GIPO Graphical Interface for Planning with Objects

R. M. Simpson
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
r.m.simpson@hud.ac.uk

Abstract

GIPO is an experimental research environment for planning domain knowledge engineering. The purpose of the GIPO project is to explore and demonstrate the range and scope of tools required to support the knowledge engineering aspects of planning systems creation and validation. GIPO provides support for both classical pre-condition planning and hierarchical planning. GIPO is object oriented and gives the range of benefits typically associated with OO methods in other fields of software engineering such as highly visual development methods, code re-use and fast reliable development.

Introduction

GIPO is an experimental research environment for planning domain knowledge engineering. GIPO is used as a research platform and in education. It has been used to support teaching of A.I. planning at undergraduate level. The use in education has motivated the development of a succession of revisions each introducing higher level conceptualisations and visualisation of planning domain knowledge. Our experience using GIPO in teaching indicates that it significantly simplifies the tasks of grasping the structure of existing planning domains and the task of creating and validating new domains.

GIPO has a rich environment for the creation of planning domain models it provides an interface that abstracts away much of the syntactic detail of encoding domains, and embodies validation checks to help the user remove errors early in domain development. Central to understanding and using the GIPO tool-set is a grasp of the underlying metaphor that plan execution involves manipulation of and changing the state of objects within the scope of the problem domain. Consequently the abstract definition of domain models can be done by defining the possible changes to instances of the types of objects that populate the domain problem space. For instance in the Dock Workers Robots example (Ghallab, Nau, & Traverso 2004) the domain can be modelled by describing the changes that can happen to the robots, the containers, and cranes.

The current version of GIPO has two major operating modes. The standard mode allows the creation of classical planning domains. The internal representation allows the capture of domains of the complexity of those describable in PDDL Version 1.7 without the use of any HTN features. The second mode enables Hierarchical Task Network planning and is supported by the *HyHTN* planner. In both modes the tool set contains graphical editors to assist in the creation of the domains, built in planners to solve developed problems and animators to graphically inspect the plans produced. Manual steppers, are provided by GIPO in both modes to assist in dynamically validating domain specifications. The steppers allow the user to create plans for well understood example problems and inspect points of failure if the expected plans cannot be built.

Scope of the GIPO Environment

Our intention in creating the GIPO 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.

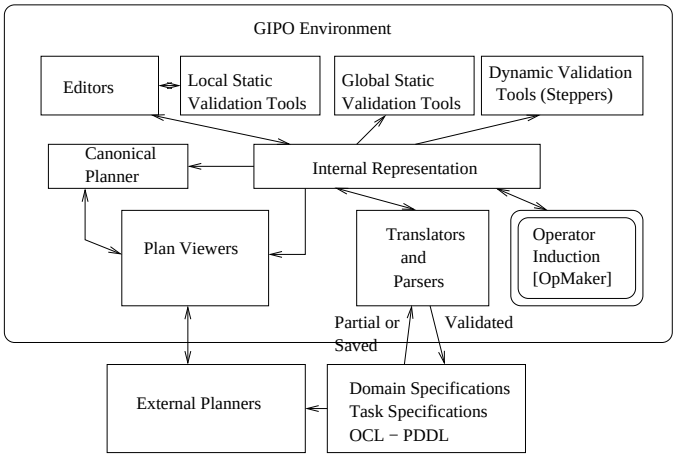

Figure 1: Architectural Breakdown of GIPO

The overall architecture of the environment is shown in figure 1. Central to GIPO is the object centred internal representation of domains which is manipulated by all major tool elements. These elements are the set of editors, augmented by *opmaker*, to create the domain specification and the associated static checking routines to inform the process. The

global static validation tool can be used to report on likely faults and omissions in the model. Once a model appears to be acceptable the plan stepper and plan animator, with the associated internal planners, 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 (Simpson *et al.* 2000). 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. Currently the interface allows planners which can input OCL version 2.1 (Liu & McCluskey 2000) or typed/conditional PDDL. As an example, we have successfully tested the integration of FF version 2.3 (Hoffmann 2000) with GIPO using our open interface without the requirement to amend any of the FF code.

Initial Domain Definition Within GIPO

GIPO provides a range of graphical editors to enable the initial creation of domain definitions. To use the basic editors the user follows the "Domain Definition Methodology" as presented in the GIPO tutorials. These basic editors closely follow the structure of domains expressed in *OCL* or OCL_h where the user must first name the classes of objects capable of participating in the problem domain. The user then defines the predicates that are used to describe object instances and then defines object class states, which roughly characterise the legal combinations of predicates that may be used to describe object instances. The concluding steps are then to define the domain operators and HTN methods, if appropriate. Problem instances can then be defined and dynamic domain testing carried out.

The basic style of domain editing as supported by the above process and editors removes from the user much of the need to have a deep understanding of domain specifications at a textual level. The use of these tools does however require reasonably sophisticated understanding of the object centric methodology. To relieve this burden further GIPO is augmented by higher level editors that abstract away from the user much of the artificial detail of domain specification. The "Object Life History Editor" is one such tool and "Opmaker" which deduces operator definitions from partially specified domains and sample plan traces is another.

Object Life Histories in GIPO

The Life History editor of GIPO allows the user to draw state machines that describe the domain dynamic object classes and automatically generate specifications from those diagrams.

In figure 2 we show the state machine for the **crane** as modelled in the DWR example. States are shown in rectangles with appropriate icons and state names, state transitions are shown as roundtangles and labelled by the name of an action that would bring about the change in state as shown by the transition arrows. The figure shows that a crane is in one of two states and that there two different actions that can trigger state changes in both directions. In figure 3 we show the **robot** state machine and show how we diagrammatically

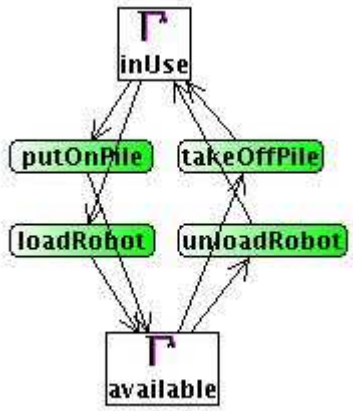

Figure 2: DWR Crane

represent the ability of a robot to change location by driving along the dock track. The "moveTo" and "takeTo" transitions are property changing transitions, that is when a robot makes one of these transitions it will change property value, in this case the location of the robot. Object classes may therefore have properties and value changing transitions alter these properties. Turning on the property inspector in the editor reveals the properties associated with the objects and any constraints placed on property changes. These are shown in figure 4. The two states shown in the robot figures 3,4 therefore represent sets of possible states individually identified by the values of the property "locatedAt". In an analogous way the transitions also represent a family of possible transitions. There is one transition for each possible location of the robot. The "loadRobot" and "unloadRobot" transitions are not property changing transitions hence when they occur the robot remains in the same location. The value changing transitions are constrained by the constraint "next" that must hold between locations for the transition to be possible. The "next" constraint will define a graph which represents the allowed route along the single line track. If robots had multiple properties then the states would represent the cross product of the possible values of the properties.

Scaling to Large Domains

Using the graphical interface, designing a large domain specification at the level of charting every object transition and all connections between them is still a complex task. We do believe, however, that the visualisations greatly expedite the task of domain definition. To assist further in providing visualisations that are easy to grasp we provide features such as the ability to selectively view part of the emerging domain and switch on and off connections linking the object class state diagrams. More importantly to aid both visualisation and re-use we add to the conceptualisation described above methods to allow some of the complexity to be encapsulated in higher order structures. Such higher order structures would form "packages". We need this both to simplify diagrams to allow the essential structure of the domain to

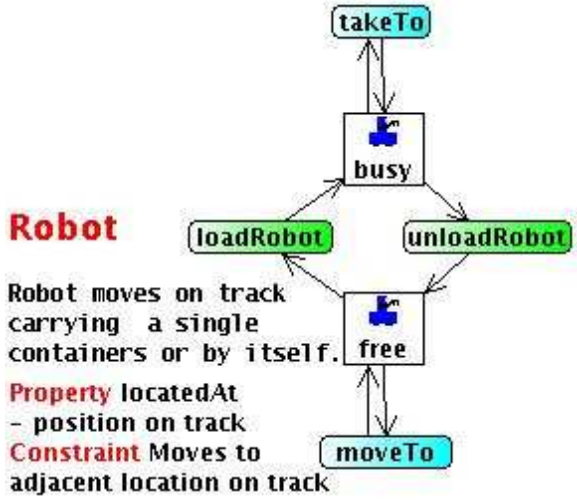

Figure 3: DWR Robot

be more easily envisaged and to allow for re-use of complex but often repeated structural elements. We require re-use both for different object types within the same domain and across multiple domains. GIPO provides mechanisms to allow domain developers to isolate structures which may be formulated into package structures providing a public interface to private sub-structures and store these in a library for re-use.

The completed DWR model is shown in figure 5. In this diagram the states of the container while on a stack are encapsulated in the package "onStack". The arrows connecting the transitions of the different object classes show how transitions must be coordinated with one another.

Working in the manner partially described above by constructing complex state machines and showing how action transitions coordinate, complete domain definitions can be built up. The textual representation of the domain is generated automatically from the diagram. To produce a testable domain all the user needs do in addition is add the information to create problem instances. GIPO also provides support for this in the "Task Editor". The user is presented with lists of predicates defining the possible states of each object class and is allowed to select possible values to instantiate both initial and goal states for tasks. This process is shown in figure 6.

OpMaker

To lower the threshold of prior knowledge required to develop planning domain models GIPO incorporates an operator induction process, called *opmaker*. *Opmaker* is aimed at the knowledge engineer with good domain knowledge but 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 con-

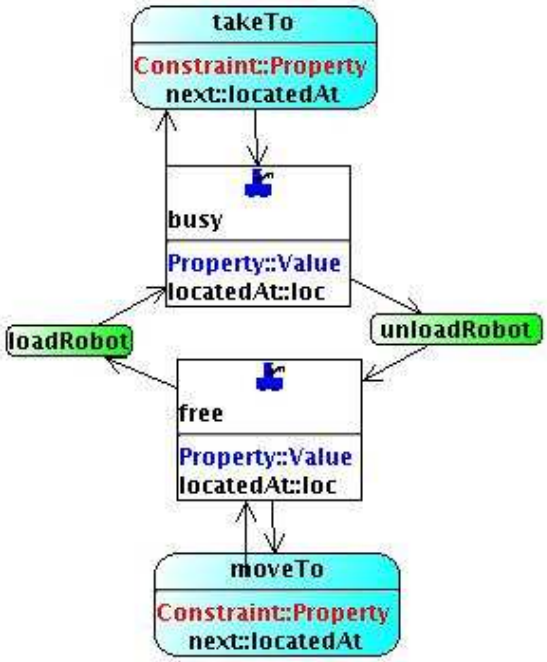

Figure 4: DWR Robot

structed the 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. This may have done using either the base editors of GIPO or by partially describing the problem using the "Life History Editor". 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 the training problem, using the task editor(see figure 6). A task specification defines an initial state for 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. 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. A snapshot of an element of the dialog carried out by *opmaker* to help infer operator structure is shown in figure 7.

The strategy *opmaker* uses relies on the structural knowledge of the domain already specified. In particular for each type of object in the domain there will exist an abstract specification of each possible state that objects of that type can be in. This specification consists of lists of lists of parameterised predicates where each sub list, when ground, defines a possible state of an object. We call such state definitions *substate class definitions*. *Opmaker* works by stepping through the training example advancing the state of each referenced object from, the initial state to the the next legal state by deducing the possible legal states of the affected objects

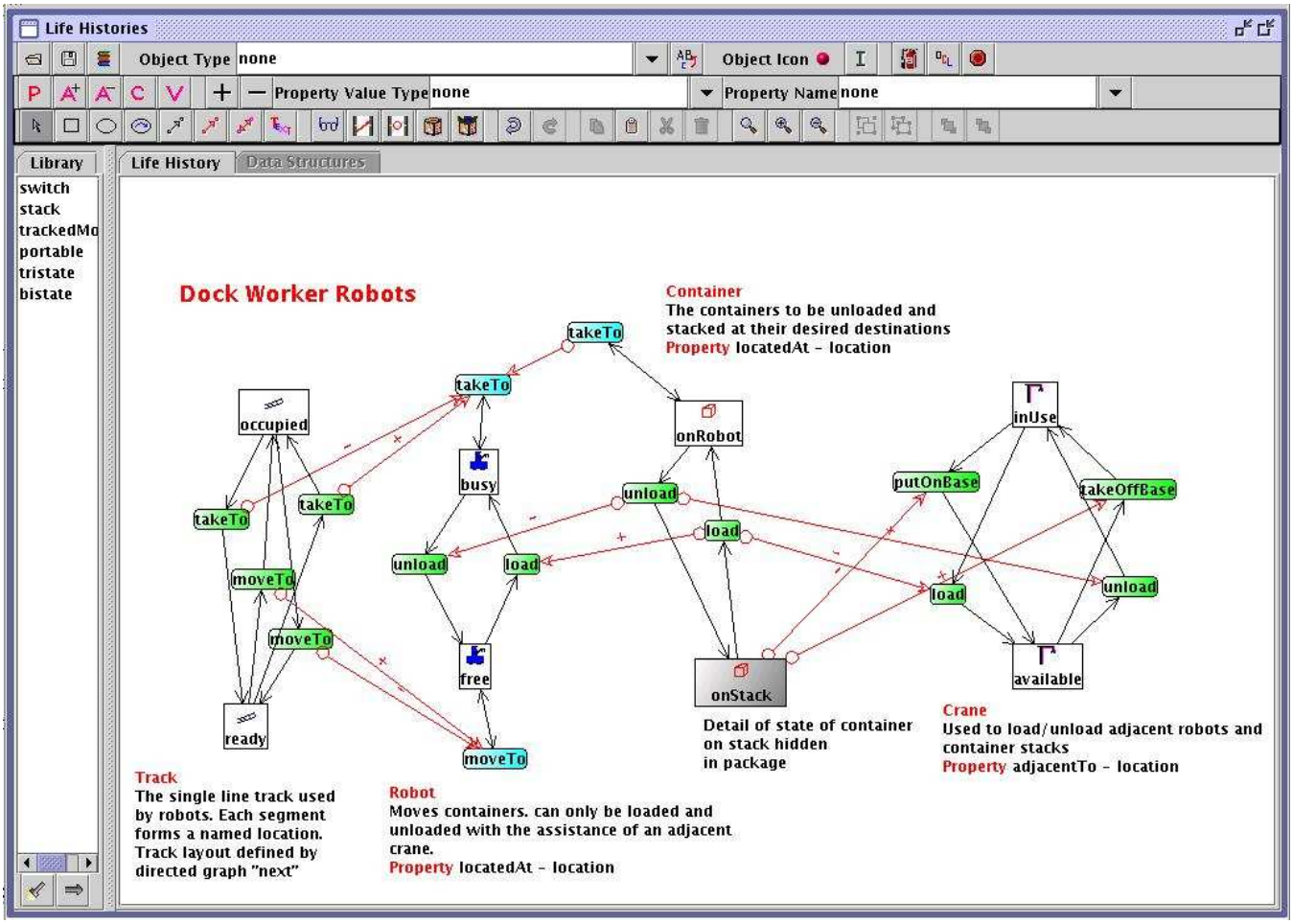

Figure 5: Screen-shot of GIPO editing DWR Domain

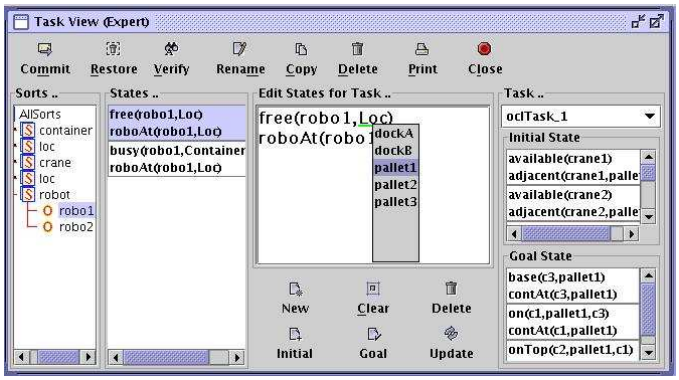

Figure 6: The GIPO Task Editor

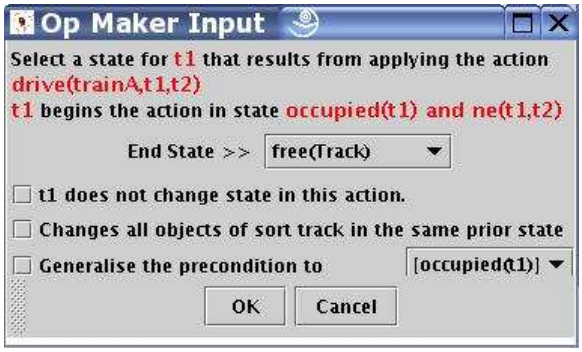

Figure 7: *opmaker* Querying User

referenced in the training action step. When there are multiple possible legal states that an object may advance to, the user is queried to determine which of the possible states the object should be in. This is shown in figure 7, where operators are being derived for a domain with trains moving on a single line track. The drop down list contains all possible legal states for the "track" instance "t1". Prior to the application of the "drive" action the track segment "t1" was "occupied". The user should confirm in this case that the result of the action will be that "t1" is now in the state "free". Once the state transitions of the named object instances are

known this information can be generalised to produce action specifications. The derived action specification will be used in future uses of the action in the training sequence and may be refined in cases where the derived action only provides a partial match with the new instance. In this way *opmaker* steps through the training sequence, querying the user and advancing the state of each object referenced in the action schema until the training sequence is exhausted.

HTN Planning

GIPO provides editors and tools to support HTN Planning as expressed in the OCL_h language (McCluskey, Liu, & Simpson 2003). In brief GIPO allows the definition of *methods* which are actions that can be defined in terms of the composition of actions as defined for classical planning. The primitive non-composite actions are defined in GIPO as described above.

HTN methods are defined in terms of three elements.

- A declaration of the changes that the composite action guarantees to bring about for identified object types.
- A definition of any precondition applying to any associated required object referenced in the guarantee.
- A graph of sub actions which if performed would bring about the guaranteed changes. This graph can contain nodes that we call "achieve goals" that represent preconditions that may have to be achieved before a specific action in the sub graph can be carried out.

In GIPO the *method* editor allows each of these segments to be represented in a graphical form. In figure 8 we see a method called "carry_direct" being defined. This is a *method* that might be used in a "logistics" type domain where packages have to be delivered by a variety of forms of transportation. The top two roundtangles define the changes guaranteed to the "package" to be transported. The guarantee requires that the package be in a state that would form a ground instance of the LHS roundtangle, namely that it is at some location *O* and that it is *waiting* and *certified*. The RHS or post condition states that the package will end up at a new destination *D*. The bottom roundtangle expresses the precondition that the city of origin of the package and of the destination be the same. The third section of a method definition is defined in a graph of actions forming the decomposition of a method. This is shown in figure 9. Both primitive actions and other methods can be used in the definition of a decomposition. Method definitions may be recursive. A decomposition may include pre-conditions that apply to the actions forming the decomposition. In figure 9 the rectangle containing the predicate $at(V, O)$ expresses the precondition that the vehicle used to *load_package* must be at the same location *O* as the defined location of the package.

To support HTN planning GIPO has a built in HTN Planner *HyHTN* which is described in (McCluskey, Liu, & Simpson 2003). To offer further support there are is also an animator for plans produced by *HyHTN* and a plan stepper. A partial snapshot of the animator in use with a "logistics" domain is shown in figure 10

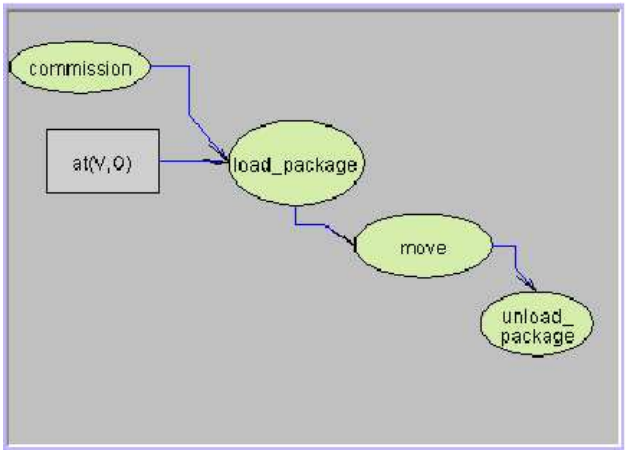

Figure 9: Method Decomposition

Domain Validation

The validation of a domain cannot be done entirely automatically, however automated assistance in this task can be provided. Within an HTN network GIPO can check that the "transparency" property (McCluskey, Liu, & Simpson 2003) is not broken by any method definition.The "transparency" property gives a guarantee that if a method's preconditions are met then the body of the method will bring about the method's postconditions. The property is checked by performing abstract execution of a "methods" decomposition body. Warnings are then displayed to the user if a step cannot be fulfilled given the specified preconditions of the method. In figure 11 we see a DWR style domain where an object is to be loaded from a gripper but where the gripper cannot be guaranteed to hold the object.

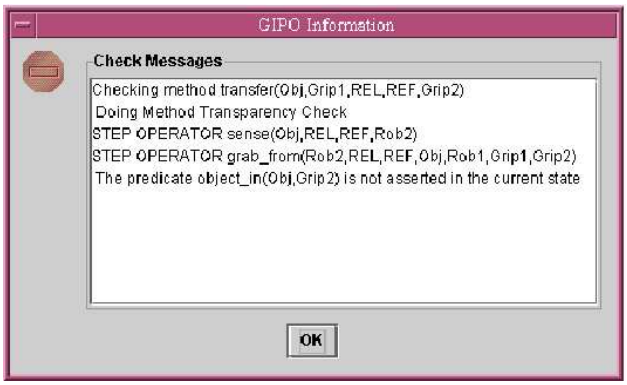

Figure 11: Output from the Transparency Tool

Within classical domains the automatic checks that GIPO can carry out tend to be at a lower syntactic level but absence of such problems as mis-spelt predicate names can still save the domain developer many hours of dynamic testing. A more powerful tool for static checking is the state and predicate use tools. These tools check the usage of each of the defined states and predicates as they are referenced in the

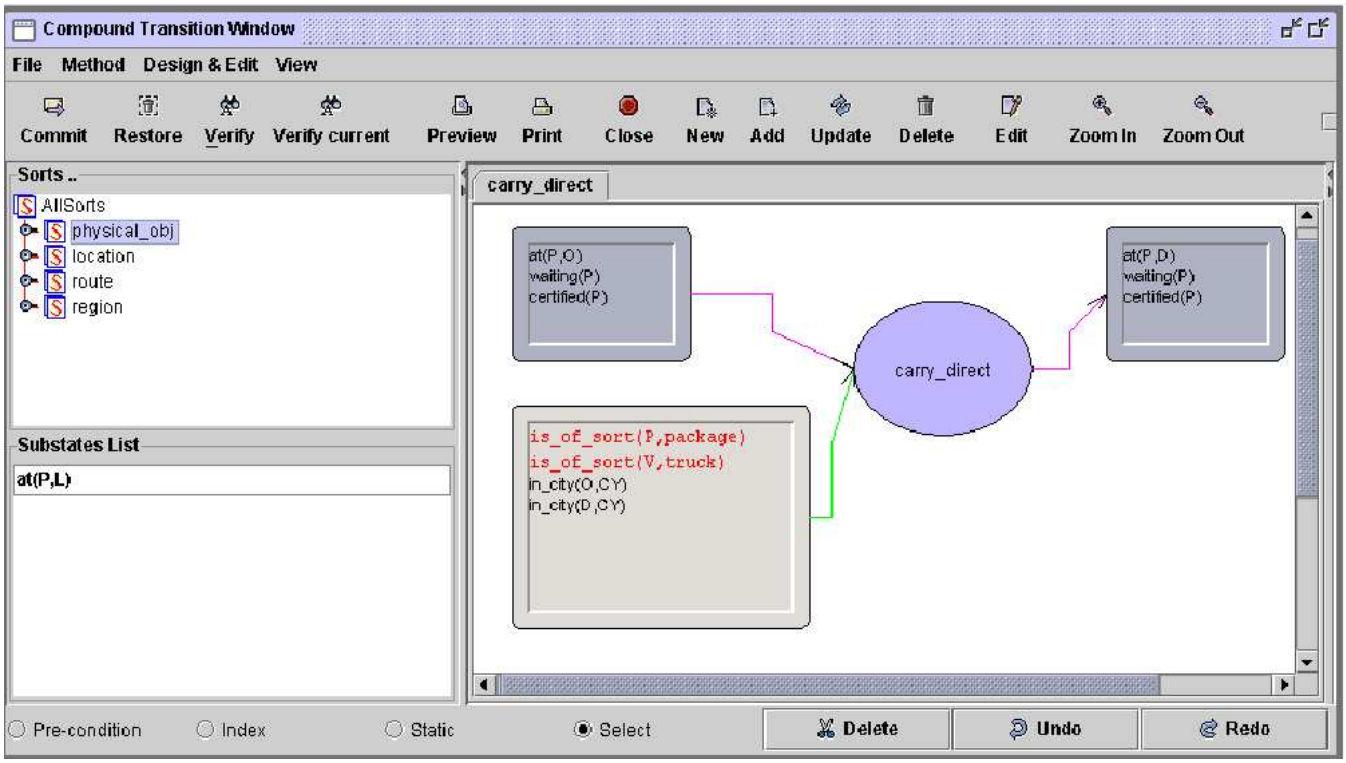

Figure 8: Method Guarantee and Preconditions

operator definitions. This is particularly useful in conjunction with *opmaker*. That a state definition is not referenced by any operators would most likely indicate that the operator coverage of the domain is still incomplete. States that are only referenced in the precondition of an operator could only play a role in the initial conditions of problem specifications, which may be correct but is worth considering explicitly. Similarly states that are only referenced in the post conditions of an operator form dead end states for objects of that kind. Again that may be what is required but is also a potential indicator of incompleteness.

Dynamic Validation

The most powerful facility that GIPO provides for dynamic validation of domains are the manual steppers. The role of the steppers is to allow the domain engineer to check that the domain specification does support known plans for well understood problems within the domain. This may be checked by running planners with the known problems, but failure to find the plan may indicate a problem or limitation of the planner rather than the domain specification. To check the domain independently of any particular planner the plan needs to be manually produced, which is done using GIPO's steppers. The stepper for classical planning works as a forward planner where the user selects the actions to solve the problem. As the application of each operator is checked the user can isolate the point where a domain definition fails to allow an action to be performed in a context where the user thinks the action should be allowed. The stepper greatly helps uncover modelling problems within a domain definition. In figure 12 a domain to test a model of multiple trains moving on a single line track is being stepped. The user is instantiating an instance of the "drive" operator to step the growing plan.

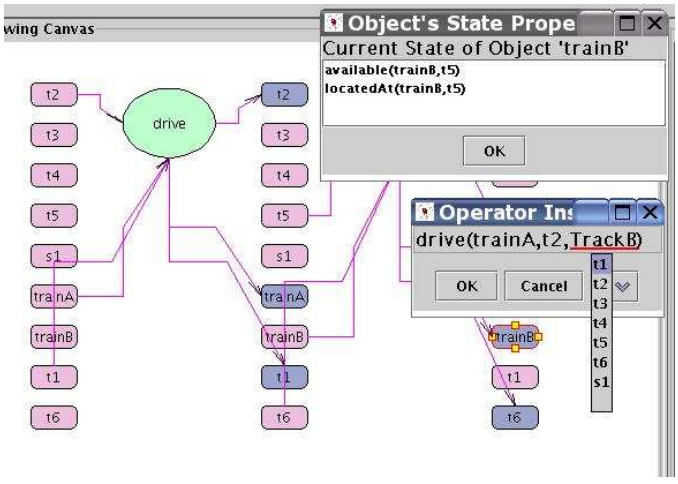

Figure 12: GIPO Stepper used to validate the DWR Domain

For HTN domains the stepper works in a top down left to right mode. When a user selects a method as part of a plan the decomposition of that method must be manually stepped. The HTN stepper incrementally produces a diagram with a

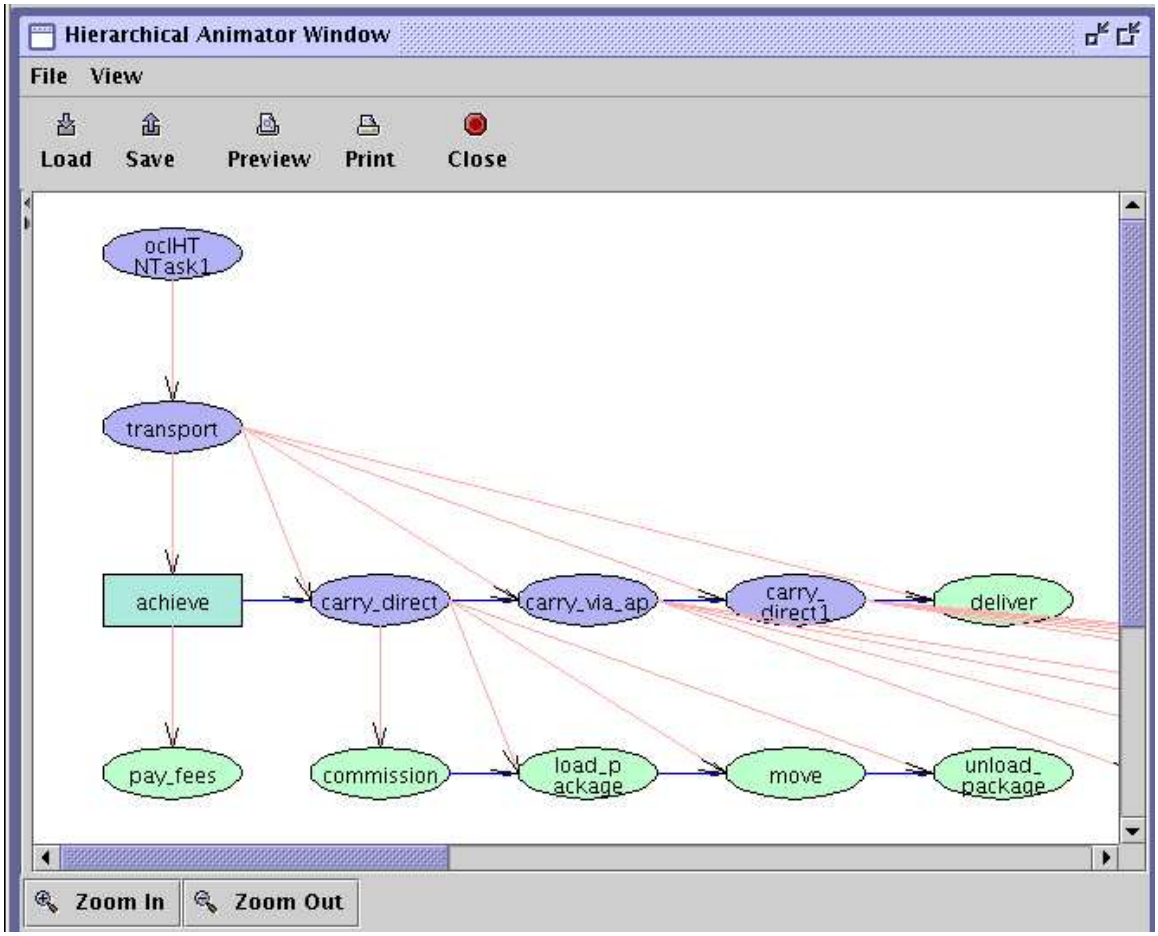

Figure 10: Partial Animation of an HTN Planner's Output

structure identical to that produced by the HTN plan animator as shown in figure 10

Implementation

GIPO is largely written in Java and is hence platform independent. The integrated planners, including *HyHTN*, are written in Prolog as are some of the other tools. The GIPO distribution includes Sicstus Prolog run time environments to support the Prolog subsystem. External third party planners can be run from within GIPO if they have a command line interface that allows the specification of input domain and problem files and they process classical PDDL. The command to run such a planner can be specified in GIPO's initialisation file. The GIPO distribution is binary though the Java sources are also made freely available.

Related Work

Similar work to our own in knowledge acquisition and engineering tends to be aimed at general KBS rather than being specific to AI Planning. For example, systems such as those based on EXPECT (Blythe *et al.* 2001) or PROTEGE (Gennari *et al.* 2003) are much more general purpose and do not aim at providing support to the very specific task of acquiring domain knowledge with a view to producing a formal specification as an output to be used with planning engines. The work on analogical reasoning (Garagnani 2004) bears some superficial similarities but differs in its purpose, in that it introduces a notation that is designed to speed up automated planning for a certain class of domains.

The Petri Net (PN) has been a well-known formalism used to specify and analyse concurrent systems (and other related systems) since the 1960's. Like the object notation used by GIPO, it is a graphical notation which is relatively easy to understand, yet has a formal basis. Petri Nets are also supported by a rich set of graphical tools. Superficially, there are many ideas in common: the idea of a 'token' in PNs is very similar to an object instance; PNs have transitions (although PN transitions are actually more like instantiated planning operators that our primitive transitions discussed here). It is not easy to compare the notations as PNs have a wide variety of extensions, but for example in the THORN system (Schof, Sonnenschein, & Wieting 1995) tokens in PNs are considered to be object instances. The main differences between our approach and PNs are that (a) PNs are aimed primarily at requirements modelling and capture for real-time systems, rather than domain modelling for

AI planning (b) PNs emphasise *execution simulation* rather than just domain modelling; (c) 'Places' in PNs do not map across naturally to the idea of states as parameterised predicates.

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.

Future Work

GIPO is still under development. The Life History editor is just at a *beta* level of release. We are still experimenting with the nature of the visualisation and with the editing mechanisms to allow the life histories to be easily produced, edited and encapsulated into re-usable library structures. User evaluation of the Life History editor is still to be carried out. It is also our intention to lift the scope of domains capable of being modelled within GIPO to correspond to PDDL level 5. We are also currently investigating links with other state machine based formalisms such as Petri Nets to see if we can further enhance the analysis of domains to support validation. GIPO is available from *http://scom.hud.ac.uk/planform/gipo*.

Acknowledgements

The GIPO software has been under development for four years. The Prolog tools and in particular the planner *HyHTN* have been developed by D. Lui. The Java interface has been developed by W. Zhoa and R. M. Simpson. Other members of the planning team at the University of Huddersfield have contributed intellectually to its development as did members of the Planform project.

References

Blythe, J.; Kim, J.; Ramachandran, S.; and Gil, Y. 2001. An Integrated Environment for Knowledge Acquisition. In *Proceedings of the International Conference on User Interfaces*.

Garagnani, M. 2004. A framework for planning with hybrid models. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling Workshop on Connecting Planning Theory with Practice*.

Gennari, J. H.; Musen, M. A.; Fergerson, R. W.; Grosso, W. E.; Crubezy, M.; Eriksson, H.; Noy, N. F.; and Tu, S. W. 2003. The evolution of Protege: an environment for knowledge-based systems development. *Int. J. Hum.-Comput. Stud.* 58.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning Theory and Practice*. Morgan Kaufmann ISBN 1-55860-856-7.

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*.

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.; Liu, D.; and Simpson, R. M. 2003. GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment. In *The Thirteenth International Conference on Automated Planning and Scheduling*.

Schof, S.; Sonnenschein, M.; and Wieting, R. 1995. High-level modeling with thorns. In *Proceedings of the 14th International Congress on Cybernetics, Namur, Belgium*.

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*.