An Integrated Graphical Tool to support Knowledge Engineering in AI Planning

R. M. Simpson, T. L. McCluskey, W. Zhao,†
R. S. Aylett, C. Doniat ‡

Department of Computing Science University of Huddersfield,Queens Gate,Huddersfield, HD11DH, UK †
Centre for Virtual Environments, University of Salford,Salford,M54WT, UK ‡
email: r.m.simpson, t.l.mccluskey, w.zhao @hud.ac.uk
tel no: 044-01484-422288 **fax:** 044-01484-421106 †
email: R.S.Aylett, c.doniat @salford.ac.uk
tel no: 044-161-295 2925 **fax:** 044-161-295 2925 ‡

May 4, 2001

Abstract

Knowledge engineering in AI planning is the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimization of appropriate planning machinery to work on them. Our long term aim is to research and develop rigorous methods for the acquisition, maintenance and validation of planning domain models. In this paper we describe such a method and illustrate it with screenshots taken from an implemented Graphical Interface for Planning with Objects system called GIPO that has been built to support an object centred approach to planning domain modelling. The purpose of the paper is not to report on new research but to support a demonstration of the tool at ECP'01.

Keywords: DEMO, Domain Modelling, Planning Tool Support.

1 Introduction

With the fielding of large planning systems [12, 16] knowledge engineering in AI planning is arguably as important a research topic as the algorithmic aspects of abstract planning engines. It has been defined as the process that deals with the acquisition, validation and maintenance of planning domain models, and the selection and optimisation of appropriate planning machinery to work on them [10]. The problems of acquiring appropriate knowledge within a planning application, and of configuring an appropriate planner to form the target software are very acute when one deals with knowledge intensive applications. This is evidenced by the effort that has been spent on interfaces in large planning systems such as SIPE [2] and O-Plan [20].

There are in fact two different reasons for providing knowledge engineering support for AI planning. The requirements and tools to support them overlap but are not identical.

- One motivation for applying a planning system to a new domain is to test the planning system itself, and this is overwhelmingly the main reason currently within the research community, expressed for example in the AIPS planning competition. Here, the engineer is a specialist in a particular planning system and in the theoretical aspects of AI planning, and requires tools which enable tests on new approaches and algorithms to be carried out quickly and accurately. This type of knowledge engineering emphasises validation and checking at a technical level and within the ontological domain of 'how planning systems work'. The problems tackled are normally some way from real-world complexity - one would not expect to run a real logistics system the way the past AIPS competitions' logistics domains have been specified.
- The second motivation is to tackle an end-user problem for which AI planning is felt to provide a better solution than conventional or other knowledge-based technologies. Here, the engineer might be a domain expert and need not necessarily have a specialist knowledge of AI planning or indeed of any AI technology. The knowledge engineering required corresponds to the classic knowledge acquisition and prototyping cycle confronted by other practitioners in the world of knowledge-based systems and requires a level of abstraction above implementation detail in order to conceptualise the problem domain itself in a consistent and adequate manner.

Our research is aimed at developing a method and tools to support the latter scenario, although the benefits of such an approach can also be enjoyed by planning experts. In this paper we review the rationale behind knowledge engineering for planning, then in section 4 we detail a graphical tool called GIPO which supports our knowledge acquisition method and which we hope to demonstrate at ECP'01.

2 Knowledge Engineering in Artificial Intelligence

The knowledge-level principle, first advanced by Alan Newell [14] holds that knowledge is to be modelled at a conceptual level independent of the detail of implementation and specific computational constructs, and this principle has formed the foundation for many knowledge engineering tools since then. The KADS (now CommonKADS) KBS methodology is also based on this insight [23] . The influential abstractions were those of 'generic tasks' supported by domain-independent and reusable problem-solving methods, of which the first was Clancy's heuristic classification method [1]. This was soon supported by the idea of a domain ontology - a set of terms and relationships which defined declaratively a domain of discourse and within which different problem-solving methods might be assembled to solve particular problems. For example Musen reports that a generic problem-solving method "propose-and-revise" was applied to the design of lift systems for buildings, to antiretroviral therapy for AIDS patients, and plausible conformations for the E.Coli ribosome [13].

Interestingly, this KBS research had little influence on the development of tools in AI planning, perhaps because many practitioners think of an AI planning system as being a generic problem-solving method in much the same sense as "propose and revise" (Valente supplies a rare exception in reference [22] - though note this was published in an HCI forum not a planning one). Yet it seems clear that a particular planning system sits at the implementation and not at the knowledge level, so that tools based on particular planners are also likely to carry implementational peculiarities through to the process of domain modelling. This matters little if the reason for modelling a domain is to test the planner, but greatly if a domain expert is trying to solve a real-world problem.

Thus even the two planners used with some success on real-world problems, SIPE and OPLAN [20], have little support for domain modelling. Indeed this deficiency was recognised in a significant application of SIPE-2, SOCAP [3], a prototype military operations planning system. It was stated that a significant part of the development time went into writing and debugging planning operators for this system [4], a conclusion that anyone who has tried applying a planning system to a real-world problem would substantiate. SIPE-2 now has an associated ACT editor which supports the graphical display, editing and input of ACTs, a formalism for representing procedural knowledge common to both SIPE and the plan execution system to which it is linked, PRS. It is argued that an ACT structure corresponds both to a planning operator or plan fragment in a generative planner, and to an operating procedure in a plan execution system [24], and as this language suggests is some way from a problem-oriented modelling language. The ACT's editor supports the AI expert, but cannot be said to support a domain expert.

OPLAN's modelling language, TF (task Formalism) was supported only by a user manual incorporating heuristics derived from experience in using it. Early work was carried out into taking the KBS approach discussed above, primarily from KADS, as a way of supporting the development of a front-end [21] but the move away from OPLAN towards a new generation planner seems to have halted this work for the present.

Other planning systems have provided internal support for the planning process, usually through graphical representation of the plan net generated, as in the PDB system that was developed to run with UCPOP. This type of support is very much oriented to dynamic testing by running the planner and is not usable in general by non-experts in the particular planning system it supports.

The problem as posed by researchers in the planning community is that while it is reasonably straightforward to develop a declarative model of the world in which planning is to take place, it is very difficult to accurately model the procedural knowledge - the behaviour of objects and agents - usually embodied in planning operators [6]. For planning researchers then, the knowledge acquisition problem is one of acquiring planning operators, though many KBS researchers would see this approach as akin to the early (1970s-mid 80s) view in their own field that knowledge acquisition was the process of 'finding the rules'. Using the KBS analogy one would argue that the acquisition of deeper domain structure is equally necessary for planning applications. This opens up the possibility of learning operators by induction [4], or alternatively in specific domains one may provide a library from which operators may be selected and possibly adapted in minor ways [15].

The problem may however be viewed at the knowledge level as one depending on a domain ontology which takes into account both the objects and relationships particular to the problem and the processes involved in the problem-solving methods of planning. Current attempts to provide such an ontology for planning at different levels of abstraction can be seen in PLANET [5], as well as in SPAR [19]. However these have not as yet been used to develop knowledge engineering tools.

3 The Planform Project

The Planform project is supported by the UK research council and involves researchers collaborating between the Universities of Huddersfield, Salford and Durham. Its aim is to research, develop and evaluate a method and supporting high level research platform for the systematic construction of planner domain models and abstract specifications of planning algorithms, and their automated synthesis into sound, efficient programs that generate and execute plans. Figure 1 shows a view of the abstract architecture of a "Planform System". Knowledge in an application domain is acquired via a GUI and represented in a domain model within an internal object-centred representation language [7]. Initial validation processes are embedded into the interface to support the efficient build-up of domain knowledge and to help remove bugs. This "model engineering phase" feeds into the "planner engineering phase" which is concerned with configuring appropriate planning machinery and heuristics with the domain model to produce an efficient and effective application.

In this paper we are solely concerned with the Model Engineering Phase, which concentrates on developing the domain model in isolation from a planning engine. This means that we can split

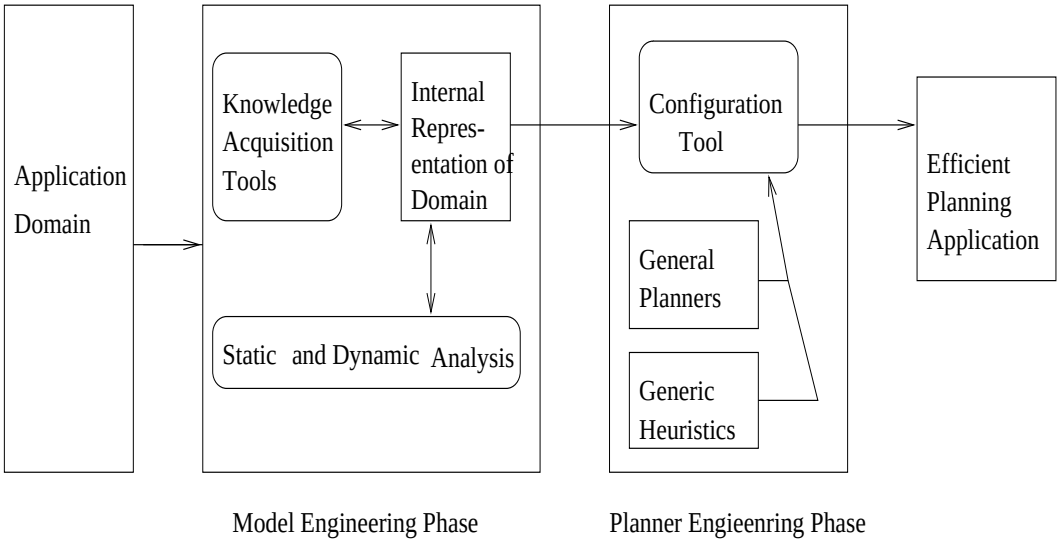

Figure 1: Architectural Breakdown of Planform

the initial validation process into separate processes such as *user inspection*, *static validation*, and *dynamic validation*. For user inspection, we have tools that map back and forth between a user-friendly, diagrammatic language and the domain model language itself. This kind of tool is similar to a graphical front-end to a formal specification language, used in the software engineering area of *methods integration* [17]. Tools for static validation essentially perform domain analysis: they reason with the model to check that it is self-consistent, to check that it is complete (in a restricted sense) and to output consequences of the model that might be useful for user inspection. For example, tools check that an operator is consistent (it never inputs a valid state and outputs an invalid state); they reason with operators and output state invariants to be visually checked by a user. Dynamic validation entails using a set of test cases with an appropriate planner and using these to test the validity of the model.

The separation of fundamental domain knowledge from algorithmic and heuristic knowledge helps alleviate the problems of domain acquisition, debugging and maintenance. In the Planform System, after validation checks have been discharged, these elements will be fused using the configuration tool. This tool would, using aspects of the domain, determine which planning technology was most appropriate and from this build up a final planning application. After an initial knowledge acquisition and static validation phase, the configuration tool can then be used to produce an application on which traditional dynamic testing and product validation can begin.

4 The Planform Acquisition and Validation Process

The knowledge acquisition method is designed on the assumption that the knowledge engineer will be trying to build descriptions of new domains using a determinate method which imposes

a loose sequence on the sub-tasks to be undertaken to develop an initial model. Once an initial rough model has been constructed development may proceed in a more iterative and experimental manner. A key design goal in building the supporting GUI tool has been to allow the creation of a specification with the tool largely automatically taking care of the detail of the syntax of the underlying specification. It should in general be impossible to construct a syntactically ill-formed specification. We also aim at allowing the specification to be created with a minimum of typing. The tool should also identify semantically ill-formed structures as the domain is being developed though we will not always require that the defects be rectified immediately. We wish to allow temporary inconsistencies in the domain model to give the engineer space to develop the strategy for adequately capturing the domain. Though consistency is not always immediately enforced it should always be possible to identify the outstanding problems.

The process of domain model development on which this is based is detailed in the literature, see references [11, 7] for more details. The domain model's ontology is described in reference [9]. Here we sketch the main steps of the knowledge acquisition process, showing how the supporting tool embodies this process. We outline two important steps of the knowledge acquisition process - acquiring domain structure and acquiring domain actions - and show how the supporting tool embodies this process. The current version of our tool assumes that prior informal analysis has already confirmed the assumptions of classical planning, such as omniscient knowledge, are adequate for the domain being described.

4.1 Acquisition of Domain Structure

The process starts with the identification of the kinds of objects that characterise the domain. The method requires that distinct collections of objects, which we call *sorts*, can be organised into a hierarchy. At this point, though it is not required, object instances for the sorts can be identified. To assist in this element of the conceptualisation we provide a visual tree editor. A snapshot of the editor is shown in figure 2 during the development of a hierarchical version of a change-tyre world. Static checking at this initial stage involves enforcing the tree structure and requiring that node names (for sorts and objects) are unique.

Once a tentative sort tree has been constructed the domain engineer should next describe the sorts by identifying predicates that characterise the states of a typical object of each sort. To create an editor for predicates (both properties and relations), where each predicate is characterised by a unique name and sequence of sort names identifying the predicates arguments, we have again provided a form of tree editor. Each of the arguments of the predicate are selected by dragging them from the visible sort tree that forms part of the editor (see figure 3). Again static checking in the form of uniqueness checks is carried out automatically on the developing domain model.

The next step in the object centred modelling process is to define sets of predicates that characterise each valid state of objects of sorts that are subject to change during the planning process. We refer to sorts subject to such change as *dynamic* where as the sorts where the objects remain

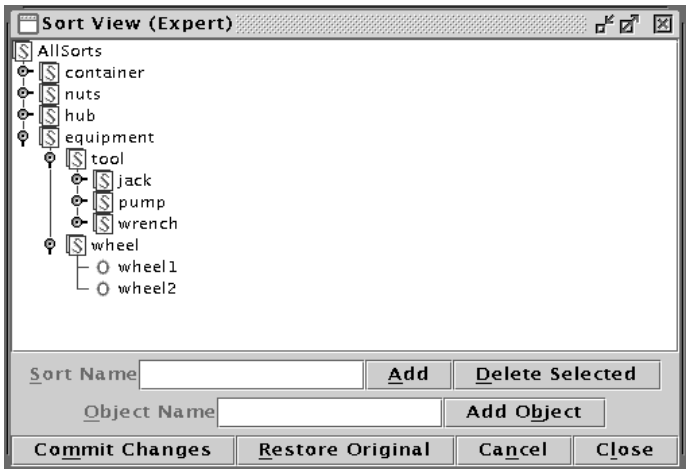

Figure 2: The Sort Editor

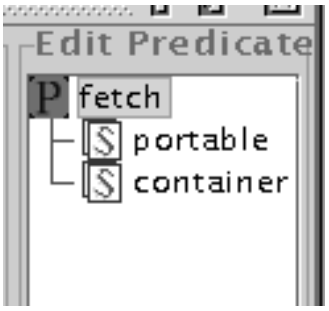

Figure 3: Example Predicate Tree

unchanged are *static*. Under classical assumptions each member of a sort may be in only one substate at any time, and that during plan execution the object goes through *transitions* which change its state.

A $substate$ of an object (called here *'substate'* to distinguish it from a state of the world, which is formed from a collection of substates) is distinguished from other possible substates by all properties and relations referring to that object that are true. If we restrict the language to the sort we're interested in, and the set of predicates to those referring to that sort, then we are interested in specifying all the Herbrand interpretations that correspond to actual substates in the domain. To illustrate consider the *boot* (*trunk*) in Russell's *Flat Tyre World*, given properties *open*, *closed*, *locked* and *unlocked* referring to a boot. There there are potentially 16 subsets of the Herbrand Base for an instance of sort Boot called boot-1, but only three that make sense in the case where the boot cannot be both open and locked:

open(boot-1) and unlocked(boot-1)
closed(boot-1) and locked(boot-1)
closed(boot-1) and unlocked(boot-1)

These three sets of predicates correspond to the only three Herbrand interpretations that model sensible states of that part of the world. If we parameterise each of these interpretations with variable "Boot" we call them *substate* definitions" (substate definitions are therefore akin to parameterised "possible worlds" restricted to one particular sort). In encoding the possible substates in this way the domain modeller is capturing the implicit negations holding between the predicates *open* and *closed* and between *locked* and *unlocked* as well as the contingent fact that the *boot* cannot be *locked* when *open*.

The problem of forming a substate definition of a *dynamic* sort is more complex when there are relational predicates referring to that sort and possibly to other *dynamic* sorts as well. This occurs in the "Flat Tyre World" when we introduce the predicate *in* which refers to *equipment* (with subsorts *wheel*, *tool* etc) as well as *boot*. Often, we may say that the predicate is used to characterise just one of the sorts referred to, normally the sort in the first argument place. For example, the relation *in* changes truth value as a result of a normal transition of a piece of equipment i.e when the equipment is removed from the boot, *in(equipment-1,boot-1)* ceases to be true. On the other hand, transitions between the instances of the three substates of boot-1 shown above are independent of the *in* predicate. Hence *in* need not be formally associated with the *boot* sort. This removes unnecessary duplication in both having to record of *boot-1* that it is in a state where *wheel-1* is *in boot-1* and recording of *wheel-1* that it is in the state of being *in boot-1*.

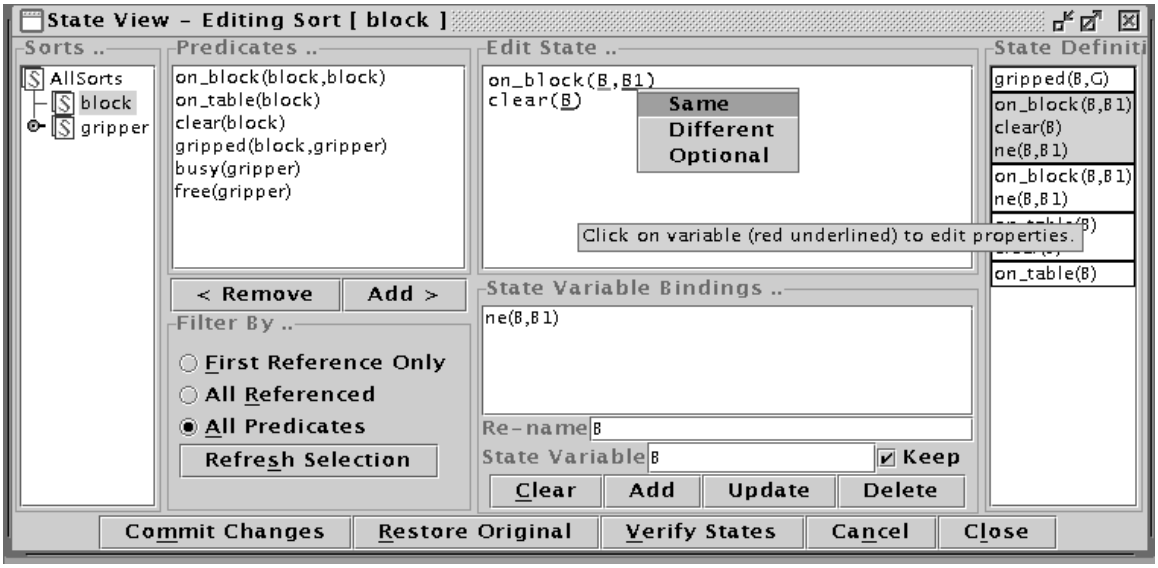

Figure 4: State Editor Tool

Providing an editor to allow domain modellers to define such substate definitions appears straight forward in that essentially the user is only required to select from the list of already defined typed predicates which will characterise the current sort being described. However this is complicated by the need in many instances to restrict the possible unification of variables of the same sort or between variables belonging to the same path in the sort tree hierarchy. This occurs for

example in the "Blocks World" where one possible state of a block is that it is *clear* and that it stands on another block. Specifying a set of Herbrand interpretations with the expression $clear(Block) \wedge on(Block, Block)$ is not adequate, as we assume that any instantiation of a substate definition forms a valid ground substate. We require $clear(Block) \wedge on(Block, Other) \wedge not\_equal(Block, Other)$ using the normal conventions for the unification of parameters. To achieve this on our graphical editor we allow the user to edit the names of variables. We provide a visual indication of possibly unifying variables and allow the user to select from a popup menu how an individual variable is to unify with a target variable and if the decision is that they must be distinct then the *not_equals* clause is automatically generated, see figure 4. This strategy for dealing with the unification of variables becomes pervasive in the set of editor tools provided.

The general use of the state editor tool is relatively self explanatory. The tool is divided into four vertical panes, where the first shows the currently defined sort tree. Selecting a sort for this tree determines which sort's states are to be edited and the remaining panes are populated relative to this sort. The second pane lists the predicates defined for the domain, which may be filtered in various ways. Selecting a predicate and pressing the add button will add that predicate to the currently edited state shown in the third vertical pane. The top section of the third pane contains the predicates added to the current state being edited, shown during the process of determining how the variables are to unify, the lower pane contains the list of automatically generated variable binding constraints. The final vertical pane shows all already defined substate definitions for the currently selected sort. Selecting any element from this list makes the substate available for editing by placing it in the state editing pane. The primary validation checks on states are done upon request using the verify command button.

The substate definitions derived from this process specify the possible Herbrand interpretations determined by considering the domain and its constraints. These form the basis of much of the static validity checks that can be carried out on the completed domain specification. At this stage validity checks are made to ensure that all *dynamic* predicates appear in one substate definition. If a predicate is dynamic then it must appear in at least one of the substate definitions for at least one of the sorts it refers to. On the other hand, if it appears in a definition for more than one sort the engineer will be warned that the formulation may contain redundant information as discussed above. This should provide the domain modeller with an initial view as to the adequacy of her selection of predicates to characterise the changes that the objects in the domain undergo.

An overarching principal behind our knowledge acquisition strategy is to allow the domain modeller to explicitly capture in a declarative manner what is known of the domain. Consequently in this spirit we provide tools to record obvious relations of implication between states and also to record impossible combinations of states. For example in the "blocks world" the domain modeller may wish to record that only one block may be placed on another block at any one time. In which case she may create a *negative invariant* composed of the predicates $on(B1, B2) \wedge on(B3, B2) \wedge not\_equal(B1, B3)$ which requires that no world description achieved during planning may validly unify with the defined negative invariant. Another type of invariant frequently encountered in planning domains is formed from *static* predicates. That

is predicates that cannot change truth value during planning. These predicates frequently define graphs or maps in "logistics" or "robot" type domains or where predicates relate to resources that might be consumed they often define sequences of values. We provide tools to allow instantiation of such predicates relative to a given set of problems in a manner very similar to the strategy we use to define predicates by drag and drop.

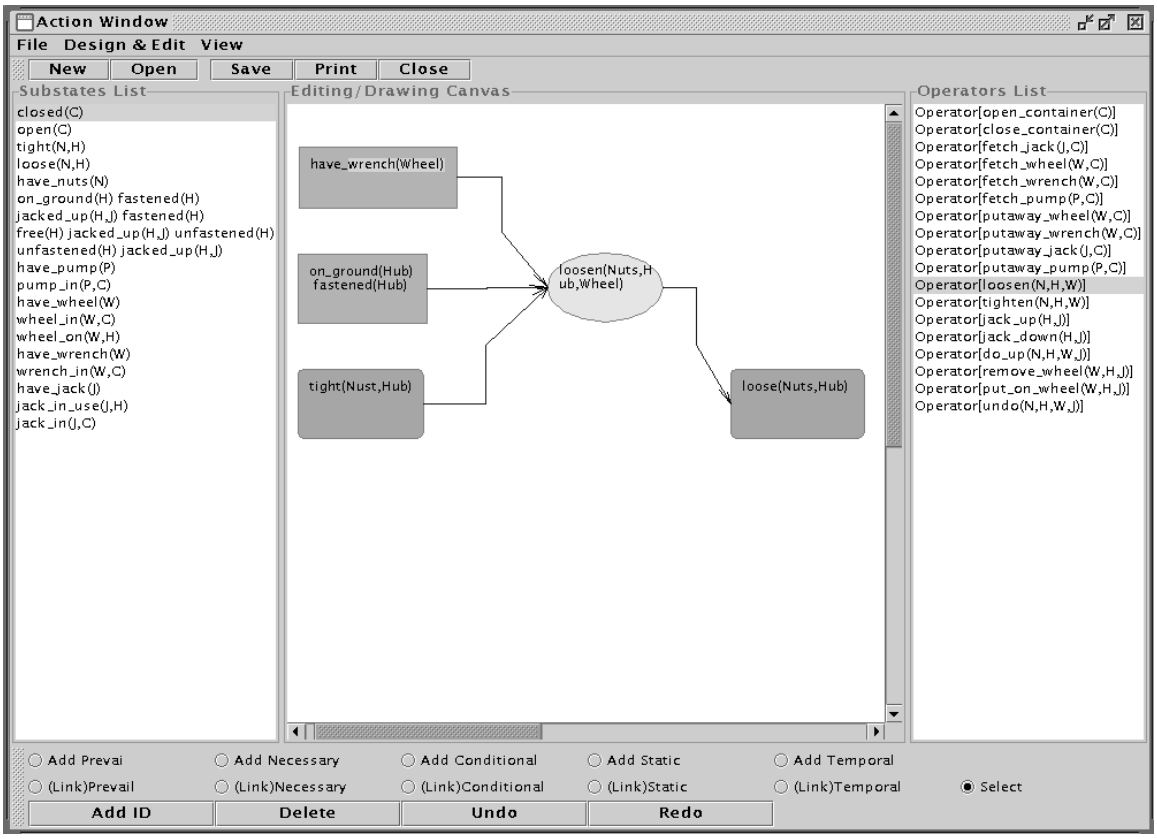

Figure 5: The Operator Editor Tool

4.2 Defining the Domain Actions

The heart of the editing suit of tools is the editor for defining operators and methods ($methods$ correspond to HTN operators [8]). Figure 5 shows a snapshot of the editor when composing an operator in the Tyre World. These editors rely on the notion that operators and methods generally cause substate changes in objects which we refer to below as *transitions*.

An action in the world is conceptualised as a set of transitions, written as (LHS $=>$ RHS). There are three modes of a transition under classical assumptions:

1. Null transition: This is one in which an object o **must** be in one of a set of substates, but

after execution of the action o will remain in that substate (i.e. LHS $=>$ LHS).

2. Necessary transition: This is one in which o **must** be in a one of a set of substates (LHS must be true), and after execution of the action o's substate will change to a certain substate RHS.

3. Conditional transition: This is one in which an object **may** be in one of a set of substates LHS. If it is in that substate when an action is performed, then the object will move to RHS.

Informally for each object changed by the application of an operator there will be a transition defining the set of substates the object may be in prior to the application of the operator and the complete determination of the substate of the object, at the level of generality dealt with by the method or operator, as a result of its application. We enable the composition of operators by the domain modeller building a simple graph of the operator by selecting the elements of the transitions from an available list of the predefined substates the object/sort is capable of being in. Once these raw state descriptions are selected to form the components of the transitions they may be edited to further control how the variables used in the substate definitions may unify and to allow some generalisation of substate definitions in the LHS of transitions.

Once a domain, with its operators and methods has been defined and a set of problem tasks described the domain modeller can run the *global verification tools* to further check the validity of the specification. *Local verification checks* can be applied to the various elements of the specification as they are created, such as checking the uniqueness of "sort" and "object" and "predicate" names, but the most powerful checks require all sections of the specification to be in place. Global checks can perform such tasks as provide a limited "reachability" analysis. Each substate that is defined for a sort can be indexed against the operators that use them either as consumers or producers. This may reveal to the domain modeller that contrary to expectation some substates cannot be produced and hence could only ever be used in the initial state of an object or that some substates cannot be consumed and hence either is only useful in the development of some other object or is the kind of substate specified only in a goal condition. Static analysis can also yield graphs of the transitions of individual sorts as mediated by the defined operators to give a graphic view of dependencies and ordering between operators and states. We are in the process of experimenting with such tools from previous work of members of the Planform project [11]

In addition to static analysis of the specification the domain modeller can dynamically check a domain against a set of problems either by using the manual *stepper* or by running a selected planning algorithm against defined test problem cases. With a stepper, the engineer chooses actions to apply in the current state to generate the consequent state and proceed in this manner to verify that the domain and operator definitions does support known plans for given problems within the domain. Where the planner is fully integrated with the environment the output will be available for animation. Other implemented planning algorithms loosely integrated with the tool

can be run from within the environment and their output will be available for textual inspection within the tool.

5 Scope of the Planform Environment

Our intention in creating the Planform Environment is not simply to develop a tool for the creation of planning domains in the internal object centred representation language but to promote the tool as a modelling tool irrespective of the final target language.

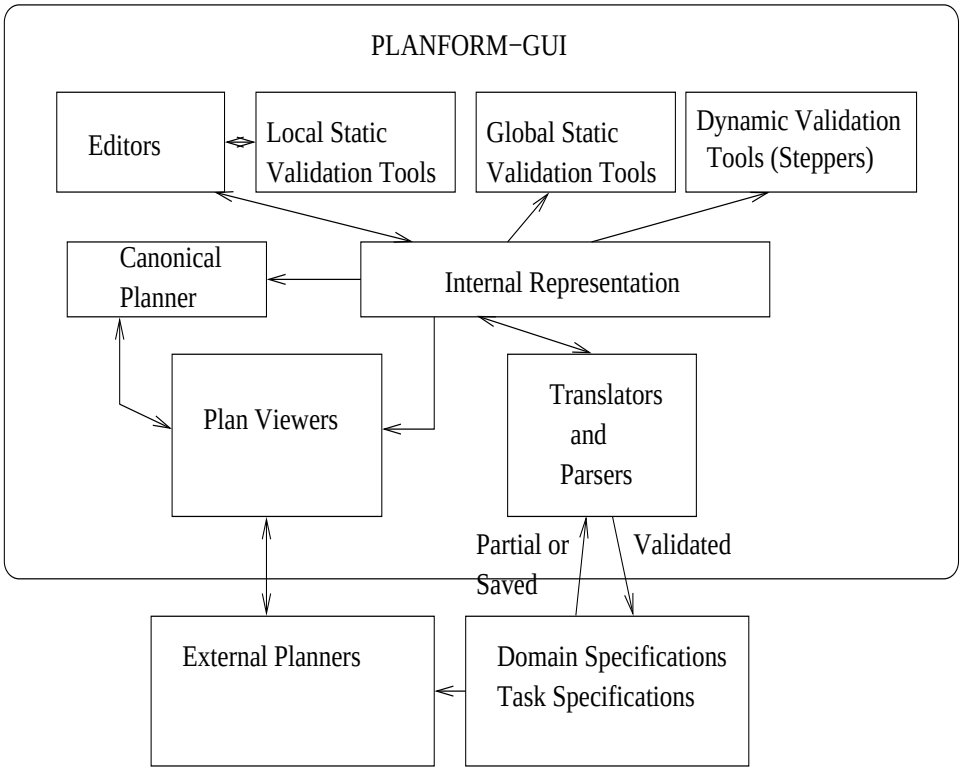

Figure 6: Architectural Breakdown of Planform GUI

The overall architecture of the environment is shown in figure 6. Central to GIPO is the object centred internal representation of domains which are manipulated by all major tool elements. These elements are the set of editors to create the domain specification along with their associated static checking routines to inform the process. The global static validation tool can be used to report on likely faults and ommisions in the model. Once a model appears to be acceptable the plan stepper and plan animator, with the associated internal planner can be used to further dynamically check the model. To enable GIPO to be used as a general domain modelling tool we have developed translators between our internal language OCL_h and PDDL [18]. We also

provide an API to enable external planning systems to interface to the tools to provide scope for testing and fielding alternative planning algorithms to those internal to GIPO.

6 Conclusions and Future Work

In this paper we have argued for a methodological approach to knowledge acquisition for planning domain knowledge resulting in a conceptualisation at the object level. We have demonstrated a tool support framework using snapshots from GIPO, a prototype GUI which supports this process for domains where classical planning assumptions are valid. This approach has the following advantages:

- The process of domain model validation can proceed independently and in advance of final product validation. Hence, domain models are developed initially independently of operational matters (a point argued for as a result of experience in the Remote Agent Experiment [12]) allowing non-planning experts to input the structure and dynamics of a domain without the need to consider how plans will be constructed by a target planner.
- Validation checks are integrated into the environment at each point of the acquisition process, with many being transparent to the user. This will reduce the emphasis on the current painstaking method of debugging via dynamic testing.
- The tool can be used to develop a domain within any concrete domain language which is consistent with the object centred planning ontology [9]. We have demonstrated this with translators to/from the PDDL language, which is based on the classical approach of encapsulating the dynamics of a domain with pre- and post condition operators.

We plan to widen the scope of the tool in several directions. Firstly, it can be extended to the non-classical domains by introducing new modalities in object transitions. For example, a probability value attached to a transition will produce operators with uncertainty of action, and transitions resulting in sets of object substates will produce non-deterministic operators and incomplete knowledge of an object's substate. Secondly, although the object centred method lifts domain acquisition to a conceptual level, the details of specifying substate definitions and transitions are still too theoretical for an unskilled user. We aim in the future to encorporate more inferrencing mechanisms to aid the unskilled user in this task.

References

[1] W.J. Clancy. Heuristic Classification. *Artificial Intelligence*, 27:289–350, 1985.

[2] David E. Wilkins. Can AI Planners Solve Practical Problems. *Computational Intelligence Journal*, 1990.

[3] R.V. Desimone, D.E. Wilkins, M. Bienkowski, and M. DesJardins. SOCAP: Lessons learned in automating miltiary applications planning. In *International Conference on Industrial and Engieering Applications of AI and Expert Systems*, 1993.

[4] M. desJardins. Knowledge Development Methods for Planning Systems. In *Planning and Learning: On to Real Applications. Papers from the 1994 AAAI Fall Symposium*, number FS-94-01. Published by AAAI Press, American Association for Artificial Intelligence, ISBN 0-929280-75-X, 1995.

[5] Y. Gill and J. Blythe. PLANET: a shareable and reusable ontology for representing plans. In *Proceedings 17th International Conference on AI*, 2000.

[6] Y. Gill, 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.

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

[8] T. L. McCluskey. 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)* , 2000.

[9] T. L. McCluskey. The OCL Ontology. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfi eld , 2001.

[10] T. L. McCluskey, R. Aler, D. Borrajo, P. Haslum, P. Jarvis, and U. Scholz. Knowledge Engineering for Planning ROADMAP. *http://scom.hud.ac.uk/planet/*, 2000.

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

[12] 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.

[13] M. A. Musen. Modern Architectures for Intelligent Systems: Reusable Ontologies and Problem-Solving Methods. In *1998 AMIA Annual Symposium, Orlando, FL*, pages 46–52. In C.G. Chute, Ed., 1998.

[14] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87 – 127, 1982.

[15] G.J. Petley, R.S. Aylett, P.W.H. Chung, and A. Rushton. Development of a Reusable Operator Library for Chemical Plant Domains. In *Proceedings, 17th Workshop UK Planning and Scheduling SIG, University of Huddersfield*, number FS-94-01, pages 145–156. ISSN 1368-5708, 1998.

[16] S.Chien, R.Hill, X.Wang, T.Estlin, K.Fayyad and H.Mortenson. Why Real-World Planning is Difficult: A Tale of Two Applications. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 287–298 . IOS Press, 1996.

[17] L. T. Semmens, R. B. France, and T. W. G. Docker. Integrating Structured Analysis and Formal Specification Techniques. *The Computer Journal*, 36:600 – 610, 1992.

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

[19] A. Tate. Roots of spar - shared planning and activity representation. *Knowledge Engieering Review*, 13:121 – 128, March 1998.

[20] A. Tate, B. Drabble, and J. Dalton. O-Plan: a Knowledged-Based Planner and its Application to Logistics. AIAI, University of Edinburgh, 1996.

[21] 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.

[22] A. Valente, V.R. Benjanins, and L. Nunes de Barros. A library of system-derived problem-solving methods for planning. *International Journal of Human-Computer Studies*, 48, 1998.

[23] B.J. Wielinga, A.Th. Schrieber, and J. Breuker. KADS - a modelling approach to knowledge engienering. *Knowledge Acquisition*, 4(1):5 – 53, 1992.

[24] D. Wilkins and K. Myers. A Multiagent Planning Architecture. In *Procceedings of AIPS*, pages 154–162, 1998.