Importing Ontological Information into Planning Domain Models

Lee McCluskey and Stephen Cresswell
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
lee,s.n.cresswell@hud.ac.uk

Abstract

We investigate an approach to alleviating the problems of knowledge engineering for AI Planning by importing object structures and object behaviours from shared knowledge structures. In this paper we describe our first steps at devising a method to inport knowledge from an application ontology into a form usable within a planning domain model. We evaluate an implemented tool called OWL2OCL which assists in the translation of ontological information into a form usable by a planner.

Introduction

In the field of Knowledge Acquisition much attention has been given to the idea of re-using and sharing bodies of both general and application-specific knowledge in the form of ontologies. The benefits of this within fields such as eScience and Knowledge Management have led to the rapid development of ontologies in both science and commerce. Ontologies are (formal) vocabularies that relate terms together in the form of a precise specification. These can range from UML diagrams, to concept hierarchies, through to axioms in Description Logic (DL). The majority of published, formal, ontologies appear to be represented in a DL variant, although some are represented in First Order Logic using a standard syntax. There are several tools available for building ontologies such as Protege-2000 (Gennari *et al.* 2003). Superficially these are similar to GUIs such as GIPO (McCluskey, Liu, & Simpson 2003) which are used to acquire information about objects in AI Planning. However, the dominant focus of ontological engineering is to capture static knowledge whereas planning is more concerned with dynamic knowledge.

Our work in Knowledge Engineering in AI Planning is aimed at alleviating the task of deploying a planning engine in a particular application, and making planning technology in general more accessible. Experience in engineering planning applications indicate a range of problem areas related to the domain modelling phase. The initial choices of how to model the application in terms of relational predicates, classes and states is a major task. The problem of re-inventing models for every application is acute: given there are many existing domain models, how can one re-use previously captured planning domain knowledge?

In the area of AI Planning, there have been rapid development of general planning engines which input PDDL domain models. The problem with these planners is that for each application the user has to assemble an adequate knowledge base in the form peculiar to this type of planner. This requires a knowledge engineering task of translating and acquiring knowledge in an application into the input form of PDDL. One solution that we are exploring is to use an existing ontology if a suitable one exists. Rather than a knowledge engineer or expert crafting the domain model, they identify the relevant ontologies to be used in the application, and a translator assembles the knowledge in a planner-friendly representation. In other words, the translator would induce domain model structure and dynamics in an operational form suitable for input to a planning engine.

We speculate that the future of planning techniques is bound up with the semantic web: progress in the use of ontological information could also enhance the ability of a web agent to automatically assemble adequate knowledge in order to solve its own planning problems. Given an application ontology and a high level goal description, an agent will have to generate a plan to achieve this goal. In current technological terms, this amounts to generating a planning domain model, acquiring, tuning and executing a suitable planning engine.

In this paper we devise a process to import ontologies in a form that they can be used as the basis of a planning domain model. We evaluate its application to both contrived and previously published ontologies, and summarise the lessons learned from the experiment.

The Input Language

Developments in the www consortium have led to the development of a standard web ontology language called OWL (Patel-Schneider, Hayes, & Horrocks 2004). There are three versions of OWL: Lite, DL and Full. For our input language we have chosen the abstract syntax of OWL DL, which can be output from Protege-2000. As is common with ontologies in description logic, the presentation consists of a set of restricted first order axioms, with variables implicit. This has an advantage over graphical input - after axioms are changed the best hierarchy can be automatically derived using a subsumption reasoning tool. This avoids the need for

the user to try to engineer and re-engineer the application to fit into neat hierarchies.

Examples of OWL ontologies are shown in Appendix A and B. As a description logic, OWL is designed for capturing the class hierarchy of objects (classes are like unary predicates in FOL). Statements in OWL are about concepts/classes, which are defined extensionally via the objects they contain. Only binary predicates are allowed, and these are used to relate classes with other classes and with data type values.

The Target Language

According to the OWL literature (Patel-Schneider, Hayes, & Horrocks 2004), a class defines a group of individuals that belong together because they share some properties. We specialise this in AI planning - objects should share the same class if they share the same *dynamic behaviour*, ie they are related to each other in a more concrete way:

- the set of properties and relations which they can have is common.
- they share a common set of states (where a state is a collection of properties/relations that hold to be true)
- they share the same state-change behaviour, that is the same transitions between states.

The definition above is fundamental to an OCL (Liu & McCluskey 2000) domain model, and we will use OCL as the target language. In OCL, objects are grouped into $sorts$ if they share the same behaviour, and each part of a world state that describes one particular object is called a substate. A characterisation of the possible substates of an object in a sort is called a substate class description. OCL is the base language of the GIPO planning environment (McCluskey, Liu, & Simpson 2003). This contains a wide range of acquisition and validation tools for manipulating planning domain models. Once generated, the translated model can be fed into GIPO where it can be refined, and output as PDDL if necessary (see figure 1).

Ontology Translation: OWL2OCL

Our task is to import a pre-existing ontology in order to use it as the basis of a planning domain model as shown in figure 1. This is similar to the general problem of "Contextualising Ontologies" as discussed by Bouquet et al (Bouquet *et al.* 2004): changing the 'global' meaning of an ontology to a local, subjective view of a domain. Although the object-structure of OWL and OCL are superficially linked, there are many problems to be overcome. Two particular problems are as follows:

– OWL ontologies are composed under the assumption of open-world reasoning, whereas planning domain models tend to assume closed-world reasoning. As a simple example, two individuals in an OWL model are not necessarily distinct unless they can be proved so. In the planning world, we assume they are distinct unless it can be proved they are the same.

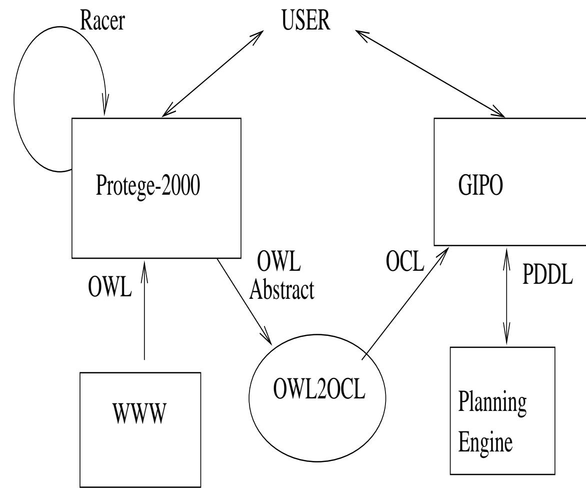

Figure 1: OWL2OCL in context

– there is no guarentee that an imported ontology is going to be complete or adequate for the planning task. In particular, ontologies written in description logic are typically *static* - they include no explicit information on the dynamics of the world.

Our initial solution is therefore to produce a *heuristic* transformation from an ontology language (OWL) to a planning domain modelling language (OCL) and let the user further validate and refine the model using the GIPO environment. As a starting point, this transformation should be able to re-create faithfully the static information of a planning domain model which has been hand-translated into an ontology. The example ontologies in Appendix A and B below are examples of this as they are encodings of familiar planning worlds.

Our overall method in this initial work assumes that a planning domain model is to be generated where all objects belong a unique distinct sort. We will map the (not necessarily disjoint) description logic classes of the ontology into a set of disjoint classes which will be primitive sorts in the planning domain. Each class will have its set of attributes generated.

The Translation Algorithm

The translation and subsequent domain acquisition process is in three stages. OWL2OCL is aimed at implementing the first two of these steps:

- Step 1: Build up a disjoint set of OCL object sorts such that for each sort, if an object belongs to it then the set of properties it can have are the same as any other object in that sort. Build up a set of predicates describing and relating the sorts.
- Step 2: Induce the individual state descriptions that objects of each sort can inhabit.
- Step 3: Build up a set of operator schema and refine the model, using GIPO. One effective way to complete the

model with operator schema would be used to induce operator descriptions from training sequences as explained in (McCluskey, Richardson, & Simpson 2002).

As we may have a complex ontology, how do we choose at what level to group the OWL objects into OCL sorts? If we choose only the leaf classes of the ontology, we may over-discriminate. Instead, we choose to base the grouping on the domain and range declarations of properties. Objects which may take the same set of properties are grouped in the same sort. For example, if we have a property which is declared as:

```
ObjectProperty(
  drives domain(person) range(vehicle))
```

then we assume that this relation can exist between any `person` and any `vehicle`, so these classes will form useful groupings for the purpose of collecting substates. In more detail, Step 1 and Step 2 are as follows:

1. Collect the set C' of OWL classes which form either the domain or range for some property.
2. Collect the set C'' of intersections of the classes in C' which are inhabited by at least one individual. In this way we create disjoint classes of objects each of which can have properties from a well-defined set - call this set S_p for each OCL class S.
3. For each class induce the substates class description, that is a characterisation of each of the states that an object of a class may inhabit, as follows. For each class $S \in C''$:
   (a) Collect the 'value' information for each individual (property) from O. That is collect all the instances of properties in S_p. We assume this an instance of a unique substate (that is all the information - the truth values of properties - is given).
   (b) For each object, generalise its set of property values to create a new class description. Form a set of new class descriptions (without duplicate equivalent classes).

What will be formed is a partial domain model which contains static and some (heuristic) dynamic knowledge.

Worked Examples

To give an idea of the transformation, we show two worked examples, using ontologies that have been written in OWL to reflect the original domain model.

(i) Appendix A: the Rocket World. The input OWL abstract syntax is shown together with our auto-generated domain model. For clarity, the namespace information has been removed.

Step 1: by examination of the domains and ranges of the declared properties, we establish that the following classes are relevant: {Cargo, Location, Rocket, Level}. All of the classes are occupied with individuals and no individual occupies more than one of the classes. Hence the set of occupied combinations of classes is identical.

Step 2: The substates occupied within each sort are induced. In figure 2, each cargo object has either an 'at' or 'in' property, hence we generate the substates of the class Cargo to be $\{at(Cargo, Location))\}$ and $\{in(Cargo, Rocket)\}$.

Object	Property	Value
C1	at	Paris
C2	at	London
C3	in	R1
C4	at	London

Figure 2: Object Properties in the OWL Rocket World

(ii) Appendix A: the Dockworkers World, taken from (Ghallab, Nau, & Traverso 2004). As with the Rocket World, OWL2OCL has succeeded in translating the object and property information into the more compact OCL formalism. Some dynamic, substate definitions have been induced successfully - for example Containers are in Piles and may or may not be at the top, and Piles, Cranes and Robots have dynamic location information. However, the quality of this is dependent on the amount and spread of information about individuals in the ontology.

Experiments

We have used OWL2OCL to translate information from a range of OWL ontologies. The discussion below draws from the use of Horrocks' 'people+pets' ('mad cows') ontology from Manchester, not because it forms the basis of any sensible planning domain, but because it at least includes a reasonably large ontology which uses many of the features of OWL and includes some ABox information. Our process will only work well if the ontology contains information about the individuals in the domain (ABox information), and preferably has the individuals spread of the their possible states.

Use of Description Logic Reasoners

In some cases it is possible to infer a more refined classification, e.g. by looking at domain and range restrictions for properties. An example is in the people+pets ontology: the concept 'dog' has no declared superclass, but a description logic reasoner can infer that 'dog' is a subclass of 'animal'. We have assumed that all implicit subclass relationships that can be found by a description logic reasoner have been made explicit, and we have used the RACER DL reasoner for this purpose (Haarslev & Moeller 2001) via Protégé2000 (Knublauch, Fergerson, & N. F. Noy 2004) for this purpose (see figure 1).

Inverse roles

OWL allows for roles to have a declared inverse form, e.g. `has_pet` is the inverse of `is_pet_of`. There is a choice in the automatic handling of inverse roles. A useful factor in making this choice is whether the property is declared as functional in one of its forms. The functional form is more convenient to handle, as we are then able to constrain the number of occurrences of the property in a substate to 0 or 1. For example the `is_pet_of` property is functional (a pet has a single owner, but an owner can have many pets),

so it is the preferable to associate the property with the pet class rather than the owner.

Property hierarchies

OWL allows for the declaration of hierarchies of properties. For example, has_father is a subproperty of has_parent. We do not handle the such properties in the conversion, as we cannot expect a planner to do inference using the role hierarchy. In principle, we could represent such cases by completing the descriptions by adding all implied super-properties for each instance. However, if the properties in question are to be considered dynamic in the planning domain, the planning operators would be required to adjust the properties appropriately at every level of the role hierarchy. The problems of representing *transitive* properties are similar to those presented by hierarchies of properties. Although we could compute transitive closure in advance, we cannot expect to define planning operators which can recompute it. [1]

Dynamic unary predicates

We assume that dynamic truth-valued data relating to individuals will be represented in the ontology using boolean data properties. For example, in the DWR domain, we record whether a location is occupied, and this is dynamic data which can be changed by a plan operator.

```
DatatypeProperty(a:occupied Functional
 domain(a:Location)
 range(xsd:boolean))
```

In the OCL representation, the boolean data properties are translated into a pair of unary predicates `occupied(Location)` and `not_occupied(Location)`.

Defining Planning Operators

In summary, the partial domain model created by OWL2OCL creates a useful starting point for domain acquisition. To complete the planning domain we need to use GIPO to:

1. Refine or alter the structure of the substate class definitions which has been extracted from the ontology.
2. Fill in missing class definitions.
3. Define planning operators in terms of transitions between the defined class definitions.

Related Work

PDDAML is a tool which translates between a Web-PDDL and DAML. It differs from OWL2OCL as it assumes the same 'semantic content' in its target and source, and does not propose to induce or hypothesise any existing knowledge. This was written with the benefit of a PDDL ontology definition (McDermott, Dou, & Qi 2004). Our work here is to try to import and extract from ontologies as much planning related information as possible, rather than to create a web-planning language. We recognise that the purpose of ontologies are generally to record the static information about a domain; this makes them unusable as a planning definition. To help the engineering of a dynamic domain, existing ontologies are in our work used as the starting point.

McNeil et al (McNeill, Bundy, & Walton 2004) describes a system for generating a PDDL planning domain from an ontology describing an agent system represented in the KIF formalism. The ontology is not restricted to static domain information, but already contains descriptions of actions. Hence the problem tackled is one of translating the planning problem between formalisms, as opposed to our aim of extracting state descriptions as a basis for defining actions.

[1] although PDDL 2.2 language allows for axioms could be used for this.

Conclusions

In this paper we have argued that knowledge available from online ontologies can be usefully imported into AI Planning domain models to act as the first step to creating a planning domain model. We have described a tool that translates 'OWL abstract' ontologies into the OCL plan language. The success of this tool depends on the amount and distribution of factual and individual knowledge (that is a full 'Abox'). Experiments with this tool illuminate other problems inherent in this approach: OWL ontologies tend to contain some richer information (such as property hierarchies) than planning domain models require, but also not sufficient dynamic information. It would be useful to know which data is static and which is dynamic. Unfortunately this is impossible to detect automatically from a static snapshot of the ontology.

For future work we plan to further develop the technique to extract more information from OWL relevant to planning. For example, number restrictions recorded in OWL can usefully be used to infer constraints on what is a valid substate, and OWL's MinCardinality restrictions could be used to add extra properties to a substate.

References

Bouquet, P.; Giunchiglia, F.; van Harmelen, F.; Serafini1, L.; and Stuckenschmidt, H. 2004. Contextualizing ontologies. *Journal of Web Semantics* 1(4):325 – 343.

Gennari, J. H.; Musen, M. A.; Fergerson, R. W.; Grosso, W. E.; Crubezy, M.; Eriksson, H.; Noy, N. F.; and Tu, S. W. 2003. The evolution of Protege: an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.* 58.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Parctice*. Morgan Kaufmann ISBN 1-55860-856-7.

Haarslev, V., and Moeller, R. 2001. RACER system description. In *International Joint Conference on Automated Reasoning, IJCAR'2001*.

Knublauch, H.; Fergerson, R. W.; and N. F. Noy, M. A. M. 2004. The protégé OWL plugin: An open development environment for semantic we b applications.

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department

of Computing and Mathematical Sciences, University of Huddersfield .

McCluskey, T. L.; Liu, D.; and Simpson, R. M. 2003. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. In *The Thirteenth International Conference on Automated Planning and Scheduling*.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *The Sixth International Conference on Artificial Intelligence Planning Systems*.

McDermott, D.; Dou, D.; and Qi, P. 2004. PDDAML: An automatic translator between PDDL and DAML. http://www.cs.yale.edu/homes/dvm /daml/pddl_daml_translator1.html.

McNeill, F.; Bundy, A.; and Walton, C. 2004. An automatic translator from KIF to PDDL. In *Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG 2004*.

Patel-Schneider, P. F.; Hayes, P.; and Horrocks, I. 2004. OWL web ontology language semantics and abstract syntax W3C recommendation 10 February 2004. http://www.w3.org/2004/OWL/.

Appendix A

The Rocket World:

```
Ontology(
 Class(Rocket partial)
 Class(Location partial)
 Class(Cargo partial)
 Class(Level partial)

 DisjointClasses(Rocket Location Cargo)

 ObjectProperty(at   domain(Cargo)
                     range(Location) )
 ObjectProperty(position
                     domain(Rocket)
                     range(Location) )
 ObjectProperty(fuel domain(Rocket)
                     range(Level) )
 ObjectProperty(in domain(Cargo)
                     range(Rocket) )

 Individual(R1 type(Rocket)
      value(position Paris)
      value(fuel full ) )
 Individual(R2 type(Rocket)
      value(position Paris)
      value(fuel full ) )
 Individual(C1 type(Cargo)
      value(at Paris) )
 Individual(C2 type(Cargo)
      value(at London) )
 Individual(C3 type(Cargo)
      value(in R1) )
 Individual(C4 type(Cargo)
      value(at London) )
 Individual(Paris type(Location))
 Individual(London type(Location))
 individual(full type(level))
 individual(empty type(level))
)

% Domain auto-generated from an OWL ontology

domain_name(owl).

% Objects

objects(cargo,[c1,c2,c3,c4]).
objects(level,[empty,full]).
objects(location,[london,paris]).
objects(rocket,[r1,r2]).

% Predicates

predicates(
   at(cargo,location),
   position(rocket,location),
   fuel(rocket,level),
   in(cargo,rocket)]).

% Object Class Definitions

substate_class(cargo,Cargo,[
   [at(Cargo,Location)],
   [in(Cargo,Rocket)]]).
substate_class(rocket,Rocket,[
   [position(Rocket,Location),
    fuel(Rocket,Level)]]).
```

Appendix B

The Dockworkers World:

```
Ontology(

 Class(a:Container partial)
 Class(a:Crane partial)
 Class(a:Location partial)
 Class(a:Pile partial)
 Class(a:Robot partial)

 ObjectProperty(a:adjacent Symmetric
  domain(a:Location)
  range(a:Location))
 ObjectProperty(a:at Functional
  domain(a:Robot)
  range(a:Location))
 ObjectProperty(a:attached Functional
  domain(a:Pile)
  range(a:Location))
 ObjectProperty(a:belong Functional
  domain(a:Crane)
  range(a:Location))
 ObjectProperty(a:holding
  domain(a:Crane)
  range(a:Container))
 ObjectProperty(a:in
  domain(a:Container)
  range(a:Pile))
 ObjectProperty(a:loaded
  domain(a:Robot)
  range(a:Container))
 ObjectProperty(a:on
```

```
  domain(a:Container)
  range(a:Container))
 ObjectProperty(a:top
  domain(a:Container)
  range(a:Pile))

 DatatypeProperty(a:empty Functional
  domain(a:Crane)
  range(xsd:boolean))
 DatatypeProperty(a:occupied Functional
  domain(a:Location)
  range(xsd:boolean))
 DatatypeProperty(a:unloaded Functional
  domain(a:Robot)
  range(xsd:boolean))

 Individual(a:a
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pa))
 Individual(a:b
  type(a:Container)
  value(a:on a:a)
  value(a:in a:pa))
 Individual(a:c
  type(a:Container)
  value(a:on a:b)
  value(a:in a:pa)
  value(a:top a:pa))
 Individual(a:d
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qa))
 Individual(a:e
  type(a:Container)
  value(a:on a:d)
  value(a:in a:qa))
 Individual(a:f
  type(a:Container)
  value(a:on a:e)
  value(a:in a:qa)
  value(a:top a:qa))
 Individual(a:g
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:pb))
 Individual(a:ga
  type(a:Crane)
  value(a:belong a:la)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gb
  type(a:Crane)
  value(a:belong a:lb)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gc
  type(a:Crane)
  value(a:belong a:lc)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gd
  type(a:Crane)
  value(a:belong a:ld)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:ge
  type(a:Crane)
  value(a:belong a:le)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:gf
  type(a:Crane)
  value(a:belong a:lf)
  value(a:empty "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:h
  type(a:Container)
  value(a:on a:g)
  value(a:in a:pb))
 Individual(a:i
  type(a:Container)
  value(a:on a:h)
  value(a:in a:pb)
  value(a:top a:pb))
 Individual(a:j
  type(a:Container)
  value(a:on a:pallet)
  value(a:in a:qb))
 Individual(a:k
  type(a:Container)
  value(a:on a:j)
  value(a:in a:qb))
 Individual(a:l
  type(a:Container)
  value(a:on a:k)
  value(a:in a:qb)
  value(a:top a:qb))
 Individual(a:la
  type(a:Location)
  value(a:adjacent a:li)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:lb
  type(a:Location)
  value(a:adjacent a:lj)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:lc
  type(a:Location)
  value(a:adjacent a:lj)
  value(a:occupied "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
 Individual(a:ld
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:le
  type(a:Location)
  value(a:adjacent a:li))
 Individual(a:lf
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:li
  type(a:Location)
  value(a:adjacent a:lj))
 Individual(a:lj
  type(a:Location))
 Individual(a:m
```

```
 type(a:Container)
 value(a:on a:pallet)
 value(a:in a:pe))
Individual(a:n
 type(a:Container)
 value(a:on a:m)
 value(a:in a:pe))
Individual(a:o
 type(a:Container)
 value(a:on a:n)
 value(a:in a:pe)
 value(a:top a:pe))
Individual(a:p
 type(a:Container)
 value(a:on a:pallet)
 value(a:in a:qe))
Individual(a:pa
 type(a:Pile)
 value(a:attached a:la))
Individual(a:pallet
 type(a:Container)
 value(a:top a:pf)
 value(a:top a:pc)
 value(a:top a:qf)
 value(a:top a:qc))
Individual(a:pb
 type(a:Pile)
 value(a:attached a:lb))
Individual(a:pc
 type(a:Pile)
 value(a:attached a:lc))
Individual(a:pd
 type(a:Pile)
 value(a:attached a:ld))
Individual(a:pe
 type(a:Pile)
 value(a:attached a:le))
Individual(a:pf
 type(a:Pile)
 value(a:attached a:lf))
Individual(a:q
 type(a:Container)
 value(a:on a:p)
 value(a:in a:qe))
Individual(a:qa
 type(a:Pile)
 value(a:attached a:la))
Individual(a:qb
 type(a:Pile)
 value(a:attached a:lb))
Individual(a:qc
 type(a:Pile)
 value(a:attached a:lc))
Individual(a:qd
  type(a:Pile)
 value(a:attached a:ld))
Individual(a:qe
 type(a:Pile)
 value(a:attached a:le))
Individual(a:qf
 type(a:Pile)
 value(a:attached a:lf))
Individual(a:r
 type(a:Container)
 value(a:on a:q)
 value(a:in a:qe)
 value(a:top a:qe))
Individual(a:ra
 type(a:Robot)
 value(a:at a:la)
 value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
Individual(a:rb
 type(a:Robot)
 value(a:at a:lb)
 value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
Individual(a:rc
 type(a:Robot)
 value(a:at a:lc)
 value(a:unloaded "true"
^^http://www.w3.org/2001/XMLSchema#boolean))
Individual(a:s
 type(a:Container)
 value(a:on a:pallet)
 value(a:in a:pd))
Individual(a:t
 type(a:Container)
 value(a:on a:s)
 value(a:in a:pd))
Individual(a:u
 type(a:Container)
 value(a:on a:t)
 value(a:in a:pd)
 value(a:top a:pd))
Individual(a:v
 type(a:Container)
 value(a:on a:pallet)
 value(a:in a:qd))
Individual(a:w
 type(a:Container)
 value(a:on a:v)
 value(a:in a:qd))
Individual(a:x
 type(a:Container)
 value(a:on a:w)
 value(a:in a:qd)
 value(a:top a:qd))

)
```

```
% Domain auto-generated from an OWL ontology

domain_name(owl).


% Objects

objects(container,[a,b,c,d,e,f,g,h,i,j,k,l,
      m,n,o,p,pallet,q,r,s,t,u,v,w,x]).
objects(crane,[ga,gb,gc,gd,ge,gf]).
objects(location,[la,lb,lc,ld,le,lf,li,lj]).
objects(pile,[pa,pb,pc,pd,pe,pf,qa,qb,qc,
      qd,qe,qf]).
objects(robot,[ra,rb,rc]).
```

```
% Predicates

predicates(
   adjacent(location,location),
   at(robot,location),
   attached(pile,location),
   belong(crane,location),
   holding(crane,container),
   in(container,pile),
   loaded(robot,container),
   on(container,container),
   top(container,pile),
   empty(crane),
   not_empty(crane),
   occupied(location),
   not_occupied(location),
   unloaded(robot),
   not_unloaded(robot)]).


% Object Class Definitions

substate_class(container,Container,[
   [on(Container,Container2),in(Container,Pile3)],
   [on(Container,Container2),in(Container,Pile3),
     top(Container,Pile4)],
   [top(Container,Pile2),top(Container,Pile3),
    top(Container,Pile4),top(Container,Pile5)]]).
substate_class(crane,Crane,[
   [belong(Crane,Location2),empty(Crane)]]).
substate_class(location,Location,[
   [],
   [adjacent(Location,Location2)],
   [adjacent(Location,Location2),occupied(Location)]]).
substate_class(pile,Pile,[
   [attached(Pile,Location2)]]).
substate_class(robot,Robot,[
   [at(Robot,Location2),unloaded(Robot)]]).
```