Opmaker2: Efficient Action Schema Acquisition

T.L.McCluskey, S.N.Cresswell, N. E. Richardson and M.M.West
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK

Abstract

The problem of formulating knowledge bases containing specifications of dynamic knowledge is a barrier to the widespread uptake of AI planning. Machine learning has been used with some success in the past, but the inputs required are either too detailed, or the learning process has required many examples. Further, learning has been confined to propositional actions or parts of actions such as preconditions. The field of ontological engineering has had an impact on the wider community in that application ontologies (which contain "static" structural knowledge of applications) are becoming widespread. Here we introduce a methodology that is based on the existence of a strong structural model of an application. Using a small number of user training sequences, we illustrate how the method can induce action schema and compound methods. To do this we extend GIPO's Opmaker system so that it can induce actions from training sequences without intermediate state information and without requiring large numbers of examples. This method shows the potential for considerably reducing the burden of knowledge engineering, in that it would be possible to embed the method into an autonomous program (agent) which required to do planning. We illustrate the algorithm as part of an overall method to induce structured domain model, and comment on initial results that show the efficacy of the induced model empirically.

Introduction

The problem of formulating knowledge bases containing specifications of dynamic knowledge is a barrier to the widespread uptake of AI planning. Current high profile applications such as the use of planning technology within NASA's Mars Rover require persistent resources comprising of teams of highly skilled knowledge engineers, In particular, a problem facing AI is to overcome the need to hard code and manually maintain action schema within agents (a problem which limits their autonomy). It is possible to use learning techniques to help overcome the problem, eg using tools which induce actions or methods from examples. One method is to embed agents with the ability to induce the detailed specification of action schema from example planning traces, possibly supplied by a trainer. Planning traces are an ordered set of action instances, where each action instance is identified by name plus the object instances that are affected or are necessarily present but not affected, by action execution. This is the kind of information normally expected as a solution to planning problems.

In this paper we describe the results of an investigation into (re)constructing action schema and planning heuristics from training sessions which compose of a handful of action traces. The main result is that it is possible for an agent to induce detailed specifications of action schema from single action traces automatically, without requiring intermediate state information for each training example. The trade-off is that the agent's domain description should contains invariants describing object relations and object states. The induced actions are detailed enough for use in planning engines. We present an algorithm for generating such domain models, and show how the primitive action schema can be built up into domain models.

In our previous work we have shown how 'flat' domain actions can be induced from examples. Actions can be induced using *Opmaker* (McCluskey, Richardson, and Simpson 2002) which has been embedded interactively in *GIPO* (Simpson et al. 2001), (Simpson 2005). *GIPO* aids domain construction, offering editors, validation tools, a graphical life-history editor and planning tools. Output from *GIPO* is the completed and validated domain being modelled in a variant of GIPO's internal language *OCL* (Liu and McCluskey 2000) or PDDL. Here we extend GIPO's Opmaker system so that it can induce actions from training sequences and its static object model alone, without intermediate state information and without requiring large numbers of examples. This considerably reduces the burden of knowledge engineering, so that a program (agent) can perform knowledge acquisition rather than it occurring through a human-driven process supported by a tool such as GIPO.

The rationale for setting up this problem is as follows. The acquisition / refinement of factual or static knowledge by agents is relatively straightforward. In the context of the internet and open systems, it is not unreasonable that an agent can acquire and refine such knowledge with some degree of autonomy. The rapid expansion of globally accessible ontologies within standard formats such as OWL, support the notion that intelligent agents will have access to factual knowledge. In contrast, the amount of effort needed to encode bug free, accurate action specifications and planning heuristics, and to maintain them, is significant. A necessary precondition of the use of current automated planning technology is that there exists a detailed action specification, and in many cases, heuristic knowledge. Hence we can ask the question: for every agent that can perform planning, must we hand code and hand maintain its action descriptions? No, if agents are to achieve this kind of autonomy, then they should be capable of learning and refining action knowledge and heuristics.

The Learning Problem

The general situation is one where an agent needs to perform reasoning about actions to achieve a desired goal, and in particular perform plan generation within an environment that it has knowledge of. Actions are real world operations that change the state of object(s) in the world in some way. The agent has knowledge of objects, and collections of similar objects making up distinct classes. It knows the possible states of a typical object of each class. It has knowledge of existing plans that other agents, or a trainer, has used. These plans are written in terms of verbs and affected objects (pick up block A with gripper B, lift up wheel A with jack B). Additionally, the agent is assumed to have axioms describing a naive physics of the world. However, the agent has not an explicit specification of actions in such a way that it can reason about their synthesis (or the agent does have such a specification but needs to refine, maintain or evolve it).

Given this situation, the learning problem is to induce a full parameterised specification of actions which can be used to do planning; and to induce heuristics which can be used to make the reasoning involved in the planning computationally tractable. Further, the agent should be able to *refine* any existing parameterised specification of actions, and heuristics, that it currently holds. The action specifications should be detailed enough so that they can be input to mainstream planning technology as epitomised by competitors in the IPC (the bi-annual international planning competition).

A Formulation of the Problem

We formulate the learning problem as follows:

INPUT: Assume the input to the learning problem is a 'model' of the world, and a set of training sequences, given as follows:
1.1 - there are a number of classes each containing a set of objects, each object belongs to one set (called a sort)
1.2 - each object of each class may be related to objects of other classes, and have property - value relationships with set of basic values (boolean or scalar). The relations and properties are defined in the usual way using predicates.
1.3 - each object of each class at a moment in time has a fixed 'state'. This state is defined by its relationship with other objects and/or the value of properties. There are a small, finite number of states for each object class.
1.4 - there is a set I of invariants relating the predicates given above. Informally, a set is adequate if any 'common sense' inference can be made from them, such as normal inferences about spatial relations.
1.5 a set of training plans of the form

(initial state, final state)
$name_1\ p_1, o_1$
$name_2\ p_2, o_2$
..
$name_q\ p_q, o_q$

$name_1..name_q$ are the names of the q actions in the training plan, and they are assumed to transform the initial state into the final state. Here $p_1, p_2, ..p_q$ are each lists of object names (

they could be null) of unchanging or 'prevail' objects required by an action, and $o_1, o_2, ..o_q$ are each lists of object names affected by the execution of the action. Each of the list of prevail objects must be present in some state, but that state does not change during action execution.

- a (possibly empty) set of existing action schema. Within this formulation, action schema are parameterised object transformations.

OUTPUT a set of action schema that - is consistent with the static domain model components (1.1 -1.4); - can be instantiated into the training plans (1.5) supplied, and will transform the initial state into the final state heuristics derived from the training plans that can be used to guide a planner

Method

The learning method is specified by the algorithm description in Figure 1. In outline, the method is:

(i) use a set of heuristics and inferences to track the changing states of each object referred to within a training example, taking advantage of the static, object-state information and invariants within the domain model. Infer full details of object transitions for each dynamic object.

(ii) use the techniques of the original Opmaker algorithm (McCluskey, Richardson, and Simpson 2002) to generalise object references and create parameterised operator schema from the specific object transitions extracted in (i) from the training examples.

To illustrate the main innovations of the method, we will use an example walk-though taken from our empirical evaluation involving an extended tyre-change domain. Assume a training sequence *SEQ* is input into *Opmaker*2 and this has components as follows:

name: do_up; prevail: wrench0,jack0, trim1; changing: hub1,nuts1
name: jack_down; changing: hub1,jack0
name: tighten; prevail: wrench0,hub1,trim1; changing: nuts1
name: apply_trim; prevail: hub1; changing: trim1,wheel5

This illustrates a short procedure for making a car wheel ready for operation once it has been hung on to an appropriate wheel hub. Informally, do_up is the operation of putting the nuts on the hub of a wheel when it is jacked up. The names such as wrench0, hub1 are references to actual objects. The prevail objects have to be necessarily present in a particular state but remain unaffected ('wrench0' is available, 'jack0' is jacking up the wheel, 'trim1' is hub1's wheel trim and has to have been removed). These objects need to be in particular states for the action to execute, and those states 'prevail' or stay the same during execution of the action. The 'changing' objects change state (hub1 becomes fastened up, the nuts1 are fastened up).

To illustrate some of the definitions in Line 1 of the algorithm in Figure 1, we have components of an object as follows:

$hub1.c$ = [unfastened(hub1), jacked_up(hub1,jack0)]
$hub1.f$ = [on_ground(hub1), fastened(hub1)]
$hub1.s$ = hub

Examples of other operations are (h and j are parameters):

$hub1.c^g$ = [unfastened(h), jacked_up(h,j)]
$hub1.c_s$ = [hub,jack]

Line 2 iterates through all the training examples. For the first training example, the problem is to determine what the new states are of hub1 and nuts1.

In Line 3, let P = [w, j, t, n, h]. In Lines 4-6, the prevail components are got from the current state classes of wrench0, jack0 and trim1, as in the original Opmaker algorithm. The loop starting on line 7 is intended to determine the destination of each object that is changed by the action being learned. hub1 is the first changing object. From the given partial definition of the domain, it has four state classes which we name S1-4:

S1 = [on_ground(h),fastened(h)],
S2 = [jacked_up(h,j),fastened(h)],
S3 = [free(h),jacked_up(h,j),unfastened(h)],
S4 = [unfastened(h),jacked_up(h,j)]

hub1's current state is not necessarily its final one, as in the training sequence it is referred to again (in the second of the sequence, *jack_down*) as a changing object. Hence line 10 is executed. X cannot be S4 (since this is currently the generalisation of the object's current state, and the

```
**program** $Opmaker2$
**In** partial domain model
**In** training sequence $SEQ$ with N actions, and each $e \in SEQ$ has components:
$e.Name$, $e.prevail$, $e.changing$ = name, unchanging objects, changing objects,
**Out** parameterised action descriptions and HTN methods
1. Definitions:
$O.c$ = current state of an object $O$
$O.s$ = sort of object $O$
$O.f$ = final state of an object $O$
$S^g$ = state class of some $groundstate$ $S$
$O^g$ = a distinct parameter which ranges through the sort of $object$ $O$
$X_s$ = set of all sorts of parameters and objects in expression $X$
2.for each $e$ in $SEQ$ do
3.        Form $P$ = list of $O^g$ for all $O$ in $e.preval \cup e.changing$;
4.        for each $O$ in list $e.preval$ do
5.                store component of the prevail $(O.s, O^g, O.c^g)$
6.        end for
7.        for each $O$ in list $e.changing$ do
8.                if $O$ is not affected by actions in the rest of $SEQ$
9.                then let $X = O.f^g$
10.                 else choose $X$ from the state classes of $O.c$ such that
11.                         $X \neq O.c^g$ and $P_s$ contains $X_s$
12.                 store transition $T = (O.s, O^g, O.c^g \Rightarrow X)$
13.                 match free vars in $T$ with those in $P$
14.         end for
15.         form actions from cross-product of all stored transitions
16.         such that the actions are consistent with invariants
17. end for
18. produce a method from the sequences of actions as in Opmaker.
**procedure** match free vars in $T$ with those in $P$
1. repeat
2.        for each parameter $X$ in transition $T$, $X \neq O$,
3.                choose a parameter $Y$ in $P$ to match with
4.                $X$ such that $Y \neq O$, $sort(X) = sort(Y)$,
5.        end for
6. until parameter match set is consistent
7. end
```

Figure 1: Outline Design of the *Opmaker2* Algorithm

object has to change state class). In Line 11 P_s (= [wrench, jack, trim, nuts, hub]) contains all the sorts in each of state classes S1,S2 and S3, and so this does not narrow down the choices. Hence 3 transitions are stored:

(hub, h, [unfastened(h),jacked_up(h,j)] $\rightarrow$ [on_ground(h),fastened(h)])
(hub, h, [unfastened(h),jacked_up(h,j)] $\rightarrow$ [free(h),jacked_up(h,j),unfastened(h)])
(hub, h, [unfastened(h),jacked_up(h,j)] $\rightarrow$ [jacked_up(h,j),fastened(h)])

Iteration of line 7 with object *nuts*1 occurs next. It has three states:

T1 = [tight(N,h)]
T2 = [loose(N,h)]
T3 = [have_nuts(N)]

This leads to 2 possible transitions:

(nuts, N, [have_nuts(N)] $\rightarrow$ [tight(N,h)])
(nuts, N, [have_nuts(N)] $\rightarrow$ [loose(N,h)])

and hence 6 possible induced action schema (line 15). These six options are then checked for consistency with the domain invariants which are shown in Figure 2. The conjunction of state constraints in both the LHS and RHS of transitions of the newly formed action schema must be consistent with these invariants. In cases where they are not, the action schema is discarded.

This reduces the number of options to a single action schema. Processing of the other 3 actions in the training sequence leads to a single interpretation of state changes, as the changing objects involved are all in their final states, and hence 3 more generalised action schemas are generated. Finally, a hierarchical method is generated (line 18) by combining the 4 action schema in a similar fashion to the original Opmaker system (McCluskey, Richardson, and Simpson 2002).

Experiments and Results

The method has been implemented and merged with the original Opmaker system. We are using the same experimental approach as we used to test the original system:

- We hand-craft training sequences from a range of domains selecting actions that will build sensible methods for that domain.
- We use Opmaker2 to induce actions and hierarchical (HTN-type) methods from the training sequences.
- Using standard planners, we compare performance using old hand-crafted action schema to the use of induced schema.

Success will be judged using the following criteria:

- If a valid set of unique new actions is defined as actions that can solve the same problems the original training sequences were aimed at, can *Opmaker*2 induce these without having to encode a great deal of invariants into the domain models?
- Is it more efficient in terms of effort time to construct a domain using *Opmaker*2?
- Is it at least as efficient, in terms of planning time, to reach goals using *Opmaker*2 defined actions and methods?

Up to now we have experimented with 2 domain models: the extended tyre world, and the hiking domain (see http://planform.hud.ac.uk/gipo/ for details of these).

Since induction sequences deliver several actions and a single method, initial sequences were tailored to produce a meaningful method, and sufficient initial sequences were composed to cover all the major sub-tasks that could be required by the domain. In each case the agent began by knowing domain knowledge but had sketchy or non-existent facts about its potential actions. For the Extended Tyre World we devised 7 sequences of between 2 and 5 actions in length. After adding 8 invariants to the domain we induced a set of actions and methods and using these we produced a domain with 22 actions and 7 methods. The new version was tested over 8 tasks in two ways - firstly using just actions in the planning and secondly using either just methods, or a combination of methods and actions. To illustrate the results, two of the actions that were induced from the running example were as follows:

```
operator(jack_down(Hub1,Jack0),
[],
[sc(hub,Hub1,[jacked_up(Hub1,Jack0),
            fastened(Hub1)] =>
  [on_ground(Hub1),fastened(Hub1)]),
sc(jack,Jack0,[jack_in_use(Jack0,Hub1)] =>
  [have_jack(Jack0)])], []).
```

1. Equivalence between hub *fastened* and nuts *tight/loose* on hub.

$$\forall\, H{:}hub\ .\ [fastened(H) \Longleftrightarrow \exists\, N{:}nuts\ .\ (tight(N,H) \vee loose(N,H))]$$

2. Equivalence between *jack_in_use* and *jacked_up*.

$$\forall\, H{:}hub\ .\ \forall\, J{:}jack\ .\ [jack\_in\_use(J,H) \Longleftrightarrow jacked\_up(H,J)]$$

3. Equivalence between hub not *free* and *wheel_on* hub.

$$\forall\, H{:}hub\ .\ [\neg free(H) \Longleftrightarrow \exists\, W{:}wheel\ .\ wheel\_on(W,H)]$$

4. Equivalence between *trim_on_wheel* and *trim_on*.

$$\forall\, T{:}wheel\_trim\ .\ \forall\, W{:}wheel\ .\ [trim\_on\_wheel(T,W) \Longleftrightarrow trim\_on(W,T)]$$

5. Only a single set of nuts can be on a hub.

$$\forall\, H{:}hub\ .\ \forall\, N_1{:}nuts\ .\ \forall\, N_2{:}nuts\ .\ \left[\ \left(\begin{array}{c} (tight(N_1,H) \vee loose(N_1,H)) \\ \wedge \\ (tight(N_2,H) \vee loose(N_2,H)) \end{array}\right)\ \Rightarrow (N_1 = N_2)\ \right]$$

6. Only a single wheel can be on a hub.

$$\forall\, H{:}hub\ .\ \forall\, W_1{:}wheel\ .\ \forall\, W_2{:}wheel\ .\ \left[\left(\begin{array}{c} wheel\_on(W_1,H) \\ \wedge \\ wheel\_on(W_2,H) \end{array}\right) \Rightarrow (W_1 = W_2)\right]$$

7. Domain constraint: If nuts are tight on a hub then the hub must be on the ground.

$$\forall\, H{:}hub\ .\ [(\exists\, N{:}nuts\ .\ tight(N,H)) \Rightarrow on\_ground(H)]$$

8. Domain constraint: if a trim is on a wheel, then the wheel is on a hub and the nuts are tight.

$$\forall\, W{:}wheel\ .\ \exists\, T{:}wheel\_trim\ .\ \left[\begin{array}{c} trim\_on\_wheel(T,W) \Rightarrow \\ (\exists\, H{:}hub\ .\ wheel\_on(W,H)) \wedge (\exists\, N{:}nuts\ .\ tight(N,H)) \end{array}\right]$$

Figure 2: Invariants encoded in the Extended Tyre World

```
operator(tighten(Wrench0,Hub1,Nuts1,Trim1),
[se(wrench,Wrench0,[have_wrench(Wrench0)]),
se(hub,Hub1,[on_ground(Hub1),fastened(Hub1)]),
se(wheel_trim,Trim1,[trim_off(Trim1)])],
[sc(nuts,Nuts1,[loose(Nuts1,Hub1)] =>
  [tight(Nuts1,Hub1)])], []).
```

Where just actions were used in planning, plan times for short plans of up to 10 to 12 actions were about the same as for the hand-crafted version of the domain. For plans longer than 12 actions both versions took increasingly long times to solve. However where methods or combinations of actions and methods were used plan times were significantly shorter. The full planning problem for this extended domain is defined to be: "A car is found to have two flat tyres, one is found to be flat and can be fixed by use of the pump, whilst the other is punctured and requires the full tyre change described in the previous version of the domain". Using just actions no solution was found to this problem after 36 hours but using methods and just a few actions a correct solution was found after 11 seconds.

Experimentation with the hiking domain is at an earlier stage. As yet no invariants have been added to the domain. Without these we do not get unique sets of example material for induction but already we have seen actions generated. We identified 5 potential methods for this domain and for four of these we obtained example sets of no larger than 6. However the fifth generated 28 example sets so either a set of invariants will be added to the agent's knowledge, or we will use theory refinement to reduce the example sets further.

From the results obtained so far we can conclude that an agent, given a 'working stock' of potential action sequences, and having domain knowledge and a 'belief' about the states of objects it 'knows' about will be able to generate its own examples and use them to supply itself with paramerised actions to suit every possible object combination. Since methods can be formed from the action sequences the agent should be able to plan efficiently and autonomously.

Related Work

The authors of (Garland, Ryall, and Rich 2001) have developed a system (Collagen) which learns task models from examples. Their works is similar to ours in that they show orderings of the task to achieve the task and these contain both primitives and non-primitives. In (Wu, Yang, and Jiang 2005) the authors describe ARMS, a system in which operators are learned without the need for user intervention. However ARMS requires many training examples containing valid solution sequences, and presently is capable of inducing only 'flat' domains.

Our work is also aimed at learning domains containing both action schema and hierarchical schema (methods) encapsulating several schema. Practical planning domains are based on 'hierarchical task network' (HTN) decomposition. The chief difference between the HTN paradigm and classical domains is that in the former 'compound' tasks can be decomposed into the simpler 'tasks' particular to classical domains. However HTNs can be difficult to construct manually and authors have worked in producing these using methods from machine learning. In (Erol, Hendler, and Nau 1996) the authors argue that HTN operators are more expressive than those of classical domains as well as being more efficient. Theoretical underpinning for 'High Level Actions' (HLAs) is presented in (Marthi, Wolfe, and Russell 2007). Each HLA admits one or more *refinements* into sequences of actions, where an action might be high level or primitive. The paper introduces a provably sound and complete algorithm which is implemented using a STRIPS-like language. The algorithm takes advantage of 'sound and complete' descriptions and, if successful, returns a primitive refinement of some high-level plans that achieves the goal set from the initial state.

In (Nejati, Langley, and Konik 2006) the authors describe how they induce *teleoreactive logic programs* from expert traces. The teleoreactive programs index methods by the goals they achieve. They use methods derived from explanation based learning to chain backwards from the end result of the sample trace. The explanation structure thus obtained is retained to produce new hierarchical structures. The method is applied to 'Depots' which involves crates that can be loaded into trucks and stacked. However the domain so constructed resulted in the successful solution of very few problems.

Further theoretical work on HTN planning is presented in (Ilghami et al. 2005). This paper introduces a formalism whereby situations are modelled where general information is available of tasks and sub-tasks, together with some plan traces but there are no details. In the early work all information about methods was required except for the preconditions. This limitation is overcome in later work by the same group (Ilghami, Nau, and Munoz-Avila 2006) a new algorithm 'HDL' (HTN Domain Learner) is presented which learns HTN domain descriptions from plan traces. Between 70 and 200 plan traces are required to induce the descriptions.

HTN-MAKER is presented in (Hogg and Munoz-Avila 2007). This receives as input a STRIPS domain model, a collection of STRIPS plans and task definitions and produces an HTN domain model. The experimental hypothesis is that after a few problems have been analysed an HTN domain model will be ultimately obtained able to solve most solvable problems. A version of the logistics-transportation domain is chosen for the experiment and good results are obtained. However these good results are not replicated for the blocks-world domain. One problem is the large number of methods which have to be learned, where one method might subsume another. They suggest choosing the most general method where this is the case. Another problem is for the planner to use methods in an infinitely recursive manner.

Conclusions

Our work and the results reported here depend on a structured view of domain knowledge about objects being available. Whereas in propositional, classical planning states are fairly arbitrary sets of propositions, we assume that the space of states is restricted in that objects are pre-conceived to be a fixed set of plausible states. Within this framework, we have described a method for inducing action schema that advances the state of the art in that it requires no intermediate state information, or large numbers of training examples, to induce a valid action schema set. Further, our preliminary results show that the hierarchical methods induced with the action schema can lead to more efficient domain models.

Opmaker2 is an improvement on Opmaker in that it reduces and in some cases eliminates the need for the user or trainer to give the system intermediate state information. After Opmaker2 automatically infers this intermediate state information, it proceeds in the same fashion as Opmaker and induces the same operator schema. Opmaker2 can then logically be seen as a superset of Opmaker, where the extra functionality in Opmaker2 removes the need to ask the trainer for more informations.

Our experiments with the "Hiking Domain" show that further development needs to be made to the Opmaker2 algorithm so that it can cope with domains with "static" knowledge.

References

Erol, K.; Hendler, J.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Annals of Mathematics and Artificial Intelligence* 69–83.

Garland; Ryall; and Rich. 2001. Learning hierarchical task models by defining and refining examples. In *Proceedings of the First International Conference on Knowledge Capture.*

Hogg, C., and Munoz-Avila, H. 2007. Learning Hierarchical Task Networks from Plan Traces. In *Proceedings of the ICAPS'07 Workshop on Artificial Intelligence Planning and Learning.*

Ilghami, O.; Nau, D. S.; Muoz-Avila, H.; and Aha, D. W. 2005. Learning preconditions for planning from plan traces and HTN structure. *Computational Intelligence* 21(4):388–143.

Ilghami, O.; Nau, D. S.; and Munoz-Avila, H. 2006. Learning to do htn planning. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 390 – 393.

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 .

Marthi, B.; Wolfe, J.; and Russell, S. 2007. Semantics for High-level Actions. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS 2007.*

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.*

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, 665–672. New York, NY, USA: ACM Press.

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning.*

Simpson, R. M. 2005. Gipo graphical interface for planning with objects. In *Proceedings* *of the International Conference for Knowledge Engineering in Planning and Scheduling.*

Wu, K.; Yang, Q.; and Jiang, Y. 2005. Arms: Action-relation modelling system for learning acquisition models. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning.*