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

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

Abstract

In this paper we explore a principled, integrated approach to the process of creating complex planning applications and introduce and evaluate a new hybrid task-reduction planner called *HyHTN*. In the short term our work is leading to an experimental research platform for investigating the synergy of integrated tools. The approach is centred around the use of a GUI called GIPO-II (based on the previously released GIPO GUI). The major innovation in GIPO-II is the ability create and maintain ' hierarchical" domain specifications, and verify them using a structural property checker, and plan using the fast forward hybrid task-reduction planner *HyHTN*.

Keywords: knowledge engineering techniques for planning and scheduling, planning with hierarchical task networks

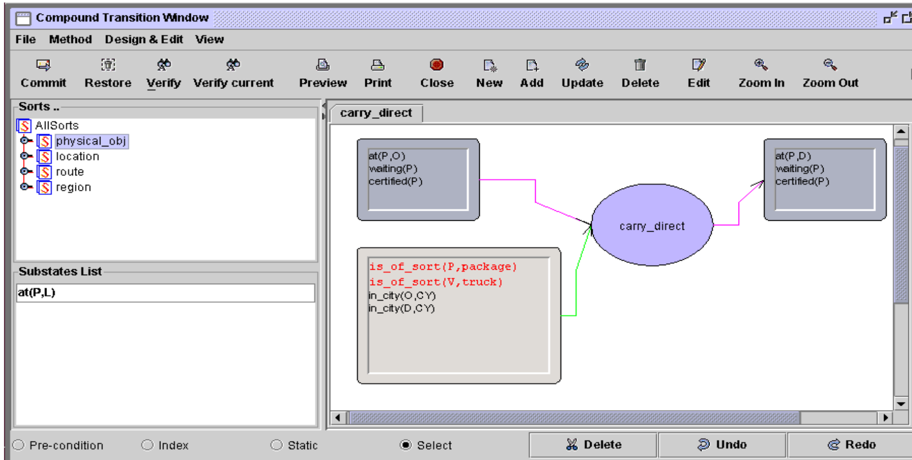
Introduction

In this paper we describe work in progress concerning an experimental environment for engineering and prototyping HTN planning applications. In contrast to operational planners which are aimed at solving real problems and applications (such as in the O-Plan development (Currie & Tate 1991)), 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 that 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. In this paper we additionally describe GIPO-II's default planner, *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). Our experimental results suggest that, in the experimental domains used, *HyHTN* is at least an order of magnitude more efficient than the EMS (McCluskey 2000) planner, and at least as fast as the state-of-the-art heuristic planner SHOP. In both cases we used the testing scenarios supplied with these planners. *HyHTN*'s code, its tests and test results, can be downloaded from http://scom.hud.ac.uk/planform/gipo. GIPO-II is planned for release in December 2002 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* (Liu & McCluskey 2000), it is based on the idea of engineering a planning domain so that *the universe of potential states of objects are 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 appli-



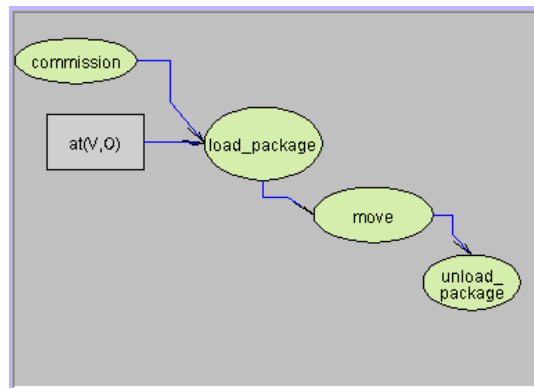


Figure 1: Method Transition Editor (top) with Decomposition Representation (bottom)

cation 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. Estlin et al forcefully argue this point in reference (Estlin, Chien, & Wang 2001). GIPO-II helps the user to specify these changeable states at each level in a similar fashion to the 'flat' version of GIPO (Simpson *et al.* 2001). After inputing an initial specification of states as described above, 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 ob-

ject must be in a certain (set of) state(s) and goes through a transition 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: every fact *not* attached to the levels of the object referred to in the LHS is assumed to persist. Otherwise the RHS must specify the new value of the fact. 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*. The latter are transitions of type 2 and 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 object's states by its decomposition and execution, and the final states of objects may depend on which decomposition is chosen (eg 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. In GIPO-II the transitions and static constraints are assembled in the transition editor and the decomposition is assembled to produce a graph as shown in Figure 1.

The Transparency Property

Hierarchical domain models in OCL_h are regulated by the semantic property of transparency – this ensures the methods are structured in a coherent manner This property should be true for every method in a model. The technical details of this are given elsewhere (McCluskey & Kitchin 1998). Key to the property is the idea that the method's decomposition into a task network will necessarily achieve the method's post-conditions (the *RHS*s of the necessary transitions indexing the method) – if this is the case, the method is called *sound*. The **transparency property** is then as follows: A method m is transparent if it and every expansion of m, consistent with its static constraints, is sound.

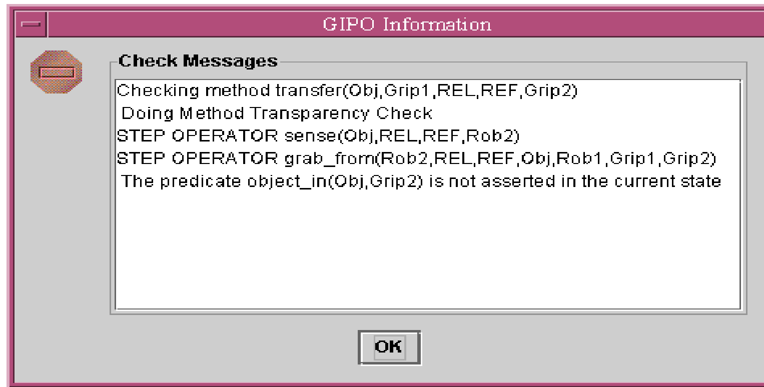
To check that the example method *carry-direct* is transparent, we first check that its decomposition necessarily achieves the post-condition - ie that the conditions

$$[at(P,D), waiting(P), certified(P)]$$

are met. This is done by examining each of the nodes in the task network and proving that the conditions are necessarily achieved, using the post-conditions of the nodes as achievers. An 'achieve-goal(G)' is treated as a transition with an unspecified *LHS*, where the post-condition (*RHS*) is G. After this is proved, we must study the consistent decompositions of the task network and re-check this property for each such decomposition. In GIPO-II, all these checks are performed by the *transparency* modelling tool which amounts to an automated verification test for each method.

Figure 2 shows the use of the transparency property checker in the "Drumstore World" (Aylett & Doniat 2002). This is a model based on a real world domain of robots handling and passing radio-active drums to each other. This scenario shows the finding of a bug during the actual coding of the HTN methods via GIPO II. The transparency checker is applied here to an HTN operator called 'transfer' whose necessary transitions and static conditions are shown in Figure 3. 'transfer' contains a decomposition for transferring an object (Obj) from one robots gripper (Grip1) to another (Grip2) in some relation (REL) to position REF (REL can be 'at' or 'near' in the current domain model).

The top diagram of Figure 2 shows the checker being called. It checks that the preconditions of the decom-



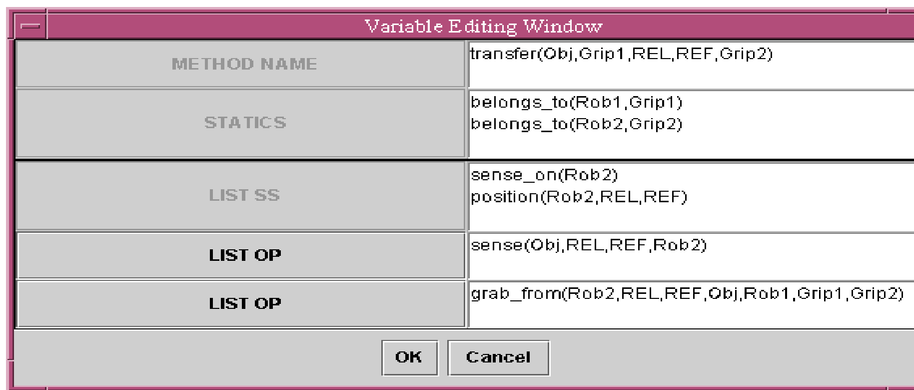


Figure 2: Steps Showing the Application of the Transparency Tool

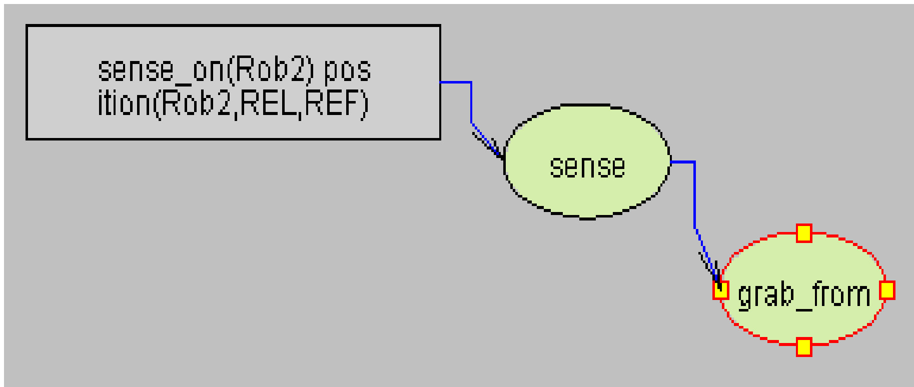
position are necessarily achieved, before checking that the post-conditions of the transfer method are met. The box shows that this is true for operator 'sense' but when it examines method 'grab_from', it detects that predicate 'object_in' has not been achieved. 'grab_from' is a primitive operator which simulates the exchange of an object from one robot to another, when the robots are next to each other.

The bottom diagram (Variable Editing Window) shows the construction window for unifying the variables within the decomposition of a method - this is essential in order to 'link up' all the components of the decomposition. It lists the decomposition of 'transfer' which starts with an achieve-goal

$$sense_on(Rob2) \& position(Rob2, REL, REF)$$

The variable parameters of achieve-goal, and 'sense' i.e. REL, REF, Rob2, Obj, have been unified together and with corresponding ones in 'grab_from' which indicates that they will instantiate to the same object. Unfortunately this process is not straightforward, and in this case the checker uncovers the error to do with the object and gripper in 'grab_from'. The user reverts to the graphical window (bottom diagram), clicks on 'grab_from' and proceeds with an inspection of the transition identified by the object (Obj) and gripper (Grip2). This shows that the parameters Grip1 and Grip2 have been entered in the wrong order in the Variable Editing Window, inhibiting the 'object_in' predicate from being achieved. With this error removed the checker proceeds successfully. This example shows the uncovering of an error in the unification of variable parameters - in fact the checker can find other modes of error such as missing parts of a decomposition.

GIPO II includes other static tools present in the GIPO release - these check 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.



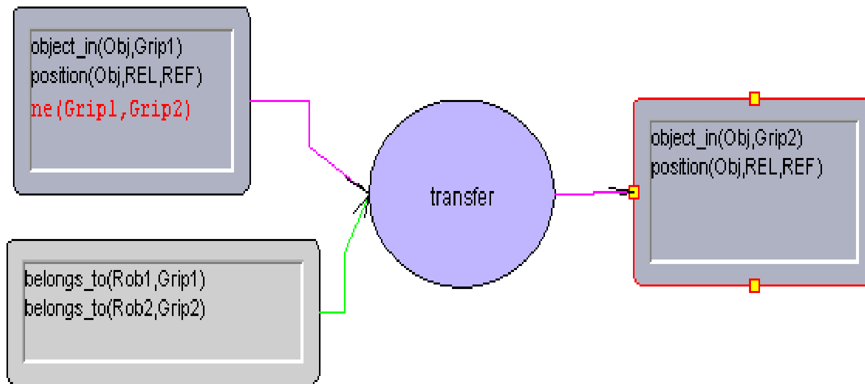


Figure 3: The 'transfer' Method (bottom) and its decomposition (top)

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, and is particularly useful when importing domain models.

Tasks Specification and Animation

A **task specification** in OCL_h has three components: (i) a task network, where a task is the name of a method *or* a condition on an object to be achieved (ii) an initial world state (iii) a set of constraints on the task network. The first component is thus similar to that used in Estlin et al's integrated planner framework (Estlin, Chien, & Wang 2001). Their 'activity-goals' are the same as method tasks, and their 'state-goals' are like our 'achieve goals'. An example task comprising a task network containing two achieve goals and a method, is as follows:

```
( [ achieve(ss(traincar,traincar1,
               [at(traincar1,city1-ts1)])),
    transport(pk-5-z,city3-cl1-z,city2-cl1),
    achieve(ss(package,pk-5,
               [at(pk-5,X),delivered(pk-5)] )) ],
  [before(1,3)],
  [serves(X,city3-x)])
```

Use of parameter X means the third node in the network can be paraphrased as 'deliver pk-5 to any destination X where X is a town centre location serving city3-x'. Tasks which are solved by a planner connected to GIPO-II are animated as shown in Figure 4. This shows a graph representing the decomposition of methods and solutions to achieve-goals, with the primitive ground operators forming a solution occurring at the leaves. As with GIPO, 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.

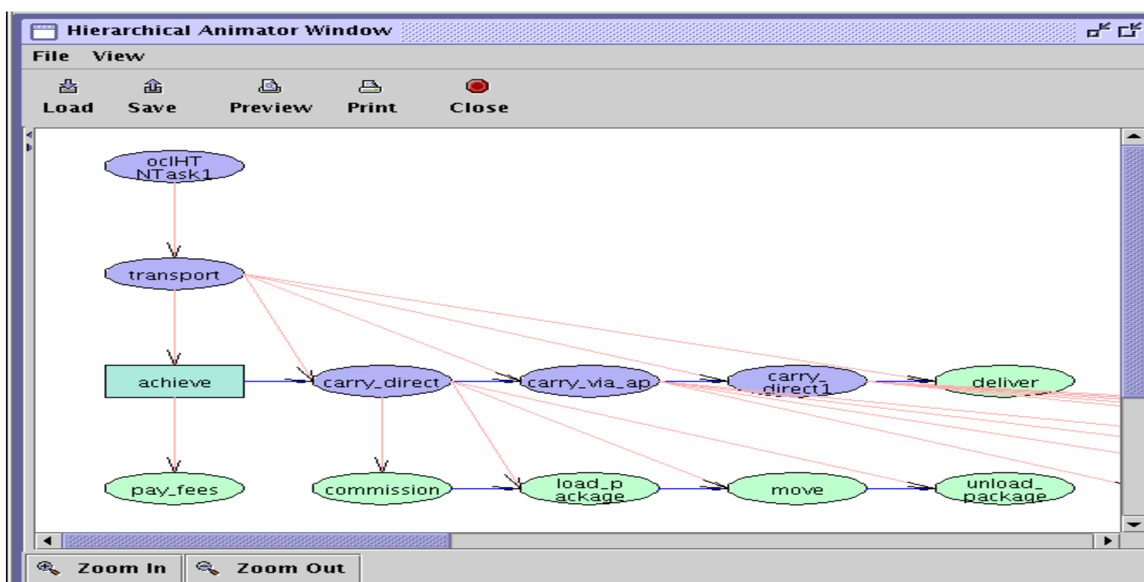


Figure 4: Partial Animation of an HTN Planner's Output

Dynamic Testing: The *HyHTN* Planner

HTN planning domain models have been used over the years to represent detailed domain-dependent heuristic information, but within a language that (up to a point) has a semantics independent of the target planning engine. An example of such pragmatic features is the **condition types** of O-Plan where conditions on the use of an action can have a range of modalities (Currie & Tate 1991). To a large extent an HTN planner is only as efficient as its heuristics encoded in the domain model. For example, if the domain model's compound tasks contained no temporal constraints on its decomposition, then efficiency is likely to be adversely affected. Or if the methods are not available which establish goal conditions then (in the absence of pre-condition planning) the task specification will fail. Hence, the completeness of the planning application is as dependent on the completeness of the method set as it is on the completeness of the planner's search strategy. Indeed, Kambampati uses the term 'operator completeness' specifically for HTN domain models, in contrast to the usual 'planner completeness' for algorithms (Kambhampati, Mali, & Srivastava 1998).

Over the past few years so-called *Hybrid* Planners (Kambhampati, Mali, & Srivastava 1998; et al. 2002) have been developed to reason with languages allowing the combination of both task specifications *and* goal conditions within problem specifications and method decompositions. Whereas HTN planners predominantly work through task expansion and constraint satisfaction, classical planners work through goal achievement. Hybrid planners are those that work with a combination of refinement strategies. Hybrid planners span the two extreme situations; at the pre-condition planning end, there is the specification of domain dynamics (and nothing else), and at the other, HTN methods that are essentially domain specific solutions.

Within a knowledge engineering environment it is important that a planning system used for dynamic testing should be able to cope with both these extremes, for the benefit of flexibility and so that operator-completeness is not an issue. *HyHTN* is a planner which has achieved a high level of performance, yet it inputs an expressive task language and can act as an HTN and/or a pre-condition planner. The power of *HyHTN* lies in two key features:

- it is a hybrid planner that combines two efficient techniques - it uses a state-advancing HTN reduction strategy for hierarchical refinements, and a fast forward search for 'achieve-goal' (pre-condition) achievement.
- it is fed with a statically-checked domain model whose methods are encased in pre- and post conditions and engineered a priori via GIPO-II to conform to the transparency property.

A particular advantage that we have exploited with a forward state advancing HTN planner is that *heuristic state-space search can be used to establish 'achieve-goal' conditions.* Thus the performance of SHOP-like algorithms in HTN planning, and the performance of FF-like algorithms in pre-condition planning can be combined into a flexible, efficient hybrid system. Secondly, the transparency property reduces the possibility of choosing methods that lead to dead - ends, as every decomposition that satisfies its static constraints is guaranteed to achieve its post-conditions.

Figure 5 shows the top level of the algorithm, which is called recursively as methods are expanded in a depth-

first, forward manner. *HyHTN* is nondeterministic at step 5: potentially, any of the first nodes of *Tasks* can be picked. If any of the branches fail then an alternative choice can be made.

Procedure 'method-expand' (shown in outline) selects an expansion of method T (T is a name and a list of parameters). In line 1 the full method is represented with a set of pre-conditions (*Pre*) which are essentially the *LHS* of transitions. Note that the *Post* is not used in the expansion as we assume that the transparency property assures us that the decomposition will be achieved.

Procedure 'apply' applies a sequence of primitive, ground operators to a ground state. Objects in the state are pushed through all the necessary transitions specified by the operator. If the operator contains any conditional transitions then objects satisfying the *LHS* of these transitions change as specified.

Procedure 'achieve-expand' calls a pre-condition planning algorithm that utilises a fast-forward state space search similar to FF (Hoffmann 2000).

procedure *HyHTN*
In *InitialWorld*, *Decomposition*, *Temps*, *Statics*
Out *Soln*= list of primitive operators
Read access domain model
Procedure

```
1.    Tasks = Decomposition;
2.    Soln = [ ];
3.    World = InitialWorld;
4.    WHILE Tasks is not empty
5.      pick and remove T from Tasks
        where no other node in Tasks
        is necessarily before T according to Temps;
6.      IF T is a primitive operator
7.          World = apply(T, World);
8.          Soln = append(Soln, [T]);
10.       ELSE IF T is a method
11.           call method_expand(World,Statics,T,Soln')
12.           Soln = append(Soln, Soln');
13.           World = apply(Soln', World);
14.       ELSE IF T is an achieve goal
15.           call achieve_expand(World,Statics,T,Soln')
16.           Soln = append(Soln, Soln');
17.           World = apply(Soln', World);
18.       END IF
19.   END WHILE
20. End.
```

Procedure *method_expand*
In *World*,*Statics*,*T*
Out *Soln*= list of primitive operators
Read access domain model

```
1.    Pick method(N,Pre,Post,Statics',Temps,Decomp)
      where N and T unify under substitution t, Pre_t is
        satisfied in World, and
2.    constraints S = append(Statics', Statics)_t are consistent
3.    call HyHTN(World, Decomp, Temps, S, Soln)
4.    end.
5. End.
```

Figure 5: The Top Level Design of the *HyHTN* Algorithm

Experimental Results

To test *HyHTN*'s efficiency we have compared it with EMS (McCluskey 2000), an HTN planner that inputs OCL_h, and SHOP (Nau *et al.* 1999). SHOP is well known for its efficiency in HTN domains. EMS was built primarily to explore the idea of hierarchical sort abstraction. It searches through a space of task networks which are stored with explicit pre- and post conditions. Initially these are the pre- and post conditions of the method that initiated the task network, but as methods are expanded these 'guard' conditions are added to, as new objects are affected by less abstract methods. The effects of task networks (which may be as small as a primitive operator, or as large as a final solution) are encapsulated in these post-conditions.

***HyHTN* vs EMS** To compare these two planners we used the 80 tasks that were used to evaluate EMS's performance and ability to 'scale up' to domains with many objects (McCluskey 2000). Four transport logistics domains were used, with the simplest containing 31 objects and the most complex containing 145 objects. EMS and *HyHTN* both input OCL_h tasks and domain models, and both were run on the same 128MB Sun under Solaris for comparison purposes. Table 1 compares the planners stating for each of the four domain models, the average time to generate a solution, the average number of nodes searched, the average solution length, and the maximum length solution in the test set. The results suggests that *HyHTN* is 20 - 30 times more efficient than *EMS* in these applications. Average solution sizes are not dissimilar, indicating that the linearity and state advancing quality of *HyHTN* does not adversely affect solution size in this case.

***HyHTN* vs SHOP** As both planners are aimed at tackling structurally complex domains, we chose as a test the large Translog domain (Andrews *et al.* 1995). This domain contains object classes such as cities, regions, packages, trucks, trains, planes, cranes, ramps etc. As an indication of the size of this domain, the OCL_h version contains 34 parameterised methods and 58 parameterised primitive operator structures. The SHOP model is of a similar size. We used the original 100 tasks in the SHOP release (with some small changes because of the difference in structure between OCL_h and the *STRIPS* style SHOP structure). The specific problems concern the transport of up to 10 packages, with 5 connected cities, 15 locations, 15 cranes to maintain one crane at each location, and 11 trucks in one location in the initial state. The packages were of different types: bulky, liquid, granular, and mail. 100 random tasks in the same complexity range as SHOP were randomly generated for *HyHTN*[1]. The original location and destination location of each package was

[1]Identical tasks could not be run on each configuration because there is a distinction between what the systems call 'domain model' and what they call 'task'. Hence we used

Model	*AvTime*		*AvNode*		*AvSoln*		*MSoln*	
	HyHTN	*EMS*	*HyHTN*	*EMS*	*HyHTN*	*EMS*	*HyHTN*	*EMS*
$T1$	0.09	0.7	26.0	39.5	16.75	19.4	40	53
$T2$	0.61	15.4	47.9	200.0	25.60	24.8	58	56
$T3$	1.64	55.0	51.1	231.5	28.75	26.5	65	64
$T4$	2.95	88.0	52.3	247.0	29.55	27.2	66	62

Table 1: Comparing *HyHTN* and *EMS* with the Translog Models

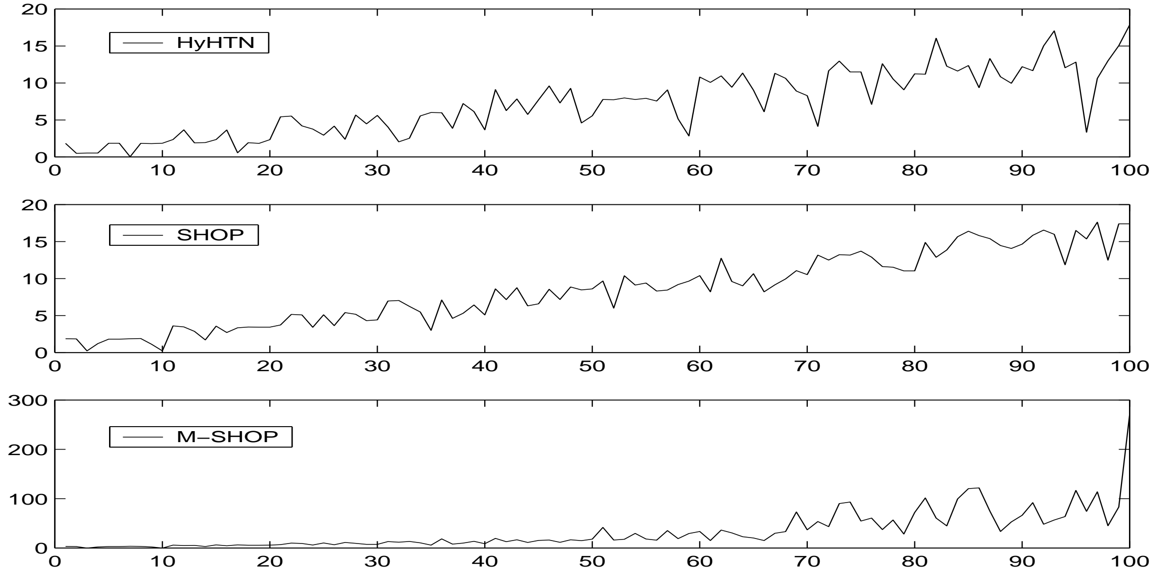


Figure 6: CPU times for *HyHTN*, SHOP and M-SHOP. The x-axis gives the problem number, and the y-axis displays the CPU time.

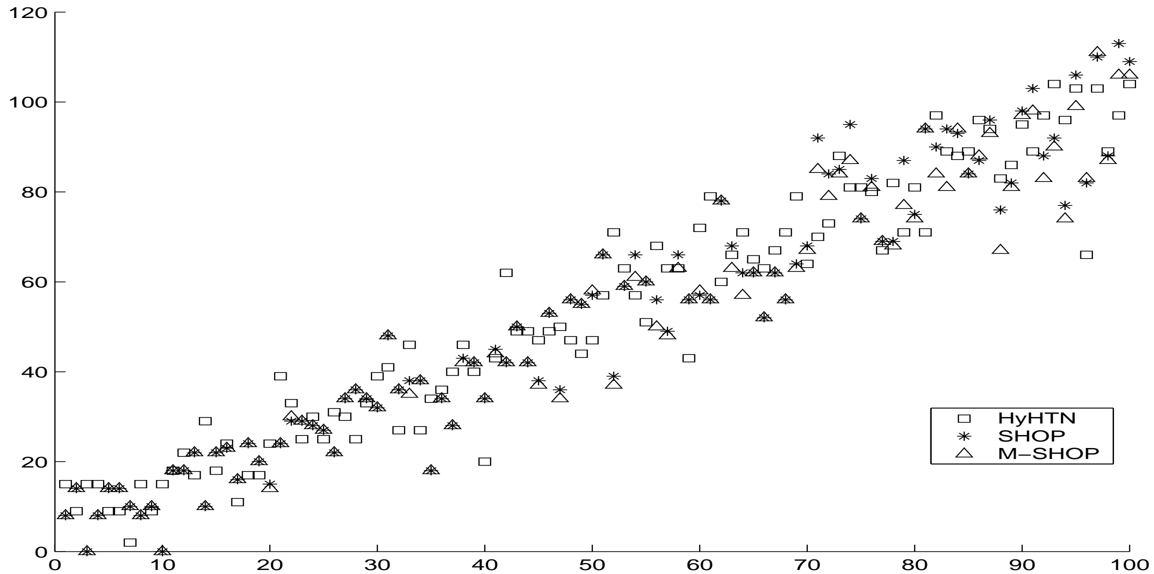


Figure 7: Number of actions in the plans found in Figure 6. The x-axis gives the problem number, and the y-axis displays the number of actions.

randomly chosen. All the tests were performed on the same 128MB Sun under Solaris, although SHOP runs under compiled Common Lisp whereas *HyHTN* runs under Sicstus Prolog.

Figure 6 plots the time in seconds against problem number for 3 configurations. $M - SHOP$ is a variation of *SHOP* which allows task decompositions to be interleaved. This means in general that its solutions should be shorter than the 'linear' planners *SHOP* and *HyHTN*, although the overhead in preventing achieved conditions from being deleted means that it is significantly slower. The results suggest that *SHOP* and *HyHTN* are similar in speed, although it is likely that a compiled Lisp implementation is much more efficient than Sicstus Prolog. Figure 7 suggests that, at least for this application, solution sizes for all the configurations are of a similar size.

Conclusions and Future Work

In this paper we have described an approach to engineer hierarchical planning knowledge, based on the construction of *transparent* hierarchical methods, and introduced the new GIPO-II tool which supports this. We have introduced a new hybrid planning algorithm *HyHTN* which can be used to prototype planning applications built using GIPO-II. It exploits state-expanding HTN planning to the full by integrating fast forward pre-condition search in order to establish achieve-goals. Finally, we have evaluated the implementation by comparing *HyHTN* against EMS and SHOP, both algorithms published at the AIPS-2000 conference. This has shown *HyHTN* to be at least an order of magnitude faster on the benchmark domains released with EMS; and at least as fast as SHOP on its Translog benchmarks. In both cases *HyHTN* was not noticeably poorer in terms of solution size.

Planners like SHOP have shown how the combination of a forward search and hierarchical planning can solve problems very efficiently. Using a search where the full world state is pushed forward has many advantages - not least that the basic representation can be very rich, predicates can be evaluated and multiple optimisation criteria can be used to help direct search. We intend to explore the scope for enriching the domain descriptions that *HyHTN* can deal with, and to work on theoretical conditions for its completeness. With the integration of this planner into the hierarchical version of GIPO we believe that we are approaching the point where we have a tool capable of dealing with practical planning problems, yet can be used as a extendible research platform.

References

Andrews, S.; Kettler, B.; Erol, K.; and Hendler, J. 1995. UM Translog: A Planning Domain for the Development and Benchmarking of Planning Systems. Technical Report CS-TR-3487, University of Maryland, Dept. of Computer Science.

Aylett, R. S., and Doniat, C. 2002. Supporting the Domain Expert in Planning Domain Construction . In *Proceedings of the AIPS'02 Workshop on Knowledge Engineering Tools and Techniques for AI Planning.*

Currie, K., and Tate, A. 1991. O-Plan: the open planning architecture. *Artificial Intelligence* 52:49 – 86.

Estlin, T.; Chien, S.; and Wang, X. 2001. Hierarchical Task Network and Operator-Based Planning: Two Complementary Approaches to Real-World Planning. *Experimental and Theoretical Artificial Intelligence* 13:379 – 395.

et al., E. A. 2002. Efficient use of hierarchical knowledge to improve the performance of a hybrid hierarchical planner. *The PLANET Newsletter* Issue No. 4:5–12.

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.*

Kambhampati, S.; Mali, A.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence.*

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., 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.*

McCluskey, T. L. 2000. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *The Fifth 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.*

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning.*

the SHOP test set and randomly generated new tasks for *HyHTN*, within the same complexity range