GIPO II: HTN Planning in a Tool-supported Knowledge Engineering Environment - Demonstration Abstract

R. M.Simpson, T. L. McCluskey and **D. Liu**
School of Computing and Mathematics
The University of Huddersfield, Huddersfield HD1 3DH, UK
ron,lee,scomdl2@zeus.hud.ac.uk

Introduction

In the demonstration we provide an overview of an experimental environment for engineering and prototyping HTN planning applications. In contrast to operational planners which are aimed at solving real problems and applications, we are trying to develop a platform that on the one hand can deal with structurally complex domains, but is also transparent and portable enough to be used for research and experimental use. Further, we are researching into a wide spectrum of planner development - the acquisition and the engineering of planning knowledge as well as the generation of plans. We introduce GIPO-II, a continuation of the work on GIPO (McCluskey, Richardson, & Simpson 2002). GIPO-II supports the building of hierarchical domain models, encoded in OCL_h, and incorporates powerful static validation techniques. For HTN planning in particular, we have implemented a semantic check on hierarchically-defined operators which allows users to evaluate the *transparency property* on them (McCluskey & Kitchin 1998). To *dynamically* test domains GIPO-II has an API for third party planners. We have developed a native hierarchical planner for GIPO-II called *HyHTN*. This is a new hybrid planner which exploits the advantages of state-advancing planners such as SHOP (Nau *et al.* 1999) and the efficiency of classical forward-search planners such as FF (Hoffmann 2000). GIPO was first released in 2001; GIPO-II is planned for release in 2003 and will be available from the same website.

Encoding OCL_h models with GIPO-II

OCL_h is a structured, formal language for the capture of hierarchical, HTN-like domains (McCluskey & Kitchin 1998). As with *OCL* it is based on the idea of engineering a planning domain so that *the universe of potential states of objects is defined first, before operator definition*. This approach has several advantages, not least that operator schema can be induced from examples, helping Knowledge Acquisition (McCluskey, Richardson, & Simpson 2002). The fact that *OCL* has traditionally used a different syntax from PDDL is largely irrelevant as GIPO-II insulates the user from detailed syntax, displaying for example object class hierarchies graphically. PDDL can be generated when required by export tools as demonstrated in reference (McCluskey, Richardson, & Simpson 2002). What is relevant and additional to the AIPS-2002 version of PDDL is that OCL_h is object centred, and designed for capturing domains hierarchically. These kinds of pragmatic additions draw a distinction between languages for communicating the physics of a domain and a more natural, graphical language for domain modelling.

A knowledge engineer uses GIPO-II to encode an OCL_h domain model firstly by grouping objects within classes under a class hierarchy. Each class in the hierarchy may have a "behaviour" in the sense that objects of that class have changeable properties and relations. An object may inherit behaviour from each class above it in the hierarchy. Thus in a transport logistics application if an object is a train-engine, it has a changeable relationship 'pulling' with train-cars. As it is a physical object, it inherits a changeable property of 'position' higher in the hierarchy. Thus objects have changeable states at various levels of the hierarchy. This type of representation is beneficial in knowledge-based applications, as more generic knowledge is stored at high levels in the hierarchy.

After inputing an initial specification of states using and supported by the GIPO-II tool, the user then uses the environment to specify primitive and hierarchical operators (the latter we call *methods*), via basic GUI tools or with the help of an induction tool (McCluskey, Richardson, & Simpson 2002). Operators and methods contain statements about *transitions of typical objects of an object class*, where a transition is written $LHS \Rightarrow RHS$, and specified in various ways, as follows:

1. identity transition: this means an object must be in a certain (set of) state(s) before the operator can be executed and stays that way; this is equivalent to a operator 'prevail' condition
2. unspecified *RHS* transition: this means an object must be in a certain (set of) state(s) before the operator can be executed and it is not specified what the final state of the object is; this is equivalent to a pre-condition
3. unspecified *LHS* transition: this means an object must be in a certain (set of) state(s) after a certain point in execution - it is not specified what the initial state of the object is; this is equivalent to posing an 'achieve-goal' in state space planning
4. specified, necessary transition: this means an object must be in a certain (set of) state(s) and goes through a transi-

tion to a certain (set of) new states; this *necessary* transition contains both pre- and post conditions

5. conditional transition: this means an object *may* be in a certain (set of) state(s); if this is the case at execution time then the object goes through a transition to a certain (set of) new state(s).

Hence this abstraction uniformly encompasses goal conditions, pre-conditions, necessary and conditional effects; further, this formulation is 'hybrid' in the sense that it is useful for both HTN and operator-based formulations. Primitive operators have the general form of being a set of parameterised transitions where each transition refers to one object. In practice we limit the scope of these operators to fit in with the planner technology we are using. OCL_h primitive operators have transitions of type 1, 4, and 5, *and* are assumed deterministic, so that whenever the *LHS* of a transition is instantiated, the *RHS* must specify a unique state of that object at one or more levels in the object hierarchy. Default persistence works as follows in this scheme for necessary and conditional transitions: if NO predicates from some level N in the hierarchy appear in the right hand side of a transition, then the state at level N persists. If predicates from level N in the hierarchy appear in the right hand side of a transition, then they must specify a unique substate class at that level. Methods (hierarchically defined operators) have the form:
(*Id*, *Transitions*, *Statics*, *Temps*, *Decomposition*)
An example method from a transport domain model is as follows (parameters are in capital letters):

```
(carry-direct(P,O,D),
  [ (package, P,
     [at(P,O), waiting(P), certified(P)] =>
     [at(P,D), waiting(P), certified(P)] )]
  [is-of-sort(P,package),is-of-sort(T,truck),
   in-city(O,CY), in-city(D,CY),
   road_route(O,D,R) ],
  [before(1,3), before(2,3),
   before(3,4), before(4,5)],
  [commission(T,P),
   achieve((truck,T,[at(T,O)])),
   load-package(P,T,O), move(T,O,D,R),
   unload-package(P,T,D) ])
```

Id is the name and parameter list of the method. This contains all the parameters used in the *Transitions*. Transitions may be of type 2 and/or 4 (the example contains only one transition of type 4). *Decomposition* contains a list of method names and/or operator names and/or or 'achieve-goals', and together with its constraints, forms a *task network* (the example contains one achieve-goal and reference to four primitive operators). *Statics* is a list of constraints on parameters in the method, and *Temps* is a list of temporal constraints on the members of *Decomposition*, where number *n* refers to the *nth* element in the decomposition list.

Methods require a statement of transition(s) of the object(s) which are necessarily changed from one state to another (in the example above, the package P is necessarily changed). An HTN operator may change many objects' states by its decomposition and execution, and the final states of objects may depend on which decomposition is chosen (e.g., the initial state of an object may be unknown as is the case for the truck T in the example above). However there exists a set of objects which are necessarily changed to a particular state, and these should be declared in the 'Transition' slot of a method's definition. GIPO-II provides strong graphical support for the construction of operator and method definitions. The composition of operators and methods is largely achievable using simple processes of drag, drop and menu selection.

Domain Validation

GIPO-II includes tools to statically check the consistency of the emerging domain specification. Some checks are invoked by the user when appropriate, others operate in the background to help prevent errors. Example features of the domain checked are that
(a) the object class hierarchy is consistent
(b) object state descriptions satisfy invariants
(c) predicate structures and operator schema are mutually consistent
(d) task specifications are consistent with the domain model.
(e) method definitions are transparent i.e. they do bring about the transitions they specify.

The process of building up a domain in GIPO-II, as it guarantees these properties, prevents the occurrence of many of the errors present in hand crafted models.

In addition to supporting static checking of the domain model, GIPO-II provides support for dynamic testing of the model. Plan construction itself is the best demonstration of the adequacy of a domain model in accurately reflecting the domain itself.

To dynamically test domains the native *HyHtn* planner can be used to attempt to solve well understood problems. Additionally GIPO-II incorporates a stepper allowing the user to plan manually. This is predominantly used in the domain debugging phase, as this incremental activity isolates bugs that have not been uncovered by the static checking tools.

References

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*.

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

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An Interactive Method for Inducing Operator Descriptions. In *The Sixth International Conference on Artificial Intelligence Planning Systems*.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.