Towards inducing HTN domain models from examples (short paper)

N. E. Richardson, T. L. McCluskey, M. M. West
School of Computing and Engineering
Department of Informatics
The University of Huddersfield,
Huddersfield, HD1 3DH, UK
n.e.richardson@hud.ac.uk, t.l.mccluskey@hud.ac.uk, m.m.west@hud.ac.uk

Abstract

Domain modelling for AI Planning can be a complex process especially if there is a large number of objects or actions or both to be modelled. This task can be facilitated by tools which induce operators or methods from examples. Further, large and complex domains are more easily constructed if domain languages are used which allow for hierarchical decomposition of domain components. Examples of such a decomposition are *object class hierarchies* and *method hierarchies*. This paper describes ongoing work which aims to produce algorithms which learn effective hierarchical decompositions from examples.

Introduction

Domain modelling is a complex, error prone process, especially when the model is complex. Capturing dynamics and behaviour using operator structures (or compositions of operators called *methods*) lies at the heart of constructing planning domains. One way to facilitate the process is to use tools which induce operators or methods using task solutions as training examples. In our previous work we have shown how 'flat' domain operators can be induced from examples. Operators can be induced using *opmaker* (McCluskey, Richardson, & 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 & McCluskey 2000) or PDDL.

Large and complex domains are more easily constructed if domain languages are used which allow for hierarchical decomposition of domain components. This makes for a richer language which more closely captures the real world situations. Methods composed of hierarchical task networks (HTN) make better sense of these worlds but are, however, difficult to construct. We are working on an extension of the induction process whereby operators are combined into task networks. To illustrate the techniques we are using we have created a hierarchical version of the familiar *briefcase* domain. Below we briefly describe this work towards creating procedures which input training sequences and a partial model containing object and class information, and outputs an HTN domain model.

Hierarchical Domains

To illustrate our method we have created a version of the familiar briecase world containing a simple structural hierarchy of object "sorts" shown in figure 1. The tree shows the hierarchical sort structure with predicates attached at appropriate levels. For example inheritance in the sort tree means that the state at_carrier applies not only to carrier but to any other sort below it on the tree. The converse does not work so that goes_in applies to box, lunch_box and pencil_box only, and not to carrier or bag.

In the *OCL* language planning domains may be hierarchical in two ways. The language structures the objects to be members of certain types called 'sorts'. For example in the hierarchical briefcase domain (HBC)

```
sorts(carrier,[bag,box]).
sorts(bag,[briefcase,suitcase]).
objects(briefcase,[bc1]).
objects(suitcase,[sc1]).
```

describes how bag (and box) are of sort 'carrier', whilst briefcase is of sort 'bag' and 'bc1' is a specific object of sort briefcase. The second example of the hierarchical nature of domains involves the methods. Methods are constructed because the sequence of actions they perform need to be packaged together for efficiency and/or effectiveness, and hence they encapsulate domain heuristics. They can be thought of as 'mini-plans' where a plan is a sequence of actions to achieve the state changes from a specified initial state to some predetermined goal state. Methods are structured into hierarchies so that some methods decompose into others or decompose into both other methods and primitives in order to complete their task. This structure in complex domains can be quite extensive and it can be difficult to see the interlacing of tasks.

Work in Progress

Using partial domain models we have been able to replicate the *GIPO*-constructed operators and methods by induction as follows. For each method in the *GIPO*-constructed domain we have compiled a file of example material including the partial domain (containing an object class hierarchy) but excluding all the operators and methods. For each notional task the files each contain a solution in the form of a named operator sequence, initial states for the objects involved and numbered example material indicating the states after the

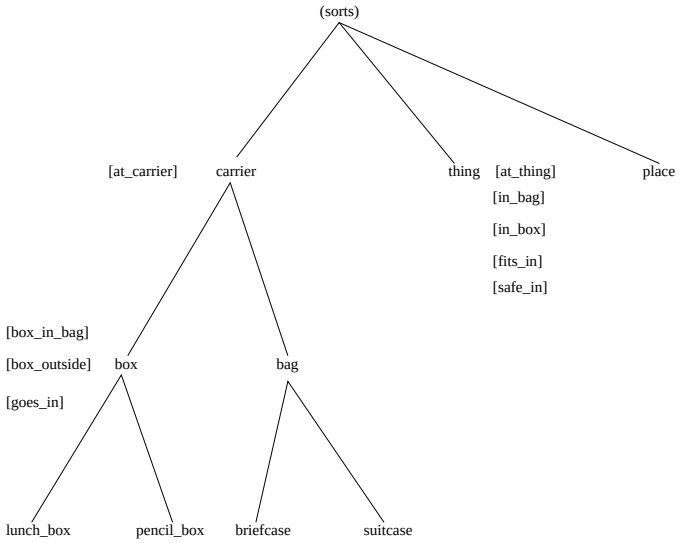

Figure 1: The Sort-Tree Showing the Levels at which Predicates Apply

application of each operator. We can think of this as using a *linear* plan to induce the operators and provide the decomposition for the method. At this stage we are choosing material for the example files carefully so that methods do not overlap their tasks but with the aim of building method hierarchies. *opmaker*, described in greater detail in (McCluskey, Richardson, & Simpson 2002), induces operator headings from those listed in the operator sequence and forms state transitions. The left hand side of any induced transition comes from either the list of initial states in the example file or from the altered state of a previously induced operator. The right hand sides of the induced transitions come from the numbered example inputs. (These can be null and induce a prevail transition.)

The induction algorithm outputs, for each example file, a set of instantiated operators and an HTN method induced from the sequence. In each case only those operators required for the methods we were replicating were induced from each file. An example induced operator put_box_in_bag is as follows.

```
operator(put_box_in_bag(Bag,Place,Box),
    %prevail
    [se(bag,Bag,[at_carrier(Bag,Place)])],
    %necessary
    [sc(box,Box,[box_outside(Box),
    at_carrier(Box,Place)] =>
    [box_in_bag(Box,Bag),
    at_carrier(Box,Place),
    goes_in(Box,Bag)])],
    %conditional
    [sc(thing,Thing,[in_box(Thing,Box),
    at_thing(Thing,Place)] =>
    [in_box(Thing,Box),
    at_thing(Thing,Place),
    safe_in(Thing,Box)])]).
```

Here the operator header lists the sorts of objects involved in the action and the prevail transition states that the bag remains at the same place. The necessary transition states that the box changes state from being outside the bag at a place to being inside the bag at the same place. Finally the conditional transition states that if a thing is in the box then it also undergoes a state change - in this case it is still in the box but the box is now in the bag.

A simple method operator induced is shown below. The task network is composed of two induced operators put_in_box and put_box_in_bag.

```
method(pack_lunch(Sandwiches,Place,
  Lunch_box,Bag),
    % pre-condition
    [],
    % Index Transitions
    [sc(thing,Sandwiches,
      [outside(Sandwiches),
       at_thing(Sandwiches,Place)]=>
      [in_box(Sandwiches,Lunch_box),
       at_thing(Sandwiches,Place)]),
     sc(lunch_box,Lunch_box,
      [box_outside(Lunch_box),
       at_carrier(Lunch_box,Place)]=>
      [box_in_bag(Lunch_box,Bag),
       at_carrier(Lunch_box,Place)])],
    % Static
    [safe_in(Thing,Lunch_box),
     goes_in(Lunch_box,Bag)],
    % Temporal Constraints
    [before(1,2)],
    % Decomposition
    [put_in_box(Box,Place,Thing),
     put_box_in_bag(Bag,Place,Box)]).
```

This format allows for any preconditions to be listed and the main transitions for the lunch_box and sandwiches are listed. The decomposition names the two operators of which this method is composed and the temporal constraints name

the order in which they must be applied.

The new operators and methods have been compared to the hand constructed set (using *GIPO*). Our initial results show that the induced sets are accurate: when used with *GIPO*'s planner and stepper tools we were able to complete all the tasks previously declared for the domain. Preliminary tests (with GIPO's planner HyHTN) show that plans formed using just induced operators run faster than those formed using both induced methods and operators but this may be because of the simplicity of the domains used. Further tests using more complex domains such as the Tyre World indicate that as the tasks become more complex, inducing methods as well as operators improves planning efficiency.

We hope to be able to demonstrate that planning is enhanced by the use of induced method hierarchies in our future work. We aim to show that as methods are learned and new methods are induced that utilise them, we can build induced method hierarchies for more complex real world situations.

Related Work

In (Ilghami, Nau, & Munoz-Avila 2006) the hierarchical domain learner (HDL) begins with an empty set of known methods. By examining plan traces and forming these into method decompositions HDL incrementally adds methods on the basis that a new method is created from the plan trace if its decomposition is different from those of other methods previously added. Inputs to HDL include a set of primitive operators and the plan traces. This differs from our work which induces both operators and a method from the example material, where the method decomposition is the ordered set of primitives induced.

In the work of (Nejati, Langley, & Konik 2006) hierarchical task networks are learned by analysing expert traces. They start from having a set of operators and a worked-out problem solution which includes a specified sequence of operators and thus differs from our system in having an original operator set.

The argument for more complex, structured operators to be used to model the difficulties faced when in real world situations is well put by Levine and DeJong (Levine & DeJong 2006). Their solution to the problem is similar to ours and they introduce a system of automatically constructing planning operators. The difference is that they shield the planner from all but the necessary elements which are visible to the planner.

In (Garland, Ryall, & Rich 2001) Garland Ryall and Rich show, in their Collagen system, that learning task models can be achieved by training examples and support from a domain expert. Their work is similar to our approach in the following ways:

- their 'task models' are similar to our HTN methods
- they show a complete recipe to achieve some task
- they show orderings of the steps to achieve the task
- they are developing a graphical user interface to aid construction
- the orderings of the steps contain primitive and non-primitive stages
- they list constraints that apply to the various steps
- user/expert guidance is required for the detail.

A more recent system that learns operators from examples is ARMS (Wu, Yang, & Jiang 2005). This system learns operator specifications without the need for user intervention or a partial domain specification. However, it requires many training examples containing valid solution sequences, and presently it is only capable of inducing "flat" operators.

References

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

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.

Levine, G., and DeJong, G. 2006. Explanation-based acquisition of planning operators. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, 152 – 161.

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.; 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*.