Incremental Domain Model Acquisition with a Human in the Loop

Alan Lindsay, Ronald P. A. Petrick
Automated Planning Lab,
Department of Computer Science,
Heriot-Watt University, Scotland, UK
{alan.lindsay,r.petrick}@hw.ac.uk

Abstract

The creation and maintenance of a domain model is a well recognised bottleneck in the use of automated planning; indeed, ensuring a planning engine is fed with an accurate model of an application is essential in order that generated plans are effective. There have been great advances in modelling assisting and model generation tools, including domain model acquisition tools that can handle noisy input. One of the key issues with many of the approaches to domain model acquisition (especially those learning from noisy observations), is that they create imperfect models. In this work we consider the development of the planning model as an iterative process. We start with a domain expert and an empty planning model. We expect that the expert has an accurate model of the problem domain and can provide (noisy) example plans and also validate plans. Our approach involves maintaining a partial model as a set of structural elements, inspired by the *LOCM* approach to domain model acquisition. The user's input is used to incrementally improve the partial model, selecting an appropriate model that explains the observations so far. We present a preliminary evaluation, which aims to assess the feasibility of approach.

Introduction

In Automated Planning the planning model plays a fundamental role. However, modelling has been identified as a bottleneck, due to the skills required to develop these models. This has inspired a variety of methods for supporting the authoring of domain models, including frameworks similar to Integrated Development Environments for use by software engineers, e.g., the GIPO (Simpson, Kitchin, and McCluskey 2007), itSIMPLE (Vaquero et al. 2007) and KIWI (Wickler, Chrpa, and McCluskey 2014) systems. These modelling tools are useful for rapid development of domains by an experienced domain modeller. Frameworks also exist to refine (Lindsay et al. 2020) or extend (Porteous et al. 2021) existing planning models, thus reducing the burden of modelling a complete domain model. Another avenue of research to aid in the modelling process is based on learning models from observations: namely that of domain model acquisition.

Domain model acquisition is the problem of learning a formal domain model of a system from some form of input data. A key issue in domain model acquisition is to learn usable planning models from noisy data. This is because in real world scenarios it can be difficult to ensure clean input data. However, it is a challenging problem as the candidate space for models grows quickly with properties of the domain. This combines with the requirement of accuracy in the domain model acquisition process, as even a single error in the final model can result in incorrect plans or unreachable goals.

```
a) (move truck loc-1 loc-2)  b) (move truck loc-1 loc-2)
   (load box-1 truck loc-2)     (load box-1 truck loc-2)
   (move truck loc-2 loc-3)     (move truck box-1 loc-3)*
   (unload box-1 truck loc-3)   (unload box-1 __ loc-3)**
```

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

In this work we notice that the use of such a model will require supervision and a user who can step in and provide plans when the system fails. We build this role explicitly into our framework. Thus we consider an incremental process of domain model acquisition, where the system builds knowledge and examples about the domain through a series of episodes. Starting from an empty model, the user's input (noisy plans and plan validation) is used to maintain a partial model, which captures the system's knowledge of the domain. We base our approach on the *LOCM*-family of domain model acquisition approaches, which synthesise planning models from action sequences (with total plan costs). The partial model is represented by a set of structural elements inspired by the *LOCM* approach to domain model acquisition. In our preliminary evaluation we examine the feasibility of our approach and use our system to learn models in the gripper domain.

In the next section we provide background to the *LOCM* approach, and then overview the related work; we present an incremental domain model acquisition framework and describe how it is used to learn dynamic models; the process for completing the dynamic models is then presented, before we consider some of the issues encountered so far; we present a preliminary evaluation and finish with the conclusions.

Background

In this section we provide an overview of the *LOCM* approach and then present some terminology.

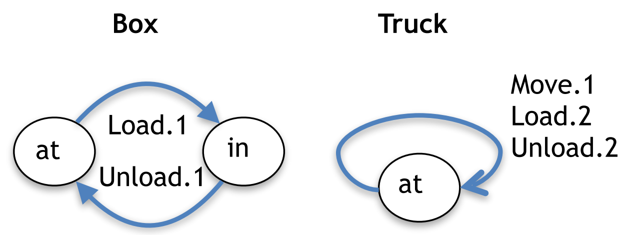

Figure 2: The finite state machines derived by *LOCM* for a transportation domain. There are two state machines: one for the box type, and another for the truck.

The *LOCM* system (Cresswell, McCluskey, and West 2009, 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 a simple transportation domain, with trucks tasked with redistributing packages on a network of locations. There are three operators: `move`, `load` and `unload` and an example plan is presented in Figure 1 a.

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 the associated object's 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. For the plan in Figure 1 a) for example, the object `box-1` has the transition sequence *load.1, unload.1*. In case the domain relies on an implicit zeroth parameters, a dummy object is added to each action, resulting in an extra transition for each action. This is important, as the structure of the plans can carry extra object-independent information about the domain structure (e.g., the implicit robot in Gripper).

The domain model that *LOCM* learns for this domain can be described by the state machines in Figure 2 (omitting the location and zero space machines as these are fully permissive), where there are two state machine that capture interesting structure: one for the box and one for the truck types. 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 (t_0, t_1) the end state of the t_0 transition is the start state of the t_1 transition. 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. We refer to the set of state transition pairs as a transition matrix (TM).

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 machines, typical examples include the location of a truck in a logistics domain. 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 box type of Figure 2 are shown in Table 1.

Noisy Plans

In (Gregory, Lindsay, and Porteous 2017), an approach to extending *LOCM* for noisy domains was presented. The intention in that work was to use a constraints based approach to incrementally remove transition pairs and add state parameters, using constraints determined by the noisy input plans to determine whether the modification was valid. Although the approach could reduce the number of errors in the model (and plans), the eventual model would typically include inaccuracies. In this work we develop a framework that places a user in the loop, with the key aim that the eventual model is useful for planning.

In this work, we consider an observed plan, π^o, is a sequence of observed actions, $\pi^o = a_0^o, \ldots, a_m^o$ and a collection of plans for the same domain can be denoted, $\Pi^o = \pi_0^o, \ldots, \pi_n^o$. Each of these sequences of actions is associated with a *true* sequence that was being observed, which we denote π^T. We use Π^T for a collection of *true* plans and a^T for the *true* action. This is related to the problem formulation in (Zhuo and Kambhampati 2013).

Related Work

Within the field of domain model acquisition a variety of input types, used processes and target representations have been investigated. Most systems use information, such as action sequences, predicates, initial and goal states and possibly intermediate states, e.g., ARMS (Wu, Yang, and Jiang 2007) and LAMP (Zhuo et al. 2010). More recent systems use sequences of images (Asai and Fukunaga 2018) and natural language action descriptions (Lindsay et al. 2017) and have examined learning and refining models from noisy data (Mourao et al. 2012; Lindsay et al. 2020). Approaches have targeted a wide range of target fragments of the PDDL language, from propositional (Wu, Yang, and Jiang 2007; Cresswell and Gregory 2011), including ADL (Zhuo et al. 2010); to learning action costs (Gregory and Lindsay

Transition	In Parameter	Out Parameter
load.1	load.2	load.3
unload.1	unload.3	unload.2

Table 1: Table of the state parameter transitions for the box type of Figure 2.

2016) and numeric constraints (Segura-Muros, Pérez, and Fernández-Olivares 2018).

Our work focuses on the *LOCM*-family of approaches, e.g., (Cresswell, McCluskey, and West 2009; Cresswell and Gregory 2011; Gregory and Cresswell 2015; Gregory and Lindsay 2016). A key focus in these works is to use a minimal amount of input, and using action headers as the input. Most other systems use other information, such as predicates, initial and goal states and intermediate states, e.g., ARMS (Wu, Yang, and Jiang 2007) and LAMP (Zhuo et al. 2010). In these systems static predicates are typically learned along with the dynamic predicates, as they form part of the state trace information. *LOP* and *ASCoL* (Jilani et al. 2015) provide alternative approaches for deriving static information. *LOP* provides a general approach to identifying missing static predicates and we have modified the approach to support our approach (Lindsay 2021).

In terms of framework, it has been common to assume that domain model acquisition is an isolated preprocess to planning. However, in (Ng and Petrick 2019) they consider an incremental approach to learning in a reinforcement learning setting. User in the loop has also been considered in both modelling assisting tools (Simpson, Kitchin, and McCluskey 2007; Vaquero et al. 2007; Wickler, Chrpa, and McCluskey 2014) and domain model acquisition (Walsh and Littman 2008). In (Walsh and Littman 2008) they considers an oracle, and consider cases where the oracle can both provide plan traces (including state information) and validate action sequences (including state information). They do not consider noisy input and also require complete intermediate state information. The expert in the loop framework that we adopt tackles the model reconciliation problem posed in (Sreedharan et al. 2020). The expert knows the model that is required and the planner initially has an empty model. Through communication and model updates, the system's model is improved so that it is closer to the target model. In (Sreedharan et al. 2020) the system produces explanations to assist the user in making changes to the model. In our approach, the system makes the model changes, reducing the burden of requiring a domain engineer.

Incremental Domain Model Acquisition Framework

We observe that in many cases, an approach to domain model acquisition that results in a noisy model will require some form of supervisor. In our framework we make the role of the domain expert (the supervisor) explicit. The supervisor plays a similar role as the domain engineer in (Sreedharan et al. 2020), except in our approach, the system is performing the model updates and the expert is able to perform some simpler functions such as plan validation and suggesting plans. We assume that the domain expert has an accurate model of a system, M^T (the target model). The intention is that during initial iterations the expert will need to provide feedback on the system's attempts and ultimately solve the problems. As more observations are made, the system's knowledge of the model builds and the generated plans will be more effective. A key aspect of this work is the use of structural elements in order to organise the partial knowledge about the model. Over time it is expected that the user's plans will include any of these structural elements that are required to capture a model effectively. The details of these structures and how they are used to generate a model is presented in the next section.

Algorithm 1 AN INCREMENTAL MODEL LEARNING ALGORITHM, WITH A DOMAIN EXPERT IN THE LOOP (uLOCM):

```
 1: function uLOCM
 2:     $K \leftarrow []$; $result \leftarrow \varnothing$
 3:     while true do
 4:         $p, K^p \leftarrow$ Usr_select_problem()
 5:         $K \leftarrow K \cup K^p$
 6:         $M \leftarrow$ Sys_make_best_model($p, K$)
 7:         $needsUsrPlan \leftarrow FALSE$
 8:         $\pi \leftarrow$ Sys_get_plan($p, M$)
 9:         if $\pi$ then
10:             $result \leftarrow$ Usr_validate_plan($p, \pi$)
11:             Sys_update_K($p, \pi, result, K$)
12:             if $result == FAILED$ then
13:                 $needsUsrPlan \leftarrow TRUE$
14:             else if $Usr\_has\_alternative(p, \pi)$ then
15:                 $needsUsrPlan \leftarrow TRUE$
16:             end if
17:         end if
18:         if $\pi == \varnothing$ OR $needsUsrPlan$ then
19:             $\pi' \leftarrow Usr\_get\_plan(p)$
20:             Sys_noisy_update$_K$($p, \pi', K$)
21:         end if
22:     end while
23: end function
```

The scenario we envisage involves a single problem domain, with a set of environments, each distinguished by a special symbol. These may be alternative system starting configurations, floors of a building, or towns for supporting logistics. We expect that for each starting configuration, the set of applicable actions is available. This can be considered as background knowledge and is specified once per new environment. An important concept that unpins the *LOCM* approaches is the use of actions and action sequences as the communication between the user and the system. This was achieved for the dynamic model in (Cresswell, McCluskey, and West 2009) and for the initial and goal states in (Cresswell, McCluskey, and West 2013). We have extended this idea in (Lindsay 2021) by allowing the model's static relationships to be specified by the reachable actions (see the section 'Completing the Partial Model').

User Functions

We expect that the domain expert will be an active participant in the loop, capable of performing three functions: indicating the initial/goal state, proposing a plan and validating a system plan. We observe that errors are common during text entry. For example, noise may be introduced when the user is entering a complete plan. However, where the system can provide alternatives that the user must select between, we assume the user can provide accurate information.

For each episode the user can specify the initial (goal) states by indicating for each object which of the applicable actions can be performed first (last). For the goal the user only needs to specify values when they want to constrain the object's final action (e.g., only for objects with a goal). For more details refer to Section 'Completing the Partial Model'.

If the system fails to create an executable plan, or if the user decides to provide an alternative the user will be asked to propose a plan, π. We anticipate that in communicating their plans the user may introduce noise, π^o. To minimise the assumptions that are made about the distribution of the noise, we limit the use of the plans. Firstly, we use the plans to provide confirmation that the problem is solvable. Secondly, we assume that the plans (over time) will exhibit examples of all of the structural elements required to define an accurate model.

Given a plan, π, generated by the system, the expert will be asked to validate the plan and indicate the failure point, if appropriate. Note that the user is never expected to provide a noiseless plan. Each of these validated plans provides both a positive example (the sequence of actions before the failure point) and a negative example (the complete sequence).

The *uLOCM* Approach

The pseudo-code for the approach is presented in Algorithm 1 and relies on user functions (distinguish by a prefix 'Usr') and additional system function (prefix 'Sys'). The system has an initially empty model and builds its knowledge of the domain incrementally through successive episodes (lines $3-22$). Central to the approach is the system's growing knowledge about the problem domain (variable K), and the use of this knowledge to propose a model (`Sys_make_best_model`, line 6). Each episode starts (line 4) with the user selecting an episode (and indicating the first and (where appropriate) final actions for each object). The system then makes its best fit model (see the next section) and attempts to generate a plan (see section 'Completing the Partial Model'). If a plan is generated (line 9) then it will be validated by the user and the knowledge of the system will be updated to include the result of the validation. If the plan is validated then if the user has no alternative to propose (line 14) then the next episode will begin. We simulate the user wanting to make an alternative plan suggestion with a probability with a parameter (not used in the evaluation). Otherwise, or if a plan was never found, the user will provide a (noisy) plan and the system's knowledge about the domain will be updated (lines $19-20$).

Partial Dynamic *LOCM* Model

At the heart of the *uLOCM* approach is the incremental building of the system's knowledge about the problem domain. A key aspect of our approach is the use of structural elements in order to organise this partial knowledge. This section describes these structures used to maintain partial knowledge, the maintenance of these structures and the creation of a dynamic model that is both consistent with the observations, but also explains the negative examples from the user validations.

Algorithm 2 GIVEN THE CURRENT KNOWLEDGE ABOUT THE PROBLEM, THIS ALGORITHM FINDS A MODEL THAT IS CONSISTENT WITH THE OBSERVATIONS AND EXPLAINS THE NEGATIVE EXAMPLES (MAKEDYNAMICMODEL):

```
 1: function MakeDynamicModel($K$)
 2:     $TMs \leftarrow getSpaceOfTMs(K)$
 3:     $candidateTMs \leftarrow$ enforceValTMs($K, TMs$)
 4:     for all $cTM \in candidateTMs$ do
 5:         $FSMs \leftarrow$ makeFSMs($cTM$)
 6:         $sSPs \leftarrow$ getSpaceOfSPs($K, FSMs$)
 7:         $candidateSPs \leftarrow$ pruneRftPSPs($K, sSPs$)
 8:         for all $cSP \in candidateSPs$ do
 9:             $M \leftarrow$ makeModel($FSMs, cSP$)
10:             if testModel($M$) & coversExamples($K, cTM, cSP$) then
11:                 return $M$
12:             end if
13:         end for
14:     end for
15: end function
```

Structures for Recording Partial Knowledge

If we denote the set of transitions, $\mathbf{T}$, a set of confidence values $\mathbf{C} = \{No, MaybeNot, Unknown, Maybe, Yes\}$, we define the current knowledge about the problem structure as $K = \langle K_{PTM}, K_{PSP} \rangle$, with $K_{PTM} : \mathbf{T} \times \mathbf{T} \mapsto \mathbf{C}$ and $K_{PSP} : \mathbf{T} \times \mathbf{Z} \times \mathbf{T} \times \mathbf{Z} \mapsto \mathbf{C}$. K_{PTM} records the current knowledge about observed and valid transition pairs. The possible values are $\{Unknown, Maybe, Yes\}$, with each transition pair starting on $Unknown$. K_{PTM} records the current knowledge about observed and refuted partial state parameters, and each partial state parameters can map to the values: $\{No, MaybeNot, Unknown\}$. We use partial state parameters to record the knowledge about state parameters because they are independent of any particular transition matrix.

Updating the Partial Knowledge

The main types of information are a validated plan or user provided noisy plan. Each noisy plan is recorded as possible structures, which will be explored during the construction of a dynamic model, below. The validated plan is broken into two. The steps prior to the failure point are used as a positive example, which can be used to confirm structure. The second part is collected and used as a negative example, which must be explained by any created model.

Given a noisy plan π^o, we update both K_{PTM} and K_{PSP}. We analyse the sequence, using the *LOCM* approach to uncover all transition pairs (PTMs), t_1, t_2. If a transition pair has not been observed: $K_{PTM}(t_1, t_2) == Unknown$ then we update the knowledge: $K_{PTM}(t_1, t_2) = Maybe$. Otherwise no change is made. This indicates that the pair is potentially valid and can be explored in the creation of the model. We similarly analyse the sequence, using the *LOCM* approach to uncover all information about every partial state parameter (PSP), t_1, p_1, t_2, p_2. If refuting observations are made: where consecutive actions for an object o match with t_1 and t_2, but the parameters at p_1 and p_2 do not match, and $K_{PSP}(t_1, p_1, t_2, p_2) == Unknown$ then we update the knowledge: $K_{PSP}(t_1, p_1, t_2, p_2) = MaybeNot$.

Given a validated sequence π, we can also update both K_{PTM} and K_{PSP}. We analyse the sequence, using the *LOCM* approach to uncover all transition pairs (PTMs), t_1, t_2. If a transition pair has not been validate: $K_{PTM}(t_1, t_2) \neq Yes$ then we update the knowledge: $K_{PTM}(t_1, t_2) = Yes$. Otherwise no change is made. This indicates that the pair is valid and will be part of any candidate model. We similarly analyse the sequence, using the *LOCM* approach to uncover all information about the partial state parameters (PSPs), t_1, p_1, t_2, p_2. If refuting observations are made: where consecutive actions for an object o match with t_1 and t_2, but the parameters at p_1 and p_2 do not match, and $K_{PSP}(t_1, p_1, t_2, p_2) \neq No$ then we update the knowledge: $K_{PSP}(t_1, p_1, t_2, p_2) = No$.

Creating a Consistent and Example Covering Dynamic Model

The next step is to generate a single dynamic model that is consistent with the partial knowledge and explains the negative examples. The pseudo-code for our approach is presented in Algorithm 2 and constitutes the specification of and search through a space of candidate dynamic models. The creation of the model consists of two main loops: firstly the selection of a transition matrix and secondly the selection of a set of state parameters. This division is natural as state parameters can only be defined in terms of a specific state, which relies on a specific set of transition pairs.

The first step (line 2) is to get the space of TMs, which is defined using the partial knowledge. This space consists of all combinations of TMs for PTMs that have been observed in any of the traces (i.e., they have value $Maybe$ or Yes in K_{PTM}). However, if we have knowledge of definite PTMs (=Yes) then these must be part of any candidate TM. Any inconsistent candidates are pruned (line 3). The algorithm then loops through each candidate TM and attempts to find a completion of the model that can be validated (lines $4-14$). In our implementation we start with the most complete TM (the least constrained model).

The finite state machines (FSM) for the candidate TM is generated (see the background) and the space of SPs is discovered for each of the FSM states. This is achieved by merging PSPs that are not fully refuted (i.e. an PSP t_1, p_1, t_2, p_2 with $K_{PSP}(t_1, p_1, t_2, p_2) \neq No$), which is all PSPs where a counter example has not been seen in the validated sequences. Of this space, we insist on enforcing the SPs if there are no countering examples (even in the noisy traces). That is if the PSPs that form an SP do not have $MaybeNot$ values or No values in K_{PSP} we enforce the constraint (line 7). The algorithm then loops through each of the remaining candidate SPs and tests whether the resulting model is feasible. In our implementation we start with the empty set of SPs (the least constrained model) and to improve efficiency we only add additional constraints if the resulting model leads to solvable problems (adding additional constraints to an unsolvable problem will not lead to a solvable problem).

The dynamic model is made by adding the SPs to the FSM states and outputting the resulting structures (Cresswell, McCluskey, and West 2009). Problem models are also created and we describe the process of specifying the initial state, goal and static relationships in the following section. The model is then tested in two ways (line 9): the model is used to solve the problems that we have observed to be solvable (i.e., the ones that the expert has already provided a solution for); and the structures are tested to ensure that all of the negative examples are covered. If the candidate model passes both of these tests then it is returned.

Completing the Partial Model

In the previous section we presented our approach for selecting a dynamic model to explain a set of observations. In order to complete the model requires the specification of the initial and goal states and the static relationships (as is the case with other versions of *LOCM*). Our aim in this work was to use only actions and action sequences for communication between the user and system. In this section we we briefly outline how this is done for the initial and goal states (as presented in (Cresswell, McCluskey, and West 2013)) and then describe our approach to specifying static relationships (Lindsay 2021).

Initial and Goal State Specification

In (Cresswell, McCluskey, and West 2013) an approach is presented in order that the initial (dynamic part) and goal states of the model can be specified without using the automatically generated state labels. In their approach they use the actions that will be used first by each object for the initial state and similarly last actions for objects with goals. These actions are then used to identify the appropriate machine state for each object and select the parameters for state parameters. For example, if one of the first actions of object $Box1$ is ($loadbox1\ truck1\ loc1$) then given the state machines in Figure 2, the box is in the at state.

Static Relationship Learning

We have developed an approach for identifying static predicates from a correct model of the system's dynamics (e.g., as typically output by *LOCM I* or *II*) and the set of reachable actions. Our approach is based on the *LOP* approach (Gregory and Cresswell 2015), which learns static relations by comparing optimal input plans with the optimal plans found using the partial domain model. The comparison between the optimal plans is used as a Boolean function that provides evidence to support the hypothesis that a static relation has gone undetected. Instead of using optimal plans, we define an alternative Boolean function, which identifies the missing constraints by comparing the actions allowed by the partial model with the set of reachable actions. This has two benefits: firstly, we do not rely on the user (or our planner) being able to provide optimal solutions; and secondly, we can use the set of applicable actions in any future scenario to calculate the ground static relationships for a specific problem.

Acquiring Positive and Negative Examples for Static Relationships Static predicates act as constraints that classify action headers as valid and invalid. In this work we have considered learning this classifier using positive and negative examples. As presented in the previous section, we have a partial model, $\mathbb{M}^P$, which we assume accurately captures the dynamics of the target system. We will use $\mathbb{M}^T$ to denote the target system.

The applicable actions are provided as input for each scenario (see Section 'The *uLOCM* Approach') and can be used directly as the positive examples, E^+. We assume that there are a set number of environments from which problems are drawn and that providing the applicable actions for these environments can be done accurately offline.

The negative examples can then be identified by expanding states using the partial model and recording those not in the applicable actions. In particular, we consider the frontier between sequences of actions that are valid (sequences of reachable actions generated by the partial model) in the target system and the first action that is not allowed in the target system. Consider an action trace, $s_0, a_1, \ldots, a_n, s_n$, which is valid in the partial model, $\mathbb{M}^P$. For some index, k, we assume that all actions, a_i $(i < k)$ are in the target actions $(a_i \in \mathbb{A}^T)$. Consequently, s_{k-1} is a valid state in $\mathbb{M}^T$, because we have assumed that the dynamics of the model are captured correctly. If the next action, a_k, is not in the applied actions $(a_k \notin \mathbb{A}^T)$ then we call a_k a *frontier* action. We call these *frontier* actions, as they indicate the separating line between the valid part of an action sequence (that transitions only using actions in the target model) and the remainder of the invalid sequence. Notice that any frontier action must have a missing constraint in $\mathbb{M}^P$. As such these frontier actions become the negative examples, E^-. The method outputs the sets of positive and negative examples.

Identifying Static Relationships From Examples Following (Gregory and Cresswell 2015), we aim to find tuples of parameters for each action that concisely capture the static relation. The first step is to identify a (potentially empty) tuple, for each action, which identifies the action parameters that must be involved in the static.

Definition 1 (Static Parameter Tuple)**.** A static parameter tuple is a tuple, $T = (i_0, \ldots, i_m)$, for an action, $a = (opname, p_0, \ldots, p_n)$, which identifies the action parameters, $(p_{i_0}, \ldots, p_{i_m})$ that must be involved in the static. In this section we adapt the *LOP* approach in order to use the positive and negative examples identified above.

Part 1: Parameters Involved in Static Relationships

The first stage in the *LOP* approach involves identifying the minimal static parameter tuple for each action. Our algorithm is presented in Algorithm 3 and generalises Algorithm 2 in (Gregory and Cresswell 2015). The starting point is to assume that all of the tuples are involved in the static parameter tuple. For example, the tuple $\{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ would be the starting point for the *move* action in our transportation example, which has three parameters. The system then incrementally considers removing each parameter from the tuple. At each step the tuple is tested to determine whether it

Algorithm 3 MINIMAL STATIC PARAMETER TUPLE FINDER (MSPT): Given an action a, and a Boolean function: `tuples_test`, find the minimal static parameter tuple that satisfies the function.

```
function MSPT($a$, tuples_test)
    $minSPT \leftarrow$ parameters($a$)
    for all $p \in minSPT$ do
        $minSPT' \leftarrow (minSPT \backslash \{p\})$
        if tuples_test([$minSPT'$]) then
            $minSPT \leftarrow minSPT'$
        end if
    end for
    return $minSPT$
end function
```

is still sufficient. For the *move* action the approach considers the tuples in order: For example, considering: $\{\mathbf{2}, \mathbf{3}\}$ and $\{\mathbf{1}, \mathbf{3}\}$, which both fail, before trying $\{\mathbf{1}, \mathbf{2}\}$, which satisfies the test. The process is then repeated, until no parameters can be removed. No further parameters can be removed and $\{\mathbf{1}, \mathbf{2}\}$ is returned.

In *LOP* that test was done using the principle of *preserving optimality*. We generalise this as a `tuples_test` Boolean function that is passed as an argument. And in our approach we test whether the tuple is sufficient to explain the positive and negative examples. This is based on the following observations:

- Static relationships are either positive or negative for all examples in a problem,
- For an action to be applicable, all associated static propositions must hold and so each positive example indicates that the associated static propositions are $True$.
- The negative examples indicate that at least one of the relevant static propositions does not hold.

It is sufficient to set all values to $False$, unless they are required by a positive example. After allocating a value to each static proposition we can determine whether the allocation of values to tuples is consistent with the examples. If the allocation is consistent then the tuple is sufficient. Refer to (Lindsay 2021) for a full presentation of the approach.

Technical Considerations

In developing this framework around *LOCM*, there have been several issues that have arisen. In this section some of these are discussed.

Single Sided State Parameters

When making the FSM for an object, the state machine can have input (or output) states, which have no inputs (outputs). In certain domains these state machines are natural. More over, any additional relationships that might lead to restrictions on the actions applicable transitioning out of (or into) the state can be handled by static facts. However, these structures cause a problem when they occur through a limitation in the training data. For example, in generating an FSM for

t_1	t_2	**Occ**	**hit**
drop.2	move.2	1	false
move.1	drop.2	1	false
move.1	pick.2	1	false
move.2	move.1	1	true
pick.2	move.2	1	false
drop.1	drop.1	6	false
drop.3	drop.3	6	false
pick.1	pick.1	9	false
pick.3	pick.3	14	false
pick.2	drop.2	25	true
drop.2	pick.2	34	true
drop.1	pick.1	37	true
drop.2	move.1	290	true
move.2	pick.2	312	true
pick.2	pick.2	402	true
drop.2	drop.2	419	true
drop.3	pick.3	481	true
move.2	drop.2	532	true
pick.2	move.1	550	true
move.1	move.2	570	true
pick.1	drop.1	906	true
pick.3	drop.3	908	true

Table 2: Occurrence and validity, for each observed transition pair in noisy plans. The pair is valid (a true hit) if it is in the target model. Zero parameter results omitted for space.

a package in a transportation problem it is common, particularly when using plans as input data, that the FSM will have three states: $at.1$, $in.1$, $at.1$. This is because in many logistics plans the package will only be loaded into a truck and then unloaded at its goal. Notice that it is not a constraint of the problem and so there are no negative examples to learn statics from. This means that it is impossible to capture constraints (e.g., the fact that a package is located at a specific location) about the initial (final) states in these cases.

As a solution we extended the machinery to handle single sided state parameters. These operate only on input and output transitions and act as parameters that can be used to constrain the initial and final states.

Target Model Limitations

The *LOCM* approaches make certain assumptions about the structure of the planning model to be learnt. In *LOCM* these assumptions include that each object belongs to a single FSM and that transitions only appear once in each FSM. As a consequence, some of the potential transition matrices are implicitly extended with additional transition pairs. Our system searches through the distinct transition matrices, as this allows it to be readily extended to target *LOCM2* domains. The system does record the extended transition matrices in order that it does not repeat computation. However, as is demonstrated in the evaluation, this can be insufficient in domains with many transitions.

Heuristic Guidance

The space of possible model configurations generated in Algorithm 2 will typically be large. It is therefore important to use heuristic guidance to direct search towards more likely models. One instance of this suggested in (Gregory, Lindsay, and Porteous 2017) is to use the number of occurrences of each transition pairs, with the assumption that erroneous structures will be observed less frequently than valid structures. Figure 2 illustrates that using occurrence to order the transition pairs might be an instructive heuristic in the gripper domain (see Section 'Evaluation' for the experimental setup). In particular, when ordered by occurrence, all 8 of the false positives occur in the lowest 10 (including the zero machine) transitions pairs.

In a similar manner, candidate state parameters can be ordered so that those with few counter examples are ordered first.

Evaluation

In this section we make a preliminary evaluation of the presented approach to domain model acquisition. We have implemented the framework presented above within a framework that can simulate user input. We have used two domains in this evaluation: gripper and logistics. For each domain, we provided 5 example scenarios: their reachable actions, initial and final states (given as an action that will be applicable first and last respectively).

User Simulation The framework also takes the target domain and problem models, which it uses to simulate user input. At the beginning of each episode the framework selects a problem symbol randomly. If the system generates a plan then the framework simulates the plan in the target model and indicate whether the simulation succeeded and whether the goal was achieved (note that no-goal simulations are very common side-effects of missing state parameters, which can lead to empty plans). If the plan fails, the step that the plan becomes invalid is indicated. Unless the plan achieves the goal, the framework then generates a plan to solve the problem. This process is parameterised to allow us to examine how the plan quality and communication noise impact the performance of the system. The first parameter, γ_0, is used to control the plan quality. During forward chaining search, γ_0 is used to determine whether to follow heuristic guidance, or whether to choose a random action. The second parameter, γ_1, is used to add noise to the generated plan. In particular, give the plan: $op_0(args_0^0, ..,); ...op_n(args_0^n, ..,), \gamma_1$ is used on each symbol to determine whether the symbol remains the same or is modified. In this work we use $\gamma_0 = 0.3$ and $\gamma_1 = 0.01$.

Feasibility Study

The key aim in the present work is to assess its feasibility and we have analysed the several key properties of the framework. For the purpose of these analyses we provide average (standard deviation) results for a thousand runs.

The Sufficient Number of Observations The approach starts with no example plans, and consequently, no information to populate the transition matrix. The first analysis we make is to count number of noisy plans required to observe all of the transition pairs in the target model. In particular, we randomly select a problem and the framework simulates the user generating a plan (as described above) and this is repeated. After each plan we test whether all of the transition pairs in the target model have been observed and if they have then we stop. At this step our approach at least has the possibility of discovering the target model.

In gripper, an average of 10.75 (9.64) user generated noisy plans were required to observe sufficient transition pairs to create the target model. For the Logistics domain an average of 7.99 (6.40) plans were sufficient.

Discovering the Target TM Given that we have observed sufficient plans, we next examine how much search effort is required to discover the target model. We first generate fifty plans for each of the five problems and test that this is at least sufficient for the target TM. We then use the search approach —ordering by number of occurences— to count the number of nodes expanded before the target node is discovered.

In gripper, On average 12948.07 (STD: 15151.40) TMs were added to queue, and 668.50 (794.87) distinct TMs were expanded. However, these only corresponded to 12.02 (3.27) unique LOCM I state machines. Consequently, the system would only need to explore on average twelve SP completions to discover the target model. Within the possible space of subsets of 33 potential candidate transition pairs, this appears promising.

In Logistics the space is much larger (the target model has 113 transition pairs) and the search through distinct TMs is prohibitively expensive. In particular, the extension of the TM to align with *LOCM* assumptions can add over 40 transition pairs. This means it is a challenge for search to even find a node with a distinct state. In our tests the first new *LOCM* TM was found after 50000 nodes were explored. This finding suggests that we should implement a specialised search to directly support the *LOCM* target representation. It also suggests that it may be necessary to get definite information without a complete model, as this will allow the system to prune more alternative models earlier.

Case Study: Gripper

In this section we use the framework to learn model for the gripper domain. Starting from an empty set of plans and background knowledge we iterate through a series of episodes. At each iteration a problem is selected and the user seeks a plan from the system. The system attempts to find a model that is consistent with its knowledge of the problem (and the other scenarios it has information of) and to generate a plan. The plan is validated and the system's knowledge is updated. In cases of failure, the user provides a solution. For the purposes of this study, we have used 250 episodes in each run and repeated each run 1000 times.

We believe that key measures for assessing the approaches usability are:

i. the number of plans that the user must provide feedback,
ii. the number of plans that the user must provide,
iii. the first episode that the system generates a valid plan and
iv. the last episode that a valid plan is not provided by the planner.

Notice that we assume that the user is always to some extent responsible, then the user must validate all plans that are produced. However, these figures give some idea of the effort and how much trust the system might gain.

The averaged results for the 250 episodes are as follows: the user had to provide feedback for 3.15 (1.49) plans; the user provided 16.40 (20.45) plans; the system first generated a valid plan at episode 8.65 (5.17); and the last failure was in episode 18.60 (22.03). During the first phase, the system is gaining information about the domain and has not observed all of the transition pairs. We would have expected that the number of failed episodes would be similar to the sufficient number of episodes to see the transition pairs in the target model (see 'The Sufficient Number of Observations'). However, we noticed that there was a single outlier (included in the figures) that had around 100 failed iterations. The typical behaviour of the system is that the system produces a few plans for the user to validate and then once it has observed all of the transitions in the target model it will discover a successful model within one or two iterations. More analysis is required to understand the reason for the unexpected result.

Overall these results appear promising for the gripper domain. However, the study has identified weaknesses too. Until a plan is generated, none of the structural features can be confirmed, meaning that each structural element must be considered a candidate for removal. We note that the generated models might be used within a model-lite (Zhuo and Kambhampati 2017) framework, which might allow a plan to be generated and the user to provide earlier validation. Also, it is well known that certain transition pairs only occur rarely in plans, meaning that many episodes can conclude with no change to the situation. It will be interesting to consider the utility of allowing additional specially constructed action sequences to be presented to the user, to provide additional validation.

Conclusions and Future Work

In this work we have considered the problem of domain model acquisition in the context of noisy input plans. We have presented a framework that allows for a series of episodes with the system's knowledge about the domain growing over time. We proposed a realistic scenario, where a supervisor would be responsible for checking the plans and providing plans when the system fails. We make explicit the role of a supervisor as a necessary part of the framework. We have presented our system, which maintains a partial knowledge of the domain and can create a model that is consistent with its current knowledge. In a preliminary evaluation we analysed the feasibility of the approach, and show that while the approach seems feasible in small domains, it is currently not suitable for more complicated domains. In the future will perform a more comprehensive evaluation of the system and extend the system so that it can make use of model-lite approaches to planning. We will also investigate the use of alternative model learners within the framework.

References

Asai, M.; and Fukunaga, A. 2018. Classical planning in deep latent space: Bridging the subsymbolic-symbolic boundary. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Cresswell, S.; and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *Proc. of the 21st Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

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

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *Proc. of the 19th Int. Conf. on Automated Planning and Scheduling*. AAAI Press.

Gregory, P.; and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *Proc. of 25th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 97–105.

Gregory, P.; and Lindsay, A. 2016. Domain Model Acquisition in Domains with Action Costs. In *Proc. of the 26th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Gregory, P.; Lindsay, A.; and Porteous, J. 2017. Domain model acquisition with missing information and noisy data. In *Proc of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

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. 2021. Reuniting the LOCM Family: An Alternative Method for Identifying Static Relationships. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Lindsay, A.; Franco, S.; Reba, R.; and McCluskey, T. L. 2020. Refining Process Descriptions from Execution Data in Hybrid Planning Domain Models. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*.

Lindsay, A.; Read, J.; Ferreira, J. F.; Hayton, T.; Porteous, J.; and Gregory, P. J. 2017. Framer: Planning models from natural language action descriptions. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

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. URL http://arxiv.org/abs/1210.4889.

Ng, J. H. A.; and Petrick, R. P. 2019. Incremental Learning of Planning Actions in Model-Based Reinforcement Learning. In *IJCAI*, 3195–3201.

Porteous, J.; Ferreira, J. F.; Lindsay, A.; and Cavazza, M. 2021. Automated Narrative Planning Model Extension. *Journal of Autonomous Agents and Multi-Agent Systems* .

Segura-Muros, J. Á.; Pérez, R.; and Fernández-Olivares, J. 2018. Learning Numerical Action Models from Noisy and Partially Observable States by means of Inductive Rule Learning Techniques. In *Proceedings of the ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. L. 2007. Planning domain definition using GIPO. *Knowledge Eng. Review* 22(2): 117–134. doi:10.1017/S0269888907001063. URL http://dx.doi.org/10.1017/S0269888907001063.

Sreedharan, S.; Chakraborti, T.; Muise, C.; Khazaeni, Y.; and Kambhampati, S. 2020. –D3WA+–A Case Study of XAIP in a Model Acquisition Task for Dialogue Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, 488–497.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. In *International Conference on Automated Planning and Scheduling*, 336–343.

Walsh, T. J.; and Littman, M. L. 2008. Efficient Learning of Action Schemas and Web-Service Descriptions. In *Proc. of 23rd AAAI Conference on Artificial Intelligence*.

Wickler, G.; Chrpa, L.; and McCluskey, T. L. 2014. KEWI - A Knowledge Engineering Tool for Modelling AI Planning Tasks. In *International Conference on Knowledge Engineering and Ontology Development*, 36–47.

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. ISSN 0269-8889. doi:http://dx.doi.org/10.1017/S0269888907001087.

Zhuo, H. H.; and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

Zhuo, H. H.; and Kambhampati, S. 2017. Model-lite planning: Case-based vs. model-based approaches. *Artificial Intelligence* 246: 1–21. ISSN 0004-3702. doi:https://doi.org/10.1016/j.artint.2017.01.004. URL https://www.sciencedirect.com/science/article/pii/S000437021730005X.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence* 174(18): 1540–1569.