Domain Model Acquisition with Missing Information and Noisy Data

Peter Gregory
Schlumberger Gould Research,
Madingley Road,
Cambridge, UK
pgregory@slb.com

Alan Lindsay and **Julie Porteous**
Digital Futures Institute,
School of Computing,
Teesside University, UK
firstinitial.lastname@tees.ac.uk

Abstract

In this work, we address the problem of learning planning domain models from example action traces that contain missing and noisy data. In many situations, the action traces that form the input to domain model acquisition systems are sourced from observations or even natural language descriptions of plans. It is often the case that these observations are noisy and incomplete. Therefore, making domain model acquisition systems that are robust to such data is crucial. Previous approaches to this problem have relied upon having access to the underlying state in the input plans. We lift this assumption and provide a system that does not require any state information. We build upon the *LOCM* family of algorithms, which also lift this assumption in the deterministic version of the domain model acquisition problem, to provide a domain model acquisition system that learns domain models from noisy plans with missing information.

Introduction

When faced with the task of creating a planning domain model that accurately models a real-world problem that needs to be solved, in most current situations an AI Planning expert must also learn to become a domain expert in the problem area to be modelled. Domain model acquisition is an area of research trying to reduce the gap between domain expert and modelling expert. Domain model acquisition is the problem of automatically generating planning models from input data of some form. This input data can vary in many ways, but typically at least contain collections of plan traces in some form. Other information that may be available are intermediate states, solution metadata (such as plan costs, or whether plans are goal-directed or optimal), etc.

In this work, we study the problem of domain model acquisition when the input plans have noisy data and missing information. This problem has previously been studied (Mourao et al. 2012) with the assumption that intermediate state information is present. We relax this assumption, and provide an algorithm that does not rely on intermediate state information being present. There are important situations in which intermediate state information cannot be accessed. For example, when translating plans created for people to follow (e.g. the machine tool calibration plans in (Parkinson et al. 2012)), the plans only mention the actions,

```
a) (new-move p1-0 p1-1 p1-2) b) (new-move p0-0 p1-1 p1-2)
   (continue p1-2 p2-2 p3-2)    (continue p1-0 p2-2 p3-2) *
   (end-move p3-2)              (end-move p3-2)
   (new-move p3-1 p2-1 p1-1)    (new-move p3-1 p2-1 p1-1)
   (end-move p1-1)              (end-move ____) **
```

Figure 1: The first plan (a) shown is an example plan from the English Peg Solitaire domain. The second plan (b) shows the same plan with noise (*) and missing information (**).

and there is no state description. Plans created by people for other people can also be prone to mistakes and oversights, or, in the language of this paper, noise and missing information. Another place in which missing and noisy data is a problem, and in fact a motivating reason for developing a domain model acquisition system of this type, is the Framer system (Lindsay et al. 2017): a domain model acquisition system that has natural language descriptions of plans as its input. Although the exact details are not important here, it should be clear enough that natural language descriptions of plans are prone to noise and missing information, and do not provide information about intermediate states.

We call our system *LC_M*, as it is a version of the *LOCM* system with noisy and incomplete data (the *C* is misplaced and the underscore represents missing data). The *LOCM* family of algorithms (Cresswell, Mccluskey, and West 2009; Cresswell and Gregory 2011; Gregory and Cresswell 2015; Lindsay et al. 2017) are domain model acquisition systems, all sharing the assumption that plan traces with no intermediate state form the input to the system. It has proven possible to correctly learn domain models with rich structure, including the vast majority of the IPC domains, whilst still adhering to this very strong assumption about the input data. In this work, we assume that the input plans are generated through some process of observation, whether human or machine. We assume that each action is observed, but that the observer may either perceive the wrong action type (i.e. the action name), and / or the wrong action parameters. Missing data can be seen if the observer fails to perceive an action or action parameter with any degree of certainty. We also have the assumption that there is an underlying deterministic planning model that would be learnt by *LOCM* if no noise and missing information were present.

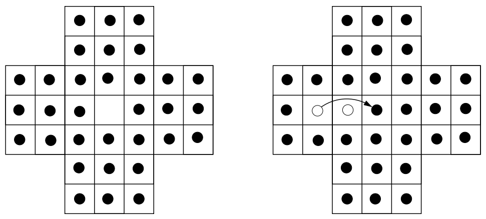
Figure 2: An English Peg Solitaire board, and an example of a move. Pegs must jump over other pegs, both removing the peg jumped over whilst leaving their current position clear.

Background

The *LOCM* system (Cresswell, Mccluskey, and West 2009; Cresswell, McCluskey, and West 2013) forms the basis for the work in this paper, therefore we provide an introduction to the most relevant parts of the *LOCM* system. To do this we use a running example of the one-player board game English Peg Solitaire. The goal is to clear the board of pegs, leaving a single peg in the middle. Pegs are cleared when an adjacent peg jumps over it, into another adjacent empty position, and this jump must be in a straight line. If the same peg performs more than one consecutive jump, then this counts as a single move in the optimisation criteria. In the planning domain, this is modelled as three different operators:

```
(new-move pos-from pos-over pos-end)
(continue pos-from pos-over pos-end)
(end-move pos)
```

The *LOCM* domain model acquisition system works by building finite state machines for each type of object in a planning domain, asserting that the behaviour of each object can therefore be defined as a finite state machine. It operates with the assumption that each action parameter asserts a transition in this state machine. Each object in a plan can be seen as going through a sequence of transitions, where a transition is defined by an action name and a parameter index. The transitions for the peg solitaire domain are shown in Table 1. For the plan in Figure 1 a) for example, the object `p1-1` has the transition sequence *new-move.2, new-move.3, end-move.1*. In Table 1, the imaginary zeroth parameters of the actions are also listed as transitions. This is important, as the structure of the plans can carry extra object-independent information about the domain structure. The zero transition sequence for the plan in Figure 1 a) then, is *new-move.0, continue.0, end-move.0, new-move.0, end-move.0*

Transition	Meaning
end-move.1	The position that a move ends on.
new-move.1	The position of the peg to move.
new-move.2	The position of the middle peg to be removed.
new-move.3	The empty position that the peg will land on.
continue.1	The position of the peg to move.
continue.2	The position of the middle peg to be removed.
continue.3	The empty position that the peg will land on.
end-move.0	The imaginary zeroth parameter of end-move.
new-move.0	The imaginary zeroth parameter of new-move.
continue.0	The imaginary zeroth parameter of continue.

Table 1: Table of transition meanings in peg solitaire domain.

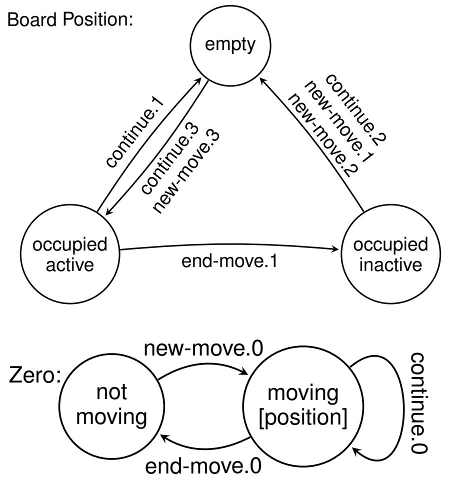

Figure 3: The finite state machines derived by *LOCM* for the Peg Solitaire domain. There are two state machines: one for the 'board position' type, and another for the zero machine, which models global dynamics of the system.

The domain model that *LOCM* learns for this domain can be described by the state machines in Figure 3, where there are two different state machine types: one for board positions and a zero state machine. The zero machine is the machine generated by assuming that each operator has a hidden zeroth parameter, and this represents zero place predicates in the domain. A crucial assumption in the *LOCM* system is that each transition appears at most once in each state machine. In order to construct the state machines for each type, *LOCM* performs an incremental unification of states, based on the transition sequences seen in the input. The consequence of the rule that each transition appears at most once, is that for a transition sequence pair A,B the end state of the A transition is the start state of the B transition. For the zero machine in Figure 3, the machine is constructed using the steps defined in Figure 4. Note, importantly, that a transition pair can change the structure of the generated state machine significantly. This is important in the context of this work, because even a small amount of noise can lead to incorrect state machines being learnt, and hence provides strong motivation to find ways of dealing with noise.

The final aspect of the *LOCM* system to discuss is the learning of state parameters. State parameters define temporary relationships that exist between different object state

Transition	In Parameter	Out Parameter
end-move.0		end-move.1
new-move.0	new-move.3	
continue.0	continue.3	continue.1

Table 2: Table of the position state parameter transitions for the moving state of the zero machine in Figure 3.

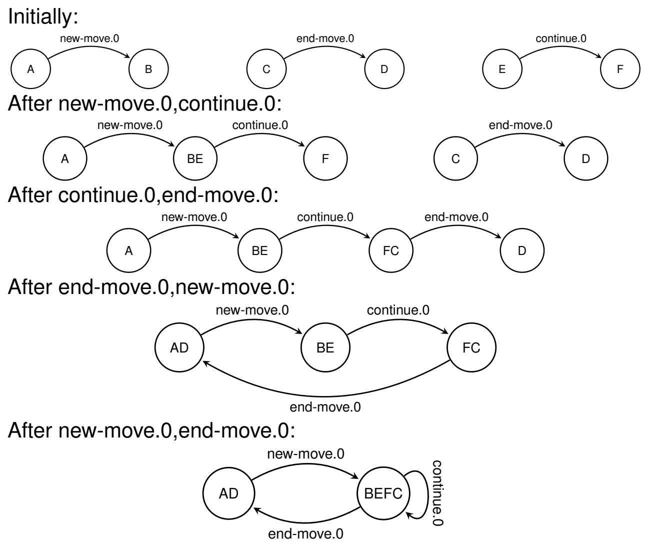

Figure 4: The progression of state unifications for the zero machine on the input plan in Figure 1 by *LOCM*. Initially, there is an assumption of independence between the different transitions. After considering the transitions in the example, the machine specified in Figure 3 is produced.

machines, typical examples include the location of a truck in a logistics domain. In the zero machine of Figure 3, for example, the moving state has the state parameter '[position]' which records the board position that is currently active during a move. In general, if a state in a *LOCM* state machine has a parameter, this means that for each pair of consecutive transitions in and out of the state there is a transition index in each that always has the same value. They effectively provide constraints between the parameters of the actions that affect the state machine with the parameter. The transition positions for the moving state in the zero machine are shown in Table 2.

Another convenient way of representing the input transitions is to look at the transition matrix. This idea was important in the *LOCM2* system (Cresswell and Gregory 2011) for deriving state machines for objects with multiple behaviours. We will, however, use the matrix for a different purpose here. The transition matrix for the board position type in the English Peg Solitaire domain is shown in Figure 5. The matrix has a row and a column for each transition that a board position can go through. A cell in the matrix is crossed if the input data sees two transitions in sequence row label and then column label.

Missing Values

In this section, we discuss how we deal with missing information in the absence of noise. It is important to discuss this first, as we use the techniques here as a sub-procedure when reasoning about plans with noise. To generate the state machines, we first split the input plans around the missing information, so that there are now a greater number of shorter input plans, with no missing information. Since we assume no noise, then we can generate *LOCM* state machines, based on

	e-m.1	n-m.1	n-m.2	n-m.3	c.1	c.2	c.3
end-move.1		X	X			X	
new-move.1				X			X
new-move.2				X			X
new-move.3	X				X		
continue.1				X			X
continue.2				X			X
continue.3	X				X		

Figure 5: The transition matrix for the English Peg Solitaire domain. The names of the transitions are abbreviated on the column labels in order to preserve space.

the parts of the input plans with no missing information. We then employ the standard *LOCM* algorithm discussed above to learn the state machines, along with their state parameters.

After this process, we then use these *LOCM* machines to fill in the missing information. In order to deal with missing information, we rely on a constraint encoding of the input plans and the state machines generated by the *LOCM* system. This constraint model is a matrix model of each plan, in which the rows correspond to time-stamped variables for each time t: the action label, the action parameters, the object *LOCM* states and the values of the state parameters. The constraints are conceptually similar to those used in the constraint-based planners SeP (Barták and Salido 2011) and Constance (Gregory, Long, and Fox 2010).

If an object is in the argument of an action, then it must transition in the correct way that its *LOCM* state machine defines. Constraints are posted to ensure that state parameters appear in the correct arguments of the actions. If an object is an appropriate filler for a missing value then a conditional constraint updates its state if is selected to fill the missing value. If an object is inappropriate or not selected then its state is unchanged between timesteps. Similarly, for missing action names, conditional constraints are posted for each of the possible action fillers. Because much of the plan is known, the vast majority of the timeline can be filled in immediately, leaving only the states in which there are gaps in the operator name and arguments. Figure 8 shows an example of a timeline, for the plan in Figure 1 a) with the missing parameter from 1 b). The missing parameter was in the *end-move.1* position of the final action, and is highlighted in bold. The only consistent value that can occupy this position is the *p1-1* object: this can be seen by the fact that the object performing the *end-move.1* transition starts in the *occupied active* state, and only *p1-1* meets this condition.

Using this type of approach to finding missing data leads to two risks. Firstly, that once the plans are split into smaller plans, there will not be enough data to correctly learn the *LOCM* state machines. However, *LOCM* typically only requires a small amount of data to learn correct domain structures this is unlikely to be an issue. The other risk to this kind of approach is that there are multiple consistent objects which could take the place of the missing argument, which is an unfortunately an unavoidable problem. An algorithmic

```
        ▷ The Missing Value Model Acquisition System
function missingValueModel(Π : a collection of plans)
    $\Pi' \leftarrow$ split plans on missing information
    $M \leftarrow LOCM(\Pi')$
    $M_c \leftarrow \mathrm{constructConstraintModel}(M, \Pi)$
    if solve($M_c$) is consistent then
        return solve($M_c$)
    else
        return inconsistent
    end if
end function
```

Figure 6: The missing value model algorithm. The function constructConstraintModel returns the constraint model described above.

description of how to perform domain model acquisition in the presence of missing information is shown in Figure 6. We now change focus to discuss how we use these results to help deal with problems involving noisy data.

Noisy Data

Consider the plan fragment in Figure 7, with a single mistake in the parameters (where p2-1 in action 5 should be p2-2). By simply changing a single object parameter, the transition sequence of two objects are corrupted: the swapped in object and the swapped out object (in this case, the objects p2-1 and p2-2). Because of the error, the transition sequence for p2-1 is now *new-move.3, end-move.1, new-move.3, new-move.1* and the transition sequence for p2-2 is *new-move.2, end-move.1, new-move.2*. Each transition pair in those sequences are invalid in the true domain. Figure 9 shows the occurrence matrix (a version of the transition matrix in which the number of times each transition pair occurs is shown in each cell) for a small number of plans, including a plan which included the error from the previous example. The values corresponding to the error are highlighted. Our technique for domain model acquisition when plans may contain noise rely on forming hypotheses about which cells in the occurrence matrix may contain errors, and trying to find replacement values which still support the data.

In this section we consider the implications of noisy data on the generated *LOCM* model. There are two ways an error may add connections to the transition matrix: where `obj`✗ (an error action symbol) adds connections in the matrix, because of the new transition pairs added, or where `obj`✓ (the correct action symbol) adds a new spanning connection, because of the missing transition. We consider each

```
1: (new-move p1-2 p2-2 p2-3)
2: (end-move p2-3)
3: (new-move p4-1 p3-1 p2-1)
4: (end-move p2-1)
5: (new-move p2-4 p2-3 p2-1)*
6: (end-move p2-2)
7: (new-move p2-1 p2-2 p2-3)
```

Figure 7: Plan fragment with single error.

time	0	1	2	3	4	5
action		n-m	c	e-m	n-m	e-m
arg1		p1-0	p1-2	p3-2	p3-1	**p1-1**
arg2		p1-1	p2-2		p2-1	
arg3		p1-2	p3-2		p1-1	
zero	NM	M	M	NM	M	NM
M-prm		p1-2	p3-2		p1-1	
p1-0	OI	E	E	E	E	E
p1-1	OI	E	E	E	OA	**OI**
p1-2	E	OA	E	E	E	E
p2-2	OI	OI	E	E	E	E
p3-2	E	E	OA	OI	OI	OI
p3-1	OI	OI	OI	OI	E	E
p2-1	OI	OI	OI	OI	E	E

Figure 8: Timeline visualisation of the plan from Figure 1 a) with the missing parameter from 1 b). Abbreviations used in the table are n-m (new-move), c (continue), e-m (end-move), M-prm (the state parameter of the moving state of the zero machine, NM (not moving), M (moving), OI (occupied inactive), E (empty) and OA (occupied active). The correct value of the missing parameter and its state value is shown in bold.

of these cases below.

Added transitions The modified action becomes a new observation, `t`, in the transition sequence of `obj`✗. If the `obj`✗ and `obj`✓ are of the same type then the added transition may induce new connections due to new otherwise unseen orderings of transitions (e.g., an end-move.1 transition added, but with no preceding jump-new-move.3 or continue-3). For example, if we consider the plan fragment:

```
(end-move p3-6)
(new-move p4-0 p4-1 p4-2)
(end-move p3-6)
```

In this example, the final end-move action argument, pos-4-2, has been replaced by pos-3-6. As a result the transition sequence for pos-3-6 includes: end-move.1;end-move.1. This will contrast with any correct sequence, which will have either jump-new-move.3;end-move.1 or continue-3;end-move.1.

If they are of different sorts then, from the definition of sort in *LOCM*, `t` will not be a correct transition of `obj`✗. However, this erroneous observation will collapse the sorts together, losing any distinction between them. Also, new connections will be made from the previous transition of `obj`✗ to `t` and then from `t` to the next transition of `obj`✗.

Missing transitions There is not always enough information to detect an error directly. In these cases we might detect the error in the transition sequence of the correct argument. For example, consider the peg-solitaire sequence below, where the last parameter of a jump-new-move action has been swapped:

```
(new-move p5-2 p4-2 p3-2)
...
(new-move p4-0 p4-1 p3-6)
(move p4-2)
```

In this example, pos-3-6 does not appear in further transitions; therefore there is no direct clue that this object is incorrect. However, if we consider the correct argument: pos-

	e-m.1	n-m.1	n-m.2	n-m.3	c.1	c.2	c.3
end-move.1		61	76	**1***		12	
new-move.1				39			5
new-move.2	**1***			47			9
new-move.3	148	**1***			37		
continue.1				8			0
continue.2				12			0
continue.3	37				15		

Figure 9: The occurrence matrix for the English Peg Solitaire domain. Instead of showing just which pairs of transitions occur, as in the transition matrix, we show the number of times that the transition pairs happen (in this instance for a set of random walks). A single error is introduced to the data, and the faulty transitions are labelled with asterisks.

4-2 then this error will add the transition pair: jump-new-move.2;end-move.1, which will not be observed in a correct trace. This indicates the possibility of a missing transition, M, which extends the current transition pair, X;Y, to X;M;Y, for some M such that connected[X][M] and connected[M][Y].

Action labels We assume that the number of arguments of each action header in the plan trace is consistent with the correct action. *LOCM* requires consistent arities for the same action headers. We therefore select the most frequent arity for each action symbol and replace any others with the missing value symbol. The implication of an incorrect action symbol is that each argument will generate potentially unexpected transition sequences, essentially acting similarly to an added transition. In the case of structural redundancy there will be no clues (e.g., fly and zoom in Zeno-travel).

Building Error Hypotheses

We define an error hypothesis as a set of symbols (either action names or action parameters) in the input plans that we have supporting evidence to believe are incorrect. In order to detect noise in the plan traces, we have formulated two distinct ways of hypothesising errors in the plan symbols. These are from the occurrence matrix and from the potential state parameters. Recall from Figure 9 that noise in the data often translates into cells in the occurrence matrix that have low values. Even a single error can impact on the structure generated by *LOCM*. Firstly, it can lead to badly-formed state machines. In Figure 4, for example, observing two new-move.0 transitions incorrectly would lead to states AD and BEFC being unified. An error can lead to state parameters not being discovered. The state parameter in the 'moving' state of the zero machine in Figure 3, for example, is supported by action sequences having co-occurring parameters. Take the action sequences:

```
a) (new-move p1-0 p1-1 p1-2)  b) (new-move p1-0 p1-1 p1-2)
   (continue p1-2 p2-2 p3-2)     (continue p1-2 p2-2 p4-2)
   (end-move p3-2)               (end-move p3-2)
```

Where the continue.3 transition is an error in sequence b).

	e-m.1	c.1	c.2	c.3
continue.1	0.01	0.00	0.00	0.00
continue.2	0.01	0.00	0.00	0.02
continue.3	1.00	1.00	0.00	0.01
new-move.1	0.01	0.00	0.01	0.00
new-move.2	0.00	0.01	0.00	0.00
new-move.3	0.99	1.00	0.00	0.01

Figure 10: A state parameter ratio matrix for the moving state in the zero state machine. Each element in the grid represents the proportion of in-out transition pairs from the moving state that accompany a transition and have the same object in their respective arguments.

LOCM will reject the state parameter in the moving state of the zero machine because the object in the continue.3 argument does not match the one in the following end-move.1 argument. In an analogous way to which we build the occurrence matrix, we build a state parameter ratio matrix in order to see how frequently objects co-occur in the plans. Figure 10 shows an example of this for the moving state of the zero state machine in Figure 3. Taking the top-left corner as an example, each time an end-move follows a continue action, the object in the first argument of the end-move is the same as the first argument in the continue action 1% of the time. Strictly, in order to support a state parameter, then all in-out actions should have a pair of transitions which always coincide. However, we note that pairs with high ratios may in fact *always* coincide without the presence of noise.

Our approach for forming hypotheses about the noise in plan traces starts by examining each element of the structures generated by *LOCM* and considering to what extent it is supported in the data. There are two main outputs of the *LOCM* analysis: the transition matrix and the state parameters. In each case we can count the number of examples in the data that either support (connections in the transition matrix) or refute (state parameters) a structural element. We interpret weak support for model structures as an indication of erroneous input and use these as starting points for fixing the data. Our approach is to focus on specific weakly supported elements and then attempt to remove them. The goal here is to find small changes to the plan traces that result in structures that are well supported in the data.

We start by attempting to remove connections (pairs: $t1; t2$) from the transition matrix. We form only hypotheses on values that fall underneath a threshold value t_{Occ}. We focus on each of the plan step pairs that are represented by $t1; t2$ and attempt to find fixes that can be explained by the model without the sequence: $t1; t2$. There should always be an alternative explanation in the case of errors. This is because we assume that correct structure will be well supported in the surrounding plan traces. For state parameters, we look for high values in the state parameter ratio matrix. Again, we apply a threshold, t_{SPR} to the ratios between sequential transitions, and therefore identify the most likely relationships obscured by noise.

```
        ▷ The LC_M Domain Model Acquisition Algorithm
function LC_M(Π : a collection of plans)
    $\mathrm{Occ} \leftarrow$ the occurrence matrix
    $\mathrm{SPR} \leftarrow$ the state parameter ratio matrices
    $\mathrm{OccTrs} \leftarrow$ the transition pairs in $\mathrm{Occ} < \mathrm{t_{Occ}}$
    $\mathrm{SPRTrs} \leftarrow$ the transition pairs in $\mathrm{SPR} > \mathrm{t_{SPR}}$
    $M \leftarrow \mathrm{missingValueModel}(\Pi)$
    for all $h \in \mathrm{hypotheses}(\mathrm{OccTrs}, \mathrm{SPRTrs})$ do
        $\Pi' \leftarrow$ replace hypothesised noise$(\Pi, h)$
        if $\mathrm{missingValueModel}(\Pi')$ is consistent then
            $M \leftarrow \mathrm{missingValueModel}(\Pi')$
        else
            return $M$
        end if
    end for
    return $M$
end function
```

Figure 11: The *LC_M* algorithm. The function hypotheses returns the hypothesised structural faults in the transitions, ordered by how much support there is in the data for the fault. The algorithm either completes when all the faults have been shown to be faults or when one hypothesis leads to an infeasible model.

The *LC_M* Algorithm

So far we have described solutions to two separate problems. Firstly, how to fill in missing data from plan traces, and secondly how to identify which structural elements seem badly supported and may be artifacts of noise in the plan traces. We now show a simple, but powerful, way in which these ideas can be combined to provide an algorithm that generates domain models in the presence of both noise and missing information. The key to the algorithm is that once structural elements are hypothesised as incorrect, all of the objects that lead to the hypothesised faulty structure can be transformed into missing information. At this point, the constraint model for discovering the most likely missing objects can be employed as firstly a test of consistency over the data and (providing the changes are consistent) will provide suitable object replacements for the noise values. The complete *LC_M* algorithm is given in Figure 11.

Consider the plan in Figure 7. Suppose that the occurrence matrix in Figure 9 and the state parameter ratio matrix in Figure 10 represent the plans from which the plan is taken. The transition pair new-move.3,end-move.1 in the matrix has a ratio of 0.99, meaning that 99% of the time, these arguments were equal in sequential actions. It seems likely that this is really 100% and the 1% remaining is an artifact of the noise. We hypothesise that this transition pair is part of a state parameter and remove any argument value that does not support this hypothesis. One part of the plan in Figure 7 that does not support the hypothesis is:

```
(new-move p2-4 p2-3 p2-1)
(end-move p2-2)
```

The hypothesis leads to the removal of the new-move.3 and end-move.1 argument values.

Domain	ER	AE	TME	SPE	TME'	SPE'
Grid	0.001	3	3	0	1	0
	0.005	24	12	1	1	0
	0.010	52	20	3	7	1
	0.050	224	36	5	15	3
	0.100	497	47	5	28	4
Gripper	0.001	3	5	3	0	0
	0.005	38	27	4	4	3
	0.010	77	36	4	12	2
	0.050	355	49	4	25	5
	0.100	784	50	4	20	4
Logistics	0.001	3	6	6	8	11
	0.005	30	37	11	9	11
	0.010	66	84	13	16	12
	0.050	301	187	13	56	14
	0.100	649	214	13	63	12
Parking	0.001	3	4	3	1	0
	0.005	38	23	5	4	3
	0.010	78	36	5	10	4
	0.050	360	50	5	18	4
	0.100	785	59	5	16	5
Peg Sol.	0.001	1	1	0	0	0
	0.005	15	12	2	6	3
	0.010	34	14	2	1	1
	0.050	134	31	2	9	5
	0.100	296	34	4	7	4
Storage	0.001	3	2	3	0	9
	0.005	29	31	12	4	3
	0.010	63	65	12	16	6
	0.050	269	175	12	76	12
	0.100	583	231	12	94	12
TyreWorld	0.001	2	4	1	1	1
	0.005	23	29	2	4	1
	0.010	49	52	2	12	1
	0.050	200	106	3	21	2
	0.100	449	142	3	47	2

Table 3: The results of the empirical evaluation on noisy data for the *LC_M* system. The abbreviations in the headings are ER (Error Rate), AE (Atomic Errors, TME (Transition Matrix Errors), SPE (State Parameter Errors), TME' (Transition Matrix Errors in corrected model), SPE' (State Parameter Errors in corrected model)

```
(new-move p2-4 p2-3 _____)
(end-move _____)
```

The constraint model confirms that the data has a consistent domain model, with the state parameter, and assigns a consistent object to the missing arguments. For Figure 7, there is only one value that can occupy these parameters: p2-2.

Empirical Analysis

In this section we present an evaluation of the system. The aim is to establish how robust our approach is to missing information and noise. In order to do this we first generate a model using plans with no errors and use this as the correct model for comparison. The noisy training data sets

	e-m.1	n-m.1	n-m.2	n-m.3	c.1	c.2	c.3
end-move.1	O	X	X	O	O	X	O
new-move.1	O	O	O	X	O	O	X
new-move.2	O	O		X	O	O	X
new-move.3	X	O	O	O	X	O	O
continue.1	O	O	O	X	O	O	X
continue.2	O	O	O	X	O		X
continue.3	X		O	O	X	O	O

Figure 12: The transition matrix for the English Peg Solitaire domain with 5% noise. The 'O' entries represent the errors in the matrix.

are each derived from these plans, which means that we can determine how well our approach to correcting the errors performs. A collection of standard benchmark planning domains is used for the evaluation.

All of our experiments are run on Mac OSX version 10.11.6 using an Intel 2 GHz i7 CPU with 8 GB system memory. *LC_M* is implemented in Java (version 1.8.65) using the Choco constraint library (Prud'homme, Fages, and Lorca 2014) version 3.3.1.

We have simulated noisy data by first generating a set of action sequences for each domain and then adding noise. In each of the domains, except Grid, we have used the first 10 problems from the standard benchmark sets and generated 5 action sequences for each problem. In Grid the 5 problems were used and 10 action sequences were generated per problem. In most domains the action sequences are random walks of a length randomly selected between 1 and 100. However, for Grid and Peg-solitaire, where random walks provide poor coverage of the transitions, a goal-directed plan is used as one of the action sequences for each problem.

Each plan is passed through a channel that simulates the introduction of certain types of noise. The generated noise is governed by the probability for recording an incorrect value and can function with or without action symbol replacement. Replacement objects are drawn from the set of objects observed in the plan and the replacement action symbols are randomly selected from the observed actions.

We first evaluate our system on action sequences with increasing numbers of missing information. We then test the system on noisy plans.

Missing Information

We first test how robust the system is at filling in plans that contain only missing information. This forms an important subsystem of our complete approach and therefore provides an indication of how robust our system will be to noise in each of the tested domains. As detailed above, the plans are split around actions that contain missing information, resulting in a smaller set of plans with no noise. These plans are then used as input to *LOCM*, which induces a transition matrix and set of state parameters. A CP model is then constructed using the original plans with variables for each missing value. This model enforces the transition and state

Error rate	0.005		0.01		0.05		0.1	
#Errors	#E_{start}	#E_{end}	#E_{start}	#E_{end}	#E_{start}	#E_{end}	#E_{start}	#E_{end}
Grid	29	2	68	5	337	51	711	240
Gripper	47	0	105	0	524	5	1076	13
Logistics	36	1	86	0	408	22	715	74
Parking	46	0	113	0	499	42	1098	218
Pegsol	21	3	36	10	207	67	408	136
Storage	39	0	69	3	366	17	717	121
Tyreworld	33	7	70	32	303	122	653	446

Table 4: The number of errors before (#E_{start}) and after (#E_{end}) parameter filling for several domains and error rates.

parameter constraints. If the split plans provide sufficient information to construct a correct model then solutions for the CP should be valid action sequences.

There are several ways that the CP model can have different output from the original plan sequences. The planning model will quite often allow alternative values for the same missing value. For example, consider a truck with 2 packages in it. If a put down action is missing the package and no further references are made to these packages then there is no information to distinguish them. As symbols are removed from the plan more symbols will appear equivalently appropriate for filling a specific action name or object. This is particularly relevant as we are only considering the dynamics of the model. As there are typically more than one argument for an action it is less likely, although possible that action symbols can be substituted in a similar manner.

As well as making equivalent replacements, the CP can also fail to produce a solution. This happens when the missing information in the context of a specific object results in either the type of the object becoming undetectable (only appears where the action symbol is missing), or the object becomes completely unobserved. In the former case, the original plans are used to guess its type as the most commonly observed type (this approach is helpful in the case of noise). Therefore the CP may fail because the correctly typed object may not be available.

Table 4 presents the number of errors in the plans (here a missing symbol is counted as an error) before and after our system has been used to fill the parameters. The results show that the system is often able to complete the majority of the missing information. It should be noted that we are reporting symbol errors as opposed to the number of actions with errors in them. For example, in the Parking domain at the 0.1 error rate, 36% of actions had at least one error.

As the error rate increases the performance degrades. This is expected because as the number of missing values increases, the number of alternative plans increases. Also, the increased missing information will increase the number of objects that become obscured, resulting in fewer of the CP models being solvable. In Gripper, Logistics, Parking, Peg-solitaire and Storage the number of failed models at each rate was 10 or fewer, leaving at least 40 completed plans that can be used to learn a model. In Tyreworld and Grid there were 21 and 14 failed models at the 0.1 rate.

Another factor is that as the amount of missing information is increased and the plans are broken up into smaller pieces the number of observations of each transition pair reduces. This can lead to erroneous state parameters being induced, or, in the case of no observations, missed transition pairs. As a result, in Tyreworld and Grid, *LOCM* induced tighter models than the original domain models at the 0.1 and 0.05 rates and also for Grid at the 0.01 rate. These extra constraints can prevent the correct objects from being filled back into the plans.

Noisy Data

In this section we evaluate the complete system on plans with noisy data. The results for missing information indicate that the underlying missing information filling system is fairly robust in most of the tested domains. In this part of the evaluation the system first identifies weakly supported structural elements and then attempts to modify the input plans so that the structure is no longer supported by the data. In this section we therefore examine whether *LC_M* is able to isolate the erroneous structural elements without breaking those parts of the structure that were not effected by the errors.

We have used the same baseline training data and simulated noise using symbol replacement rates of: 0.001, 0.005, 0.01, 0.05 and 0.1. In this evaluation we have considered noise in the arguments of the actions and not in the action name. The system starts by using the noisy plans to generate the transition pairs and state parameter structures. The set of hypotheses for weakly supported structure elements are created from the set of all transition pairs and partial state parameters. These are ordered using a notion of how well they are supported in the data. For state parameters this uses the number of matched parameters against not matching parameters; and for transitions pairs this uses the number of occurrences of the transition pair against other pairs. A threshold value of 0.5 was used to prune the weakest of these hypotheses. E.g., a partial state parameter hypothesis is pruned if it is supported by less than half of the occurrences in the data. We test each hypothesis by first of all identifying the actions and the specific arguments in each plan that are inconsistent with the hypothesis. These arguments are replaced by the missing information symbol. The missing information filling system is then used to determine if there exist completions of the plans that support the hypothesis. If there are then the hypothesis is accepted, otherwise the hypothesis is rejected. The next hypothesis is then tested.

Table 3 presents the number of transition matrix and partial state parameter differences, as well as the underlying symbol differences. It shows that for low levels of noise, *LC_M* typically corrects the majority of structural errors introduced. As the noise level increases to 10%, fewer structural errors are corrected. However, their number is still typically reduced by half. State parameters are less well supported than transition matrix errors, in some cases introducing errors, even. However, several domains do see reductions in state parameter errors. Note that it is the transition matrix that determines the state machines in *LOCM* and so it is in some ways more important to reduce the errors in the transition matrix.

It may be informative to consider the effect of 5% noise, in order to understand why it becomes so hard to correct errors at this level. Figure 12 shows the transition matrix from Figure 5 overlayed with the transitions induced by this level of noise. As can be seen, there are only three transition pairs that are correctly identified as missing. Despite this, 22 of these erroneous transition pairs were detected and removed by *LC_M*. Clearly the fact that nine invalid transition pairs remain is problematic, and future work will determine whether this number can be reduced still further. The interaction between different hypotheses provides one of the most problematic difficulties: the decisions made in supporting an early hypothesis can make it impossible to support a later one, for example. One solution to this issue could be to consider the entire current hypothesis set *simultaneously*.

Related Work

In addition to the *LOCM* family of algorithms, there is a great amount of work in planning domain model acquisition without noise and missing information, from the TRAIL system (Benson 1996) to Opmaker (McCluskey et al. 2009; Richardson 2008), ARMS (Wu, Yang, and Jiang 2007), LAMP (Zhuo et al. 2010) and ASCOL (Jilani et al. 2015). To our knowledge, there is no other domain model acquisition system targeting noisy and incomplete input, except for (Mourao et al. 2012), which depends on intermediate state information. In addition to the planning community, there is wide and active interest in automatic model acquisition in many of the sub-fields of combinatorial search and beyond, for example in constraint satisfaction (O'Sullivan 2010; Bessiere et al. 2014), general game playing (Björnsson 2012; Gregory, Björnsson, and Schiffel 2015), and software engineering (Reger, Barringer, and Rydeheard 2015).

Conclusions

Modelling planning domains is a difficult and time consuming activity in general. Tools that can assist a domain expert in formulating a representation of their planning problem can save time and widen the usage of planning technologies. Domain model acquisition systems allow models to be learnt from existing plans. However, the process that observes the input plans may be prone to errors itself, making the input plans an unreliable source due to noise and missing information. In this work, we have presented a technique for performing domain model acquisition in the presence of noise and missing information.

LC_M returns the most likely underlying deterministic domain based on the frequency of support for various domain structures, such as the occurrence matrix cells and *LOCM* state parameters provided in the input data. The result of this is a system that learns planning domain models in a larger number of real-world situations. Although the performance of *LC_M* is already promising, structural flaws will remain for input data generated using a high-noise process, though reduced. An important future line of research will focus on the limits of such an endeavour: whether more errors can be removed based on better hypothesis schemes, or whether there is a technical limit beyond which errors cannot be distinguished from valid structure.

Acknowledgements

This work is supported by EPSRC Grant EP/N017447/1.

References

Barták, R., and Salido, M. A. 2011. Constraint satisfaction for planning and scheduling problems. *Constraints* 16(3):223–227.

Benson, S. S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation.

Bessiere, C.; Coletta, R.; Daoudi, A.; Lazaar, N.; Mechqrane, Y.; and Bouyakhf, E. H. 2014. Boosting Constraint Acquisition via Generalization Queries. In *European Conference on Artificial Intelligence*.

Björnsson, Y. 2012. Learning Rules of Simplified Boardgames by Observing. In *European Conference on Artificial Intelligence*, 175–180.

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, 42 – 49.

Cresswell, S. N.; Mccluskey, T. L.; and West, M. M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *ICAPS*, 338 – 341.

Cresswell, S.; McCluskey, T.; and West, M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28(2):195 – 213.

Gregory, P., and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling*, 97–105.

Gregory, P.; Björnsson, Y.; and Schiffel, S. 2015. The GRL System : Learning Board Game Rules With Piece-Move Interactions. In *GIGA*.

Gregory, P.; Long, D.; and Fox, M. 2010. Constraint Based Planning with Composable Substate Graphs. In *European Conference on Artificial Intelligence*, 453–458.

Jilani, R.; Crampton, A.; Kitchin, D. E.; and Vallati, M. 2015. Ascol: A tool for improving automatic planning domain model acquisition. In *AI*IA 2015, Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, September 23-25, 2015, Proceedings*, 438–451.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning models from natural language action descriptions. In *ICAPS*.

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.

Mourao, K.; Zettlemoyer, L.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Uncertainty in Artifical Intelligence*, 614 – 623.

O'Sullivan, B. 2010. Automated modelling and solving in constraint programming. In *AAAI*.

Parkinson, S.; Longstaff, A.; Crampton, A.; and Gregory, P. 2012. The Application of Automated Planning to Machine Tool Calibration. In *International Conference on Automated Planning and Scheduling*.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2014. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Reger, G.; Barringer, H.; and Rydeheard, D. 2015. Automata-based Pattern Mining from Imperfect Traces. *ACM SIGSOFT Software Engineering Notes* 40(1):1–8.

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.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review* 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. *Artificial Intellgence* 174:1540–1569.