Acquisition of Object-Centred Domain Models from Planning Examples

S. N. Cresswell and **T. L. McCluskey** and **M. M. West**
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
{s.n.cresswell,t.l.mccluskey,m.m.west}@hud.ac.uk

Abstract

The problem of formulating knowledge bases containing action schema is a central concern in knowledge engineering for AI Planning. This paper describes *LOCM*, a system which carries out the automated induction of action schema from sets of example plans. Each plan is assumed to be a sound sequence of actions; each action in a plan is stated as a name and a list of objects that the action refers to. *LOCM* exploits the assumption that actions change the state of objects, and require objects to be in a certain state before they can be executed. The novelty of *LOCM* is that it can induce action schema without being provided with any information about predicates or initial, goal or intermediate state descriptions for the example action sequences. In this paper we describe the implemented *LOCM* algorithm, and analyse its performance by its application to the induction of domain models for several domains. To evaluate the algorithm, we used random action sequences from existing models of domains, as well as solutions to past IPC problems.

Introduction

Formulating and maintaining knowledge bases containing action specifications is considered a central challenge in knowledge engineering for AI Planning. In particular, a problem facing AI is to overcome the need to hard-code and manually maintain action representations within deliberative agents (a problem which limits their autonomy). There is a need for many types of knowledge acquisition in planning: depending on the application and available tools, the domain model may be defined by a domain expert, by a planning expert, or synthesised from existing formalisms, or assembled from example plan traces. In this paper we describe *LOCM*, an implemented algorithm that induces a planning domain model from example plan traces. We analyse its performance by its application to action sequences from several domains.

Planning traces are input into *LOCM* as 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. Each plan is assumed to be a sound sequence of actions. This means that all the preconditions of every action could be met at the point it is executed, either by earlier actions or some initial state.

Working under the assumptions of Simpson et al's object-centric view of domain models (Simpson, Kitchin, and McCluskey 2007), we assume that a planning domain consists of sets (called $sorts$) of object instances, where each object behaves in the same way as any other object in its sort. In particular, sorts have a defined set of states that their objects can occupy, and an object's state may change (called a state transition) as a result of action instance execution. *LOCM* works by assembling the transition behaviour of individual sorts, the co-ordinations between transitions of different sorts, and the relationships between objects of different sorts. It does so by exploiting the idea that actions change the state of objects, and that each time an action is executed, the preconditions and effects on an object are the same. Under these assumptions, *LOCM* can induce action schema without the need for background information such as specifications of initial/goal states, intermediate states, fluents or other partial domain information. All other current systems e.g. *Opmaker*2 (McCluskey et al. 2009), ARMS (Wu, Yang, and Jiang 2005), and the SLAF approach (Shahaf and Amir 2006) require some of this background knowledge as essential to their operation.

The *LOCM* System

LOCM Inputs and Outputs

The inputs to *LOCM* are a set of sound sequences of action instances. Using the well known *tyre-world* as an example, the following is a sequence containing four action instances, where an action is a name followed by a sequence of affected objects:

open(c1); fetch_jack(j,c1); fetch_wrench(wr1,c1); close(c1);

These sequences may be supplied by a trainer, observed from an operational planning system, or they could be generated from an existing solver and domain model (in part of the empirical evaluation below we have used a random walk generator to supply example sequences). The trainer is expected to include references to all objects that are needed for each action to be carried out.

The output of *LOCM* (given sufficient examples) is a domain model consisting of sorts, object behaviour defined

by state machines, predicates defining associations between sorts, and action schema in solver-ready form.

The *LOCM* Method

Phase 1: Extraction of state machines In our approach, an object of any given sort occupies one of a fixed set of parameterised states. In Phase 1, we assume an object's state can be defined without parameters (parameters specify associations with other objects). *LOCM* starts by first collecting the set of all transitions occurring in the example sequences. A transition is defined by a combination of action name and action argument position. For example, an action *fetch_wrench(wr1,cntnr)* gives rise to two transitions: *fetch_wrench.1*, and *fetch_wrench.2*. Each transition describes the state change of objects of a single sort in isolation. For every transition occurring in the example data, a separate *start* and *end* state are generated. The trajectory of each object is then tracked through each training sequence. For each consecutive pair of transitions T_1, T_2, experienced by an object Ob, we assume that $T_1.end = T_2.start$.

Using a training set from the tyre world, suppose some object $c1$ goes through a sequence of transitions given in the example introduced above:

open(c1); fetch_jack(j,c1); fetch_wrench(wr1,c1); close(c1);

Let us assign names S_1 to S_8 to the states of $c1$ as it is affected by each transition:

$$\begin{array}{lll} S_1 & \Longrightarrow open.1 \Longrightarrow & S_2 \\ S_3 & \Longrightarrow close.1 \Longrightarrow & S_4 \\ S_5 & \Longrightarrow fetch\_jack.2 \Longrightarrow & S_6 \\ S_7 & \Longrightarrow fetch\_wrench.2 \Longrightarrow & S_8 \end{array}$$

Using the example action sequence, and the constraint on consecutive pairs of transitions, we can then deduce that $S_2 = S_5$, $S_6 = S_7$, $S_8 = S_3$. Suppose our example set contains another action sequence:

open(c2); fetch_wrench(wr1,c2); fetch_jack(j,c2); close(c2);

We deduce that $S_2 = S_7$, $S_8 = S_5$, $S_6 = S_3$, and hence $S_2, S_3, S_5, S_6, S_7, S_8$ all refer to the same state. If additionally we have the sequence:

close(c3); open(c3);

then $S_4 = S_1$ can be deduced, and hence we have tied together individual states to partially construct a state machine for containers (Fig. 1). A more formal description of the algorithm follows [1]:

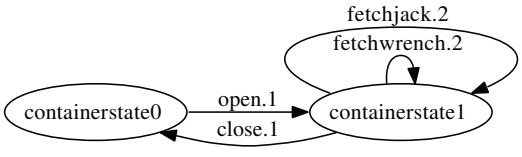

Figure 1: An incomplete state machine for containers in tyre-world

procedure **LOCM_I** (Input action sequence Seq)
For each combination of action name A and
  argument pos P for actions occurring in Seq
  Create transition $A.P$, comprising
    new state identifiers $A.P.start$ and $A.P.end$
    Add $A.P$ to the transition set TS
Collect the set of objects Obs in Seq
  For each object Ob occurring in Obs
    For each pair of transitions T_1, T_2
        consecutive for Ob in Seq
      Equate states $T_1.end$ and $T_2.start$
  end
end
Return TS, transition set
       OS, set of object states remaining distinct

At the end of phase 1, *LOCM* has derived a set of state machines, each of which can be identified with a sort.

Phase 2: Identification of state parameters Each state machine describes the behaviour of a single object in isolation, without consideration of its association with other objects e.g. it can distinguish a state of a wrench corresponding to being *in some container*, but does not make provision to describe *which* container it is in.

In the object-centred representation, the dynamic associations between objects are recorded by *state parameters* embedded in each state. Phase 2 of the algorithm identifies parameters of each state by analysing patterns of object references in the original action steps corresponding to the transitions. For example, consider the state *wrench_state0* for the *wrench* sort (Fig. 2). Considering the actions for *putaway_wrench(wrench,container)*, and *fetch_wrench(wrench,container)*. For a given wrench, consecutive transitions *putaway_wrench*, *fetch_wrench*, in any example action sequence, always have the same value as their *container* parameter. From this observation, we can induce that the state *wrench_state0* has a state variable representing *container*. The same observation does not hold true for *wrench_state1*. We can observe instances in the training data where the wrench is fetched from one container, and put away in a different container.

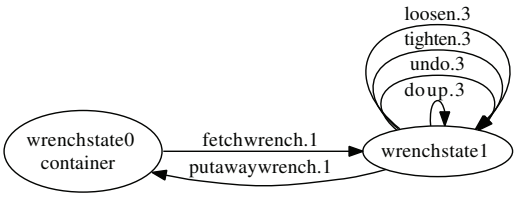

Figure 2: Parameterised states of wrench.

[1] Whereas our system is designed to use multiple training sequences. To simplify, the presentation here uses only a single sequence.

This second phase of the algorithm performs inductive learning such that the hypotheses can be refuted by the examples, but never definitely confirmed. This phase generally requires a larger amount of training data to converge than Phase 1 above. Phase 2 is processed in three steps, shown below in the algorithmic description. The first two steps generate and test the hypothesised correlations in action arguments, which indicate the need for state parameters. The third step generates the set of induced state parameters.

procedure **LOCM_II** (Input action sequence Seq,
Transition set TS, Object set Obs)
Object state set OS)
Form hypotheses from state machine
For each pair $A_1.P_1$ and $A_2.P_2$ in TS
such that $A_1.P_1.end = S = A_2.P_2.start$
For each pair $A_1.P_1'$ and $A_2.P_2'$ from TS and S in OS
with $A_1.P_1'.sort = A_2.P_2'.sort$
and $P_1 \neq P_1', P_2 \neq P_2'$
(i.e. a pair of the other arguments
of actions A_1 and A_2 sharing a common sort)
Store in hypothesis set HS the hypothesis
that when any object *ob* undergoes sequentially
the transitions $A_1.P_1$ then $A_2.P_2$,
there is a single object ob',
which goes through both of the corresponding
transitions $A_1.P_1'$ and $A_2.P_2'$
(This supports the proposition that state S
has a state parameter which can record
the association of *ob* with ob')
end
end
Test hypotheses against example sequence
For each object Ob occurring in Obs
For each pair of transitions $A_1.P_1$ and $A_2.P_2$
consecutive for Ob in Seq
Remove from hypothesis set HS any hypothesis
which is inconsistent with example action pair
end
end
Generate and reduce set of state parameters
For every hypothesis remaining in HS
create the state parameter supported by the hypothesis
Merge state parameters on the basis that
a transition occurring in more than one transition pair
is associated with the same state parameter in each occurrence
end
return: state parameters and correlations with action arguments

Phase 3: Formation of action schema Extraction of an action schema is performed by extracting the transitions corresponding to its parameters, similar to automated action construction in the OLHE process in (Simpson, Kitchin, and McCluskey 2007). One predicate is created to represent each object state. The output of Phase 2 provides correlations between the action parameters and state parameters occurring in the start/end states of transitions. For example, the generated *putaway_wrench* action schema is:

```
(:action putaway_wrench
 :parameters (?wrench1 - wrench ?container2 - container)
 :precondition (and (wrench_state1 ?wrench1)
                    (container_state1 ?container2))
 :effect (and (wrench_state0 ?wrench1 ?container2)
              (not (wrench_state1 ?wrench1))))
```

The generated predicates *wrench_state0*, *wrench_state1*, *container_state1* can be understood as *in_container*, *have_wrench* and *open* respectively. The generated schema can be used directly in a planner.

Evaluation of *LOCM*

LOCM has been implemented in Prolog incorporating the algorithm detailed above. Here we attempt to analyse and evaluate it by its application to the acquisition of existing domain models. We have used example plans from two sources: (1) Existing domains built using GIPO III. In this case, we have created sets of example action sequences by random walk. (2) Domains which were used in IPC planning competitions. In this case, the example traces come from solution plans in the publicly released competition solutions.

We have successfully used *LOCM* to create state machines, object associations and action schema for 3 domains. Evaluation of these results is ongoing, but initial results show that state machines can be deduced from a reasonably small number of plan examples, whereas inducing the state parameters requires much larger training sets. The total number of steps in sets of training sequences required for deriving state machines and parameters is summarised below for three example domains.

Domain	Examples	# steps (state)	# steps (params)
Tyre-world	Random	125	2327
Blocks	Random	34	250
Driverlog	Competition	205	3046

Tyre-world (GIPO version). A correct state machine is derived, corresponding closely to the domain as constructed in GIPO. The induced domain contains extra states for the *jack* sort, but this model is valid (see fig. 3). After training to convergence there are 3 parameter flaws, leading to some faults in action schema (see the end of this section for a discussion of flaws).

Blocks (GIPO version). A correct state machine is derived. After training to convergence there are 3 parameter flaws. The low number of steps needed to derive the state machine is due to there being only 2 sorts in the domain, both of which are involved in every action.

Driverlog (IPC strips version). State machines and parameters are correct for all sorts except trucks. For trucks, the distinction of states with/without driver is lost, and an extra state parameter (driver) is retained.

Randomly-generated example data can be different in character from purposeful, goal-directed plans. In a sense, random data is more informative, because the random plan is likely to visit more permutations of action sequences which a goal-directed sequence may not. However, if the useful, goal-directed sequences lead to induction of a state machine with more states, this could be seen as useful heuristic information.

Where there is only one object of a particular sort (e.g. gripper, wrench, container) all hypotheses about matching

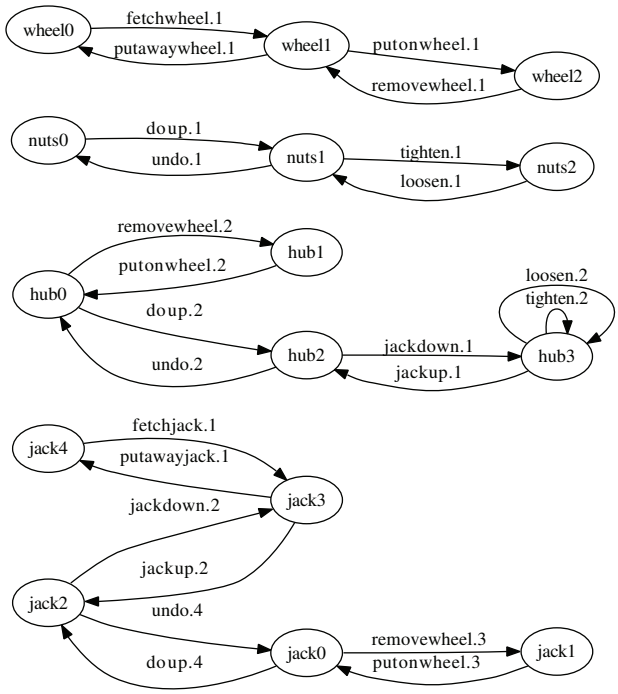

Figure 3: Other state machines induced from the tyre-world.

that sort always hold, and the sort tends to become an internal state parameter of everything. For this reason, it is important to use training data in which more than one object of each sort is used.

The induced models may contain detectable flaws: the existence of a state parameter has been induced, but there are one or more transitions into the state which do not set the state parameter. The flaws usually arise because state parameters are induced only by considering pairs of consecutive transitions, not longer paths. The inconsistency may indicate that an object reference is carried in from another state without being mentioned in an action's argument. In this case a repair to the model can be proposed, which involves adding the "hidden" parameter to some states, but a further cycle of testing against the example data would be required to check that the repair is consistent. This will be further developed in future work.

The most fundamental limitation is whether it is possible to correctly represent the domain within the limitations of the representation that we use for action schema.

- We assume that an action moves the objects in its arguments between clearly-defined substates. Objects which are passively involved in an action may make a transition to the same state, but cannot be in a *don't care* state.
- Static background information, such as the specific fixed relationships between objects (e.g. which places are connected), is not analysed by the system. In general, this can lead to missing preconditions. The *LOCM* algorithm assumes that all information about an object is represented in its state and state parameters. In general, this form of information may vary anyway between training examples.

Related Work

LOCM is distinct from other systems that learn action schema from examples in that it requires **only** the action sequences as input; its success is based on the assumption that the output domain model can be represented in an object-centred representation. Other systems require richer input: ARMS (Wu, Yang, and Jiang 2005) makes use of background knowledge as input, comprising types, relations and initial and goal states, while SLAF (Shahaf and Amir 2006) appears to efficiently build expressive actions schema, but requires as input specifications of fluents, as well as partial observations of intermediate states between action executions. The *Opmaker*2 algorithm detailed in (McCluskey et al. 2009) relies on an object-centred approach similar to *LOCM* but it too requires a partial domain model as input as well as a training instance. The TIM domain analysis tool (Fox and Long 1998) uses a similar intermediate representation to *LOCM* (i.e. state space for each sort), but in TIM, the object state machines are extracted from a complete domain definition and problem definition, and then used to derive hierarchical sorts and state invariants.

Conclusion

In this paper, we have described the *LOCM* system and its use in learning domain models (comprising object sorts, state descriptions, and action schema), from example action sequences containing no state information. Although it is unrealistic to expect example sets of plans to be available for all new domains, we expect the technique to be beneficial in domains where automatic logging of some existing process yields plentiful training data, e.g. games, online transactions. The work is at an early stage, but we have already obtained promising results on benchmark domains, and we see many possibilities for further developing the technique.

References

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *J. Artif. Intell. Res. (JAIR)* 9:367–421.

McCluskey, T.; Cresswell, S.; Richardson, N.; and West, M. M. 2009. Automated acquisition of action knowledge. In *International Conference on Agents and Artificial Intelligence (ICAART)*, 93–100.

Shahaf, D., and Amir, E. 2006. Learning partially observable action schemas. In *AAAI*. AAAI Press.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning Domain Definition Using GIPO. *Journal of Knowledge Engineering* 1.

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*.