(v) := d$

In other words, axioms are evaluated in a layer-by-layer fashion using fixed point computations, which is very similar to the semantics of stratified logic

programs. It is easy to see that the layering property from Definition 9.1.2 guarantees that the algorithm terminates and produces a deterministic result. Having defined the semantics of axioms, we can now define the state space of an MPT.

**Definition 9.1.4.** ***MPT State Transition Graph***
*The* ***state transition graph*** *of an MPT* $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$*, denoted as* $\mathcal{S}(\Pi)$*, is a directed graph. Its vertex set is the set of states of* $\mathcal{V}$*, and it contains an arc* $\langle s, s' \rangle$ *iff there exists some operator* $\langle pre, eff \rangle \in \mathcal{O}$ *such that:*

- $pre \subseteq \mathcal{A}(s)$*,*
- $s'(v) = d$ *for all effects* $cond \rightarrow v := d \in eff$ *such that* $cond \subseteq \mathcal{A}(s)$*, and*
- $s'(v) = s(v)$ *for all other fluents.*

Having defined state transition graphs for MPT tasks, we can introduce MPT plans, the MPT planning problem Plan-MPT, the MPT plan existence PlanEx-MPT and the MPT bounded plan existence problem PlanLen-MPT in a similar way to the definitions in Chap. 2. Indeed, we can consider the set of all MPTs a (very general) planning domain in the sense of Definition 2.2.2, and thus no further definitions of these minimization and decision problems are required.

The plan existence and bounded plan existence problems for MPTs are easily shown to be PSPACE-hard because they generalize the corresponding problems for propositional STRIPS, which are known to be PSPACE-complete [19]. Moreover, from Theorems 2.3.2 and 2.3.4, we know that they belong to PSPACE, so they are PSPACE-complete. Similarly, it is easy to see from Theorem 2.3.1 that Plan-MPT belongs to EXPO $\setminus$ NPS because plans may be exponentially long (but not longer), and explicit search in the state transition graph can be implemented to run in exponential time.

This concludes our formal introduction of MPT planning. In the following section, we turn to the issue of generating multi-valued planning tasks from PDDL planning tasks.

## 9.2 Translation Overview

Translation is performed in a sequence of transformation steps. Starting from a PDDL specification, we apply some well-known logical equivalences to compile away types and simplify conditions and effects in the *normalization* step. Next, the *invariant synthesis* step computes mutual exclusion relations between atoms, which are later used for synthesizing the MPT variables. The *grounding* step performs a relaxed reachability analysis to compute the set of ground atoms, axioms and operators that are considered relevant for the planning task and computes a grounded PDDL representation. Invariant synthesis and grounding are not related to one another and could just as well be performed in the opposite order. Finally, the *MPT generation* step chooses

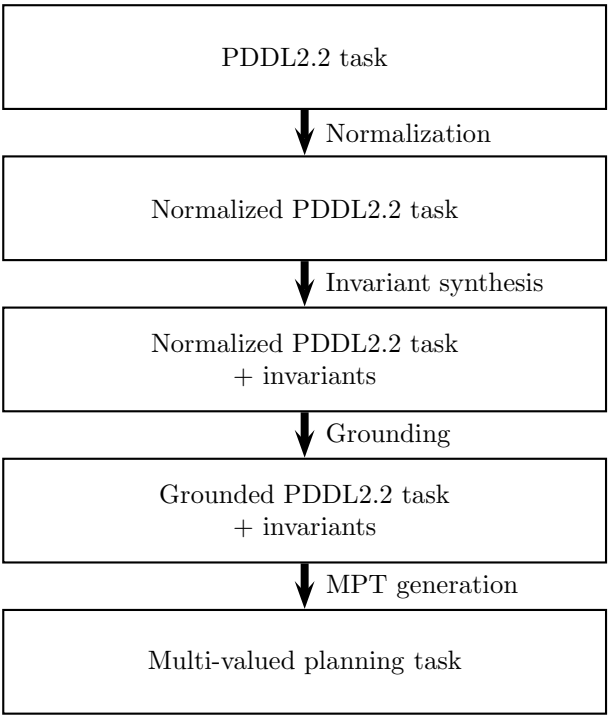


**Fig. 9.1.** Overview of the translation algorithm

the final set of state variables by using the information from invariants and grounding and produces the MPT output.

The translation process in outlined in Fig. 9.1. In the following sections, we will discuss the various transformation steps in sequence.

However, before we do so, we should point out that of these four steps, only three are necessary to convert a PDDL task to an MPT: the invariant synthesis step can be omitted. However, without the use of invariants, there would be a 1:1 correspondence between (relevant) ground atoms of the PDDL task and state variables of the MPT; in particular, all state variables in the generated MPT would be binary. Recalling the motivating example from the previous chapter, this would imply that the causal graph of the resulting MPT would have the undesirable form shown in Fig. 8.7, rather than the much more structured form shown in Fig. 8.6.

## 9.3 Normalization

The normalization step has three responsibilities: Compiling away types, simplifying conditions, and simplifying effects. The current implementation of

the translator cannot handle PDDL types in their full generality: Type inheritance and the `either` construct are not supported. It would not be difficult to add these to the mix, but these seem to be unused language features, and we did not want to waste implementation effort on them. So we only deal with primitive types and the built-in standard type `object`, to which all objects belong.

### 9.3.1 Compiling Away Types

As indicated earlier, types are compiled away as soon as the planning task is read in. For each type occurring in the input, and for the type `object`, we introduce a new unary predicate with the same name. Typed constructs occur in PDDL2.2 specifications in a semantically meaningful way in three places:

1. Definition of domain constants and objects of the task (*typed objects*).
2. Definition of formal parameters of schematic operators (*typed operators*).
3. Definition of quantified variables in existential and universal conditions and universal effects (*typed quantifiers*).

Typed objects are translated into new atoms for the initial state. For example, the specification `someobj - sometype` leads to a new initial atom `(sometype someobj)`. Moreover, for each object `someobj`, we introduce an initial atom `(object someobj)`.

Typed operators are transformed by introducing new preconditions. For example, for an operator with parameter specification `:parameters (?par1 - type1 ?par2 - type2)` and precondition $\varphi$, the parameter specification is replaced by `:parameters (?par1 ?par2)` and the precondition is replaced by `(and (type1 ?par1) (type2 ?par2)` $\varphi$`)`.

Typed quantifiers in conditions are compiled away with the usual first-order logic idioms, so that condition `(exists (?v - type)` $\varphi$`)` translates to `(exists (?v) (and (type ?v)` $\varphi$`))` and condition `(forall (?v - type)` $\varphi$`)` translates to `(forall (?v) (imply (type ?v)` $\varphi$`))`.

Similarly, typed quantifiers in effects are compiled into conditional effects, so that the effect `(forall (?v - type)` $e$`)` becomes `(forall (?v) (when (type ?v)` $e$`))`.

After types have been eliminated, we are left with a PDDL task in the sense of Definition 9.1.1. We will thus use the more concise logical notation from that definition in the following, rather than the more lengthy PDDL syntax. For example, we will write $\varphi \vee \psi$ instead of `(or` $\varphi$ $\psi$`)` and $\varphi \rhd e$ instead of `(when` $\varphi$ $e$`)`.

### 9.3.2 Simplifying Conditions

In PDDL tasks, general first-order formulae may occur in many places: goal formula, axiom bodies, operator preconditions and conditions of conditional effects. Our aim is to replace all these with simple conjunctions of literals.

Towards this goal, we first eliminate implications with the equivalence $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$ and translate the resulting conditions into first-order negation normal form using de Morgan's laws for first-order logic.

The next step is slightly tricky. If there are any universally quantified conditions, we rewrite the outermost universal quantification in all conditions with the equivalence $\forall x\varphi \equiv \neg\exists x\neg\varphi$. This might seem somewhat silly because this transformation destroys negation normal form, so after the rewrite, we introduce a new axiom for the subformula that violates the normal form property, $\exists x\neg\varphi$. Formally, if $\text{free}(\exists x\neg\varphi) = \{v_1, \ldots, v_k\}$, we introduce a new derived predicate `new-pred` of arity $k$, defined by the axiom $\texttt{new-pred}(v_1, \ldots, v_k) \leftarrow \psi$, where $\psi$ is the translation of $\exists x\neg\varphi$ to negation normal form. We can then replace the original condition $\forall x\varphi$ by $\neg\texttt{new-pred}(v_1, \ldots, v_k)$. If several variables are universally quantified together within the same expression, we transform them together, introducing only one new derived predicate for the quantifier group. We repeat this step until there are no more universally quantified conditions. Note that only universally quantified *conditions* are translated, not universal *effects*, which also use the $\forall$ notation. Universal effects cannot simply be compiled away, so we deal with them separately in a later stage.

If after elimination of universal quantifiers the goal condition is *not* a simple conjunction, we replace it by a new axiom, since the following transformations sometimes require splitting several conditions into two, which is easy to do for axiom bodies, operator preconditions and effect conditions, but not possible in our formalism for goal conditions, of which there can be only one. So for example, if the goal is $\varphi \vee \psi$, we introduce a new parameter-less derived predicate `goal-pred` and a new axiom $\texttt{goal-pred} \leftarrow \varphi \vee \psi$, replacing the original goal with the atom `goal-pred`.

The next step is the elimination of disjunctions. We move disjunctions to the roots of conditions by applying the equivalences $\exists x(\varphi \vee \psi) \equiv \exists x\varphi \vee \exists x\psi$ and $\varphi \wedge (\psi \vee \psi') \equiv (\varphi \wedge \psi) \vee (\varphi \wedge \psi')$ and the laws of associativity and commutativity. In theory, moving disjunctions over conjunctions can lead to an exponential increase in formula size, which we could avoid by introducing new axioms for component formulae. In practice, the conditions encountered in actual planning domains are not problematic in this regard, and we decided that the potential savings in the size of the representation were not worth the overhead of maintaining the state of another derived variable during search.

After disjunctions have been moved to the root of all formulae, we can eliminate them by splitting the surrounding structures: If the disjunction $\varphi \vee \psi$ is part of an axiom body, we generate two axioms with identical head, one with body $\varphi$ and one with body $\psi$. If the disjunction is part of an operator precondition, we replace the operator by two copies of the original, one with precondition $\varphi$ and one with precondition $\psi$. Finally, if the disjunction is part of an effect condition, we replace the conditional effect $(\varphi \vee \psi) \rhd e$ by $(\varphi \rhd e) \wedge (\psi \rhd e)$.

Next, we move existential quantifiers out of conjunctions by applying the equivalence $(\exists x\varphi) \wedge \psi \equiv \exists x(\varphi \wedge \psi)$. The equivalence only holds when $x \notin$

free($\psi$), so to avoid trouble here and later, we first rename all variables bound by quantifiers to some unique name.

Having moved existential quantifiers to the root of conditions, we eliminate them as follows: For axioms, we simply drop them, following the PROLOG convention that all free variables in the body that are not part of the head are implicitly existentially quantified. For operator preconditions, we also drop them, adding the existentially quantified variables to the parameter list of the schematic operator. For effect conditions, we replace $(\exists x \varphi) \rhd e$ by $\forall x : (\varphi \rhd e)$.

### 9.3.3 Simplifying Effects

After the somewhat laborious simplification of conditions, effect simplification is conceptually very simple. First, universal and conditional effects are moved into conjunctive effects by the equivalences $\forall x : (e \wedge e') \equiv (\forall x : e) \wedge (\forall x : e')$ and $\varphi \rhd (e \wedge e') \equiv (\varphi \rhd e) \wedge (\varphi \rhd e')$. Second, conditional effects are moved into universal effects by the equivalence $\varphi \rhd (\forall x : e) \equiv \forall x : (\varphi \rhd e)$. Finally, nested effects of the same type are flattened, i. e., conjunctive effects containing conjunctive effects are collapsed into a single conjunctive effects with more conjuncts, universal effects containing universal effects are collapsed into a single universal effect quantifying over more variables, and nested conditional effects of the type $\varphi \rhd (\psi \rhd e)$ are transformed to $(\varphi \wedge \psi) \rhd e$. Note that this latter modification preserves the previously generated normal form for effect conditions.

After these transformations, the possible nesting of effects is thus restricted to the simple chain *conjunctive effect* $\succ$ *universal effect* $\succ$ *conditional effect* $\succ$ *simple effect*. However, not all effect types must necessarily be present, e. g. conditional effects *need not* occur within universal effects, etc. To enforce a regular effect structure, we replace simple effects $e$ not surrounded by conditional effects by $\top \rhd e$ ($\top$ is seen as the empty conjunction, so this condition is in normal form), conditional effects $e$ not surrounded by universal effects by $\forall : e$ (quantifying over zero variables), and universal effects $e$ not surrounded by conjunctive effects by a conjunctive effect containing the singleton $e$.

As a result, after normalization each operator has a list (conjunction) of effects, each a simple effect with an associated set of universal quantifiers and an associated condition, both of which can be trivial. Thus it is not necessary to store normalized operator effects in a tree structure; a flat vector is sufficient.

### 9.3.4 Normalization Result

This concludes the normalization step. In Fig. 9.1, we referred to the output of the normalization phase as a *normalized PDDL2.2 task*. Let us formalize this notion here for the benefit of further discussion:

**Definition 9.3.1.** ***Normalized PDDL Tasks***
*A* ***normalized PDDL task*** *is a PDDL task that satisfies the following structural restrictions:*

- *The goal formula is a conjunction of literals.*
- *All axiom bodies are conjunctions of literals (except for the possible implicit existential quantification of free variables not occurring in the axiom head).*
- *All operator preconditions are conjunctions of literals.*
- *All effect conditions are conjunctions of literals.*
- *All operator effects are conjunctions of universally quantified conditional simple effects.*

In the following, we will refer to the individual simple effects of an operator in a normalized PDDL task as being arranged in an *effect list*. For the simple effect $e$ occurring within the universal conditional effect $\forall vars : \varphi \rhd e$, we will refer to *vars* as the set of *bound variables* of $e$ and to $\varphi$ as the *condition* of $e$. If $e$ is a positive literal, we will call it an *add effect*, otherwise a *delete effect*.

## 9.4 Invariant Synthesis

An invariant is a property of a world state in a planning task which is satisfied by all world states that are reachable from the initial state. Many invariants are uninteresting; for example, the property "At least five state variables are true" is an invariant in most propositional STRIPS planning tasks, but does not seem to entail a useful piece of information for a planner. Other invariants would be useful to know but are too difficult to verify. For example, "This state is not a goal state" is an invariant iff the planning task is not solvable, so confirming the invariance of that state property is PSPACE-hard.

Nevertheless, invariants are a useful tool for many planning systems, which is why they have been studied by many researchers in a variety of contexts [41, 45, 100, 101], often involving SAT-based planning. For the purposes of translating propositional planning tasks to a multi-valued formalism, *mutual exclusion* (*mutex*) invariants are especially interesting. A mutex invariant states that certain propositions can never be true at the same time. This affects translation because a set of propositions which are pairwise mutually exclusive can be easily encoded as a *single* state variable whose value specifies *which* of the propositions is true (if any is true at all), rather than as a number of state variables encoding the truth value for each proposition individually.

Invariance is usually proven inductively: First, one shows that a hypothesized property is true in the initial state. Then, one shows that if the property is true in some state, it must also be true in all successor states. Together, this implies that the property is true in all reachable states, and thus an invariant.

As mentioned before, the automatic discovery of invariants is a hard problem in general, but for many relevant types of state properties, sufficient conditions exist that can be checked quickly. Still, synthesizing invariants is costly,

and for this reason, we are interested in algorithms working directly with the first-order PDDL description of a planning task, not on a grounded representation. Indeed, our algorithm goes beyond this requirement by not relying on the information in the *task* file of the PDDL input at all, solely exploiting information present in the *domain* file. This is a valuable feature, but it rules out the possibility of proving mutex conditions, because a mutex cannot be established without checking the initial state. Instead, we use a slight generalization of mutexes.

**Definition 9.4.1.** ***Monotonicity Invariant Candidates***
*A* ***monotonicity invariant candidate*** *for a PDDL task $\Pi$ is given by a pair $\mathcal{P} = \langle V, \Phi \rangle$, where $V$ is a set of first-order variables called the* ***parameters*** *of the candidate, and $\Phi$ is a set of atoms. Variables occurring freely in $\Phi$ which are not parameters are called* ***counted variables*** *of the candidate.*

*For $V = \{v_1, \ldots, v_m\}$ and $\Phi = \{\varphi_1, \ldots, \varphi_k\}$, we write $\mathcal{P}$ symbolically as $\forall v_1 \ldots v_m\ \varphi_1 + \cdots + \varphi_k \downarrow$. In the special case $V = \emptyset$, we write $\forall \cdot\ \varphi_1 + \cdots + \varphi_k \downarrow$.*

In the following, we will mostly refer to monotonicity invariant candidates as *invariant candidates* or simply *candidates*; we do not consider other kinds of invariant candidates.

The preceding definition defines the syntax for invariant candidates; we now have to provide the semantics. Since this is somewhat involved, we provide an example from a transportation domain first. Consider the invariant candidate $\langle \{p\}, \{\mathtt{at}(p, l), \mathtt{in}(p, v)\} \rangle$, where $p$, $l$ and $v$ are variable symbols. We write this as $\forall p\, \mathtt{at}(p, l) + \mathtt{in}(p, v) \downarrow$ and read it as "For all packages $p$, the number of locations $l$ such that $\mathtt{at}(p, l)$ is true plus the number of vehicles $v$ such that $\mathtt{in}(p, v)$ is true, is non-increasing." In our terminology, $p$ is the parameter of the candidate, while $l$ and $v$ are the counted variables. This invariant candidate is an actual invariant – it *does* hold in all reachable states – and it is one of the invariants found by our algorithm in a LOGISTICS-like domain. Let us now formalize what it means for a candidate to be an invariant.

**Definition 9.4.2.** ***Monotonicity Invariants***
*Let $\Pi$ be a PDDL task and let $\mathcal{P} = \langle V, \Phi \rangle$ be a monotonicity invariant candidate for $\Pi$.*

*An* ***instance*** *of $\mathcal{P}$ is a function $\alpha$ mapping the variables in $V$ to objects of the planning task $\Pi$.*

*The set of* ***covered facts*** *of an instance $\alpha$ is the set of all ground atoms of the planning task $\Pi$ which unify with some $\varphi \in \Phi$ under $\alpha$, i. e., the set of all ground atoms $\varphi_0$ of $\Pi$ for which there exists a variable map $\beta \supseteq \alpha$ such that $\beta(\varphi) = \varphi_0$ for some $\varphi \in \Phi$.*

*The* ***weight*** *of an instance $\alpha$ in a state $s$ is the number of covered facts of $\alpha$ which are true in $s$.*

*The monotonicity invariant candidate $\mathcal{P}$ is called a* ***monotonicity invariant*** *iff for all instances $\alpha$ of $\mathcal{P}$, all states $s$ reachable from the initial state of $\Pi$ and all successor states $s'$ of $s$, the weight of $\alpha$ in $s'$ is no greater than the weight of $\alpha$ in $s$.*

The definition is probably best understood by considering the previously discussed example invariant. Similar to our convention for invariant candidates, we usually refer to monotonicity invariants simply as *invariants*.

As hinted before, monotonicity invariants are useful for grouping a number of related propositions into a single multi-valued variable: If we have found an invariant for a planning task *and* a given instance of that invariant has weight 1 in the initial state, then the facts covered by that instance are pairwise mutually exclusive. This is how the synthesized invariants are utilized during the later stages of translation.

So how do we generate invariants? Since the number of feasible candidates is too high for a guess-and-check algorithm, we follow a *guess, check and repair* approach: Starting from a set of a few simple initial elements, we try to prove that the candidates are indeed invariants. Whenever this is the case, we keep the invariant and do not consider it further. However, when the proof fails, we try to detect *why* this is the case and refine the candidate to generate more candidates that do not fail for the same reason (although they might fail for other reasons). From a high-level perspective, this is basically a search problem, and indeed we solve it using standard breadth-first search with a closed list. What remains to be said is how the search space of the algorithm is defined:

- *Initial states:* What are the initial candidates?
- *Termination test:* How do we prove that a candidate is an invariant?
- *Successor set:* How do we refine a candidate for which this proof fails?

We will deal with these questions in the following.

### 9.4.1 Initial Candidates

Before starting the actual invariant synthesis algorithm, we check which predicates are affected by operators at all: Some predicates, including but not limited to those representing types, are *constant* in the sense that atoms over these predicates have the same truth values in all states. Such predicates are no longer needed after grounding, so we need not consider them for invariant candidates. Of course, a constant predicate trivially satisfies a monotonicity invariant, but these are not very useful.

Therefore, we limit the set of interesting predicates to all *modifiable fluent predicates*, i. e., predicates which occur within operator effects (as part of a simple effect, not merely as part of an effect condition). Note that this also excludes derived predicates. In theory, there is no reason why there should be no monotonicity invariants involving derived predicates, but in practice we have not seen examples of this, and detecting them would require a more global view of the task definition and hence more effort than we would like to spend. We will come back to the issue of derived predicates when discussing our method for proving invariance.

The set of initial invariant candidates consists of all those candidates (up to isomorphism, i. e., renaming of variables) which contain at most one counted variable and exactly one atom, over a modifiable fluent predicate, whose parameters are distinct variables. In our experience, invariants with several counted variables per atom are exceedingly rare; in fact, we have not seen an example in practice.

To illustrate the initialization of invariant candidates, we show the three candidates generated for the binary `at` predicate in the LOGISTICS domain:

$$\forall x\, \mathtt{at}(x, l) \downarrow \tag{9.1}$$

$$\forall l\, \mathtt{at}(x, l) \downarrow \tag{9.2}$$

$$\forall x, l\, \mathtt{at}(x, l) \downarrow \tag{9.3}$$

Similar candidates are introduced for the `in` predicate. Intuitively, the first candidate states that no object can be at more locations in the successor state than in the current state, the second candidate states that no location can be occupied by more objects in the successor state than in the current state, and the third candidate states that a given object cannot occupy a given location in the successor state if this is not the case in the current state.

Candidates (9.2) and (9.3) are obviously not invariants. Candidate (9.1) is not an invariant either because an object which is currently inside a vehicle can be at some location in the successor state while being at no location in the current state. However, we will see that we can refine (9.1) into an invariant.

### 9.4.2 Proving Invariance

When is an invariant candidate an invariant? We stated that invariants are usually proved by establishing their truth in the initial state and using inductive arguments for the effects of operator application. For monotonicity invariants, only the inductive step is necessary; there is nothing special to prove about the initial state. So in order to prove that a given invariant candidate is an invariant, we must show that no operator can increase the weight of any of its instances. An operator increases the weight of some instance of an invariant candidate iff the number of covered facts that it makes true is greater than the number of covered facts that it makes false. If an operator does not increase the weight of any instance, then we say that it is *balanced* with regard to the invariant.

Ultimately, we are interested in instances of monotonicity invariants that give rise to mutexes, so that only instances of weight 1 are relevant for us. For this reason, we use the following condition which is slightly stronger than balance.

**Definition 9.4.3.** ***Threatened Invariant Candidates***
*An invariant candidate $\mathcal{P}$ is* ***threatened*** *by a schematic operator iff one of the following two conditions holds:*

- *The operator has an add effect that can increase the weight of an instance of $\mathcal{P}$ in some state, but no delete effect that is guaranteed to decrease the weight of the same instance in the same state. In this case, we say that the operator is* ***unbalanced*** *with regard to $\mathcal{P}$.*
- *When ignoring delete effects, the operator can increase the weight of some instance of $\mathcal{P}$ in some state by at least 2. In this case, we say that the operator is* ***too heavy*** *for $\mathcal{P}$.*

Clearly, not being threatened by any schematic operator is a sufficient condition for being a monotonicity invariant. The definition gives rise to the algorithm shown in Fig. 9.2. Most of the actual work is in unifying operator parameters and quantified variables of universal conditions; the algorithm simplifies significantly in STRIPS domains. We do not want to discuss the algorithm in all detail, instead focusing on two points that require some explanation, namely the satisfiability and entailment tests that occur towards the end of algorithms *check-operator-too-heavy* and *check-operator-unbalanced*.

For the heaviness test, two add effects can only lead to an operator being too heavy if the operator is actually applicable ($o'$.precondition is true), both add effects apply ($e$.condition and $e'$.condition are true) and the add effects actually add propositions that were not true previously ($e$.atom and $e'$.atom are false). For the imbalance test, an add effect is unbalanced by default. However, it becomes balanced if whenever the operator is actually applicable ($o'$.precondition is true), the add effect triggers ($e$.condition is true) and actually adds something ($e$.atom is false), then something is deleted at the same time, which means that the delete effect triggers ($e'$.condition is true) and deletes something that was previously true ($e'$.atom is true).

Coming back to the earlier LOGISTICS example, all three initial candidates are threatened by the same operator `unload-truck`, whose add effect $\texttt{at}(x, l)$ is not balanced. Thus, as indicated before, none of (9.1)–(9.3) is an invariant.

There are a few subtleties about the algorithm which we want to point out briefly:

- We duplicate universal effects at the beginning of *check-operator-too-heavy* so that we can detect if two different instantiations of the same universal effect can simultaneously increase the weight of some instance of the invariant candidate.
- Where Fig. 9.2 contains statements like "Let $o'$ be a copy of $o$ where variables are renamed so that…", the question arises whether such a renaming is uniquely determined, and what to do if it is not. Indeed, renamings are unique (and easy to compute) as long as all atoms of the candidate refer to *different* predicates, which is usually the case. However, the algorithm generalizes to invariant candidates with several occurrences of the same predicate, like $\forall x\, \texttt{at}(x, y) + \texttt{at}(y, x) \downarrow$. This requires that all possible (non-isomorphic) renamings must be considered for $o'$ in algorithm *check-operator-unbalanced*. In our experience, invariants of this type are not very useful, although Fast Downward implements them correctly.

**algorithm** prove-invariant($V$, $\Phi$):
    **for each** schematic operator $o$ with add effect for some predicate in $\Phi$:
        **call** check-operator-too-heavy($o$, $V$, $\Phi$).
        **call** check-operator-unbalanced($o$, $V$, $\Phi$).
    **accept** candidate as an invariant.

{ In the following, the *variables* of an operator include both its operator parameters and quantified variables of its effects. We assume that all variable names are unique, and that whenever a variable is renamed, the change is immediately reflected in program variables referring to effects of the operator. For example, if $e = \mathtt{at}(p, l)$ is an effect of operator $o$ and $p$ is renamed to $v$ within $o$, then $e$ becomes $\mathtt{at}(v, l)$. }

**algorithm** check-operator-too-heavy($o$, $V$, $\Phi$):
    Let $o'$ be a copy of $o$.
    Duplicate all (non-trivially) quantified effects of $o'$.
    Assign unique names to all quantified variables in effects of $o$ and $o'$.
    **for each** pair $(e, e')$ of add effects of $o'$ that affect a predicate in $\Phi$:
        **if** the variables of operator $o'$ can be renamed so that
          ($e$.atom $\neq$ $e'$.atom **and**
           covers($V$, $\Phi$, $e$.atom) **and** covers($V$, $\Phi$, $e'$.atom) **and**
           $o'$.precondition$\wedge e$.condition$\wedge e'$.condition$\wedge\neg e$.atom$\wedge\neg e'$.atom
           is satisfiable):
            **reject** candidate. { The operator is too heavy. }

**algorithm** check-operator-unbalanced($o$, $V$, $\Phi$):
    **for each** add effect $e$ of $o$ that affects a predicate in $\Phi$:
        Let $o'$ be a copy of $o$ where the variables are renamed so that
          covers($V$, $\Phi$, $e$.atom) is true. Do not rename two variables
          to the same variable except when forced.
        **for each** delete effect $e'$ of $o'$ that affects a predicate in $\Phi$:
            **if** the quantified variables of $e'$ can be renamed in $o'$
              so that (covers($V$, $\Phi$, $e'$.atom) **and**
                $o'$.precondition $\wedge$ $e$.condition $\wedge$ $\neg e$.atom $\models$
                $o'$.precondition $\wedge$ $e'$.condition $\wedge$ $e'$.atom):
                **continue** with next add effect $e$.
                { This add effect is balanced. }
        **reject** candidate. { The operator is unbalanced. }

**function** covers($V$, $\Phi$, $\psi$):
    **for each** $\varphi \in \Phi$:
        **if** the counted variables in $\varphi$ (those not in $V$)
          can be renamed so that $\varphi = \psi$:
            **return true**.
    **return false**.

**Fig. 9.2.** Algorithm for proving that an invariant candidate $\langle V, \Phi \rangle$ is an invariant

- We have noted before that we do not consider invariants involving derived predicates. This is because axioms correspond to operators that have a single add effect, but no delete effect. Invariant candidates including derived predicates can thus never be balanced, except if the axiom body is unsatisfiable, which is not a very interesting case. Since we do not consider derived predicates within invariants, we can ignore axioms completely during invariant synthesis.
- Instead of using full-blown satisfiability and entailment tests, more limited tests are possible if they only err in the "conservative" direction. In practice, Fast Downward only employs simple structural entailment tests. However, this is due to scarcity of development time, not to conserve runtime, and we intend to extend the test to more complete logical reasoning.

One final subtlety concerns the semantics of PDDL operators with conflicting effects. Note that our balance test does not special-case the possibility that $e$.atom equals $e'$.atom, i. e., that the same atom is added and deleted. For the PDDL semantics that we adhere to, an operator which would add and delete the same atom would be invalid and thus inapplicable, not threatening any invariant candidates. We call this the *consistent effect semantics*. However, under another commonly accepted semantics, the add effect would "win" in such a case. We call this the *add-after-delete semantics*. Using the *add-after-delete* semantics, we would need to add the condition $e$.atom $\neq e'$.atom before the entailment test in *check-operator-unbalanced*. We believe that there is no commonly agreed "correct" semantics for PDDL with regard to this issue: For some standard benchmarks, such as MysteryPrime, only the consistent effect semantics are reasonable, while for others, such as Rovers, only the add-after-delete semantics make sense. Without going into further details, we note that it is not too difficult to adjust the algorithm to use the add-after-delete semantics.

### 9.4.3 Refining Failed Candidates

As indicated in the overview of the invariant synthesis algorithm, we do not give up immediately if we cannot prove a given candidate to be an invariant. Instead, we try to *refine* it by adding atoms that can restore balance. In algorithmic terms, whenever we reject an invariant candidate $\langle V, \Phi \rangle$, we try to generate a set of new candidates of the form $\langle V, \Phi \cup \{\varphi'\} \rangle$.

Whether or not this is promising depends on the reason why the candidate was rejected. If it was rejected because an operator is too heavy, then no possible refinement that adds an atom to the candidate can change this fact, and we give up on the candidate completely. If, however, it was rejected because of unbalanced operators, there is hope that we can deal with the flaw by adding an atom that can match some delete effect of the threatening operator, balancing the unbalanced add effect.

The refinement algorithm is shown in Fig. 9.3. The actual implementation in Fast Downward does not generate all possible refining atoms $\varphi'$ naïvely,

**algorithm** refine-candidate($V$, $\Phi$):
  Select some schematic operator $o$ such that
    check-operator-unbalanced($o$, $V$, $\Phi$) fails.
  **for each** atom $\varphi'$ over variables from $V$ and at most one other variable
      for which covers($V$, $\Phi$, $\varphi'$) is not true:
    $\Phi' := \Phi \cup \{\varphi'\}$
    Simplify $\Phi'$ by removing atoms from $\Phi$ that are covered by $\varphi'$.
    (These cannot contribute to the weight of an instance of $\langle V, \Phi' \rangle$.)
    Simplify $\Phi'$ by removing unused parameters.
    **if** check-operator-too-heavy($o$, $V$, $\Phi'$) does not fail:
      Add $\langle V, \Phi' \rangle$ to the set of invariant candidates.

**Fig. 9.3.** Algorithm for refining an unbalanced invariant candidate $\langle V, \Phi \rangle$

but rather uses information from the set of delete effects of the threatening operator $o$ and the failed call to *check-operator-unbalanced* to only create candidates for $\varphi'$ for which there is a chance that the new balance check will succeed. Since this is conceptually straight-forward, we do not go into more detail about this technique.

Instead, let us return to the LOGISTICS example. Recall that candidate (9.1), $\forall x\, \mathtt{at}(x, l) \downarrow$, is threatened by the operator `unload-truck`, whose add effect $\mathtt{at}(x, l)$ is unbalanced. The operator has only one delete effect, namely $\neg\mathtt{in}(x, t)$. Indeed, $\mathtt{in}(x, t)$ is a suitable refinement atom for $\varphi'$ without further variable renaming, since the `unload-truck` operator is balanced with regard to the refined candidate $\forall x\, \mathtt{at}(x, l) + \mathtt{in}(x, t) \downarrow$. So we add this candidate to the set of currently considered candidates. At a later stage, it will be considered by *prove-invariant*, which will show that it is indeed an invariant.

By contrast, the other two candidates cannot be suitably refined. In order to refine (9.3), $\forall x, l\, \mathtt{at}(x, l) \downarrow$ to balance the `drive-truck` operator, we would need to add the atom $\mathtt{at}(x, l')$, which is the only delete effect of that operator. However, this atom covers the original atom $\mathtt{at}(x, l)$ (note that the converse is not true, because only $l'$ is a counted variable), leading to the candidate $\forall x, l\, \mathtt{at}(x, l')$ where parameter $l$ is unnecessary, so that it simplifies to $\forall x\, \mathtt{at}(x, l')$. This candidate is isomorphic to (9.2) and hence not considered again.

Considering candidate (9.2) and the `drive-truck` operator, the only possible refinement is $\forall \cdot\, \mathtt{at}(x, l') \downarrow$ ("The total number of `at` propositions is non-increasing"), which turns out to be violated by the `unload-truck` operator, but can be further refined to $\forall \cdot\, \mathtt{at}(x, l') + \mathtt{in}(x, l') \downarrow$ ("The total number of `at` and `in` propositions is non-increasing"). This latter candidate is actually an invariant. However, its only instance clearly has a weight greater than 1 in the initial state of any non-trivial LOGISTICS task and thus turns out not to provide any mutex information.

| | |
|---|---|
| Logistics | $\forall x\ \texttt{at}(x,l) + \texttt{in}(x,t) \downarrow$ |
| Blocksworld | $\forall\cdot\ \texttt{handempty}() + \texttt{holding}(b) \downarrow$ |
| | $\forall b\ \texttt{holding}(b) + \texttt{clear}(b) + \texttt{on}(b',b) \downarrow$ |
| | $\forall b\ \texttt{holding}(b) + \texttt{ontable}(b) + \texttt{on}(b,b') \downarrow$ |
| Grid | $\forall\cdot\ \texttt{armempty}() + \texttt{holding}(k) \downarrow$ |
| | $\forall\cdot\ \texttt{at-robot}(l) \downarrow$ |
| | $\forall\cdot\ \texttt{open}(d) + \texttt{locked}(d) \downarrow$ |
| | $\forall\cdot\ \texttt{locked}(d) \downarrow$ |
| | $\forall d\ \texttt{open}(d) + \texttt{locked(d)} \downarrow$ |
| | $\forall d\ \texttt{locked}(d) \downarrow$ |
| | $\forall k\ \texttt{holding}(k) + \texttt{at(k,l)} \downarrow$ |

**Fig. 9.4.** Invariants found in some standard benchmark domains

### 9.4.4 Examples

This concludes our description of the invariant synthesis algorithm. To give an impression of the kind of invariants it generates, Fig. 9.4 shows some of the results obtained on IPC domains. The invariants found in the Grid domain are most interesting, as they include some monotonicity information that is *not* a mutex: The third Grid invariant states that the total number of open and locked doors never increases, the fourth invariant states that the number of locked doors never increases, and the sixth invariant states that a door which is not locked can never become locked.

### 9.4.5 Related Work

Before moving on to the next translation step, we should point out that the algorithm described in this section is not the only approach to invariant synthesis proposed in the literature. Therefore, we now provide a brief comparison to four other approaches, sorted in decreasing order of relatedness:

- Edelkamp and Helmert's algorithm [34] proposed for the MIPS planner [35, 36],
- Gerevini and Schubert's DISCOPLAN [45, 46],
- Rintanen's invariant synthesis algorithm [101], and
- Fox and Long's TIM [27, 41].

We point out that apart from the first algorithm in the list, all of these were developed independently from ours, although all but the last one follow very similar ideas.

Edelkamp and Helmert's algorithm is the most closely related approach. In fact, Fast Downward's algorithm can be considered as the extension of the MIPS algorithm to non-STRIPS domains. Compared to the original algorithm, Fast Downward's invariant synthesis incorporates some cosmetic and

performance improvements, but the main difference is the coverage of universal and conditional effects. On STRIPS domains, both algorithms generate the same set of invariants.

DISCOPLAN uses a very similar guess, check and repair approach. However, the method for refining invariant candidates appears to be quite different, although this is somewhat difficult to assess because the algorithm is not completely described in the literature and source code of an implementation is not available. One major difference is that Fast Downward's algorithm immediately refines an invariant as soon as an operator is discovered which threatens it. DISCOPLAN, on the other hand, first collects all threats to an invariant for *all* operators, and only then generates refinements, which attempt to address all these threats at the same time. On the one hand, collecting threats across operators allows making more informed choices in invariant refinement. On the other hand, it appears that this approach incurs a performance penalty. For example, while Fast Downward's invariant synthesis algorithm always terminates in very short time for all IPC benchmark tasks, DISCOPLAN fails on 46 of the 50 IPC4 Airport tasks by running out of time. Another drawback of DISCOPLAN is that, although it is not limited to STRIPS, it can only deal with a subset of ADL features, which is not sufficient for the IPC benchmarks. Finally, also for STRIPS domains, there are some invariants important for an efficient MPT encoding which our algorithm discovers but DISCOPLAN misses. For example, in the Driverlog domain, our approach can prove that a given driver can only be at one place or inside one truck at the same time, which allows encoding driver location in a single variable. An encoding based on the invariants found by DISCOPLAN would need to introduce a separate state variable for each driver-location and driver-truck pair. On the positive side, DISCOPLAN can generate many classes of invariants beside mutexes; however, these are not relevant to PDDL-to-MPT translation.

Rintanen's algorithm follows the same guess-check-repair structure as our algorithm and DISCOPLAN. One main difference (and advantage) of Rintanen's algorithm is that its "check" step uses the information from *all* current invariant candidates, rather than just the one currently being considered, to strengthen the induction hypothesis. An interesting difference is that, unlike our algorithm, it always proceeds from stronger invariant candidates to weaker ones. Note that for inductive proofs, both strengthening and weakening an invariant candidate can be a promising refinement strategy. In particular, weaker statements are not necessarily easier to prove than stronger ones because the induction hypothesis is also weaker. A problem of Rintanen's algorithm is that it is limited to STRIPS and that it is not sufficiently efficient for many of the IPC benchmarks. For this reason, we have not made a detailed comparison regarding the kinds of invariants it can or cannot find; from our limited experience, we believe the approaches to be comparable in this respect, at least for the mutexes we are interested in. Like DISCOPLAN, Rintanen's approach can find more general classes of invariants.

Finally, Fox and Long's TIM (for *type inference module*) is (or can be interpreted as) an invariant synthesis algorithm which follows a conceptually very different approach to the other algorithms described here, focusing on the notion of *property spaces* which are generated from the type structure of the task, which is in turn based on a *type inference* technique which gives the system its name. TIM was originally [41] limited to STRIPS and thus not directly usable for us. It has since been extended to handle ADL constructs [27] in parallel to the development of our invariant synthesis algorithm.

## 9.5 Grounding

Having computed monotonicity invariants, the next translation step is to obtain a grounded representation of the normalized PDDL task.

**Definition 9.5.1.** ***Grounded PDDL Tasks***
*A* ***grounded PDDL task*** *is a PDDL task such that all literals occurring in the goal formula, axioms and operators are ground literals.*

Before performing the actual axiom and operator instantiation that yields the grounded representation, we try to determine which ground atoms of the PDDL task can actually become true. In a typical planning task, most ground atoms can never be true, either because they are not type-correct (for example, `at(vehicle1, vehicle2)`), or for more subtle reasons (for example, `at(vehicle1, loc1)` where there is no path from the initial location of `vehicle1` to `loc1`). Instantiating operators or axioms in such a way that their preconditions or bodies are necessarily false in every reachable state would be wasteful.

Determining whether or not a given atom can ever be true is as difficult as planning itself, but an over-approximation of the set of reachable atoms can be computed efficiently based on the idea of *relaxed planning tasks* in the sense of HSP and FF [16,68]. Instead of computing the set of reachable atoms of a PDDL task $\Pi$ itself, we thus compute the reachable atoms of a relaxed planning task $\mathcal{R}(\Pi)$, which differs from $\Pi$ as follows:

- Negative literals in axiom bodies, operator preconditions, effect conditions and goal condition are assumed to be always true.
- Delete effects of operators are ignored.

It is easy to see that the set of reachable atoms of $\mathcal{R}(\Pi)$ is a superset of the set of reachable atoms of $\Pi$, so any ground atoms not reachable in $\mathcal{R}(\Pi)$ need not be represented in the grounded version of $\Pi$.

The nice property of relaxed planning tasks is that computing their reachable atoms is conceptually simple. Nevertheless, this step is the most time-critical part of the whole translation component, because the set of reachable atoms can be huge in some of the benchmark domains, especially those with a

comparatively simple logical structure like LOGISTICS or SATELLITE. Therefore, it is important to compute reachable atoms efficiently. This is what the *Horn exploration* algorithm is designed for.

### 9.5.1 Overview of Horn Exploration

The idea of Horn Exploration is to encode the atom reachability problem for relaxed planning tasks as a set of logical facts and rules, i. e., as a logic program. This allows us to efficiently compute the set of reachable atoms by computing the *canonical model* of that logic program, which is the set of ground atoms implied by the program. The algorithm consists of three steps: Generating the logic program, translating it into a normal form, and computing its canonical model. Before going into detail for each of these steps, let us formally define what we mean by a logic program:

**Definition 9.5.2.** ***Positive Logic Programs***
*Let $\mathcal{L}$ be a first-order language.*

*A **positive Horn clause** over $\mathcal{L}$ is a formula of the form $\varphi_1 \wedge \cdots \wedge \varphi_k \rightarrow \psi$ ($k \geq 0$), where $\varphi_i$ and $\psi$ are (usually not ground) atoms over $\mathcal{L}$. It can be written as $\psi \leftarrow \varphi_1, \ldots, \varphi_k$. Using this notation, $\psi$ is called the **head** and $\varphi_1, \ldots, \varphi_k$ is called the **body** of the clause. Positive Horn clauses are usually assumed to be universally quantified. For a given positive Horn clause $\chi$ with free$(\chi) = \{v_1, \ldots, v_k\}$, we define $\chi_\forall = (\forall v_1 \ldots v_k : \chi)$. Similarly, for a set of positive Horn clauses $\mathcal{R}$, we define $\mathcal{R}_\forall = \{\ \chi_\forall \mid \chi \in \mathcal{R}\ \}$.*

*A **positive logic program** over $\mathcal{L}$ is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is a set of ground atoms over $\mathcal{L}$ called the set of **facts** and $\mathcal{R}$ is a set of positive Horn clauses over $\mathcal{L}$ called **rules**.*

*The **canonical model** of a positive logic program $\langle \mathcal{F}, \mathcal{R} \rangle$ is the set of all ground atoms $\varphi$ with $\mathcal{F} \cup \mathcal{R}_\forall \models \varphi$.*

Next, we show how to translate the reachability problem into a positive logic program. Afterwards, we demonstrate how to translate this logic program into a particularly simple form and how to compute the canonical model of the simplified logic program efficiently.

### 9.5.2 Generating the Logic Program

Due to the fact that the PDDL task has been normalized, generating the logic program is conceptually easy. A ground atom is reachable in the relaxed task iff it is true in the initial state or there exists some axiom or operator of the relaxed task that can make it true. Therefore, the set of facts of the logic program is formed by the atoms in the initial state of the planning task, and the set of rules is derived from the axiom and operator definitions. Additionally, we introduce a rule for the goal of the planning task to detect solvability of the relaxed task; if it is unsolvable, the original task is unsolvable too, which we can report immediately and stop planner execution.

Recall from Sect. 9.3 that at this stage, all conditions occurring in the PDDL task are conjunctions of literals. For such conjunctions $\varphi$, we denote the conjunction of all *positive* literals of $\varphi$ by $\varphi^+$. In the context of logic programs, we follow the PROLOG convention of using uppercase letters for first-order variables and lower-case letters for constants. The exploration rules for a normalized PDDL tasks are generated as follows:

- *Axioms:* For schematic axioms $a = \varphi \leftarrow \psi$ with $\psi^+ = \psi_1^+ \wedge \cdots \wedge \psi_m^+$ and $\text{free}(\varphi) \cup \text{free}(\psi) = \{X_1, \ldots, X_k\}$, we generate the *axiom applicability rule*

  $$a\texttt{-applicable}(X_1, \ldots, X_k) \leftarrow \psi_1^+, \ldots, \psi_m^+.$$

  and the *axiom effect rule*

  $$\varphi \leftarrow a\texttt{-applicable}(X_1, \ldots, X_k).$$

- *Operators:* For schematic operators $o$ with parameters $\{X_1, \ldots, X_k\}$ and precondition $\varphi$ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *operator applicability rule*

  $$o\texttt{-applicable}(X_1, \ldots, X_k) \leftarrow \varphi_1^+, \ldots, \varphi_m^+.$$

  and for each add effect $e$ of $o$ adding the atom $\psi$ with quantified variables $\{Y_1, \ldots, Y_l\}$ and effect condition $\varphi$ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *effect trigger rule*

  $$e\texttt{-triggered}(X_1, \ldots, X_k, Y_1, \ldots, Y_l) \leftarrow o\texttt{-applicable}(X_1, \ldots, X_k), \varphi_1^+, \ldots, \varphi_m^+.$$

  and *effect rule*

  $$\psi \leftarrow e\texttt{-triggered}(X_1, \ldots, X_k, Y_1, \ldots, Y_l).$$

- *Goal rule:* For the goal $\varphi$ with $\varphi^+ = \varphi_1^+ \wedge \cdots \wedge \varphi_m^+$, we generate the *goal rule*

  $$\texttt{goal-reachable}() \leftarrow \varphi_1^+, \ldots, \varphi_m^+.$$

The correctness of these rules should be evident. The reader might wonder why we sometimes introduce new predicates that do not seem necessary. For example, axiom applicability rule and axiom effect rule could be combined into a single rule without introducing the auxiliary predicate $a$-`applicable`. The purpose of these predicates is to track which axioms and operators must be instantiated when grounding the PDDL task. For example, in the LOGISTICS domain, we will not generate a ground operator `(fly-airplane plane1 loc1 loc3)` if `loc3` is not an airport location, since in this case the canonical model of the logic program does not include the atom `fly-airplane-applicable`(`plane1`, `loc1`, `loc3`). The operator applicability predicates serve the additional purpose of "factoring out" common subexpressions. Without them, all operator preconditions would need to be repeated in each effect trigger rule (or effect rule, if effect trigger rules were similarly eliminated).

### 9.5.3 Translating the Logic Program to Normal Form

After the logic program has been generated, it is translated into the following normal form:

**Definition 9.5.3.** ***Normal Form for Positive Logic Programs***
*An atom in first-order logic is called* ***variable-unique*** *if it does not contain two occurrences of the same variable. (For example, an atoms like $P(X, Y, X)$ is not variable-unique because variable $X$ occurs twice. Repetitions of constants are allowed.)*

*A rule of a positive logic program is called* ***variable-unique*** *if the head and all atoms of the body are variable-unique.*

*A rule of a positive logic program is called a* ***projection rule*** *if it is variable-unique and it is of the form $\varphi \leftarrow \varphi_1$ with $free(\varphi) \subseteq free(\varphi_1)$. In other words, projection rules are unary rules where all variables in the head occur in the body.*

*A rule of a positive logic program is called a* ***join rule*** *if it is variable-unique and it is of the form $\varphi \leftarrow \varphi_1, \varphi_2$ with $free(\varphi_1) \cup free(\varphi_2) = free(\varphi) \cup (free(\varphi_1) \cap free(\varphi_2))$. In other words, join rules are binary rules where all variables occurring in the head occur in the body, and all variables occurring in the body but not in the head occur in* both *atoms of the body.*

*A positive logic program is* ***in normal form*** *if all rules are either projection rules or join rules.*

The names of the rule types in Definition 9.5.3 are reminiscent of the related database-theoretic operations from relational algebra: Projection rules correspond to the projection operator $\pi$ and join rules correspond to the natural join operator $\bowtie$ (or strictly speaking, a combination of natural join and projection). We will now describe how to convert the positive logic program from the previous section into normal form.

First, we eliminate duplicate variable occurrences as follows: If any rule contains atoms with duplicate occurrences of the same variable $X$, we change one occurrence of $X$ in any such atom into a new variable $X'$ and add the atom $\texttt{equals}(X, X')$ to the body of the rule. We repeat until no further such transformations are possible. If we needed to introduce any `equals` atoms, we add the fact $\texttt{equals}(o, o)$ to the initial state for each object $o$ of the planning task.

Second, for any variable $X$ that occurs in the head but not in the body of a rule, we add the atom $\texttt{object}(X)$ to the rule body. (Remember from Sect. 9.3.1 that $\texttt{object}(o)$ is true for any object $o$ of the planning task.)

Third, all rules with an empty body are converted into facts. Their heads must be ground atoms because all variables occurring in the head must occur in the (in this case, empty) body after the previous transformation.

After these transformations, all remaining unary rules are projection rules; we still need to normalize rules with two or more atoms in the body. As a first step towards this goal, we determine if the body of such a rule contains any

**algorithm** greedy-join(*rule*):
  **while** |*rule*.body| > 2:
    Choose $\varphi$, $\varphi' \in$ *rule*.body such that $\varphi \neq \varphi'$ and
      join-cost(*rule*, $\varphi$, $\varphi'$) is minimal.
    $X_1, \ldots, X_k :=$ join-vars(*rule*, $\varphi$, $\varphi'$)
    Generate a new predicate symbol $p$ with arity $k$.
    Generate a new join rule $p(X_1, \ldots, X_k) \leftarrow \varphi, \varphi'$.
    *rule*.body := *rule*.body $\setminus \{\varphi, \varphi'\} \cup \{p(X_1, \ldots, X_k)\}$

**function** join-vars(*rule*, $\varphi$, $\varphi'$):
  { Compute the relevant variables for the predicate generated
    by joining $\varphi$ and $\varphi'$. }
  **return** free($\{\varphi, \varphi'\}$) $\cap$ free($\{$*rule*.head$\} \cup$ *rule*.body $\setminus \{\varphi, \varphi'\}$).

**function** join-cost(*rule*, $\varphi$, $\varphi'$):
  *new-arity* := |join-vars(*rule*, $\varphi$, $\varphi'$)|
  *max-old-arity* := max(|free($\varphi$)|, |free($\varphi'$)|)
  *min-old-arity* := min(|free($\varphi$)|, |free($\varphi'$)|)
  **return** (new-arity − max-old-arity, new-arity − min-old-arity, new-arity).
  { Cost estimates are triples which are compared lexicographically.
    We prefer joins where "new arity" − "max old arity" (the increase
    in arity) is small and consider the other criteria only in case of ties. }

**Fig. 9.5.** The greedy join algorithm for decomposing a rule into join rules

variables that occur in no other atom of the rule, neither in the body nor in the head. If this is the case, such variables are projected away as follows: We are given the rule $\varphi \leftarrow \varphi_1, \ldots, \varphi_k$, where free($\varphi_i$) $= \{X_1, \ldots, X_k\}$ contains variables not present in any of the other atoms, say $\{X_{j+1}, \ldots, X_k\}$. Then we introduce a new predicate $p$ and replace the original rule by the two rules $\varphi \leftarrow \varphi_1, \ldots, \varphi_{i-1}, p(X_1, \ldots, X_j), \varphi_{i+1}, \ldots, \varphi_k$ and $p(X_1, \ldots, X_j) \leftarrow \varphi_i$.

After this transformation, all binary rules are valid join rules. In the last normalization step, we split rules with $m > 2$ atoms in the body into $m - 1$ join rules by applying the *greedy join algorithm*, illustrated in Fig. 9.5. The algorithm iteratively picks two atoms from the rule body and joins them, introducing a new predicate for the result of the join and replacing the two atoms in the rule body by an instance of that new predicate. This process is repeated until the body of the rule no longer contains more than two atoms.

The order in which atoms are joined is critical for the speed of evaluating the join rules. To see this, consider the rule $p(X) \leftarrow q(X), s(X, Y), t(Y)$. One possible decomposition into join rules yields the rules $u(X) \leftarrow s(X, Y), t(Y)$ and $p(X) \leftarrow q(X), u(X)$. Another possible decomposition yields the rules $v(X, Y) \leftarrow q(X), t(Y)$ and $p(X) \leftarrow s(X, Y), v(X, Y)$. We can expect that the canonical model of the first decomposition contains relatively few instances of the intermediate predicate $u$, maybe about as many as it contains instances

of $t$. On the other hand, the canonical model of the second decomposition contains as many instances of the intermediate predicate $v$ as the product of the number of instances of $q$ and $t$, which can be much higher.

Since the performance of our algorithm for computing the canonical model of a logic program is closely related to the model size, we prefer to generate smaller intermediate results. The greedy join algorithm tries to achieve this goal by preferring to join atoms that contain many common variables and lead to intermediate predicates of low arity.

### 9.5.4 Computing the Canonical Model

Having translated the logic program into normal form, we are ready to compute the canonical model. We use a queue-based approach, distinguishing between reachable atoms that have already been processed, which means that the consequences of their being reachable have already been evaluated (*closed atoms*), and reachable atoms that still need to be processed (*open atoms*). Open atoms are those that are currently stored in the queue, while closed atoms are those that were enqueued once, but no longer are.

Our algorithm, shown in Fig. 9.6, stores open atoms in the *queue* variable, while both open and closed atoms are stored in the result variable *canonical-model*. Additionally, it uses the following data structures:

- *Rule matcher*: A rule matcher is an indexing structure that supports efficient unification queries on the bodies of logic programs. When given a ground atom $a$, the rule matcher determines all projection rules $\varphi \leftarrow \varphi_1$ and join rules $\varphi \leftarrow \varphi_1, \varphi_2$ such that $\varphi_1$ or $\varphi_2$ unifies with $a$, i. e., such that it is possible to substitute objects for variables in $\varphi_1$ or $\varphi_2$ in such a way that $a$ is obtained. The rule matcher reports the matched rules and whether $\varphi_1$ or $\varphi_2$ was matched (if both unify with $a$, two matches are generated).
  Note that matching ground atoms to the rules they can trigger is simple if the rules do not contain constants in the body. Unfortunately, some of the IPC4 benchmarks contain a huge number of operator schemas involving constants (most importantly, the STRIPS formulation of the AIRPORT domain), and an efficient indexing structure is important for those. Rule matchers are implemented as decision-tree like data structures very similar to *successor generators*, which are discussed in Chap. 10. Because of that similarity and because they are not central to the instantiation algorithm, we do not discuss rule matchers further.
- *Join rule indices:* Each join rule $r = \varphi \leftarrow \varphi_1, \varphi_2$ maintains two hash tables $r.\text{index}_1$ and $r.\text{index}_2$ that map instantiations of the common variables of $\varphi_1$ and $\varphi_2$ to instantiations of the variables of $\varphi_1$ and $\varphi_2$, respectively.
  At any time (except during updates) and for any assignment *key* to the common variables of $\varphi_1$ and $\varphi_2$, $r.\text{index}_1[key]$ contains those variable mappings $\alpha \supseteq key$ for the variables of $\varphi_1$ for which $\alpha(\varphi_1)$ belongs to the closed

```
algorithm calculate-canonical-model(F, R):
    for each join rule r ∈ R:
        r.index₁ := make-empty-hashtable()
        r.index₂ := make-empty-hashtable()
    rule-matcher := build-rule-matcher(R)
    queue := make-queue(F)
    canonical-model := F
    { In the following, enqueuing a fact means adding it to queue and
      canonical-model if it is not yet an element of canonical-model. }
    while queue is not empty:
        current-fact := queue.pop()
        for each match m ∈ rule-matcher.match(current-fact):
            if m refers to φ₁ in a projection rule r = φ ← φ₁:
                Let α be the variable assignment
                  for which α(φ₁) = current-fact.
                Enqueue α(φ).
            else if m refers to φ₁ in a join rule r = φ ← φ₁, φ₂:
                Let α be the variable assignment
                  for which α(φ₁) = current-fact.
                key := α restricted to free(φ₁) ∩ free(φ₂)
                Add α to r.index₁[key].
                Enqueue (α ∪ β)(φ) for each β ∈ r.index₂[key].
            else if m refers to φ₂ in a join rule r = φ ← φ₁, φ₂:
                { Handled analogously to the previous case. }
```

**Fig. 9.6.** Computing the canonical model of a positive logic program $\langle \mathcal{F}, \mathcal{R} \rangle$ in normal form

set. Similarly, $r.\text{index}_2[key]$ contains those variable mappings $\beta \supseteq key$ for the variables of $\varphi_2$ for which $\beta(\varphi_2)$ belongs to the closed set.
This information can be exploited for quickly determining all possible instantiations of $\varphi_2$ that match a given instantiation of $\varphi_1$, or vice versa, as is done in the algorithm. Note that the variable assignment $\alpha \cup \beta$ considered in the algorithm is indeed a function, since $\alpha$ and $\beta$ agree on all variables for which they are both defined.

To motivate the soundness of *compute-canonical-model*, we state an important invariant which holds before and after each iteration of the **while** loop: *All non-closed atoms which can be derived in one step from the closed atoms using the rules of the logic program* $\langle \mathcal{F}, \mathcal{R} \rangle$ *are open atoms.* This implies that upon termination of the algorithm, when there are no more open atoms and hence *canonical-model* holds exactly the set of closed atoms, the model is closed under application of $\mathcal{R}$. Because it also contains all facts from $\mathcal{F}$ and only contains facts that can be derived from $\mathcal{F}$, it thus contains exactly the canonical model of $\langle \mathcal{F}, \mathcal{R} \rangle$.

The invariant is obviously true initially, since there are no closed atoms at the beginning of the algorithm. With our descriptions of the data structures of *compute-canonical-model*, the reader should have no trouble verifying that it remains true after each iteration of the **while** loop.

This concludes our discussion of the Horn exploration algorithm. One final word on performance: If we assume that the arity of predicates in the logic program is bounded by a constant, then all basic operations of *calculate-canonical-model* can be performed in constant time. The runtime of the algorithm then typically scales roughly linearly in the combined size of its input and output (the computed canonical model). However, runtime can be worse if there are many situations where the algorithm tries to enqueue an atom that is already part of the canonical model.

### 9.5.5 Axiom and Operator Instantiation

With the help of the canonical model, instantiating axioms and operators is very straight-forward. To compute the grounded representation, we scan through the set of ground atoms in the canonical model in the order in which they were generated, creating axiom and operator instances as follows:

- When encountering atoms of the form $a$-`applicable`$(x_1, \ldots, x_k)$ where $a$ is a schematic axiom, we generate a ground instance of $a$ with the parameters substituted with $x_1, \ldots, x_k$.
- When encountering atoms of the form $o$-`applicable`$(x_1, \ldots, x_k)$ where $o$ is a schematic operator, we generate a ground instance of $o$ without effects. Like in the case of axioms, the parameters of the operator are substituted with $x_1, \ldots, x_k$, and the precondition is instantiated accordingly.
- When encountering atoms of the form $e$-`triggered`$(x_1, \ldots, x_k, y_1, \ldots, y_l)$ where $e$ is an effect of some operator $o$, we look up the set of already generated ground operators to find the operator $o(x_1, \ldots, x_k)$. This operator must have been generated previously because an $e$-`triggered` atom can only be derived after the corresponding $o$-`applicable` atom. Having found the ground operator, we attach to it the effect obtained by instantiating the variables in $e$ with $y_1, \ldots, y_l$.

After a single pass through the canonical model, we have thus generated a grounded PDDL task which is equivalent to the normalized PDDL task we started with.

## 9.6 Multi-valued Planning Task Generation

Together with the invariants synthesized earlier, the grounded PDDL task generated in the previous stage provides all the information we need for transforming the STRIPS task into a multi-valued planning representation, which constitutes the final translation step.

```
algorithm compute-mutex-groups(invariants, P_f, s_0):
    for each invariant I ∈ invariants:
        for each instance α of I:
            if weight(α, s_0) = 1:
                Create a mutex group containing all atoms in P_f
                  covered by α.
```

**Fig. 9.7.** Computing mutex groups from the set of monotonicity invariants *invariants*, the set of reachable atoms $\mathcal{P}_f$ and the initial state $s_0$

Recall from Definition 9.1.2 that a multi-valued planning task (MPT) is given by a 5-tuple $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$ of variables $\mathcal{V}$, each with an associated finite domain, initial state $s_0$ and goal $s_\star$, axioms $\mathcal{A}$ and operators $\mathcal{O}$. We start by defining suitable variables and variable domains; everything else then more or less falls into place.

### 9.6.1 Variable Selection

Each variable of the generated MPT corresponds to one or more (reachable) ground atoms of the STRIPS task. We start by enumerating the set $\mathcal{P}$ of all such atoms, partitioned into atoms $\mathcal{P}_f$ which are instances of *modifiable fluent predicates* or *derived predicates* and atoms $\mathcal{P}_c$ which are instances of *constant predicates* (cf. Sect. 9.4.1).

We want to represent as many ground atoms by a single state variable as possible. To achieve this, we first determine the set of *mutex groups* induced by the computed invariants. Mutex groups are computed in a straightforward manner by instantiating the monotonicity invariants in all possible ways, checking for each if it has weight 1 in the initial state, and if so, which atoms from $\mathcal{P}_f$ it covers. The algorithm is shown in Fig. 9.7; the actual implementation in Fast Downward uses an indexing structure for efficiently determining the set of reachable atoms covered by a given invariant instance.

Normally, not every mutex group will correspond to an MPT state variable, since the same atom can be part of several mutex groups, but of course only needs to be encoded once. As an example of this phenomenon, consider Fig. 9.8, which shows the mutex groups of a BLOCKSWORLD task with four blocks. If, for example, we decide to encode mutex groups (1)–(4) with four multi-valued state variables, then we only need to encode one atom from each of the other groups, since all instance of the `on` and `holding` predicates are already represented. Therefore, the translator would first generate four state variables with domains consisting of seven values each, namely `holding`$(x)$, `clear`$(x)$, `on`$(\mathtt{a}, x)$, `on`$(\mathtt{b}, x)$, `on`$(\mathtt{c}, x)$, `on`$(\mathtt{d}, x)$ and the seventh option "none of the other six is true". (Of these seven values, two – block $x$ being on top of itself and none of the six atoms being true – are actually impossible.) After-

(1) {holding(a), clear(a), on(a, a), on(b, a), on(c, a), on(d, a)}
(2) {holding(b), clear(b), on(a, b), on(b, b), on(c, b), on(d, b)}
(3) {holding(c), clear(c), on(a, c), on(b, c), on(c, c), on(d, c)}
(4) {holding(d), clear(d), on(a, d), on(b, d), on(c, d), on(d, d)}
(5) {holding(a), ontable(a), on(a, a), on(a, b), on(a, c), on(a, d)}
(6) {holding(b), ontable(b), on(b, a), on(b, b), on(b, c), on(b, d)}
(7) {holding(c), ontable(c), on(c, a), on(c, b), on(c, c), on(c, d)}
(8) {holding(d), ontable(d), on(d, a), on(d, b), on(d, c), on(d, d)}
(9) {holding(a), holding(b), holding(c), holding(d), handempty()}

**Fig. 9.8.** Mutex groups for a BLOCKSWORLD task with four blocks. Some atoms, e. g. on(a, a), are reachable in the relaxed task although they are never true in the "real" task

wards, it would encode the truth values of the remaining atoms ontable($x$) and armempty() with binary state variables.

In this case, there was at least one atom in each mutex group that was unique to this particular group, so that the resulting encoding is not much better than an encoding which simply takes all mutex groups and introduces a state variable for each. However, in other cases, one group can be completely covered by others; examples of this can be found in the AIRPORT domain. In this case we would like to cover the set of reachable atoms with as few state variables as possible.

Unfortunately, as we have seen in Part II, set cover problems of this kind are NP-complete [43, problem SP5] and indeed not even $c$-approximable [7], so we limit our covering efforts to the greedy algorithm shown in Fig. 9.9, which is the best approximation algorithm known for this problem, achieving an $O(\log n)$-approximation [7]. Iteratively, we pick a mutex group $P$ of maximal cardinality and introduce a new MPT state variable with domain $P \cup \{\bot\}$, where $\bot$ stands for "none of the elements of $P$ is true". We then remove all covered elements from all other mutex groups, removing groups that no longer contain more than one element. This process is repeated until all mutex groups have been removed. At this stage, the remaining uncovered atoms $p$ are represented by binary variables with domain $\{p, \bot\}$.

After execution of the algorithm, for each reachable atom $p \in \mathcal{P}_f$ there is exactly one MPT variable whose domain includes $p$. The translation will ensure that this variable, which we denote as $var(p)$ in the following, assumes the value $p$ in an MPT state iff $p$ is true in the corresponding state of the PDDL task. With this information, we can now go about converting the rest of the PDDL task to the MPT representation.

### 9.6.2 Converting the Initial State

We start by converting the initial state, which is the easiest step. For each atom $p \in \mathcal{P}_f$ that is in the initial state, we set the initial value of $var(p)$ to

**algorithm** choose-variables($\mathcal{P}_f$, *mutex-groups*):
    *uncovered* := $\mathcal{P}_f$
    **while** *mutex-groups* $\neq \emptyset$:
        Pick a mutex group $P$ of maximal cardinality.
        Create an MPT variable $v$ with domain $\mathcal{D}_v = P \cup \{\bot\}$.
        *uncovered* := *uncovered* $\setminus P$
        *mutex-groups* := $\{\ P' \setminus P \mid P' \in$ *mutex-groups* $\}$
        *mutex-groups* := $\{\ P' \mid P' \in$ *mutex-groups* $\wedge\ |P'| \geq 2\ \}$
    Create an MPT variable $v$ with domain $\{p, \bot\}$ for all remaining elements of *uncovered*.

**Fig. 9.9.** Greedy algorithm for computing the MPT variables and variable domains

$p$. MPT variables for which there is no initial state atom $p$ with $var(p) = p$ are initialized to $\bot$. Note that different initial state atoms $p$, $p'$ must satisfy $var(p) \neq var(p')$, because $p$ and $p'$ could only be represented by the same MPT variable if they were mutually exclusive, which implies their not being in the initial state together. Therefore, the converted initial state is indeed well-defined.

### 9.6.3 Converting Operator Effects

Translating the state changes incurred by operator effects requires some care. For add effects setting an atom $p$ to true, conversion is easy: Such an effect is always translated to an MPT effect setting $var(p)$ to $p$, because we know $p$ to be true after operator application if the effect fires.

However, for delete effects setting an atom $p$ to false, the correct translation is not as clear. We cannot simply set $var(p)$ to $\bot$ ("none of the variables represented by $var(p)$ is true") unconditionally, because this is not always correct: It could be the case that another effect of the same operator triggers simultaneously and adds another atom represented by the same variable, or that $p$ *was not true* when the operator was applied, but some other atom represented by $var(p)$ was.

Therefore, the correct translation is to set $var(p)$ to $\bot$ only if we know that $p$ was previously true and that no effect adding an atom represented by $var(p)$ triggers simultaneously, and not to do anything if this is not the case. If the other effects of the operator that add atoms represented by $var(p)$ have effect conditions $\chi_1, \ldots, \chi_k$, then this is achieved by adding $p \wedge \neg\chi_1 \wedge \cdots \wedge \neg\chi_k$ to the effect condition of the delete effect.

If some of the formulas $\chi_i$ are proper conjunctions (i. e., neither constant true nor singleton literals), this results in an effect condition which is not a conjunction of literals. In this case, we introduce a new derived variable $v_i$ that evaluates to true whenever $\neg\chi_i$ is true, and use $v_i$ in the effect condition instead.

All things considered, this conversion of delete effects looks very complicated, and indeed in most cases easier translations are possible. For this purpose, we detect two common special cases, with which we deal differently:

- If we see that whenever the delete effect triggers, some add effect affecting the same variable must trigger as well, because it has the same or a more general effect condition, then we do not need to represent the delete effect in the MPT at all. The add effect will take care of the value change of its affected variable.
- On the other hand, if we see that no add effect affecting the same variable can trigger at the same time, because no such effect exists or each of their effect conditions is inconsistent with the condition of the delete effect, then we can convert the delete effect to an effect setting $var(p)$ to $\bot$. If $p$ is not already part of the operator precondition or effect condition, we must add it to the effect condition to make sure that $var(p)$ is only cleared if it was previously set to $p$.

In most cases, translating delete effects is straight-forward because the two simpler cases are by far more common than the general case. In particular, for operators without conditional effects, one of the special cases always applies.

### 9.6.4 Converting Conditions

The third major translation step is the conversion of grounded conditions of the PDDL task, which occur in the goal, in operator preconditions and effect conditions and in axiom bodies.

To translate a grounded condition, we first check if it contains any atoms not in $\mathcal{P}_f$. These have constant truth values, so that the condition can be simplified accordingly. If this leads to a constant false condition, we react accordingly (for the goal, we report that the task is unsolvable; for axiom bodies, operator preconditions or effect conditions, we remove the axiom, operator or effect).

Having considered trivially false conditions, we translate each positive literal $p$ in the condition to the pairing $var(p) = p$. Translating negative literals $\neg p$ is slightly more tricky. Recall the Blocksworld example discussed earlier, where we generated the MPT state variable $v$ with $\mathcal{D}_v = \{\texttt{holding(a)}, \texttt{clear(a)}, \texttt{on(a,a)}, \texttt{on(b,a)}, \texttt{on(c,a)}, \texttt{on(d,a)}, \bot\}$, and consider a condition including the atom $\neg\texttt{on(c,a)}$. If the condition also contains some positive literal concerning variable $v$, for example the atom $\texttt{clear(a)}$, then we do not need to encode $\neg\texttt{on(c,a)}$ at all, because it is implied by the other literal. However, otherwise there is no simple way to represent $\neg\texttt{on(c,a)}$ as an MPT condition. We would need to write something like $v \neq \texttt{on(c,a)}$, but conditions of this form are not supported by the representation.

Therefore, in situations like this, similar to what we did when translating difficult effect conditions that arise for complicated delete effects, we introduce a new derived variable $\texttt{not-}p$ with domain $\{\top, \bot\}$ and generate an axiom

$(\mathtt{v} = d) \rightarrow (\mathtt{not\text{-}}p := \top)$ for each value $d \in \mathcal{D}_v \setminus \{p\}$. The pairing $\mathtt{not\text{-}}p = \top$ can then serve as a translation of the literal $\neg p$.

If we wanted to avoid introducing new axioms, we could further normalize the PDDL task so that no negative literals occur in conditions. There are well-known translation methods to achieve such a normal form, but for our purposes, our method has the advantage that no new non-derived state variables are introduced, keeping the memory requirements of search states small.

### 9.6.5 Computing Axiom Layers

As a final translation step, we must compute the axiom layers for the MPT representation so that the semantics match with those of stratified logic programs (cf. Definitions 9.1.1 and 9.1.2).

This is done as follows: Whenever the body of an axiom $a$ includes the condition $v = \bot$ for some derived variable $v$, then all axioms $a'$ with affected variable $v$ must be evaluated before $a$, i. e. we introduce an ordering constraint $a' \prec a$. If the axiom definitions of the original PDDL task corresponded to a stratifiable logic program, then the graph containing all such ordering constraints will be acyclic. Thus, we can use a topological sort algorithm to assign the individual axioms to axiom layers: The first axiom layer contains all axioms without predecessors in the graph, the second axiom layer contains all axioms whose predecessors belong to the first layer, and so on, until layers are assigned to all axioms.

### 9.6.6 Generating the Output

Having partitioned the axioms into layers, we have finished translating the PDDL task. Before generating output, the translator applies a few post-processing techniques to simplify the generated task where possible.

Most importantly, if there are two axioms with the same head, $a = (cond \rightarrow v := d)$ and $a' = (cond' \rightarrow v := d)$ with $cond \subset cond'$, then $a$ is triggered whenever $a'$ is triggered, so $a'$ is unnecessary. In such a case, which occurs frequently in domains where axioms encode transitive closures, we say that $a$ *dominates* $a'$ and only keep $a$. Similarly, we do not keep several copies of the same axiom which only differ in the order in which the conditions are listed.

Once post-processing is completed, the generated MPT is written to disk in a simple text format suitable for easy parsing by the other components of the planner.

## 9.7 Performance Notes

Before moving on to the other components of Fast Downward, let us briefly discuss the performance of the translation component and compare it to the related MIPS system.

### 9.7.1 Relative Performance Compared to MIPS Translator

As discussed in detail in Sect. 8.2, there are a number of other approaches to planning that use multi-valued planning tasks or similar formalisms as their base input. However, there exists only one earlier approach exploiting multi-valued planning tasks within a *PDDL planner* and thus requiring the sort of translation described in this chapter, namely the MIPS planning system [35]. In many ways, the Fast Downward translator can be considered a further development of the MIPS translator, which is described in an article by Edelkamp and Helmert [34].

The main difference between the MIPS translator as described in that article and the Fast Downward translator described in this chapter is that the latter is more general. The original MIPS algorithm cannot deal with ADL-style conditions or effects, with derived predicates, or with schematic operators involving constants. Moreover, its runtime typically scales exponentially in the number of schematic operators. This is not very problematic for constant-free STRIPS domains which typically exhibit a small number of operators, constant across the tasks of the domain. However, some of the IPC4 domains contain partially pre-instantiated operators leading to very large domain specifications. For example, 15 of the 50 AIRPORT tasks (STRIPS formulation) of IPC4 contain more than 1300 schematic operators each. Dealing better with such high numbers of schematic operators was one of the key motivations for our new developments in invariant synthesis (Sect. 9.4) and grounding (Sect. 9.5). We point out that Edelkamp has independently extended the translation algorithm of MIPS since IPC2. However, there is no published work on these efforts, so we do not provide a comparison.

To compare the relative performance of the original MIPS translator and Fast Downward's translator, we applied both to those 566 IPC1–4 benchmark tasks that the MIPS translator can handle, i. e., pure STRIPS tasks without domain constants. This is a somewhat unfair problem suite for the Fast Downward translator because its key performance improvements are in efficiently dealing with complex domain descriptions during grounding – a complication which does not arise for these benchmarks. We should also point out that the MIPS translator is implemented in C++, whereas Fast Downward's translator is implemented in the higher-level Python language. It is reasonable to expect that a C++ reimplementation of the translator could lead to a speedup of at least one order of magnitude on large tasks.

The overall result of our comparison is that in general, the MIPS translator is clearly the faster of the two systems. To discount the impact of very

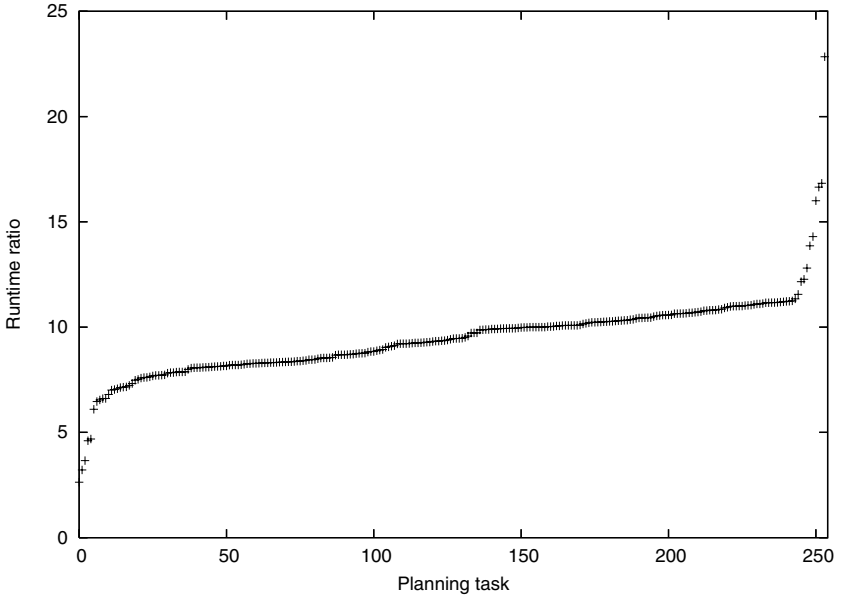


**Fig. 9.10.** Runtime comparison between the MIPS translator (faster) and Fast Downward translator (slower). Each data point corresponds to one planning task. Data points are sorted by runtime ratio

easy tasks, we further limited the benchmark set to those tasks for which either translator required at least one second of runtime, obtaining a total of 254 data points. For all of these, the MIPS translator was the faster of the two algorithms (or implementations). Figure 9.10 shows the ratio between the runtimes of the two translators on these 254 benchmark tasks. In 243 out of 254 cases, this ratio is between 6 and 13, with five outliers below and six outliers above this region. Note that on the horizontal axis, task are sorted by the observed runtime ratio, not by task size or any other scaling measure, so the upward slope of the curve cannot be interpreted as any kind of asymptotical scaling behaviour. This mode of display was chosen because there was no visible correlation between the observed speedup and task size (or some other apparent measure of task complexity). In other words, the algorithms appear to scale equally well, which is no surprise given that our translation algorithm is very similar to the MIPS translator on this fragment of PDDL.

Regarding the kinds of translations they generate, the two translators are interchangeable. For this reason, as a speed optimization, Fast Downward can be configured to use the C++-based MIPS translator on those domains for which it is applicable. (The performance results in the following section are all with respect to the Python-based Fast Downward translator, in order to allow comparisons across domains.)

| measure | mean ± dev. | 25% | median | 75% | max. |
|---|---|---|---|---|---|
| translation time | 12.35 ± 82.52 s | 0.39 s | 1.19 s | 4.28 s | 1960.0 s |
| size of output | 672 ± 3253 KB | 23 KB | 93 KB | 326 KB | 81748 KB |
| state variables | 565 ± 3176 | 25 | 55 | 240 | 61842 |
| operators/axioms | 7046 ± 38308 | 289 | 1009 | 3469 | 989250 |

**Fig. 9.11.** Some statistics on the performance and output characteristics of the translation component. The first column shows the aspect of the task being measured, the second column shows the mean and standard deviation for the respective measure, and the remaining columns show the 25%, 50% (median), 75% and 100% (maximum) percentiles. Statistics are based on the 1442 propositional benchmarks from the fully automated tracks of IPC1–4

### 9.7.2 Absolute Performance

In addition to comparisons to other similar techniques, it is of course also of importance how fast the translator computes its result in *absolute* terms. On a state-of-the-art computer, it is sufficiently efficient to generate MPT encodings for all 1442 IPC1–4 benchmark tasks. Moreover, compared to the time required in the search component, translation time is essentially negligible in the vast majority of cases. (Some exceptions to this exist in "structurally simple" domains like SATELLITE and LOGISTICS.) A short summary of "average" performance for the translator (for different notions of average) is provided in Fig. 9.11. All experiments were conducted on a machine with a 3.066 GHz Intel Xeon CPU, setting a memory limit of 2 GB.

To get an impression of the size and translation cost of a "typical" task, the mean values, which are heavily influenced by some very large PSR tasks, are misleading. The comparatively high standard deviations show that the distributions are highly irregular, so the percentile information is probably more meaningful than the mean values. To get an impression of what very large planning tasks look like, Fig. 9.12 provides information about the "largest" five input tasks according to each of the four measures *translation time*, *encoding size of translated task*, *no. of state variables in translated tasks* and *no. of operators and axioms in translated tasks*. As an extreme example, the largest PSR instance from IPC4, task PSR-LARGE #50, could only just be translated within the memory bound, consuming more than 1.9 GB of RAM before completing translation after 32:40 minutes. However, with 61842 relevant state variables, almost all of them derived variables, this task is far from being solvable with current domain-independent planning technology anyway. For comparison, at IPC4, Fast Downward could only solve the PSR-LARGE instances up to #31. The largest solved PSR instance comprises 5807 relevant MPT state variables and needed 39 seconds for translation. Of the other competitors, the best system could solve 11 tasks of the PSR-LARGE benchmark set, of which the largest comprises 527 state variables. MPT translation took 4 seconds for this instance.

| **translation time** | **task** | **time** |
|---|---|---|
| | PSR-Large #50 | 1 960.00 s |
| | Satellite #33 | 1 355.50 s |
| | PSR-Large #48 | 1 145.80 s |
| | PSR-Large #46 | 920.45 s |
| | Satellite #32 | 634.80 s |
| **size of output** | **task** | **size** |
| | Satellite #33 | 81 748 KB |
| | Satellite #32 | 52 032 KB |
| | Satellite #36 | 34 758 KB |
| | Satellite #31 | 29 997 KB |
| | Satellite #35 | 27 672 KB |
| **state variables** | **task** | **amount** |
| | PSR-Large #50 | 61 842 |
| | PSR-Large #48 | 48 210 |
| | PSR-Large #46 | 40 357 |
| | PSR-Large #49 | 36 790 |
| | PSR-Large #44 | 32 174 |
| **operators/axioms** | **task** | **amount** |
| | Satellite #33 | 989 250 |
| | Satellite #32 | 638 665 |
| | Satellite #36 | 428 109 |
| | Satellite #31 | 368 990 |
| | Satellite #35 | 342 193 |

**Fig. 9.12.** Statistics for the five largest planning tasks in each of the four categories