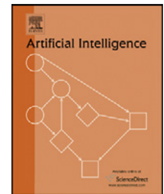
Artificial Intelligence 173 (2009) 503–535

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint



Concise finite-domain representations for PDDL planning tasks ☆

Malte Helmert

Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Georges-Köhler-Allee 052, 79110 Freiburg, Germany

ARTICLE INFO

Article history:
Received 16 November 2007
Received in revised form 22 October 2008
Accepted 29 October 2008
Available online 25 November 2008

Keywords:
Automated planning
Problem reformulation
PDDL
SAS^+

ABSTRACT

We introduce an efficient method for translating planning tasks specified in the standard PDDL formalism into a concise grounded representation that uses finite-domain state variables instead of the straight-forward propositional encoding.

Translation is performed in four stages. Firstly, we transform the input task into an equivalent normal form expressed in a restricted fragment of PDDL. Secondly, we synthesize invariants of the planning task that identify groups of mutually exclusive propositions which can be represented by a single finite-domain variable. Thirdly, we perform an efficient relaxed reachability analysis using logic programming techniques to obtain a grounded representation of the input. Finally, we combine the results of the third and fourth stage to generate the final grounded finite-domain representation.

The presented approach has originally been implemented as part of the Fast Downward planning system for the 4th International Planning Competition (IPC4). Since then, it has been used in a number of other contexts with considerable success, and the use of concise finite-domain representations has become a common feature of state-of-the-art planners.
© 2008 Elsevier B.V. All rights reserved.

1. Introduction

Consider the transportation planning task illustrated in Fig. 1. There are three cars, a train, and two parcels, located in two cities comprising several locations each. The cars may move along a network of roads within their respective city of origin, while the train moves along a single railway link that connects the two cities. Parcels may be loaded into any vehicle that is present at the same location, and parcels carried by a vehicle may be unloaded to the current location of that vehicle at any time. The objective is to move each parcel to a designated goal location.

1.1. PDDL representations

In order to find a plan for this example task using a general-purpose planning system, we must first represent it in a way that such a system can reason about. Since its inception in 1998 [38], the Planning Domain Definition Language (PDDL) has become the de-facto standard language for representing classical planning tasks. The original PDDL formalism, as used in the first two International Planning Competitions, was purely logic-based and can be considered a syntactic variant of the earlier ADL language [39] (excluding the support for functional fluents, which are present in ADL). Since then, the language has been extended to more easily express additional aspects of real-world planning tasks, such as numbers and durations [21], state variables whose values are derived from the values of other state variables [18], and most recently plan constraints and preferences [24].

☆ This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS). See www.avacs.org for more information.

E-mail address: helmert@informatik.uni-freiburg.de.

0004-3702/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2008.10.013

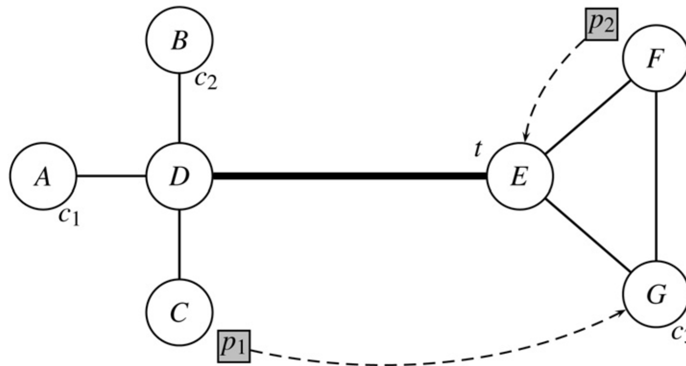


Fig. 1. A transportation planning task. Deliver parcel p_1 from C to G and parcel p_2 from F to E, using the cars c_1, c_2, c_3 and train t. The cars may only use roads (thin edges), the train may only use the railway (thick edge).

In PDDL, planning tasks are described in terms of *objects* of the world (cars, locations, parcels), *predicates* that describe static or dynamic relations that hold between these objects (whether or not two given locations are connected by a road, whether or not a given parcel is currently inside a given vehicle), *operators* that manipulate these relations (moving a car from one location to another, unloading a parcel), an *initial state* that describes the situation before plan execution, and a *goal specification* describing the objectives that solution plans must achieve.

While PDDL itself is a (restricted) first-order formalism, all state-of-the-art planning systems compile the input specification into a propositional representation at an early stage by *grounding* predicates, operators and goal specifications. Many planners go even further and transform the grounded task into a particularly simple syntactic form called *propositional STRIPS*, where states of the world can be represented as sets of (satisfied) atomic propositions and operators are represented in terms of which propositions must be true for the operator to be applicable (preconditions), which propositions the operator makes true (add effects), and which propositions it makes false (delete effects). The example task can be naturally modelled in propositional STRIPS; (part of) such a representation is shown in Fig. 2.

PDDL- or STRIPS-based representations of planning tasks have a number of desirable features. Due to the close relationship to first-order logic (for ungrounded PDDL) and propositional logic (for grounded PDDL), the semantics are easy to understand for researchers and practitioners with a background in formal logics. Moreover, representing all properties of a world state in terms of truth values has the appeal of simplicity. There is a certain mathematical elegance to the formalism, and it clearly achieves the language designers' maxim of describing planning tasks in terms of their "physics, not advice" [38].

1.2. Finite-domain representations

The absence of any form of "advice" from the PDDL representation is appropriate for a language designed for general problem solvers, but it comes at a price, to be paid by planning algorithms that have to reason about the represented task. In particular, the state space induced by a propositional representation such as the one shown in Fig. 2 is very unstructured. A priori, a proposition like `at-p1-a` (stating that the first parcel is at location A) bears no closer relationship to `at-p1-b` (stating that the first parcel is at location B) than to, say, `in-p2-t` (stating that the second parcel is currently inside the train). However, if we take into account their intended meaning, propositions that represent potential locations of the same parcel are clearly more closely related to each other than to ones that encode properties of the other parcel. In particular, only one of the propositions of the form `at-p1-`x can be true at the same time in any feasible world state. To the planner, there appear to be as many as $2^{35} \approx 3.4 \cdot 10^{10}$ feasible world states in the example task, corresponding to all valuations of the 35 propositional state variables, yet in truth the number of relevant states is only $11616 \approx 1.2 \cdot 10^4$, as all other valuations are not reachable from the given initial state.

An alternative representation of the example task is shown in Fig. 3. This representation uses general *finite-domain* variables, not just binary ones, to represent the state of the world. For example, a single variable `p1` with a domain of 11 values completely encodes the state of the first parcel, subsuming the information of all propositions `at-p1-`x and `in-p1-`y from the STRIPS encoding. Using this representation, the set of feasible world states coincides with the set of syntactically legal ones.

In this article, we present an efficient algorithm for translating planning tasks specified in PDDL 2.2 into a compact finite-domain representation. The algorithm has been implemented as part of the Fast Downward planner [29] and used by a number of other planning algorithms [3,27,31,48,49]. It extends an earlier algorithm by Edelkamp and Helmert [15] which also translates PDDL tasks to finite-domain representations, but is limited to a much smaller language fragment (STRIPS, no typing, no domain constants in operator definitions).

As far as we know, no other algorithms for this problem have been described in the literature, so the main contribution of this article is the first description of a method to generate concise finite-domain representations from arbitrary (non-

Propositions:
`at-p1-a`, `at-p1-b`, `at-p1-c`, `at-p1-d`, `at-p1-e`, `at-p1-f`, `at-p1-g`,
`at-p2-a`, `at-p2-b`, `at-p2-c`, `at-p2-d`, `at-p2-e`, `at-p2-f`, `at-p2-g`,
`at-c1-a`, `at-c1-b`, `at-c1-c`, `at-c1-d`,
`at-c2-a`, `at-c2-b`, `at-c2-c`, `at-c2-d`,
`at-c3-e`, `at-c3-f`, `at-c3-g`,
`at-t-d`, `at-t-e`,
`in-p1-c1`, `in-p1-c2`, `in-p1-c3`, `in-p1-t`,
`in-p2-c1`, `in-p2-c2`, `in-p2-c3`, `in-p2-t`
Init:
`at-p1-c`, `at-p2-f`, `at-c1-a`, `at-c2-b`, `at-c3-g`, `at-t-e`
Goal:
`at-p1-g`, `at-p2-e`
Operator `drive-c1-a-d`:
PRE: `at-c1-a` ADD: `at-c1-d` DEL: `at-c1-a`
Operator `drive-c1-b-d`:
PRE: `at-c1-b` ADD: `at-c1-d` DEL: `at-c1-b`
Operator `drive-c1-c-d`:
PRE: `at-c1-c` ADD: `at-c1-d` DEL: `at-c1-c`
...
Operator `load-c1-p1-a`:
PRE: `at-c1-a`, `at-p1-a` ADD: `in-p1-c1` DEL: `at-p1-a`
Operator `load-c1-p1-b`:
PRE: `at-c1-b`, `at-p1-b` ADD: `in-p1-c1` DEL: `at-p1-b`
Operator `load-c1-p1-c`:
PRE: `at-c1-c`, `at-p1-c` ADD: `in-p1-c1` DEL: `at-p1-c`
...
Operator `unload-c1-p1-a`:
PRE: `at-c1-a`, `in-p1-c1` ADD: `at-p1-a` DEL: `in-p1-c1`
Operator `unload-c1-p1-b`:
PRE: `at-c1-b`, `in-p1-c1` ADD: `at-p1-b` DEL: `in-p1-c1`
Operator `unload-c1-p1-c`:
PRE: `at-c1-c`, `in-p1-c1` ADD: `at-p1-c` DEL: `in-p1-c1`
...

Fig. 2. Propositional STRIPS representation of the transportation planning task.

numeric, non-temporal) PDDL tasks. From a high-level perspective, our approach follows very similar ideas to the algorithm of Edelkamp and Helmert, but the generalization beyond STRIPS requires significant extensions to the core components of the translation algorithm, *invariant synthesis* (Section 5) and *grounding* (Section 6). (Indeed, even though the emphasis in this article is on the overall goal of transforming PDDL tasks into a concise finite-domain representation, we believe that the invariant synthesis and grounding algorithms we present are also useful for planning algorithms that work on traditional PDDL representations, so the algorithms presented in Sections 5 and 6 may be seen as additional contributions of this paper.)

1.3. Why finite-domain representations?

Before diving into more technical matters, let us briefly discuss why compact finite-domain representations might be desirable. We already noted that in the STRIPS representation, unlike the finite-domain representation, there is a vastly larger number of syntactically valid states than feasible (reachable) states in the planning task. This is not necessarily problematic – for example, a planning algorithm based on forward search, such as Hoffmann and Nebel's FF [33], will never encounter any of the infeasible states, so there is no obvious advantage to the finite-domain representation. However, a number of other planning approaches do benefit significantly from the changed representation:

- Planning algorithms based on SAT-solving [35,36] can use SAT representations that disallow exploring partial valuations that assign inconsistent values to a single finite-domain variable. This is an example of the more general notion of *mutex constraints*, which is critical to the performance of SAT planners [43]. A naive SAT encoding would need to search the

```
Variables:
    p1, p2 ∈ {at-a, at-b, at-c, at-d, at-e, at-f, at-g,
              in-c1, in-c2, in-c3, in-t}
    c1, c2 ∈ {at-a, at-b, at-c, at-d}
    c3     ∈ {at-e, at-f, at-g}
    t      ∈ {at-d, at-e}
Init:
    p1 = at-c, p2 = at-f
    c1 = at-a, c2 = at-b, c3 = at-g, t = at-e
Goal:
    p1 = at-g, p2 = at-e
Operator drive-c1-a-d:
    PRE: c1 = at-a    EFF: c1 = at-d
Operator drive-c1-b-d:
    PRE: c1 = at-b    EFF: c1 = at-d
Operator drive-c1-c-d:
    PRE: c1 = at-c    EFF: c1 = at-d
...
Operator load-c1-p1-a:
    PRE: c1 = at-a, p1 = at-a    EFF: p1 = in-c1
Operator load-c1-p1-b:
    PRE: c1 = at-b, p1 = at-b    EFF: p1 = in-c1
Operator load-c1-p1-c:
    PRE: c1 = at-c, p1 = at-c    EFF: p1 = in-c1
...
Operator unload-c1-p1-a:
    PRE: c1 = at-a, p1 = in-c1    EFF: p1 = at-a
Operator unload-c1-p1-b:
    PRE: c1 = at-b, p1 = in-c1    EFF: p1 = at-b
Operator unload-c1-p1-c:
    PRE: c1 = at-c, p1 = in-c1    EFF: p1 = at-c
...
```

Fig. 3. Finite-domain representation of the transportation planning task.

full space of syntactically valid STRIPS states within each variable layer. In addition to the use of mutexes, the recently successful MaxPlan planner [7] also uses the finite-domain representation to derive so-called *londex constraints*, which are reported as the key innovation of the planner. Londex constraints need finite-domain representations to be effective: for binary state variables, they offer no additional pruning power over mutex constraints.

- Planners that perform a symbolic exploration of the state space with binary decision diagrams (BDDs) can use the finite-domain representation to reduce the number of variables required in the BDD encoding, compared to a naive encoding. Moreover, the finite-domain representation leads to a variable ordering where closely related propositions are grouped together, which is critical to good performance of BDD exploration [17].
- Heuristic planning approaches using pattern databases or other homomorphism abstractions [14,27,31] benefit from the more concise finite-domain encoding because larger subtasks can be solved and stored in memory and thus used as an abstraction heuristic. Moreover, as for BDDs, the finite-domain representation groups related propositions which should be considered together in abstractions.
- Planners based on constraint programming [11,47] can build more efficient CSP representations from finite-domain representations than from a direct encoding of the state variables in the PDDL representation. As a case in point, CPlan by van Beek and Chen [47] uses hand-tailored CSP encodings of standard planning domains. In most cases, its state representations are identical to the finite-domain representations generated by the algorithm presented in this article.
- Compilations to integer programming (IP) can use the finite-domain representation to get more concise IP representations. By modelling the value changes of a single finite-domain variable as a network flow problem, which is very naturally expressible as a linear or integer program, it is also possible to use a richer notion of “plan steps” than in traditional Graphplan-like encodings. This helps reduce the IP size and often translates to better performance [49].
- Planning approaches based on problem decomposition, such as the causal graph heuristic [28] used in the Fast Downward planner [29], benefit from the simpler causal structure of the finite-domain representation. To illustrate this,

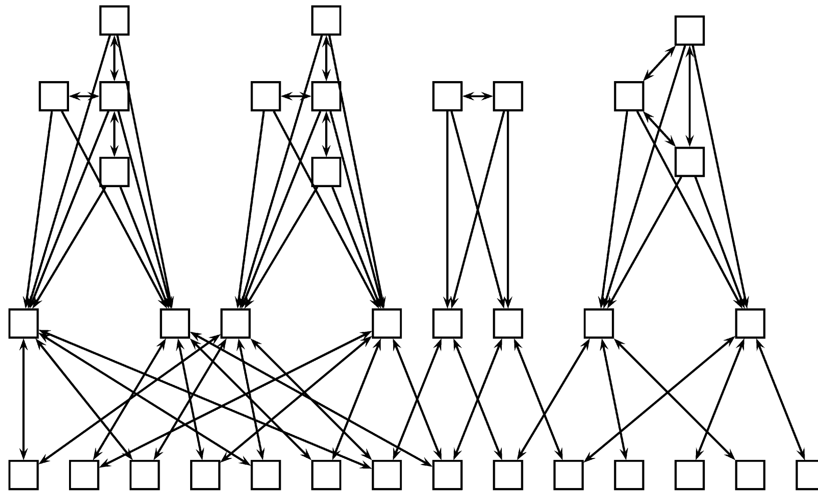


Fig. 4. Causal graph for the example task (STRIPS representation).

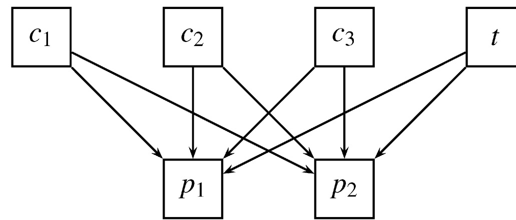


Fig. 5. Causal graph for the example task (finite-domain representation).

compare the causal graph of the STRIPS encoding of the example problem (Fig. 4) to the causal graph of its finite-domain counterpart (Fig. 5). Indeed, it has recently been shown that the causal graph heuristic degenerates to an inferior variant of the additive heuristic on binary representations [30].

This concludes our discussion of the potential advantages of concise finite-domain representations. In the next section, we formally introduce PDDL and finite-domain representations, before we begin describing the translation algorithm in Section 3.

2. Definitions

As remarked in the introduction, PDDL is the language in which planning tasks are most commonly expressed. In particular, the planning tasks of the international planning competitions (IPC) are expressed in PDDL, so a planning system must be able to deal with this language in order to participate. In this work, we consider the *non-numerical*, *non-temporal* fragment of PDDL 2.2, i.e., "level 1" of that language (where level 2 introduces numerical state variables and level 3 introduces temporal planning features). We do not consider the most recent additions to the language, namely the capabilities for expressing plan constraints and preferences in PDDL 3 [24]. However, these features are orthogonal to the issue of binary vs. finite-domain encoding, so that extending our work in this direction is conceptually easy.

Our definition of PDDL tasks uses common notations from first-order logic which we assume to be known; we refer to the literature [13] for formal definitions. Throughout the section, we assume that all logical formulae are over a first-order language $\mathcal{L}$ which consists of sufficiently many constant symbols (*objects* in PDDL terminology), relation symbols (*predicates*) and variable symbols. There are no function symbols, unless one considers constants to be 0-ary functions. We use the notation free(φ) to refer to the set of free variables of a first-order formula φ.

Definition 1 *(PDDL operators).* A **PDDL operator** is a pair $\langle \chi, e \rangle$, which consists of a (possibly open) first-order formula χ called its **precondition** and a **PDDL effect** e. PDDL effects are recursively defined by finite application of the following rules:

- A first-order literal l is a PDDL effect called a **simple effect**.
- If $e_1, \ldots, e_n$ are PDDL effects, then $e_1 \wedge \cdots \wedge e_n$ is a PDDL effect called a **conjunctive effect**.
- If χ is a first-order formula and e is a PDDL effect, then $\chi \rhd e$ is a PDDL effect called a **conditional effect**.

- If $v_1, \ldots, v_k$ are variable symbols and e is a PDDL effect, then $\forall v_1 \ldots v_k : e$ is a PDDL effect called a **universally quantified effect** or **universal effect**.

Free variables of simple effects are defined as for literals in first-order logic. Free variables of other effects are defined by structural induction:

- $\text{free}(e_1 \wedge \cdots \wedge e_n) = \text{free}(e_1) \cup \cdots \cup \text{free}(e_n)$,
- $\text{free}(\chi \rhd e) = \text{free}(\chi) \cup \text{free}(e)$,
- $\text{free}(\forall v_1 \ldots v_k : e) = \text{free}(e) \setminus \{v_1, \ldots, v_k\}$.

The set of free variables of a PDDL operator is defined as $\text{free}(\langle \chi, e \rangle) = \text{free}(\chi) \cup \text{free}(e)$. Free variables are also called **parameters** of the operator.

PDDL operators define the ways in which a planning algorithm can move from one world state to another. If the current state satisfies the precondition of an operator, then the operator may be applied, leading to a new state which is like the old one except that it is modified in certain ways specified by the effect of the operator. An operator with parameters cannot be applied directly; it must first be *grounded* by substituting concrete objects for the parameters.

Definition 2 *(PDDL axioms).* A **PDDL axiom** is a pair $\langle \varphi, \psi \rangle$ such that φ is a first-order atom and ψ is a first-order formula with $\text{free}(\psi) \subseteq \text{free}(\varphi)$. We write the axiom $\langle \varphi, \psi \rangle$ as $\varphi \leftarrow \psi$ and call φ the **head** and ψ the **body** of the axiom.

A set $\mathcal{A}$ of PDDL axioms is called **stratifiable** iff there exists a total preorder $\preceq$ on the predicate symbols of $\mathcal{A}$ such that for each axiom where predicate Q occurs in the head, we have $P \preceq Q$ for all predicates P occurring in the body, and $P \prec Q$ for all predicates P occurring in a negative literal in the translation of the body to negation normal form.

Axioms provide a way of defining certain predicates based on other, "more basic" predicates. For example, given an `ontop` predicate, we can define its transitive closure `above` with the two axioms $\texttt{above}(x, y) \leftarrow \texttt{ontop}(x, y)$ and $\texttt{above}(x, z) \leftarrow \exists y(\texttt{ontop}(x, y) \wedge \texttt{above}(y, z))$.

Stratifiability of a set of axioms is necessary for ensuring that the outcome of axiom evaluation is well-defined. Without such a condition, it would be possible to specify rules of the form "$P(x)$ is true whenever $P(x)$ is false." Intuitively, $P \prec Q$ means that the truth value of atoms over P must be determined before the truth value of atoms over Q.

Definition 3 *(PDDL tasks).* A **PDDL task** is given by a 4-tuple $\Pi = \langle \chi_0, \chi_\star, \mathcal{A}, \mathcal{O} \rangle$ with the following components:

- χ_0 is a finite set of ground atoms called the **initial state**.
- $\chi_\star$ is a closed formula called the **goal formula**.
- $\mathcal{A}$ is a finite stratified set of PDDL axioms.
- $\mathcal{O}$ is a finite set of PDDL operators.

Predicates occurring in the head of an axiom in $\mathcal{A}$ are called **derived predicates**. Predicates occurring in the initial state or in simple effects of operators in $\mathcal{O}$ are called **fluent predicates**. The sets of derived and fluent predicates are required to be disjoint.

We assume that the reader is already familiar with PDDL semantics and point to the language definition [18,21] for more information. Apart from syntactic differences, there are three aspects of non-numerical, non-temporal PDDL 2.2 not captured by our definition:

- There are no operator names. Our translation algorithm maps each grounded PDDL operator to a unique finite-domain representation operator, so that an implementation need only propagate operator names, and any plans generated for the translated task need not undergo any form of post-processing to apply to the original task.
- 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 translation algorithm works, there is no need for such a distinction.
- There are no types. Our translation algorithm compiles away types into unary predicates in the very first processing step (see Section 4.1), so we can assume untyped representations for all following stages.

With PDDL as a starting point, let us now introduce the kinds of planning tasks that the translation algorithm generates, which we call FDR (finite-domain representation) tasks. FDR tasks are based on the SAS$^+$ planning formalism [2,34], extended with axioms and conditional effects.

The definition exhibits a number of similarities, but also a few differences between PDDL tasks and our planning model. FDR tasks only allow simple conjunctions in goals, axioms and operators, and conditional effects cannot be nested. Moreover,

PDDL tasks use first-order concepts such as schematic operators whose variables can be instantiated in many different ways, while FDR tasks are grounded. These two differences to PDDL, in particular the use of a grounded representation, are due to a desire to keep the FDR formalism simple, to reduce the burden for planners that use it. Indeed, all the planning approaches for finite-domain representations listed in the previous section have been introduced for (and, in most cases, require) grounded representations. (We believe, however, that many of the translation ideas introduced in this article can be adapted to schematic finite-domain representations where such representations appear more desirable.)

Definition 4 *(Planning tasks in finite-domain representation (FDR tasks)).* A **planning task in finite-domain representation (FDR task)** 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**, where each variable $v \in \mathcal{V}$ has 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** 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 variable assignment is called a **state** if it is defined for all fluents and none of the derived variables in $\mathcal{V}$. It is called an **extended state** if it is defined for all variables 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 (FDR) **axioms** over $\mathcal{V}$. Axioms are triples $\langle cond, v, d \rangle$, 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.
  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 must appear with a unique value in all 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 (FDR) **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 commonly write $cond \to v := d$ in place of $\langle cond, v, d \rangle$.

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

Definition 5 *(Extended states defined by a state).* Let s be a state of an FDR task Π 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($\mathcal{A}_1, \dots, \mathcal{A}_k$, 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 $i \in \{1, \dots, k\}$:
    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 4 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 FDR task.

Definition 6 *(FDR state spaces).* The **state space** of an FDR task $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$, 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 $(cond \to v := d) \in eff$ with $cond \subseteq \mathcal{A}(s)$, and
- $s'(v) = s(v)$ for all fluents v where no such effect exists.

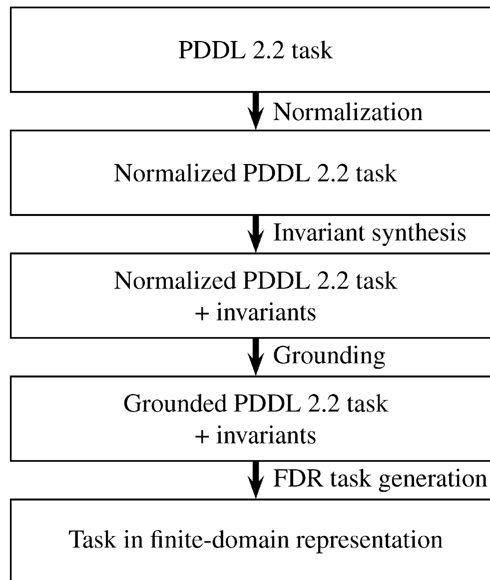


Fig. 6. Overview of the translation algorithm.

Finally, we can define the FDR planning problem.

Definition 7 *(FDR planning).* FDR-PLAN is the following search problem: Given an FDR task Π with initial state s_0, goal $s_\star$ and axioms $\mathcal{A}$, compute a path in $\mathcal{S}(\Pi)$ from s_0 to some state s' with $s_\star \subseteq \mathcal{A}(s')$, or prove that none exists.

FDR-PLANEX is the following decision problem: Given an FDR task Π with initial state s_0, goal $s_\star$ and axioms $\mathcal{A}$, does $\mathcal{S}(\Pi)$ contain a path from s_0 to some state s' with $s_\star \subseteq \mathcal{A}(s')$?

The FDR-PLANEX problem is easily shown to be PSPACE-hard because it generalizes the plan existence problem for propositional STRIPS, which is known to be PSPACE-complete [6]. It is also easy to see that the addition of non-binary domains, axioms and conditional effects does not increase the theoretical complexity of FDR planning beyond propositional STRIPS. Thus, we conclude our formal introduction of FDR planning by stating that FDR-PLANEX is PSPACE-complete. In the following section, we turn to the problem of generating concise finite-domain representations from PDDL representations.

3. Translation overview

Translation is performed in four stages. Starting from a PDDL specification, we first apply some well-known logical equivalences to compile away types and simplify conditions and effects in the *normalization* stage (Section 4). Next, the *invariant synthesis* stage computes mutual exclusion relations between atoms, which are later used for synthesizing the FDR variables (Section 5). The *grounding* stage 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 (Section 6). Invariant synthesis and grounding are not related to one another and could just as well be performed in the opposite order. Finally, the *FDR task generation* stage chooses the final set of state variables by using the information from invariants and grounding and produces the FDR output (Section 7).

The complete translation process is outlined in Fig. 6. Before we begin the detailed discussion of these stages in the following sections, we should point out that of these four stages, only three are necessary to convert a PDDL task to an FDR task: the invariant synthesis stage 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 FDR task; in particular, all state variables in the generated FDR task would be binary. Therefore, invariants are important for obtaining a *concise* finite-domain representation.

4. Normalization

The normalization stage has three responsibilities: compiling away types, simplifying conditions, and simplifying effects. Its result is a *normalized PDDL 2.2 task*, which is a PDDL task with a number of strong syntactical restrictions.

Definition 8 *(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.

4.1. Compiling away types

As suggested earlier, types are compiled away as the very first processing step. 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 PDDL 2.2 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)`, plus an additional atom `(supertype someobj)` for each supertype of `sometype`, including the universal supertype `object`.

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

Typed quantifiers in conditions are compiled away with the usual logic idioms, turning `(exists (?v - type)` φ`)` into `(exists (?v) (and (type ?v)` φ`))` and `(forall (?v - type)` φ`)` into `(forall (?v) (imply (type ?v)` φ`))`.

Finally, typed universal effects are compiled into universal conditional effects, so `(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 3. We will thus use the more concise logical notation from that definition in the following, rather than lengthy PDDL syntax. For example, we write $\varphi \vee \psi$ instead of `(or` φ ψ`)` and $\varphi \rhd e$ instead of `(when` φ e`)`.

4.2. 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 $\varphi \to \psi \equiv \neg\varphi \vee \psi$ 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 counterproductive 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 $\mathrm{free}(\exists x\neg\varphi) = \{v_1, \dots, v_k\}$, we introduce a new derived predicate `new-pred` of arity k, defined by the axiom `new-pred`$(v_1, \dots, v_k) \leftarrow \psi$, where ψ is the translation of $\exists x\neg\varphi$ to negation normal form. We can then replace the original condition $\forall x\varphi$ by $\neg$`new-pred`$(v_1, \dots, v_k)$. 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 be compiled away easily, so we deal with them separately in a later stage.

If after elimination of universal quantifiers the goal condition is *not* a simple conjunction (i.e., if it contains disjunctions or existential quantifiers), 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 parameterless derived predicate `goal-pred` and a new axiom `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, so that axioms are not necessary. (For the intended applications of finite-domain representations mentioned in Section 1.3, we believe that of two otherwise identical representations, the one that uses fewer state

variables is usually preferable, so we attempt to avoid introducing new state variables unless there is a compelling reason to do so.)

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 φ and one with body ψ. If the disjunction is part of an operator precondition, we replace the operator by two copies of the original, one with precondition φ and one with precondition ψ. 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 $x \notin \text{free}(\psi)$, 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 can eliminate them. For axioms, we simply drop them, following the logic programming 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 $\forall x : (\varphi \rhd e)$.

4.3. 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; for example, a universal effect may, but need not, contain a conditional effect. 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 conjunctive effect, where each conjunct is 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.

This concludes the normalization stage. For the sake of the following discussion, we briefly recapitulate the structural restrictions for normalized PDDL tasks (Definition 8):

- The representation is untyped.
- All formulas (goal, preconditions, effect conditions, axiom bodies) are conjunctions of literals.
- The effect of each operator is a conjunctive effect whose parts are of the form $\forall v_1 \dots v_k : \varphi \rhd e$, where e is a simple effect.

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 $\forall v_1 \dots v_k : \varphi \rhd e$, we will refer to $\{v_1, \dots, v_k\}$ as the set of *bound variables* of e and to φ as the *condition* of e. If e is a positive literal, we will call it an *add effect*, otherwise a *delete effect*.

5. Invariant synthesis

An invariant of a planning task is a property 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 for most propositional STRIPS planning tasks, but does not seem to entail a useful (i.e., exploitable) piece of information for a planner. Other invariants would be useful to know but are too difficult to verify. For example, "The goal is not satisfied" is an invariant iff the planning task is not solvable, so confirming the invariance of that state property is PSPACE-hard for a propositional STRIPS task.

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 [20,25,41,42]. Section 5.5 discusses related work on invariants, and why we introduce a new invariant synthesis algorithm in the following instead of applying one of the algorithms from the literature. The short answer is that most algorithms from the literature are limited to STRIPS domains. Moreover, some of them are prohibitively expensive for the largest planning tasks in the IPC benchmark suite.

For the purposes of translating planning tasks to a finite-domain representation, *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 (or that none of them 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 its 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* part of the PDDL input at all, solely exploiting information present in the *domain* part. This is a valuable feature, but it rules out the possibility of directly proving mutex conditions, because a mutex cannot be established without checking the initial state. Instead, we consider a slight generalization of mutexes.

Definition 9 *(Monotonicity invariant candidates).* A **monotonicity invariant candidate** for a PDDL task Π is given by a pair $\mathcal{I} = \langle V, \Phi\rangle$, where V is a set of first-order variables called the **parameters** of the candidate, and Φ is a set of atoms. Variables occurring freely in Φ 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 $\mathcal{I}$ 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*, as 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. This is somewhat involved, so we provide an example from the Logistics domain first. Consider the 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 Logistics. Let us now formalize what it means for a candidate to be an invariant.

Definition 10 *(Monotonicity invariants).* Let $\mathcal{I} = \langle V, \Phi\rangle$ be a monotonicity invariant candidate of a PDDL task Π.

An **instance** of $\mathcal{I}$ is a function α mapping the variables in V to objects of Π.

The set of **covered facts** of an instance α of $\mathcal{I}$ is the set of all ground atoms of the planning task Π which unify with some $\varphi \in \Phi$ under α, i.e., the set of all ground atoms φ_0 of Π 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 α of $\mathcal{I}$ in a state s is the number of covered facts of α which are true in s.

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

Similar to our convention for invariant candidates, we usually refer to monotonicity invariants simply as *invariants*.

The definition is probably best understood by considering the previously discussed example invariant. To get an instance of the candidate $\forall p\, \mathtt{at}(p,l) + \mathtt{in}(p,v) \downarrow$, we must map p to a particular object, say by $\alpha = \{p \mapsto \mathtt{package1}\}$. The set of covered facts of α in a given state s then consists of all atoms of the form $\mathtt{at}(\mathtt{package1}, l)$ or $\mathtt{in}(\mathtt{package1}, v)$ that are satisfied in s. In a reachable state, there will typically only be one such atom, for example given by the mapping $\beta = \alpha \cup \{l \mapsto \mathtt{location1}\}$, so the weight of α in s will be 1. But even if we consider strange states where α has a greater weight than 1, it is easy to see that the weight of α in a successor state of s is never greater than the weight of α in s. This is true for all instances of the candidate, so it is indeed an invariant.

As hinted before, monotonicity invariants are useful for grouping a number of related propositions into a single finite-domain 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 there are too many feasible candidates to enumerate exhaustively, we follow a guided *guess, check and repair* approach. Starting from a set of a few simple initial candidates, we try to prove that a given candidate is indeed an invariant. 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 a search problem, and indeed we solve it using standard breadth-first search, using a closed list to avoid exploring the same

invariant candidate twice. (This guarantees termination of the algorithm.) To fully specify the invariant synthesis algorithm, it thus suffices to discuss its search space:

- *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?

In the following, we deal with these three questions in sequence.

5.1. Initial candidates

Before starting the actual invariant synthesis, 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 more global reasoning than the proof methods we use for fluent predicates. 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, mutexes based on 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:

$$\forall x \; \mathtt{at}(x, l) \downarrow \tag{1}$$

$$\forall l \; \mathtt{at}(x, l) \downarrow \tag{2}$$

$$\forall x, l \; \mathtt{at}(x, l) \downarrow \tag{3}$$

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 (2) and (3) are obviously not invariants. Candidate (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 (1) into an invariant.

5.2. Proving invariance

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 11 *(Threatened invariant candidates).* An invariant candidate $\mathcal{I}$ 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 $\mathcal{I}$ 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 $\mathcal{I}$.
- When ignoring delete effects, the operator can increase the weight of some instance of $\mathcal{I}$ in some state by at least 2. In this case, we say that the operator is **too heavy** for $\mathcal{I}$.

Clearly, not being threatened by any schematic operator is a sufficient condition for being a monotonicity invariant. Note that just showing that no operator is unbalanced in the sense of the definition is not sufficient for invariance, as

the balance test considers different add effects in isolation. For example, an operator might have two add effects, each of which is individually balanced by a delete effect. However, the operator might still cause a net increase of the weight of an invariant instance if the two balancing delete effects can be identical. This is not always obvious; for example, consider an (incorrectly modelled) operator for moving something which is currently at two locations l_1 and l_2 to two other locations l_3 and l_4:

$$\begin{aligned} \text{Precondition:} &\quad \texttt{at}(x, l_1) \wedge \texttt{at}(x, l_2) \\ \text{Add effects:} &\quad \texttt{at}(x, l_3) \wedge \texttt{at}(x, l_4) \\ \text{Delete effects:} &\quad \texttt{at}(x, l_1) \wedge \texttt{at}(x, l_2) \end{aligned}$$

At first glance, the operator does not seem to be problematic for the monotonicity invariant candidate $\forall x\, \texttt{at}(x, l) \downarrow$, but actually it is: in the case $l_1 = l_2$ and $l_3 \neq l_4$, it increases the number of locations that x is currently at. Attempting to capture such subtleties in the balance test makes this test more complicated, and even more so in the presence of universally quantified effects, which can add an arbitrary number of facts. We avoid such complications by adding the heaviness test, which would reject the invariant candidate because the operator can increase the weight of its instances by two.

Clearly, the heaviness test is stricter than necessary. If the above operator were extended with the precondition $l_1 \neq l_2$, then $\forall x\, \texttt{at}(x, l) \downarrow$ might indeed be an invariant, but it would still be rejected due to the operator being too heavy for it. In the context of translation to finite domain representation, this does not appear to be problematic because we are only interested in invariants for the purpose of generating mutex groups. In that setting, the weight of interesting invariant instances is at most one, so operators that add more than one fact either do not exist or are never applicable, and typical PDDL models do not contain schematic operators which are never applicable.

Definition 11 gives rise to the algorithm shown in Fig. 7. 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 discuss the algorithm in full detail, instead focusing on two points that require some explanation, namely the satisfiability and entailment tests that occur towards the end of functions *is-operator-too-heavy* and *is-add-effect-unbalanced*.

For the heaviness test, two add effects can only lead to an operator being too heavy in states s where the operator is actually applicable (o'.precond is true in s), the triggering conditions of both add effects are satisfied (e.cond and e'.cond are true in s) and the add effects actually add propositions that were not true previously (e.atom and e'.atom are false in s). An operator is considered too heavy by *is-operator-too-heavy* if all these conditions can hold together, i.e., their conjunction is satisfiable. (Even if the conjunction is satisfiable, it is of course possible that none of the satisfying states is reachable from the initial state, so this may be overly conservative.)

For the imbalance test, an add effect leads to an imbalance by default. However, it can be balanced if whenever the operator is actually applied in a state s (which requires that o'.precond is true in state s) and the add effect triggers (e.cond is true in s) and actually adds something (e.atom is false in s), then something is deleted at the same time, which means that the delete effect triggers (e'.cond is true in s) and deletes something that was previously true (e'.atom is true in s). Function *is-add-effect-unbalanced* conducts logical entailment tests to check if a balancing delete effect is guaranteed to exist. (Again, this may be overly conservative because all states in which balance is violated might be unreachable.)

Coming back to the earlier LOGISTICS example, all three initial candidates are threatened by the same operator `unload-truck`:

$$\begin{aligned} \text{Precondition:} &\quad \texttt{package}(x) \wedge \texttt{truck}(t) \wedge \texttt{location}(l) \wedge \texttt{at}(t, l) \wedge \texttt{in}(x, t) \\ \text{Add effects:} &\quad \texttt{at}(x, l) \\ \text{Delete effects:} &\quad \texttt{in}(x, t) \end{aligned}$$

The operator is unbalanced with regard to all invariant candidates due to the add effect $\texttt{at}(x, l)$. Thus, as indicated before, none of (1)–(3) is an invariant. We will discuss possible refinements of the candidates shortly.

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

- We duplicate universal effects at the beginning of *is-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. 7 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

```
function prove-invariant(V, Φ):
    for each schematic operator o:
        if is-operator-too-heavy(o, V, Φ):
            return false. { Reject the candidate. }
        for each add effect e of o that affects a predicate in Φ:
            if is-add-effect-unbalanced(o, e, V, Φ):
                return false. { Reject or refine the candidate. }
    return true. { Accept the candidate. }

function is-operator-too-heavy(o, V, Φ):
    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′.
    for each pair (e, e′) of add effects of o′ that affect a predicate in Φ:
        if the parameters of operator o′ can be renamed so that
           (e.atom ≠ e′.atom and
            covers(V, Φ, e.atom) and covers(V, Φ, e′.atom) and
            o′.precond ∧ e.cond ∧ e′.cond ∧ ¬e.atom ∧ ¬e′.atom
            is satisfiable):
            return true. { The operator is too heavy. }
    return false. { The operator is not too heavy. }

function is-add-effect-unbalanced(o, e, V, Φ):
    Let o′ be a copy of o where the parameters are minimally
      renamed so that covers(V, Φ, e.atom) is true.
    for each delete effect e′ of o′ that affects a predicate in Φ:
        if the quantified variables of e′ can be renamed so that
           (e.atom ≠ e′.atom and covers(V, Φ, e′.atom) and
            o′.precond ∧ e.cond ∧ ¬e.atom ⊨ e′.cond ∧ e′.atom):
            return false. { e′ balances e. }
    return true. { The add effect is unbalanced. }

function covers(V, Φ, ψ):
    for each φ ∈ Φ:
        if the counted variables in φ (those not in V)
           can be renamed so that φ = ψ:
            return true.
    return false.
```

Fig. 7. Algorithm for proving that an invariant candidate $\langle V, \Phi \rangle$ is an invariant.

function *is-add-effect-unbalanced*. In our experience, invariants of this type are not very useful, but our implementation does support them.

- 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 in the sense of Definition 11, except if the axiom body already entails the head, which is not a very interesting case.
- Because the operator preconditions and effect conditions are in normal form (conjunctions of literals), the satisfiability test in function *is-operator-too-heavy* is performed on a conjunction of literals, which is possible in linear time. (Just check if any two literals in the conjunction are complementary.) Similarly, function *is-add-effect-unbalanced* tests entailment between two conjunctions of literals, which is also possible in linear time.

One final subtlety concerns the semantics of PDDL operators with "conflicting" effects. Note that our balance test requires that e.atom does not equal e'.atom, i.e., the atom that is added is different from the one which is deleted. The reason for this is that PDDL semantics mandate that if the same atom is both added and deleted simultaneously, it is actually added, so that an atom cannot balance itself.

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

Fig. 8. Algorithm for refining an unbalanced invariant candidate $\langle V, \Phi \rangle$.

5.3. *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 basic refinement algorithm is shown in Fig. 8. The actual implementation does not generate all possible refining atoms φ' naively, but rather uses information from the set of delete effects of the threatening operator o and the failed call to *is-add-effect-unbalanced* to only consider atoms φ' 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 invariant candidate (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 φ' 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.

In contrast, the other two candidates cannot be suitably refined. For (3), consider the `drive-truck` operator:

Precondition:	$\mathtt{truck}(t) \wedge \mathtt{location}(l) \wedge \mathtt{location}(l') \wedge \mathtt{city}(c) \wedge$ $\mathtt{in\text{-}city}(l, c) \wedge \mathtt{in\text{-}city}(l', c) \wedge \mathtt{at}(t, l')$
Add effects:	$\mathtt{at}(t, l)$
Delete effects:	$\mathtt{at}(t, l')$

In order to refine (3), $\forall x, l\, \mathtt{at}(x, l) \downarrow$, to balance this operator, we would need to add the atom $\mathtt{at}(x, l')$, as $\mathtt{at}(t, l')$ is the only delete effect of the operator and t unifies to x. 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 (1) and hence not considered again.

Considering candidate (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"), but this has more than one counted variable and thus will not be considered by *refine-candidate*. Supposing that we removed the restriction to candidates with at most one counted variable, $\forall \cdot\, \mathtt{at}(x, l') \downarrow$ would turn out to be violated by the `unload-truck` operator, but could 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 a monotonicity invariant. However, its only instance clearly has a weight greater than 1 in the initial state of any non-trivial Logistics task and thus is not useful for providing any mutex information. (Of course, it would still be a monotonicity invariant, and be part of the output of the algorithm, if the restriction to at most one counted variable were removed. To derive mutex information from monotonicity invariants, we must also consider the initial state information; this happens in the "Variable selection" stage of the translation algorithm, described in Section 7.1.)

LOGISTICS	$\forall x$ at(x,l) + in(x,t) $\downarrow$
BLOCKSWORLD	$\forall\cdot$ handempty() + holding(b) $\downarrow$
	$\forall b$ holding(b) + clear(b) + on(b',b) $\downarrow$
	$\forall b$ holding(b) + ontable(b) + on(b,b') $\downarrow$
GRID	$\forall\cdot$ armempty() + holding(k) $\downarrow$
	$\forall\cdot$ at-robot(l) $\downarrow$
	$\forall\cdot$ open(d) + locked(d) $\downarrow$
	$\forall\cdot$ locked(d) $\downarrow$
	$\forall d$ open(d) + locked(d) $\downarrow$
	$\forall d$ locked(d) $\downarrow$
	$\forall k$ holding(k) + at(k,l) $\downarrow$

Fig. 9. Invariants found in some standard benchmark domains.

5.4. Examples

This concludes our description of the invariant synthesis algorithm. To give an impression of the kind of invariants it generates, Fig. 9 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* covered by mutexes: 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.

5.5. Related work

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

- Edelkamp and Helmert's algorithm [15] proposed for the MIPS planner [16,17],
- Scholz's algorithm for finding *c-constraints* [44],
- Gerevini and Schubert's DISCOPLAN [25,26],
- Rintanen's invariant synthesis algorithm [42],
- Bonet and Geffner's algorithm for generating mutexes [5], and
- Fox and Long's TIM [9,20].

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, our algorithm can be considered an extension of the MIPS algorithm to non-STRIPS domains. Compared to the original algorithm, our method incorporates some cosmetic and performance improvements, but the main difference is the coverage of universal and conditional effects. Note, however, that this is no small difference, as it is much easier to reason about STRIPS operators than about the more general class of operators occurring in normalized PDDL tasks. On STRIPS domains, both algorithms generate the same set of invariants.

Scholz's algorithm is very similar to Edelkamp and Helmert's, with only slight differences in the way that failed invariant candidates are refined to generate new invariant candidates. It shares the weakness of being limited to STRIPS domains.

DISCOPLAN also uses a very similar guess, check and repair approach. However, its method for refining invariant candidates is quite different. In particular, while our algorithm immediately refines an invariant as soon as a threatening operator is discovered, DISCOPLAN first collects all threats to an invariant for *all* operators. Only then does it generate 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 our algorithm always terminates in less than a second on all IPC benchmark tasks, DISCOPLAN exceeds a 30 minute timeout on large instances of the IPC4 AIRPORT domain. It should be noted, though, that DISCOPLAN generates many classes of invariants besides mutexes, because it was designed as a general invariant synthesis tool, not with finite-domain representations in mind. This difference in purpose should be taken into account when comparing runtime results, as it is likely that better runtime results could be obtained for DISCOPLAN by only considering mutual exclusion invariants. (Modifying DISCOPLAN's algorithm to derive such a specialization appears feasible, but an appropriate modification of the available implementation appears to be a practically non-trivial task.) Apart from efficiency concerns, another consideration is that even though DISCOPLAN is not limited to STRIPS, it can only deal with a subset of ADL features which is not sufficiently rich for all IPC benchmarks. Finally, even for STRIPS domains, we found that some invariants important for a concise finite-domain encoding which our algorithm discovers were missed by DISCOPLAN. 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. (After discussing this point with the DISCOPLAN authors, the algorithm has been amended, so that the most recent version of DISCOPLAN now finds this invariant.)

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, opposite to 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 propositional 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 than mutexes.

Bonet and Geffner's algorithm for generating mutexes can be seen as a special case of Rintanen's algorithm that starts from a different (weaker) set of invariant candidates and immediately rejects all failed candidates instead of trying to refine them. Like Rintanen's algorithm, it is limited to STRIPS and works on a grounded representation, which makes it much more expensive to compute for large IPC benchmarks than our first-order algorithm. To keep the runtime manageable, the algorithm puts some severe restrictions on the potential mutex pairs to consider. For example, if `at`(x) is a set of propositions that encodes the location of an object on a graph, the algorithm fails to prove mutual exclusion if the diameter of the graph is greater than 2. (Examples of this arise in the AIRPORT, DRIVERLOG, GRID, MPRIME, MYSTERY and TPP domains.)

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, based on the notion of *property spaces* which are generated from the type structure of the task, which is in turn derived by a *type inference* technique which gives the system its name. TIM was originally [20] limited to STRIPS and thus not directly usable for us. It has since been extended to handle some ADL constructs [9], independently of the development of our invariant synthesis algorithm.

6. Grounding

After computing monotonicity invariants, the next translation stage generates a variable-free representation of the normalized PDDL task, a process which is called *grounding*.

Definition 12 *(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 (i.e., do not contain variables).

Grounding is a conceptually simple operation. If O is the set of objects of the task, a variable x in a parameterized structure (operator, axiom or universally quantified effect) can be eliminated by replacing the original structure with $|O|$ copies, one for each object $o \in O$, where x is substituted with o in the respective copy. If the PDDL task were not already in normal form, quantifiers in conditions could be similarly eliminated by replacing $\exists x\varphi$ with the disjunction $\bigvee_{o\in O} \varphi[x/o]$ and $\forall x\varphi$ with the conjunction $\bigwedge_{o\in O} \varphi[x/o]$.

In general, the grounded task can be exponentially larger than the original one: for example, an operator with k parameters gives rise to $|O|^k$ many ground instances, and k can grow linearly with the task size. However, in practice the number of parameters k is usually low, and in particular it is fixed for a given planning domain. Moreover, the exponential blowup through grounding is computationally unavoidable, since planning with grounded representations is exponentially easier than planning with schematic representations [19].

In practice, for the majority of common planning domains, grounding is not a time-critical operation, and simple grounding schemes like the one outlined above suffice. However, there are exceptions to this rule: the naive grounding algorithm is not computationally feasible for the whole spectrum of IPC planning domains. To illustrate that grounding can be challenging, we tested the grounding algorithm implemented in the FF planner [33] on the IPC benchmark suite, imposing a runtime limit of 30 minutes and a memory limit of 2 GB. (FF does not directly support derived predicates, but for the purposes of grounding, derived predicates can be treated as if they were operators with a single effect, which we did for this experiment.) FF is particularly suited for comparison because it is one of the few planners that support the full ADL subset of the PDDL language, and because its grounded tasks follow a very similar normal form to the one in this article. (The only significant difference is that it compiles away negative literals in conditions.) Moreover, it uses a fairly sophisticated grounding procedure, originally introduced by Koehler and Hoffmann for the IPP planner [37].

Our experiment showed that even though grounding in FF is blazingly fast for the vast majority of benchmarks, there are scaling issues in some domains. In particular, the grounding procedure failed on 57 tasks (28 from the OPTTELEGRAPH domain, 2 from PATHWAYS, and 27 from PSR-LARGE) by exhausting the memory limit. Being unable to ground a planning task would not be a significant problem if these tasks were beyond reach of current planners, but most of them are not. For

Operator `load-truck`(p,t,l):
Precond.: `package`$(p) \wedge$ `truck`$(t) \wedge$ `location`$(l) \wedge$ `at`$(t,l) \wedge$ `at`(p,l)
Effect: `in`$(p,t) \wedge \neg$`at`(p,l)

Operator `unload-truck`(p,t,l):
Precond.: `package`$(p) \wedge$ `truck`$(t) \wedge$ `location`$(l) \wedge$ `at`$(t,l) \wedge$ `in`(p,t)
Effect: `at`$(p,l) \wedge \neg$`in`(p,t)

Operator `drive-truck`(t,l,l',c):
Precond.: `truck`$(t) \wedge$ `location`$(l) \wedge$ `location`$(l') \wedge$ `city`$(c) \wedge$
`in-city`$(l,c) \wedge$ `in-city`$(l',c) \wedge$ `at`(t,l)
Effect: `at`$(t,l') \wedge \neg$`at`(t,l)

Operator `load-airplane`(p,a,l):
Precond.: `package`$(p) \wedge$ `airplane`$(a) \wedge$ `location`$(l) \wedge$ `at`$(a,l) \wedge$ `at`(p,l)
Effect: `in`$(p,a) \wedge \neg$`at`(p,l)

Operator `unload-airplane`(p,a,l):
Precond.: `package`$(p) \wedge$ `airplane`$(a) \wedge$ `location`$(l) \wedge$ `at`$(a,l) \wedge$ `in`(p,a)
Effect: `at`$(p,l) \wedge \neg$`in`(p,a)

Operator `fly-airplane`(a,l,l'):
Precond.: `airplane`$(a) \wedge$ `airport`$(l) \wedge$ `airport`$(l') \wedge$ `at`(a,l)
Effect: `at`$(a,l') \wedge \neg$`at`(a,l)

Fig. 10. Schematic operators of the Logistics domain.

algorithm	**operators**
naive	$5.82 \cdot 10^{10}$
checking types	$1.94 \cdot 10^{8}$
checking static preconditions	$3.00 \cdot 10^{6}$
checking relaxed reachability	$1.53 \cdot 10^{5}$
checking relaxed reachability and pruning no-ops	$1.51 \cdot 10^{5}$

Fig. 11. Number of ground operator generated by different grounding algorithms for task Logistics #28 (IPC1).

example, the results reported by Richter et al. [40] show that their landmark-based planner can solve (in at least one of the planner configurations described in the paper) all these tasks except for 16 of the largest PSR-Large instances under similar time and memory constraints. Furthermore, even significantly increasing the memory limit for the grounding procedure does not eliminate the bottleneck in the grounding procedure: with a 28 GB memory limit, grounding with FF still fails on 26 tasks (11 from OptTelegraph, 15 from PSR-Large), of which 16 are solved by the planner of Richter et al. with a 3 GB memory bound.

6.1. Improving the naive grounding algorithm

How can we perform grounding more efficiently than the naive algorithm that instantiates each variable with each possible object? The key observation here is that many ground operators produced by the naive algorithm are not applicable in any reachable state of the task, and thus can be safely omitted from the grounded representation.

We illustrate this point with the operators of the Logistics domain, shown in Fig. 10, using instance #28 from IPC1 [38] as a running example. There are 490 objects in this task, so that the naive algorithm generates 490^k ground instances of each operator with k parameters. There are 5 operators with 3 parameters each and one operator with 4 parameters, so we end up with $5 \cdot 490^3 + 490^4 = 5.82 \cdot 10^{10}$ operators (cf. first row of Fig. 11), which is clearly infeasible. We will now discuss a number of increasingly sophisticated techniques to reduce this number, leading up to the ideas underlying the new grounding method introduced in this article, which is then discussed in the remainder of this section.

6.1.1. Exploiting type information

A brief look at the output of the naive algorithm shows that the vast majority of operators is useless. For example, `load-truck(city1, truck4, airport1)` requires that `package(city1)` is true, which is never the case in a reachable state. Indeed, even though the Logistics domain from IPC1 is untyped, it is easy to see that the unary predicates `airplane`,

airport, city, location, package and truck are used as "implicit types": they are static (do not appear in operator effects) and serve to restrict the possible instantiations of the operator parameters.

As a first enhancement, we can take such (implicit or explicit) typing information into account. The example task has 42 packages, 83 trucks, 5 airplanes, 20 cities and 340 locations (17 in each city). One location in each city is an airport, so there are 20 airports. Exploiting this information, we only need to generate $42 \cdot 83 \cdot 340$ ground instances of load-truck and unload-truck, $83 \cdot 340 \cdot 340 \cdot 20$ ground instances of drive-truck, $42 \cdot 5 \cdot 340$ ground instances of load-airplane and unload-airplane and $5 \cdot 20 \cdot 20$ ground instances of fly-airplane, for a total of $1.94 \cdot 10^8$ ground operators (second row of Fig. 11). Despite the improvement of more than two orders of magnitude, this is still infeasibly large for a 2 GB memory limit.

6.1.2. Checking static preconditions

The vast majority of generated ground operators are instances of drive-truck, and most of them are not useful because the preconditions on the in-city predicate can never be true. Like the type predicates previously considered, in-city is a *static* predicate, so a natural second enhancement would be to avoid generating operators which violate a precondition on *any* static predicate, not just unary ones. This reduces the number of ground instances for drive-truck to $83 \cdot 340 \cdot 17 \cdot 1$ (after parameters t and l are instantiated arbitrarily, there are only 17 valid options for parameter l' and only one valid option for parameter c). The other operators do not mention any non-unary static predicates, so their number of ground instances remains the same, for a total of $3.00 \cdot 10^6$ ground operators (third row of Fig. 11), another improvement of roughly two orders of magnitude.

An important caveat, though, is that unlike the naive grounding algorithm and the algorithm that takes type information into account, a grounding algorithm that filters on static preconditions cannot be implemented in such a way that it is guaranteed to run in linear time in the size of its output, unless $\mathsf{P} = \mathsf{NP}$. To see this, consider a planning task with just a single operator, whose precondition is a conjunction of atoms over static predicates. Such an operator then defines a *constraint satisfaction problem* (CSP) [10], where the domains of the variables are given by the objects of the task, the constraint schemas are given by the facts for the static predicates in the initial state, and the preconditions of the operator correspond to the constraints. Deciding whether or not an operator has *any* valid instantiation is thus akin to deciding CSP solvability, which is NP-complete. The problem is already NP-hard in the case where (using planning terminology instead of CSP terminology) there are only three objects and predicates are at most binary, by reduction from 3-COLORABILITY [22, problem GT4], or in the case where there are only two objects and predicates are at most ternary, by reduction from 3-SAT [22, problem LO2]. The actual problem we need to solve, finding all valid instantiations of an operator, corresponds to finding *all* solutions to a CSP. This is even harder: even with only two objects and at most binary predicates (a case where deciding solution existence is easy), *counting* the number of solutions is #P-complete [8].

For this reason, a grounding algorithm that checks all static preconditions will in general do some wasteful work. One approach is to enumerate all operators that satisfy the type constraints and reject those which violate a static precondition. This approach is no faster than the algorithm that generates all type-correct operators without checking non-unary static predicates, but it generates fewer operators and hence requires less space, since infeasible ground operators can be immediately removed.

A more elaborate idea is to reject a partially instantiated operator as soon as one of its static preconditions can no longer become true given the current partial assignment. This is the approach taken by the IPP grounding algorithm [37], and it is closely related to a technique called *forward checking* in the CSP literature [10], generalized to constraints on more than two variables.

The pruning power of the forward checking approach depends on the order in which variables of an operator are instantiated. To see this, let us return to the LOGISTICS example. The only operator where we can potentially obtain a benefit over the simpler type-checking algorithm is drive-truck. If its parameters are instantiated in the order shown in Fig. 10, the forward checking technique offers no benefit over the simpler approach: for all type-correct choices of truck t and locations l and l', there exists a feasible value for the city c for each static precondition. (In most cases, there exists no value for c that satisfies both in-city preconditions *simultaneously*, but this is not detected by the forward-checking algorithm.)

However, if the variables are instantiated in the opposite order, many partial instantiations of l' and c or of l, l' and c can be rejected immediately. The total number of partial instantiations to consider is then limited to $1 + 20 + 20 \cdot 340 + 20 \cdot 17 \cdot 340 + 20 \cdot 17 \cdot 17 \cdot 83 = 602161$ (counting, in this sequence, instantiations of $0, 1, \ldots, 4$ variables). This is only 26% larger than the number of ground operators that satisfy all static preconditions, 479740. In contrast, the simpler algorithm needs to test $1.92 \cdot 10^8$ instances of this operator, which is a factor of 400 larger than the number of surviving instantiations.

The price for the improvement offered by the forward checking algorithm is that large indexing structures are needed for efficiently checking whether a given partial instantiation of a static predicate can be extended to a full instantiation. We believe that the space requirements of these indexing structures are to a large degree responsible for the failures of the IPP/FF grounding algorithm on some IPC tasks. We remark that the size of these index structures grows exponentially with the arity of predicates, and the domains on which we observe failures are all among the few domains in the IPC suite where predicate arity is not bounded by 2 (the maximal arity is 3 in PATHWAYS and 4 in OPTTELEGRAPH and PSR).

6.1.3. Checking relaxed reachability

In practice, for our Logistics example, the grounding algorithm that filters on static preconditions is fast enough to allow grounding in reasonable time, at least if we use the forward checking idea and instantiate variables in a favourable order. However, significant enhancements are still possible and worthwhile, as reducing the number of ground operators has a beneficial influence on most planning algorithms. For example, the per-state overhead of a search algorithm tends to increase with the number of ground operators it must test for applicability.

In the Logistics example, more than 90% of the generated operators are still unreachable. The reason for this is that these operators have infeasible preconditions involving the *non-static* predicate `at`. In particular, a truck can never be at a location that does not belong to its initial city, and an airplane can never be at a non-airport location. By respecting these constraints, we can restrict the number of ground instances to $42 \cdot 83 \cdot 17$ for `load-truck` and `unload-truck`, to $83 \cdot 17 \cdot 17$ for `drive-truck`, and to $42 \cdot 5 \cdot 20$ for `load-airplane` and `unload-airplane`. (The number of instantiations of `fly-airplane` cannot be reduced further.) This results in a total of $152911 = 1.53 \cdot 10^5$ ground operators, another improvement by more than an order of magnitude (fourth row of Fig. 11).

It is thus desirable to also rule out ground operators with infeasible preconditions on non-static predicates. However, there is a problem: checking whether a given atom can ever be satisfied in a reachable state is as hard as planning itself. Thus, in practice we need to compute an approximation of the set of reachable facts, which is at the same time conservative (includes all reachable facts) but also tight (excludes as many facts as possible). One such approximation method is the use of *delete relaxations* [33]. Instead of computing the set of reachable facts of a normalized PDDL task Π itself, we compute the reachable facts of a relaxed planning task $\mathcal{R}(\Pi)$, which differs from Π as follows:

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

The set of reachable atoms of $\mathcal{R}(\Pi)$ is a superset of the set of reachable atoms of Π. (This follows from the fact that the so-called h^+ heuristic is *completeness-preserving*; see Hoffmann's article [32] for details.) In many practical cases, this superset relationship is quite tight. For example, in the Logistics domain, the sets of reachable atoms of $\mathcal{R}(\Pi)$ and Π are identical. In other words, if we restrict the grounded representation of the Logistics task to those operators whose preconditions are reachable in the delete relaxation, then no further pruning is possible without removing operators that are actually reachable. (However, we remark that one may in some cases safely remove reachable operators. In our example, the grounded task contains about 1% *no-op* operators, namely movements of a vehicle from location l to l itself. The last row of Fig. 11 shows the number of remaining operators after no-op pruning. Our grounding procedure does not detect such no-ops, but they are filtered out in a final post-processing stage when the finite-domain representation is generated.)

Computing the reachable atoms of a relaxed planning task is much easier than for general planning tasks. In particular, if a relaxed planning task is already grounded, its reachable atoms can be computed in linear time with the marking algorithm for propositional Horn logic [12]. Indeed, the grounding algorithm of FF proceeds by first applying the IPP grounding algorithm (i.e., the forward checking algorithm described above), and then computing the reachable atoms and operators of the delete relaxation to further reduce the grounded representation. While this leads to a tight representation, the drawback of the approach is that it has the same time and space requirements as the IPP algorithm, as it generates the IPP output as an intermediate result. As discussed in the introduction to this section, there are a number of planning tasks for which this approach fails.

For this reason, we have designed a new grounding algorithm which generates the set of facts and operators (and axioms) that are reachable in the relaxed task *directly*, without ever considering any facts that are not relaxed reachable. We now turn to a description of this algorithm.

6.2. Overview of Datalog exploration

The basic idea of our new grounding algorithm, which we call *Datalog 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 consists of the set of ground atoms that it logically implies. The algorithm consists of three steps: generating the logic program, translating it into a normal form that supports efficient evaluation, 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. As in the other parts of the paper, we assume that our logical vocabulary does not contain function symbols of non-zero arity.

Definition 13 *(Datalog programs).* A **Datalog rule** (also called **positive Horn clause**) is a first-order formula of the form $\varphi_1 \wedge \cdots \wedge \varphi_k \rightarrow \psi$ $(k \geqslant 0)$, where φ_i and ψ are (usually not ground) atoms. It can be written as $\psi \leftarrow \varphi_1, \ldots, \varphi_k$. Using this notation, ψ is called the **head** and $\varphi_1, \ldots, \varphi_k$ is called the **body** of the rule. Datalog rules are usually assumed to be universally quantified: for a given Datalog rule χ with $\text{free}(\chi) = \{v_1, \ldots, v_k\}$, we define $\chi_\forall = (\forall v_1 \ldots v_k : \chi)$. Similarly, for a set of Datalog rules $\mathcal{R}$, we define $\mathcal{R}_\forall = \{\chi_\forall \mid \chi \in \mathcal{R}\}$.

A **Datalog program** is a pair $\langle \mathcal{F}, \mathcal{R} \rangle$, where $\mathcal{F}$ is a set of ground atoms called the set of **facts** and $\mathcal{R}$ is a set of Datalog rules called the set of **rules**.

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

Next, we show how to translate the relaxed reachability problem into a Datalog 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.

6.3. Generating the logic program

Reachability in a relaxed normalized PDDL task is straight-forward to represent as a logic program. A ground atom is reachable in the relaxed task iff it is true in the initial state or there exists some reachable 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 whether the relaxed task is solvable; if not, the original task is also unsolvable, which we can report immediately to stop the translation process early.

Recall from Section 4 that at this stage, all conditions occurring in the PDDL task are conjunctions of literals. For such conjunctions φ, we denote the conjunction of all *positive* literals in φ by φ^+. 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 and predicates. The exploration rules for a normalized PDDL task 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 φ 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 ψ with bound variables $\{Y_1, \ldots, Y_l\}$ and effect condition χ with $\chi^+ = \chi_1^+ \wedge \cdots \wedge \chi_n^+$, we generate the *effect trigger rule*

  $$e\texttt{-triggered}(X_1, \ldots, X_k, Y_1, \ldots, Y_l)$$
  $$o\texttt{-applicable}(X_1, \ldots, X_k), \chi_1^+, \ldots, \chi_n^+$$

  and *effect rule*

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

- *Goal:* For the goal φ 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, as they are just literal translations of the PDDL semantics for the relaxed planning task. The reader might wonder why we sometimes introduce new predicates that do not seem necessary for computing the set of reachable facts. For example, axiom applicability rules and axiom effect rules could be combined into a single rule without introducing the auxiliary predicate a`-applicable`. The purpose of these auxiliary 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).

6.4. Translating the logic program to normal form

Having generated a logic program representation, we need to find a way to efficiently generate the canonical model. Returning to the Logistics example, one of the generated rules is

$$\texttt{drive-truck-applicable}(T, L, L', C) \leftarrow \texttt{truck}(T), \texttt{location}(L),$$
$$\texttt{location}(L'), \texttt{city}(C), \texttt{in-city}(L, C), \texttt{in-city}(L', C), \texttt{at}(T, L).$$

Given a set of reachable facts, we need to determine the possible instantiations of the rule for which all conditions in the body are reachable. We must do this without systematically trying out all possible instantiations – otherwise, nothing is gained over the naive instantiation method. Moreover, we prefer to evaluate the rule *incrementally*: whenever a new instantiation of a predicate in the body is derived (say, the fact at(truck1, loc5)), we want to derive new consequences of the rule without re-generating previously derived facts. In order to achieve these objectives, we consider a particular class of Datalog programs.

Definition 14 *(Datalog programs in normal form).* A first-order logic atom is called **variable-unique** if it does not contain two occurrences of the same variable. (For example, an atom like $P(X, Y, X)$ is not variable-unique because variable X occurs twice. Repetition of constants is allowed.) A Datalog rule is called **variable-unique** if its head and all atoms in its body are variable-unique.

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

A Datalog rule is called a **join rule** if it is variable-unique and of the form $\varphi \leftarrow \varphi_1, \varphi_2$ with $\text{free}(\varphi_1) \cup \text{free}(\varphi_2) = \text{free}(\varphi) \cup (\text{free}(\varphi_1) \cap \text{free}(\varphi_2))$. 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 Datalog program is **in normal form** if all rules are projection or join rules.

The names of the rule types in Definition 14 are reminiscent of the related database-theoretic operations from relational algebra [45]: projection rules correspond to the projection operator π and join rules correspond to the natural join operator $\bowtie$ (or strictly speaking, a combination of natural join and projection). The advantage of Datalog programs in normal form is that each rule can be incrementally evaluated very efficiently. (We will describe an algorithm for this later on.) Note that the same is *not* true for general rules – as discussed in Section 6.1.2, deciding whether a general Datalog rule has any valid instantiation is equivalent to CSP solvability.

We now describe how to convert Datalog programs to normal form. Firstly, 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 rule body. We repeat until no further such transformations are possible. If any such transformation was necessary, we add the fact $\texttt{equals}(o, o)$ to the logic program for each object o of the planning task.

Secondly, 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 Section 4.1 that $\texttt{object}(o)$ is true for any object o of the planning task.)

Thirdly, 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. This is essentially a *conjunctive query optimization problem*, commonly studied in database theory [46]. As a first step towards normalizing such a rule, we determine if any atom in its body contains 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, following the general principle for query optimization that projections should be performed as early as possible. In detail, assume we are given the rule $\varphi \leftarrow \varphi_1, \ldots, \varphi_m$, where $\text{free}(\varphi_i) = \{X_1, \ldots, X_k\}$ 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 rules $\varphi \leftarrow \varphi_1, \ldots, \varphi_{i-1}, p(X_1, \ldots, X_j), \varphi_{i+1}, \ldots, \varphi_m$ and $p(X_1, \ldots, X_j) \leftarrow \varphi_i$.

After this transformation, all binary rules are valid join rules, while rules with $m > 2$ atoms correspond to m-ary joins which still need to be decomposed into binary joins. How exactly this decomposition is performed can in general greatly affect performance. To see this, consider the rule

$$\texttt{drive-truck-applicable}(t, l, l', c) \leftarrow \texttt{in-city}(l, c), \texttt{in-city}(l', c), \texttt{at}(t, l).$$

from Logistics. (For brevity, we omit some additional conditions from the body of the actual rule, which are not relevant to the discussion.) One possible decomposition into binary join rules results in the following two rules:

$$\texttt{temp}_1(t, l, c) \leftarrow \texttt{in-city}(l, c), \texttt{at}(t, l).$$
$$\texttt{drive-truck-applicable}(t, l, l', c) \leftarrow \texttt{temp}_1(t, l, c), \texttt{in-city}(l', c).$$

Another possible decomposition results in these rules:

$$\texttt{temp}_2(t, l, l', c) \leftarrow \texttt{in-city}(l', c), \texttt{at}(t, l).$$
$$\texttt{drive-truck-applicable}(t, l, l', c) \leftarrow \texttt{temp}_2(t, l, l', c), \texttt{in-city}(l, c).$$

```
algorithm greedy-join(rule):
    while |rule.body| > 2:
        Choose φ, φ' ∈ rule.body such that φ ≠ φ' and
          join-cost(rule, φ, φ') is minimal.
        X_1, ..., X_k := join-vars(rule, φ, φ')
        Generate a new predicate symbol p with arity k.
        Generate a new join rule p(X_1, ..., X_k) ← φ, φ'.
        rule.body := rule.body \ {φ, φ'} ∪ {p(X_1, ..., X_k)}

function join-vars(rule, φ, φ'):
    { Compute the relevant variables for the predicate generated
      by joining φ and φ'. }
    return free({φ, φ'}) ∩ free({rule.head} ∪ (rule.body \ {φ, φ'})).

function join-cost(rule, φ, φ'):
    new-arity := |join-vars(rule, φ, φ')|
    max-old-arity := max(|free(φ)|, |free(φ')|)
    min-old-arity := min(|free(φ)|, |free(φ')|)
    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 for ties. }
```

Fig. 12. The greedy join algorithm for decomposing a rule into join rules.

Of the two possibilities, the first is vastly preferable: the number of reachable instances of $\texttt{temp}_1$ is only as large as the number of reachable instances of `at`, because for any reachable fact $\texttt{at}(t,l)$, exactly one city c will satisfy $\texttt{in-city}(l,c)$. The number of reachable instances of $\texttt{temp}_2$, on the other hand, is the *product* of the number of reachable instances of `at` and the total number of locations, which is much larger than the number of reachable instances of `drive-truck-applicable`.

Finding a good decomposition is thus key to good performance. Unfortunately, finding the best possible ordering is generally not easy. Indeed, the closely related problem of *join ordering* is one of the central problems in query optimization for databases, and typical query optimizers address it by exhaustively considering an exponentially large space of possibilities [46, Chap. 11].

The fact sets we have to deal with in planning tasks are not as large as the relations considered in database systems, so our normalization procedure does not quite go to such lengths. Instead of an exhaustive search that finds a globally optimal decomposition, it applies a sequence of greedy (locally optimal, according to some simple heuristics) transformations without backtracking over any choices it makes. The algorithm, called *greedy-join*, is shown in Fig. 12. To decompose a rule, it iteratively picks two atoms φ and φ' from the rule body and joins them, introducing a new predicate p 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. In each step of the algorithm, the two atoms φ and φ' to join are picked in order to minimize the (estimated) effort for computing their join, according to the following three rules (without loss of generality, φ has at least as many variables as φ'):

(1) Prefer joins where the arity of p minus the number of variables of φ is smallest.
(2) If ties need to be broken, prefer joins where the arity of p minus the number of variables of φ' is smallest.
(3) If ties still need to be broken, prefer joins where the arity of p is lowest.

The net effect of these rules is that the algorithm prefers joining atoms with a large number of common variables. In the Logistics example above, this heuristic steers clear of the second decomposition, where the sets of variables of the joined atoms are disjoint.

6.5. Computing the canonical model

Once the Datalog program has been converted to normal form, we are ready to compute its canonical model. We use an incremental approach, shown in Fig. 13.

```
algorithm compute-canonical-model(F, R):
    for each join rule r ∈ R:
        r.index_1 := make-empty-hashtable()
        r.index_2 := make-empty-hashtable()
    rule-matcher := make-rule-matcher(R)
    queue := make-empty-queue()
    canonical-model := ∅
    { In the following, enqueuing a fact means adding it to queue and
      canonical-model if it is not yet an element of canonical-model. }
    Enqueue all facts in F.
    while queue is not empty:
        current-fact := queue.pop()
        for each match m ∈ rule-matcher.match(current-fact):
            if m refers to φ_1 in a projection rule r = φ ← φ_1:
                Let α be the variable assignment
                  for which α(φ_1) = current-fact.
                Enqueue α(φ).
            else if m refers to φ_1 in a join rule r = φ ← φ_1, φ_2:
                Let α be the variable assignment
                  for which α(φ_1) = current-fact.
                key := α restricted to free(φ_1) ∩ free(φ_2)
                Add α to r.index_1[key].
                Enqueue (α ∪ β)(φ) for each β ∈ r.index_2[key].
            else if m refers to φ_2 in a join rule r = φ ← φ_1, φ_2:
                { Analogous to the previous case. }
```

Fig. 13. Computing the canonical model of a Datalog program $\langle \mathcal{F}, \mathcal{R} \rangle$ in normal form.

The algorithm maintains two variables to keep track of reachable facts. Variable *canonical-model* records the set of all atoms that the algorithm has determined to be reachable so far. It grows monotonically throughout execution of the algorithm and contains the result upon termination. Variable *queue* keeps track of all atoms which are currently *open*, i.e., have been determined reachable, but have not yet been considered for matching conditions in the rule bodies. We say that an atom is *closed* if it is contained in *canonical-model* (has been reached) and is not open.

In addition to *canonical-model* and *queue*, the algorithm uses the following data structures:

- *Rule matcher*: A rule matcher is an indexing structure that supports efficient unification queries on the bodies of Datalog 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 φ_1 or φ_2 unifies with a, i.e., such that it is possible to substitute objects for variables in φ_1 or φ_2 in such a way that a is obtained. The rule matcher reports the matched rules and whether φ_1 or φ_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 IPC 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* [29].
- *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 φ_1 and φ_2 to instantiations of the variables of φ_1 and φ_2, respectively.
  At any time (except during updates) and for any assignment *key* to the common variables of φ_1 and φ_2, $r.\text{index}_1[\mathit{key}]$ contains those variable mappings $\alpha \supseteq \mathit{key}$ for the variables of φ_1 for which $\alpha(\varphi_1)$ belongs to the closed set. Similarly, $r.\text{index}_2[\mathit{key}]$ contains those variable mappings $\beta \supseteq \mathit{key}$ for the variables of φ_2 for which $\beta(\varphi_2)$ belongs to the closed set. We call this property the *index invariant*.
  The index information can be exploited for quickly determining all possible instantiations of φ_2 that match a given instantiation of φ_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 well-defined function, since α and β agree on all variables for which they are both defined.

The algorithm clearly terminates, as each loop iteration produces a new closed atom (by removing an open atom from *queue*), which can only happen a finite number of times. To motivate its correctness, we state an important invariant of the **while** loop: *all atoms that can be derived in one step from the rules $\mathcal{R}$ and the closed atoms are contained in the canonical-model set.*

This property is true when the while loop is first entered because there are no closed atoms at this stage, and no atoms can be derived from $\mathcal{R}$ and the empty set of atoms. It is not hard to prove that the property is also preserved by a single iteration of the **while** loop: each such iteration causes the single new atom *current-fact* to become closed, so we only need to consider one-step derivations that make use of this atom, and possibly other atoms that are already closed. The processing of *current-fact* then causes precisely those atoms which can be generated by such derivations to be enqueued, adding them to *canonical-model* if they are not already present. Thus, the property is indeed a loop invariant. (A more formal proof would need to establish this invariant simultaneously with the index invariant.)

The loop invariant implies that upon termination of the algorithm, when all atoms in the *canonical-model* set are closed, the set 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 $\langle \mathcal{F}, \mathcal{R} \rangle$.

6.6. 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`$(x_1, \dots, x_k)$ where a is a schematic axiom, we generate a ground instance of a with the parameters substituted with $x_1, \dots, x_k$.
- When encountering atoms of the form o-`applicable`$(x_1, \dots, x_k)$ 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 $x_1, \dots, x_k$, and the precondition is instantiated accordingly.
- When encountering atoms of the form e-`triggered`$(x_1, \dots, x_k, y_1, \dots, y_l)$ where e is an effect of some operator o, we look up the set of already generated ground operators to find the operator $o(x_1, \dots, x_k)$. 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 $y_1, \dots, y_l$.

After a single pass through the canonical model, this procedure has produced a grounded representation of the normalized PDDL task.

6.7. Performance

We conclude our discussion of the Datalog exploration algorithm with some remarks on performance. In practice, the only performance-critical part for grounding is algorithm *compute-canonical-model*: all the processing that happens before is generally negligible (linear-time for a fixed domain), and the processing that happens afterwards, while not always negligible in absolute terms, only requires linear time in the combined size of its input and output, which is asymptotically optimal.

How costly, then, is computing the canonical model? To simplify the analysis, we only consider the case where the planning domain is fixed. In that case, the *rules* of the Datalog program are fixed, and only the facts differ from instance to instance. Under this assumption, the runtime of *compute-canonical-model* is linear in the number of attempts to *enqueue* a fact (see Fig. 13). In the best case, this number is identical to the number of facts in the canonical model for the initial Datalog program (before normalization), so that the overall grounding algorithm runs in linear time in its combined input and output size and is hence asymptotically optimal. However, this is not always the case, as there are two possible sources of inefficiency:

- *Duplicates*: There may be several attempts to enqueue the same fact (e.g., if projection rules project different facts to the same fact, or because the fact is generated by different rules). For a given run of the algorithm, we define the *duplicate ratio* as the total number of attempts to enqueue a fact, divided by the size of the canonical model upon completion.
- *Irrelevant facts*: Facts that correspond to temporary predicates introduced during normalization of the Datalog program are irrelevant for the final instantiation stage. For a given run of the algorithm, we define the *irrelevance ratio* as the size of the canonical model, divided by the number of facts in the canonical model that are relevant (i.e., do not refer to predicates introduced during normalization).

The overhead of *calculate-canonical-model*, compared to an idealized algorithm that could generate the set of relevant facts in linear time in its size, is the product of the duplicate ratio and irrelevance ratio. In the perfect case, both numbers would be equal to 1. However, as we saw in Section 6.4 for the different decompositions of the `drive-truck-applicable` rule into binary joins, the irrelevance ratio in particular can be considerably larger, especially if the greedy join algorithm makes poor choices. Indeed, the existence of a grounding algorithm with only polynomial overhead in the *general case*, where the Datalog rules are not fixed, would prove $\mathsf{P} = \mathsf{NP}$.

It is thus natural to ask how large these ratios become in practice. To answer that question, we applied the grounding algorithm to all tasks of the IPC1–5 benchmark suite, measuring the duplicate ratio and irrelevance ratio for each. The results

domain	DR	IR	domain	DR	IR
Airport	1.16–1.49	1.90–3.41	OptTelegraph	1.22–1.27	2.75–2.82
Assembly	1.34–1.57	1.09–1.28	Pathways	1.12–1.60	1.48–1.72
Blocksworld	1.56–1.72	1.67–1.67	Philosophers	1.10–1.13	2.66–2.79
Depot	1.35–2.47	1.50–2.48	Pipes-NoTank	1.40–2.15	2.30–3.44
Driverlog	1.19–1.65	1.36–2.26	Pipes-Tank	1.35–3.91	1.43–4.99
FreeCell	2.01–3.48	1.26–2.28	PSR-Large	1.03–1.07	3.04–3.65
Grid	1.99–2.23	1.34–1.49	PSR-Middle	1.02–1.08	3.01–3.70
Gripper	1.28–1.35	1.68–1.74	PSR-Small	1.32–1.99	1.00–1.00
Logistics	1.11–1.53	1.48–2.29	Rovers	1.05–1.35	2.35–27.96
Miconic-Full	1.08–3.00	1.65–6.48	Satellite	1.22–1.97	1.02–1.72
Miconic-Simple	1.11–1.74	1.33–1.63	Schedule	1.70–2.65	1.19–1.27
Miconic-STRIPS	1.02–1.34	1.74–2.29	Storage	1.08–2.41	1.56–2.46
Movie	1.18–1.24	1.00–1.00	TPP	1.03–1.64	2.34–3.71
MPrime	1.76–2.74	1.12–1.84	Trucks	1.70–2.71	1.03–1.37
Mystery	1.32–2.47	1.28–2.25	Zenotravel	1.42–2.42	1.22–2.51
Openstacks	1.54–2.54	1.05–1.39			

Fig. 14. Duplicate ratios (DR) and irrelevance ratios (IR) for the IPC benchmarks. For each domain, we report the range of ratios observed across all instances.

are summarized in Fig. 14, which shows the minimal and maximal duplicate and irrelevance ratios that were observed in each domain. The results show that the duplicate ratios are generally benign, always staying below a value of 4. The irrelevance ratios are also usually low, consistently staying below a value of 5 for all but two domains: Miconic-Full and Rovers. For Miconic-Full, the irrelevance ratios can become as large as 6.48; however, the ratios are actually inversely correlated with problem size, becoming lower as the problem sizes increase. Only the 20% smallest instances have irrelevance ratios over 3, and all instances of above average size have irrelevance ratios below 2. Thus, there are no serious scalability issues in this domain.

The only domain which produces somewhat worrying results is Rovers, where the irrelevance ratios become as large as 27.96, significantly more than for all other domains. Moreover, the ratios tend to increase with scaling problem size. Closer inspection reveals that an unfortunate choice by the greedy join algorithm is to blame for this sub-par performance. In normalizing the rule for the predicate communicate-rock-data-applicable(r, l, p, x, y), which has 11 conditions in the body, the following four conditions remain after 7 join steps:

$$\mathtt{temp}_1(r,p) \equiv \mathtt{waypoint}(p) \wedge \mathtt{have\text{-}rock\text{-}analysis}(r,p)$$

$$\mathtt{temp}_2(r,x) \equiv \mathtt{rover}(r) \wedge \mathtt{available}(r) \wedge \mathtt{waypoint}(x) \wedge \mathtt{at}(r,x)$$

$$\mathtt{temp}_3(l,y) \equiv \mathtt{lander}(l) \wedge \mathtt{chan\text{-}free}(l) \wedge \mathtt{waypoint}(y) \wedge \mathtt{at\text{-}lander}(l,y)$$

$$\mathtt{visible}(x,y)$$

At this point, there are three possible joins that the greedy join algorithm considers optimal: joining $\mathtt{temp}_1$ with $\mathtt{temp}_2$, joining $\mathtt{temp}_2$ with visible, and joining $\mathtt{temp}_3$ with visible. Of these three possibilities, our implementation arbitrarily picks the second, which happens to be the worst possible choice because there is a very large number of reachable instantiations for $\mathtt{temp}_2(r,x) \wedge \mathtt{visible}(x,y)$, most of which cannot be extended to feasible assignments for all conditions. The reason for this is that there is only one value for y for which $\mathtt{temp}_3(l,y)$ is reachable (there is only one lander and it is not mobile, so that at-lander has only one reachable instance).

To determine the impact of this unfortunate tie breaking decision, we have instrumented the greedy join algorithm to override its default choice and instead join $\mathtt{temp}_3$ with visible at this choice point, and similarly for an equivalent choice point for the isomorphic communicate-soil-data-applicable predicate. With this modification, the worst-case irrelevance ratio in the Rovers domain reduces from 27.96 to 4.83. By additionally overriding the subsequent join decision to prevent the algorithm from joining $\mathtt{temp}_1$ with $\mathtt{temp}_2$, the worst-case irrelevance ratio can be further reduced to 1.97. In terms of overall runtime for the grounding algorithm, the reduced irrelevance ratio translates to an order-of-magnitude improvement.

This small case study illustrates two things: on the one hand, we clearly see that join ordering choices can have a significant impact on the performance of the grounding algorithm and thus do require attention. On the other hand, the essentially linear scaling behaviour for all domains except Rovers indicates that the heuristic decisions of the greedy join algorithm are usually quite solid.

algorithm compute-mutex-groups(*invariants*, $\mathcal{P}_f$, s_0):
  for each invariant $I \in$ *invariants*:
    for each instance α of I:
      if weight$(\alpha, s_0) = 1$:
        Create a mutex group containing all atoms in $\mathcal{P}_f$ covered by α.

Fig. 15. Computing mutex groups from the set of monotonicity invariants *invariants*, the set of reachable atoms $\mathcal{P}_f$ and the initial state s_0.

(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. 16. Mutex groups for a BLOCKSWORLD task with four blocks. Some atoms, such as on(a, a), are reachable in the relaxed task although they are never true in the "real" task.

7. Generating the finite-domain representation

Together with the invariants synthesized earlier, the grounded PDDL task generated in the previous stage provides all the information we need for producing the finite-domain representation in the final translation stage. Recall from Definition 4 that an FDR task is given by a set of finite-domain variables $\mathcal{V}$, an 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.

7.1. Variable selection

Each variable of the generated FDR task corresponds to one or more (reachable) ground atoms of the STRIPS task. We start by extracting the set $\mathcal{P}$ of all such atoms from the canonical model and partitioning them 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. Section 5.1).

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 straight-forward manner by instantiating the monotonicity invariants in all possible ways, checking for each instance 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. 15. The actual implementation uses an indexing structure to efficiently determine the set of reachable atoms covered by a given invariant instance.

Normally, not every mutex group will correspond to an FDR 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. 16, 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 finite-domain state variables, then we only need to encode one atom from each of the other groups, since all on and holding atoms 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(a, x), on(b, x), on(c, x), on(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.) Afterwards, it would encode the truth values of the remaining atoms ontable(x) and handempty() 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 prefer covering the set of reachable atoms with as few state variables as possible.

Unfortunately, set cover problems of this kind are NP-complete [22, problem SP5] and indeed not even approximable with a constant-factor approximation ratio [1], so we limit our covering efforts to the greedy algorithm shown in Fig. 17, which is among the best approximation algorithms known for this problem, achieving an $O(\log n)$-approximation [1]. Iteratively, we pick a mutex group P of maximal cardinality and introduce a new FDR 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 $\{p, \bot\}$.

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 a 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 a variable v with domain $\{p, \bot\}$ for each remaining
   element of *uncovered*.

Fig. 17. Greedy algorithm for computing the FDR variables and variable domains.

After execution of the algorithm, for each reachable atom $p \in \mathcal{P}_f$ there is exactly one FDR 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 FDR 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 finite-domain representation.

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

7.3. 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 FDR 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 behaviour 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. In other situations, the delete effect should not cause a change in the value of $var(p)$. If the other effects of the operator that add atoms represented by $var(p)$ have effect conditions $\chi_1, \dots, \chi_k$, then this is achieved by adding $p \wedge \neg\chi_1 \wedge \cdots \wedge \neg\chi_k$ to the effect condition of the translated effect.

If some of the formulae χ_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 simpler 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 finite-domain representation 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 unconditional effects, one of the special cases always applies.

7.4. Converting conditions

The third major translation step is the conversion of conditions of the grounded PDDL task. Conditions occur in the goal specification, in operator preconditions, in conditional effects 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).

If the condition is not trivially false, we translate each of its positive literals p into the pairing $var(p) = p$. Translating negative literals $\neg p$ is slightly more tricky. Recall the BLOCKSWORLD example discussed earlier, where we generated the FDR 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 represented by variable v, for example the atom clear(a), then we do not need to encode $\neg\texttt{on(c,a)}$ at all, because it is implied by the condition $v = \texttt{clear(a)}$. However, otherwise there is no simple way to represent $\neg\texttt{on(c,a)}$ as an FDR 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 non-conjunctive effect conditions that may arise for certain delete effects, we introduce a new derived variable not-p with domain $\{\top, \bot\}$ and generate an axiom $(v = d) \rightarrow (\texttt{not-}p := \top)$ for each value $d \in \mathcal{D}_v \setminus \{p\}$. The pairing $\texttt{not-}p = \top$ 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 compilation methods to achieve such a normal form [23]. However, this transformation method may introduce many more state variables than necessary, which runs counter to our objective of a concise finite-domain representation. (If it can be avoided, there is little point in having two finite-domain variables in the representation with the property that the value of each is a function of the value of the other.) As a compromise, we might consider to *only* use such a compilation method for propositions that appear in negative literals for which our translation method would otherwise introduce a new axiom. In practice (on the IPC domains), negative conditions are not commonly used, so the choice of mechanism for dealing with them makes little difference.

7.5. Computing axiom layers

As a final translation step, we must compute the axiom layers for the finite-domain representation in such a way that the semantics for stratified logic programs is matched (cf. Definitions 2 and 5).

This is done as follows: whenever the body of an axiom with affected variable v includes the condition $v' = \bot$ for some derived variable v', then the value of v' must be computed before the value of v, so we introduce an ordering constraint $v' \prec v$. Similarly, if some axiom affecting v includes the condition $v' = d$ for derived variable v' and $d \neq \bot$, we introduce an ordering constraint $v' \preceq v$.

Axiom layers can be derived from these ordering constraints in two stages. In the first stage, we compute the strongly connected components of the graph induced by the $\preceq$ relation. All axioms that affect variables in the same strongly connected component belong to the same axiom layer.

To compute the ordering of axiom layers, we topologically sort the graph whose vertices are the axiom layers and which has an arc from layer L' to layer L iff some axiom in L' affects v', some axiom in L affects v, and $v' \prec v$. The ith axiom layer is then the one which appears at the ith position in the computed topological order. If the axioms of the original PDDL task are stratifiable, such a topological sort is always possible.

7.6. Post-processing

With axioms partitioned into layers, the translation is complete. Before generating output, we apply 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 $a' = (cond' \rightarrow v := d)$ such that $cond \subset cond'$, 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 remove a' from the representation.

8. Discussion

Having finished our presentation of the translation algorithm, the question arises how one should evaluate it. There are two important criteria to consider:

- *Performance:* How quickly is the representation computed?
- *Quality:* How good is the generated finite-domain representation?

8.1. Notes on performance

We will keep the discussion of performance short: on a state-of-the-art computer, the translation algorithm is sufficiently efficient to generate finite-domain representations for all IPC1-5 benchmark tasks in reasonable time. In particular, in the different planning systems that use the translator, including Helmert and Richter's Fast Downward [29], van den Briel et

Runtime at most 1s	1054 tasks
Runtime 1s–10s	452 tasks
Runtime 10s–30s	67 tasks
Runtime 30s–60s	10 tasks
Runtime 60s–120s	14 tasks

Runtime beyond 120s:

SATELLITE #35	129.79s	PSR-LARGE #49	306.30s
PSR-LARGE #47	145.80s	SATELLITE #32	371.98s
SATELLITE #31	147.97s	PSR-LARGE #44	376.69s
PSR-LARGE #42	154.36s	PSR-LARGE #46	570.23s
PSR-LARGE #43	181.99s	PSR-LARGE #48	714.73s
PSR-LARGE #45	182.47s	SATELLITE #33	805.66s
SATELLITE #36	188.52s	PSR-LARGE #50	1190.82s
PSR-LARGE #40	197.57s		

Fig. 18. Absolute runtime on the IPC1-5 tasks.

al.'s Integer Programming planner [49] and Helmert et al.'s *flexible abstraction heuristics* system [31], the conversion to FDR is never the main bottleneck of the overall planning algorithm – translation time is negligible compared to search time in the vast majority of cases. (Some exceptions to this exist in "structurally simple" domains like SATELLITE and LOGISTICS.)

A summary of translation time for the IPC1-5 benchmark suite is shown in Fig. 18. All experiments were conducted on a machine with a 2.66 GHz Intel Xeon CPU under a 2 GB memory limit. The algorithm was implemented in the Python language, and we estimate that a speed improvement by a factor of 10–100 is easily achievable with a C++ implementation. Even so, 65.4% of the instances are translated in less than one second, and 93.4% in less than ten seconds. Fewer than 2% require more than one minute, and fewer than 1% require more than two minutes. The instances that take longest to translate are all from the PSR-LARGE and SATELLITE domains and are very large: for example, the largest SATELLITE instance has 989250 operators, and the largest PSR-LARGE instance has 544209 state variables (however, a backchaining analysis reveals that only 60467 of these are relevant to the goal).

In addition to overall runtime, it is also interesting to identify which of the different stages of the algorithm form the main bottlenecks, and whether this varies from domain to domain. To address this question, we produced detailed runtime results for all tasks that required an overall runtime beyond 1 s in the first experiment. For each domain, we report the *minimal* and *maximal* percentage of overall runtime spent on each stage of the algorithm (Fig. 19). For example, the entry in column "FDR" for the LOGISTICS domain indicates that on those LOGISTICS tasks which required an overall runtime above one second (21 of the 63 instances), between 21% and 30% of the overall runtime was spent on the final stage of the algorithm, generating the finite-domain representation.

The results show that normalization and invariant synthesis are not time critical. In absolute numbers, normalization never requires more than 0.74 s, and invariant synthesis never requires more than 0.38 s. In a typical domain, about 70% of the time is required for grounding, and about 25% for FDR generation. There are four domains where grounding can require more than 85% of the runtime (MICONIC-FULL, PSR-LARGE, PSR-MIDDLE, ROVERS). Relating this observation to our earlier analysis of the Datalog grounding algorithm (Fig. 14), these are precisely the four domains with the highest average irrelevance ratio. For completeness, we mention that within the grounding stage, the vast majority of time is spent computing the canonical model (Section 6.5) and performing the final operator and axiom instantiation (Section 6.6). The initial steps of the algorithm, generating (Section 6.3) and normalizing (Section 6.4) the logic program together require at most 1.88 s, and there is only one task (PSR-SMALL #25) for which they require more than 1 s.

8.2. Notes on quality

So what about the quality of the representation? In almost all planning domains we considered (including all IPC1-5 domains), the algorithm generates finite-domain representations that are very close or identical to one we would have designed manually. However, while this is encouraging, it is not how the quality of the representation should be measured.

Ultimately, whether or not a finite-domain representation is useful depends on how well it serves its intended purpose. In Section 1.3, we discussed a number of possible uses for finite-domain representations, including SAT planning, symbolic state space exploration with BDDs, heuristic planning with pattern databases and other homomorphism abstractions, planning using integer programming compilations, and heuristic planning based on causal graph decompositions.

For SAT planning, the MaxPlan system [7], joint winner of the optimal propositional track of IPC5, has clearly established the usefulness of finite-domain representations. Although the exact details of the FDR translation method of MaxPlan have not been published, it is directly inspired by the techniques presented in this article, and follows a very similar overall strategy (personal communications). As Chen et al. report, finite-domain representations play a critical role in planner performance [7] – in the binary state variable case, the *londex constraints* that are the key innovation of MaxPlan reduce

domain	#tasks	I/O	norm.	invar.	ground.	FDR
AIRPORT	30/50	8–18%	1–4%	2–8%	48–64%	20–27%
ASSEMBLY	4/30	5–8%	$< 1\%$	$< 1\%$	80–84%	10–13%
DEPOT	7/22	4–6%	$< 1\%$	$< 1\%$	65–70%	26–30%
DRIVERLOG	5/20	4–6%	$< 1\%$	$< 1\%$	69–73%	22–26%
FREECELL	76/80	3–5%	$< 1\%$	$< 1\%$	67–71%	24–28%
GRID	4/5	4%	$< 1\%$	$< 1\%$	69–72%	24–27%
LOGISTICS	21/63	3–6%	$< 1\%$	$< 1\%$	68–75%	21–30%
MICONIC-FULL	95/150	1–5%	0–2%	$< 1\%$	84–92%	7–10%
MPRIME	23/35	3–6%	$< 1\%$	$< 1\%$	70–75%	22–27%
MYSTERY	14/30	3–6%	$< 1\%$	$< 1\%$	70–74%	22–27%
OPENSTACKS	7/30	4–6%	$< 1\%$	$< 1\%$	63–65%	30–33%
OPTTELEGRAPH	41/48	4–7%	$< 1\%$	$< 1\%$	60–68%	25–35%
PIPES-NOTANK	22/50	3–6%	$< 1\%$	$< 1\%$	71–75%	20–24%
PIPES-TANK	43/50	2–5%	$< 1\%$	$< 1\%$	71–76%	19–26%
PSR-LARGE	43/50	5–7%	$< 1\%$	$< 1\%$	80–87%	7–16%
PSR-MIDDLE	27/50	5–8%	$< 1\%$	$< 1\%$	81–87%	8–14%
PSR-SMALL	2/50	29–34%	4%	1–11%	39–56%	11–13%
ROVERS	22/40	2–7%	$< 1\%$	$< 1\%$	77–95%	4–17%
SATELLITE	18/36	1–6%	$< 1\%$	$< 1\%$	66–75%	20–34%
STORAGE	12/30	5–6%	$< 1\%$	$< 1\%$	64–67%	27–32%
TPP	15/30	3–5%	$< 1\%$	$< 1\%$	73–76%	20–24%
TRUCKS	20/30	5–9%	$< 1\%$	$< 1\%$	62–69%	25–33%
ZENOTRAVEL	7/20	4%	$< 1\%$	$< 1\%$	73–76%	21–24%

Fig. 19. Distribution of runtimes across the different stages of the translation algorithm, by domain. Only tasks with overall runtime above 1 s are taken into account. The second column shows the number of such tasks in each domain, together with the number of total tasks in that domain. The following columns show, in order, the percentage of time that was spent on input/output (including parsing), normalization (Section 4), invariant synthesis (Section 5), grounding (Section 6) and FDR generation (Section 7). Ranges denote the minimum and maximum percentage for all considered tasks in the domain.

to the usual mutex constraints used by all state-of-the-art SAT planners, so londex constraints require non-trivial FDR encodings.

For planning with BDDs, finite-domain representations have always been critical for performance, to the point where compilations that use a direct propositional encoding lead to prohibitively bad performance. Indeed, this is the reason why Edelkamp and Helmert's algorithm for devising concise finite-domain representations [15] was originally developed. We remark that for those domains that their algorithm can handle – STRIPS domains without constants in operator definitions – the encodings generated by the algorithm presented here are generally equivalent to those found by the Edelkamp and Helmert algorithm.

For planning with homomorphism abstractions, the situation is similar. For example, the flexible abstraction heuristics of Helmert et al. [31] critically rely on the finite-domain representation generated by the method presented here, degrading by several orders of magnitude and solving significantly fewer tasks if a direct propositional encoding is used instead.

The same performance degradation can be observed in heuristic planning based on causal graph decompositions [29] – without the concise FDR translation, the causal graph heuristic is not competitive with other approaches.

Finally, van den Briel et al. [49] use our FDR translation algorithm within their Integer Programming compilation approach and report significant performance advantages over earlier approaches based on propositional encodings.

In summary, there is a wide spectrum of planning techniques that can significantly benefit from automatically derived concise finite-domain representations using the techniques presented in this work.

Acknowledgements

Many thanks to Silvia Richter, Sylvie Thiébaux and the anonymous reviewers for their very helpful feedback on earlier drafts of this paper.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, Complexity and Approximation, Springer-Verlag, 1999.

[2] C. Bäckström, B. Nebel, Complexity results for SAS$^+$ planning, Computational Intelligence 11 (4) (1995) 625–655.

[3] J. Benton, M. van den Briel, S. Kambhampati, A hybrid linear programming and relaxed plan heuristic for partial satisfaction planning problems, in: Boddy et al. [4], pp. 34–41.

[4] M. Boddy, M. Fox, S. Thiébaux (Eds.), Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007), AAAI Press, 2007.
[5] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1) (2001) 5–33.
[6] T. Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1–2) (1994) 165–204.
[7] Y. Chen, X. Zhao, W. Zhang, Long-distance mutual exclusion for propositional planning, in: M.M. Veloso (Ed.), Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007), 2007.
[8] N. Creignou, S. Khanna, M. Sudan, Complexity Classifications of Boolean Constraint Satisfaction Problems, SIAM Monographs on Discrete Mathematics and Applications, vol. 7, SIAM, 2001.
[9] S. Cresswell, M. Fox, D. Long, Extending TIM domain analysis to handle ADL constructs, in: AIPS '02 Workshop on Knowledge Engineering Tools and Techniques for A.I. Planning, 2002.
[10] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[11] M.B. Do, S. Kambhampati, Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP, Artificial Intelligence 132 (2) (2001) 151–182.
[12] W.F. Dowling, J.H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, Journal of Logic Programming 1 (3) (1984) 367–383.
[13] H.-D. Ebbinghaus, J. Flum, W. Thomas, Mathematical Logic, 2nd ed., Springer-Verlag, 1994.
[14] S. Edelkamp, Planning with pattern databases, in: A. Cesta, D. Borrajo (Eds.), Pre-proceedings of the Sixth European Conference on Planning (ECP'01), Toledo, Spain, 2001.
[15] S. Edelkamp, M. Helmert, Exhibiting knowledge in planning problems to minimize state encoding length, in: S. Biundo, M. Fox (Eds.), Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99), in: Lecture Notes in Artificial Intelligence, vol. 1809, Springer-Verlag, Heidelberg, 1999.
[16] S. Edelkamp, M. Helmert, On the implementation of MIPS, in: P. Traverso, M. Veloso, F. Giunchiglia (Eds.), Proceedings of the AIPS-2000 Workshop on Model-Theoretic Approaches to Planning, 2000.
[17] S. Edelkamp, M. Helmert, The model checking integrated planning system (MIPS), AI Magazine 22 (3) (2001) 67–71.
[18] S. Edelkamp, J. Hoffmann, PDDL2.2: The language for the classical part of the 4th International Planning Competition, Tech. Rep. 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 2004.
[19] K. Erol, D.S. Nau, V.S. Subrahmanian, Complexity, decidability and undecidability results for domain-independent planning, Artificial Intelligence 76 (1–2) (1995) 65–88.
[20] M. Fox, D. Long, The automatic inference of state invariants in TIM, Journal of Artificial Intelligence Research 9 (1998) 367–421.
[21] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research 20 (2003) 61–124.
[22] M.R. Garey, D.S. Johnson, Computers and Intractability — A Guide to the Theory of NP-Completeness, Freeman, 1979.
[23] B.C. Gazen, C.A. Knoblock, Combining the expressivity of UCPOP with the efficiency of Graphplan, in: S. Steel, R. Alami (Eds.), Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97), in: Lecture Notes in Artificial Intelligence, vol. 1348, Springer-Verlag, 1997.
[24] A. Gerevini, D. Long, Plan constraints and preferences in PDDL3, Tech. Rep. R. T. 2005-08-47, Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia, 2005.
[25] A. Gerevini, L. Schubert, Inferring state constraints for domain-independent planning, in: C. Rich, J. Mostow (Eds.), Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), AAAI Press, 1998.
[26] A. Gerevini, L. Schubert, Discovering state constraints for planning: DISCOPLAN, Tech. Rep. 2005-09-48, Department of Electronics for Automation, University of Brescia, 2005.
[27] P. Haslum, A. Botea, M. Helmert, B. Bonet, S. Koenig, Domain-independent construction of pattern database heuristics for cost-optimal planning, in: Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-2007), AAAI Press, 2007.
[28] M. Helmert, A planning heuristic based on causal graph analysis, in: S. Zilberstein, J. Koehler, S. Koenig (Eds.), Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), AAAI Press, 2004.
[29] M. Helmert, The Fast Downward planning system, Journal of Artificial Intelligence Research 26 (2006) 191–246.
[30] M. Helmert, H. Geffner, Unifying the causal graph and additive heuristics, in: Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008), 2008.
[31] M. Helmert, P. Haslum, J. Hoffmann, Flexible abstraction heuristics for optimal sequential planning, in: Boddy et al. [4], pp. 176–183.
[32] J. Hoffmann, Where 'ignoring delete lists' works: Local search topology in planning benchmarks, Journal of Artificial Intelligence Research 24 (2005) 685–758.
[33] J. Hoffmann, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[34] P. Jonsson, C. Bäckström, State-variable planning under structural restrictions: Algorithms and complexity, Artificial Intelligence 100 (1–2) (1998) 125–176.
[35] H. Kautz, B. Selman, Planning as satisfiability, in: B. Neumann (Ed.), Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 92), John Wiley and Sons, 1992.
[36] H. Kautz, B. Selman, Pushing the envelope: Planning, propositional logic, and stochastic search, in: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), AAAI Press, 1996.
[37] J. Koehler, J. Hoffmann, Handling of inertia in a planning system, Tech. Rep. 122, Albert-Ludwigs-Universität Freiburg, Institut für Informatik, 1999.
[38] D. McDermott, The 1998 AI Planning Systems competition, AI Magazine 21 (2) (2000) 35–55.
[39] E.P.D. Pednault, ADL: Exploring the middle ground between STRIPS and the situation calculus, in: R.J. Brachman, H.J. Levesque, R. Reiter (Eds.), Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89), Morgan Kaufmann, 1989.
[40] S. Richter, M. Helmert, M. Westphal, Landmarks revisited, in: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI-2008), AAAI Press, 2008.
[41] J. Rintanen, A planning algorithm not based on directional search, in: A.G. Cohn, L. Schubert, S.C. Shapiro (Eds.), Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98), Morgan Kaufmann, 1998.
[42] J. Rintanen, An iterative algorithm for synthesizing invariants, in: H. Kautz, B. Porter (Eds.), Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), AAAI Press, 2000.
[43] J. Rintanen, Compact representation of sets of binary constraints, in: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006), 2006.
[44] U. Scholz, Extracting state constraints from PDDL-like planning domains, in: AIPS-2000 Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning, 2000.
[45] J.D. Ullman, Principles of Database and Knowledge-Base Systems, vol. I: Classical Database Systems, Computer Science Press, 1988.
[46] J.D. Ullman, Principles of Database and Knowledge-Base Systems, vol. II: The New Technologies, Computer Science Press, 1989.
[47] P. van Beek, X. Chen, CPlan: A constraint programming approach to planning, in: Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), AAAI Press, 1999.

[48] M. van den Briel, J. Benton, S. Kambhampati, T. Vossen, An LP-based heuristic for optimal planning, in: C. Bessiere (Ed.), Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007), in: Lecture Notes in Computer Science, vol. 4741, Springer-Verlag, 2007.

[49] M. van den Briel, T. Vossen, S. Kambhampati, Reviving integer programming approaches for AI planning: A branch-and-cut framework, in: S. Biundo, K. Myers, K. Rajan (Eds.), Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), AAAI Press, 2005.