Knowledge Formulation for AI Planning

T. L. McCluskey and R. M. Simpson

Department of Computing and Mathematical Sciences, University of Huddersfield, UK.
Queensgate Huddersfield HD1 3DH
Telephone 44 (0) 1484 422288, Fax 44 (0) 1484 421106
t.l.mccluskey@hud.ac.uk, r.m.simpson@hud.ac.uk

Abstract. In this paper we present an overview of the principle components of GIPO, an environment to support knowledge acquisition for AI Planning. GIPO assists in the knowledge formulation of planning domains, and in prototyping planning problems within these domains. GIPO features mixed-initiative components such as generic type composition, an operator induction facility, and various plan animation and validation tools. We outline the basis of the main tools, and show how an engineer might use them to formulate a domain model. Throughout the paper we illustrate the formulation process using the *Hiking Domain*.

1 Introduction

In recent years AI planning technology has improved significantly in the complexity of problems that can be solved, and in the expressiveness of domain representation languages used. Real and potential applications (e.g. in space technology [19, 6], in information gathering [2], in travel plan generation [12], in Grid Computing [6], in e-commerce and e-work [5]) and community events such as the AIPS/ICAPS Planning Competition have moved the field on considerably in the last 10 years. Work in planning algorithms continues to keep a logical separation between planning engine and domain knowledge, but the problems of acquiring, constructing, validating and maintaining this kind knowledge are still considerable. In fact, in the planning area, the problem of knowledge engineering is still a barrier to making planning technology more accessible [16]. Planning domain description languages, of which PDDL [1, 7] is the most used, reflect the interests of plan engine builders in that they emphasise *expressiveness* (what can be represented) rather than *structure* (in what way things are expressed).

There are peculiarities of planning that clearly distinguish engineering planning knowledge from general expert knowledge. The ultimate use of the planning domain model is to be part of a system involved in the 'synthetic' task of plan construction. This distinguishes it from the more traditional diagnostic or classification problems familiar to knowledge based systems. Also, the knowledge elicited in planning is largely knowledge about actions and how objects are effected by actions. This knowledge has to be adequate in content to allow efficient automated reasoning and plan construction.

With some exceptions (eg [3, 4] and other papers cited below), there is little research literature on planning domain knowledge acquisition. Not too many years ago tools for planning domain acquisition and validation amounted to little more than syntax checkers. 'Debugging' a planning application would naturally be linked to bug finding through dynamic testing, reflecting the 'knowledge-sparse' applications used to evaluate planners. The development of the two pioneering knowledge-based planning systems O-Plan and SIPE has led by necessity, to consideration of knowledge acquisition issues and resulted in dedicated tool support. The O-Plan system has for example its 'Common Process Editor' [24] and SIPE has its Act Editor [20]. These visualisation environments arose because of the obvious need in knowledge intensive applications of planning to assist the engineering process. They are quite specific to their respective planners, having been designed to help overcome problems encountered with domain construction in previous applications of these planning systems.

In this paper we describe a knowledge acquisition method supported by an experimental tools environment, GIPO, for engineering and prototyping planning applications. Like the research centred around application-oriented planners mentioned above, we are trying to develop a platform that can assist the acquisition of structurally complex domains; however our aim is to produce a system that can be used with a wide range of planning engines, and is transparent and portable enough to be used for research and experimentation.

GIPO (Graphical Interface for Planning with Objects) is an experimental GUI and tools environment for building planning domain models, providing help for those involved in knowledge acquisition and the domain modelling. 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. It integrates a range of planning tools - plan generators, a plan stepper, a plan animator, a random task generator and a reachability analysis tool. These 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 beta version of GIPO(2) is available from the GIPO web site *http://scom.hud.ac.uk/planform/GIPO*.

The contribution of this paper is that it draws together the main GIPO tools underlying a staged knowledge formulation method for this kind of domain model. The 3 phases of the method are shown in overview in Figure 1. These are:

- firstly, a static model of objects, relations, properties and constraints is derived from an informal statement of requirements. This is then augmented with local dynamic knowledge in the form of object behaviour. This step may involve and be informed by, the re-use of common design patterns of planning domain structures [23].
- secondly, initial representations of the actions available to the planner are derived from the behaviour model and the initial requirements. This process may be assisted by the use of induction techniques to aid the acquisition of detailed operator descriptions [21].

– thirdly, the model is refined and debugged. This involves the integration of plan generation, plan visualisation, and planning domain model validation tools.

These steps and components combine to provide a tool assisted interface for supporting planning domain knowledge acquisition which provides many opportunities to identify and remove both inconsistencies and inaccuracies in the developing domain model. Additionally GIPO provides an opportunity to experiment with the encoding and planning engines available for dynamic testing. GIPO has a public interface to facilitate the integration of third party planning engines processing domains and problems expressed in PDDL.

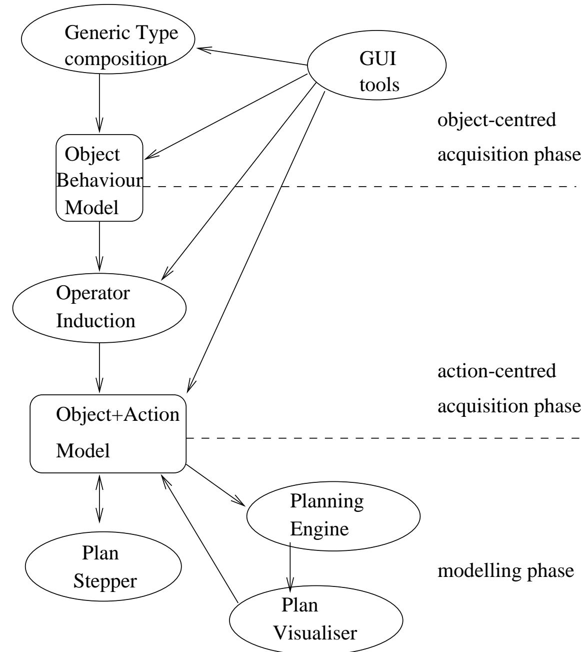

Fig. 1. An Overview of the Knowledge Engineering Process using GIPO

1.1 Underlying Representation Formalism

We briefly review the underlying representation language since this influences and underlies the KE process. The language family used is OCL_h [17, 15]. OCL_h a structured, formal language for the capture of both classical and hierarchical, HTN-like domains. Knowledge is captured by describing changes that the objects in the domain undergo as the result of actions and events. The domain definition is structured around classes of objects, the states that these objects may inhabit, and the possible transitions affected by planning operators. Objects involved in planning have their own local state. Here the state of the world is a vector of variable's values, where each variable holds the value of its own state. As the world 'evolves' the values of the state variables change. This differs from conventional wisdom in AI planning as represented by the standard domain description language PDDL. In *OCL_h the universe of potential states of objects are defined first, before operator definition*. This amounts to an exhaustive specification of what can be said about objects at each level in the class hierarchy. Of course, GIPO insulates the user from detailed syntax, displaying for example object class hierarchies graphically. Whereas the PDDL family is aimed at providing a minimumly adequate common language for communicating the physics of a domain, OCL_h provides a more natural, richer and partially graphical language for domain modelling. More natural because it is structured around the notion of *objects* and *object classes* and is richer because it captures constraints applying to the physics of the domain in addition to the bare physics of actions.

Either manually, or through automated tool support, we acquire for each class of object a set of "typical situations" that an object of that class 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 parameterised node in the state-machine. In a simple case, a class 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.

The developed substate class definitions are then used in the definition of operators in our object-centred language, which are conceptualised as sets of parameterised *transitions*, written

$$(C, O, LHS \Rightarrow RHS)$$

where O is an object constant or variable belonging to class C, and LHS and RHS are substate classes. This means that O moves from a situation unifying with *LHS* to a situation unifying with *RHS*. Transitions can be necessary, conditional or null. Null transitions have an empty *RHS* implying that O must be in *LHS* and stays in that situation after operator execution. A necessary transition requires that there must be an object O that makes the required transition for the operator to be applicable. A conditional transition specifies that any object O in a state that unifies with *LHS* will make the transition to state *RHS*. *OCL* operators accordingly specify pre and post conditions for the objects participating in the application of an operator but in a way that requires the changes to objects to conform to the defined legal states for such objects.

The *OCL* family of languages additionally provides formalisms to capture problem specifications and to capture *static* knowledge, that is knowledge of *objects* within the domain that is not subject to change as a result of the application of domain operators/actions but may be referred to by the specification of such operators/actions.

2 Example Domain

GIPO has been used to build and model many domains since its first version in 2001. Here we will use a simple model of the *Hiking Domain* to exemplify the main features of the KA method. The reader can download more complex GIPO models from our web-site, including the transport logistics domains which contain several hundred static facts and objects in a hierarchy of 30 object classes.

Imagine you want to have a hiking holiday in the Lake District in North West England. You and your partner would like to walk round this area in a long clockwise circular route over several days. You walk in one direction, and do one "leg" each day. But not being very fit, you do not wish to carry your luggage and tent around with you. Instead you use two cars which you have to carry your tent/luggage and to carry you and your partner to the start/end of a leg, if necessary. Driving a car between any two points is allowed, but walking must be done with your partner and must start from the place where you left off. As you will be very tired when you have walked to the end of a leg, you must have your tent up already there and erected so you can sleep the night, before you set off again to do the next leg in the morning. Actions include walking, driving, moving and erecting tents, and sleeping. The requirement for the planner is to work out the logistics and generate plans for each day of the holiday.

The initial pre-formulation phase in the acquisition process is to examine the requirements of the problem, identify the main objects and actions, and make tentative decisions on the kind of planning problem involved. We are developing different flavours of GIPO to accommodate fundamental differences in the outcomes of this phase. For example, it might be considered necessary to model driving, walking, erecting tents, and sleeping as durative processes. If at this stage it was decided that we needed to model time, processes, events and actions explicitly in the domain, then we would use a particular version of GIPO equipped to handle these mechanisms [22]. However, assume we (at least initially) decide that we require to perform these actions in a sequential fashion, and that we are not concerned about how long they take to perform only the order that they must be performed in, then we can represent the actions as instantaneous and we have a "classical planning" problem. We will give obvious names to these actions: 'drive','walk', 'sleep','put up' and 'put down' tents, 'get in' and 'get out' of cars. Object classes will be car, tent, person, couple and location.

3 Initial Domain Formulation: Generic Types and Generic Type Composition

The first phase of our GIPO-supported method is the formulation of a domain ontology augmented with a specification of object behaviour in the form of the typical state changes that objects go through. This can be done manually via GIPO's GUI, but in this section we advocate the use of predefined "design patterns", which we call *Generic Types*. They can be used to help a domain author to develop a domain model using concepts at a higher level of abstraction than is provided by the underlying specification language. Traditional languages for the specification of planning domains, such as PDDL, or even the object-level OCL_h allow the authors of a new domain great freedom in their choice of representation of the domain details. This freedom is we contend for the most part unnecessary and provides an unwanted conceptual barrier to the development of effective domain definitions. In software engineering it is becoming common place to use "design patterns" to help structure the design and coding of applications. Within the AI planning community the notion of a "generic type" has been used to identify common structure across domains with a view to exploiting these similarities to improve plan generation processing speed [8, 13, 14]. In collaboration with Fox and Long [23] we have developed a fusion of these ideas to identify a set of structural elements that are capable of being used to help structure a wide range of potential planning domains. In this way we provide the domain engineer with a set of concepts at a higher level of generality to allow them to more easily identify the object behaviour of their domain. Broadly, a generic type then defines a class of classes of objects all subject to common transformations during plan execution. Within OCL we refer to *classes* which are sets of objects all subject to the same characterisation and transformations. A generic type accordingly ranges over the types or classes of individual domains. The degree of commonality in the characterisation and transformations that these types or classes must share have been described in the literature in terms of parameterised state machines describing the patterns of transformations that the objects undergo. Within GIPO the domain engineer is presented with a range of templates, corresponding to the identified generic types, which she instantiates to create a first cut definition of the object behaviour of the domain.

3.1 Basic Generic Types

Perhaps the most useful of the "generic types" is that of a *mobile*. A *mobile* can initially be thought of as describing the types of objects that move on a map. They can very simply be characterised by the parameterised state machine shown in figure 2

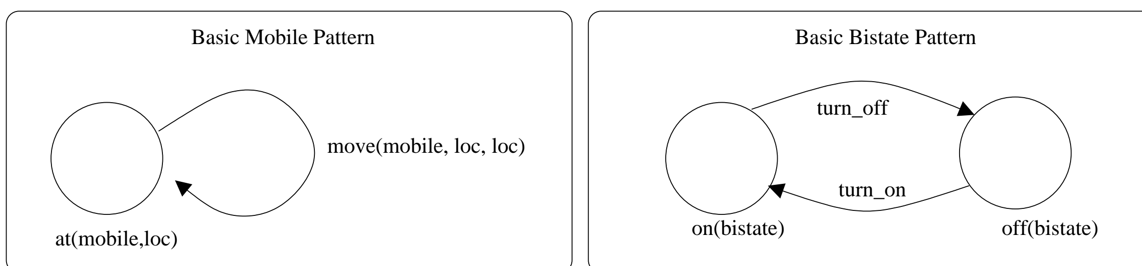

Fig. 2. Basic Mobile and Bistate Generic Types

In this one-state-machine the state is characterised by the property of the mobile object corresponding to its *locatedness* and the transition is identified by the action causing it to move. For this generic type to be applicable to a class in a particular domain there must be a type such that there is an action that changes the truth value of a *n* placed predicate $N >= 2$, typically the predicate *at*, where one argument identifies the class in question, *mobile*, and another a value *loc* which changes as a result of the application of the operator. In other words the value of *loc* differs in the pre- and post-conditions of the action in the reference to the *at* predicate. No other predicate referencing *mobiles* should change truth value in the same action definition in this most basic form of the mobile prototype. The naming of predicates and arguments

may (and will) be different in different instances of the generic type. This is a very weak characterisation of a mobile, in that domains that describe actions that perform transformation on some property of the objects in question might fulfil the above requirements and hence be characterised as a mobile. We have identified a number of variations of the pattern that need to be distinguished from one another. In particular we identify cases where the class mobile must be associated with the class *driver* such that there needs to be an instance of a *driver* associated and co-located with the mobile to enable the mobile's *move* action to take place. The *move* action will also result in the driver changing location in step with the *mobile*. Other flavours of mobiles are associated with *portables* i.e. objects that can be "moved" in associations with mobiles. We also distinguish mobiles that consume or produce resources when they make a transition from one state to another.

A second common and very general *generic type* we call a *bistate*. A *bistate* defines a class *bistate* which is characterised by two one place predicates which we call *off* and *on* both referring to the same *bistate*. There is also a pair of actions that allow the bistate to "flip" from one state to the other. A bistate can be thought of in analogy to a switch than can be turned off and on, see figure 2 Basic Bistate Pattern.

3.2 Generic Types in the Hiking Domain

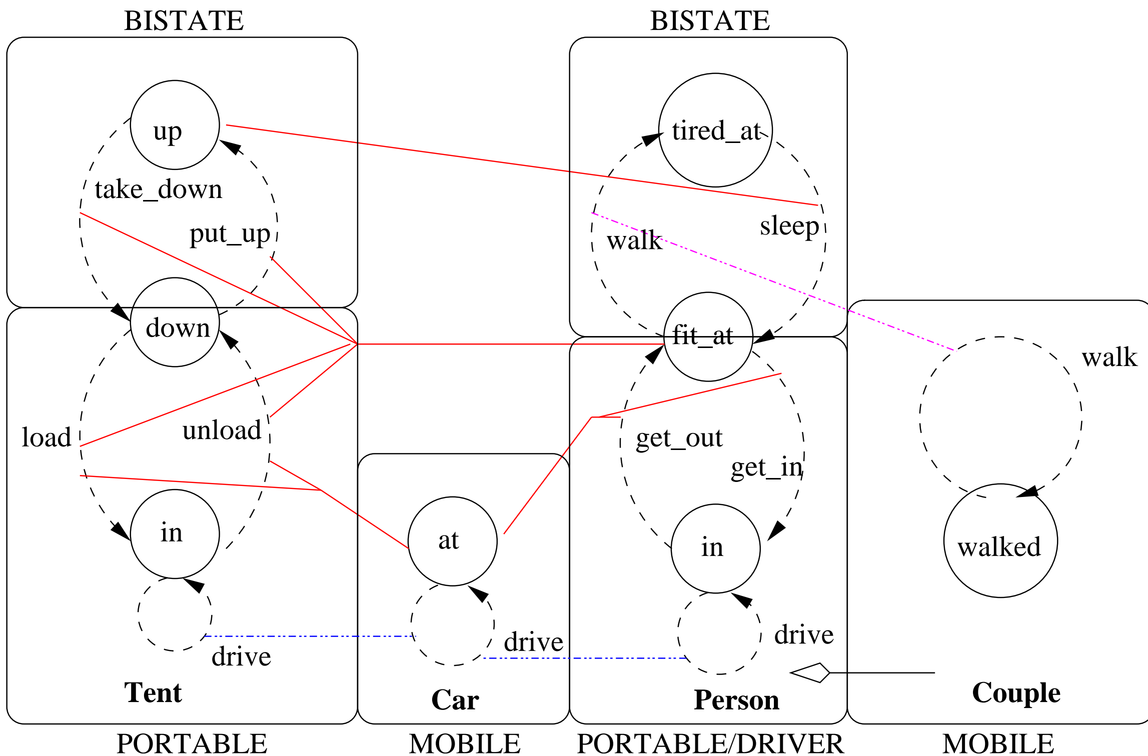

Fig. 3. Generic Types instantiated for the Hiking Domain

In this section we use the example of the Hiking domain to illustrate the way in which generic types can be used as design patterns to support a more abstracted view of the structure within a planning domain during the engineering process. Within the Hiking domain there are at least two candidates which can be identified as mobiles. There are the walkers themselves and there are the cars that are used to carry the tent from one overnight stop location to the next. Though both these candidates can be described as mobiles

neither are adequately characterised by the simple pattern of a mobile illustrated in diagram 2. The walkers which we aggregate into *couples* are mobiles constrained to move from location to location as constrained by a directed graph linking the route locations, which is our *map* of the Lake District. The cars are also *mobiles* but are mobiles that require *drivers* and are used to carry passengers and the tent from overnight stop to overnight stop. The tent itself is a *bistate* which can be either *up* or *down* but is also a *portable* and as such can be *in* the car while being moved from location to location. Individual walkers are also characterised by more than one *generic type* pattern, as a couple they are mobiles, as individuals they are *driver*s or *portable*s but are also *bistates* as they can "flip" between the states of being *fit* and *tired* as they successively walk and sleep. The generic types and the links between them are shown in diagram 3.

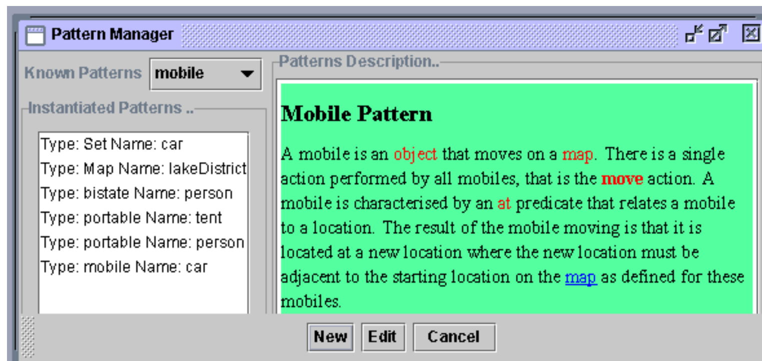

Fig. 4. The Pattern Manager

The GIPO Generic Type Editors To enable the domain developer to use the identified generic types to structure a domain we have developed a series of dialogs which we have integrated into the *GIPO* domain development tool.

The dialogues allow the user to choose the relevant patterns and then tailor them to the problem in hand. In simple cases tailoring is simply a matter of naming the components of the pattern in an appropriate way. In more complex cases the user must add optional components to the pattern again by form filling and in the most complex cases ensure that domains using multiple patterns allow them to interact with each other in the correct way. This may involve unifying states from related patterns or associating common actions that facilitate transitions in multiple patterns. The set of "generic type" dialogues form a domain editor in such a way that the user committing her choices in the editing dialogues will result in the formal domain specification being automatically generated. We illustrate the process with snapshots taken from the "Pattern Manager" in figure 4 which is used to control the addition and editing of patterns known and instantiated within the domain. We also show the main dialog for defining the parameters of the "mobile" pattern in figure 5. At the end of this stage, GIPO will contain an initial formulation of the domain ontology including a characterisations of the states that objects in each class can inhabit.

4 Induction of Object Behaviour

The set of editors integrated into GIPO along with the use of generic type design patterns is adequate to develop fully working domain models. The process, however, requires considerable skill and familiarity with the differing A.I. planning paradigms to be successfully used. To lower the threshold of prior knowledge required to develop planning domain models we have developed an operator induction process, called *opmaker* in its GIPO integrated form, aimed at the knowledge engineer with good domain knowledge but

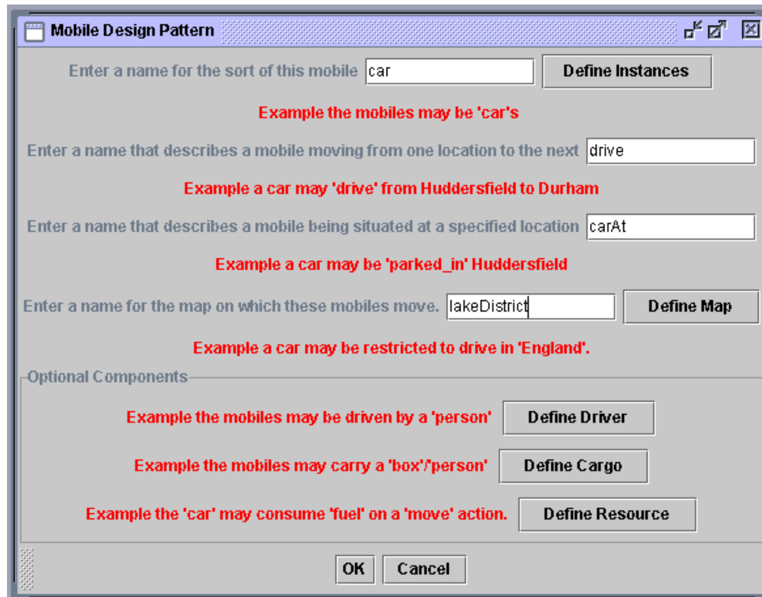

Fig. 5. The Mobile Dialog

weaker general knowledge of A.I. planning. *Opmaker* requires as input an initial structural description of the domain along with training data in the form of a well understood problem drawn from the domain accompanied with an action sequence adequate to solve the training problem. In particular we assume that the modeller has partially constructed her domain model and has reached the stage where there exists at least a partial model with a valid class hierarchy, predicate definitions and substate class definitions. We may have done this using either the base editors of GIPO or by partially describing the problem using generic type design patterns. Some operators may have been developed and will be used if available but are not necessary. In addition to run *opmaker* the user must specify, using the task editor, the training problem. A task specification allocates an initial state to every object in the problem and the desired state of a subset of these objects as the goal state to be achieved. The user now supplies *opmaker* with the training sequence of actions. A snapshot of the *opmaker* interface is shown in figure 6. An action is simply the chosen name for the action followed by the names of all objects that participate in that application of the action. A good sequence of operators would ideally include instances of all operators required in the domain, though this is not required by *opmaker* and the action set can be built up incrementally using different problem instances. For the Hiking domain a good sequence would be one that enabled the couple to complete the first leg of the tour, and be rested and ready to start the next with their cars in the correct place. Such a sequence would include all the operators required. A fragment of the sequence is shown below.

```
putdown tent1 fred keswick
load fred tent1 car1 keswick
getin sue keswick car1
drive sue car1 keswick helvelyn tent1
```

The user is encouraged to think of each action in terms of a sentence describing what happens. For example in the last action we think of this as “Sue drives car1 from Keswick to Helvelyn taking the tent with her.” We sketch the application of the algorithm for inducing the “drive” operator assuming that the earlier operators have been derived and that the specified start situation in the problem placed “Sue”,”Fred”, the tent “tent1”, and both cars in Keswick and that the tent was erected.

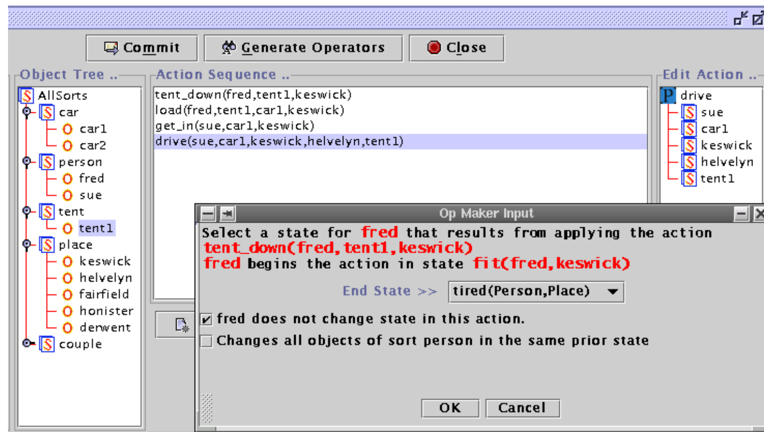

Fig. 6. A Screen Shot of *opmaker*

Prior to reaching the point of the derivation of the *drive* operator we initialise the *worldstate* to that of all objects as described in the initial state of the chosen task and then as each operator is derived we advance the *worldstate* by applying the operator to change the states of the affected objects. In considering the *drive* operator we detect that this is the first application of the operator. Therefore to create the *drive* operator we generate a new operator skeleton and set the parameterised name of the operator to $drive(sue, car1, keswick, helvelyn, tent1)$. We then iterate over the dynamic objects in the parameterised name (these are the objects of classes for which substates have been defined). In this example it is all the objects excluding the two location arguments *keswick* and *helvelyn*. For the first object to be considered *sue* we find her state as stored in the *worldstate*, which for the operator sequence given would be $in(sue, car1, keswick)$. This will form the basis of the left hand side of the transition for this object instance. We now query the user about the states of *sue* that results from applying the *drive* operator. In the dialogue with the user we attempt to gather all the additional information required to correctly identify the transition made by the object, *sue*. We ask whether or not *sue* does change state, if not we add a null transition to the operator. In our example case however *sue* will change state, and the user will select from a supplied list of possible states which will be the resulting state. The selected state will form the basis for the *RHS* element of the transition. As part of the dialog we also ask if any other object of class person in the same prior state would make the same transition. Depending on the answer to this question we treat the transition either as a conditional change or as a required change of the operator and add the transition accordingly. When a transition is added as conditional the object name is removed from the operators parameter list. Only the objects referred to in the prevail and necessary sections are recorded in the parameter list.
Though we now have the main structure for the transition made by *sue* we still have some details to clarify. First we detect that *keswick* is recorded as next to *helvelyn* in a list of static facts provided as part of the initial domain model, we accordingly query the user again to ask if this is a required condition of applying the operator. That is, must the two places be recorded as *next* to one another to enable the application of the *drive* operator. If it is required then we add the fact to the *LHS* of the transition. In the example domain it is not required that *sue* can only drive between adjacent locations, she may drive to any location, hence the predicate will not be added to the *LHS*.

We continue the above process by iterating over the remaining dynamic objects $car1$ and $tent1$ The result is that we have a fully instantiated instance of the *drive* operator. We generalise the operator by replacing the object names with variable names maintaining a one to one relationship between the object names and the variable names. Finally we apply the operator to the *worldstate* to advance the state ready for consideration of the next operator and our derivation of the operator *drive* is complete.

Using this procedure the algorithm induces the following necessary transition for objects of the sorts corresponding to *sue* and $car1$.
$(person, Person0, [in(Person0, Car0, Place0)] \Rightarrow [in(Person0, Car0, Place1)])$
$(car, Car0, [at(Car0, Place0)] \Rightarrow [at(Car0, Place1)])$

For *tent*1 the conditional transition
$(tent, Tent0, [loaded(Tent0, B, Place0)] \Rightarrow [loaded(Tent0, B, Place1)])$
The variables, starting with upper case letters, in the transitions above are all typed either explicitly in the transition or implicitly by their occurrence in the strongly typed predicates defining the transition.

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 - tent)
                     (when (loaded ?x5 ?x2 ?x3)
                        (and (loaded ?x5 ?x2 ?x4)
                        (not (loaded ?x5 ?x2 ?x3)))))))
```

The current version of *opmaker* allows existing operators to be used in the induction process and is capable of refining the operators to add additional features, such as new conditional clauses or static constraints. It is limited however in that the multiple uses must be consistent with one another it will not deal with conflicts. Operator refinement and use in inducing hierarchical operators is the subject of ongoing work [18].

5 Refining and Validating Domain Models

GIPO with its many graphical editors ensures that any domain specification created within the system is at least syntactically correct, in terms of both the representation of the domain in OCL_h and their automatically generated translations into *PDDL*. Automatic validation of the domain is further enabled by the *type* system of OCL_h and the requirement that the static legal states of all dynamic objects are defined in state clauses, the substate class definitions. As these states have been explicitly enumerated operations can be checked to ensure that they do not violate the defined states. Defined tasks for the planning system can also be checked against these state definitions for legality. These static validation checks are capable of uncovering many potential errors within a domain specification.

When the knowledge engineer has assembled a complete, statically validated, specification of the domain there is still the need to test the specification. Dynamic testing can be done by running the domain specification with a trusted planner against test problems. Ultimately this must be done but it is essentially “black box” testing and will provide little insight into problems when the desired output is not created. This problem is exacerbated when we take into account that the current generation of planning engines are not fully robust. To help overcome these limitations within the GIPO environment we provide a plan visualiser to allow the engineer to graphically view the output of successful plans generated by integrated planning engines and more usefully we provide a *plan stepper* to allow the engineer to single step a hand crafted plan when the planning engine either fails to produce any plan or produces an unexpected plan. The attraction of single stepping a hand crafted plan is that it allows a separation in the testing process between validating that domain knowledge is adequate to support the desired plan and knowing that the planning engine is adequate to generate the plan. The animator and stepper both present plans at the level of objects, object states and the changes brought about to the objects by the application of actions.

In the *stepper* shown in figure 7 we see the emerging plan at a stage where the tent needs to be moved to the next overnight stop. The design engineer manually selects the operations to build the plan and instantiates the operation variables. Unfortunately in the case shown there is a mistake in the definition of the *drive* operation in that it fails to guarantee that objects loaded in the car move when the car moves. This can be seen by the engineer double clicking on the tent state following the drive operation and as we see in the pop-up window that the tent is at the wrong location, it should be in the same location as the car, namely “helvelyn”. Also when the engineer next tries to unload the tent at the new destination the system generates an error message to indicate why that cannot be done. In this way the engineer can explore the adequacy of the domain specification.

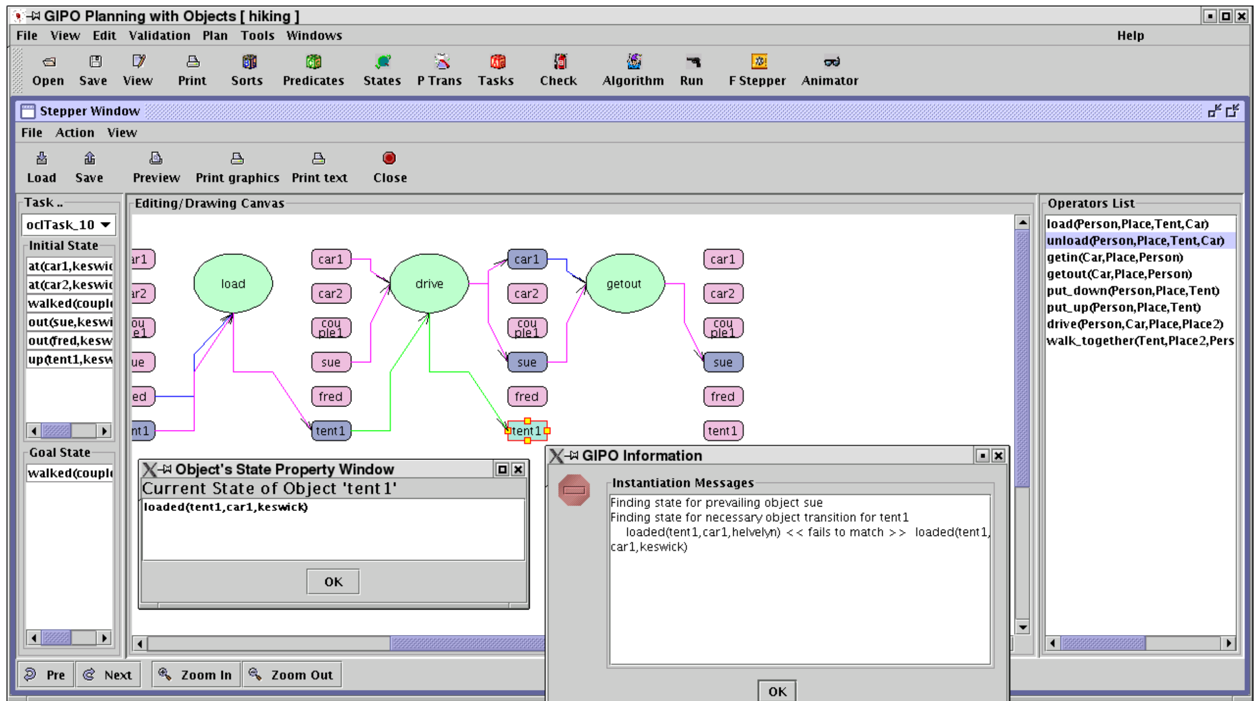

Fig. 7. Testing and Validation Phase

6 Related work

From a knowledge based system point of view, GIPO is a knowledge formulation tool that pre-supposes a common generic conceptual domain model for its range of applications. It assumes that domain knowledge is object-centred, and that actions (and events and processes) in the domain cause well-defined state changes to those objects. GIPO views a planning engine as a generic Problem Solving Method (PSM) - one that can output ordered action sets when input with some declarative goal or procedural abstract task. [1] In this regard GIPO is similar to KBS tools that allow the user to select PSM components depending on the type of reasoning that best suits the derivation of solutions to problems.

As far as the authors are aware, no environments as sophisticated as GIPO have been built to help acquire knowledge in a form that can be used with a range of available planning engines. The environments that have been built for use in applications either tend to be aimed at specific AI planners, or are aimed at capturing more general knowledge. Several notable works in the knowledge acquisition literature such as EXPECT [9] and Protege [11] fall into the latter category. EXPECT is an extremely rich system that uses ontologies and knowledge acquisition scripts to generate and drive dialogues with users to acquire and maintain knowledge bases of a diverse nature. As currently conceived, EXPECT is not designed to interface to AI planning engines and hence enable the generation of plans though it does allow reasoning about plans.

Part of our goal was to try and help bring the very sophisticated planning technology of the planning engines to a potentially wider user base. Our ambition falls some way between that of providing knowledge acquisition interfaces for specific planning engines and the goal of facilitating the gathering of knowledge for a broad spectrum of automated reasoning tasks. We are still exploring what sort of knowledge is needed for such sophisticated tasks and what automated help can be provided to assist end users gather the required knowledge. In the longer run it may be desirable to integrate the type of system we are creating with systems such as EXPECT or Protege as that would facilitate deployment of systems where complex planning is only

[1] This reflects the swing to the development of self-contained hybrid and adaptive planners in the planning literature; approaches to break down planning algorithms into smaller components so that plan engines can be built up rapidly from them for specific domains are yet to be proven (e.g. see [10]).

one part of a broader knowledge based system. Until we have more experience in providing systems just to support the planning element we feel such integration would be too ambitious.

7 Conclusion

Knowledge acquisition and the validation of models of planning domains is notoriously difficult, due to the need to encode knowledge about actions and state change, as well as static and structural knowledge. We see GIPO as a prototype for a range of portable and planner-independent environments that will be used to ease these KA problems. Using high level tools such as the generic type adaptor, operator induction and plan stepping, domain experts with a limited knowledge of AI planning can build up domain models. Using GIPO's interface to state-of-the-art planners via the competition-standard PDDL language, domain experts can prototype planning applications using these planning engines and test them for their suitability to the application.

We have and continue to produce several flavours and versions of GIPO that can be used to formulate knowledge once the nature of the domain requirements are clear. Although implemented and tested, the tools described here are still being evaluated and are the subject of future development. Several hundred down-loads of GIPO have been recorded and we have a small user base around the world. A particularly promising direction is GIPO's use in teaching - it has been used successfully in a one year course in AI with undergraduate students. Particular avenues for future research and development include (a) the identification and build-up of a portable library of generic types for planning applications (b) upgrades of GIPO's automated acquisition tools to deal with domains involving continuous processes, actions and events.

Acknowledgements

This research was supported by the EPSRC grant no. GRM67421/01. Thanks go to our collaborators Ruth Aylett, Maria Fox and Derek Long. We are also grateful to members of our team at Huddersfield, including Weihong Zhao, Donghong Liu, Beth Richardson and Diane Kitchin.

References

1. AIPS-98 Planning Competition Committee. PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
2. G. Barish and C. A. Knoblock. Speculative Execution for Information Gathering Plans. In *The Sixth International Conference on Artificial Intelligence Planning Systems*, 2002.
3. R. Benjamins, L. Barros, and A. Valente. Constructing Planners through Problem-Solving Methods. In B. Gaines and M. Musen, editors, *Proceedings of the Tenth Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW'96)*, 1996.
4. V.R. Benjamins and N. Shadbolt. Preface: Knowledge acquisition for planning. *Int. J. Hum.-Comput. Stud.*, 48:409–416, 1998.
5. S. Biundo, D. Barrajo, and T. L. McCluskey. Planning and Scheduling: A Technology for Improving Flexibility in e-Commerce and Electronic Work. In *Proceedings of e2002, The eBusiness and eWork Annual Conference, Prague The Czech Republic*, 2002.
6. J. Blythe, Eva Deelman, Yolanda Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The Role of Planning in Grid Computing. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS 2003*, 2003.
7. M. Fox and D.Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*, 2001.
8. M. Fox and D. Long. The Automatic Inference of State Invariants in TIM. *JAIR*, 9:367–421, 1997.
9. Y. Gil, J. Blythe, J. Kim, and S. Ramachandran. Acquiring Procedural Knowledge in EXPECT. In *Proceedings of the AAAI 2000 Workshop on Representational Issues for Real-World Planning Systems*, 2000.
10. J. Hertzberg. On Building a Planning Tool Box. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 3–18. IOS Press, 1996.

11. John H. Gennari, Mark A. Musen, Ray W. Fergerson, William E. Grosso, Monica Crubzy, Henrik Eriksson, Natalya Fridman Noy, Samson W. Tu. The evolution of Protege: an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.*, 58, 2003.
12. C. Knoblock, S. Minton, J. L. Ambite, M. Muslea, J. Oh, and M. Frank. Mixed-Initiative, Multi-Source Information Assistants. In *Proceedings of WWW'01*, 2001.
13. D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *Proc. of 5th Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 196–205. AAAI Press, 2000.
14. D. Long and M.Fox. Planning with generic types. Technical report, Invited talk at IJCAI'01 (Morgan-Kaufmann publication), 2001.
15. T. L. McCluskey. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *The Fifth International Conference on Artificial Intelligence Planning Systems*, 2000.
16. T. L. McCluskey, R. Aler, D. Borrajo, P. Haslum, P. Jarvis, I.Refanidis, and U. Scholz. Knowledge Engineering for Planning Roadmap. *http://scom.hud.ac.uk/planet/*, 2003.
17. T. L. McCluskey and D. E. Kitchin. A Tool-Supported Approach to Engineering HTN Planning Models. In *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*, 1998.
18. T. L. McCluskey, N. E. Richardson, and R. M. Simpson. An Interactive Method for Inducing Operator Descriptions. In *The Sixth International Conference on Artificial Intelligence Planning Systems*, 2002.
19. N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence*, 103(1-2):5–48, 1998.
20. K. Myers and D. Wilkins. The Act-Editor User's Guide: A Manual for Version2.2. SRI International, Artificial Intelligence Center, 1997.
21. N E Richardson. *an Operator Induction Tool supporting Knowledge Engineering in Planning*. PhD thesis, School of Computing and Mathematics, University of Huddersfield , forthcoming, 2004.
22. R. M. Simpson and T. L. McCluskey. Plan Authoring with Continuous Effects. In *Proceedings of the 22nd UK Planning and Scheduling Workshop (PLANSIG-2003), Glasgow, Scotland*, 2003.
23. R. M. Simpson, T. L. McCluskey, Maria Fox, and Derek Long. Generic Types as Design Patterns for Planning Domain specifications. In *Proceedings of the AIPS'02 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*, 2002.
24. A. Tate, S. T. Polyak, and P. Jarvis. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh, 1998.