Position Paper: Reasoning About Domains with PDDL

Alexander Shleyfman and **Erez Karpas**
Faculty of Industrial Engineering and Management
Technion — Israel Institute of Technology

Abstract

One of the major drivers for the progress in scalability of automated planners has been the introduction of the Planning Domain Definition Language (PDDL) and the International Planning Competition (IPC). While PDDL provides a convenient formalism to describe planning problems, there is a significant gap with regards to describing domains. Although PDDL is split into a domain description and a problem description, the domain description is not enough to specify a domain completely, as it does not constrain the possible problems in the domain. For example, there is nothing in the blocksworld PDDL domain description which says that a block can not be on top of itself in the initial state. In this position paper, we argue that PDDL domains should be extended to incorporate a new section which constrains possible problems in the domain. We propose such an extension, and describe several use cases where this extension can be useful.

Introduction

The domain-independent planning community has made significant progress scaling up planners, allowing them to address bigger and more complicated problem instances. One of the major drivers for this progress has been the introduction of the International Planning Competition (IPC), with its standard language for describing planning problems — PDDL, the Planning Domain Definition Language (McDermott 2000). The PDDL language was further extended to support additional features, which were introduced in later iterations of the IPC (Fox and Long 2003; Edelkamp and Hoffmann 2004; Gerevini and Long 2005).

PDDL splits the definition of a planning problem into two parts: domain and problem. The domain describes the types of object this domain deals with, along with schemas for the predicates used to describe the state of the world and the operators used to change it. The problem describes the specific objects in the world in this problem instance, as well as the initial state and the goal. Typically, a domain in the IPC is defined by a single PDDL domain description (usually in a separate file), and a random problem instance generator[1]. Planners are then evaluated based on their performance on a set of problem instances generated by the problem generator.

While this is a reasonable way to evaluate how well planners solve planning *problems*, we claim that this makes it exteremely difficult to reason over a *domain*. For example, consider the well known BLOCKSWORLD domain, which features the predicate ON(x, y), indicating that block x is directly on top of block y. We would like to be able to prove that a block can never be on top of itself. This is fairly easy to do using techniques such as relaxed reachability. However, relaxed reachability takes an initial state as input, and the initial state is only described in the PDDL problem. In fact, there is nothing preventing us from generating an instance of BLOCKSWORLD in which ON(A, A) does appear in the initial state. Thus, in order to be able to prove that a block is never on top of itself, we would need to analyze the random problem instance generator and understand it. Furthermore, even if we were able to analyze the problem generator, some domains are defined simply by a domain description and a set of problems, making this impossible.

In this short position paper, we argue that the PDDL language needs to be further extended, in order to allow for automated reasoning about *domains*, rather than only single problem instances. We first propose such an extension, and then provide several example use cases where the ability to reason over a domain, specified using our extended version of PDDL, could be helpful.

Background

We begin with a brief review of PDDL. For the full details, we refer the reader to the various papers describing the different versions of PDDL (McDermott 2000; Fox and Long 2003; Edelkamp and Hoffmann 2004; Gerevini and Long 2005). As previously mentioned, PDDL divides the definition of a planning problem into two parts: the domain, and the problem, which typically are contained in two different files. The division allows for the same domain file to be used with multiple problem files.

A PDDL domain consists of a description of the possible types of objects in the world. A type t can inherit from another type s, so that all objects with type s are also of type t. While there is a small controversy regarding whether the type hierarchy must form a proper tree, or can be a graph, this issue is irrelevant for the purposes of this paper. The

[1]Some domains have a separate domain description for each problem instance. We will address this issue in the final discussion.

domain also consists of a set of constants, which are objects which appear in all problem instances of this domain.

The second part of the domain description is a set of predicates. Each predicate is described by a name and a signature, consisting of an ordered list of types. Given a set of objects, we can *ground* the given predicates, yielding a set of propositions which describe the state of the world. Note, wowever, that these objects are only given as part of the problem description, and not in the domain description. The domain also describes a set of derived predicates, which are predicates associated with a logical expression. The idea is that the value of each derived predicate is computed automatically by evaluating the logical expression associated with it.

Finally, the domain description consists of a set of operators. Each operator is also described by a name, a signature, a precondition, and an effect. The signature is now an ordered list of named parameters, each with a type. The precondition is a logical formula. The basic building blocks of the formula are the abovementioned predicates. These can be combined using the standard first order logic logical connectives. We remark that the predicates can only be paramterized by the operator parameters, the domain constaints, or, if they appear within the scope of a forall or exists statement, by the variable introduced by the quantifier. The effect of the operator is similar, except that it described a partial assignment, rather then a formula, and thus can not contain any disjunctions. An operator can also be grounded given a set of objects, yielding grounded actions.

A PDDL problem is much simpler than the domain. It consists of a set of objects, each associated with a type (if a type is not specified, the object is assumed to be of a default type), and a description of the initial state and the goal. The initial state is described by the list of propositions (grounded predicates) that are true in it, where any proposition that is not listed it assumed to be false. The goal is also a logical expression, similarly to the precondition of an operator, except that it can refer to all objects in the problem instance. Although the goal can be an arbitrarily complex logical expression, in most existing planning benchmarks domains, it is a simply conjunction of positive propositions. In the rest of this paper, we will assume the goal takes this simple form, and discuss more complex goals in the conclusion.

As mentioned above, a domain can be grounded given a set of objects, which are described in the problem. Most modern planners start by grounding the given planning problem, and operate on the grounded problem description. However, if our intention is to reason over a domain, this approach is not practical, as there is no single problem to ground over. In the next section, we present our proposal to extend PDDL to allow some reasoning over a domain, even when a problem instance is not given.

Extending PDDL

The heart of our proposed extension to PDDL is to add *constraints* about the problem to the domain description. Following the BLOCKSWORLD example from the introduction, we could specify that in the initial state of any legal instance of BLOCKSWORLD, no block is on top of itself. We could then use a lifted version of relaxed reachability analysis to infer that no block can ever be on top of itself. We remark that these are different than the constraints introduced in PDDL 3.0 (Gerevini and Long 2005), which constrain possible *plans* for a given problem.

```
(forall (?x) (not (init (on ?x ?x))))

(forall (?x ?y ?z) (implies
(and (init (on ?y ?x)) (init (on ?z ?x)))
(= ?z ?y)))

(forall (?x) (or
(init (on-table ?x))
(exists (?y) (init (on ?x ?y)))))

(forall (?x) (not (goal (on ?x ?x))))

(forall (?x ?y ?z) (implies
(and (goal (on ?y ?x)) (goal (on ?z ?x)))
(= ?z ?y)))
```

Figure 1: BLOCKSWORLD Domain Constraints

Specifically, we propose to add another section to the PDDL domain description, which will consist of a set of constraints. Each constraint will be a first order logic statement, which can refer to domain constants, and, of course, to variables introduced by each quantifier within its scope. However, the basic building blocks will not be predicates, but rather predicates prepended with a modal operator, specifying if this refers to the initial state or the goal. One caveat is that we can not check whether some proposition if false in the goal, as the goal is only a partial state. We also explicitly allow the usage of the (object) equality predicate. As the following examples will show, it is quite useful.

The interpretation of these constraints is, naturally, as constraints over a problem description. We can treat each problem as specifying a full initial state, and a partial goal state (as we assume the goal only describes the propositions we want to be true). Thus, we can evaluate each constraint, and check whether a given problem satisfies it.

Figure 1 shows how our extension can be applied to BLOCKSWORLD. The first constraint states that a block is never on top of itself in the initial state. The second constraint states that there can be at most one block on top of another block (i.e., if y and z are both on top of x, then they must be the same block). The third constraint states that every block must be on top of another block or on the table in the initial state. Finally, the last two constraints are similar to the first two, except they are applied to the goal.

Another example highlights the differences between two different versions of the LOGISTICS domain: the one used in the first IPC (1998) and the one used in the second IPC (2000). Even though the PDDL domain description was the same in both competitions, LOGISTICS-98 is still much harder to solve than LOGISTICS-2000. This is because in the instances generated for second IPC, there was an implicit constraint, that there is eaxctly one truck in each city. This constraint is shown in Figure 2.

```
(forall (?c - city ?s ?t - truck) (implies
   (and (exists (?l - location) (and (init (in-city ?l ?c)) (init (in ?t ?l))))
        (exists (?l - location) (and (init (in-city ?l ?c)) (init (in ?s ?l)))))
   (= ?s ?t)))
```

Figure 2: LOGISTICS-2000 Additional Constraint

Use Cases

So far, we have only proposed an extension to PDDL, without explaining why we believe such an extension is useful. In this section we provide several use cases where our proposed extension can be useful. We remark that we have not implemented any of these ideas, we simply claim that they are feasible.

Learning and Using Domain Control Knowledge

There has been a significant body of work on learning, and using, domain control knowledge. While a full review of all the relevant literature is beyond the scope of this paper, we review some influential works in this area. First, the original STRIPS system had a macro learning component, which attempted to generalize successful plans from one problem to others (Fikes, Hart, and Nilsson 1972). This is, in fact, an example of explanation based learning (EBL) (e.g., (Mooney and Bennett 1986; Minton 1990)), where a system typically look at a single example and attempts to generalize it.

Another example is the TLPlan planner (Bacchus and Kabanza 2000), which was able to exploit manually coded domain-specific control knowledge expressed in a temporal logic. Later work tried to learn such rules automatically (Yoon, Fern, and Givan 2008). In fact, the learning track in the international planning competition (IPC), introduced in 2011 (Fern, Khardon, and Tadepalli 2011), focuses on learning domain control knowledge. In the learning track, each competitor is given access to a PDDL domain file and a random problem generator. The competitor is then given a very long time to produce a domain control knowledge (DCK) file, which the planner can then use to solve new problems in the domain, with the intent that the DCK willl help the planner improve its performance.

With the way this track is set up, the best type of guarantee that can be provided is a probably approximately correct (PAC) (Valiant 1984) style guarantee, i.e., that there is a high probability that the learner has learned something that is fairly good. However, there is no way to guarantee that the learned domain control knowledge will work, because there is no characterization of all possible instances in the domain, but only a sample of problem instances. Adopting the proposed extension to PDDL will allow learners to *prove* something about what they are learning.

For example, suppose we wanted to make the Fast Downward translator (Helmert 2009) more efficient by learning what propositions are grouped together into a finite-domain variable. We might be able to learn, for example, that the location of a truck in LOGISTICS is always a single finite domain variables. In fact, since the translator looks for invariants in a lifted way in the domain, and then generates possible mutex groups from invariants which have a single matching fact that is true in the initial state, we believe this would be relatively straightforward to so. Of course, this is only possible if we know that AT(T, L) has exactly one true proposition for each given truck T in the initial state — something which is easily described using our proposed PDDL extension.

Similar invariants can be seen in the BLOCKSWORLD domain. The same as trucks, blocks each are represented as single finite domain variables, which are generated using the invariants founded in the domain description, and the predicates in the initial state. This mutexes however, are not enough to randomly generate a "realistic" BLOCKSWORLD problem. As we mentioned before single block can not be place upon itself, and thus there are no predicates of the form ON(x, y). However, consider a problems with two blocks A and B, where block A is placed on top of block B and block B is placed on top of block A. It's easy to see that this position satisfies the condition described in the precious section, but in the same time, it's both "unrealistic" and unsolvable, given the blocks A and B have some other positions in the goal description. Even more so, this "ouroboros"[2] of a sort can be extended to a cycle of an arbitrary length, making this condition hard to detect without using reachability analysis.

Generalized Planning

A somewhat similar use case occurs in generalized planning. In generalized planning, the objective is to generate a controller which can solve all possible problems from a given planning domain. Examples of work on generalized planning include generating plans with loops and branching (Srivastava, Immerman, and Zilberstein 2011) and finite state controllers (Bonet, Palacios, and Geffner 2009; Aguas, Celorrio, and Jonsson 2016). Again, the issue is that with no formal specification of a domain, it is impossible to prove that a controller will solve all problems in a domain.

On the other hand, using our proposed PDDL extension, it is very easy (in theory) to use the following scheme. First, call a generalized planner on a given set of problems in the domain of interest. Second, verify if the resulting controller solves all possible problems in the domain. If the answer is yes, we have a controller that can solve all problems in the domain. Otherwise, generate a counter example, add it to the given set if problems, and repeat. Of course, the problem of verifying if the given controller works for all possible problems in the domain, and generating a counter example if it does not is undecidable (as we can generate a domain that corresponds to a Turing machine, and each problem corresponds to a given instance terminating). Nevertheless, efficient (incomplete) termination analyzers do exist, thus allowing us to hope this idea might work in practice on some domains of interest.

[2]A serpent eating its own tail.

Almost Automatic Random Problem Generators

When creating a new domain in PDDL, the burden of specifying which problems are legal and which are not falls to the problem generator. For example, the problem generator for BLOCKSWORLD will never generate a problem in which on(A, A) appears in the initial state. However, this knowledge is part of the problem generator's code. On top of this, the problem generator provides some distribution on the problems.

With our proposed extension, the first part of the random problem generator's job could be automated. The only implementation necessary in a random problem generator would be just the random part — the distribution.

While we believe this would be beneficial by itself, this also has the potential of enabling bootstrapping approaches (Arfaee, Zilles, and Holte 2010), where larger and larger problem instances must be generated. Of course, the issue of where the distribution comes from is still a criticial component of such an approach, which is beyond the scope of our proposed PDDL extension.

State Estimation

Finally, another use case comes from the combination of planning with real world sensing. Consider, for example, a camera looking at a BLOCKSWORLD scene. The camera, along with the image processing and object recognition software that looks at its output, will typically produce a set of real-world coordinates for the position of each block. These coordinates will typically have some error associated with them, due to sensor noise, lighting conditions, probabilistic image processing algorithms, and more.

A state estimator will look at the history of these measurements to produce the symbolic description of the current state. Without telling the state estimator that a block can only be on top of one other block, we might end up with states containing both on(A, B) and on(A, C). However, if our state estimator was able to infer mutual exclusion invariants for the domain, it could reject samples which violate these constraints, yielding more accurate state estimates.

Case Study: Discovering Domain Mutexes

As a first step to demonstrate reasoning over a domain, rather than over individual problems, we used the Fast Downward translator (Helmert 2009), in order to examine invariant candidates in PDDL domains from IPC benchmarks. As previously mentioned, the Fast Downward translator identifies lifted invariant candidates looking only at the PDDL domain. For example, the translator identifies that for a given truck T, the number of locations L for which AT(T, L) holds does not increase for any applicable action. The translator then checks whether this invariant candidate generates a set of mutexes, by checking if the number of locations each truck T is at in the initial state is 1 or less.

In this case study, we used the invariants discovered by the Fast Downward translator for each domain. For invariant, we checked whether it always led to mutexes in all instantiations of the invariant in all problems. If so, then it is likely safe to add a problem constraint derived from this invariant to the domain. However, without an explicit extension to PDDL, we can never know that this is a true lifted mutex, or whether the random problem generator just happened to only generate problems where this invariant happened to lead to mutexes.

Experimental Results

For our experiment we used the International Planning Competition benchmarks (IPC'98 – IPC'11), from which we excluded all the benchmarks that have more than one domain description file. In the relevant benchmarks we count the invariant candidates extracted by the Fast Downward translator, and the check which of those invariant were grounded to mutexes, and which were not, due to the fact that the number of initial state predicates participating in these invariants exceeded 1. The results are presented in Table 1. Note that there are no domain invariants that haven't been grounded to a mutex due to absence of the appropriate initial states facts.

Most of the invariants in these benchmarks are either always grounded, or always *overcrouded* – there are at least 2 predicates in the initial state that participate in that invariant. However, there are some invariants that are mixed. Detailed analysis shows that this happens mostly due to the fact that there is a smaller invariant that is contained in a larger one – say, in the LOGISTICS domain all the locations of a given truck T constitute an invariant, but all the locations of of all the trucks are also an invariant of the domain. The later one we hold only in the case where there is exactly one truck in the problem. Thus, mixed invariants can be seen grounded in the small problems of the domains, but getting overcrowded in the large ones.

Conclusion

In this paper, we have proposed an extension to PDDL which will allow for automated formal reasoning about domains. This extension will make no difference to the task of solving a single planning problem, with the possible exception of first validating the given problem instance. However, as we have illustrated in the previous section, such an extension will allow us to perform formal reasoning over a domain description, as well as provide a cleaner definition of what constitutes a planning domain.

While the focus of this paper has been on classical planning, our proposal becomes perhaps even more relevant in the context of non-deterministic planning. Specifically, finite state controllers are very useful with non-deterministic and partially observable planning problems, and state estimation is a must for realistic applications that involve sensing in a partially observable world.

The paper does not presume to provide the definitive, best possible, extension to PDDL. Two issue that were already mentioned are that some domains have a separate domain description for each problem instance, and that the goal can be a logical formula, not just a single conjunction. With regards to the first issue, this is usually the result of simplifying ADL (Pednault 1989) to STRIPS for the sake of planners that can not handle ADL. We argue this is not a real issue

Domain	Inv	Pure	Over	Mixed
airport-adl	8	6	0	2
assembly	0	0	0	0
barman-opt11-strips	3	3	0	0
barman-sat11-strips	3	3	0	0
blocks	3	3	0	0
depot	5	4	1	0
driverlog	2	2	0	0
elevators-opt11-strips	3	3	0	0
elevators-sat11-strips	3	3	0	0
floortile-opt11-strips	5	4	1	0
floortile-sat11-strips	5	4	1	0
freecell	7	6	1	0
grid	7	5	2	0
gripper	3	3	0	0
logistics00	1	1	0	0
logistics98	1	1	0	0
miconic-simpleadl	1	1	0	0
miconic	1	1	0	0
movie	0	0	0	0
mprime	3	3	0	0
mystery	3	3	0	0
no-mprime	2	2	0	0
no-mystery	3	3	0	0
nomystery-opt11-strips	2	2	0	0
nomystery-sat11-strips	2	2	0	0
openstacks	8	5	3	0
optical-telegraphs	7	6	1	0
parking-opt11-strips	4	3	1	0
parking-sat11-strips	4	3	1	0
pegsol-opt11-strips	2	1	1	0
pegsol-sat11-strips	2	1	1	0
philosophers	7	6	1	0
pipesworld-notankage	2	1	1	0
psr-large	0	0	0	0
psr-middle	0	0	0	0
rovers	12	6	3	3
satellite	2	1	0	1
scanalyzer-opt11-strips	0	0	0	0
scanalyzer-sat11-strips	0	0	0	0
sokoban-opt11-strips	3	2	1	0
sokoban-sat11-strips	3	2	1	0
storage	3	3	0	0
tidybot-opt11-strips	3	3	0	0
tidybot-sat11-strips	3	3	0	0
tpp	5	5	0	0
transport-opt11-strips	2	2	0	0
transport-sat11-strips	2	2	0	0
trucks	3	3	0	0
visitall-opt11-strips	1	1	0	0
visitall-sat11-strips	1	1	0	0
woodworking-opt11-strips	7	6	1	0
woodworking-sat11-strips	7	6	1	0
zenotravel	2	2	0	0

Table 1: Inv – number of invariants in the domain; Pure – number of invariants that are always grounded; Over – number of invariants that always have at least two predicates in the initial state; Mixed – number of invariants that sometimes are grounded, and sometimes have to many predicted in the initial state.

here, as reasoning over our proposed extension will require ADL-like reasoning (specifically, quantifiers). Furthermore, it is possible to perform reasoning over a domain using the complex ADL domain specification, and then planning using the simplified STRIPS version of the *given problem*.

The second issue, of complex goals, deserves further discussion. It could be possible to modify our proposed extension to PDDL to contain more general statements about the goal, such as "the goal entails X" or "the goal contains X as a subexpression in a location specified by y". We are skeptical that such statements would be of use in modeling domains of interest to the planning community, and so we do not propose them here.

Finally, despite the abovementioned issues, we believe this paper serves as a starting point for a discussion about what exactly constitutes a domain, and on what the automated planning community can contribute on top of state-of-the-art automated planners.

References

Aguas, J. S.; Celorrio, S. J.; and Jonsson, A. 2016. Generalized planning with procedural domain control knowledge. In *Proc. IJCAI 2016*, 285–293.

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2010. Bootstrap learning of heuristic functions. In *Proc. SoCS 2010*, 52–60.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *AIJ* 116(1-2):123–191.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS 2009*.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th International Planning Competition. Technical Report 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

Fern, A.; Khardon, R.; and Tadepalli, P. 2011. The first learning track of the international planning competition. *Machine Learning* 84(1-2):81–107.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and executing generalized robot plans. *AIJ* 3:251–288.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical Report R. T. 2005-08-47, Dipartimento di Elettronica per l'Automazione, Università degli Studi di Brescia.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

McDermott, D. 2000. The 1998 AI Planning Systems competition. *AI Magazine* 21(2):35–55.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *AIJ* 42(23):363–391.

Mooney, R. J., and Bennett, S. 1986. A domain independent explanation-based generalizer. In *Proc. AAAI 1986*, 551–555.

Pednault, E. P. D. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR 1989*, 324–332.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2011. A new representation and associated algorithms for generalized planning. *AIJ* 175(2):615–647.

Valiant, L. G. 1984. A theory of the learnable. *CACM* 27(11):1134–1142.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *JMLR* 9:683–718.