Dynamic Object Creation for PDDL Planning

Stefan Edelkamp[1,2]
[1]Computer Science Department,
Czech Technical University
edelkste@fel.cvut.cz
[2]School of Computer Science
Faculty of Mathematics and Physics
Charles University
edelkamp@ktiml.mff.cuni.cz

Alberto Lluch Lafuente
DTU Compute
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark
albl@dtu.dk

Ionut Moraru
School of Computer Science
University of Lincoln
Lincoln, UK
imoraru@lincoln.ac.uk

Abstract

For[1] automated planning robot behaviour in a rather unexplored world, often there are exogenous events introducing new objects triggered by the actions that were unforeseen in the initial state of the planning domain model. This suggests extending PDDL, on every of its language levels, by a feature for dynamic object creation. Syntactically, this is dealt with by adding a keyword *new* to PDDL. Semantically, the core challenge in dealing with dynamically created objects is that they lead to a growing state representation during the plan space exploration. As almost all existing automated planners are not designed to deal with varying state vectors —mainly due to relying on grounding of the lifted PDDL input in their static analysis stage— in this short paper we present a first solution to solve the dynamic object creation problem via a translation of PDDL model into a term rewriting tool, where plans can be found.

Introduction

Unforeseen things happen to all of us every day. It is widely seen as a matter of intelligence for any agent and especially us humans to be able to adapt to situations that are new.

In machine learning this requirement of intelligent behavior is referred to as a problem of generalization and has let to distinguish between training and test sets (Flach 2012).

In exploratory robotics, however, like a rover operating on mars, often we have unexpected exogenous events triggered by controlled or uncontrolled actions (Fox, Howey, and Long 2005). In terms of action planning, new objects may pop up (and disappear) during executing a plan.

By introducing the keyword *new* to the planning domain definition language PDDL (Fox and Long 2003), domain objects might be created and all according predicates and functions initialized. One apparent option is to use the *new* operator as an action effect. Quantification can then be used to initialize the predicates and functions for coping with the new object. Together with this initialization of variables, the requested PDDL language extension for dynamic object creation are considerably small.

While in software model checking and graph transformation newly created objects like processes or graph nodes/edges are common (see e.g. (Distefano, Rensink, and Katoen 2002; Artho and Visser 2019; Holzmann 2004)), current planning technology is limited in the ability to deal with dynamic objects. While syntactically rather easy, introducing them to PDDL domains will lead to varying state vectors and substantial internal changes of planners and static analyzers (Helmert 2006). At least conceptually, for explicit-state forward-chaining planning, a varying state representations can be made available.

There are extensions to PDDL that are more of syntactical nature (Nebel 1999). For example object arrays and records could be compiled away much in the spirit to what can be done with ADL types and negated preconditions, while conditional effects are *essential* and may lead to an exponential blow-up (Pednault 1989). In complexity terms, the main problem of dynamic objects, besides proper duplicate detection, is a potential infinite state space. In this paper we suggest PDDL to allow for actions that create new objects. After stating the problem and its complexity we illustrate that rewriting tools like Maude (Clavel et al. 2007) allow to do planning with dynamically created objects.

Problem Statement and Complexity

For the dynamic creation of domain objects the syntactical modifications to PDDL (Fox and Long 2003) are small and rather straight-forward. Beside ordinary action effects, we additionally allow object creation with another effect of the following syntax:

```
(new (a - <type>) (:init (<formula>*))
```

where `a` is the new object and `type` would be the new object type. As with the initial state, the closed world assumption in PDDL suggests that all predicates involving `a`, not explicitly mentioned in the formula for `:init` are false. Similarly, a deletion operator could be introduced.

[1]This paper is a shortened version of an ICAPS IPC workshop submission *Introducing Dynamic Object Creation to PDDL Planning* from April 2019, publicly accessible from `https://openreview.net/forum?id=rkxRj58y5N`. The original work also contains other options for solving dynamic object creation problems like (Numerical) Planning, (Software) Model Checking and Automated Theorem Proving. We endorse the ICAPS 2024 publication *Planning with Object Creation* by Augusto B. Corrła, Giuseppe De Giacomo, Malte Helmert, and Sasha Rubin available at `https://ojs.aaai.org/index.php/ICAPS/article/view/31466` as subsequent but independent research.

```
(define (domain blocksworld)
(:requirements :strips :typing)
(:predicates (clear ?x - block) (on-table ?x - block)
 (arm-empty) (holding ?x - block) (on ?x - block ?y - block))
(:action pickup [...])
(:action putdown [...])
(:action stack [...])
(:action pop-up
  :precondition ()
  :effect (and (new (?x - block)
      (:init (on-table ?x) (clear ?x)))))
(:action pick-new
  :precondition (and (arm-empty))
  :effect (and (new (?x - block) (:init (holding ?x)))))
(:action unstack [...])
```

Figure 1: Blocksworld domain with create-block actions.

An example that illustrates how one would use such a PDDL extension is **Blocksworld** with dynamic block creation (Slaney and Thiébaux 2001). The specification in Fig. 1 includes two actions (`pop-up` and `pick-new`) containing the *new* operator to create blocks. Planning goals could demand the existence of new blocks in between the initial ones, thus requiring plans that include actions creating such new blocks.

This simple extension of PDDL makes planning undecidable.

Proposition 1 (Complexity of Object Creation) *STRIPS planning with object creation is undecidable.*

The argument is as follows. Based on the Bylander's (1994) STRIPS encoding of a Turing machine for proving the PSPACE-completeness of STRIPS, it is easy to deduce that with newly created objects, even STRIPS planning, becomes undecidable, as we may encode the working of a Turing machine with an infinite tape and cells being created on the fly. Undecidability follows from the Halting problem.

As with the undecidability of planning with numbers this does not say anything about the fragment of benchmark problems looked at. Moreover, as in every semi-decidable setting, finding plans efficiently may still be possible.

Limits and Possibilities of Compilation

Most recent planners ground the PDDL inputs, instantiating the proposition, function, and action parameters to the objects. While there are different planners that do not necessarily ground the lifted input domain, like TLPLAN (Bacchus and Kabanza) or version of POPF (Benton, Coles, and Coles 2012), at least in competitive planning, the problem of dynamic objects has been neglected for a long time.

The problem of dynamic object creation goes back to the introduction of PDDL2.1 (Fox and Long 2003). Workarounds have been found for domains that require the creation new objects. One example is the *Settlers* domain as proposed for the International Planning Competition 2002. One critical action for constructing a cart is shown in Fig. 2. There are similar ones for constructing trains and ships. The

```
(:action build-cart
  :precondition (and (>= (available timber ?p) 1)
                     (potential ?v))
  :effect (and
     (decrease (available timber ?p) 1)
     (is-at ?v ?p) (is-cart ?v)
     (not (potential ?v))
     (assign (space-in ?v) 1)
     (forall (?r - resource)
       (and (assign (available ?r ?v) 0)))
     (increase (labour) 1)))
```

Figure 2: Action in Settlers domain building a cart.

critical aspect for these domain is that the number of potential vehicles is fixed in the problem description and only the type of the object as well as its initial load is fixed in the action effect.

The Settlers domain was designed for the numeric track as a tough resource management problem. In particular, resources can be combined to construct vehicles of various kinds. Since these vehicles are not available initially, this is an example of a problem in which new objects are created. As PDDL does not conveniently support this concept at present, it was necessary to name *potential* vehicles at the outset, which can be realized through construction. A very high degree of redundant symmetry exists between these *potential* vehicles, since it does not matter which vehicle names are actually used for the vehicles that are realized in a plan. Planners that begin by grounding all actions can be swamped by the large numbers of potential actions involving these potential vehicles, which could be realized as one of several different types of actual vehicles.

As it is certainly desirable to have planners that can deal with the new language feature for object creation feature, one would like existing planners to solve transformed benchmark domains without it.

Such a transformation to ordinary PDDL is possible along the lines of the above mentioned approaches, if objects are not deleted (as it is in the two examples above), and if one has an upper bound on all possible objects to be generated available. In general, however, finding such a super set is as difficult as the plan existence problem, so that we cannot expect a fully automated compiling scheme without the problem designer's help.

We briefly sketch some transformation details to clarify the semantics of the language extension. Though this will not be formal construction, it indicates necessary changes.

The transformation uses some special predicates as flags to govern the appropriate object handling. Most importantly we include a predicate `(is-created ?a - <name>)` in the domain description, and precondition every parameter object in every action, if it is already created.

An action with the effect `(new (a - <name>) (:init (<formula>*))` is transformed introducing an ordinary add-effect `(is-created ?a)`. Since `?a` is not yet a parameter of the action the compilation will include an extra parameter into the transformed action.

For example, consider action `create-block` that

has no parameter and the only effect is `(new (a - block) (:init (holding ?a)))`. It will be transferred to an action `create-block (?a - block)` having `(holding ?a)` in the effect list. Certainly, the translated action should require `(not (is-created ?a))` included in the precondition list. Actually it would only be necessary to include the additional predicate `not-is-created` into the precondition list and to guarantee that in each action the predicates `not-is-created` and `is-created` are complement to each other.

Planning via Conditional Term Rewriting

A term rewriting rule is a pair of terms to indicate that the left-hand side can be replaced by the right-hand side. A term rewriting system is a set of such rules.

Roughly speaking, STRIP problems can be encoded in term rewriting by encoding each operator operators $o = (P, A, D)$as a rule $P \cup D \Rightarrow A \cup (P \setminus D)$. For action planning operators that have complex preconditions that cannot be capture by pattern matching, *conditional rewriting* (Meseguer 1992) rules of the form $LHS \Rightarrow RHS$ `if` $COND$ can be used.

Indeed, conditional rewriting logic (Meseguer 1992) could be easily used as a semantic framework to define the semantics of action planning languages with the unique advantage of making such semantics *executable* (e.g. via the Maude system (Clavel et al. 2007)) thus providing interpreters and planners for free. This could be easily done for the PDDL extension with dynamic object creation under discussion in this paper. Alternatively, one can use such rewriting sytems to model lifted problem representations directly, even with dynamic object creation. An example of such a direct modelling is the Blocksworld Maude model in Figure 3.

Planning in Maude can be easily done with the reachability capabilities of the engine. For instance, the `search` command allows to specify initial and goal states and ask for a rewriting sequence (i.e. a plan) between from the initial state to one of the goal states. For example, the command

```
Maude> search empty & clear(c) & clear(b) & table(a) &
table(b) & on(c,a) =>* empty & clear(a) & table(c) &
         on(a,b) & on(b,c)
```

finds the a plan to a classical blockworlds planning problem without object creation.

An example where the planning goal requires new blocks can be seen here

```
Maude> search [1] empty & clear('c) & clear('b) & table('a)
& table('b) & on('c,'a) & next(0)
=>* empty & clear(0) & table('c) & on('a,'b) &
on('b,'c) & on(x:Nat,'a) & next(y:Nat)
```

where the goal configuration requires some block `x` to be on top of block `a`. Invoking such a command yields a plan including block creation:

```
unstack putdown pickup stack
pickup stack create pickup stack
```

```
mod BLOCKS-WORLD is
 protecting QID .
 sorts BlockId Prop State .
 subsort Qid < BlockId .
 subsort Prop < State .
 op on-table : BlockId -> Prop .
 op on : BlockId BlockId -> Prop .
 op clear : BlockId -> Prop .
 op holding : BlockId -> Prop .
 op empty : -> Prop .
 op 1 : -> State .
 op _&_ : State State -> State [assoc comm id: 1] .
 vars X Y : BlockId .
 rl [pickup] : empty & clear(X) & on-table(X) => holding(X) .
 rl [putdown] : hold(X) => empty & clear(X) & on-table(X) .
 rl [unstack] : empty & clear(X) & on(X,Y) => holding(X) & clear(Y) .
 rl [stack] : holding(X) & clear(Y) => empty & clear(X) & on(X,Y) .
 --- Natural numbers as id's as well
 pr NAT .
 subsort Nat < BlockId .
 --- New operator for the next free id
 op next : Nat -> Prop .
 --- Variable for states and naturals
 vars S : State .
 vars n : Nat .
 --- Rule for creating blocks
 rl [create] : next(n)
  => table(n) & clear(n) & next(s(n)).
 --- Rules for deletion of anonymous roofs
 rl [delete1] : table(n) & clear(n) => 1 .
 rl [delete2] : on(n,X) & clear(n) => clear(X) .
endm
```

Figure 3: Dynamic Blocksworld in Maude.

Maude as Proof-of-Concept Planner?

The Maude engine is not optimised for specific instances like planing problems for which the use of ad-hoc planners is certainly to be preferred for performance purposes. However, we advocate that Maude could be used to provide reference semantics to PDDL extensions under study such as the one we propose here. This would allow the community to try new ideas and validate their feasibility as well as to serve as a reference implementation for ad-hoc planners.

For example, for the purpose of exploring possibilities for planning with dynamic object creation one could use Maude to implement algorithms to deal with some of the problems that arise in such domains. One typical example would be the need to reuse object identifiers, essential to manage such infinite state spaces (which can be reduced to finite ones if the number of objects per state is bounded). This could be achieved by testing the effectiveness of symmetry reduction and similar techniques (Lluch-Lafuente, Meseguer, and Vandin 2012) work in Maude before transferring them to dedicated planners.

Semantic Canveats

In the examples we have seen the newly created objects are not related to other existing or simultaneously created objects. There could be cases where

this is not the case. Consider, for instance, the case of a Mars rover, where objects of interest (of type `object-of-interest`) can be detected whenever a traverse action was executed. Each object of interest has an assigned location (of type location), which is modeled with the predicate `object-at(object-of-interest, location)`. A traverse action between 2 locations to an action where both an object-of-interest and a location are created such that the created object of interest is at a newly created location could look as follows.

```
(:action traverse
  :parameters (?l1 ?l2 - location)
  :precondition (rover-at ?l1)
  : effect (and (not (rover-at ?l1)) (rover-at ?l2)
     (new (?o - object-of-interest ?l - location)
         (:init (object-at ?o ?l)))))
```

If objects are added, there must be some clever mechanism to set them in relation with other objects that have already existed. One possibility could be *conditional initializations*. Take, for instance a planning task with a predicate `(pred ?o1 ?o2 - obj)` that should hold for a newly created object `?a` of type `obj` and all existing objects `?b` of type `obj` if and only if another predicate `(cond ?o - obj)` holds for `?b`. One could use the same syntax that is used for conditional effects, i.e.,

```
(new (?a - object) (:init (forall (?b - object)
    (when (cond ?b) (pred ?a ?b)))))
```

The introduction of *new* and *delete* features will lead to states of variable size, but also to variable size of ground actions. This is a challenge on the implementation side as concepts like invariant synthesis or reachability analysis are significantly harder if the number and form of actions is unclear.

Of course there is an immediate influence of added and deleted objects on the goal (and other conditions that want to refer to the objects). A question that immediately comes to mind is what happens in an instance with:

```
(:objects o1 o2 o3 - object)
(:goal (and achieved(o1) achieved(o2) achieved(o3)))
```

where the predicate "achieved" must be true for all (initially existing) objects `o1`, `o2` and `o3`. What happens now if `o1` is deleted? The intuitive answer would be that it is no longer possible possible to reach a goal state, because the object is explicitly mentioned in the goal and it's impossible to achieve that goal once `o1` has disappeared. However, what happens if `achieved(o1)` was true in the moment that `o1` disappeared? Furthermore, what happens if the goal is given via as quantifier as

```
(:objects o1 o2 o3 - object)
(:goal (forall (?o - object (achieved(?o)))))
```

which is equivalent under the current PDDL semantics. Intuitively, we would assume now that removing o1 is not a dead end as it is still possible to achieve the goal for all existing objects since `o1` is not mentioned explicitly, but then the two goals are no longer equivalent. Similarly, what happens in the second case if another object of type `my-object-type` is created.

The sketched idea of mapping PDDL to Maude could be further followed to explore all these semantic options in a formal way, with the support of a basic tool for reachability analysis (Maude) for experimentation.

Related Work and Discussion

In *creation planning* (Fuentetaja and de la Rosa 2016) irrelevant objects are compiled to counters. The authors discuss how to deal with cases where objects are created in PDDL. They do not use a *new* operator, but specialized predicates. Later on, they discussed adding a keyword to the action description `:newobjects` in between `:precondition` and `:effect`. The motivating example is the *pizza domain*. Given a slice, an action *cuts* it into two new slices of half size. Each new slice is represented as a domain object and identified by a constant. The special predicate *notexist* represents that the corresponding slice identifier has not been used before the application of the action.

Their compilation to counters is insightful and involved using rewriting of the domain (including involved actions, initial and goal state) via a translation of the (lifted) PDDL input into one with numbers, eventually serving a compiled planning problem numeric planner to solve. The work relates to compilations of functional STRIPS (Geffner 2000).

This paper proposes an extension of PDDL that allows to dynamically create and destroy objects and a technology to actually solve such extended planning tasks.

AI and action planning are both concerned about mastering the unknown. Therefore, object creation is both an important extension to PDDL modeling, as well as fascinating topic for research and planner development. With this short paper, we propose the introduction of such features to PDDL.

We showed that conditional term rewriting is one adequate solution to the problem. Mapping PDDL to Maude input and extending it for creating new objects by means of rewriting PDDL directly in Maude is feasible, as it has been already done for many languages (including Java, C, process algebras, etc.). We have seen a first proof-of-concept, in which we played with simple, though interesting examples, to see what is gained from using Maude. New insights for PDDL or planners, and yet another planner are possible. Term rewriting can also be tool for solving additional (not only plan scheduling) problems of planning specifications like confluence checks to see which actions can be safely executed concurrently and which not.

One of Maude's advantages is the fact that the language is reflective, which enables powerful meta-programming applications and the re-use of techniques. Investigating the issue of strategies/heuristics for Maude could, on the other hand, be the contribution of the planning community to Maude. There have been some efforts towards the efficiency of rewriting in the form of competitions, but we not aware on solving reachability problems with heuristics.

Acknowledgements. We are indebted to the reviewers of a preliminary version of this paper, who suggested relevant related works, further examples and insights on our ideas.

The presented work has been supported by the Czech Science Foundation (GAČR) under the research project number 22-30043S.

References

Artho, C., and Visser, W. 2019. Java pathfinder at SV-COMP 2019 (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS*, 224–228.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artif. Intell.* 69(1-2):165–204.

Clavel, M.; Durán, F.; Eker, S.; Lincoln, P.; Martí-Oliet, N.; Meseguer, J.; and Talcott, C. L., eds. 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer.

Distefano, D.; Rensink, A.; and Katoen, J. 2002. Model checking birth and death. In Baeza-Yates, R. A.; Montanari, U.; and Santoro, N., eds., *Foundations of Information Technology in the Era of Networking and Mobile Computing, IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), August 25-30, 2002, Montréal, Québec, Canada*, volume 223 of *IFIP Conference Proceedings*, 435–447. Kluwer.

Flach, P. 2012. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. New York, NY, USA: Cambridge University Press.

Fox, M., and Long, D. 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.* 20:61–124.

Fox, M.; Howey, R.; and Long, D. 2005. Validating plans in the context of processes and exogenous events. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, 1151–1156.

Fuentetaja, R., and de la Rosa, T. 2016. Compiling irrelevant objects to counters. special case of creation planning. *AI Commun.* 29(3):435–467.

Geffner, H. 2000. Functional strips: A more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*. Boston, MA: Springer US. 187–209.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.* 26:191–246.

Holzmann, G. J. 2004. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Lluch-Lafuente, A.; Meseguer, J.; and Vandin, A. 2012. State space c-reductions of concurrent systems in rewriting logic. In Aoki, T., and Taguchi, K., eds., *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, volume 7635 of *Lecture Notes in Computer Science*, 430–446. Springer.

Meseguer, J. 1992. Conditioned rewriting logic as a united model of concurrency. *Theor. Comput. Sci.* 96(1):73–155.

Nebel, B. 1999. Compilation schemes: A theoretical tool for assessing the expressive power of planning formalisms. In *Advances in Artificial Intelligence, Annual German Conference (KI)*, 183–194.

Pednault, E. P. D. 1989. ADL: exploring the middle ground between STRIPS and the situation calculus. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, 324–332.

Slaney, J. K., and Thiébaux, S. 2001. Blocks world revisited. *Artif. Intell.* 125(1-2):119–153.