An Interactive Method for Inducing Operator Descriptions

T. L. McCluskey, N. E. Richardson and R. M.Simpson
School of Computing and Mathematics
The University of Huddersfield, Huddersfield HD1 3DH, UK
lee,scomner,ron@zeus.hud.ac.uk

Abstract

Specifying operator descriptions for planning domain models, especially using standard pre- and post condition symbolism, is a slow and painstaking process. This is because one is trying to capture what is essentially procedural knowledge in a declarative way in a language whose design is influenced by the construction of planning engines. The problem is acute if non-planning experts are undertaking this task, and/or the operators are complex or hierarchical. In this paper we describe *opmaker*, a method in which the domain expert specifies the declarative structure of the domain (in terms of an object hierarchy, object descriptions etc) and provides training operator sequences. This input is made in the context of a tools environment supporting planner domain acquisition and modelling. *opmaker* then induces a set of parameterised operator descriptions from these examples, removing the need for the user to become involved in complex parameter manipulation within the underlying symbolic, logic-based language. We discuss the empirical evaluation of the implemented induction algorithm with the help of a range of domains, and draw conclusions for future work.

Introduction

Accurately describing a planning domain is a difficult task for the domain experts if they do not have specialist knowledge of AI planning. Realistic planning domain models are hard to encode, debug and maintain, and the development process is laborious. As planners and planning applications become larger, the problems of engineering planning domain models become more acute. Engineering platforms are required that allow a domain expert to enter domain knowledge at a high level of abstraction, and to facilitate the gluing together of planning tools to help in domain modelling (Simpson *et al.* 2001; Tate, Polyak, & Jarvis 1998). In particular, if AI planning is to provide a solution for end-user problems then a *method* of construction of detailed domains is required.

A graphical domain construction and validation tool, GIPO, (Simpson *et al.* 2001) has been recently released. This is an experimental GUI and tools environment for building classical planning domain models, providing help for those involved in knowledge acquisition and the subsequent task of domain modelling. For the former, it provides an interface that abstracts away much of the syntactic details of encoding domains, and embodies validation checks to help the user remove errors early in domain development. For the latter, it integrates a range of planning tools - plan generators, a stepper, an animator, a random task generator, a reachability analysis tool - all to help the user explore the domain encoding, eliminate errors, and determine the kind of planner that may be suitable to use with the domain. A major, acknowledged problem with the initial version of GIPO, however, is that its use by a non-planning-expert during the knowledge acquisition phase is limited. Most difficult for the inexperienced user is to provide the usual parameterised operator descriptions. GIPO alleviates this problem by adopting an object modelling approach, where the user describes how typical object descriptions change through the execution of the operator. This still requires, however, the writer to encode and track parameters in a detailed way within the body of operator schema.

One way of alleviating this knowledge engineering bottleneck is to use induction to acquire operators. An important success factor for such an inductive approach is that the induction tool is not seen as "stand alone", but is seen as an interactive tool in a diverse tools environment (McCluskey & West 2001). There has been comparatively little work done on this in the planning area, particularly in the context of an overall knowledge engineering method. Notable exceptions include the the work of Wang (Wang 1995), Huffman, Pearson and Laird (Huffman, Pearson, & Laird 1992) and Grant (Grant 1996). In this paper we describe an implemented algorithm for the induction of parameterised, hierarchical operator descriptions from example sequences and declarative domain knowledge. Essentially, the user supplies examples of action sequences by describing all the objects that these operations involve. Objects that have a changeable state are called *dynamic*. Where there is a choice of the target state for a dynamic object in an operation, the algorithm requires the user to point and click on that state. The whole process helps the user abstract away from the particular syntax and

consequential errors, and in particular having to encode operator schema using a symbolic language with subtle uses of parameters. For expediency we have built a prototype for inducing operator schema assuming the usual classical assumptions, although there appears no reason why the tool could not be generalised to help knowledge acquisition with more expressive languages.

The Domain Acquisition Process

An overview of the object-centred approach using GIPO

"GIPO" is an acronym for Graphical Interface for Planning with Objects, pronounced GeePO and is available from *http://helios.hud.ac.uk/planform/gipo*. It was first demonstrated at the 6th European Conference on Planning in September 2001 (Simpson *et al.* 2001). The role of GIPO is to facilitate first domain knowledge capture and secondly domain modelling. The latter process we see as one of developing an existing model to make problems expressed in the model more tractable for existing planning technology. Domain acquisition focuses on allowing the user, perhaps a domain specialist rather than a specialist in planning technology, to adequately capture domain structure. The approach used in domain acquisition encapsulated by GIPO requires that the user specifies objects, the sorts that the objects belong to, and predicates that relate these objects, in the style of OCL (for an in-depth discussion of this object-centred language and method the reader can consult the literature e.g. (Liu & McCluskey 2000; McCluskey & Porteous 1997)). Additionally, and as important, the user describes for each sort a set of "typical situations" that an object of that sort may inhabit as a result of the planning process. We refer to the definitions of these "typical situations" as "substate classes". If one thinks of transitions of an object in the domain being modelled as arcs in a state-machine, then each substate class corresponds to a node in the state-machine. In a simple case, a sort may have only one such substate class (node). For example, if it is enough to record only the position of a car in a domain model then all possible situations of the car may be recorded as simply "at(Car,Place)", where *Car* and *Place* range through all car and place objects respectively. On the other hand, in a hierarchical domain, an object such as a car may have relations and attributes inherited from different levels, where each level is modelled as a state-machine involving different substate classes. Also, the user is expected to enter more details of the structure of the domain by describing any known invariants, including those instances of domain predicates which are always true.

The GIPO Environment

To allow the modeller to capture this information in GIPO we provide an integrated series of editors to construct the various elements of the domain specification. In the editors we allow the specification to be built up

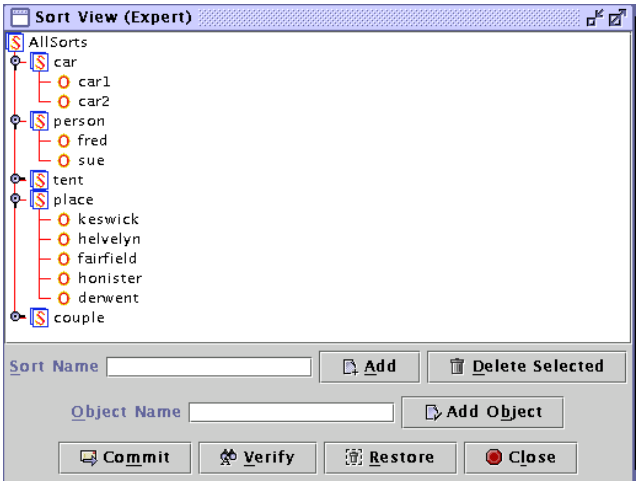

Figure 1: The Sort Hierarchy Editor

using standard GUI graphical structures in such a way that the user is never required to deal at the level of syntax with anything more complex than the rules for forming identifiers. In GIPO the use of graphical dialog construction techniques allows the user to focus on the conceptual details of the domain model rather than on the formal description language. In addition to the editing tools GIPO also provides tools to assist in the validation of a domain specification. As each element of the specification is created it can be checked and cross-checked against other elements of the specification. We provide dynamic tools to enable users to check that the domain definition does support the derivation of plans. To this end we provide integrated planning engines along with an animator to view the planners output. A great benefit of the object centred view is that it allows us to animate the planners output by showing the results as graphs representing state changes in the objects affected by the application of operators. To deal with cases where the planning engine does not produce the expected results we provide a *stepper* to allow the manual construction of a plan to allow the user to check that the defined operators will apply in the situations expected of them.

GIPO is "open" in the sense that third party planners which take as their input PDDL with *strips*, *typing* and *conditional-effects* can be integrated into the system and their output viewed in the animator. We have demonstrated this by integrating Hoffmann's **FF** (Hoffmann 2000) with GIPO without changing any of the FF code. To enable us to integrate third party planners we have provided an export facility to save defined planning domains in PDDL. We are also developing a PDDL import facility (Simpson *et al.* 2000). We see the import and export facilities of GIPO as important in that they allow us to integrate third party tools for planning or domain analysis into GIPO without requiring that these tools be developed using our internal representa-

tion language.

We illustrate the use of GIPO by describing elements of the *sort editor*, the *state editor* and the *operator editor*. The *sort editor* Figure 1, is a simple tree editor to allow the user to describe the sort structure in the domain with the branches of the tree representing sorts and the leafs representing objects.

The editor forces the user to categorise the objects in the domain and allows us to provide software to verify that the constructed domain conforms to rules about inheritance and well formed tree structure. In a similar manner we provide editors to allow the user to define the predicates in the domain and the substate classes characterising the potential states of objects during planning. At the point of defining the substate classes of the domain the process becomes more complex as the user must now deal with decisions about when variables used in a state definition co-designate. We try to make the process of defining substates conceptually simple by allowing the user to collect together the already defined predicates into sets then to select variables and have the potentially unifying variables underlined. The user can then right click on the underlined variables and choose from a menu whether or not the chosen variable must refer to the same or a different object from that of the initial target variable. This process is illustrated in Figure 2 where we see a state of a "block" in the blocks world being defined. In the example we define the state of a block at the top of a block tower. In this example it is insufficient to describe the state of the block in terms of the typed predicates *on(Block,Block)* and *clear(Block)* we must require that the second argument to "on" is always bound to a different block from the other arguments.

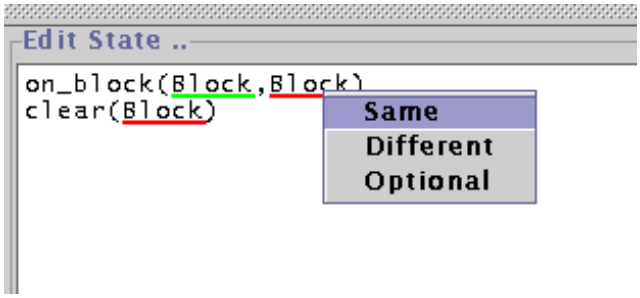

Figure 2: Fragment from the State Definition Editor

In the *operator editor* the problems of co-designation of variables becomes even more problematic. Operators in our object-centred language are conceptualised as sets of parameterised *transitions*, written $(S, O, LHS \Rightarrow RHS)$, where O is an object constant or variable belonging to sort S, and LHS and RHS are substate classes. This means that O moves from situation LHS to situation RHS. Transitions can be necessary, conditional or null. Null transitions are prevail conditions where O must be in LHS and stays in that situation after operator execution.

program *opmaker*(OS: training sequence)
In partial domain model
Out parameterised operator descriptions
1.for each *op* in *OS* do
2. form name and parameter list P;
3. for each dynamic O of sort S in P do
4. get RHS from user input
5. induce necessary substate class LHS
6. form transition $T = (S, O, LHS \Rightarrow RHS)$
7. match free vars in T with those in P
8. end for
9. for all conditional transitions
10. get LHS from user input
11. get RHS from user input
12. form '$\forall\, O \in S, (S, O, LHS \Rightarrow RHS)$'
13. end for
14. end for
procedure match free vars in T with those in P
1. repeat
2. for each parameter X in transition T, $X \neq O$,
3. choose a parameter Y in P to match with
4. X such that $Y \neq O, sort(X) = sort(Y)$,
5. end for
6. until parameter match set is consistent
7. end

Figure 3: Outline Design of the *opmaker* Algorithm

In the GIPO operator editor we provide a graph representation of an operator where the nodes are the LHS and RHS states of the object sorts involved in the operator. Each such node contains an editable state definition. Difficulty arises due to the possible co-designation of variables across the different nodes presented to the user. We deal with this by providing the same sort of underlining and right click mechanism described in the state editor above, but the process remains complex and error prone. It is the avoidance of this complexity that largely motivates the development of the *opmaker* method which hides from the user the inherent complexity of operator definition.

The *opmaker* Method

Inducing Flat Operator Descriptions

We present the *opmaker* method in two stages. In the first stage (this subsection) we will explain a simpler form used in non-hierarchical (or *flat*) domain models - that is models in which properties and relations are all "attached" to primitive sorts and where operator definitions cannot be decomposed. In the next subsection we will detail the hierarchical version.

As an illustrative example for the flat version, and to explain the details of the basic algorithm, we give as an example *the Hiking Domain*, a new planning domain with documentation and description on the GIPO resource page. In the variant of the domain we use, two people (hikers) want to walk together a long clockwise circular route (over several days) around the Lake District of NW England. They do one "leg" each day, as

```
% Sorts
sorts(primitive_sorts,
     [car,person,tent,place,couple]).
% Objects
objects(car,[car1,car2]).
objects(tent,[tent1]).
objects(person,[sue,fred]).
objects(couple,[couple1]).
objects(place, [keswick,helvelyn,fairfield,
        honister,derwent]).
% Predicates
predicates([ up(tent,place), down(tent,place),
   loaded(tent,car,place), in(person,car,place),
   fit(person,place), tired(person,place),
   at(car,place), partners(couple,person,person),
   walked(couple,place), next(place,place)]).
% Object Class Definitions
substate_classes(person,Person,[
    [tired(Person,Place)],
    [fit(Person,Place)],
    [in(Person,Car,Place)] ]).
substate_classes(couple,Couple,[
    [walked(Couple,Place),
         partners(Couple,Person1,Person2)] ]).
substate_classes(tent,Tent,[
    [up(Tent,Place)],
    [down(Tent,Place)],
    [loaded(Tent,Car,Place)] ]).
substate_classes(car,Car,[
    [at(Car,Place)] ]).
% Atomic Invariants
atomic_invariants([
    partners(couple1,sue,fred),
    next(keswick,helvelyn),
    next(helvelyn,fairfield),
    next(fairfield,honister),
    next(honister,derwent)]).
```

Figure 4: Partial Hiking Domain Model

they get tired and have to sleep in their tent to recover for the next leg. Their equipment is heavy, so they have two cars which can be used to carry their tent and themselves to the start/end of a leg, if necessary. Figure 4 is a symbolic representation in OCL of the partial domain model which was specified using GIPO. Identifiers starting with capital letters refer to parameters, whereas identifiers starting with lower case are either sort names or constants. It contains 11 objects of 5 sorts, with 10 predicates.

We assume that we have been constructing our domain model, either textually or using GIPO, and have reached the stage where we have a partial model as laid out in Figure 4. Assume the following training sequence is supplied to *opmaker*. This could be entered by a user via the GIPO GUI, or it could be acquired from some other source. The example sequence below was made up as

- it exemplifies the solution of the simple problem of the hikers doing the first leg of their tour and being rested and ready for the second
- it contains an instance of all the required operator schema, so that the algorithm can induce enough operators to solve the full hiking problem.

```
putdown tent1 fred keswick
load fred tent1 car1 keswick
getin sue keswick car1
drive sue car1 keswick helvelyn
getout sue helvelyn car1
unload sue tent1 car1 helvelyn
putup tent1 sue helvelyn
getin sue helvelyn car1
drive sue car1 helvelyn keswick
getout sue keswick car1
walktogether sue fred couple1 keswick helvelyn
sleepintent sue fred tent1 helvelyn
```

Each item in the sequence consists of the operator's given name followed by a list of objects that are involved in the operation. We follow the algorithm in Figure 3 through for operator "drive", the fourth item in the sequence. This operator has conditional components in that it may carry other objects. Hence in Line 2 $name = drive$, and P is formed from constants $sue, car1, keswick$ and $helvelyn$ as 4 parameters say A, B, C, D representing typical objects of person, car and place (twice) respectively. Within the loop starting on Line 3, for the dynamic object parameter A, the user will be shown the range of substate classes and asked to pick one that the object is in after execution of the operator (there are three classes to choose from for sort *person* as shown in Figure 4). Assume the user picks

$$[in(Person', Car', Place')]$$

The algorithm then forms the LHS of the transition in Line 5 as it tracks the current substate class of *sue*. LHS is set to be

$$[in(Person, Car, Place)]$$

as the previous item in the sequence

```
getin sue keswick car1
```

left object *sue* in this substate class. In this case in Line 6 the following transition is formed:

$$(person, A, [in(Person, Car, Place)] \Rightarrow [in(Person', Car', Place')])$$

As a special case, the user could have chosen to point to "null" rather than any of the substate classes in Line 4. In that case a prevail condition would be formed in the same manner as the LHS of a transition. In Line 7 procedure "match free vars" forces *Person* and $Person'$ to be unified with A, and *Car* and Car' are unified with B, since they are the only parameters of the correct type in the example operator. *Place* and

Place′ are unified with the two *different* parameters of sort 'place' in the example, using a matching heuristic:

Where there is more than one parameter of the same sort in the example, match them to distinct parameters in the transition.

This choice of match has to be consistent with any invariants of the domain model as checked in Line 6. of the procedure. This check may eliminate inconsistent choices of parameters for some predicates. For example, when building up the "walktogether" operator (using the eleventh item in the training sequence) the algorithm is forced to pick independent variables for the last two "partners" parameters $Person1, Person2$ in the substate class:

$[walked(Couple, Place),$
$partners(Couple, Person1, Person2)]$

as in the case where $Person1 = Person2$, no "partners" instance exists.

Returning to the current example, $Place, Place'$ are bound to C, D respectively, and the final transition is:

$$(tent, A, [in(A, B, C)] \Rightarrow [in(A, B, D)])$$

Using a similar procedure the algorithm induces the following transition for a car. In this case, since there is only one class for a car, no user input is required:

$$(car, B, [at(B, C)] \Rightarrow [at(B, D)])$$

The loop exits on Line 8 after two iterations as the last two parameters are of static rather than dynamic sorts. For the conditional part of this operator, the user picks the LHS and RHS substate classes for each sort affected (Lines 9 - 13 in the algorithm). Since the main object parameter is universally quantified in the OCL model for conditional objects, this is selected as a dummy variable. The conditional transitions are:

$(person, X, [in(X, B, C)] \Rightarrow [in(X, B, D)]),$
$(tent, Y, [loaded(Y, B, C)] \Rightarrow [loaded(Y, B, D)])$

The final induced operator after execution of the loop starting in Line 4 is formed from these two necessary and two conditional transitions. After having been translated into PDDL by GIPO, the induced drive operator is as follows:

```
(:action drive
 :parameters ( ?x1 - person ?x2 - car
               ?x3 - place ?x4 - place)
 :precondition (and (in ?x1 ?x2 ?x3)
                    (at ?x2 ?x3))
 :effect (and (in ?x1 ?x2 ?x4)
              (not (in ?x1 ?x2 ?x3))
              (at ?x2 ?x4)
              (not (at ?x2 ?x3))
              (forall ( ?x5 - person )
              (when
                (in ?x5 ?x2 ?x3)
                (and (in ?x5 ?x2 ?x4)
                     (not (in ?x5 ?x2 ?x3))))
              (forall ( ?x6 - tent)
              (when (loaded ?x6 ?x2 ?x3)
                (and (loaded ?x6 ?x2 ?x4)
                (not (loaded ?x6 ?x2 ?x3)))))))
```

Inducing Hierarchical Operators

The construction and maintenance of valid knowledge-based domain models has long been a problem, some might say a bar, to the use of planning techniques in realistic applications. When encoding such domain models there appears to be the need to represent:

- a set of primitive as well as hierarchical or HTN operators. Each HTN operator contains "canned plans" and various types of conditions.
- a collection of structural (static) domain constraints
- attributes and relations arranged in an abstraction hierarchy

We use as a target language OCL_h (McCluskey 2000), a language which extends the object-centred features introduced above. OCL_h compares to O-Plan's TF in its structuring capabilities (McCluskey, Jarvis, & Kitchin 1999), yet is amenable to theoretical analysis (McCluskey & Kitchin 1998). As with the "flat" version of *opmaker* we assume the user uses GIPO to create a partial domain model. This model is more elaborate, however, in that the complete state of each object is determined by relations and attributes attached to the object's primitive sort as well as those attached to its supersorts.

Figure 5 gives as an example part of such a hierarchy in a transport logistics domain. The name of each sort is followed by its parameter, with a list of substate classes underneath. Predicates may be instantiated by choice of objects of the correct sort for the predicate's definition. Only one instantiation of the substate classes at each level is true of an object. For example, an object of sort train will therefore have 3 hierarchical components to its defining state - one component from its primitive sort, one component from its "railv" (rail vehicle) sort, and one from the root sort. The hierarchy allows transitions and operators to suppress details of parts of the hierarchy if they are not relevant.

The overall method of inducing an operator set is recursive:

1. the user provides the system with an example sequence OS.
2. Any operator names in OS not known are induced as primitive operators.
3. The generalised form of OS is given a name and "canned" as a potential hierarchical (HTN) operator.

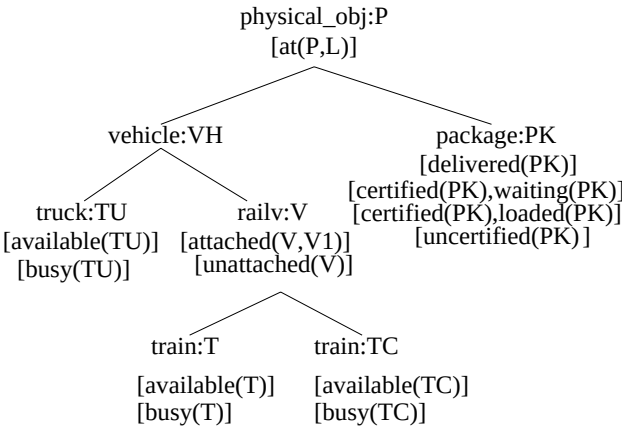

Figure 5: Part of the Translog Sort Hierarchy

Further training sequences may contain the known HTN operator induced from earlier training sequences. In step 2., the induction of *primitive operators* in a hierarchical domain model varies from the algorithm in Figure 3 with respect to two extensions:

Abstract Transitions: (Line 4/5 in Figure 3) Firstly, the user may select one or several of the hierarchical components for the RHS of the transition. Hierarchical components not selected are assumed to persist, and the LHS is constructed from the system drawing only on the current state of the object in those components of interest.

Induction of Statics: Secondly, static constraints to be put on the operator are induced. The atomic invariants in the partial domain model are searched for any containing a subset of the objects in the operator's heading. Any that are found are added to the operator, after the invariant's objects are changed to equivalent parameters to the matching objects in the operator's heading. In the Hiking example used above, the invariant "next(keswick,helvelyn)" would be found for the "drive" operator. The invariant's objects would be generalised to the corresponding parameters in the operator schema's heading, and the generalised invariant added to the operator's prevail condition.

The main addition to the primitive algorithm is an extra process for the **induction of HTN operators.** These operators (called *methods* in OCL_h) are induced after the primitive operators have been induced using an example sequence. Note that, as discussed above, generalisation of a term leads the object constants in that term to be generalised to parameters which range through the primitive sort of that object. The process is as follows:

1. all dynamic objects involved in the items of the training sequence are represented in the method heading
2. the static constraints are induced using the same process as used in primitive operators
3. the transitions of the method are the full generalised transitions of the whole training sequence
4. the dynamic preconditions of the method are the initial states of any dynamic objects involved in the training sequence whose states did not change
5. the decomposition of the method is populated with the generalised training sequence

An example of an auto generated HTN method is as follows. This is induced from a training sequence to move a package to a train station using a truck locally available:

```
pay_fees pk_3
commission truck_2
local_move truck_2 city1_cl1 city1_cl2 local_roads
load_package pk_3 truck_2 city1_cl2
move truck_2 city1_cl2 city3_ts1 road_route_42
unload_package pk_3 truck_2 city3_ts1
```

The generalisation of constants to variables ranging through the primitive sort of the constant is transparent in this example as the parameters are formed from the original constants. Since there is as yet no standard (PDDL) syntax for HTN operators, we use the OCL_h syntax described in reference (McCluskey 2000):

```
% name
method(move_package_local(Pk_3,Truck_2),
% dynamic precondtions
[  ],
% list of necessary transitions
[sc(package,Pk_3,
  [at(Pk_3,City1_cl2),uncertified(Pk_3)] =>
  [at(Pk_3,City3_ts1),waiting(Pk_3),
                      certified(Pk_3)]),
 sc(truck,Truck_2,
  [at(Truck_2,City1_cl1),movable(Truck_2),
   available(Truck_2)] =>
  [at(Truck_2,City3_ts1),movable(Truck_2),
           available(Truck_2)])],
% static constraints
[  route_available(Road_route_42),
  connects(Road_route_42,City1_cl2,City3_ts1),
  connects(Road_route_42,City3_ts1,City1_cl2),
  route_available(Local_roads),
  connects(Local_roads,City1_cl1,City1_cl2)
],
% temporal  constraints
[before(1,2),before(2,3),before(3,4),
 before(4,5),before(5,6)],
%  decomposition
[pay_fees(Pk_3),
commission(Truck_2),
local_move(Truck_2,City1_cl1,
                     City1_cl2,Local_roads),
load_package(Pk_3,Truck_2,City1_cl2),
move(Truck_2,City1_cl2,City3_ts1,Road_route_42),
unload_package(Pk_3,Truck_2,City3_ts1)]
```

Experimental Evaluation

The flat version of *opmaker* is already integrated into GIPO and can be downloaded from GIPO's website for use. An example screen shot is shown in Figure 6.

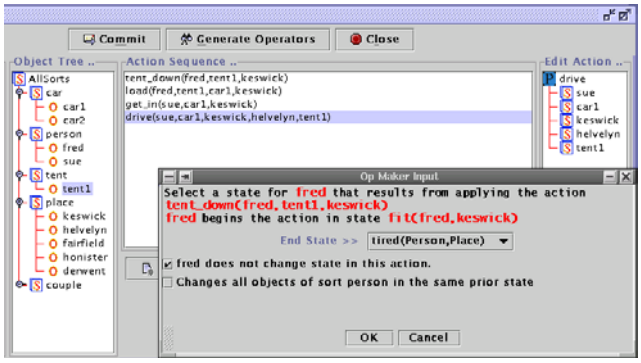

Figure 6: A Screen Shot of *opmaker* within GIPO

The hierarchical version is currently stand-alone, but results and test runs are also available from the GIPO website. Our experimental evaluation of *opmaker* was designed:

– to compare the accuracy of what was induced with hand crafted operator sets

– to compare the robustness of the algorithm when input with different operator sequences

– to test the effectiveness of the tool within an integrated environment

The Hiking Domain was described using the GIPO GUI, resulting in the OCL partial domain model shown above. Then the operators were induced using the *opmaker* implementation, using the operator sequence shown above. A full operator set was generated, passing all local and global validation checks in the GIPO system. One error was eventually found in the requirements - predicate "(next ?x4 ?x5)" was not included in the "walktogether" operator. This error was found after GIPO was used to translate the domain model from its internal object-centred form into the conditional PDDL form shown in the appendix. This was fed into Hoffman's FF (Hoffmann 2000), and the solution output allowed the couple to walk around the Lake District in one day! The "next" predicate had to be added, restricting the hikers to walk between two adjacent areas before becoming tired. With this addition, FF gave a more sensible 54 operator solution sequence in 8 seconds, running on a Sun Ultra 5.

We have performed an initial evaluation of the flat version with several variants of the Hiking Domain, as well as the standard Briefcase World, Rocket World and Tyre World. *opmaker* induces full operator sets for each of these domains, given an example sequence which includes instances of all the operators. For the Briefcase World, the induced and hand-crafted sets were equivalent. With the Rocket World, the same problem occurred as that in Hiking Domain: the only apparent requirements error was the need for there to be a route between two cities before a rocket could move from one to the other. Otherwise, the induced operator set was equivalent to the hand coded set. With the Tyre World, 18 operator schema were induced as expected. Here, the difference between the hand-crafted and induced OCL operators was that in operators "undo nuts", "remove wheel", and "put on wheel" an extra pre-condition was introduced:

$$jack\_in\_use(J, H)$$

This is in fact is a correct pre-condition, as it is implied by the rest of the domain model, but it was left out of the hand-crafted operators presumably because it is implied.

The hierarchical version has been used to generate sets of primitive operators and methods for the four Translog domains used in reference (McCluskey 2000), and the set of operators and methods in the Drum Store Domain (available from the GIPO website). Full sets of primitive operators were induced from partial domain models, and HTN methods encapsulating the training sequence. For these domains we found several of the simple primitive operators equivalent. With the majority of operators we found differences between the induced and handcrafted. These fell into three main classes:

- missing constraints: The induction of constraints using the objects involved in the transition is approximate, and in some cases includes unwanted constraints, and in other cases misses necessary constraints.
- missing pre-conditions: The rule of inducing the LHS of a transition to contain the same hierarchical levels as the user has indicated for the RHS, sometimes misses preconditions.
- undergeneralised parameters. The rule of generalising constants to variables in the object's primitive sorts is sometimes too restrictive. For example, the *load_package* induced operator in the Translog worlds was only applicable to loading packages into trucks if the training example used involved trucks.

As regards varying the operator sequences, we found little variation in the induced operators. Whereas inductive procedures are often very sensitive to ordering of training examples, the "strong model" within which the generalisation is taking place makes the problem not acute in this case. Although the lack of variation and the accuracy of the induced operators is encouraging, experiments with the more complex domains uncovered problems. The hand-crafted sets, the induced sets and the examples they were induced from are available from the resource page of GIPO.

Directions for Future Work

opmaker is intended to *induce* operator schema from example sequences with the minimum of user interac-

tion. Although our initial tests suggest that this tool will be very useful in generating operator sets, we expect it to sometimes make mistakes in its initial generalisation procedure. From our tests, the main errors that occurred were ones of generating over-general or underspecific constraints on transitions.

A simple example from the Hiking world highlights this problem. The missing constraint "next(keswick,helvelyn)" in the walktogether operator is in fact correctly induced when the hierarchical form of opmaker is used, as it includes the "induction of statics" procedure. Also, the "route" predicate would be added to the "move" operator in the rocket world example mentioned above. Unfortunately, while solving the missing constraint problem with the "walktogether" and "move" operators, this procedure over-specialises some operators. For example, the requirement on the "drive" operator in the Hiking Domain is that we can drive between *any* two locations. The instance of drive in the example sequence above, using the "induction of statics" procedure, would add a "next" constraint on the source and destination of the car - leading to an over-specialisation.

There appears two ways to overcome these problems: One way is to assume the induced operators are approximations, and let the user manipulate them using the GIPO tool. Most of the hard work of assembling the operator would be done, but the user would still be expected to fully understand the representation language's semantics. Another way is to use *theory revision* techniques driven by further examples of the operator's application. Operators can be incrementally generalised to cover several examples of their use in hierarchical domains. Theory revision may be used to overcome other problems such as when incorrect parameter choices are made in *opmaker*'s "match parameters" procedure. Although this did not occur in our tests, for more elaborate operators the choice of parameter matching is not always deterministic.

Finally, we have not as yet evaluated *opmaker* in the face of "noisy data", where the user makes a mistake in the operator sequence. This could manifest itself in the user leaving out an object which is in fact affected by an operator, or leaving out an operator application from the example sequence. Again, the revision of operators given more example sequences may be effective to solve the problem of noisy data.

Related work

There has been comparatively little work done in the areas of operator induction and theory revision of AI planning domains, particularly where that work has been within an existing knowledge engineering environment.

Wang (Wang 1995) describes a system, OBSERVER, of learning operators automatically and incrementally by observation of expert solution traces and practice. Inputs to this system are (a) the domain description language (object types and predicates), (b) experts' problem solving traces (i.e. action sequences) where each action consists of the operator name, the pre-state and the post-state (c) practice problems for (initial state and goal descriptions) to allow for learning-by-doing operator refinement. Given these inputs the system automatically acquires the preconditions and effects of the operators.

Another relevant work is Grant's thesis (Grant 1996). He shows how operators may be induced from the description language. In an object based description language objects are represented by the different states in which they can exist. For example in a Hiking Domain the object tent can be "up" or "down" or "loaded" in the car. To get from one state to another the object has to go through a transition and the transition is initiated by an operator. Grant discusses how transitions can be put together into a state transition network, for example:

$up(tent, place) \rightarrow$
$down(tent, place) \rightarrow$
$loaded(tent, car, place)$

Grant shows that by considering the transition from one complete state description to another a suitable operator may be deduced but where there are choices to be made for the object state then a goal can be reached ultimately from the most general state. In the tent example the most general state would be when the tent is down since, from that state it can be either loaded or put up.

Both these works have obvious similarity with our own, the main point of contrast being that *opmaker* is situated within an existing acquisition and modelling environment, and so benefits from the partial model created using that environment, and the use of diverse tools to check what it has induced. Further, *opmaker* is aimed at inducing operators in hierarchical domain models.

In contrast, Huffman, Pearson and Laird (Huffman, Pearson, & Laird 1992) present an analysis of domain theory problems which is based on the premise that "Almost every domain theory is actually an approximation. This is due to the frame problem: the preconditions and effects of actions are extremely difficult to describe fully except in limited domains." They identify different types of domain theory imperfections when it comes to dealing with operators as: overgeneral preconditions; overspecific preconditions; incomplete postconditions; extraneous postconditions; and missing operators. These ideas are influencing our future work as we progress with the theory revision stage. Having found a means of inducing operators successfully, though not necessarily accurately, we hope to use theory revision to refine the new operators induced in our domain models.

Conclusion

In this paper we have described an implemented algorithm *opmaker* for extracting operator schema from

examples containing the list of objects affected by the operator. We argue that such a tool greatly alleviates the task of operator encoding, and fits well into an engineering environment for planning domain acquisition and modelling. Such a tool appears necessary if non-experts are to input domains for use with planning engines. *opmaker* relies on the following cues to help induce operators:

- a partial domain model of object sort and behaviour
- the completeness and coherence of the example sequences of operators
- high level user input, when necessary, in the form of one "click" indicating a choice of substate class

The algorithm induces operators, using a high level, partial model of the domain, as well as using the causal structure of the example sequence to track the state of an object.

Although our initial experiments were very encouraging, we recognise that *opmaker* can generate operators that contain bugs and the thrust of our future work will be in further developing the validation procedures and developing a theory revision system which can aid the correction of the faulty theory and the building of robust domain models. Alongside the future development of GIPO to run hierarchical planners we intend to further integrate the *opmaker* method to build suitable hierarchical operators.

References

Grant, T. J. 1996. *Inductive Learning of Knowledge-Based Planning Operators*. Ph.D. Dissertation, de Rijksuniversiteit Limburg te Maastricht.

Hoffmann, J. 2000. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In *Proceedings of the 14th Workshop on Planning and Configuration - New Results in Planning, Scheduling and Design*.

Huffman, S. B.; Pearson, D. J.; and Laird, J. E. 1992. Correcting imperfect domain theories: A knowledge-level analysis. In S. Chipman and A. Meyrowitz, editors, Kluwer Academic Press, 1992.

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield .

McCluskey, T. L., and Kitchin, D. E. 1998. A Tool-Supported Approach to Engineering HTN Planning Models. In *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence* 95:1–65.

McCluskey, T. L., and West, M. M. 2001. The automated refinement of a requirements domain theory. *Journal of Automated Software Enginnering, Special Issue on Inductive Programming* 8(2):195 – 218.

McCluskey, T. L.; Jarvis, P.; and Kitchin, D. E. 1999. OCL_h: a sound and supportive planning domain modelling language. Technical report, Department of Computer Science, The University of Huddersfield.

McCluskey, T. L. 2000. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling Systems (aips-2000)* .

Simpson, R. M.; McCluskey, T. L.; Liu, D.; and Kitchin, D. E. 2000. Knowledge Representation in Planning: A PDDL to OCL_h Translation. In *Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*.

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*.

Tate, A.; Polyak, S. T.; and Jarvis, P. 1998. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh.

Wang, X. 1995. Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition. In *Proceedings of the 12th International Conference on Machine Learning*.

Appendix: Auto-generated PDDL

```
(define (domain hiking)
  (:requirements :strips :equality :typing :conditional-effects)
  (:predicates
              (up ?x1 - tent ?x2 - place)
              (down ?x1 - tent ?x2 - place)
              (loaded ?x1 - tent ?x2 - car ?x3 - place)
              (in ?x1 - person ?x2 - car ?x3 - place)
              (fit ?x1 - person ?x2 - place)
              (tired ?x1 - person ?x2 - place)
              (at ?x1 - car ?x2 - place)
              (partners ?x1 - couple ?x2 - person ?x3 - person)
              (walked ?x1 - couple ?x2 - place)
              (next ?x1 - place ?x2 - place))
  (:action putdown
         :parameters ( ?x1 - tent ?x2 - person ?x3 - place)
         :precondition (and (fit ?x2 ?x3)(up ?x1 ?x3))
         :effect (and (down ?x1 ?x3)(not (up ?x1 ?x3))))
  (:action load
         :parameters ( ?x1 - person ?x2 - tent ?x3 - car ?x4 - place)
         :precondition (and (fit ?x1 ?x4)(at ?x3 ?x4)(down ?x2 ?x4))
         :effect (and (loaded ?x2 ?x3 ?x4)(not (down ?x2 ?x4))))
  (:action getin
         :parameters ( ?x1 - person ?x2 - place ?x3 - car)
         :precondition (and (at ?x3 ?x2)(fit ?x1 ?x2))
         :effect (and (in ?x1 ?x3 ?x2)(not (fit ?x1 ?x2))))
  (:action drive
         :parameters ( ?x1 - person ?x2 - car ?x3 - place ?x4 - place)
         :precondition (and (in ?x1 ?x2 ?x3)(at ?x2 ?x3))
         :effect (and (in ?x1 ?x2 ?x4)(not (in ?x1 ?x2 ?x3))
                 (at ?x2 ?x4)(not (at ?x2 ?x3))
                 (forall ( ?x5 - person )
                   (when (in ?x5 ?x2 ?x3)(and (in ?x5 ?x2 ?x4)(not (in ?x5 ?x2 ?x3)))))
                 (forall ( ?x6 - tent)
                   (when (loaded ?x6 ?x2 ?x3)(and (loaded ?x6 ?x2 ?x4)(not (loaded ?x6 ?x2 ?x3)))))))
  (:action getout
         :parameters ( ?x1 - person ?x2 - place ?x3 - car)
         :precondition (and (at ?x3 ?x2)(in ?x1 ?x3 ?x2))
         :effect (and (fit ?x1 ?x2)(not (in ?x1 ?x3 ?x2))))
  (:action unload
         :parameters ( ?x1 - person ?x2 - tent ?x3 - car ?x4 - place)
         :precondition (and (fit ?x1 ?x4)(at ?x3 ?x4)(loaded ?x2 ?x3 ?x4))
         :effect (and (down ?x2 ?x4)(not (loaded ?x2 ?x3 ?x4))))
  (:action putup
         :parameters ( ?x1 - tent ?x2 - person ?x3 - place)
         :precondition (and (fit ?x2 ?x3)(down ?x1 ?x3))
         :effect (and (up ?x1 ?x3)(not (down ?x1 ?x3))))
  (:action walktogether
         :parameters ( ?x1 - person ?x2 - person ?x3 - couple ?x4 - place ?x5 - place)
         :precondition (and (fit ?x1 ?x4)(fit ?x2 ?x4)(walked ?x3 ?x4)(partners ?x3 ?x1 ?x2))
         :effect (and (tired ?x1 ?x5)(not (fit ?x1 ?x4))
                 (tired ?x2 ?x5)(not (fit ?x2 ?x4))
                 (walked ?x3 ?x5)(not (walked ?x3 ?x4))))
  (:action sleepintent
         :parameters ( ?x1 - person ?x2 - person ?x3 - tent ?x4 - place)
         :precondition (and (up ?x3 ?x4)(tired ?x1 ?x4)(tired ?x2 ?x4))
         :effect (and (fit ?x1 ?x4)(not (tired ?x1 ?x4))
                 (fit ?x2 ?x4)(not (tired ?x2 ?x4)))))
```