Reuniting the LOCM Family: An Alternative Method for Identifying Static Relationships

Alan Lindsay
Automated Planning Lab,
Heriot-Watt University, Edinburgh, Scotland, UK
alan.lindsay@hw.ac.uk

Abstract

The *LOCM*-family of domain model acquisition approaches examine synthesising planning models from action sequences. The resulting systems have proven effective at accurately uncovering planning model dynamics, static relationships and action costs. Within the family, the *LOP* system provides a general approach for identifying static relationships; however, *LOP* requires optimal plans as input, which can be impractical for (potentially hand-crafted) action sequences. In particular, in domains where the actions are associated with costs, specifying appropriate action sequences may require metric optimisation, or perhaps less intuitive, unit-cost optimisation. The aim of this work has been to address this conflict, by unifying the input requirements of the *LOCM*-family of systems. As a start we have developed an approach for identifying static predicates, which does not rely on unit-cost optimal plans. Instead it operates 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 identifies the missing constraints by comparing the actions allowed by the model of the dynamics with the set of reachable actions. We demonstrate the approach's accuracy on several benchmark planning domains and also show that it can identify the static relationships more accurately than the existing *LOP* approach.

1 Introduction

Planning models play a fundamental role in Automated Planning. 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. There are three main ways in which domain model acquisition systems vary: the nature of the input data they receive, the expressiveness of the target language and the query system by which they acquire the input data. In this work we focus on the *LOCM*-family of domain model acquisition approaches, which synthesise planning models from action sequences (with total plan costs). We observe that the input requirements across the *LOCM*-family can lead to the approaches not being applicable. The most strict of the requirements comes from the identification of static relationships in the *LOP* system (Gregory and Cresswell 2015). In particular, *LOP* relies on optimal plans, which introduces a strong requirement on the input action sequences. Ensuring that (potentially hand-crafted) action sequences are optimal is not always practical. This can be compounded in domains where the actions have action costs, requiring metric optimisation, or the use of the less intuitive unit-cost optimality.

A core idea in the *LOP* approach is to use a Boolean function to prune a hypothesis space. The hypotheses are ordered and *LOP* starts with the most complex model, incrementally simplifying that model. At each step the Boolean function is used as a proxy to determine if the simpler model is sufficient to capture the required structure. The *LOP* approach relies on testing optimal plan lengths, and considers a hypothesis sufficient if it maintains the optimal plan lengths. However, we observe that there are other Boolean functions that can be used to validate hypotheses and not all require optimal plans.

We consider the problem of creating static predicates as a classification problem: a function that determines whether an action is applicable or not. Our approach operates from a set of positive and negative examples, which provide action headers (an action name and its arguments) that are either *conceivable* or not *conceivable* in a planning problem. We define a Boolean function, which uses the examples to validate static relationships and can replace the optimality check used in *LOP*. We present the results of an empirical evaluation, which tests the approach using benchmark planning problems. The results demonstrate that our approach is effective at identifying the underlying static relationships in the target models and that it is more accurate than *LOP*.

The structure of the paper is as follows: first the *LOCM*-family of approaches are presented; then a definition and a

```
(define (domain transport)
  (:types
        location target locatable - object
        vehicle package - locatable
        capacity-number - object
  )
  (:predicates
     (road ?l1 ?l2 - location)
     (at ?x - locatable ?v - location)
     (in ?x - package ?v - vehicle)
     (capacity ?v - vehicle ?s1 - capacity-number)
     (capacity-predecessor ?s1 ?s2 - capacity-number)
  )
  (:functions
     (road-length ?l1 ?l2 - location) - number
     (total-cost) - number
  )
  (:action drive
    :parameters (?v - vehicle ?l1 ?l2 - location
                 ?c - capacity)
    :precondition (and
        (at ?v ?l1)
        (capacity ?v ?c)
        (road ?l1 ?l2)
      )
    :effect (and
        (not (at ?v ?l1))
        (at ?v ?l2)
        (increase (total-cost) (road-length ?l1 ?l2))
      ))
 (:action pick-up
    :parameters (?v - vehicle ?l - location
                 ?p - package ?s1 ?s2 - capacity-number)
    :precondition (and
        (at ?v ?l)
        (at ?p ?l)
        (capacity-predecessor ?s1 ?s2)
        (capacity ?v ?s2)
      )
    :effect (and
        (not (at ?p ?l))
        (in ?p ?v)
        (capacity ?v ?s1)
        (not (capacity ?v ?s2))
        (increase (total-cost) 1)
      ))
  (:action drop
    :parameters (?v - vehicle ?l - location
                 ?p - package ?s1 ?s2 - capacity-number)
    :precondition (and
        (at ?v ?l)
        (in ?p ?v)
        (capacity-predecessor ?s1 ?s2)
        (capacity ?v ?s1)
      )
    :effect (and
        (not (in ?p ?v))
        (at ?p ?l)
        (capacity ?v ?s2)
        (not (capacity ?v ?s1))
        (increase (total-cost) 1)
      )))
```

Figure 1: Transport Domain: An example of a domain with static relationships and action costs. This domain provides a running example through the paper.

method for obtaining the positive and negative examples is described, and we present the approach for generating static relationships from the examples; we then reflect on the input requirements for the *LOCM*-family and evaluate our approach; finally we conclude and suggest future works.

2 Background: The *LOCM* Family

The domain model acquisition system that we introduce fits into the *LOCM*-family of approaches that use the *LOCM* or *LOCM2* system as a pre-processing step. In order to describe these systems, and also our own, we introduce the Transport domain as a running example. This introduction follows (Gregory and Lindsay 2016).

```
PLAN 1: COST 143
drive  truck2 loc3 loc1 cpy3
pickup truck2 loc1 pkg1 cpy2 cpy3
drive  truck2 loc1 loc3 cpy2
pickup truck2 loc3 pkg3 cpy1 cpy2
pickup truck1 loc2 pkg4 cpy2 cpy3
drive  truck2 loc3 loc1 cpy1
drop   truck2 loc1 pkg1 cpy1 cpy2
drive  truck2 loc1 loc3 cpy2
drive  truck2 loc3 loc2 cpy2
pickup truck1 loc2 pkg2 cpy1 cpy2

PLAN 2: COST 151
pickup truck2 loc6 pkg1 cpy3 cpy4
pickup truck2 loc6 pkg3 cpy2 cpy3
drive  truck2 loc6 loc2 cpy2
pickup truck1 loc6 pkg4 cpy1 cpy2
drive  truck1 loc6 loc2 cpy1
```

Figure 2: Two plans from the Transport domain. Collections of plans form the input to many domain model acquisition systems.

2.1 Running Example: The Transport Domain

Figure 1 shows the Transport domain, which is a typical logistics-type planning domain. It has three operators, each with an action cost. There are static predicates used in each of the actions. The *drive* action is constrained to allow traversing between certain pairs of locations. The *pick-up* action increments and the *drop* action decrements a count of the number of packages in the truck. This is encoded using a static predicate that captures the ordering. Note that the domain is slightly modified from the benchmark domain. An extra parameter and precondition have been added to the drive action. This change does not alter the search space, but is required for *LOCM* and *LOCM2* to correctly learn the domain dynamics (Gregory and Cresswell 2015).

2.2 The *LOCM* Algorithms

The *LOCM* family of systems (Cresswell, McCluskey, and West 2009; Cresswell and Gregory 2011; Gregory and Cresswell 2015; Gregory and Lindsay 2016) operate with the assumption that each object is represented by a parameterised finite state machine. They use no other information besides the action sequences, such as those shown in Figure 2 for the Transport domain, i.e. no information about types, predicates, initial or final states. This is possible because of restricting assumptions about the form of the domain model. The key assumptions of the *LOCM*-family of algorithms are: that the behaviour of each object is described by one (or more in *LOCM2*) finite state machines whose arcs are the transitions that change the state of the object. And crucially, each of these transitions can only occur once in an object's FSM.

An example of the output of *LOCM* is shown in Figure 3. The state machines are shown for the Truck and the Package types. The meaning of the Package FSM is that a package can be in two states (in and out of a truck). Within a truck, it has an association with a truck, and when outside of a

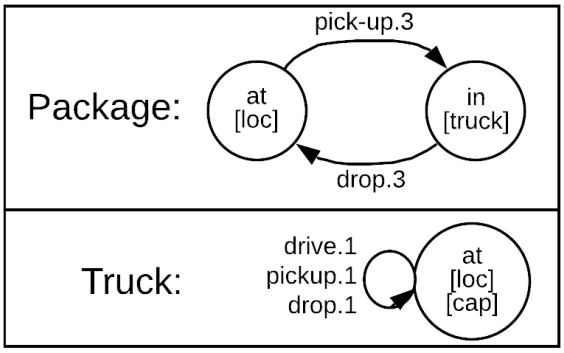

Figure 3: The finite state machines derived by *LOCM* for the truck type in the transport domain for the two interesting object types: package and truck. The truck state machine has a single state, with two state parameters for the location of truck and the capacity of the truck.

truck it has an association with a location (represented by the state parameters in square brackets). The Truck type is represented by a single state FSM, where the only interesting structure is in the state parameters.

The *LOP* system (Gregory and Cresswell 2015) extends the model of the dynamics created by the *LOCM* and *LOCM2* algorithms. It learns static relations by comparing optimal input plans with the optimal plans found using the induced domain model of *LOCM2*. Assuming that *LOCM2* has detected the dynamics of the problem correctly, then if the induced plan is shorter, this provides evidence to support the hypothesis that a static relation has gone undetected.

ASCoL is an alternative approach for learning static relationships, which is also compatible with *LOCM*. *ASCoL* is based on the assumption that static relationships will only exist between same type action parameters. Following *LOP*, we consider a more general definition of static relationships: as relationships over subsets of action parameters.

The final step is to learn action costs using the *NLOCM* system (Gregory and Lindsay 2016). Operating from costed action sequences the system identifies the actions that must incur cost and also identifies the action parameters that lead to alternative costs.

This work aims to address the requirement of optimal plans for the *LOP* system. Ensuring that (potentially hand-crafted) action sequences are optimal is not always practical. This is more complicated in domains where the actions have action costs. For example, consider confirming that the input plans in Figure 2, are optimal. This would require metric optimisation, or the use of unit-cost optimality, which are unlikely to be intuitive for a user.

3 Acquiring Positive and Negative Examples For Static Relationships

A motivation for this work is to open up the *LOCM*-family of approaches to a wider range of applications. The *LOP* approach relied on testing optimal plan lengths as a proxy to

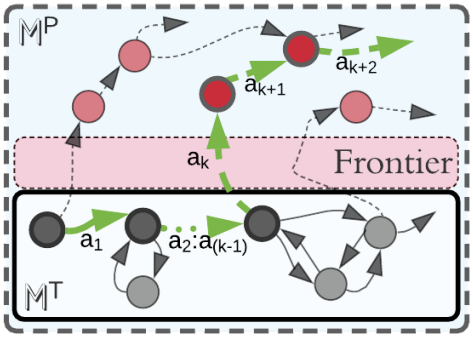

Figure 4: The diagram presents a compartmentalised illustration of the state system of the partial model, $\mathbb{M}^P$. It emphasises that the target model is a subcomponent of the partial model. The solid arrows are transitions in the target model, $\mathbb{M}^T$. The more permissive $\mathbb{M}^P$ can allow additional actions, and where these actions transition from states in $\mathbb{M}^T$, we call them frontier actions. An example sequence, $a_1, \ldots, a_{k+2}$ is shown in green. We consider example sequences consisting of any number of actions from $\mathbb{M}^T$ followed by a single frontier action.

determine if a hypothesis model simplification was still sufficient to capture the required structure. Ensuring that (potentially hand-crafted) action sequences are optimal is not always practical. In this work we consider the problem as creating a classifier that distinguishes between applicable and non-applicable actions. We describe a suitable set of positive and negative examples and suggest alternatives for how these examples can be acquired.

3.1 Positive and Negative Examples

In this work we will assume that we have a partial model, $\mathbb{M}^P$, which we assume is acquired from a domain model acquisition system, such as *LOCM*. In particular, we assume that $\mathbb{M}^P$ accurately captures the dynamics of the target system. It is also typical for a model generated by *LOCM* to capture typing information for the objects and action parameters. Our approach will identify any missing typing information, although missing typing information will impact on performance. We will use $\mathbb{M}^T$ to denote the target system.

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 we start from the existing partial model, we start by constraining the set of possible examples (positive and negative) to actions modelled in the partial model, $\mathbb{A}^P$. We seek positive examples that are actions that are *conceivable* in the target system. This means that there is a situation where the action could be applicable. At this stage it is not necessary to consider whether the action is actually *reachable* from the initial state. The negative examples can be any action that is not *conceivable* in the target system. The problem is then to identify a set of static relationships that are missing from the partial model and separate the positive and negative examples (this is the subject of the following section).

In order to allow concise separation between the examples, the negative examples should not include action headers that are simply not reachable and would otherwise be applicable. For example, in Logistics a truck positioned in $City1$ can move between the airport and location in $City1$. There is nothing inconceivable about the truck moving between the airport and location in $City2$, it is simply not possible because it is not reachable. Including these action headers as negative examples will still lead to a correct model being learned; however, it may be less concise. For example, in the Logistics example, the truck parameter would be included in the *move* action static relationship.

One way to ensure this relationship is to consider a frontier between sequences of actions that are reachable 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, $\mathbb{M}^P$. Such action sequences can be separated into three groups, as illustrated in Figure 4. 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. This is illustrated in in the bottom of Figure 4. 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 (middle of Figure 4). 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.

For example, if we consider Plan 1 in Figure 2 we can see that driving from $loc1$ to $loc3$ and from $loc3$ to $loc2$ are possible and the associated *drive* actions will be in $\mathbb{A}^T$ (e.g., bottom of Figure 4). However, the partial model, $\mathbb{M}^P$, would also allow driving directly from $loc1$ to $loc2$, which is not possible in $\mathbb{M}^T$. Some *drive* actions from $loc1$ to $loc2$ will be applicable from states in $\mathbb{A}^T$, e.g., $(drive\ truck2\ loc1\ loc2\ cpy2)$ in place of the eighth action in the plan. This is therefore part of the frontier (e.g., middle of Figure 4). We notice that there are different ways that such examples could be acquired.

All applicable actions In some systems identifying the set of applicable actions might be possible. For example, there might be input validation on terminal input, or the space may be fully covered in the examples. The applicable actions can be used directly as the positive examples. The negative examples can then be identified by expanding states using the partial model and recording those not in the applicable actions. This approach is presented in more detail below.

Human-in-the-loop Either the user could include examples of inapplicable actions with the input plan sequences (i.e., noting at stages of a plan that certain actions are inapplicable), or the system could present the user with example sequences generated from the partial model and ask the user to note the first inapplicable action.

Of these alternative approaches, we have used the all applicable actions method in this work.

Algorithm 1 FRONTIER FINDER(FF): Set positive examples (E^+) as reachable actions; and find negative examples (E^-) at the frontier of applicability.

```
1: function FF(s, 𝔸^T, 𝔸^P)
2:     E⁻ ← []; E⁺ ← 𝔸^T
3:     discovered ← [s]; queue = []
4:     queue.enqueue(s)
5:     while ! queue.isEmpty() do
6:         s ← queue.dequeue()
7:         for all {a ∈ 𝔸^P | applicable^P(s, a)} do
8:             s' ← apply^P(s, a)
9:             if a ∉ 𝔸^T then
10:                E⁻ ← E⁻ + a
11:            else
12:                if ! s' ∈ discovered then
13:                    queue.enqueue(s')
14:                end if
15:            end if
16:        end for
17:    end while
18:    return E⁺, E⁻
19: end function
```

3.2 The Frontier of Inapplicable Actions

We are interested in identifying the *frontier* actions because they must have missing constraints. Notice that we assume that the dynamics of the system are captured correctly in the partial model. As a *frontier* action is applicable in the partial model then the only reason it is not applicable in the target model is that the action must be missing a constraint. For actions applied from other states in the partial model (e.g., top part of Figure 4) we cannot be sure. It is possible that these actions are just not reachable from the initial state and would have been applicable in other initial states.

Our approach starts from the set of applicable actions, searching through the target model's search space in order to discover the set of frontier actions. The pseudocode for the `Frontier Finder` method is presented in Algorithm 1, which modifies a standard breadth first search. The positive examples (E^+) are declared upfront as the set of applicable actions in the target model (line 2). The initial state of the problem (discoverable using the partial model and the action sequences used to learn it) is added to the queue and the search proceeds as usual. At each state expansion the applicable actions of the partial model ($\mathbb{A}^P$) are each examined (lines 7-16). If the action is not in $\mathbb{A}^T$ then it is added to the frontier set (the set of negative examples, E^-). Otherwise, the resulting state is added to the queue and search proceeds. Notice that only states that are consistent with the target model can be added to the queue. This is because the dynamics of the partial model are assumed to be correct and the context is irrelevant for static predicates. As a consequence, every action added to E^- is a single step, transitioning from a consistent and reachable state. This means that every action in E^- must be missing a constraint in its preconditions (otherwise it would be in $\mathbb{A}^T$). The method outputs the sets of positive and negative examples.

Algorithm 2 SUFFICIENT PARTITION TEST: Given a tuple partition P, and a set of positive (E^+) and negative (E^-) examples, determine if the tuples in P can separate the positive and negative examples.

```
1: function IsSufficientPartition(P, E+, E−)
2:     TVmaps ← getTupleValMaps(P, E+, E−)
3:     return canExplainExmpls(TVmaps, E+, E−)
4: end function
5:
6: function getTupleValMap(P, E+, E−)
7:     TVmaps ← list()
8:     for all tup ∈ P do
9:         tupvalmap ← map()
10:        for all group ∈ groupby(tup, E+ ∪ E−) do
11:            val = (∃e ∈ group e.target == True)
12:            for all e ∈ group do
13:                tupvalmap[e] ← val
14:            end for
15:        end for
16:        TVmaps ← tupmaps + tupvalmap
17:    end for
18:    return tupvalmap
19: end function
20:
21: function canExplainExmpls(TVmaps, E+, E−)
22:    for all ⟨e⟩ ∈ E+ ∪ E− do
23:        okExample ← e ∈ E+
24:        for all tupvalmap ∈ TVmaps do
25:            if ! tupvalmap[e] then
26:                okExample ←! okExample
27:                break
28:            end if
29:        end for
30:        if ! okExample then
31:            return False
32:        end if
33:    end for
34:    return True
35: end function
```

4 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.*

Each static parameter tuple can subsequently be divided into a collection of smaller tuples. In this section we adapt the *LOP* approach in order to use the examples identified in the previous section.

4.1 A Boolean Function Based on Applicable Actions

At the heart of the *LOP* approach is a Boolean function, which is used to prune tuples from the hypothesis space. In place of testing optimal plan length, our approach uses the set of positive and negative examples generated by Algorithm 1. Our Boolean function tests whether a particular group of tuples is sufficient to separate the examples.

Definition 2 (Action Tuple Projection)**.** *An action's tuple projection for a ground action, $a = (opname, o_0, \ldots, o_n)$, for a tuple, $T = (i_0, \ldots, i_m)$, is the tuple, $(o_{i_0}, \ldots, o_{i_m})$, identifying the action's arguments for each of the tuple's indexes.*

Static relationships are either positive or negative for all examples in a problem. This means that only a single Boolean value can be allocated for each distinct action tuple projection. Notice that for an action to be applicable, all associated static propositions must hold. Therefore each of the positive examples can be used to set the appropriate values for each associated action tuple projection. In general the negative examples indicate that at least one of the relevant static propositions does not hold. The allocation of values to the remaining tuples could be allocated by finding a consistent model (with the requirement of the negative examples). However, in terms of information content of the examples it suffices to set all values to $False$, unless they are required by a positive example.

The pseudocode capturing this strategy is presented in Algorithm 2. The main function, `IsSufficientPartition`, takes as argument a set of tuples (this is the partition, P, for reasons that will become clear), and the positive and negative examples. The output of the function is a Boolean that indicates whether the tuples separate the examples.[1]

The first stage (line 2 and lines 6-19) is to create a *tupvalmap* for each of the tuples. Each map assigns a Boolean value to each of the examples. This is achieved by grouping each of the example actions by their tuple projection (line 10), determining the appropriate value for the projection (line 11): $True$ only if there is a positive example in the group of examples, and allocating that value to each of the examples in the group.

The second stage (line 3 and lines 21-35) iterates through each example and ensures that the value for each tuple is appropriate (lines 24-29). This is determined based on the target value of the tuple (line 23). In the case of positive examples, this test ensures that all of the values are $True$. For the negative examples, this test ensures at least one example is $False$. If this test fails then the function return $False$ (lines 30-32). Otherwise, the function continues to test the next example.

4.2 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 al-

[1] Sets of examples for different problems are each tested separately and the partition fails if it fails for any problem.

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.

```
1: function MSPT(a, tuples_test)
2:     minSPT ← parameters(a)
3:     for all p ∈ minSPT do
4:         minSPT' ← (minSPT\{p})
5:         if tuples_test([minSPT']) then
6:             minSPT ← minSPT'
7:         end if
8:     end for
9:     return minSPT
10: end function
```

Algorithm 4 STATIC PARAMETER TUPLE PARTITIONING (PART): Given a partition P, and a Boolean function: `tuples_test`, find a minimal partitioning of P (with respect to `rank`) that satisfies the function.

```
1: function PART(P, tuples_test)
2:     if ! tuples_test(P) then
3:         return ⊥
4:     end if
5:     minP ← P
6:     for all P' ∈ refinement(P) do
7:         minP' ← PART(P')
8:         if rank(minP') < rank(minP) then
9:             minP ← minP'
10:        end if
11:    end for
12:    return minP
13: end function
```

gorithm 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 $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ would be the starting point for the *drive* action in Transport, which has 4 parameters. The system then incrementally considers removing each parameter from the tuple. At each step the tuple is tested to determine whether it is still sufficient. 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. In our approach this function wraps the `IsSufficientPartition` function in Algorithm 2, by passing our examples, along with the largest partition. For the *drive* action the approach considers the tuples in order: $\{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$, $\{\mathbf{0}, \mathbf{2}, \mathbf{3}\}$, $\{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$. Tuple, $\{\mathbf{1}, \mathbf{2}, \mathbf{3}\}$ satisfies the test. The process is then repeated, until no parameters can be removed. 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. No further parameters can be removed and $\{\mathbf{1}, \mathbf{2}\}$ is returned.

4.3 Part 2: Partitioning the Parameter Tuple

The second stage in the *LOP* approach involves identifying a partitioning of the minimal static parameter tuple for each action. The aim is to divide the tuple up, mitigating the combinatorial growth of parameter sets. Our algorithm is presented in Algorithm 4 and generalises Algorithm 3 in (Gregory and Cresswell 2015). Starting with the singleton set of the complete tuple (e.g., the partition: $[\{\mathbf{1}, \mathbf{2}\}]$, from the example above), the approach aims to recursively divide each element of the set into smaller parts. The recursive algorithm first tests whether the partition is sufficient. Again we replace the approach in *LOP* with a general `tuples_test` Boolean function and our approach uses the `IsSufficientPartition` function in Algorithm 2. The algorithm then recursively explores a collection of refinements. The `refinement` function generates each of the possible ways that one of the partition's tuples can get split into two parts. In the Transport example, the function would return one possible refinement: $[\{\mathbf{1}\},\{\mathbf{2}\}]$. The algorithm attempts to recursively refine each of these refinements. The refined partition is then tested against the best so far and the winner retained. The criteria used to determine the best partition is based on the partitions rank: the size of the largest tuple in the partition, minus the total number of partitions (Gregory and Cresswell 2015).

4.4 Part 3: Identifying Universal Statics

The final stage in the *LOP* approach involves determining whether the static relationships are universal or not. A static relationships is universal if it is shared between every problem instance. This part of the system can be generalised from (Gregory and Cresswell 2015) in a similar way to the approaches above, as it requires a single test to determine whether using the combined set of all observed statics in every problem is consistent. In our approach the idea will be to check the consistency of the set of statics using the examples. This part of the system is still under development.

5 Evaluation

In this section we evaluate our approach for identifying static relationships by testing it using several benchmark planning domains. We have implemented the frontier search and the first and second parts of the approach (described in Sections 3 and 4) in the *STAATIC* system, which means that our system can identify and partition the tuple of parameters that are constrained by a static relationship. The system does not currently identify universal static predicates. The starting point for our system is a correct dynamic model (e.g., the output of the LOCM system). Our approach is compared with the results reported for the LOP system, which is the most related approach. We have selected five planning domains to demonstrate the approach and to highlight some of the differences with the LOP approach.

For each domain we used the benchmark domain and generated the set of reachable actions for one problem. Our system operates from the no statics domain model, a set of no statics problem models and the set of reachable actions for each of these problems. The system generates the frontier for a given maximum state count and records the positive and negative examples for each problem (Section 3). These examples are then used for each action to first identify a

Domain	Benchmark Domain			10x1		100x1		*LOP*	
	$\|Ops\|$	MA	SR	SR	Err	SR	Err	SR	Err
Blocksworld	4	2	0	0	**0**	0	**0**	0	**0**
Driverlog	6	4	2	1	1	2	**0**	2	1
Freecell	10	7	10	10	1	10	**0**	- (4)	-(6)
Miconic	4	2	4	3	1	4	**0**	4	2
Zeno-travel	5	6	3	3	**0**	3	**0**	3	**0**

Table 1: The table presents results and domain properties for five benchmark planning domains. We report the number of actions with static relationships (SR) and the number of errors (Err) for two versions of *STAATIC*: 10 states and 1 problem (10x1) and 100 states and 1 problem (100x1), and the reported results for *LOP*. Bold is used to indicate the fewest errors for a domain.

minimal set of parameters that require constrained and then subsequently partition this set into smaller groupings, if appropriate (Section 4). The final step is to use the partitions to add the new static predicates to the domain and problem models.

For comparison we used two versions of our approach: 10 states and 1 problem (10x1) and 100 states and 1 problem (100x1). We have added an upper limit to the number of states that are expanded in Algorithm 1. We test the approach with a maximum of 10 and 100 expanded states.

5.1 Discussion

Table 1 presents the results of running our system on each of the domains. The two versions of *STAATIC* are compared to the reported results for *LOP* (Gregory and Cresswell 2015). For each approach we report the number of actions with static relationships (SR) and the number of errors (Err). The table also presents number of actions ($|Ops|$), max action arity (MA) and number of actions with static relationships (SR) for each target domain. The table shows that the approach is effective at uncovering the correct parameter tuples. In most of the domains Part 2 finds no improvement (no partitioning of the static parameter tuple), which is correct. The Freecell domain allows partitioning and we comment on this aspect below. We will now consider the results for each domain.

Driverlog: The driverlog domain uses two binary static predicates to encode the driving map and the walking map. The system correctly identifies each of these static predicates with 100 expanded states. For the *walk* action it correctly identified the static parameter tuple: $\{\mathbf{1}, \mathbf{2}\}$ (corresponding to the source and destination locations) and concluded that it cannot be partitioned. The 10 expanded states did not include any states with a driver in a truck and therefore the approach failed to provide any negative examples for the *drive-truck* action. *LOP* was able to identify the statics correctly. However, *LOP* detected the path map static as a universal static relationship. The direct use of positive and negative examples by our approach should help in identify that this is not a universal relationship (once extended to examine universal predicates).

Blocksworld: The blocksworld domain has no static predicates. Both *STAATIC* and *LOP* correctly identified empty tuples for each of the actions.

Miconic: The miconic domain uses binary static predicates in each of its actions. The system correctly identifies each these static relationships with 100 state expansions. The statics used in the *board* and *depart* actions declare the initial and final locations (respectively) for each lift user. The *up* and *down* actions use binary relationships to constrain movement between floors. Whereas *LOP* was not able to identify the statics for the *up* and *down* actions (because they do not impact on optimal plan length), our approach correctly identifies these statics.

Zeno Travel: In Zeno-travel the system correctly identifies the three actions that require static propositions: *zoom*, *fly* and *refuel*. *Fly* and *refuel* require binary relations to encode incremental steps: either incrementing (*refuel*) or decrementing (*fly*). The *zoom* operator uses two fuel levels and uses three parameters $\{\mathbf{3}, \mathbf{4}, \mathbf{5}\}$ to detail the step using two binary relationships. *LOP* identified the importance of the top and bottom levels and made a binary predicate (missing out the middle parameter), which models an *n-plus-two* relation instead. Our approach correctly identifies the triple. This is because our approach is based directly from examples, whereas *LOP* uses a proxy (optimality test).

Freecell: In Freecell all of the actions have parameters involved in static relationships. Moreover, some of the domains have very high arity, making this domain more challenging than some of the others. With 100 expanded states, *STAATIC* is able to correctly identify all the static parameter tuples. For example, the system correctly identifies the tuple $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$ for the *moveb* action. With 10 expanded states, *STAATIC* incorrectly identifies the tuple in one action. In the domains that 10 expanded states is insufficient, it is possible that an alternative approach for selecting the expanded states (e.g., plan steps or random) would allow a more comprehensive coverage in fewer states. The *LOP* system was not able to complete the benchmark version of Freecell. Its suboptimal version uncovered some noisy static information.

In part 2 of the approach, *STAATIC* also found partitionings of the static parameter tuples. For example, in the *moveb* action the static parameters tuple, $\{\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{3}\}$, gives rise to 7 alternative refinements: $[\{\mathbf{0}\},\{\mathbf{1}, \mathbf{2}, \mathbf{3}\}],[\{\mathbf{1}\},\{\mathbf{0}, \mathbf{2}, \mathbf{3}\}],\ldots$. Its analysis identifies

that the best partitioning is $[\{\mathbf{0},\mathbf{1}\}\{\mathbf{2},\mathbf{3}\}]$. This splits the parameters into two: a static relationship between the *?card* and *?new-card* parameters, corresponding to the *canstack* predicate and the *?cols* and *?ncols* parameters, corresponding to the *successor* predicate. This corresponds to the factoring in the benchmark domain. In two of the actions (*homefromfreecell* and *sendtohomeb*) it finds tuples of 7 parameters and explores the 63 refinements (see Part 2). In each case it returns the partition: $[\{\mathbf{0},\mathbf{1},\mathbf{2},\mathbf{3},\mathbf{4}\},\{\mathbf{5},\mathbf{6}\}]$.

Each of the discovered partitionings correctly identifies the separable components of parameters. In the benchmark encoding there are further divisions and sharing of static facts between actions. E.g., in the 5-tuple in the *homefromfreecell* action, the parameters are linked by binary relationships: *suit*, *value* and *successor*. It is possible that factorings could be discovered; however, how to ensure that they would correspond to meaningful abstractions is less obvious.

5.2 Summary

Overall the system performs well and the results demonstrate that *STAATIC* accurately identifies static relationships in several domains. It is particularly interesting that our approach is able to improve on the *LOP* approach in several domains. The use of examples, as opposed to differences in the optimal plan lengths, has allowed our system to more accurately uncover the underlying static representation used in the benchmark planning domains. However, there are some efficiency considerations. When run without types in Freecell, our system runs out of memory during preprocessing. Although the system could be implemented more efficiently, it is worth considering the computational issue. Our approach currently requires grounding the planning model with only type information (no additional static predicates). This is problematic in domains with high arity actions with lots of objects. E.g., Freecell has arity seven actions. It is possible that this issue can be mitigated by using dynamic grounding and sampling the actions at each state. We are interested in the future in investigating how many positive and negative examples are required and ultimately, whether an approach without all reachable actions is feasible.

6 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 2016) and numeric constraints (Segura-Muros, Pérez, and Fernández-Olivares 2018). *STAATIC* is situated within the *LOCM*-family. A key focus in these works is to use a minimal amount of input, typically sequences of action headers (possibly associated with a cost). Within the *LOCM* family, *LOP* and *ASCoL* (Jilani et al. 2015) are alternative approaches for deriving static information. *LOP* provides a general approach to identifying missing static predicates. The aim of the *STAATIC* system is similar; however, it operates from different input data. The intention is to provide alternative input requirements for using the approaches in the *LOCM*-family, that can each be more applicable for different scenarios.

The approach used by *LOP* and *STAATIC* of ordering a hypothesis space and using a Boolean function to prune inconsistent hypotheses is related to the framework proposed in (Mehta, Tadepalli, and Fern 2011), which uses an oracle to validate plans. The *LOP* approach is based on comparing optimal plan lengths. Subsequently, the effectiveness of exploring the hypothesis space using a layering strategy was demonstrated (Gregory and Lindsay 2016). Similar ideas have recently been used to learn planning models from a state space graph representation (Bonet and Geffner 2020). Each of the approaches to domain model acquisition comes with trade-offs. The systems that can target rich propositional fragments, e.g., (Wu, Yang, and Jiang 2007) and LAMP (Zhuo et al. 2010), require additional types of input. The *LOCM*-based approaches tackle the problem in several parts, using a heuristic approach to learning the dynamics of the system, but require only action sequences. The approach described in (Bonet and Geffner 2020) is parameterised and constructs a large hypothesis space, which can become impractical for even quite small sized problems and the approach can lead to multiple alternative hypotheses.

7 Conclusion and Future Work

In this paper we present a new approach for identifying static relationships from sequences of action headers. The motivation for this work was to consolidate the input requirements of the *LOCM*-family of systems. As a start we have explored alternative approaches for uncovering static relationships. Until *STAATIC* the requirements for uncovering static relationships required optimal plans for the identification of static relationships in *LOP*, which is not always practical. In this work we have developed an approach for identifying static predicates, which operates from a correct model of the system's dynamics (e.g., the output of *LOCM* I or II) and the set of reachable actions. The approach first generates a set of positive and negative examples, which identify allowed and not allowed actions. The next step is to use the examples to identify the actions that must have additional constraints. The approach therefore substitutes the need of generating unit-cost optimal plans, with the requirement of providing the set of reachable actions. We have tested our approach on five benchmark planning domains and it identified the static predicates in each domain. Moreover it was more accurate than *LOP* in matching the encodings of the static relationships used in the benchmark domains. In future work we will test using dynamic grounding and sampling the actions and investigate relaxing the requirement of all reachable actions.

Acknowledgments

This work was funded and supported by the ORCA Hub (orcahub.org), under EPSRC grant EP/R026173/1.

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*.

Bonet, B., and Geffner, H. 2020. Learning first-order symbolic representations for planning from the structure of the state space. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI)*.

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

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

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 *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Jilani, R.; Crampton, A.; Kitchin, D. E.; and Vallati, M. 2015. ASCoL: A tool for improving automatic planning domain model acquisition. In *Proceedings of the International Conference of the Italian Association for Artificial Intelligence*.

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)*.

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)*.

Mehta, N.; Tadepalli, P.; and Fern, A. 2011. Autonomous learning of action models for planning. *Advances in Neural Information Processing Systems* 24:2465–2473.

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.

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.

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.

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.

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.