

# Extracting State Constraints from PDDL-like Planning Domains

Ulrich Scholz  
Darmstadt University of Technology  
Alexanderstraße 10, 64283 Darmstadt, Germany  
scholz@informatik.tu-darmstadt.de

May 24, 2006

## Abstract

We describe a new method for inferring state constraints from PDDL-style planning problems. Planning systems can use state constraints as valuable domain-dependent knowledge to cut the search space and to speed up planning. A large and important class of state constraints are *c*-constraints which are prevalent among the constraints inferred by current techniques. These techniques are restricted to *c*-constraints of a certain structure. The described method finds many of the *c*-constraints which are found by other methods as well as complex *c*-constraints which cannot be inferred by current techniques. The presented method takes PDDL input, is domain-independent, and does not use the initial state of a planning problem.

## 1 Introduction

State constraints, also called state axioms or action invariants, are a valuable part of domain-dependent knowledge for solving planning problems. Such a state constraint is a formula  $F$  over states, such that: *If  $F$  holds for a state  $s$ , then  $F$  holds for any state reachable from  $s$ .* For propositional, STRIPS-like planning, several authors demonstrate the usefulness of state constraints by applying them in clausal form. They extract those constraints from the planning problem [BF95, FNP<sup>+</sup>97, Rin98], from the planning domain [Sch98], or add them as domain-dependent knowledge [KS96].

Currently, the planning community moves away from STRIPS towards PDDL [MGH<sup>+</sup>98], a language whose syntax has features of first order logic. PDDL allows to describe complex planning problems in a compact way. For example, PDDL allows parameterized operators with quantification and context-dependent effects. The description of a planning domain in such a way helps to avoid unnecessary repetitions and clarifies its structure.

A corresponding representation of state constraints uses predicates with quantified variables instead of propositional clauses. In this paper we present an algorithm to infer quantified *c*-constraints. Constraints of this class result from operators which simultaneously require and delete a predicate with the same parameter vector, in which case we say they *consume* that predicate. State constraints of this kind are the most prevalent among the ones which are currently automatically inferred and used. *c*-Constraints assert that a domain object can only have exactly one property from a set of possible ones at a time, e. g. that a block is **on** at most one location. Constraints of this type are present in well known planning domains, like **blocksworld** and **logistics**.

The algorithm presented in this paper allows to find *c*-constraints of higher complexity than those inferable with other methods known from literature. The following sections introduce *c*-constraints, explain the mechanics of a domain which leads to *c*-constraints, and give an the algorithm for inferring them. The final section shows directions for further work and gives concluding remarks.

## 2 c-Constraints

The method described in this paper allows to infer c-constraints, a class of state constraints, from PDDL planning domains. In propositional planning domains, a c-constraint is a result of a set  $M$  of facts with the following properties: If exactly one fact of  $M$  is active in a state  $s$ , at most one will be active in any state reachable from  $s$ . If no fact of  $M$  is active, they stay inactive for all times. Formally, a formula  $F_M(s) = |M \cap s| \leq 1$  over states  $s$  is a c-constraint of a domain  $\mathcal{D}$ , iff  $M$  is a set of facts with the following properties:  $M$  has a size of two or more, every ground operator of  $\mathcal{D}$  which adds a fact of  $M$  consumes another element of  $M$ , and no ground operator of  $\mathcal{D}$  adds two or more facts of  $M$ . It is obvious that a c-constraint is a state constraint. A c-constraint  $F_M$  depends only on the set  $M$  so we use the term c-constraint for  $M$ , too.

It has several disadvantages to represent a c-constraint as set of facts: It is often the case that many c-constraints of a domain have the same structure. For example in `blocksworld`, blocks can only be on one location and for each block there is a corresponding c-constraint which asserts this property. The representation of c-constraints by sets of facts does not reveal that they are related. Furthermore, the number of facts in such sets can be large. If we represent such sets of facts by sets of predicates with quantified variables, these properties of `blocksworld` can be stated with a single c-constraint consisting of two predicates. To distinguish a predicate symbol from a specific predicate together with its (possibly substituted) parameter vector, we call the first predicate and the second atom.

Corresponding to parameterized operators in PDDL, we describe the set of facts which represent a c-constraint with a set of atoms whose parameters are domain objects or quantified variables. The variables of such sets can be restricted by inequality relations. Such a set of atoms  $M$  is just another representation for a set of ground facts if the variables in  $M$  are instantiated in all possible ways. For example, the `swap_n_replace` domain which is given below has a c-constraint  $\{(P A A), (P A B), (P B A), \dots, (Q B B)\}$ , if  $A$  and  $B$  are the only domain objects. We write this set as  $\{L \mid L = (P x y) \vee L = (Q y x)\}$ .

```
(define (domain swap_n_replace)
  (:action swap
    :parameters (?p1 ?p2)
    :precondition (P ?p1 ?p2)
    :effect (and (not (P ?p1 ?p2)) (Q ?p2 ?p1)))
  (:action replace
    :parameters (?p1 ?p2 ?p3)
    :precondition (Q ?p1 ?p2)
    :effect (and (not (Q ?p1 ?p2)) (P ?p3 ?p2))))
```

## 3 Iterative Construction of c-Constraints

The c-constraints of a domain are a result of the consumption of atoms in operators. We use this property of operators to find c-constraints with a fixpoint computation. The computation starts with a set  $M_0$  consisting of a single fact. Each set  $M_i$  corresponds to the hypothesis that the atoms in  $M_i$  are a subset of a c-constraint. In step  $i$  of the computation we calculate  $M_i$  by adding new atoms to the set  $M_{i-1}$  which are consumed by operators that add an atom of  $M_{i-1}$ . We repeat this process until we cannot further extend  $M_i$  and we have reached a fixpoint. In the following we describe how to extend and reject hypotheses such that all conditions of the definition of c-constraints are met and the hypothesis in a fixpoint is a c-constraint.

Let us assume that the set  $M_{i-1}$  is a subset of a c-constraint  $M$ . Every operator that adds an atom of  $M_{i-1}$  has to consume an atom of  $M$  in turn. We extend a hypothesis by choosing an atom  $L\gamma$  of  $M_{i-1}$ , where  $\gamma$  is the substitution of the parameter vector of  $L$ . For each operator  $o$  which adds  $L$  we compute the most general substitution  $\sigma$  for the parameter vector of  $o$  such that  $o\sigma$  adds at most  $L\gamma$ . We know from the definition of c-constraints that  $o\sigma$  has to consume an atom of  $M$ , so we extend the hypothesis  $M_{i-1}$  with a consumed atom of  $o\sigma$ . That way, each hypothesis is larger or of equal size than its preceding one. For the initial hypothesis we use a single atom  $L\sigma_0$  where  $L$  is a predicate and  $\sigma_0$  substitutes every parameter of  $L$  with an arbitrary but fixed domain object.

We find the most general substitution  $\sigma$  for an operator  $o$  by using the substitution  $\gamma$  for the parameter vector of the added atom. The other parameters of  $o$  are substituted with new variables. With such a substitution, all consistent ground instances of operator  $o\sigma$  add a ground instance of  $L\gamma$ . We can restrict the substitution  $\sigma$  further if some of the instantiations of  $o\sigma$  are inconsistent in an unambiguous way: If  $o\sigma$  has the same predicate in both its added and deleted effects and their parameter vectors differ in exactly one position, the substitution of these two parameters has to be distinct. The size of a hypothesis is not allowed to decrease, so we do not set these parameters unequal if both are part of substitution  $\gamma$ . Otherwise we add the corresponding inequality to substitution  $\sigma$ .

Let us give an example how this works in the `blocksworld` domain as given below. As our initial hypothesis  $M_0$ , we choose the atom `(on X Y)` where  $X$  and  $Y$  are arbitrary objects. The fact `(on X Y)` is added by operator  $o_1 = \text{move-from-table}(?p_0 ?p_1)$  with the substitution  $\sigma_1 = \{?p_0 \setminus X, ?p_1 \setminus Y\}$ . Operator  $o_1\sigma_1$  consumes `(on-table X)` which extends  $M_0$  and results in  $M_1 = \{(\text{on } X \ Y), (\text{on-table } X)\}$ . Atom `(on-table X)` is in the added effects of operator  $o_2 = \text{move-onto-table}(?p_0 ?p_1)$  with the substitution  $\sigma_2 = \{?p_0 \setminus X, ?p_1 \setminus z\}$ , where  $z$  is a new variable. Operator  $o_2\sigma_2$  consumes `(on X z)` which extends  $M_1$ . This results in hypothesis  $M_2 = \{L \mid L = (\text{on } X \ z) \vee L = (\text{on-table } X)\}$  which is a fixpoint. Therefore,  $M_2$  is a c-constraint of the `blocksworld` domain for every block  $X$ . Note that the current version of our algorithm does not use explicit inequalities between parameters of operators. In a `blocksworld` domain without inequalities it is possible to move a block onto itself.

```
(define (domain blocksworld)
  (:action move
   :parameters (?p1 ?p2 ?p3)
   :precondition (and (clear ?p1)
                     (on ?p1 ?p2) (clear ?p3))
   :effect (and (on ?p1 ?p3)
                (clear ?p2) (not (on ?p1 ?p2))
                (not (clear ?p3))))
  (:action move-onto
   :parameters (?p1 ?p2)
   :precondition (and
                 (on ?p1 ?p2) (clear ?p1))
   :effect (and (on-table ?p1)
                 (clear ?p2)
                 (not (on ?p1 ?p2))))
  (:action move-from
   :parameters (?p1 ?p2)
   :precondition (and (clear ?p1)
                     (on-table ?p1) (clear ?p2))
   :effect (and (on ?p1 ?p2)
                 (not (clear ?p2))
                 (not (on-table ?p1))))))
```

How do we reject a hypothesis which does not lead to a c-constraint? If a set of atoms  $M$  is not a c-constraint of a domain, this domain contains an operator  $o$  which either adds an atom of  $M$  but does not consume any, or  $o$  adds two or more atoms of  $M$ . The first case is covered in the following way: If we have to consider an operator which does not consume an atom and therefore violates the definition of c-constraint, we reject the current hypothesis. To cover the second cause for rejection, we construct a set  $\overline{M}_i$  for each hypothesis  $M_i$ . We know that all added effects of an operator  $o$ , besides the element of  $M_i$ , cannot be element of the hypothesis. We collect these atoms in  $\overline{M}_i$  and refuse any extension of  $M_i$  with atoms that have an instantiation which is also element of  $\overline{M}_i$ .

We use  $\overline{M}_i$  to collect further atoms which enables us to detect inconsistencies more early. A c-constraint holds for a state iff exactly one of its elements is active. Operators which consume two or more atoms of a c-constraint are not applicable in such states. If we choose a consumed atom of an operator  $o$  to extend a hypothesis  $M_i$ , we add all other consumed atoms of  $o$  to  $\overline{M}_i$ . Note that with this mechanism the algorithm does not find some c-constraints which it could find otherwise but we are not aware of any domain for which this is the case.

Let us show an example of rejecting a hypothesis. Reconsider the `blocksworld` example given above. Again we start with `(on X Y)` and our hypothesis  $M_1$  is  $\{(\text{on } X \ Y), (\text{on-table } X)\}$ . Now we choose atom `(on X Y)` again, which is also in the added effects of  $o_3 = \text{move}(?p_0 ?p_1 ?p_2)$  under the substitution  $\sigma_3 = \{?p_0 \setminus X, ?p_1 \setminus u, ?p_2 \setminus Y, u \neq Y\}$ . Note that instantiating the second and third parameter of `move` with the same object makes this operator inconsistent. We choose `(clear Y)` as extension, which results in  $M_2 = \{(\text{on } X \ Y), (\text{on-table } X), (\text{clear } Y)\}$  and  $\overline{M}_2 = \{L \mid L = (\text{clear } u) \vee L = (\text{on } X \ u), u \neq Y\}$ . We now consider `(on-table X)` and the operator `move-onto-table(X v)`. This operator consumes only `(on X v)`. This atom collides with  $\overline{M}_2$  and we reject hypothesis  $M_2$ . If we choose `(on X z)` from the consumed atoms of  $o_3\sigma_3$ , we reach the same fixpoint as above.

```

1 proc extend_M( $L\gamma, \mathcal{H}$ ) where  $\mathcal{H} = (M, \overline{M}, \mathbb{X})$ 
2 if ( $L\gamma$  conflicts with  $\overline{M}$ ) return;
3 if ( $L\gamma$  extends  $M$ )
4    $M := M$  extended by  $L\gamma$ 
5   foreach operator  $o$  do
6     foreach  $L$  in the added effects of  $o$ , where  $L^i$  is the  $i$ th occurrence of  $L$  do
7        $\sigma$  is the substitution for  $o$  such that  $L\gamma$  is added by  $L^i$  of  $o\sigma$ 
8       if ( $o\sigma$  is not consistent) continue
9        $E := (X, \overline{X})$  where
10         $X =$  consumed atoms of  $o\sigma$ 
11         $\overline{X} =$  added atoms of  $o\sigma$  without  $L^i\sigma$ 
12         $\mathbb{X} := \mathbb{X} \cup \{E\}$ 
13      od
14    od
15  if ( the set of extension tuples  $\mathbb{X}$  is empty )
16    if ( there are two or more ground facts in  $M$  )
17      accept  $M$  as c-constraint
18    return
19  choose and delete an  $E$  from  $\mathbb{X}$ , where  $E = (X, \overline{X})$ 
20  if ( $\overline{X}$  conflicts with  $M$ ) return
21   $\overline{M} := \overline{M}$  extended by all atoms in  $\overline{X}$ 
22  foreach atom  $P\delta$  in  $X$  do
23     $X' := X$  without  $P\delta$ 
24    if ( $X'$  conflicts with  $M$ ) continue
25     $\overline{M}' := \overline{M}$  extended by  $X'$ 
26    extend_M(  $P\delta, (M, \overline{M}', \mathbb{X})$  )
27  od
28 .

```

Table 1: The algorithm to infer c-constraints from PDDL-like domains. If the current hypothesis  $M$  results in a conflict, it is rejected (lines 2, 20 and 24). If the current extension  $L\gamma$  is not already included in  $M$ , the algorithm computes all possible extension tuples  $E$  and adds them to  $\mathbb{X}$  (lines 3 to 14). If there are no more extension tuples,  $M$  is a c-constraint (line 17). If not, the algorithm chooses an extension tuple and updates the set  $\overline{M}$  of atoms which cannot be part of the hypothesis (lines 19 to 21). Then the algorithm recurses by trying each element of  $X$  as new extension under the refined hypothesis (lines 22 to 27).

## 4 The Algorithm

In the previous section we explained a method for inferring c-constraints from PDDL planning domains. In this section we outline an algorithm which implements this method. The algorithm, as given in Table 1, is a recursive procedure which takes an atom and a triple  $(M, \overline{M}, \mathbb{X})$  and either accepts  $M$  as c-constraint, refines the triple and recurses, or rejects it.  $\overline{M}$  is a set of atoms which cannot be element of  $M$  and  $\mathbb{X}$  is a set of extension tuples. Such an extension tuple  $(X, \overline{X})$  consists of two sets of atoms, where the atoms in  $X$  are possible extensions of  $M$  and the atoms in  $\overline{X}$  extend  $\overline{M}$ . The algorithm tries every atom of  $X$  as possible extension. As explained in the previous section, we do not allow operators which consume two or more atoms of a c-constraint, so the other elements of  $X$  extend  $\overline{M}$ .

Conflict and extension are defined as follows. Recall that a set of atoms  $M$  is a set of facts, which results from instantiating the atoms of  $M$  in all possible ways. An atom  $P$  can extend a set  $M$  iff there is a ground instantiation of  $P$  which is not element of  $M$ . An atom  $P$  resp. an set of atoms  $X$  can conflict with  $M$  iff there is a ground instantiation of  $P$  which is element of  $M$  resp.  $X$  and  $M$  have a common fact. To extent  $M$  by  $P$ , we add  $P$  to  $M$ . If there is an atom  $P'$  in  $M$  and all ground instances of  $P'$  are also ground instances of  $P$ , we remove  $P'$  from  $M$ . A substitution is found as explained in the previous section.

The algorithm is correct. Let us assume the opposite. Then the algorithm computes a set  $M$  for a

domain  $\mathcal{D}$  and  $M$  is not a c-constraint for  $\mathcal{D}$ . Either this implies that  $M$  contains less than two facts, which is excluded in line 16, or there has to be a state  $s$  in which one fact of  $M$  is active, an operator  $o$  of  $\mathcal{D}$ , applicable in  $s$ , and the application of  $o$  results in state  $s'$  where two or more facts of  $M$  are active. There can be two reasons for this: (1)  $o$  adds two or more facts of  $M$ , and (2)  $o$  adds a fact of  $M$  without consuming any. In case (1), the algorithm chooses one added atom of  $o$  (line 22) and excludes the others (line 25). We assumed that two added effects of  $o$  are in  $M$ . Therefore  $M$  has a conflict with  $\bar{M}$  and the hypothesis is rejected in lines 2 or 20. Case (2) leads to a contradiction, too. All consumed predicates (line 10) are tried as extension (line 22) and therefore extend the hypothesis (line 4). According to the assumption, none of them becomes part of  $M$ , so no recursive call in line 26 succeeds and the algorithm does not find a state constraint. Both cases fail so the assumption is false and the algorithm is correct.

This algorithm does not find all c-constraints because it uses simplifications to cut the search space. We mentioned earlier that it does not handle operators which consume two or more atoms of a c-constraint. Another restriction is the use of atoms instead of ground facts. The presented algorithm considers the same consumed effect for all ground instantiations of an operator. Although c-constraints of this kind are the most prevalent, there are domains like the `explode` domain below where different instantiations of an operator consume different facts of a c-constraint. For this domain, the given algorithm finds only the two c-constraints  $\{L \mid L = (\text{one } x) \vee L = (\text{center})\}$  and  $\{L \mid L = (\text{two } x) \vee L = (\text{center})\}$  where all instantiations of `out` resp. `in` consume the same atom.

```
(define (domain explode)
  (:action out
    :parameters (?p1)
    :precondition (and (one ?p1) (two ?p1))
    :effect (and
      (not (one ?p1)) (not (two ?p1)) (center)))
    (:action in
      :parameters (?p1)
      :precondition (center)
      :effect (and
        (not (center)) (one ?p1) (two ?p1))))
```

The presented method infers the variables of c-constraints from the parameters which are used to define a domain in PDDL. If a PDDL domain is defined with ground operators, the algorithm will find c-constraints which consist of sets of ground facts. Ground operators do not have inequalities and cannot be instantiated in different ways, so some restrictions of the method do not apply to them. For a domain like `explode`, the algorithm infers more c-constraints for the version with ground operators than for the parameterized version. This can result in exponential runtime if the number of c-constraints is exponential in the size of the domain, like it is the case in the `explode` domain.

Let us give a preliminary complexity result for propositional domains. Procedure `extend_M` can be invoked once for every consumed effect of an operator  $o$  and the number of times  $o$  is considered is bounded by the number of facts in the added effects of  $o$ . The recursion is started with every ground fact, so this is the minimum number of invocations. In its given version, the algorithm can find a c-constraint several times, once for each initial hypothesis. With a small enhancement, a c-constraint is found at most once. Let  $\text{max}_{\text{add}}$  and  $\text{max}_{\text{cons}}$  be the maximum number of added and consumed facts of an operator, respectively, and  $\text{nb}_{\text{ops}}$  and  $\text{nb}_{\text{facts}}$  be the number of operators and facts of the domain, respectively. Then the worst case complexity of the enhanced version is  $O((\text{max}_{\text{add}} \cdot \text{max}_{\text{cons}})^{\text{nb}_{\text{ops}}} + \text{nb}_{\text{facts}})$  for propositional domains.

In its current version, the algorithm does not use some major features of PDDL like inequalities and types. For this reason we use simplified test domains. The algorithm finds the c-constraints of the simplified `blocksworld` domain as well as the c-constraint  $\{L \mid L = (\text{at } x Y) \vee L = (\text{in } x Z)\}$  for arbitrary domain objects  $Y$  and  $Z$  of the `rocket` and the `logistics` domain.

## 5 Discussion and Conclusion

We have provided a correct algorithm for inferring a large part of the class of c-constraints, an important class of state constraints. Some of these c-constraints could not be found before. They are derived from domain descriptions, without the use of the initial state of a planning problem. The algorithm uses

operators that have a predicate together with its parameter vector both in the preconditions as well in the deleted effects. The algorithm does not pose further requirements on the parameter vector of operators and predicates.

There are other systems which generate state constraints of this kind: DISCOPLAN [GSb98] covers *sv-constraints* and *implicative constraints*. The first are inferred for predicates which are always consumed and added simultaneously and whose parameters differ in exactly one place. DISCOPLAN cannot infer sv-constraints with more than one predicate, so they correspond to a subset of the constraints found by the presented method. The implicative constraints found by DISCOPLAN can include predicates which do not appear in the effects of operators. It finds some inequalities between the parameters of state constraints, e.g. that the predicate `on` of `blocksworld` is irreflexive. Such constraints are not covered by the presented method. On the other hand, DISCOPLAN cannot find all implicative constraints based on consumption, for example that in `blocksworld` all blocks can only be `on` one surface.

Another system is TIM [FL98]. It extracts from a domain the possible transitions of predicates and their parameters during the execution of a plan. These transitions are further analyzed and result in a variety of different state constraints, of which some are also found by the presented method. TIM does not find inequalities between the parameters of state constraints. Similar to DISCOPLAN, TIM uses the position of arguments and it cannot handle state constraints with predicates whose parameters are not independent. As a result, TIM does not infer the state constraint of the `swap-n-replace` domain.

This is work in progress. The current version of the algorithm can have an exponential runtime in terms of input size. Planning domains with such a behavior seem to have a regular structure and we hope to cover them as a special case. The algorithm does not cover mayor features of PDDL. It is straight forward to include types and explicit inequality relations. Our future work will consist of extending the algorithm to a system which handles and analyzes full PDDL.

## References

- [BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In C. Mellish, editor, *Proc. 13th International Joint Conference on Artificial Intelligence*, pages 1636–1642, Montréal, Canada, August 1995. Morgan Kaufmann.
- [FL98] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, December 1998.
- [FNP<sup>+</sup>97] Norman Y. Foo, Abhaya Nayak, Maurice Pagnucco, Pavleo Peppas, and Yan Zhang. Action localness, genericity and invariants in STRIPS. In M. Pollack, editor, *Proc. 14th International Joint Conference on Artificial Intelligence*, pages 549–554, Nagoya, Japan, August 1997. Morgan Kaufmann.
- [GSb98] Alfonso Gerevini and Lenhard Schubert. Inferring state constraints for domain-independent planning. In *Proc. 15th National Conference on Artificial Intelligence*, pages 905–912, Madison, USA, July 1998. AAAI Press/MIT Press.
- [KS96] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proc. 13th National Conference on Artificial Intelligence*, pages 1194–1201, Portland, USA, August 1996. AAAI Press/MIT Press.
- [MGH<sup>+</sup>98] Drew McDermott, Malik Ghallab, Adele Howe, Craig A. Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wikins. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
- [Rin98] Jussi Rintanen. A planning algorithm not based on directional search. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Proc. 6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 617–624, Trento, Italy, June 1998. Morgan Kaufmann.
- [Sch98] Ulrich Scholz. Strategien zur Domänenanalyse. In *12. Workshop Planen und Konfigurieren“(PuK ’98), Bericht tr-ri-98-193, Reihe Informatik*, pages 17–22, FB Mathematik/Informatik, Warburgerstraße 10, 33098 Paderborn, Germany, April 1998.