Generalised Domain Model Acquisition from Action Traces

Stephen Cresswell
The Stationery Office
St. Crispins, Duke Street,
Norwich, NR3 1PD, UK
stephen.cresswell@tso.co.uk

Peter Gregory
Computer and Information Sciences
University of Strathclyde
Glasgow, UK
pg@cis.strath.ac.uk

Abstract

One approach to the problem of formulating domain models for planning is to learn the models from example action sequences. The *LOCM* system demonstrated the feasibility of learning domain models from example action sequences only, with no observation of states before, during or after the plans. *LOCM* uses an object-centred representation, in which each object is represented by a single parameterised state machine. This makes it powerful for learning domains which fit within that representation, but there are some well-known domains which do not.

This paper introduces *LOCM2*, a novel algorithm in which the domain representation of *LOCM* is generalised to allow *multiple* parameterised state machines to represent a single object. This extends the coverage of domains for which an adequate domain model can be learned. The *LOCM2* algorithm is described and evaluated by testing domain learning from example plans from published results of past International Planning Competitions.

Introduction

Many activities can be represented as deterministic state-transition systems. For example, games, business processes and other human activities. Once modelled in an appropriate representation language, it is possible to reason about these systems, for example in plan generation. A formal model of an activity can be created by hand by an expert in both modelling state-transition systems and the target domain. However, it is possible to automatically infer the underlying transition system from sample action sequences of the domain. Using such an approach removes the necessity for the domain expert to also be an expert at modelling transition systems.

The *LOCM* system (Cresswell, McCluskey, and West 2009) learns planning domain models from sets of example plans. Its distinguishing feature is that the domain models are learned without any observation of the states in the plan or of predicates used to describe them. This works because of some restrictive assumptions about the form of the model describing the domain. In particular, the objects are grouped into sorts, and the behaviour available to objects of any given sort is described by a single parameterised state machine. *LOCM* is very powerful at finding a model that fits within this representation. However, in many examples, there is no model within this representation which sufficiently captures the semantics of the domain. The expressiveness of the representation used by *LOCM* is more restrictive than STRIPS. The consequence being that there are domains which can be modelled in STRIPS which cannot be represented by the formalism of *LOCM*. If the domain cannot be expressed within the representation used by *LOCM*, this will result in an over-simplified model which is too permissive. A common feature of planning domains which cause this problem is where objects have separate aspects of their behaviour.

In the present paper, we weaken the assumptions of *LOCM* to allow a more expressive representation, thereby enabling a wider range of domain models to be satisfactorily learned. Specifically, we allow the separate aspects of an object's behaviour to be represented by separate state machines. This necessitates that we also develop new techniques for analysis of example plans and synthesis of state machines. In the rest of the paper, we introduce some different visualisations of training data, we study examples where *LOCM* cannot represent the underlying domain model. We then discuss the limitations of *LOCM*, propose a generalised model to overcome these limitations, propose an algorithm (that we call *LOCM2*) capable of deriving instances of the generalised model, and evaluate the new algorithm on a collection of benchmark problems.

Background

The *LOCM* system of (Cresswell, McCluskey, and West 2009) forms a specification of a planning domain in the form of planning operators in the STRIPS sublanguage of PDDL. The planning domain is formed by analysis of sequences of example plans, where each action appears as an action name and arguments in the form of a vector of object names. From this, the system synthesises a representation of state by grouping objects into sorts, forming a state machine for each object sort, and then enhancing states with parameters which record associations between objects.

In the following, we use the term *transition* to characterise the state change made by an object when it appears at a particular argument position of a particular action - e.g. *board_truck.1* labels the state transition made by an object appearing in the first parameter of the *board_truck* action.

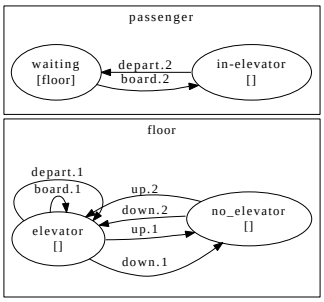


Figure 1: The FSMs produced by *LOCM* that describe the Miconic domain.

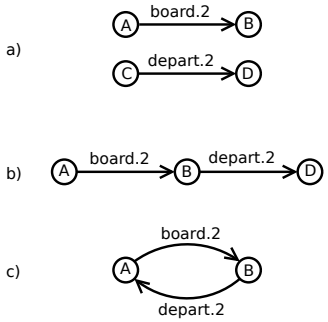


Figure 2: State unification in *LOCM*

Throughout the paper, we consider states of single objects. Objects states are atomic in our representation. The object state is not a collection of statements about the object, but is instead used to form a single statement about the object.
The assumptions of *LOCM* are:

1. We consider that each invocation of a planning operator causes a specific state change in each of the objects given as arguments.
2. The behaviour of each object is described by a single FSM.
3. Objects are grouped into sorts, and objects of the same sort are described by identical state machines.
4. Each argument position of each action always takes objects of the same sort.
5. Each transition appears only once in the FSM.

As we shall see, these assumptions can result in the derived models being oversimplified.

Example: Miconic Domain The Miconic domain models the use a single elevator in a building. There are four actions available: board, depart, up and down. Figure 1 shows the output that *LOCM* produces in this domain. Two state machines are sufficient to represent this domain. The first represents the behaviour of passengers, who can transition between being in and out of the elevator. The other represents the state of individual floors in the building: if the elevator is at the current floor, then passengers can board and depart.

One aspect of the *LOCM* system critical to this work is the unification of candidate states due to the example plans. We will now demonstrate how state unification produces the 'passenger' FSM from Figure 1. Suppose the only example plan available to *LOCM* is the following:

```
1:  (board  floor1 passenger1)
2:  (up     floor1 floor2)
3:  (depart floor2 passenger1)
4:  (board  floor2 passenger1)
```

Consider the object passenger1: passenger1 is a parameter of actions 1, 3 and 4 in positions board.2, depart.2 and board.2. Figure 2a) shows our initial knowledge of the behaviour of passenger1: board and depart both change the state of passenger1 in some way. Now, considering actions 1 and 3, we have evidence that the the end state of board.2 is the start state of depart.2. This means we can *unify* state B and C, this produces the FSM in Figure 2b) where B and C are collapsed into one state. Finally, when we consider actions 3 and 4, we discover that the end state of depart.2 is the start state of board.2, and states A and D are unified. Thus, *LOCM* discovers the passenger FSM in Figure 1.

A later stage of *LOCM* analyses the associations between objects, and this results in state machines which are parameterised - i.e. a state may contain a variable which records a dynamic association to another object. An example can be seen in Figure 1: the passenger sort is related to a floor when in the waiting state. This part of the analysis in *LOCM2* is equivalent to that in *LOCM*, and is not described in this paper. For details, see (Cresswell, McCluskey, and West 2010).

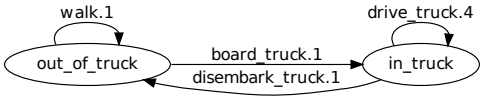
Similarly, *LOCM2* performs the same zero-analysis as *LOCM*. Zero-analysis deals with "background" state which is not associated with a named object (e.g. an implicit "hand" object that picks and places cards in freecell). If each action is treated as though it has a single "background" object as its 0^{th} argument, such background state can be revealed.

Representing Sorts

We now provide a short discussion of two different ways of representing the dynamic behaviour of a sort using state-machines. Throughout the paper we will use the driverlog domain as an example domain. The driverlog domain is a transportation domain with sorts driver, truck, package and location. Packages need to be transported in trucks, which require drivers in order to drive. Drivers can walk between locations, while trucks can drive between them.

State-Centred Representation

In the first representation, used by *LOCM*, the states in the state-machine represent the individual states of the sort, connected by the actions that enable transitions between two states of the underlying sort. An example is given below, for the sort *driver* from the driverlog domain. The sort has two states: depending on whether or not the driver is in a truck.

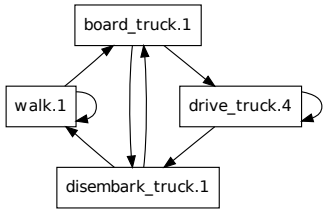


LOCM forms a state machine for each sort by analysing the sequence of transitions made by each individual object. The state machines are constrained to have the property that a transition may not label more than one edge. Under this restriction, the STRIPS representation of an operator can be composed from state transitions brought about on each of its arguments. If any transition labels were allowed to occur on more than one edge, then a direct construction of the operator would require the use of conditional effects.

Transition-Centred Representation

The *transition-centred representation* is, again, a state machine representation. However, here the states in the FSM represent the transitions that can affect the state of objects of a certain sort. An edge exists between two transitions, a and b, if the transitions represented by a and b can be consecutive for some object in an action sequence. This is a more expressive representation than the state-centred representation, in that not every transition-centred graph has a directly equivalent state-centred graph.

The state machine shown below represents the *driver* sort discussed previously.



It is convenient for part of our analysis to also consider the matrix representation of the transition graph. The matrix for our example is given below.

	1.	2.	3.	4.
1. board_truck.1		x		x
2. disembark_truck.1	x		x	
3. walk.1	x		x	
4. drive_truck.4		x		x

We can extract the transition-centred representation directly from the observed action sequences. This is done by considering all transitions which affect a given object, and then for each pair of transitions, $\langle a, a' \rangle$, consecutive for the object, mark the cell a, a' in the matrix. However, unlike the state-centred representation, the transition-centred representation cannot be used directly for building STRIPS-like operators.

Equivalence of Representations

Note that the transition matrix representation (and equivalent transition-centred graph representation) are more expressive than the simple state machine representation. Hence, in order to represent transition-centred model in STRIPS, we need to consider the circumstances under which a conversion from the transition matrix to an exactly equivalent state machine can be made.

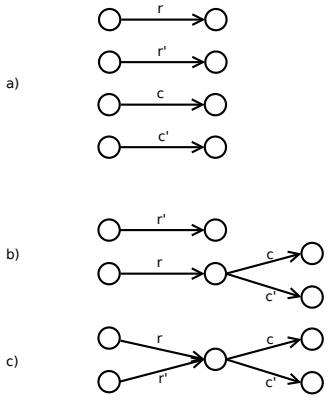


Figure 3: Over-generalising behaviour of *LOCM*

Equivalence Conditions A row in the transition connection matrix describes, for a given transition, which transitions may follow. Hence any two transitions which end in the same state will have the same pattern in the row. Under the assumption that each transition may only appear once in the state-based representation, we can say that any two row patterns must either be identical or non-overlapping.

Overlapping but non-identical patterns can be detected by looking for the following fingerprint (for any r, r', c, c'):

	$\ldots$	c	$\ldots$	c'	$\ldots$
$\vdots$		$\vdots$		$\vdots$	
r	$\ldots$	x	$\ldots$	x	
$\vdots$		$\vdots$		$\vdots$	
r'	$\ldots$	x	$\ldots$	o	
$\vdots$					

If this fingerprint is not present, we say that the matrix is *well-formed*.

Definition 1 (Well-formed transition connection matrix) *A matrix is well-formed if, for any two rows, r and r', and any two columns c and c', such that the transition pairs $\langle r, c \rangle$, $\langle r, c' \rangle$ and $\langle r', c \rangle$ are included, the transition pair $\langle r', c' \rangle$ is also included.*

The well-formedness property is effectively enforced by *LOCM*, even if this results in a model permitting transition pairs which are not observed in the example sequences. Figure 3 shows how *LOCM* would unify the states observed in the badly-formed matrix in Definition 1, and why this leads to over-generalisation. In Figure 3a), the four actions r, r', c and c' are listed. In Figure 3b) we see the state after we have considered row r: c and c' have both been observed following r, therefore the start states of c and c' must be the end state of r. Now, consider adding r' to the FSM: since c has been observed following r' the end state of r' is unified with the start state of c. This is shown in Figure 3c) and demonstrates the unfortunate result of allowing the unobserved sequence r', c'. The unification of states used in the *LOCM* algorithm is equivalent to making any two overlapping rows (or columns) into the union of both. In the rest

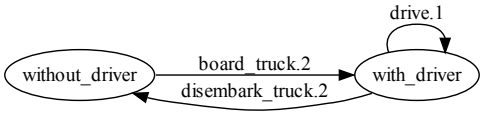
of the paper, we refer to a transition pair not observed in the example sequence, but filled in by generalisation, as a *hole*.

Example: Driverlog

For the sort *truck* in driverlog, the *LOCM* analysis produces only a single state. This is an incorrect interpretation of the domain, as it is not possible to perform two consecutive *board_truck* actions on the same truck, for example. An adequate model should at least have states with/without driver – the truck *should* only drive when it has a driver. However the pairs of consecutive transitions possible are defined by the following transition matrix, which is not well-formed:

	1.	2.	3.	4.	5.
1. drive_truck.1	x		x	x	x
2. board_truck.2	x		x	x	x
3. disembark_truck.2		x		x	x
4. load_truck.2	x	x	x	x	x
5. unload_truck.2	x	x	x	x	x

The hole at transition pair $\langle 3, 3 \rangle$ is correct because it is not possible for a driver to disembark from a truck twice consecutively. However, *LOCM* over-generalises and fills the hole. If the *load_truck* and *unload_truck* actions were not included, and only the *drive_truck.1, board_truck.2* and *disembark_truck.2* were considered, we would obtain a well-formed matrix, which can be converted to the following state machine. This correctly models the behaviour of the truck sort:



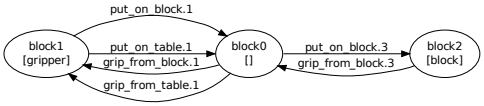
Suppose we modify the driverlog domain so that the capacity of the truck allows only a single package to be loaded (e.g. a container truck). In this case the matrix is different because it is not possible to have two consecutive *load_truck.2* or two consecutive *unload_truck.2* transitions.

	1.	2.	3.	4.	5.
1. drive_truck.1	x		x	x	x
2. board_truck.2	x		x	x	x
3. disembark_truck.2		x		x	x
4. load_truck.2	x	x	x		x
5. unload_truck.2	x	x	x	x	

In this case, it can be seen that the *load_truck.2* and *unload_truck2* transitions form a separate machine with two states.

An Example: Blocksworld

blocksworld (with gripper) has four operators. Despite being able to correctly infer some blocksworld encodings (McCluskey et al. 2009), *LOCM* cannot correctly infer the domain in the case of the four operator domain. The output from training *LOCM* on a four operator blocksworld action trace is as follows:



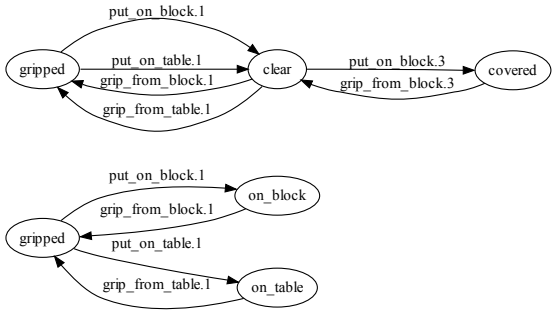
We can recognise the states as: (block1=gripped, block0=clear, block2=covered). Note that the resulting state machine essentially describes only the *top* of the block. Any information about whether the block is standing on another block or on the table has been obscured. To understand how this occurs, it is helpful to look at the transition connection matrix:

	1.	2.	3.	4.	5.	6.	
1. grip_from_block.1			x	x			} gripped
2. grip_from_table.1			x	x			
3. put_on_block.1	x	o				x	} clear
4. put_on_table.1	o	x				x	
5. grip_from_block.3	x	x				x	
6. put_on_block.3					x		} covered

In the matrix above, holes are shown with the symbol o. The *LOCM* analysis fills the holes, and the consequence is that it permits incorrect transition pairs (such as $\langle 3, 2 \rangle$, meaning that a block may be put down on a block, and then picked from the table). It is in this way that we mean that the state machine only captures the behaviour of the *top* of a block. However, if we consider the matrix formed by the subset of transitions numbered $\{1, 2, 3, 4\}$, this matrix is well-formed and can be expressed exactly as a state machine.

	1.	2.	3.	4.	
1. grip_from_block.1			x	x	} gripped
2. grip_from_table.1			x	x	
3. put_on_block.1	x				} on_block
4. put_on_table.1		x			} on_table

The state machine formed from this matrix captures the states of the *bottom* of a block. Hence we can represent the behaviour of the block with two state machines, representing the top and the bottom of the block:



Any given block occupies one state in each state machine.

In summary, the above examples illustrate situations where the assumption of *LOCM* (that each sort can be represented by a single state machine) are inadequate. This arises when the behaviour of an object has separate aspects. We now show how we can overcome this problem by using multiple state machines per sort.

The *LOCM2* Algorithm

The key idea of *LOCM2* is that the behaviour of a sort is represented by *multiple* finite state machines, with each machine characterised by a set of transitions. The set of state

machines is formulated to capture the set of observed transition pairs.

Firstly, the criterion introduced in Definition 1 allows us to check the state machine formed from all transitions for the sort, and determine whether it is well-formed. If it is well-formed, then a single state-machine suffices, and the *LOCM* analysis is adequate.

If the state machine is not well-formed, then it contains one or more *holes*. The reader will recall that these are transition pairs not observed in the data, that would be permitted if approximated by a single state machine. In this situation, the *LOCM2* algorithm searches to discover a set of valid state machines that correctly model the holes, such as the machines shown for the driverlog and blocksworld domains in the previous section.

Definition 2 (Valid transition subset) *If E is a set of example sequences, Srt is a sort, T_{all} is the set of all transitions observed in E for Srt, P is the set of transition pairs observed in E for Srt, and s is a subset of T_{all}. A transition connection matrix M is formed from s using all pairs $\langle t_1, t_2\rangle \in P$ such that $\{t_1, t_2\} \subseteq s$. s is deemed to be valid iff:*

- *M is well-formed by Definition 1, and*
- *M can be validated against the example sequences E.*

Even if the matrix of a transition set is well-formed, it may still be inconsistent with the example sequences, e.g. it may contain inappropriate dead-ends.

Unlike *LOCM*, the *LOCM2* algorithm is heuristic in nature. This is because we are faced with a combinatorial problem which contains many possible solutions. In general, we can consider that the set of valid transition subsets for a given sort form a lattice. Figure 4 shows a lattice for the *block* sort from the blocksworld domain, in which there 6 transitions. Each node in this lattice represents a set of transitions which defines a state machine. The task is to select a set of state machines such that the behaviour of the sort is adequately described.

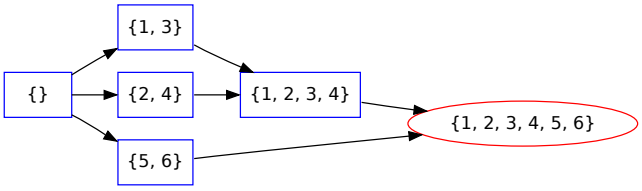


Figure 4: A lattice generated from the blocksworld domain. The rectangular nodes represent transition sets which are both well-formed and tested against example sequences. The elliptical node represents the set of all transitions, which in this case is not well-formed.

The strategy is to include a state machine with all of the transitions for the sort (as in *LOCM*), but to additionally check for holes. Given that the holes indicate behaviour missed in the *LOCM* analysis, further analysis is performed with the aim of selecting valid subsets, such that the each transition pair forming a hole[1] occur together in some subset.

It is possible for a matrix to contain holes which cannot be described by any FSM formed from a subset of its transitions. In this case, search fails and the single FSM is used (as in *LOCM*).

Procedure *select_transition_sets*
Input:
T_{all} – set of observed transitions for sort
H – set of holes - each hole is a set of one or two transitions.
P – set of pairs $\langle t_1, t_2\rangle$, meaning t_1 and t_2 occur as consecutive transitions.
E – set of example sequences of actions
Output:
S – set of transition sets.

begin
1. $S \leftarrow \emptyset$

/* Ensure each hole is included */
2. **for each** $h \in H$
3. **If** there is no set $s' \in S$ such that $h \subseteq s'$ **then**
4. By breadth-first search, form s, the smallest set such that $h \subseteq s \subset T_{all}$ and
5. s is valid with respect to P, E by Definition 2
6. $S \leftarrow S \cup \{s\}$
end If
end for

/* Remove redundant sets */
7. **If**, for any two sets s_1 and s_2 in S $s_1 \subset s_2$ **then** $S \leftarrow S \setminus \{s_1\}$
end If

/* Include all-transitions machine, even though it might not be well-formed */
8. $S \leftarrow S \cup \{T_{all}\}$
return S
end

To illustrate the algorithm, we will now step through it with the example of the blocks sort from the blocksworld domain. The arguments are:
$T_{all} = \{1, 2, 3, 4, 5, 6\}$
$H = \{\{1, 4\}, \{2, 3\}\}$
$P = \{\langle 1, 3\rangle, \langle 1, 4\rangle, \ldots\}$

At step 1, we initialise the solution set S to an empty set. At step 2, we begin the loop through the set of holes H, selecting the first hole $h = \{1, 4\}$. Step 3 checks that h is not already solved by existing transition set in S, but as S is empty at this point, we do need to search for a solution.

In the search (step 4), we attempt to discover a set which includes h, and is well-formed and valid against the test data. This is done by performing breadth first search through

[1] A hole is detected as an ordered pair $\langle t_1, t_2\rangle$, but the ordering is not relevant in the body of the algorithm, so we treat each hole as set $\{t_1, t_2\}$.

sets of increasing size, starting with $s = \{1, 4\}$, then forming and testing candidates sets in the sequence $\{1, 2, 4\}$, $\{1, 3, 4\}$, $\{1, 4, 5\}$, $\{1, 4, 6\}$, $\{1, 2, 3, 4\}$, etc. Each candidate is tested first for well-formedness, which is a simple test on the matrix formed from the transition set. If this test succeeds, an additional test is performed against the example data. For this example, all transition sets which pass both tests are shown in Figure 4. The solution found by the search procedure is $s = \{1, 2, 3, 4\}$. We can now add this to the solution set S, so $S = \{\{1, 2, 3, 4\}\}$.

Returning to step 2 for the the next iteration, we consider the next hole from H, and so we have $h = \{2, 3\}$. At step 3, we find that there is already a set in S which includes h, so there is no need for further search.

Step 7, is to tidy up redundancy in the solution set, but this not needed in this case. Step 8, extends S to include the all-transitions set T_{all}. So the solution set returned by the algorithm is $S = \{\{1, 2, 3, 4\}, \{1, 2, 3, 4, 5, 6\}\}$. These two transition sets are the transition sets that formed the two state machines in the blockworld example in the previous section.

Algorithm Properties

Algorithmic Complexity

LOCM2 is a practical algorithm with typical runtimes of less than five seconds. The sensitivity of the runtime is to the number of transitions in a sort. A search for a subset from N transitions could explore all 2^N candidate transition subsets. However, N is small in all domains we have encountered. The highest ($N = 14$) was in the rovers domain, and this completed in less than one minute.

Heuristic Learning

Recall the Miconic elevator example from Figure 1. The FSMs resulting from executing *LOCM2* on the example plans from the IPC2 competition are show in Figure 5. The FSM learned for the passenger sort differs from the one shown previously. This is due to the fact that no sequences were observed in which passengers depart the elevator and then board it again. Doing so would always lead to sub-optimal plans in the domain, as the elevator is of fixed capacity. This is a form of heuristic learning: the model does not learn valid transitions that the planners do not use in the learning data.

Static Information

The current *LOCM2* system infers the dynamic properties of a planning domain. In reality, planning domains also rely on static information, such as connectedness of a road network. In order to illustrate this, we present the results from the Peg Solitaire domain. In this domain, there are three actions: new-move, continue-move and end-move. Both the new-move and continue-move actions perform a jump of a peg. The dynamics of the domain are captured in Figure 7, where it can also be seen that zero-analysis discovers some of the necessary dynamics of the domain.

The state machine at the bottom of Figure 7 shows the dynamics of the position sort. The state occupied1 is the state

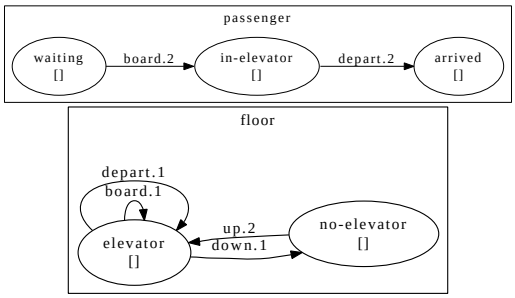


Figure 5: Heuristic learning in the Miconic domain.

in which a position is occupied by a peg, and there is no current move to continue. From this state, either the peg at the position is removed by starting a new move, or by another peg jumping over it. Both of these cases leave the position in the state empty. To become occupied again, a peg can be moved from elsewhere into the position. The PDDL generated by *LOCM2* is equivalent to the original PDDL, except that the static information is not present (the important static information is that the three locations in the parameters of the jump actions are in a line). Cresswell, McCluskey, and West discuss an approach to extracting static information in *LOCM*. but this relies on manually providing a hint.

Overfitting

Overfitting occurs in *LOCM* and *LOCM2* when the training data is not extensive enough. Heuristic learning may be a desirable form of overfitting. Overfitting will typically lead to overly-specific state machines, in which the correct state unifications have not been performed. In general it is an undesirable property.

Evaluation of *LOCM2*

In order to provide an empirical evaluation of *LOCM2* we have used several standard planning benchmarks as test-cases. This is useful for several reasons: they are already familiar to the planning community, the fact that the domains are already defined makes validation of *LOCM2* easier and there are already relatively large collections of plans available. These plan libraries are the collections of plans that were produced in the various planning competitions.

Domains

We now present the outcome of running the *LOCM2* algorithm on several benchmark domains. We first present domains which required *LOCM2* analysis, and could not be represented by the *LOCM* representation. We then discuss domains which can be represented by both representations.

LOCM2 Required

Driverlog As previously described, the Driverlog domain requires the new features in *LOCM2* as the truck sort has independent behaviours (to transport both drivers and packages).

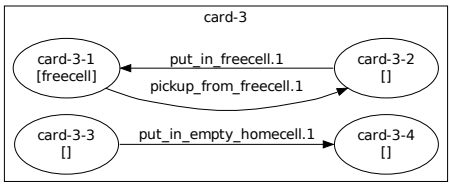


Figure 6: One of the state machines for sort *card* in the AoP-freecell domain, demonstrating a learned refinement of the sort.

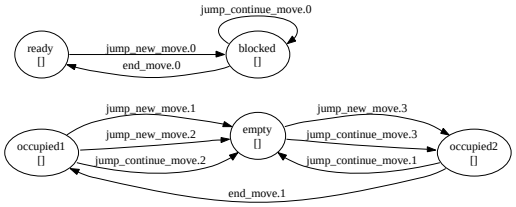


Figure 7: The derived state machines for the Peg Solitaire domain. This is an example where zero-analysis is required in order to correctly model the domain.

Four Operator Blocksworld Again, as previously discussed in detail, in order to correctly model each block, separate state machines are required.

AoP Freecell There are some interesting aspects to the AoP-freecell example. This is the version where test data was generated from humans playing the game.

Whereas the output from *LOCM* is strong enough to model the domain (Cresswell, McCluskey, and West 2010), the *LOCM2* analysis refines this further. As there are stacks of cards in the domain, there is effectively an embedded instance of the blocks world, albeit one with named table positions, for which the previous *LOCM* analysis is adequate. The *LOCM2* model contains two additional state machines for *card*. The first additional machine introduces distinct states for the cards according to what they are placed on - this is analogous to the state machine for the bottom of a block in blocks world, which is not strictly required in this case.

The final state machine for *card* has the interesting feature that it contains two subgraphs which are not connected (Figure 6). This distinguishes cards which can be placed in an empty homecell (i.e. aces) from cards which can be moved to and from freecells. Whereas it is technically possible for aces to be moved to and from freecells, this is never useful in the game, and consequently does not occur anywhere in the training data. Hence, the analysis has effectively revealed a hierarchical refinement of the sort *card*.

Storage The Storage domain models a warehouse in which crates must be transported by a hoist. Storage has a complex type-hierarchy for its location types. There are three important location types: store-area, transit-area and place. *LOCM2* correctly identifies this subtyping relationship.

LOCM **Required** There are many domains which only contain sorts with individual behaviours, in which the original state machines which would have been produced by the previous approach are still produced using *LOCM2* analysis.

Logistics This domain is correctly learned and *LOCM2* learns heuristic information. This information regards how the packages are transported. After a package is unloaded from a truck, the only action available to it is being loaded into an aircraft (also vice versa). This is because there are no situations in the example plans in which packages are transferred from truck-to-truck. This type of heuristic learning could potentially be useful. If not, then models could be learned from random-walks instead of goal-directed plans.

Grid The model learned for the Grid domain is accurate. The location sort has three states: locked, unlocked-occupied and unlocked-unoccupied. Overfitting to the data again provides heuristic learning in the following way. In Grid, there are two ways to put a key down, put-down and pickup-and-lose (the first drops the current key, the second exchanges it with another key). If the path to the goal location of the key currently held is blocked, then it is always better (in terms of minimising plan steps) to exchange the currently held key for the required key than to perform the atomic actions. As all of the example plans exploited this rule, the learned model does not allow keys to be picked up once they are dropped with no exchange.

Miconic The Miconic example is shown in Figure 5, and was previously discussed in length.

Peg Solitaire The dynamic elements of the Peg Solitaire domain are learned correctly, as previously discussed.

Sokoban The dynamic aspects are modelled well, however, much of the interesting structure is static. For instance, the connections in the grid define where the crates can be pushed to.

Depots Depots has an embedded blocksworld-like aspect to it, where crates can be stacked on top of each other. As such, it seems that it should require multiple state machines to represent the towers, in the same way as blocksworld. The crucial aspect of depots that means that it can be represented using only a single state machine per sort is that there are explicitly named bottoms to each stack (named pallets in the domain) which cannot be lifted. As such, a crate will never be the bottom item in a stack, and the second state machine that was required in blocksworld is not required in depots.

Discussion

The *LOCM2* approach is different in character from *LOCM*, in that domain models constructed by *LOCM* are tightly constrained by the choice a simple representation for the domain which leads to single solution, whereas the more general domain representation of *LOCM2* allows many possible solutions to the same problem, and it is necessary to apply some heuristic criteria to select a single solution.

The *LOCM2* algorithm provides good coverage of dynamic aspects of benchmark problems. There remain a number of situations which cause problems, preventing full coverage of STRIPS domains:

- Domains which include dynamic many-many relationships. In *LOCM2* (as in *LOCM*), dynamic associations between objects are represented by parameters attached to FSM states. In the case of a dynamic many-many relationship (e.g., adding or removing roads between pairs of locations), this would require a variable number of parameters, and the systems cannot learn or represent this case. The case is rare in benchmark domains.
- Domains in which the same object may appear more than once in the arguments of actions (e.g. Rovers, in which the waypoints often occur more than once in the parameters of an action). This provides a difficulty for our analysis. However, we do not believe that this is a theoretical limitation, but only a limitation in our implementation.
- Domains in which an operator deletes a condition which is not a precondition (e.g. satellite). Effectively, this is a conditional effect, and is not truly in the STRIPS fragment of PDDL.
- Domains in which an operator includes the same condition in both the add and the delete effects (e.g. rovers). This obscure PDDL encoding can be used to suppress concurrent usage of some resource. *LOCM* models this as a precondition only.

Related Work

Systems which learn models from action traces vary in the amount of the additional information required, the observability of the intermediate states, and in the expressiveness of the formalism in which plan operators are formed.

In all cases, the building blocks of the state representation are given as input - e.g. a set of predicates. *LOCM* is unique in not needing to observe any states, and synthesising the state representation. This is only possible by assuming that plan operators are formed in a restricted language. *LOCM2* improves on this.

ARMS (Wu, Yang, and Jiang 2007), can learn STRIPS domain models with partial or no observation of intermediate states in the plans, but does at least require predicates to be declared. The LAMP system (Zhuo et al. 2010) extends the expressiveness of the action representations to include PDDL representations with quantifiers and logical implications.

The *Opmaker2* system (McCluskey et al. 2009; Richardson 2008) is based around the object-centred representation of OCL. This assumes that a small number of plans are handcrafted as part of the the process of defining a planning domain, typically including initial and goal states, but not intermediate states. A partial domain model given as input includes a specification of objects in each sort, states available in each sort and state invariants.

The system described in (Shahaf and Amir 2006) learns plan operators using a richer representation of action. However, it does require partial observations of state.

More generally, the TIM system (Fox and Long 1998) constructs state-machines similar to the ones contained in this work, except with the full knowledge of the domain and problem. This analysis is designed to reveal implicit type information and state invariants.

Conclusion

We have presented a new technique for automatically acquiring domain models from example action sequences with no additional information. We have identified that the representation of the previous state-of-the-art algorithm limits the domains it can learn to those where no sort has independent behaviours. There are several standard benchmark domains which, therefore, are not correctly identified by this algorithm.

We have shown that, by using a new representation where each object is represented by multiple state-machines, these benchmark domains can be represented. The main contribution of this work is to show how these multiple state machines can be generated using only example action sequences.

References

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of object-centred domain models from planning examples. In Gerevini, A.; Howe, A. E.; Cesta, A.; and Refanidis, I., eds., *ICAPS*. AAAI.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2010. Acquiring planning domain models using LOCM. Accepted for publication in Knowledge Engineering Review. Available from http://eprints.hud.ac.uk/9052.

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

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

Richardson, N. E. 2008. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. Ph.D. Dissertation, School of Computing and Engineering, University of Huddersfield, UK.

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

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *Knowl. Eng. Rev.* 22(2):135–152.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artif. Intell.* 174:1540–1569.