Action Knowledge Acquisition with Opmaker2

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, U.K.
{T.L.McCluskey,S.N.Cresswell,
N.E.Richardson,M.M.West}@hud.ac.uk

Abstract. AI planning engines require detailed specifications of dynamic knowledge of the domain in which they are to operate, before they can function. Further, they require domain-specific heuristics before they can function efficiently. The problem of formulating domain models containing dynamic knowledge regarding actions is a barrier to the widespread uptake of AI planning, because of the difficulty in acquiring and maintaining them. Here we postulate a method which inputs a $partial$ domain model (one without knowledge of domain actions) and training solution sequences to planning tasks, and outputs the full domain model, including heuristics that can be used to make plan generation more efficient.

To do this we extend GIPO's $Opmaker$ system [1] so that it can induce representations of 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 is required to do planning. We illustrate the algorithm as part of an overall method to acquire a planning domain model, and detail results that show the efficacy of the induced model.

Keywords: Planning and Scheduling; Machine Learning.

1 Introduction

Applications of AI planning technology require persistent resources comprising of teams of highly skilled engineers to formulate and maintain a planner's knowledge base. The amount of effort needed to encode error free, accurate action specifications and planning heuristics, and to maintain them, is significant. *Actions* are real world operations that change the state of object(s) in the world in some way. These actions are invariably encoded in planning knowledge bases as generalised representations called *operator schema*. Additionally, heuristics are often hand coded in the form of *methods* which encapsulate the preferred solutions of a generalised subtask. Our work is aimed at automating the formulation of such operators and methods by employing a trainer to create training tasks and example solution sequences of these tasks. The solutions are fed to a knowledge acquisition tool, $Opmaker2$, as a sequence of action instances, where each action instance is identified by name plus the object instances that are affected by, or are necessarily present at, action execution. The sequences are produced by a trainer - a domain expert who may not be familiar with the languages and notations

J. Filipe, A. Fred, and B. Sharp (Eds.): ICAART 2009, CCIS 67, pp. 137–150, 2010.

used by planners. $Opmaker2$ constructs operator schema and planning heuristics from training sessions which are composed of a handful of such action sequences. In other words, it outputs detailed specifications of operator schema from single action traces automatically, without requiring intermediate state information for each training example. The induced actions are detailed enough for use in planning engines and compare well with hand crafted operators.

This paper describes $Opmaker2$, an extension of the earlier *Opmaker* system [2], in that the latter is an interactive learning tool, whereas the former can be run in batch mode without the need for user assistance. $Opmaker$ was implemented within the GIPO system [1], an experimental tools environment for use in the acquisition of AI planning knowledge, containing a wide range of engineering and validation tools. GIPO was based on the planning language of OCL [3]. To motivate the rest of the paper, we will describe in a little more depth the problem that we are aiming to solve, in terms of a learning, or more specifically a *knowledge acquisition* problem. Automated planning systems can be logically described as having three components.

(a) *The domain model* (sometimes referred to as a *domain description*) is the specification of the objects, structure, states, goals and dynamics of the domain of planning. The language family used for the communication of domain models is PDDL [4], although in this paper we use a higher level language called *OCL*[5] for domain modelling. Component (a) is further split into:
(i) knowledge of objects, object sorts, domain constraints, and possible states of objects - collectively called static knowledge.
(ii) knowledge of action and change - knowledge of dynamics. This knowledge is in both PDDL and *OCL* represented as a set of parameterised operator schema representing generic actions in the domain of interest.

(b) The *planning engine* is the software that reasons with the knowledge in (a) to solve planning goals. The development of fast planning engines which can deal with expressive variants of PDDL (e.g. modelling domains containing durative actions and metric resources) has been a primary goal of the AI Planning community.

(c) A set of *planning heuristics*. The general problem of AI Planning is well known to be intractable, and a set of heuristics for each domain is required to make the application of (b) to (a) tractable. Whereas the form and content of (a) and (b) are well understood, what form heuristics take is more contentious. Putting domain heuristics with the planning engine may limit its application (they anticipate the domain). Encoding heuristics into the domain model when constructing it is equally contentious - as the authors of PDDL claim it is for "physics and nothing else"[4].

The knowledge acquisition problem that this paper addresses is:

> *Given knowledge of (a)(i), can we design a simple process to enable a system to automatically acquire knowledge of type (a)(ii) and (c)?*

The reason for setting up this knowledge acquisition problem is that hand crafting knowledge of dynamics (in particular operator schema), and planner and domain specific heuristics, is much harder than acquiring knowledge of type (a)(i). The difficulty in acquiring knowledge of actions is invariably pointed out in reports of AI planning applications (for example, in reports of Space applications [6]).

The general method that we are proposing is for a system to acquire knowledge from examples of solved tasks, represented as sequences of actions, given to it by a benevolent trainer. *Operator schema* (type (a)(ii) knowledge) are induced from each example action, whereas *heuristics* (type (c) knowledge) are induced from the whole sequence of actions the trainer uses to solve a task. The heuristics are in the form of HTN-type methods.

The rest of the paper is structured as follows: in section 2 we outline the $Opmaker2$ system, starting with its inputs and outputs, and then detail the operation of its state-deriving component. We use a *tyre-change domain* to illustrate the algorithm which contains the knowledge acquisition process. Section 3 contains our experimental results, and Section 4 a brief survey of related work.

2 The Opmaker2 System

In this section we describe the $Opmaker2$ system, and explain how it can form a solution to the knowledge acquisition problem introduced in the last section. We will use as a running example throughout the rest of the paper a domain which represents changing the tyre of a car wheel. This domain is an extended version of the simpler 'tyre world'[7]. It involves knowledge about such objects as tyre, wheel, nuts, wheel-trim, jack, wrench, and such actions as undo-nuts, put-on-wheel etc. In $Opmaker2$, components of type (a)(i) knowledge are referred to collectively as the *partial domain model* $\mathcal{PDM}$. For our running example, the partial domain of the tyre-change domain is provided in the appendix, in the native code of OCL. There are two inputs to $Opmaker2$: the $\mathcal{PDM}$ and a set of hand crafted solution sequences to planning tasks. A $\mathcal{PDM}$ consists of:

object identifiers and sort names: denoted $Objs$ and $Sorts$ respectively; there are a number of sorts, each containing a set of objects where each object belongs to one set (called a sort). An example of an object is $hub1$ belonging to the hub sort. The behaviour of each object in a sort is assumed to be the same as all others in the sort.

predicate definitions: denoted *Prds*, where each object of each sort may be related to objects of other sorts, and have property - value relationships with sets of basic values (boolean or scalar). Examples are $on\_ground(hub)$, $jacked\_up(hub, jack)$, relating to whether an object of sort hub is on the ground or jacked up.

object state expressions: denoted *Exps*; these define all the possible values of an object's state. An object's state is defined by its relationship with other objects and/or the value of its properties. Sorts are engineered so that the object state space is defined by a small number of expressions. For example, the tyre-change $\mathcal{PDM}$ specifies that any object H of sort hub can occupy a state satisfying exactly one of the following object expressions:

```
[on_ground(H),fastened(H)],
[jacked_up(H,J),fastened(H)],
[free(H),jacked_up(H,J),unfastened(H)],
[unfastened(H),jacked_up(H,J)]
```

(as a convention we choose upper case variables as parameters - here J represents any object of sort $jack$).

domain invariants: denoted *Invs*; these are used to define domain constraints and are written in terms of the predicates given above. Informally, a set of invariants is adequate if it disqualifies states which are inconsistent. For example: "Only a single wheel can be on a hub".

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

The second input is a set of *solution sequences* and the tasks that they solve. These are supplied by a trainer (a domain expert). For the purposes of training in $Opmaker2$, we define a task in terms of:

- an initial state comprising the initial states of objects in the domain,
- a set of desired goal states for a set of objects.

A solution sequence solves such a task and is written in terms of verbs (action names) and affected objects. The trainer is expected to include references to all objects that are needed for each action to be carried out, indicating whether or not the objects change as a result of the action. Typical tasks in the domain should be chosen that often form the basis of solutions to larger tasks. For example, in the sequence below a changed wheel is secured on the hub and the vehicle is made ready for use.

```
do_up       unchanged - wrench0, jack0, wheel1;
            changing - hub1, nuts1
jack_down   unchanged - null
            changing - hub1, jack0
tighten     unchanged - wrench0, hub1;
            changing - nuts1
apply_trim unchanged - hub1;
            changing - trim1,wheel1
```

Objects preceded by **unchanged** remain unaffected by the action, but have to be present in the state during execution of the action. In the first element of the sequence, `wrench0`, `jack0` and `wheel1` all have to be in a certain state specified by initial state of the task (`wrench0` is available, `jack0` is jacking up the hub, and `wheel1` is trimless to allow the nuts to be screwed). The **changing** objects *must* change state (`hub1` becomes $fastened$ and `nuts1` are $done\_up$).

The output of $Opmaker2$ is a full domain model, consisting of:

operator schema: they make up the knowledge of type (a)(ii), and represent actions or events that change objects' states. They are specified by a name, a list of parameters, and a set of object transitions. Transitions may be null (in which case they act as *prevail* conditions), necessary or conditional. The template of a schema is as follows:

head: name(list of parameters)
body: ≥ 0 prevail conditions;
≥ 1 necessary transitions;
≥ 0 conditional transitions

Prevails are represented by object state expressions, whereas necessary and conditional transitions are written in the form $LHS \Rightarrow RHS$, where LHS, RHS are object state expressions.

methods: each training sequence results in a parameterised method, similar in form to hierarchical (HTN) methods found in AI Planning. A method comprises of a name, prevail conditions, and a sequence whose members can comprise both operator schema and (other) methods. Methods can be used as a heuristic in planning engines as they encapsulate preferred ways to solve planning problems.

2.1 The Opmaker2 Algorithm

The main innovation of Opmaker2 is that it computes its own intermediate states using a combination of heuristics and inference from the $\mathcal{PDM}$ and the training tasks and solutions. This gives a fully automated solution to the knowledge acquisition problem described above - there is no need for user advice. In contrast, its predecessor $Opmaker$ is a *mixed initiative* knowledge acquisition tool which requires the same inputs as above (a $\mathcal{PDM}$ and a set of solution sequences to tasks) and, additionally, it requires user advice. As $Opmaker$ creates an operator schema from each action in a training solution sequence, it asks the user to input, if needed, the target state that each object would occupy after execution of the action. In order to build up transitions that form an operator schema, the LHS is taken as the current state of the object (object transitions are tracked as each action is processed). The RHS is taken from the user input, which indicates, where there is a choice, the state an object is left in (this becomes that object's current state). Having the start and end states for each object involved in the action, $Opmaker$ proceeds with a *generalisation* phase where object instances are replaced with sort parameters, which then form the parameter variables $X_1, \ldots, X_n$ of the resulting operator schema. In supplying the solution sequences, the trainer specifies what objects take part in what actions. As actions are executed, objects go through state transitions and occupy intermediate states en route to reaching their goal states. The space of states that an object may occupy are defined by the state expressions of the $\mathcal{PDM}$. To be able to automatically acquire operator schema, $Opmaker$ was able to resolve exactly what are the intermediate states of each object affected in the training sequence by asking for user advice.

In contrast, *Opmaker2* uses a procedure called *DetermineStates*, which performs this function by tracking the changing states of each object referred to within a training example. It takes advantage of the static, object-state information and invariants within the domain model. The output from *DetermineStates* is, for each point in the training sequence, a map which associates each object with a unique state value. Uniqueness is not guaranteed, however, and depends on the information in the $\mathcal{PDM}$, hence sometimes this map may return a set of states rather than a unique one (we return to this problem below). Once the map determining intermediate states has been generated, the

techniques of the original $Opmaker$ algorithm are used to generalise object references and create parameterised operator schema.

A Description of the *DetermineStates* Procedure. To illustrate the workings of the Procedure, we will use the example tyre domain solution sequence to form the initial stage of an example walk-though. Let us consider $A(1) - A(4)$ = `do_up`, `jack_down`, `tighten`, `apply_trim` as given above. The algorithm is as follows:

Procedure ***DetermineStates***
In:
$\mathcal{PDM}$,
I, F are maps of objects to their Initial, Final state, resp.
$T = A(1)..A(N)$: training sequence of N actions
Out:
maps $C_i, i = 1, \ldots, N+1$, mapping from object names
to object states such that action $A(i)$ of T
represents a transition from C_i to C_{i+1}
Define $A.c$ to be the set of $A.obj$'s changing objects
1. $C_1 := I; C_{N+1} := F$;
2. for each $i \in 1, \ldots, N$
3. for each object $O \notin A(i).c$
4. $C_{i+1}(O) := C_i(O)$;
5. end for
6. for each object $O \in A(i).c$
7. if $O \notin A(i+1).c \cup \ldots \cup A(N).c$ then
8. $C_{i+1}(O) := F(O)$
9. else
10. **choose** $C_{i+1}(O) :=$ any legal state using $\mathcal{PDM}$
11. with parameters bound to objects in $A(i)$ or $C_i(O)$
12. test the choice using the following constraints
13. – $C_{i+1}(O) \neq C_i(O)$
14. – the transition $C_i(O) \Rightarrow C_{i+1}(O)$
15. must be consistent with transitions at
16. previous occurrences of $A(i).name$
17. end if
18. end for
19. test that the conjunction of $C_{i+1}(O)$ for all O
20. is consistent with $\mathcal{PDM}$'s invariants
21. end for

In **Line 1** the first and last components of the map C are initialised to be the same as the initial and final state respectively. The algorithm then iterates for all actions in the sequence. When $i = 1$, **Lines 3-5** define C_2 as the same as C_1 when applied to non-changing objects in the domain. **Lines 6-18** attempt to determine the rest of map C_2 where it is applied to objects that change as a result of the execution of $A(1)$. **Line 6** identifies the changing objects (`hub1` and `nuts1`) - let us consider `hub1`. **Lines 7-8** look ahead to see if `hub1` will not change again in a subsequent action and find that it does in the second action in the sequence. If we had chosen an example where the object does not change again after the first action then **Line 8** would set the object's state to be the

final state. Considering **Line 10**, using the partial domain model there are four potential values for C_2(`hub1`):

```
a. [on_ground(hub1),fastened(hub1)]
b. [free(hub1),jacked_up(hub1,jack0),unfastened(hub1)]
c. [jacked_up(hub1,jack0),fastened(hub1)]
d. [unfastened(hub1), jacked_up(hub1,jack0)]
```

Lines 12-16 of the algorithm determine which of these states is appropriate. The constraint in **Line 12** makes sure the new object state is different from the last. `hub1`'s current state is `[unfastened(hub1), jacked_up(hub1,jack0)]`, so this eliminates d. **Lines 13-14** checks that an object state has no unreferenced parameters (if part of the state description references an object not taking part in the transition, then that state would be inappropriate). This does not eliminate any of the choices in the example. **Lines 15-16** check that the union of all the chosen states (in this case incorporating choices for `hub1` and `nuts1`) are consistent. Using these constraints, the states a. and b. are eliminated, leaving c. to be chosen as the value of C_2(`hub1`).

To complete the $Opmaker$ process, once the state space map C has been determined, operator instances are constructed by creating *prevail* components for each unchanging object, and creating *necessary transitions* for each object that is changed by an action. These instances are generalised to schema on the basis that each object in a sort behaves the same, and can be replaced by a *sort parameter*. The systems stores the definition of the operator schema and checks them against any previous definition. Finally, a method is generated by combining the induced operator schema, using the original $Opmaker$ code.

Non-deterministic choices in the selection of an object's state expression, and the binding of the variables in the object state expression (**Lines 10-11**) mean that sometimes the new state cannot be uniquely determined. However, we have found that this depends on the strength of the invariants that are supplied with the $\mathcal{PDM}$.

3 Experiments and Results

$Opmaker2$ has been implemented in Sicstus Prolog incorporating the algorithm detailed above. We use the same experimental approach that was used to test the original $Opmaker$ system, which was to:

1. Compose training tasks and solution sequences from a range of domains that have already been captured within a hand-crafted model. The set of training tasks should contain at least one instance of each action in the domain, and each task is selected on the basis of whether it is likely to form building blocks for the solution of more complex tasks. The (initial) partial domain model input into $Opmaker2$ is the hand crafted domain without its operator schema.
2. Use $Opmaker2$ to induce operator schema and methods from the training tasks and solution sequences, and the partial domain model.

3. Use a planning engine to check that the automatically acquired operator schema can solve the same set of problems that the hand-crafted set has been applied to.
4. Use a planning engine to compare performance of the old hand-crafted action schema versus the induced schema and methods. In this case HyHTN [8], a HTN planner which can take advantage of the induced methods, was used. For a comparison with a planner which uses only operator schema (without methods), we use Hoffmann's FF planner [9]. [1]

Success is judged using the following kinds of criteria:

1. Uniqueness: is a set of unique operator schema acquired from the training tasks and the partial domain model that originated from the hand crafted domain model? Or, more subtly, can $Opmaker2$ induce unique schema without having to encode many invariants into the domain models?
2. Validity: Can a set of operator schema output from $Opmaker2$ be used by a planner to solve the same tasks that the original training sequences were aimed at?
3. Efficiency: Is it more efficient, in terms of planning time, to solve tasks using $Opmaker2$ defined operator schema and methods, rather than the original hand-crafted operators?

We detail the results for the extended tyre domain below, and describe other domains on which we have experimented. More details can be found in a recent doctoral thesis [10].

Results in the Extended Tyre Domain. The handcrafted version of the extended tyre domain has 26 objects in 9 sorts, with 22 operators. We engaged a researcher (who was not the author of $Opmaker2$ software) to create 7 sequences of tasks of between 2 and 5 actions in length, encapsulating useful subtasks such as taking a wheel off a hub, or bringing tools out of the car's boot. When input to $Opmaker2$ with the initial partial domain model, procedure *DetermineStates* did not have enough information to discover unique sequences of states for all objects in the training sequences. However, adding extra 'common sense' invariants to the partial domain model (shown in the appendix) was sufficient to allow *DetermineStates* to generate a unique set of state sequences, leading to a set of 22 operator schema generated [10]. An example follows:

```
operator(putaway_jack(Container1,Jack2),
   [(container,Container1,[open(Container1)])],
   [(jack,Jack2,[have_jack(Jack2)]
   =>[jack_in(Jack2,Container1)])],
   []
).
```

On inspection, these were identical in structure to the original hand crafted version. This was confirmed by running the full domain model with a planner and ensuring that all tasks were correctly solved. In addition to operators, the 7 sequences of training tasks

[1] We use the GIPO tool to translate the generated OCL domain models into PDDL (the strips version with typing, equality, conditional effects) so that they can be input to generally available planners.

input lead to 7 methods being output. For example, one of the 7 generated methods encapsulating solution heuristics is as follows:

```
method(ex_putaway_tools(Boot,Jack0,Wrench0),
    % Dynamic constraints
    [],
    % Necessary transitions
    [(container,Boot,[open(Boot)]=>[closed(Boot)]),
    (jack,Jack0,[have_jack(Jack0)]=>[jack_in(Jack0,Boot)]),
    (wrench,Wrench0,[have_wrench(Wrench0)]
    =>[wrench_in(Wrench0,Boot)])],
    % Temporal constraints
    [before(1,2),before(2,3)],
    % Static constraints
    [],
    % Decomposition
    [putaway_wrench(Boot,Wrench0),
    putaway_jack(Boot,Jack0),
    close_container(Boot)
    ]).
```

Generating plans up to 10-12 operations in length was possible with standard planning engines, but tasks demanding solutions of greater length were not possible with the planning engines at our disposal. However, when the induced operator schema and the methods were used together with HyHTN, plan times were significantly shorter. For example, a complex planning problem for this extended domain is paraphrased as: "A car has two flat tyres: one is intact and can be fixed by use of the pump, whilst the other is punctured and requires a full tyre change". No solution was found to this problem after 36 hours using FF or HyHTN without the induced methods. However, using the induced domain schema and methods a correct solution of length 24 was found by HyHTN after only 11 seconds. It is not surprising that HTN-type domain models are so efficient: this is supported by fielded planning applications. What is significant here is that both the operator schema and the HTN-type methods used in the domain model were generated by $Opmaker2$.

Experiments with other Domains. We experimented with an OCL encoding of a Blocks Domain, with 7 blocks stacked on a table. 6 action names were devised and one long training sequence that solved the following task was created: given a set of seven blocks stacked bottom to top `block1` to `block7`, use a gripper to move one block at a time until the blocks are in two stacks. The order of the blocks in these stacks (bottom to top) is `block6`, `block2`, `block4` form first stack; `block5`, `block1`, `block7`, `block3` form the other stack. A 22 solution sequence was composed and fed into $Opmaker$ in 6 separate batches, to enable methods and operator schema to be induced. With the original partial domain model enhanced with 4 invariants, 6 operator schema were output by $Opmaker2$. These operators were identical in structure to the hand-coded ones for this domain, and can be used operationally by planning engines. Table 1 shows that the overall task can be tackled in chunks (Tasks 1 - 6), as well as in one sequence (task 7). Each of the 7 tasks resulted in unique and accurate operator schema.

Table 1. Operator Testing in Blocks World Full Problem

Task No.	*Actions*	*Operator Schema*
1	4	2
2	2	2
3	2	2
4	4	4
5	2	2
6	2	2
7	22	6

The Hiking Domain was used to illustrate the original $Opmaker$, and models 'lazy' hikers (recreational walkers) who use two cars to carry their equipment around a long (several day) circular route. Automated planning is used to work out the logistics of where to leave their cars, put up their tent, transport their luggage etc. For $Opmaker2$ to produce an accurate, unique set of operator schema, the partial domain model required one extra invariant to strengthen it sufficiently. This compares well with the original use of the domain [2] which required a fairly laborious interactive session before outputting operator schema.

4 Related Work

Many machine learning systems are driven by the input of both positive and negative examples. Whilst it was thought to be advantageous to use both kinds of example, many systems like $Opmaker2$, use only positive examples. In particular Vere's [11] Maximal Unifying Generalisation (1987), and Wang's [12] OBSERVER system learn from just positive examples, whilst Grant's [13] POI system learns from positive examples and uses a default rule to provide negative information which boosts the positive training instances. $Opmaker2$ is similar: it uses positive examples in the solution sequence and it makes deductions from the partial domain model.

Learning expressive theories from examples is a central goal in the Inductive Logic Programming community. In his thesis [14], Benson describes an ILP method for learning more expressive operator schema than $Opmaker2$, using multiple examples. However, the focus of $Opmaker2$ is to learn from (ideally) one example sequence, and to learn heuristics as well as operator structure.

Perhaps closest to our work is ARMS [15], a system in which operators are learned without the need for user intervention. Further work by these authors [16] involves learning recursive HTN structures. The authors focus on matching sub-sequences to tasks assuming no knowledge of observed states achieved by low-level actions. The output consists of pairs of action sequences and the high-level tasks achieved by them. As with our system they begin with solution sequences of defined tasks, and compare learned methods to hand-crafted ones to judge success. Whilst ARMS does not require a partial domain model, it requires many training sets (about 40 training sets is quoted). Once learned they were fine tuned by domain experts by hand. By contrast

our system does not require multiple examples, as we focus on an expert transferring heuristic knowledge encapsulated in a handful of well chosen examples solution sequences.

5 Conclusions

In this paper we have set up a knowledge acquisition problem which is very relevant to tackling the central problem of using AI planning engines - the acquisition of formulations of actions (in the form of operator schema), and acquisition of heuristics (in the form of HTN-type methods). Our work and the results reported here depend on a structured view of partial domain knowledge about objects being available. Whereas in propositional, classical planning (epitomised by the PDDL language [4]), states are fairly arbitrary sets of propositions, we assume that the space of states is restricted in that objects are pre-conceived to occupy a fixed set of plausible states. Within this framework, we have described a method for inducing operator 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 operator set. Further, our results give some evidence that the methods induced with the operator schema lead to more efficient domain models.

$Opmaker2$ is an improvement on $Opmaker$ in that it 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. Our experimental results show, however, that partial domain models may have to be strengthened with extra invariants before a unique set of operator schema can be synthesised. Hence, we could summarise our work as arguing for the creation of planning domain models by the crafting of a strong partial domain model, and a set of training tasks, *rather* than crafting operator schema and planning heuristics manually.

There are several directions for future work:

1. Can our work be extended to capturing domains with durative or probabilistic actions, or other, more expressive formulations for action? What extra details would be required as input to the operator induction process?
2. Can the $Opmaker2$ system be extended to deal with model maintenance (for instance by incremental learning), so that old operator schema can be refined in the presence of new example solution sequences?
3. What resilience does our approach offer in the face of errors in training tasks or in the partial domain model?

Finally, we believe that this line of research is essential if *intelligent agents* are to have general planning capabilities. If this is to be the case, it seems unlikely that intelligent agents will always rely on human experts to encode and maintain their knowledge. It seems reasonable that they would need the capability to acquire knowledge of actions themselves, perhaps by observing the actions of other agents, and using pre-existing static domain knowledge, to induce operator schema and domain heuristics.

References

1. Simpson, R.M., Kitchin, D.E., McCluskey, T.L.: Planning Domain Definition Using GIPO. Journal of Knowledge Engineering 1 (2007)
2. McCluskey, T.L., Richardson, N.E., Simpson, R.M.: An Interactive Method for Inducing Operator Descriptions. In: The Sixth International Conference on Artificial Intelligence Planning Systems (2002)
3. McCluskey, T.L., Porteous, J.M.: Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. Technical Report RR9606, School of Computing and Maths, University of Huddersfield (1996)
4. AIPS-98 Planning Competition Committee: PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
5. Liu, D., McCluskey, T.L.: The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield (2000)
6. Chien, S.A. (ed.): 1st NASA Workshop on Planning and Scheduling in Space Applications. NASA, Oxnard, CA (1997)
7. Russell, S.J.: Execution architectures and compilation. In: Proc. IJCAI (1989)
8. McCluskey, T.L., Liu, D., Simpson, R.M.: GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. In: Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (2003)
9. Hoffmann, J.: A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In: Proceedings of the 14th Workshop on Planning and Configuration - New Results in Planning, Scheduling and Design (2000)
10. Richardson, N.E.: An Operator Induction Tool Supporting Knowledge Engineering in Planning. PhD thesis, School of Computing and Engineering, University of Huddersfield, UK (2008)
11. Vere, S.: In Pattern Directed Inference Systems. Academic Press, New York (1978)
12. Wang, X.: Learning Planning Operators by Observation and Practice. PhD thesis, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsberg, PA 15213 (1996)
13. Grant, T.J.: Inductive Learning of Knowledge-Based Planning Operators. PhD thesis, de Rijksuniversiteit Limburg te Maastricht, Netherlands (1996)
14. Benson, S.S.: Learning Action Models for Reactive Autonomous Agents. PhD thesis, Dept. of Computer Science, Stanford University (1996)
15. Wu, K., Yang, Q., Jiang, Y.: Arms: Action-relation modelling system for learning acquisition models. In: Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, California, USA (2005)
16. Yang, Q., Pan, R., Pan, S.J.: Learning recursive htn-method structures for planning. In: Proceedings of the ICAPS 2007 Workshop on Artificial Intelligence Planning and Learning (2007)

APPENDIX

```
% Sorts
sorts(primitive_sorts,[container,nuts,hub,
 pump,wheel, wrench,jack,wheel_trim,tyre]).

% Objects
objects(container,[boot]).
objects(nuts,[nuts1,nuts2,nuts3,nuts4]).
```

```
objects(hub,[hub1,hub2,hub3,hub4]).
objects(pump,[pump0]).
objects(wheel,[wheel1,wheel2,
               wheel3,wheel4,wheel5]).
objects(wrench,[wrench0]).
objects(jack,[jack0]).
objects(wheel_trim,[trim1,trim2,trim3,trim4]).
objects(tyre,[tyre1,tyre2,tyre3,tyre4,tyre5]).

% Predicates
predicates([ closed(container),open(container),
 tight(nuts,hub),loose(nuts,hub),have_nuts(nuts),
 on_ground(hub),fastened(hub),jacked_up(hub,jack),
 free(hub),unfastened(hub),have_pump(pump),
 pump_in(pump,container),have_wheel(wheel),
 wheel_in(wheel,container),wheel_on(wheel,hub),
 have_wrench(wrench),wrench_in(wrench,container),
 have_jack(jack),jack_in_use(jack,hub),
 jack_in(jack,container),trim_on(wheel_trim,wheel),
 trim_off(wheel_trim),fits_on(tyre,wheel),
 full(tyre),flat(tyre),punctured(tyre)]).

% Object State Expressions
substate_classes([
 container(C,[[closed(C)], [open(C)] ]),
 nuts(N,[[tight(N,H)],[loose(N,H)],[have_nuts(N)]]),
 hub(H, [[on_ground(H),fastened(H)],
    [jacked_up(H,J),fastened(H)],
    [free(H),jacked_up(H,J),unfastened(H)],
    [unfastened(H),jacked_up(H,J)] ]),
 pump(Pu, [[have_pump(Pu)],[pump_in(Pu,C)] ]),
 wheel(Wh, [[have_wheel(Wh)],[wheel_in(Wh,C)],[wheel_on(Wh,H)]]),
 wrench(Wr,[[have_wrench(Wr)],[wrench_in(Wr,C)]]),
 jack(J,[[have_jack(J)],[jack_in_use(J,H)],[jack_in(J,C)] ]),
 wheel_trim(WT,[[trim_on(WT,Wh)],[trim_off(WT)]]),
 tyre(Ty, [[full(Ty)],[flat(Ty)],[punctured(Ty)]]) ]).

% Invariants
atomic_invariants([  fits_on(tyre1,wheel1),
  fits_on(tyre2,wheel2), fits_on(tyre3,wheel3),
  fits_on(tyre4,wheel4), fits_on(tyre5,wheel5)]).
invariant( all(H:hub,fastened(H)<==>
  ex(N:nuts,tight(N,H)\/loose(N,H))) ).
invariant( all(H:hub,all(J:jack,jack_in_use(J,H)
  <==>jacked_up(H,J))) ).
invariant( all(H:hub,~free(H)<==>ex(W:wheel,wheel_on(W,H))) ).
invariant(
 all(T:wheel_trim,all(W:wheel,trim_on_wheel(T,W)
 <==>trim_on(W,T))) ).
%Hub may only have one set of nuts attached
```

```
invariant(all(H:hub,all(N1:nuts,all(N2:nuts,
 (tight(N1,H)\/loose(N1,H)) /\
 (tight(N2,H)\/loose(N2,H))==>(N1=N2) ))) ).
%Hub may only have one wheel attached.
invariant( all(H:hub,all(W1:wheel,all(W2:wheel,
 wheel_on(W1,H)/\wheel_on(W2,H)==>(W1=W2) ))) ).
%If nuts are tight then hub must be on the ground.
invariant( all(H:hub, ex(N:nuts,tight(N,H))==>on_ground(H))).
%If a trim is on a wheel, then the wheel is on
% a hub and the nuts are tight.
invariant(
 all(W:wheel,ex(T:wheel_trim,trim_on_wheel(T,W))
  ==>
 ex(H:hub,wheel_on(W,H)/\ex(N:nuts,tight(N,H))))).
```