Using Planning Domain Features to Facilitate Knowledge Engineering*

Gerhard Wickler
Artificial Intelligence Applications Institute
University of Edinburgh
Edinburgh, Scotland

Abstract

This paper defines a number of features that can be used to characterize planning domains, namely *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features can be used to provide additional information about the operators defined in a STRIPS-like planning domain. Furthermore, the values of these features may be extracted automatically; efficient algorithms for this are described in this paper. Alternatively, where these values are specified explicitly by the domain author, the extracted values can be used to validate the consistency of the domain, thus supporting the knowledge engineering process. This approach has been evaluated using a number of planning domains, mostly drawn from the international planning competition. The results show that the features provide useful information, and can highlight problems with the manual formalization of planning domains.

Introduction

Specifying a planning domain and a planning problem in a formal description language defines a search space that can be traversed by a state-space planner to find a solution plan. It is well known that this specification process, also known as *problem formulation* [Russell and Norvig, 2003], is essential for enabling efficient problem-solving though search [Amarel, 1968].

The Planning Domain Definition Language (PDDL) [Fox and Long, 2003] has become a de-facto standard for specifying STRIPS-like planning domains and problems with various extensions. PDDL allows for the specification of some auxiliary information about a domain, such as types, but this information is optional.

Domain Features

In this paper we will formally define four domain features that can be used to assist knowledge engineers during the problem formulation process, i.e. the authoring of a planning domain which defines the state space. These features may also be exploited by a planning algorithm to speed up the search, but this possibility depends on the actual planning algorithm used and will not be evaluated in this paper. The features defined here are: *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features are not new, at least at an informal level. Their specification is either already part of PDDL or could easily be added to the language.

The values these features take for a given domain can also be computed independent of their explicit specification. A comparison of the computed features to the ones specified in the formal domain definition can then be used to validate the formalization, thus supporting the domain author in producing a consistent domain. Applying this approach to various planning domains shows that the features defined here can be used to identify certain representational problems.

Related Work

Amongst the features mentioned above, domain types have been discussed most in the planning literature. A rigorous method for problem formulation in the case of planning domains was presented in [McCluskey and Porteous, 1997]. In the second step of their methodology types are extracted from an informal description of a planning domain. Types have been used as a basic domain feature in TIM [Fox and Long, 1998]. Their approach exploits functional equivalence of objects to derive a hierarchical type structure. The difference between this approach and our algorithm will be explained in the relevant section below. This work has later been extended to infer generic types such as mobiles and resources that can be exploited to optimize plan search [Coles and Smith, 2006].

The distinction between rigid and fluent relations [Ghallab *et al.*, 2004] is common in AI planning and will be discussed only briefly. Inconsistent effects of different actions are exploited in the GraphPlan algorithm [Blum and Furst, 1995] to define the mutex relation. However, this is applied to pairs of actions (i.e. fully ground instances of operators)

*This work has been sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-09-1-3090. The University of Edinburgh and research sponsors are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

rather than operators. Reversible actions, as a domain feature, are not related to regression of goals, meaning this feature is unrelated to the direction of search (forward from the initial state or regressing backwards from the goal). The reversibility of actions (or operators) does not appear to feature much in the AI planning literature. However, in generic search problems they are a common technique used to prune search trees [Russell and Norvig, 2003].

Preprocessing of planning domains is a technique that has been used to speed up the planning process [Dawson and Siklossy, 1977]. Perhaps the most common preprocessing step is the translation of the STRIPS (function-free, first-order) representation into a propositional representation. An informal algorithm for this is described in [Ghallab *et al.*, 2004, section 2.6]. A conceptual flaw in this algorithm (highlighted by the analysis of inconsistent effects) will be briefly discussed in the conclusions of this paper.

Type Information

Many planning domains include explicit type information. In PDDL the `:typing` requirement allows the specification of typed variables in predicate and operator declarations. In problem specifications, it allows the assignment of constants or objects to types. If nothing else, typing tends to greatly increase the readability of a planning domain. However, it is not necessary for most planning algorithms to work.

In this section we will show how type information can be inferred from the operator descriptions in the planning domain definition. If the planning domain includes explicit type information the inferred types can be used to perform a consistency check, thus functioning as a knowledge engineering tool. In any case, type information can be used to simplify parts of the planning process. For example, if the planner needs to propositionalize the planning domain, type information can be used to limit the number of possible values for variables, or a ground backward searcher may use this information to similar effect.

The formalism that follows is necessary to show that the derived type system is **maximally specific** given the knowledge provided by the operators, that is, any type system that further subdivides a derived type must necessarily lead to a search space that contains type inconsistent states.

Type Consistency

The simplest kind of type system often used in planning is one in which the set of all constants C used in the planning domain and problem is divided into disjoint types T. That is, each type corresponds to a subset of all constants and each constant belongs to exactly one type. This is the kind of type system we will look at here.

Definition 1 (type partition) *A* ***type partition*** $\mathcal{P}$ *is a tuple* $\langle C, T, \tau\rangle$ *where:*

- C *is a finite set of* $n(C) \geq 1$ *constant symbols* $C = \{c_1, \ldots, c_{n(C)}\}$,
- T *is a set of* $n(T) \leq n(C)$ *types* $T = \{t_1, \ldots, t_{n(T)}\}$, *and*
- $\tau : C \rightarrow T$ *is a function defining the type of a given constant.*

A type partition divides the set of all constants that may occur in a planning problem into a set of equivalence classes. The availability of a type partition can be used to limit the space of world states that may be searched by a planner. In general, a world state in a planning domain can be any subset of the powerset of the set of ground atoms over predicates P with arguments from C.

Definition 2 (type function) *Let* $P = \{P_1, \ldots, P_{n(P)}\}$ *be a set of* $n(P)$ *predicate symbols with associated arities* $a(P_i)$ *and let* $T = \{t_1, \ldots, t_{n(T)}\}$ *be a set of types. A* ***type function*** *for predicates is a function*

$$arg_P : P \times \mathbb{N} \rightarrow T$$

which, for a given predicate symbol P_i *and argument number* $1 \leq k \leq a(P_i)$ *gives the type* $arg_P(P_i, k) \in T$ *of that argument position.*

This is the kind of type specification we find in PDDL domain definitions as part of the definition of predicates used in the domain, provided that the typing extension of PDDL is used. The type function is defined by enumerating the types for all the arguments of each predicate.

Definition 3 (type consistency) *Let* $\langle C, T, \tau\rangle$ *be a type partition. Let* $P_i \in P$ *be a predicate symbol and let* $c_1, \ldots, c_{a(P_i)} \in C$ *be constant symbols. The ground first-order atom* $P_i(c_1, \ldots, c_{a(P_i)})$ *is* ***type consistent*** *iff* $\tau(c_k) = arg_P(P_i, k)$. *A world state is* ***type consistent*** *iff all its members are type consistent.*

Thus, for a given predicate P_i there are $|C|^{a(P_i)}$ possible ground instances that may occur in world states. Clearly, the set of type consistent world states is a subset of the set of all world states. The availability of a set of types can also be used to limit the actions considered by a planner.

Definition 4 (type function) *Let* $O = \{O_1, \ldots, O_{n(O)}\}$ *be a set of* $n(O)$ *operator names with associated arities* $a(O_i)$ *and let* $T = \{t_1, \ldots, t_{n(T)}\}$ *be a set of types. A* ***type function*** *for operators is a function*

$$arg_O : O \times \mathbb{N} \rightarrow T$$

which, for a given operator symbol O_i *and argument number* $1 \leq k \leq a(O_i)$ *gives the type* $arg_O(O_i, k) \in T$ *of that argument position.*

Again, this is exactly the kind of type specification that may be provided in PDDL where the function is defined by enumeration of all the arguments with their types for each operator definition.

Definition 5 (type consistency) *Let* $\langle C, T, \tau\rangle$ *be a type partition. Let* $O_i(v_1, \ldots, v_{a(O_i)})$ *be a* STRIPS *operator defined over variables* $v_1, \ldots, v_{a(O_i)}$ *with preconditions* $precs(O_i)$ *and effects* $effects(O_i)$, *where each precondition/effect has the form* $P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ *or* $\neg P_j(v_{P_j,1}, \ldots, v_{P_j,a(P_j)})$ *for some predicate* $P_j \in P$. *The operator* O_i *is* ***type consistent*** *iff:*

- *all the operator variables* $v_1, \ldots, v_{a(O_i)}$ *are mentioned in the positive preconditions of the operator, and*
- *if* $v_k = v_{P_j,l}$, *i.e. the kth argument variable of the operator is the same as the lth argument variable of a precondition or effect, then the types must also be the same:* $arg_O(O_i, k) = arg_P(P_j, l)$.

The first condition is often required only implicitly (see [Ghallab *et al.*, 2004, chapter 4]) to avoid the complication of "lifted" search in forward search. We will use this condition shortly to show that a type consistent system is closed.

Given a type partition $\langle C, T, \tau \rangle$ and type functions arg_P and arg_O, we can define a most general state-transition system over all type consistent states as follows:

Definition 6 (state-transition system Σ^*) *Let $\langle C, T, \tau \rangle$ be a type partition. Let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of predicate symbols with associated type function arg_P and let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of type consistent operators. Then $\Sigma^* = (S^*, A^*, \gamma)$ is a (restricted) state-transition system, where:*

- *S^* is the powerset of the set of all type consistent ground atoms with predicates from P and arguments from C,*
- *A^* is the set of all (type consistent) ground instances of operators from O, and*
- *γ is the usual state transition function for* STRIPS *actions: $\gamma(s, a) = (s-$ effects$^-(a))\cup$ effects$^+(a)$ iff action a is applicable in state s*[1].

This state-transition system forms a super-system to a state-transition system defined by a planning problem containing a type consistent initial state, and a set of type consistent operator definitions, in the sense that the states of that system (the reachable states from the initial states) must be a subset of S^* and the actions must be a subset A^*. It is therefore interesting to observe that Σ^* is closed:

Proposition 1 (closed Σ^*) *Let $s \in S^*$ be a type consistent state, i.e. a type consistent set of ground atoms. Let $a \in A^*$ be a type consistent action that is applicable in s. Then the successor state $\gamma(s, a)$ is a type consistent state in S^*.*

To show that the above is true, we need to show that every atom in $\gamma(s, a)$ is type consistent. Each atom in $\gamma(s, a)$ was either in the previous state, s, in which case it was type consistent by definition, or it was added as a positive effect. Since the action is an applicable instance of a type consistent operator O_i there must be a substitution σ such that $\sigma(\mathit{precs}^+(O_i)) \subseteq s$. Furthermore, this substitution grounds every operator variable because type consistency requires all of them to occur in the positive preconditions. Given the type consistency of s, all arguments in $\sigma(\mathit{precs}^+(O_i))$ must agree with arg_P. Given the type consistency of O_i, all arguments of a must agree with arg_O, and therefore so must the effects $\sigma(\mathit{effects}(O_i))$. Hence, all positive effects are type consistent, meaning every element of $\gamma(s, a)$ must be type consistent. ■

Derived Types

The above definitions assume that there is an underlying type system that has been used to define the planning domain and problems in a consistent fashion. We shall continue to assume that such a type system exists, but it may not have been explicitly specified in the PDDL definition of the domain. We shall now define a type system that is derived from the operator descriptions in the planning domain.

Definition 7 (type name) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators. Let P be the set of all the predicate symbols used in all the operators. A* ***type name*** *is a pair $\langle N, k \rangle \in (P \cup O) \times \mathbb{N}$.*

A type name can be used to refer to a type in a derived type system. There usually are multiple names to refer to the same type. The basic idea behind the derived types is to partition the set of all type names into equivalence classes, and then assign constants used in a planning problem to different equivalence classes, thus treating each equivalence class as a type.

Definition 8 (O-type) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of* STRIPS *operators over operator variables $v_1, \ldots, v_{a(O_i)}$ with conds(O_i) = precs$(O_i)\cup$ effects(O_i) and all operator variables mentioned in the positive preconditions. Let P be the set of all the predicate symbols used in all the operators. An* ***O-type*** *is a set of type names. Two type names $\langle N_1, i_1 \rangle$ and $\langle N_2, i_2 \rangle$ are in the same O-type, denoted $\langle N_1, i_1 \rangle \equiv_O \langle N_2, i_2 \rangle$, iff one of the following holds:*

- *$N_1(v_{1,1}, \ldots, v_{1,a(N_1)})$ is an operator with precondition or effect $N_2(v_{2,1}, \ldots, v_{2,a(N_2)}) \in$conds$(N_1)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$,*
- *$N_2(v_{2,1}, \ldots, v_{2,a(N_2)})$ is an operator with precondition or effect $N_1(v_{1,1}, \ldots, v_{1,a(N_1)}) \in$conds$(N_2)$ which share a specific variable: $v_{1,i_1} = v_{2,i_2}$, or*
- *there is a type name $\langle N, j \rangle$ such that $\langle N, j \rangle \equiv_O \langle N_1, i_1 \rangle$ and $\langle N, j \rangle \equiv_O \langle N_2, i_2 \rangle$.*

Definition 9 (O-type partition) *Let (s_i, g, O) be a* STRIPS *planning problem. Let C be the set of all constants used in s_i. Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types derived from the operators in O. Then we can define the function $\tau : C \to T$ as follows:*
$\tau(c) = t_i : \forall R(c_1, \ldots, c_{a(R)}) \in s_i : (c_j = c) \Rightarrow \langle R, j \rangle \in t_i$

Note that $\tau(c)$ is not necessarily well-defined for every constant mentioned in the initial state, e.g. if a constant is used in two relations that would indicate different derived types (which rely only on the operator descriptions). In this case the O-type partition cannot be used as defined above. However, if appropriate unions of O-types are taken then this results in a new type partition for which $\tau(c)$ is defined. In the worst case this will lead to a type partition consisting of a single type. Given that this approach is always possible, we shall now assume that $\tau(c)$ is always defined.

Definition 10 *Let $T = \{t_1, \ldots, t_{n(T)}\}$ be the set of O-types for a given set of operators O and let $P = \{P_1, \ldots, P_{n(P)}\}$ be the predicates that occur on operators from O. We can easily define type functions arg_P and arg_O as follows:*

$$arg_P(P_i, k) = t_i : \langle P_i, k \rangle \in t_i \text{ and}$$
$$arg_O(O_i, k) = t_i : \langle O_i, k \rangle \in t_i$$

Proposition 2 *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau \rangle$ be the O-type partition derived from this problem. Then every state that is reachable from the initial state s_i is type consistent.*

[1]See the definition of a STRIPS operator in [Ghallab *et al.*, 2004, page 28] and the discussion of inconsistent effects below.

To show this we first show that the initial state is type consistent. Since the definition of τ is based on the argument positions in which they occur in the initial state, this follows trivially.

Next we need to show that every action that is an instance of an operator in O is type consistent. All operator variables must be mentioned in the positive preconditions according to the definition of an O-type. Furthermore, if a precondition or effect share a variable with the operator, these must have the same type since $\equiv_O$ puts them into the same equivalence class.

Finally we can show that, if action a is applicable in a type consistent state s, the resulting state $\gamma(s, a)$ must also be type consistent. Every atom must come either from s in which case it must be type consistent, or it comes from a positive effect, which, given the type consistency of a means it must also be type consistent. ■

This shows that the type system derived from the operator definitions is indeed useful as it creates a state space of type consistent states. However, the question that remains is whether it is the best or even only type system. Clearly, there may be other type systems that give us type consistent state space. The system that consists just of a single type is a trivial example. A better type system would divide the set of constants into more types though, as this reduces the size of a type consistent state space. We will now show that the above type system is maximally specific given the knowledge provided by the operators.

Theorem 1 *Let (s_i, g, O) be a* STRIPS *planning problem and let $\langle C, T, \tau\rangle$ be the O-type partition derived from this problem. If two constants c_1 and c_2 have the same type $\tau(c_1) = \tau(c_2)$ then they must have the same type in every type partition that creates a type consistent search space.*

The first step towards showing that the above holds is the insight that operators can be used to constrain types in both directions, forward and backward. If an operator variable v_i appears in a precondition and an effect, then the type of the position of the predicate in the effect must be subset of the type of the position in the precondition or the application of the operator may lead to a state that is not type consistent. Since types are defined by an equivalence relation, however, the two types must actually be the same type. Hence the type in the effect also constrains the type in the precondition.

Now, for two type names to be in the same O-type, there must be a connecting chain $\langle R_0 O_1 R_1 \ldots O_n R_n\rangle$ of alternating first order literals and operators such that R_{i-1} and R_i are conditions of O_i which share an operator variable as the j_{i-1}th and j_ith argument respectively. The variable that is shared may vary along the chain. For each step along the chain, if a constant may occur in the j_{i-1}th position in R_{i-1} it may also occur in the j_ith position in R_i. Thus, there may be two type consistent states that are connected by O_i and which contain instances of R_{i-1} and R_i. Since both states are type consistent, both instances must be type consistent, too.

Now let us assume that c_1 appears as j_0th argument in R_0 and let c_2 appears as j_nth argument in R_n. Furthermore, let us assume these exists a type partition that assigns c_1 and c_2 to different types. Since c_1 is the j_0th argument in R_0 there may be another state in which c_1 appears as j_nth argument in R_n. Thus it appears in the same position of the same predicate as c_2, which means it must have the same type to be type consistent. ■

An Efficient Algorithm

The algorithm to derive domain types t_d treats types as sets of predicate and argument-number pairs. That is $t_d \subseteq 2^{P\times\mathbb{N}}$. Each domain type t_d corresponds to exactly one type $t \in T$. The only argument taken by the algorithm is the set of operator definitions O.

```
function extract-types($O$)
  $pTypes \leftarrow \emptyset$
  $vTypes \leftarrow \emptyset$
  for every $op \in O$ do
    extract-types($op, pTypes, vTypes$)
  return $pTypes$
```

The variable $pTypes$ contains the O-types that have been discovered so far. Initially there are no O-types and the set is empty. $vTypes$ is a set of pairs of variables (used in operator definitions) and O-types, best implemented as a map and also initially empty. The procedure then analyzes each operator in the given set, thereby building up the type system incrementally.

```
function extract-types($op, pTypes, vTypes$)
  for every $p \in pre(op) \cup eff(op)$ do
    for $i = 1$ to $a(p)$ do
      $t_{pi} \leftarrow t_d \in pTypes : \langle rel(p), i\rangle \in t_d$
      $\langle v, t_v\rangle \leftarrow v_t \in vTypes : \exists t_d : v_t = \langle arg(i,p), t_d\rangle$
      if $undef(\langle v, t_v\rangle)$ do
        if $undef(t_{pi})$ do
          $t_{pi} \leftarrow \{\langle rel(p), i\rangle\}$
          $pTypes \leftarrow pTypes \cup t_{pi}$
        $vTypes \leftarrow vTypes \cup \langle arg(i,p), t_{pi}\rangle$
      else
        if $undef(t_{pi})$ do
          $t_v \leftarrow t_v \cup \{\langle rel(p), i\rangle\}$
        else
          merge-types($t_v, t_{pi}, pTypes, vTypes$)
```

The analysis of a given operator goes through every precondition and effect of the operator, looking at every argument position in turn. The next steps of the algorithm depend on whether the predicate-position combination has been used before (in which case it will appear in the $pTypes$) and whether the variable at that position has been used before (in which case it will be a key in the $vTypes$). If only one or neither have been used, the algorithm simply adds the relevant elements to the $pTypes$ and the $vTypes$. If both have been used it may be necessary to merge the respective O-types.

```
function merge-types($t_1, t_2, pTypes, vTypes$)
  if $t_1 = t_2$ do
    return
  $pTypes \leftarrow pTypes - \{t_1, t_2\}$
  $t_{new} \leftarrow t_1 \cup t_2$
  $pTypes \leftarrow pTypes \cup \{t_{new}\}$
  for every $\langle v, t_v \rangle \in vTypes$ do
    if $(t_v = t_1) \vee (t_v = t_2)$ do
      $vTypes \leftarrow vTypes - \langle v, t_v \rangle$
      $vTypes \leftarrow vTypes + \langle v, t_{new} \rangle$
```

Of course, no action is required if the type of the variable and the type for the predicate-position combination is the same. Otherwise we replace the two sets representing the (previously different) types in $pTypes$ with a new type that is the union of the two sets. Also we need to update the pairs in $vTypes$ to ensure that keys that previously had one of the now removed types as value will now get the new type as their new value.

It is easy to see that the algorithm runs in polynomial time. Furthermore, the analysis performed by the algorithm uses only the operator descriptions, and thus its run time does not depend on the problem size.

This algorithm shares the input with TIM [Fox and Long, 1998], namely the operator specifications. Both algorithms use the argument positions in which parameters occur in preconditions and effects as the basis for their analysis. TIM uses this information to construct a set of finite state machines to model transitions of objects, whereas our algorithm builds the equivalence classes directly. The result produced by TIM is a hierarchical type system that is used to derive state invariants. In contrast, the type system derived by our algorithm is flat, meaning it may be less discriminating than the structure derived by TIM. However, we could show that the types derived by our algorithm are maximally specific for given operator descriptions. In addition, a flat type system can be used to enrich the operator definitions explicitly by simply adding unary predicates as type preconditions.

Evaluation

To evaluate the algorithm we have applied it to a small number of planning domains. To avoid any bias we used only planning domains that were available from third parties, mostly from the international planning competition. Since the algorithm works on domains and the results have to be interpreted manually only a limited number of experiments was possible. Random domains are not suitable as they cannot be expected to encode an implicit type system. The algorithm has been used on random domains, but this did not result in any useful insights.

A planning domain on which the algorithm has been used is the DWR domain [Ghallab *et al.*, 2004]. In this domain types are defined explicitly, so it was possible to verify consistency with the given types. The algorithm produced the following, listing the argument positions in predicates where they are used (the $pTypes$):

```
type: [loaded-0, unloaded-0, at-0]
type: [attached-0, top-1, in-1]
type: [occupied-0, attached-1, belong-1,
  adjacent-1, adjacent-0, at-1]
type: [belong-0, holding-0, empty-0]
type: [loaded-1, holding-1, on-1, on-0,
  in-0, top-0]
```

The first type states that it is used as the first argument in the `loaded`, `unloaded` and `at` predicate. This corresponds exactly to the `robot` type in the PDDL specification of the domain. Similarly, the other types correspond to `pile`, `location`, `crane` and `container`, in this order. The main difference is that the derived types do not have intelligible names.

The other domains that were used for testing did not come with type information specified in the same way as the DWR domain. However, they all use unary predicates to add type information to the preconditions (but not every unary predicate is a type). The domains used are the following STRIPS domains from the international planning competition: `movie`, `gripper`, `logistics`, `mystery`, `mprime` and `grid`. The algorithm derives between 3 and 5 types for each of these domains which appears consistent with what the domain authors had in mind. The only domain that stands out is the first, in which each predicate has its own type. However this appears to be appropriate for this very simple domain.

Static and Fluent Relations

Another domain feature that is useful for the analysis of planning domains concerns the relations that are used in the definition of the operators. The set of predicates used here can be divided into static (or rigid) relations and fluent (or dynamic) relations, depending on whether atoms using this predicate can change their truth value from state to state.

Definition 11 (static/fluent relation) *Let $O = \{O_1, \ldots, O_{n(O)}\}$ be a set of operators and let $P = \{P_1, \ldots, P_{n(P)}\}$ be a set of all the predicate symbols that occur in these operators. A predicate $P_i \in P$ is **fluent** iff there is an operator $O_j \in O$ that has an effect that uses the predicate P_i. Otherwise the predicate is **static**.*

The algorithm for computing the sets of fluent and static predicate symbols is trivial and hence, we will not list it here.

There are at least two ways in which this information can be used in the validation of planning problems. Firstly, if the domain definition language allowed the domain author to specify whether a relation is static or fluent then this could be verified when the domain is parsed. This might highlight problems with the domain. Secondly, in a planning problem that uses additional relations these could be highlighted or simply removed from the initial state.

The computation of static and fluent relations has been tested on the same domains as the derived types. As is to be expected, nothing interesting can be learned from this experiment.

Inconsistent Effects

In a STRIPS-style operator definition the effects are specified as and add- and delete-lists consisting of a set of (function-free) first-order atoms, or a set of first-order literals where positive elements correspond to the add-list and negative elements correspond to the delete-list. Normally, the definition of an operator permits potentially inconsistent effects, i.e. a positive and a negative effect may be complementary.

Operators

Definition 12 (potential inconsistency) *Let O be a planning operator with positive effects $e^p_1, \ldots, e^p_{n(e^p)}$ and negative effects $e^n_1, \ldots, e^n_{n(e^n)}$, where each positive/negative effect is a first-order atom. O has* ***potentially inconsistent effects*** *iff O has a positive effect e^p_i and a negative effect e^n_j for which there exists a substitution σ such that $\sigma(e^p_i) = \sigma(e^n_j)$.*

It is fairly common for planning domains to define operators with potentially inconsistent effects. For example, the move operator in the DWR domain is defined as follows:

```
(:action move
  :parameters (?r ?fr ?to)
  :precondition (and (adjacent ?fr ?to)
    (at ?r ?fr) (not (occupied ?to)))
  :effect (and (at ?r ?to) (occupied ?to)
    (not (occupied ?fr)) (not (at ?r ?fr))))
```

This operator has a positive effect `(at ?r ?to)` and a negative effect `(at ?r ?fr)`. These two effects are unifiable and represent a potential inconsistency. Since this is a common feature in planning domains there is no need to raise this to the domain author. Effects that are necessarily inconsistent may be more critical.

Definition 13 (necessary inconsistency) *Let O be a planning operator with positive effects $E^p = \{e^p_1, \ldots, e^p_{n(e^p)}\}$ and negative effects $E^n = \{e^n_1, \ldots, e^n_{n(e^n)}\}$, where each positive/negative effect is a first-order atom. O has* ***necessarily inconsistent effects*** *iff O has a positive effect e^p_i and a negative effect e^n_j such that $e^p_i = e^n_j$.*

None of the domains used in the experiments above specified an operator with necessarily inconsistent effects. Given the definition of the state-transition function for STRIPS operators [Ghallab *et al.*, 2004] as

$$\gamma(s, a) = (s - E^n) \cup E^p$$

it should be clear that the negative effect e^n_j can be omitted from the operator description without changing the set of reachable states. If $e^n_j \notin s$ then its removal from s will not change s, and the addition of e^p_i ensures that $e^n_j \in \gamma(s, a)$ because $e^p_i = e^n_j$. If $e^n_j \in s$ it will be removed in $\gamma(s, a)$, but it will subsequently be re-added. Thus, the presence of the negative effect does not change the range of the state-transition function.

From a knowledge engineering perspective this means that an operator with necessarily inconsistent effects indicates a problem and should be raised to the domain author. However, this is only true for simple STRIPS operators where actions are instantaneous and thus, all effects happen simultaneously. If effects are permitted at different time points then only those that are necessarily inconsistent at the same time point must be considered a problem.

Actions

Since actions are ground instances of operators, there is no need to distinguish between necessarily and potentially inconsistent effects. All effects must be ground for actions and therefore inconsistent effects are always necessarily inconsistent. Even if necessarily inconsistent operators are not permitted in a domain, actions with inconsistent effects may still occur as instances of operators with potentially inconsistent effects.

Whether it is desirable for the planner to consider such actions depends on the other effects of the action. For example, in the DWR domain no action with inconsistent effects needs to be considered. However, if an action has side effects then it may make sense to permit such actions. For example, circling an aircraft in a holding pattern does not change the location of the aircraft, but it does reduce the fuel level. If such side effects are important actions with inconsistent effects may need to be permitted. And, of course, every action has the side effect of taking up a step in a plan.

If actions with inconsistent effects are considered by the planner, this may lead to further complications. This is because the definition of the state-transition function first subtracts negative effects from a state and then adds positive effects. For actions that have no inconsistent effects this order is irrelevant. However, if actions with inconsistent effects are permitted the result may be surprising. For example, returning to the move operator in the DWR domain, this has been defined with a positive effect `(occupied ?to)` and a negative effect `(occupied ?fr)`. Thus, the action `(move r loc loc)` will result in a state in which `(occupied loc)` holds. Now suppose the domain had been defined using the predicate `free` instead of `occupied`. In this case the result of `(move r loc loc)` would result in a state in which `(free loc)` holds. This problem occurs only with inconsistent effects.

None of the domains used in the tests above require actions with inconsistent effects and thus, they can be ignored by the planner. The following algorithm can be used to find the applicable actions (without inconsistent effects) in a given state.

```
function addApplicables($A, o, p, \sigma, s$)
  if not empty($p^+$) then
    let $p_{next} \in p$
    for every $s_p \in s$ do
      $\sigma' \leftarrow$ unify($\sigma(p_{next}), s_p$)
      if valid($\sigma'$) then
        addApplicables($A, o, p - p_{next}, \sigma', s$)
  else
    for every $p_{next} \in p^-$ do
      if falsifies($s, \sigma(p_{next})$) then return
    for every $e_p \in$ effects$^+(o)$ do
      for every $e_n \in$ effects$^-(o)$ do
        if $e_p = e_n$ then return
    $A \leftarrow A + \sigma(o)$
```

The algorithm adds all instances of operator o that are applicable in state s to the set of actions A. The parameter p represents the remaining preconditions (initially empty) and a substitution σ (also initially empty) will be built up by the algorithm. It first deals with the remaining positive preconditions and uses those to construct the substitution for all the parameters of the operators. Note that we require an operator to mention all its parameters in the positive preconditions. When the positive preconditions have been tested, the algorithm checks the negative preconditions under σ which must now be fully ground. Finally, the algorithm tests for inconsistent effects by doing a pairwise comparison between positive and negative effects. This algorithm can also be used to generate the actions for the next action layer in a planning graph. A goal regression version is slightly different as it is no longer guaranteed that all the operator parameters will be bound after the unification with a goal (and possibly static preconditions).

Reversible Actions

A common feature in many planning domains (and in many classic search problems) is that they contain actions that can be reversed by applying another action. There is usually no need to consider such actions during the search process.

Reversible Operators

The idea here is to apply the concept of reversibility to operators: an operator may be reversed by another operator (or the same operator), possibly after a suitable substitution of variables occurring as parameters in the operator definition. Note that this definition is somewhat narrow as it demands this pattern to be consistent across all instances of the two operators, i.e. it excludes the possibility of an operator sometimes being reversed by one operator, and sometimes by another, depending on the values of the parameters.

Definition 14 (reversing operators) *An action a that is applicable in a state s is* ***reversed by an action*** *a' if the state that results from applying the sequence $\langle aa' \rangle$ in s results in s, i.e. the state remains unchanged. An operator O is* ***reversed by an operator*** *O' under substitution σ' iff for every action $a = \sigma(O)$ that is an instance of O:*

- *if a is applicable in a state s then $a' = \sigma(\sigma'(O'))$ is applicable in $\gamma(s, a)$ and*
- *$\gamma(\gamma(s, a), a') = s$.*

For example, consider the `(move ?r ?l1 ?l2)` operator from the DWR domain. This can be reversed by another move operation with different parameters, as defined by the substitution $\sigma' = \{$`?l1`$\leftarrow$`?l2`,`?l2`$\leftarrow$`?l1`$\}$, i.e. `(move ?r ?l1 ?l2)` is reversed by σ'(`(move ?r ?l1 ?l2)`)=`(move ?r ?l2 ?l1)`.

While this definition captures the idea of a reversing operator, it is not very useful from a computational point of view. Another way to avoid exploring states that are the result of the application of an action followed by its reverse action is to store all states in a hash table and test whether the new state has been encountered before, an approach that is far more general than just testing for reversing actions. Computationally, it is roughly as expensive as the test suggested by the above definition. The key here is that both are state specific. A definition of reversibility that does not depend on the state in which an action is applied would be better.

From a domain author's perspective, it is often possible to specify which operators can be used to reverse another operator, as we have shown in the DWR move example above. If this information is available during search then there is no need to apply the reverse action, generate the state, and compare it to the previous state. Instead a relatively simple substitution test would suffice: $a' = \sigma(\sigma'(O'))$.

Proposition 3 *Let O_1 be an operator with positive effects E_1^p and negative effects E_1^n that is reversed by O_2 with positive effects E_2^p and negative effects E_2^n under substitution σ'. Then the two sets of positive/negative effects must cancel each other:*

$$E_1^p = \sigma'(E_2^n) \text{ and } E_1^n = \sigma'(E_2^p)$$

Suppose there is a positive effect in E_1^p that is not in $\sigma'(E_2^n)$. Now suppose an instance of O was applied in a state in which the effect in question does not already hold. The effect would then be added by the instance of O but it would not be deleted by the reversing action, and thus the original state and the state resulting from the two actions in sequence would not be the same. A similar argument holds for an effect in E_1^n that is not in $\sigma'(E_2^p)$. ■

This means we can let the domain author specify reversing operators and then use the above necessary criterion for validation. Or we could treat the above criterion as sufficient and thus exclude a portion of the search space. This may lead to an incompleteness in the search, but the domains we have used for our evaluation do not show this problem.

Unique Reversibility

In fact we have made an even stronger assumption to carry out some experiments with the domains mentioned above: we have assumed that there is at most one operator that reverses a given operator. We have then, for each domain, done a pairwise test on all the operators defined in the domain to see whether the necessary criterion holds. This resulted in discovering that the move operator can be reversed by itself with a substitution automatically derived from the operator definition, and similarly it discovered the reversibility between the take and put operators and the load and unload operators in the DWR domain.

Perhaps surprisingly, the unique reversibility was not given for all domains. The `logistics` domain contains load and unload operators for trucks and airplanes. These are specified as four distinct operators. However, in terms of their effects the two load operators and the two unload operators cannot be distinguished. The only difference lies in the preconditions where the `?truck` parameter is required to be a truck and the `?airplane` parameter is required to be an airplane.

This result can be interpreted in two ways: one could argue that the necessary condition may not be used as sufficient in this domain. Or one could argue that this domain contains redundancy that can be removed by merging the two load and unload operators, which would not change the set of reachable states in this example but means the planner has fewer actions to consider. Either way, testing for the necessary reversibility condition has highlighted this domain feature.

Conclusions

This paper has defined four planning domain features that can be used by knowledge engineers to provide information about the domain they are encoding. The formal definition of the features was used to design algorithms that can extract the actual feature values from the domain description. The algorithms are based on the domain description only, i.e. they do not require a planning problem as input. The extracted features can then be compared to the feature values specified by the domain author to validate the domain description. This approach has been evaluated using domains taken mostly from the international planning competition. The result shows that features were consistent with those available in the domains, where explicitly specified. Those features that were not specified were extracted and manually verified, to ensure they are consistent with the given set of operators.

The first feature, the type system, is a rather simple, flat division into equivalence classes. This may not be suitable for very complex planning domains, but the domains we have analyzed do not exhibit much hierarchical structure. The advantage of such a type system is that it can be easily added to the operator descriptions in the form of unary preconditions. Furthermore, we showed that the type system derived by our algorithm is the most specific type system of its kind based solely on the operator descriptions. An open question is whether this is identical to the least general generalization [Plotkin, 1969] used in machine learning. The algorithm could be refined to derive a hierarchical type system if one takes into account the directionality of the operators, but for a type system consisting of equivalence classes this is irrelevant. Also, the algorithm described in this paper should also be applicable to hierarchical task network domains, but this has not yet been implemented.

Actions with inconsistent effects are another feature we have defined. For most domains, such actions are probably not desirable. In fact, the admission of such actions leads to a different planning problem as the state spaces with or without such actions may be different for the same planning domain and problem. Also, planners that translate a STRIPS planning problem (with negative preconditions) into a propositional problem (without negative preconditions) need to be more careful if actions with inconsistent effects are permitted. The translation method described in [Ghallab *et al.*, 2004, section 2.6] does not work in this case as it introduces independent predicates for a predicate and its negations, which can become true in the same state if an action with inconsistent effects is applied. This would render the planner potentially unsound.

The final feature which defines reversible actions is somewhat different as it can only be usefully used as a necessary criterion to test whether one operator is the reverse of another. The more strict, sufficient definition does not provide any computational advantage. The difference is simply that the necessary criterion can be computed on the basis of the operator descriptions, whereas the sufficient test requires knowledge of the state in which an action is applied. The difference is quite subtle though, and may not matter in practice. The necessary criterion requires the positive and negative effects to cancel each other. However, if a state contains an atom that is also added by the first action, but then deleted by the second action, then the state will be changed. If an operator listed all the relevant atoms also as preconditions, this exception would not hold.

Implementations of the algorithms described in this paper (in Java) exist. They are currently being ported to PHP where they can be used as part of an extension to MediaWiki that allows the semi-formal specification of planning knowledge to support distributed development and sharing of procedural knowledge.

References

Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Elsevier/North-Holland, 1968.

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642. Morgan Kaufmann, 1995.

Andrew Coles and Amanda Smith. Generic types and their use in improving the quality of search heuristics. In *Proc. 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*, 2006.

Clive Dawson and Laurent Siklossy. The role of preprocessing in problem-solving systems. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 465–471. Morgan Kaufmann, 1977.

Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

Maria Fox and Derek Long. PDDL2.1 : An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*. Morgan Kaufmann, 2004.

T.L. McCluskey and J.M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95:1–65, 1997.

Gordon Plotkin. A note on inductive generalization. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2nd edition, 2003.