Problem Solving Using Classical Planners
AAAI Technical Report WS-12-12

A Planning-Based Approach for Generating Planning Problems

Raquel Fuentetaja and **Tomás de la Rosa**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
rfuentet@inf.uc3m.es, trosa@inf.uc3m.es

Abstract

Most of the research in Automated Planning relies on the evaluation of different techniques over a set of benchmarks. The generation of planning tasks for these benchmarks is done using generators coded ad-hoc. Instead, we propose an approach for generating planning problems automatically given the domain definition and some declarative semantics-related information provided by the user. The approach consists of modelling the task of generating planning problems also as a planning problem. The main contribution of this work is that the generation of planning problems is partially handled in a domain-independent way, which leads to a saving of time and effort for researchers. Additionally, the declarative input to the generator facilitates the modification of its behavior. This is a feature of interest for generating different problem distributions.

Introduction

The automated planning community pursues the development of solvers that can synthesize plans in a domain-independent way. For this reason, researchers are continuously designing new benchmarks and studying the behavior of planners with problems of different size and difficulty. Planning problems for these benchmarks are built using ad-hoc domain-dependent generators. In fact, the call for domains of the IPC (International Planning Competition) usually asks for the new domains together with their problem generators. In this paper, we argue this task can be performed in a more generic way. The most direct consequence is the saving of time and effort, since it is expected to be easier to express a declarative input than to code the generator. It will be useful for IPC organizers, researchers building planners, modelling new domains or working in learning for planning, where the training problems should be varied (Fern, Khardon, and Tadepalli 2011; Mehta, Tadepalli, and Fern 2011).

To our knowledge, the work in this paper constitutes the first attempt to build a domain-independent problem generator. Now, it is restricted to typed STRIPS domains expressed in PDDL (Planning Domain Definition Language) (Fox and Long 2003), and with predicates with a maximum arity of two. In this context, a planning domain is a tuple $(\mathcal{T}, \mathcal{P}, \mathcal{A})$, where $\mathcal{T}$ is a set of types (possibly organized as a hierarchy); $\mathcal{P}$ is a description of the domain predicates defined over types; and $\mathcal{A}$ is a set of parameterized actions. Each action $a \in \mathcal{A}$ has three parts: preconditions, $pre(a)$; add effects, $add(a)$; and delete effects, $del(a)$. For STRIPS planning all these parts are expressed conjunctions of lifted literals in predicate logic. A planning problem is a tuple $(\mathcal{C}, \mathcal{I}, \mathcal{G})$, where $\mathcal{C}$ defines the objects (constants) in the problem, together with their types; $\mathcal{I}$ is the initial state, defined as a set of grounded atoms, and $\mathcal{G}$ refers to the problem goals, expressed as a set of grounded atoms that defines a partial state. Our work is devoted to build a domain-independent generator of such problems.

Planning problems are generated correctly when the initial state is *valid*, and the problem is *solvable* (i.e. goals are reachable from the initial state). Valid states are those that represent a possible situation of the world (i.e. whose truth value could be eventually true). Given a domain in PDDL, humans are usually able to determine which states are valid for this domain. The domain actions help in the interpretation, since the execution of actions in valid states produce also valid states. However, the task of generating valid states is complex as it can require information about the meaning of predicates, that is not expressed in the domain definition.

Regarding the goals, to generate solvable problems, at least one state containing the goals has to be reachable from the initial state. Reachable states (from a given initial state) are those whose truth value can be true by executing actions from the initial state. The set of reachable states is a subset of the set of valid states.

In this paper we propose an approach to generate correct planning problems in a domain-independent fashion. We deal first with the generation of initial valid states and then with the generation of goals.

Action-based State Generation

Our technique for generating valid states is based on our observation of the human behavior while defining problems for a domain. Usually, the first task is to draw the initial scenario. Then, objects are introduced one by one. The introduction of a new object in the scenario consists of characterizing, totally or partially, the state of the object. Also, it can imply to modify the characterization of other objects already in the scenario. Consider an example in the Blocksworld do-

```
(define (generator-input blocks)

   (:objects ((block 4)))

   (:semantic-order
      (on (block) (block :before)))

   (:creation-scenario
      (handempty))

   (:predicate-constraints
      (holding (block :empty)))

   (:goal-constraints 2
      (on (block) (block))))
```

(a) Input file.

```
(:action insert_ontable
   :parameters (?x - block)
   :precondition (and (not (inserted ?x)))
   :effect (and (ontable ?x) (clear ?x) (inserted ?x)))

(:action insert_holding
   :parameters (?x - block)
   :precondition (and (not (inserted ?x)) (handempty))
   :effect (and (not (handempty)) (holding ?x)
                (inserted ?x)))

(:action insert_on_block
   :parameters (?x - block ?y - block)
   :precondition (and (not (inserted ?x)) (inserted ?y)
                      (clear ?y))
   :effect (and (inserted ?x) (not (clear ?y))
                (clear ?x) (on ?x ?y)))
```

(b) Generating domain.

```
(define (problem generator-blocks)

(:domain blocks-generator)

(:objects block-1 block-2
          block-3 block-4 - block)

(:init (handempty))

(:goal (and
         (inserted block-1)
         (inserted block-2)
         (inserted block-3)
         (inserted block-4))))
```

(c) Generating problem.

Figure 1: Example in the Blocksworld domain.

main. The initial scenario contains the table and an empty hand. Then, blocks are placed one by one. Introducing a new block supposes to put it onto another previously introduced block, on the table or in the hand.

The introduction of an object in the scenario can be viewed as an *insertion* or *creation* action, in the sense that the object does not exist before, but this action makes the object to appear in the scenario *magically*. Such creation actions can be modelled in PDDL. Figure 1 (b) shows the insertion actions for Blocksworld. We will refer to the set of insertion actions for a domain as the *generating domain*.

Knowing the generating domain and the *object distribution*, i.e. the number of objects of each type, which for our system is provided as an input, we define the *generating planning problem* $(\mathcal{C}_g, \mathcal{I}_g, \mathcal{G}_g)$, where $\mathcal{C}_g$ is generated automatically from the object distribution; $\mathcal{I}_g$ is the set of atoms in the initial scenario; and $\mathcal{G}_g$ is the set of artificial *inserted* predicates over the defined objects.[1] Figure 1 (c) shows an example of a generating problem for Blocksworld. From the planning perspective, solving this problem is trivial, just applying insertion actions randomly until all objects have been inserted (there is no need for backtracking). We will obtain a valid initial state for the original domain after removing all the *inserted* literals from the final state reached by the solution to the generating problem. This method has enough power to generate all valid states in Blocksworld and also in other domains.

Figure 2 shows an overview of our approach. For obtaining the generating domain it is necessary to use some semantics-related information provided by the user. The inputs are: the original domain and a file with PDDL-like syntax (see Figure 1 (a) for an example) containing the object distribution, and some user-provided information comprising: the semantic order, the creation scenario, and the predicate and goal constraints. The semantic order is a type of semantic information; the creation scenario represents the state of the scenario before inserting objects; finally, the predicate and goal constraints allow to restrict the predicates that can appear in the initial state and the goals, respectively. All these inputs are explained in detail in the following sections. The Generating Domain Builder generates both the generating domain and the generating problem. They are the inputs of a planner that executes actions randomly until a goal state. The cleaned final state of this planning process is the initial state for the generated problem. Then, in the most general case, goals are generated by random walks, executing the original domain actions over the generated initial state.

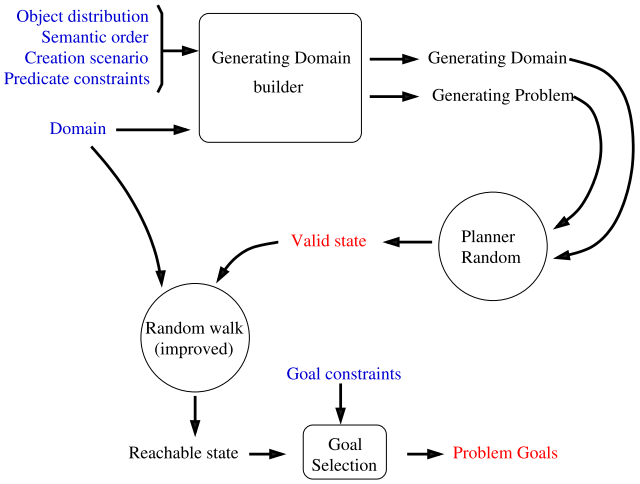

Figure 2: Overview of the domain-independent generator.

In the following sections we explain the inputs of our system. Then, we introduce the procedure to obtain the generating domain and the generating problem. Finally, we describe the generation of goals and some results.

Additional Semantic Information

Domain modelers use predicates to define characteristics and relations in PDDL. However, these predicates lack of semantic information. In this section we define two types of semantics-related information useful for building the generating domain: the *semantic order* and the *relation properties*. It could be interesting to study different kind of representations for this knowledge, though this is not the main topic of this paper.

[1] Usually, the object distribution is also an input for ad-hoc generators.

The Semantic Order

The semantic order represents the order for a predicate, if any, in which it makes sense to insert objects in a scenario. For binary predicates it is enough to mark the parameter that has to be inserted before. For instance in Blocksworld, inserting a block when placing it onto another only makes sense when the latter is already in the scenario. Thus, we will define the semantic order for the predicate *on* as *(on (block) (block :before))* (see Figure 1(a)).[2]

We will refer as *before term* to variables appearing in a position labelled as before in a literal affected by a semantic order.

Relation Properties

The relation properties are used to express the rest of the inputs of our system: the creation scenario, and the predicate and goal constraints.

Predicates represent relations or properties in the world. A binary relation $\mathcal{R} \subseteq \mathcal{A} \times \mathcal{B}$ is a subset of the cartesian product of two sets A (the domain) and B (the codomain). Similarly, a unary relation $\mathcal{R} \subseteq \mathcal{A}$ is a subset of the elements in A. For PDDL predicates, that are defined over types, A and B are sets of objects of a particular type.

Some binary and unary relations can be characterized by their properties, which provide additional information not included in the domain definition. We define the following language to define relation properties:

```
(functor (type-A (:property)*) (type-B (:property)*))
```

where *functor* is the predicate name; and *type-A* and *type-B* are the types of its arguments. We allow the user to express the following *properties*:

- **total**: $\forall a \in A$, a is related to at least one $b \in B$.
- **unique**: $a \in A$ is related to at most one $b \in B$.
- **complete**: $a \in A$, if related, is to all $b \in B$.
- **empty**: no element in A is related to $b \in B$.
- **partial**: $a \in A$ can be related to one or more $b \in B$.

These properties can be combined to characterize other types of relations: a **function** is a total and unique relation; a **partial function** is a partial and unique relation; and the **cartesian product** is a total and complete relation.

Properties have been defined from the point of view of A (properties associated to type-A). When the property is associated to B (type-B), the definition is the same, but interchanging A and B. For example, in Gripper, *(at (ball :function) (room :partial))* indicates that each ball is only in one room and rooms can have (or not) one or more balls.

When the domain and codomain of a binary relation are the same set, there are special properties. We consider the properties **irreflexive**, **symmetric** and **transitive**, with the usual semantics defined in digraph theory (Kolman, Busby, and Ross 1996). These properties are useful for characterizing predicates that represent paths.

[2]In this work we consider only binary and unary predicates. However, the approach can be generalized.

For unary relations, we consider the following properties: **total**, when the relation contains all elements in A; **unique**, when the relation contains solely one element of A; **partial**; when the relation contain some elements in A; and **empty**, when the relation is empty. For example in Blocksworld, *(holding (block :unique))* indicates that only one block can be held by the arm.

We use relation properties to impose constraints over initial states or set of goals. These constraints may be wanted by a benchmark designer for considering particular distribution of problems.

The Creation Scenario

The *creation scenario* represents the initial state of the generating problem. It contains the state of the scenario before inserting objects. The user can include in it either grounded literals or *generating relation properties*.

Generating relation properties are relation properties associated to a piece of code that generates the instances of the corresponding predicate automatically. The algorithms for generating such instances are straightforward. They comprise iteration and random selection over the corresponding objects, generating a relation holding the properties. For example, the code associated to a *function* for *(pointing (satellite :function) (direction))* iterates over all satellites, associating to each one a unique direction chosen randomly.

The creation scenario should be used to create the zero-arity predicates that define the scenario before the application of insertion actions, and to create literals (static or not) not added by any domain action. Since no action adds these facts, they can not be generated by the generating domain. Regarding static facts, one can imagine domains with very complicated configurations of the static part. Though some types of configurations can be generalized in some way (for example grids), we believe it is not possible to generate all conceivable configurations in a domain-independent way. We allow to generate static facts by defining generating relation properties in the creation scenario. However, in this case, the validity of the generated facts have to be controlled by the user. This is only possible when the interactions among these facts can be controlled by defining properties over the predicates individually. Otherwise, static facts have to be included extensionally in the creation scenario. On the other hand, the creation scenario can be used also to include other non static literals. When a predicate is defined in the creation scenario (either extensionally or by generating relation properties), all insertion actions of the generated domain adding that predicate are pruned.

Obtaining the Generating Domain

In this section we explain how to obtain the generating domain and the generating problem. The first subsection is devoted to describe the concept of *partial characterization*, related to the method we use to build insertion actions

Partial Characterizations

There are domains that allow the insertion of partially characterized objects during the generation of a valid state. For

example, in a transportation domain, vehicles can be situated at cities first, and then they can be characterized as empty. As the position of a truck and its content are independent characteristics, it is not necessary to include both at the same time. We define the dependency of predicates for a type as:

Definition 1 (Dependency of predicates for a type t)**.** Two predicates, P and P' with a common parameter x, of type t, are dependent for that type when there is a domain action that having P' as precondition, adds P and deletes P'. The relation of dependency is symmetric and transitive. [3]

Using the relation of dependency, the set of predicates over a type t can be partitioned into exclusive subsets we call *independence subsets*. Each independence subset contains only predicates that are dependent among them, but independent of the predicates in the rest of the independence subsets. For instance, in the Blocksworld domain, the only independence subset is $\{holding, clear, ontable, on\}$, since there are not independent predicates over objects of type block (i.e. when inserting a block, it should be specified completely). In Satellite, we get the following independence subsets for the type satellite: $\{\{pointing\}, \{power\_avail\}\}$, since both predicates are relative to independent characteristics of satellites. We denote the set of independence subsets for type t as $I(t)$.

Given a state s, the sub-state of an object o is the set of atoms in s referred to o. A sub-state is valid when it is a subset of a valid state.

Definition 2. A *partial characterization* of an object is the subset of atoms in the object sub-state whose predicates are in the same independence subset.

Thus, the sub-state of an object is composed by as many partial characterizations as different independence subsets for the object type (note a partial characterization can be empty). Besides, a partial characterization is valid when it is a subset of a valid sub-state. A valid partial characterization for objects of type t can be represented by the predicates it contains together with the argument position of the object. Given the independence subset i of type t, $I_i(t)$, we denote the set of valid partial characterizations for objects of type t wrt. $I_i(t)$ as $C^i(t)$. The set of predicates for each valid partial characterization, $C^i_j(t)$, is an element of the power set of $I_i(t)$: $C^i_j(t) \in 2^{I_i(t)}$. For example, in Blocksworld the set $\{ontable\_1, clear\_1\}$ is a valid partial characterization of blocks for the only independence set in this domain.

To maintain validity, the actions in the original domain modify some valid partial characterizations of the affected objects, ensuring the resulting partial characterizations are *valid*. We take advantage of this feature for building the generating domain. If we focus in just one object affected by an action, its partial characterizations before and after applying the action are both valid. The modifications (add and dels) the action performs over the rest of objects can be considered as *side effects* of the changes in that object for ensuring the validity of the generated state. For the sake of clarity, we will assume each action in the domain only modifies a partial characterization wrt. an independence subset.

[3]For the sake of clarity and without lost of generality we consider only domains with primitive types. Domains with type hierarchies can be flattened easily.

Extracting valid partial characterizations

We use the domain actions to determine a set of valid partial characterizations for each type in the domain. Our goal is not to detect all valid partial characterizations, but only those necessary for correctly inserting new objects in the scenario, assuming the provided semantic orders are correct.

Let $a(x_1, \ldots, x_n)$ be an action with parameters $x_1, \ldots, x_n$. Then, given a parameter x_k of a, the literals in a can be partitioned into two exclusive subsets: $L(a, x_k)$, the set of literals in a containing the term x_k; and $N(a, x_k)$, the set of literals in a without the term x_k. At the same time, considering the action parts, these subsets can be partitioning also into the following exclusive subsets:

- *Maintained precs* (man): literals $l \in pre(a) \setminus del(a)$.
- *Deleted precs* ($delp$): literals $l \in pre(a) \cap del(a)$
- *Extra-deletes* ($edel$): literals $l \in del(a) \setminus pre(a)$.
- *Added* (add): literals $l \in add(a)$.

Considering these partitions wrt. a parameter x_k, an action follows the scheme showed in Figure 3, where the action part is represented with sub-indexes. For example, $L_{man}(a, x_k)$ represents the set of maintained preconditions containing the parameter x_k.

```
Scheme action a(x_k)
Pre:  L_man(a,x_k), L_delp(a,x_k), N_man(a,x_k), N_delp(a,x_k)
Add:  L_add(a,x_k), N_add(a,x_k)
Del:  L_delp(a,x_k), L_edel(a,x_k), N_delp(a,x_k), N_edel(a,x_k)
```

Figure 3: Scheme of the action a for the parameter x_k.

Let $C^i_{pre}(t)$ and $C^i_{post}(t)$ be two valid partial characterizations for objects of type t wrt. the independence set $I_i(t)$. Assume these are the characterizations before and after applying an action. The action definition may or may not specify such partial characterizations completely. To make this distinction we denote as $\hat{C}^i_{pre/post}(x, t, a)$ the partial characterizations we can deduce from an action a for an object of type t represented in the action parameter x. Then, the following sets define a **subset** of $C^i_{pre}(t)$ and $C^i_{post}(t)$ respectively:

- $\hat{C}^i_{pre}(x, t, a)$, containing the predicates in $L^i_{delp}(a, x) \cup L^i_{man}(a, x)$, i.e. the set of literals over x in $pre(a)$ wrt. $I_i(t)$. $\hat{C}^i_{pre}(x, t, a) \subseteq C^i_{pre}(t)$.
- $\hat{C}^i_{post}(x, t, a)$, containing the predicates in $L^i_{add}(a, x) \cup L^i_{man}(a, x)$, i.e. the set of added facts plus maintained preconditions over x wrt. $I_i(t)$. $\hat{C}^i_{post}(x, t, a) \subseteq C^i_{post}(t)$.

In some cases $\hat{C}^i_{pre}(x, t, a)$ and/or $\hat{C}^i_{post}(x, t, a)$ specify completely a partial characterization of x wrt. $I_i(t)$, i.e. $\hat{C}^i_{pre}(x, t, a) = C^i_{pre}(t)$ and/or $\hat{C}^i_{post}(x, t, a) = C^i_{post}(t)$. In this paper we identify the situations in which the latter holds,

to extract a set of valid partial characterizations for objects of type t.

Definition 3. $\hat{C}^i_{post}(x,t,a)$ is *maximal* when there are no other actions a' with parameters x' of type t, for which $\hat{C}^i_{post}(x,t,a) \subset \hat{C}^i_{post}(x',t,a')$ or $\hat{C}^i_{post}(x,t,a) \subset \hat{C}^i_{pre}(x',t,a')$.

Given an action a and a parameter x that is not a before term neither in the literals of $pre(a)$ nor in literals of $add(a)$, such that $L^i_{add}(a,x) \neq \emptyset$, $\hat{C}^i_{post}(x,t,a)$ specifies completely a valid partial characterization for type t (i.e. $\hat{C}^i_{post}(x,t,a) = C^i_{post}(t)$). The only exception appears when $C^i_{pre}(t)$ and $C^i_{post}(t)$ share literals over x not considered in a, which means there are maintained literals over x wrt. the independence set $I_i(t)$ not appearing in $pre(a)$. In such a case, maximal $\hat{C}^i_{post}(x,t,a)$ are complete, and therefore valid. We compute first all maximal partial characterizations for all actions and their parameters that are not before terms. Thus, we obtain the following set of valid partial characterizations for objects of type t:

$$\hat{C}^i(t) = \bigcup_{x,a \in \mathcal{A}} \{\hat{C}^{i*}_{post}(x,t,a)\} \quad (1)$$

where x is a parameter of a with type t and $\hat{C}^{i*}_{post}(x,t,a)$ is $\hat{C}^i_{post}(x,t,a)$ for maximal characterizations and $\emptyset$ for non-maximal characterizations.

Then, for independence sets $I_i(t)$ for which $\hat{C}^i(t)$ is empty, since in all actions with parameters of type t, they are before terms in literals of $pre(a)$ or $add(a)$, we generate maximal partial characterizations in the same way, but considering parameters x that are before terms in a, such that $L^i_{add}(a,x) \neq \emptyset$.

In Blocksworld, we obtain:

$$\hat{C}(block) = \{\{ontable\_1, clear\_1\}, \{on\_1, clear\_1\}, \{holding\_1\}\}$$

Generation of insertion actions

We compute insertion actions for inserting the partial characterizations in $\hat{C}^i(t)$ for each type t. These actions consider the side effects of the characterizations they add.

Definition 4. The **side effects** for a parameter x_k of the action a are the added, maintained and deleted literals not containing x_k, but related with literals in $L_{add}(a,x_k)$ or $L_{man}(a,x_k)$, directly or through other literals not deleted by a.[4] We denote side effects using the symbol $^+$.

To build the insertion actions we select the actions a with parameters x_k that generated the maximal partial characterizations defined by Equation 1 for parameters that are not before terms. From each action a an insertion action, a_g, is generated using the scheme in Figure 4. The insertion action simulates that the partial characterization of objects with the type of x_k it generates has not been included in the scenario yet. When executed, a_g inserts such partial characterization and their side effects.

Side effects due to added and maintained literals, $N^+_{add}(a,x_k)$ and $N^+_{man}(a,x_k)$, have to exist after the insertion of the objects represented by x_k. Deleted literals, $N^+_{delp}(a,x_k)$ and $N^+_{edel}(a,x_k)$, reflect conditions that have to be removed after the insertion, where $N^+_{delp}(a,x_k)$ are also needed in the scenario before it occurs.

In the preconditions of a_g we have to include also the maintained literals over x_k not belonging to the added partial characterization, $L_{man}(a,x_k) \setminus L^i_{man}(a,x_k)$. They belong to other partial characterizations. In addition, we include $(not\,(inserted\_label\_i\ x_k))$, where $label\_i$ represents the label corresponding to the independence subset of the added characterization. Finally, in the add effects we include the corresponding $(inserted\_label\_i\ x_k)$. The parameters of an insertion action are those variables in its preconditions and/or effects.

Scheme for the insertion action $a_g(x_k)$
Pre:
$L_{man}(a,x_k) \setminus L^i_{man}(a,x_k)$
$N^+_{man}(a,x_k)$, $N^+_{delp}(a,x_k)$
$(not\,(inserted\_label\_i\ x_k))$
Add:
$L^i_{add}(a,x_k)$, $L^i_{man}(a,x_k)$, $N^+_{add}(a,x_k)$
$(inserted\_label\_i\ x_k)$
Del:
$N^+_{delp}(a,x_k)$, $N^+_{edel}(a,x_k)$

Figure 4: Scheme of the insertion action for a non-before parameter x_k when $\hat{C}^i_{post}(x_k,t,a) \in \hat{C}^i(t)$ (is maximal) .

Then, we take the actions a that generated the maximal partial characterizations for parameters x_k that are before terms. We remove from a all literals containing the *not before term* of the semantic order, since in order to insert x_k we simulate that the object referring to that term is not inserted. Thus, we obtain a new action a_r, which can be viewed as an abstraction of the original action a. Then, an insertion action is created from a_r following the scheme in Figure 4.

Additional insertion actions could be generated for creating literals not added by any action, considering maximal partial characterizations obtained from $\hat{C}^i_{pre}(x,t,a)$. However, in this case, side effects can not be controlled.

The explained procedure is finite, since the number of parameters of each action is finite. Also, the generated actions only insert valid partial characterizations of objects considering its side effects. Therefore, they implement the same notion of validity defined by the original domain. For this, the predicates with arity higher than zero should not be opaque, in the sense all relevant objects have to be represented actually as objects. Otherwise, relations hidden by the representation can not be detected. The effect of this representation change is minimal as it does not increase the state space size. For example, the unary predicate *(at-robby location)* in Gripper should be represented as the binary predicate *(at robot location)*. If in Blocksworld the hand

[4] Related literals are computed using the transitive closure. We consider that zero-arity predicates are related to all literals.

were modelled with *(empty hand)*, we would obtain an insertion action for introducing an empty hand in the scenario, and, it would not be necessary to include *(handempty)*.

Once we have the generating domain, the generating problem is built containing as many objects of each primitive type as indicated by the object distribution. These objects are automatically generated appending numbers to the corresponding type, getting constants in the form *type-i*. The initial state contains the *creation scenario*, that in some cases is empty. Finally, the goals contain literals of type $(inserted\_label\_p\,y))$ for which there exists an insertion action adding them. Figure 1 (c) shows an example.

Generation of Goals

The user can express, by input parameters, some constraints and preferences about the goals to be generated. Specifically, the user can indicate: (1) the predicates that can appear as goals, and (2) the desired total number of goals or, alternatively, the maximum number of goals for each of the allowed predicates. This allows the user to generate goals with a particular proportion of predicates.

We have three cases for the generation of goals: (1) based on generating relation properties, for domains with connected state spaces in which interactions among goals can be controlled by defining relation properties; (2) based on a new valid state, for domains with connected state spaces but where the interactions among goals can not be controlled with relation properties; and (3) based on a reachable valid state, for domains with state spaces which may be non-connected. In (2), to ensure reachability of goals, it is enough to generate a valid state and select some goals from it. This guarantees generated goals are not mutex. However, there are also domains where the generation of non-mutex goals can be controlled by means of relation properties, i.e., case (1). In such domains, is not even necessary to have a valid state for selecting goals. They can be generated by selecting a particular number of literals holding a relation property. For (3), we generate a reachable state by the application of the original domain actions from the initial state. The most simple way of doing this is by random walks. However, random walks (without restarts) may not generate states far from the initial one, specially in domains with reversible actions. For this reason, we use an RRT (Rapidly-exploring Random Tree) approach (LaValle, Kuffner, and Jr. 1999), that produces a more uniform coverage. For RRT, the initial and sampled states are generated using the automatic generator; and distances between states are computed using the relaxed plan heuristic (Hoffmann and Nebel 2001).

Results

We have defined input files for 8 domains (e.g. Figure 1 (a) for Blocksworld). Using these automatic generators, we have generated problems with several object distributions, verifying that they were correct and solvable. These domains are: Blocksworld, Depots, Parking, Logistics, Driverlog, Gripper, Ferry, and Satellite. For the last three domains problems can be generated completely also by defining *relation properties* in the creation scenario. Although there exist ad-hoc generators for these domains, in the near future, we plan to make our generators publicly available, so researchers can have different generator behaviors just by changing input parameters or specified constraints.

Related Work

Our work is related to the automatic generation of mutex invariants. Particularly, with the techniques that synthesize invariants using only information from the domain definition (Helmert 2009; Edelkamp and Helmert 1999; Scholz 2000; Gerevini and Schubert 1998). In fact, if the complete set of invariants for valid states is known, it could be defined a procedure for generating valid states holding all invariants. However, to our knowledge, none of the previous techniques generate a complete set of invariants for valid states. For example, we do not know any method able of detecting and expressing that in Blocksworld a state with $on(a,b), on(b,c), on(c,a)$ is not valid. An alternative approach is to allow the user to define invariants (Haslum and Scholz 2003), though we think it will be more difficult for the user to specify or to complete invariants than to define the inputs of our system. On the other hand, we have the intuition that the definition of relation properties could be useful to improve current techniques for synthesizing invariants, since they provide some of the information that can be found by analyzing valid problems.

Regarding the kind of problems we are able to generate, our generator is not balanced. For instance, in Blocksworld, not all valid states have the same probability of being generated. Specifically, for Blocksworld there is a balanced ad-hoc generator (Slaney and Thiébaux 2001).

Conclusions and Future Work

We have introduced an approach for generating planning problems automatically given the domain definition and some information expressed declaratively. To our knowledge there is no other previous work dealing directly with this problem. We believe the automatic generator is more flexible than ad-hoc ones since by changing the inputs, users are able to generate different kinds of problem distributions without changing the generator.

Main limitations are: (1) for domains with interacting static predicates, all literal instances should be included in the creation scenario; and (2) relation properties allow to apply quantifiers over the set of objects of a type. Now, we can not express predicate constraints implying more than one predicate. However, they can be extended to deal with bounded quantifiers (Bacchus and Kabanza 2000), which would allow the definition of constraints over more generic sets, as for example the objects holding another relation. In the future, we plan to continue our work in this direction.

Acknowledgements

This work has been partially supported by the Spanish MCINN project TIN2011-27652-C03-02.

References

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

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the European Conference on Planning (ECP)*.

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

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 61–124.

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th American Association for the Advancement of Artificial Intelligence Conference (AAAI)*.

Haslum, P., and Scholz, U. 2003. Domain knowledge in planning: Representation and use. In *Workshop on PDDL. International Conference on Automated Planning and Scheduling (ICAPS)*.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence (AI)* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Kolman, B.; Busby, R.; and Ross, S. 1996. *Discrete mathematical structures*. Prentice Hall.

LaValle, S. M.; Kuffner, J.; and Jr. 1999. Randomized kinodynamic planning. In *IEEE International Conference in Robotics and Automation*, 473–479.

Mehta, N.; Tadepalli, P.; and Fern, A. 2011. Autonomous learning of action models for planning. In *Advances in Neural Information Processing Systems*.

Scholz, U. 2000. Extracting state constraints from PDDL-like planning domains. In *Proceedings of the Conference on Artificial Intelligence Planning Systems (AIPS)*.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence (AI)* 125:119–153.