Structural Domain Definition using GIPO IV

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

Abstract

GIPO IV is an experimental environment for planning domain knowledge engineering. The purpose of the tool is to demonstrate the range and scope of tools required to support the knowledge engineering aspects of planning systems creation. In particular GIPO provides support for the encoding and validation of planning domain knowledge, for both classical pre-condition planning and hierarchical planning domains. GIPO IV is a significant improvement on previous versions in that it (a) extends the expressivity of domains describable within the classical tools of GIPO (b) extends the acquisition facilities with the introduction of a more structured and scaleable approach to knowledge capture.

Introduction

GIPO provides a rich environment for the creation of planning domain models 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. 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. The intention is that the domain designer work at a higher level of abstraction than that required by literal-based planning domain definition languages. The current version of GIPO, GIPO IV, inherits much of the functionality of GIPO III but can represent more expressive domains: the internal representation allows the capture of domains of the complexity of those describable in PDDL Version 2.2. To support domain definition 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 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 prove invalid when abstractly executed.

The Architecture of the GIPO Environment

Our intention in creating the GIPO IV Environment is to construct a tool suitable for use with a range of target planning languages. We believe that the conceptualisation of domain description within GIPO is at a more general level than that presupposed by planning languages such as PDDL or NDDL and that enables the possibility to translate the graphical domain descriptions to such target planning languages. The user can select the target language to be used by associated planners, currently either PDDL or OCL (though we plan to extend this to NASA's NDDL).

The overall architecture of the environment is shown in Figure 1. At the heart of GIPO is the object centred internal representation of domains which is manipulated by all major tool elements. These elements are the set of editors

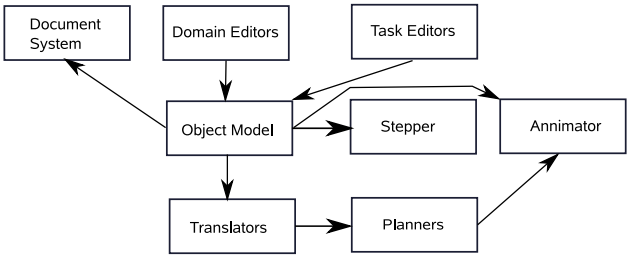

Figure 1: Gipo Archicture

to create the domain specification and the associated static checking routines to inform the process. 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. We 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 integrated FF version 2.3 (Hoffmann 2000) with GIPO using our open interface without the requirement to amend any of the FF code. The link to external planners is very flexible and can be used with any planner which can be run from the command line and takes as inputs PDDL domain and task files. Python scripts are used to provide any

necessary parameters and call the planner. Similarly python scripts post process the output of the planner to remove any extraneous text and leave only the output plan in a standard form. The pre and post processing Python scripts need to be written specifically for each planner but typically are small and easy to write.

Domain Definition Within GIPO IV

GIPO provides an editor to create "Concept Diagrams" in the style of UML class diagrams to define the kinds of objects involved in the domain to be defined. Relationships of inheritance and aggregation can be specified between the concept types. The diagrams also provide opportunity to define "properties" that are common to all object instances of the various concepts defined.

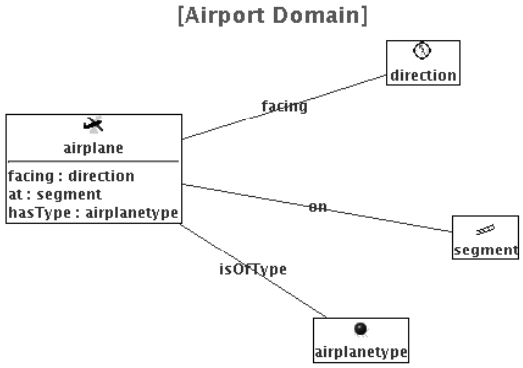

Figure 2: Concept Diagram

Object Life Histories in GIPO

The Life History editor of GIPO allows the user to draw state machines that describe the dynamics of the objects of a chosen object class. A "Life History" diagram is used to name the states that object instances from the object class can occupy and to show the possible transitions between states. An object making a transition may be changing state, and may change values of some of its properties. Within the Life History editor in addition to naming the possible states of an object, states can be further annotated to show properties of object classes that hold only when an instance of the object is in the specified state. In the "translog" domain a *package* may have a property *vehicle_id* which is only applicable when the package is in a *loaded* state. In cases like this the property is being used to record an association with a package and the truck or train it is loaded into. That association only exists while the package is loaded inside a carrier. In this way we differentiate between properties that apply to every possible state of objects of a class, these are defined in the concept diagram, and those that only have limited scope to some of the states of an object. These are defined in the life history diagram. An example of a life history diagram for aeroplanes in the "Airport" domain is given in Figure 3. In the example diagram the round node attached to the *pushing* state indicates that this is the only legal possible states for an aeroplane to start at in any problem instance. The circular nodes attached to the *airborne* and *parked* nodes indicates that these are possible final states of aeroplanes in problem instances. In the diagram the start circular nodes are differentiated from the stop nodes by colour. The use of start and stop nodes is optional as in many domains there is no clear distinction to be made between initial and final states outside the context of specific problem instances.

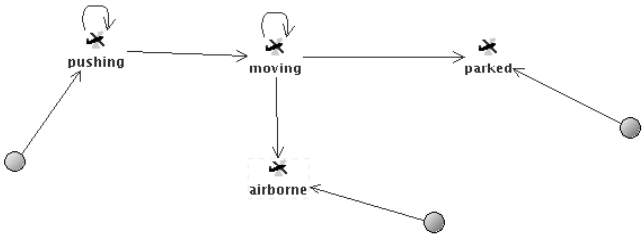

Figure 3: Airplane Life History

Life history diagrams must be created for each concept type where instances of that concept type change their states or attributes as a result of planning. In the Airport domain *aeroplanes* and *runway segments* may be regarded as dynamic while *directions* and *airplane types* may be regarded as static. Choosing which concepts types should be dynamic and which static is a design decision which may both reflect the physics of the domain and the ease of capturing the dynamics of the domain.

Coordination Diagrams

Coordination diagrams are used to show how objects of two or more concept types coordinate their dynamic movements. At least one of the concept types shown in a coordination diagram must be dynamic and have a corresponding life history diagram. Coordination diagrams allow transitions and states to be linked. There are two types of links that may hold between dynamic objects. The first showing that transition must, or may conditionally, happen at the same time. The second between a transition and a state indicates that an object must occupy the designated state before an object of the other concept type can make the indicated transition. Links between dynamic concept types and static types have no temporal meaning and are only used to show that a transition of the dynamic type depends on the availability of instances of the static concept type with appropriate properties. In the diagram 4 there are multiple transitions between the *blocked* and *free* states of runway segments. This is required because the various numbered transitions are linked to transitions of aeroplanes that occur at different times. Linking a single transition between *blocked* and *free* with *same time* links to both the *pushing* self loop transition and the *moving* self loop transition would imply that in order for an instance of a runway segment to change from the *blocked* state to the *free* state there must simultaneously be an aeroplane that is making the *pushing* transition and

one that is making the *moving transition*. What we want to depict in this domain is that there are multiple ways that a runway segment may become *free*.

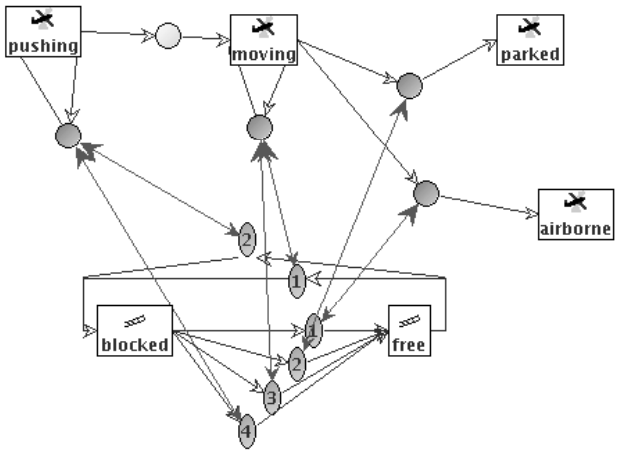

Figure 4: Coordination Diagram Airport Domain

Specifying Non-Graphical Constraints

Many domain constraints are captured graphically, and are automatically translated into symbolic representations. To fully capture constraints, however, the design engineer must provide information about how properties and non state attributes change as object transitions are made. Some constraints can be specified in terms of relationships between the properties that can be captured in simple relational predicates. One example is the assertion that the properties of two objects must have equal values. In many classical domains such simple constraints are all that are required to complete the domain specification. More complex constraints require compound predicate calculus expressions to capture the constraint or are constraints that refer to dynamic relations that cannot be captured in terms of constraining the functional values of object properties. Constraints of both sort can be specified using right click dialog boxes linked to specific transitions on either life history diagrams or coordination diagrams.

Coordinating Properties

The basic assumption made about properties is that their values persist when objects change state unless otherwise specified. Accordingly in life history diagrams where the diagram relates to changes of state of objects of a single concept type the primary constraints that need to be shown are those indicating how a property changes when a transition is made. A secondary type of constraint is that different properties of an object have a defined relationship to one another before a transition can be made.

A typical example of a property constraint might be when a Mobile moves from location to location along a defined route. To define the route the domain designer may want a static predicate *(next location location)* to define their route.

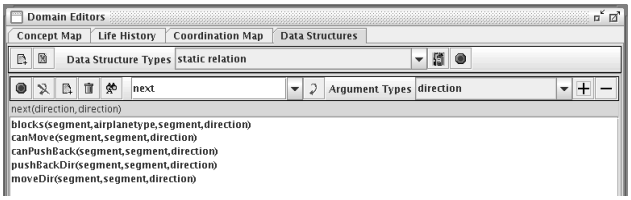

Figure 5: Defining Static relations

The predicate can be defined in the "Data Structures" editor as shown in figure 5. The data structure editor can be used to define predicate prototypes for either static or dynamic relations. Note that the designer does not directly define state predicates nor property predicates, they are derived from the state and property names provided in the various diagrams. When a property constraint is defined the dialog box shown

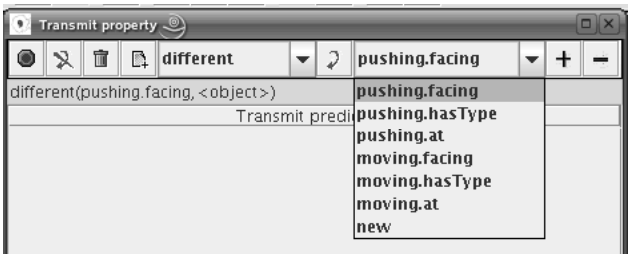

Figure 6: Constraining Property Values

in figure 6 presents the user with all defined relations that refer to one or more of the types of properties relevant to the transition. They then select the appropriate qualified properties to fill in the argument places to the constraint predicate. Only arguments of the correct type will be accepted by the dialog box.

Complex Constraints and Updates

Complex constraints may be required in a domain, either where "fluents" are involved or where there is a requirement to quantify over property values or relations. One such complex constraint is required in the "Airport" domain where a plane may only move to a new segment of runway if a plane of that type and facing in the direction of the plane will not block any other plane currently occupying other segments of the runway. To allow such complex constraints to be defined we have developed a semantically and syntactically aware dialog. This enables an algebraic expression to be built up incrementally, such that at any point in the process the designer can only add appropriate elements. In figure 7 we see that the designer has already chosen "not exists" to capture the notion of none and the highlighted argument position is over the quantifier brackets. In the drop down box of possible elements to add to the expression at this point, the designer is only presented with possibilities representing objects. The appropriate choice of range of possibilities presented to the designer at any one point is made possible because the dialog box is generated after the user selects a

transition in the constraint editor. That transition must refer to an object of the specific type and its relation to objects of other types as shown graphically by the temporal connections drawn between transitions and states in the coordination diagrams. In this way the system is aware of the possible objects and their properties that may be involved in a constraint on a transition.

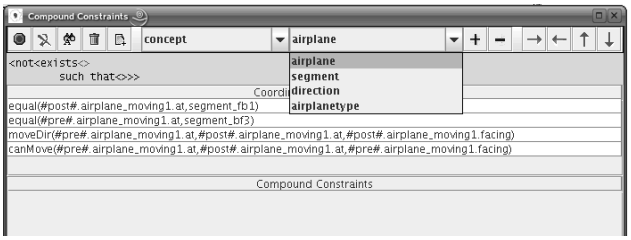

Figure 7: Defining Complex Constraints

Defining Domain Problems

To produce a testable domain all the user need do in addition to the process described above 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 allows the user to select possible values to instantiate both initial and goal states for tasks. This process is shown in diagram 8.

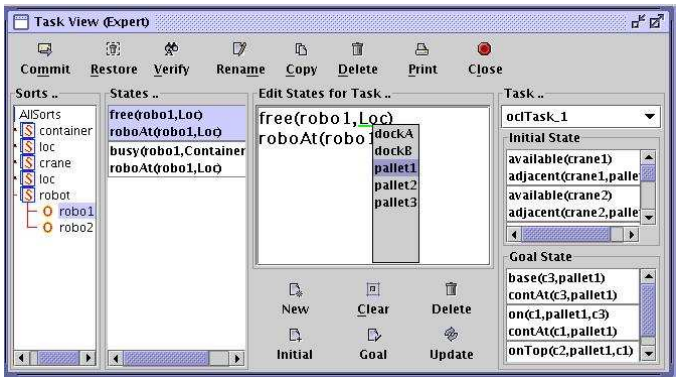

Figure 8: The GIPO Task Editor

The Representation of Time

The representation of time in Life History diagrams is determined by the designer’s choice of target language. In ”classical” mode time is not represented explicitly. Actions are instantanous and plans are simply ordered, or partially ordered sequences of actions. In ”durative action” mode actions/transitions take time and effects may be specified to take place at the start or at the end of such transitions. In this mode the constraints dialog allows the option of defining when a predicate changes value. Finally in ”continuous process” mode transitions are instantanous but states persist over time. In this mode duration dependant update functions can be defined and associated with specific states.

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 conceptualisation and the visualisations provided by GIPO greatly expedite the task of domain definition. To assist with the visualisation we have split the task into three main types of diagram as described above. This is one of the main innovations distinguishing GIPO IV from GIPO III. The objective of this was to limit the amount of information being carried in any one diagram. By their nature ”Concept diagrams” which only have one node per object class are unlikely to be very large. Individual life history diagrams with the domains we have been experimenting with drawn from the planning competition are typically fairly small. By contrast ”Coordination Diagrams” can potentially grow to be quite large but we allow the coordination between different object classes to be split between different coordination diagrams. For example a domain with three object classes, A,B and C could have the links between them shown on three coordination diagrams. Class A with B, class A with C, class B with C. In practice if Bs are never linked with Cs except when both are linked to As the third diagram would be unnecessary or if the three life histories are small one coordination diagram may be the clearest way of showing the information. In this way GIPO provides flexibility to the design developer to decide the best way of presenting the domain details.

Domain Validation

The validation of a domain (as with software in general) cannot be done fully automatically, but assistance in this task can be provided. Within classical domains the automatic checks that GIPO can carry out tend to be at a syntactic level but absence of such problems can still save the domain developer many hours of dynamic testing. The type checking inherent in the various constraint dialogs also ensures that predicates are only supplied with arguments of an appropriate type. GIPO also provides assistance with dynamic testing. The most powerful facility that GIPO supports is the manual steppers. The steppers work as forward planners 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 diagram 9 a domain to test a model of multiple trains moving on a single line track is being stepped.

Related Work

The most closely related work to our own with a similar level of ambition is *itSimple* (Vaquero, Tonidandel, & Silva 2005), which was also exhibited at the first ICKEPS competition. Superficially our work has gravitated closer to theirs by our

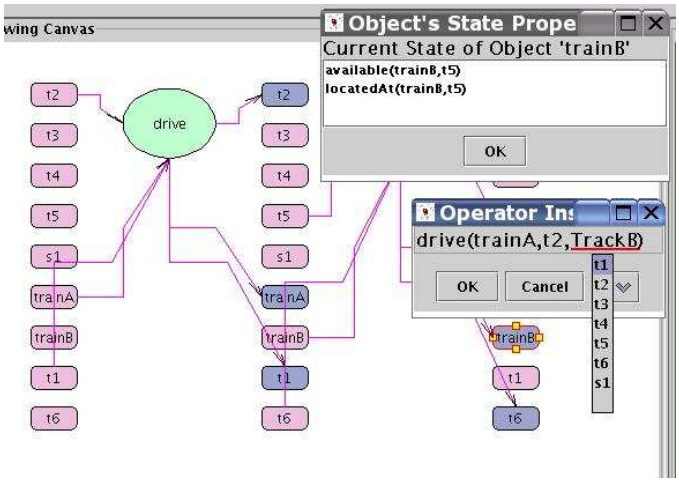

Figure 9: The GIPO Classic Domain Stepper

adopting OO class diagrams to present an overview of the intended systems. Unlike *itSimple* we have not followed the conventions of *UML* for our diagrams, prefering instead to design our own to better capture the semantics inherent in planning applications. Systems such as *Sipe* (Wilkins 2000) and *O-Plan* (Tate *et al.* 2005) differ from our own in many ways but from our perspective primarily in their being designed around specific planning engines. GIPO maintains an open API for planning engines and tries to roughly track developments in the capabilities of planning engines as indicated in the evolving standard of PDDL. Other work in the knowledge engineering field of planning tend to focus on different points in the engineering life cycle. For example *ModPlan* (Edelkamp & Mehler 2005) focuses primarily in the area of improving and transforming domain representations to allow successful planning, rather than assistance in the development of initial models. *ARMS* (Wu, Yang, & Jiang 2005) by contrast tries to induce planning models from large numbers of plan instances, although this approach seems limited by the very need to have large number of valid plan examples. In KBS, 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.

In the area of engineering there are a number of graphical modelling tools some of which overlap in purpose with our own. In particular the system *Ptolemy II* (Lee 2003) among its many capabilites allows the user to model hybrid systems which are very closely related to the semantic model of PDDL level 5. Potentially the types of tool presented in systems such as *Ptolemy* could be used to animate plans controlling complex dynamic processes.

Future Work

GIPO IV is still under development. The graphical editors are at a *beta* level of release. We are still experimenting with the nature of the visualisations and with the editing mechanisms to allow complex constraints to be easily expressed. The Life History editor of GIPO III has been thoroughly tested by students studying AI at the University of Huddersfield, where they were tasked with choosing potential planning applications and designing and testing their domain definitions within GIPO III. The domains produced by the students can be seen on the GIPO website, referenced below. GIPO IV is to be further developed in conjunction with a project to model planning problems with ”flood disaster management systems”. While this work is ongoing the interface and modelling capabilities of GIPO itself will remain fluid. GIPO is available from *http://scom.hud.ac.uk/planform/gipo*.

References

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

Edelkamp, S., and Mehler, T. 2005. Knowledge acquisition and knowledge engineering in the ModPlan workbench . http://icaps05.uni-ulm.de/.

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.

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*.

Lee, E. A. 2003. Overview of the Ptolemy Project. http://ptolemy.eecs.berkeley.edu/.

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 .

Tate, A.; Dalton, J.; Levine, J.; Polyak, S.; and Wickler, G. 2005. O-Plan - Open Planning Architecture. http://www.aiai.ed.ac.uk/oplan.

Vaquero, T. S.; Tonidandel, F.; and Silva, J. R. 2005. The itSIMPLE tool for modeling planning domains. http://icaps05.uni-ulm.de/.

Wilkins, D. 2000. SIPE-2: System for Interactive Planning and Execution. http://www.ai.sri.com/ sipe.

Wu, K.; Yang, Q.; and Jiang, Y. 2005. ARMS: Action-relation modelling system for learning action models. http://icaps05.uni-ulm.de/.