A Tool-Supported Approach to Engineering HTN Planning Models

T. L. McCluskey D. E. Kitchin
The School of Computing and Mathematics,
The University of Huddersfield,
Huddersfield, UK

Abstract

Our research concerns formal, expressive, object-centred languages and tools for use in engineering domains for planning applications. In this paper we extend our recent work on an object-centred language for encoding precondition planning domains to a language called OCL_h, designed for HTN planning. Domain encodings for HTN planners are particularly troublesome, because they tend to be used in knowledged-based applications requiring a great deal of 'domain engineering', and the abstract operators central to an HTN model do not share the fairly clear declarative semantics of concrete pre- and post condition operators. Central to our approach is the development, in parallel, of the abstract operator set and the hierarchical state specification of the objects that the operators manipulate. In this paper we define and illustrate a transparency property, together with a transparency checking tool, which helps the developer to encode a clear planning model in OCL_h. Our encoding of the Translog domain is used as an extended example to illustrate the approach.

1 Introduction

Past approaches to domain encoding for precondition planning and HTN planning have been centred around the development of operators. Overwhelming attention has been paid to them, the expressive nature of their constituents, and the kinds of hierarchy they may be placed in. Encoding complex domains in these operator-based languages is as influential and error prone as the coding of planning algorithms. While the algorithmic details of planners are of great theoretical and practical significance, the method of encoding domain representations has had relatively little attention. It is timely now, given the recent work to clarify and formalise hierarchical algorithms (e.g. [6], [8], [7]) to attempt to provide the first steps towards a systematic, tool-supported method of building models for HTN planning engines with clear planner-independent semantics, using a language and approach that supports good engineering principles such as design transparency, maintainability and proof obligation.

In [11] the foundation for an *object − centred* approach to domain encoding was laid, and exemplified with a language which was capable of encoding domains for precondition, classical planning. The advantages of this approach are that it emphasises the precise definition of a planning state as an amalgam of objects' 'substates', with operators constrained to be consistent with respect to the state; and it provides a natural form of abstraction by grouping objects together into *sorts*, which are sets of objects which share the same transitions through substates. The regularity of an object-centred approach provides the foundation for powerful pre-processing tools for use in two important areas: validation of the model with respect to the domain requirements, and compiling the model for speed-up in later plan generation.

In overview, HTN planners input a domain model which contains abstract solutions to problems, and constraints on how these are to be detailed. One must engineer a domain to a very refined state which includes a set of primitive operators as well as hierarchical operators. Also, the hierarchical structure tends to 'angle' the planner's application to solving certain abstract goals. It has been argued that hierarchical planners are the only type that have been applied to realistic problems, and that there is a need to better understand them [4]. Further, it has also been proved that hierarchical decomposition planners lead to more expressive languages of plan sequences [6]. Kambhampati [8] sums the advantages of HTN planning as giving the modeller more control over the types of solutions to be generated, and adding to plan generation efficiency by the provision of 'canned' plans. The encoding of tasks is an approach better suited to the capture of procedural knowledge than the primitive operators of precondition planners.

We believe that the major drawback with current 'open ended' languages used for HTN models is that they lead to opaque encodings: for example, it is not easy to give operators a clear semantics, as both the abstract and primitive variety are often context-dependent. Taking these operators out of their hierarchical context, and applying them to a valid state, it is not necessarily the case that a valid state (should such a concept be defined) is produced. In this paper we extend the method and language detailed in [11] to HTN planning models, and introduce new notation and a central new domain property, to support it. In sections 2 and 3 we describe the object-centred language OCL, a hierarchical version of that used in [12], and how it can be extended naturally to OCL_h to encode models for HTN planners. In section 4 we use the framework developed in section 3 to define what we call the 'model transparency property', a property which holds in a model if the main methods (the abstract operators) are structured in a coherent manner. We go on to outline a method for constructing hierarchical domain models based on the transparency property, which we have used to encode the 'Translog Domain', and give details of the tool developed to support the encoding of transparent models. In section 5 we evaluate the approach using this tool.

2 OCL

As a main thrust of this paper is to show that the benefits of using an object-centred language (OCL) extend to HTN planning applications, we will use the *Translog* domain to exemplify both the basic ideas here, as well as the more advanced ideas in later sections. Our version (OC-Translog[1]) is derived directly from Andrews et al's partial requirements description [14], and the encoding written for the UMCP planner itself. Translog was written as a benchmark for HTN planners, being an approximation of a realistic application (transport logistics planning). It has a rich structure involving the transportation of various kinds of packages (liquids, livestock, mail, valuables etc) between national locations (cities, airports, regions) via carriers (planes, trucks, trains) with varying characteristics.

A domain model written in an OCL contains definitions of objects, sorts, predicates, and has atomic invariants, substate class expressions, general invariants, and operators. The object-centred approach is rooted in the following ideas (the reader is referred to [11] for more details of the basic object-centred approach in planning).

2.1 Objects and sorts

Dynamic and static objects are specified in the model to reflect the requirements of a domain. All objects belong to a unique primitive sort, and all sorts are arranged in a strict 'sort hierarchy', with primitive sorts at the lowest levels. For example part of the hierarchy for *dynamic* primitive sorts (that is those sorts containing dynamic objects) in OC-Translog is:

$sorts(physical\_obj, [vehicle, package, crane, plane\_ramp])$
$sorts(vehicle, [land\_carrier, airplane, train])$
$sorts(land\_carrier, [mail, flatbed, regular, tanker, hopper, livestock, auto])$
$sorts(package, [valuable, hazardous, perishable, normal])$
$objects(valuable, [pkg\_1, pkg\_2, pkg\_3])$
$objects(tanker, [truck\_1, truck\_2])$

Example 1.

2.2 Predicates and substates

All dynamic objects in a world state have a 'local state', called a substate. A valuable package (pkg_1) might have its substate represented as the conjunction:

$$[at(pkg\_1, city1), uncertified(pkg\_1), is\_not\_insured(pkg\_1)]$$

Example 2.

If no instance of a predicate p has a changing truth value in the planning domain, we call that predicate *static*, and assume that the truth values of static predicates are described in the model. If a predicate is not static then it is dynamic.

The set of valid substates an object of a sort may occupy are specified in groups called 'substate class expressions' written in terms of predicates. Every binding of variables in a substate class expression that makes any included static predicates true gives a *valid* ground substate. To fit in with the sort hierarchy, we assume that a substate has n *hierarchical components* representing its primitive sort (s_1) and $n-1$ supersorts $s_2, ..., s_n$. A hierarchical substate class expression for an object of primitive sort s is therefore the conjunction :

$$h_1 \& h_2 \& ... \& h_n$$

where each h_j is one component of sort s_j's substate class expressions. We will use Example 3 to explain:

$substate\_classes(physical\_obj, [[at(O, L)]])$
$substate\_classes(vehicle, [[fuel\_level(V, F)]]).$
$substate\_classes(tanker, [[movable(T), available(T)],$
$\quad [movable(T), busy(T, P)],$
$\quad [valve\_open(T), busy(T, P)],$
$\quad [hose\_connected(T), busy(T, P)]])$
$substate\_classes(package, [[uncertified(P)],$
$\quad [certified(P)],$
$\quad [lifted\_up(P, C), certified(P)],$
$\quad [loaded(P, V), certified(P)],$
$\quad [delivered(P)],])$
$substate\_classes(valuable, [[is\_not\_insured(P)],$
$\quad [is\_insured(P)]])$

[1] Our full definition of OC-Translog can be downloaded from http://www.hud.ac.uk/schools/comp+maths/research/Artform/planning.html. Other planning domains specified in OCL, such as the R3 multi-robot, the Tyre, Rocket and Briefcase worlds are also available from this page, as are tools that support the development of OC models

Example 3.

The first line of the example states that a physical object (O) must have some location (L), and the second records that vehicles have a fuel level property. Any object of primitive sort tanker (a specialisation of physical object and vehicle by Example 1) inherits the location and fuel level substates, and additionally must be situated in one of four substate classes described by exactly one of the predicate sets. Hence tanker T is either busy with a package (P) or available for commissioning. Both a ground substate and a substate class expression are interpreted under a local closed world assumption, i.e. any predicates appearing in a class expression at the same level in the hierarchy, but not stated in a particular expression are assumed to be false in the range of substates that expression denotes.

Substate specification is a major part of the OC process: it forces the developer to think deeply about the 'life history' of the objects that the resultant plan will manipulate. Further, it forces the developer to precisely but implicitly specify all possible substates, giving more opportunity for integrity checking and model preprocessing to help in debugging, plan generation and execution. The number of hierarchical substate classes of an object is potentially the number of combinations of choosing a class from each level (although this is normally restricted using invariants as explained below). For example, a valuable package would potentially have have $1 * 5 * 2 = 10$ hierarchical classes.

2.3 Substate transitions

Objects undergo transitions as a consequence of actions, for example a package being insured might go through the following transition:

$$[at(pkg\_1, city1), uncertified(pkg\_1), is\_not\_insured(pkg\_1)] \Rightarrow$$
$$[at(pkg\_1, city1), uncertified(pkg\_1), insured(pkg\_1)]$$

Example 4.

We allow transitions to be *specified* so that (i) hierarchical components which persist do not need to be recorded (hence the first two predicates were redundant in the example above). (ii) all valid hierarchical components that match the LHS (left hand side) change to the unique substate given by the RHS (i.e. the specifications are many-to-one). For example

$$busy(tanker\_1, pkg_1) \Rightarrow movable(tanker\_1), available(tanker\_1)$$

Example 5.

specifies that the tanker moves from either of its three busy substates into one where it is movable and available, *and* its position and fuel levels do not change.

2.4 Operators

Operators encapsulate substate transitions that typical objects may go through. They have 3 components. The first contains prevail conditions which are predicates which must be true throughout operator execution. The second is a set of necessary state transition specifications: objects must be found whose substates match with their LHS for the operator to be applicable. The operator then changes an object which has a substate matching with *S*, to the unique substate *T*, for all transitions $S \Rightarrow T$ specified. Operators are interpreted so that all the changes they encapsulate occur in parallel. The third component contains conditional effects, and specifies the substate changes of objects if their substate happens to satisfy a certain condition. Example:

$$\begin{array}{l}
operator(pick_up_heavy_package(P, Crane, L), \\
\quad \%\texttt{prevail}: \\
\quad [[at(Crane, L), is_of_sort(Crane, crane)]], \\
\quad \%\texttt{necessary changes}: \\
\quad [[[at(P, L), certified(P)] \Rightarrow [lifted_up(P, Crane), certified(P)]], \\
\quad [[idle(Crane)] \Rightarrow [lifting(Crane, P)]]], \\
\quad \%\texttt{conditional changes}: \\
\quad [])
\end{array}$$

Example 6.

This example shows how an operator encapsulates the hierarchical substate changes of a crane and a package. Sequences of operators have a correspondence with transition sequences between substate classes as, for example, in OC-Translog, where T is a tanker, we might have:

$$[movable(V), available(V)] \Rightarrow [movable(T), busy(T, P)] \Rightarrow$$
$$[valve\_open(T), busy(T, P)] \Rightarrow [hose\_connected(V), busy(V, P)]$$

Example 7.

These transitions are brought about by the operator sequence 'commission(V,P)', (which means commission a vehicle for transporting a package), 'open_valve(V)', 'connect_hose(V)'.

2.5 Invariants

Invariants include atomic facts (giving the static structure to a model), rules and inconsistency constraints. Their purpose is to (a) document the assumptions of the modeller, and therefore tighten the model and help in model maintenance, and (b) be used in pre-processing aids to help debug the model, for example in checking operator consistency (c) be used in pre-processing aids to tease out useful but implicit information to speed-up online planning (d) be used in a limited way during plan generation. Examples of a rule and constraint for Translog are:

$$[loaded(P, V)] implies [at(V, L), at(P, L)]$$
$$inconsistent([certified(P), not\_insured(P)])$$

Example 8.

Both invariants constrain the set of objects' valid substates, the first rule specifies that a vehicle and a package must occupy the same location if that vehicle is loaded up with the package, while the second adds a constraint on the combinations of hierarchical substates of a package.

2.6 States and goals

A 'world state' is precisely defined as a mapping between every dynamic object and its current substate, which is made up of *hierarchical components* i.e. instantiated substate class expressions from each level of the object's sort's hierarchy. A *well-formed* world state is one in which, additionally, the conjunction of the predicates in the range of this mapping conforms to the model's invariants. A small part of a well-formed world state of Translog might be:

$$[pkg\_1 \rightarrow at(pkg\_1, city1), uncertified(pkg\_1), is\_not\_insured(pkg\_1)]$$
$$[truck\_2 \rightarrow at(truck\_2, city2\_ap1), fuel\_level(truck\_2, full), hose\_connected(truck\_2), busy(truck\_2, pkg\_2)]$$

Example 9.

A planning problem is viewed as one (or more) state transitions. For example, if the goal condition is $at(pkg\_1, city2), delivered(pkg\_1)$, then the problem is to change the current substate of pkg_1 to a substate which satisfies this goal condition.

3 Object-centred HTN planning

An **OCL model** consists of a specification of objects, sorts, predicates, invariants (atomic and non-atomic), substate classes, and operators. Previously OC languages have been used to encode and compile various planning models, and have been used as the input language for a range of precondition planners such as an object centred variant of UCPOP [9]. Realistic applications involving planning differ from those used for theoretical investigation in many ways, chiefly in the sheer amount of knowledge to be encoded. This manifests itself in the form of domain structure which we classify as 'static knowledge' for the purposes of plan generation. Many domain models in the planning literature exhibit complex goal structure but have little or no static knowledge at all (e.g. our OC encoding of the familiar Tyre world mentioned above has *no* static predicates). The structuring and economic encoding of knowledged-based application requires a hierarchy allowing certain state transitions (effected by several primitive operators) to be guarded by chunks of conditions. As well as using hierarchy for pragmatic reasons, realistic planning requires planning sequences as solutions which are not possible to obtain using precondition planning alone, as Erol et al point out in [6].

Much effort is invested in an hierarchical encoding of a domain, and heuristics for the use of the resulting model are embedded in the way the hierarchy is arranged to reflect the main kinds of tasks to be performed. To make sure that complex plans can be generated, a HTN domain encodes parametrised, constraint-laden solution fragments, and planning consists of assembling the fragments together, making sure all constraints are met. In this section we describe the extensions to OCL to handle the requirements outlined above, and call the resulting language OCL_h. In OCL_h we use two additional types of abstract, hierarchical operator: method operators and filter operators. As in Yang's formulation [16], and following in Erol's work, the effects of the hierarchical operators are clearly related to the primitive operators that result in expansions of the hierarchy, but, as we shall see, the object-centred framework allows us to go further.

3.1 Method operators

These are operators which, like primitive operators, specify dynamic object transitions. They are 'indexed' by one necessary substate change $S \Rightarrow T$, which specifies a transition of a typical object o [2] of sort s from a (set of) substate class(es) to a unique substate class. This may contain one or *more* hierarchical components of o's sort hierarchy - for example, if the object in question is a valuable package, part of the substate could be $[is\_insured(P)]$, but there could be other hierarchical components, such as $[delivered(P)]$ (from the substate class for package) and $[at(P, D)]$ (from the substate class for physical object - see Example 3). S must contain dynamic predicates from o's sort hierarchy only, but may contain static predicates which act as filters, determining under what structural conditions a transition can take place.

As well as a name and an index, methods contain a "body" comprising of a partial order of nodes. Many bodies may share the same name and index, but with differing static predicates, to indicate under what conditions the method operator should be used. For example, in OC-Translog there are several method operators for moving a package from one location to another with the name $carry(P, O, D)$ and an index $[at(P, O), certified(P)] \Rightarrow [atP, D), certified(P)]$, but the static predicates show which method is to be used under which conditions.

A node can be one of four types:
(a) the name of a method operator
(b) the name of a primitive operator

[2] Note that this does not restrict other objects being involved in the transition, but is a commitment to the *main object*.

(c) the name of a filter operator
(d) an expression of the form $achieve(G)$, where G is the partial or full description of the substate of a dynamic object.

When methods are expanded, they may change the substate of many other objects apart from o (ie. they may have side effects embedded in their expansions). For example, the most abstract method for Translog is to transport a package P to a location D from location O:

$$
\begin{aligned}
&\mathit{method}(\mathit{transport}(P, O, D),\\
&[\mathit{at}(P, O), \mathit{is\_of\_sort}(P, \mathit{package}), \mathit{not}(O = D)] \Rightarrow\\
&[\mathit{at}(P, D), \mathit{delivered}(P)],\\
&[\mathit{before}(1, 2), \mathit{before}(2, 3)],\\
&[1 : \mathit{achieve}([\mathit{certified}(P)]),\\
&2 : \mathit{carry}(P, O, D),\\
&3 : \mathit{deliver}(P, D)]).
\end{aligned}
$$

Example 10.

Although the necessary substate change concerns the package, the achievement of this will necessarily involve other objects, such as a vehicle, whose substates will also change. The body defines a way to achieve the goal substate (that is the RHS of the arrow, which contains two hierarchical components) from any substate matching the LHS.

3.2 Filter operators

Filter operators form part of the expansion of method operators. The purpose of filter operators is to bring about the right conditions for a substate transition that has been specified by a method operator at a higher level. To do this they contain static predicates, which act as filters, and an ordering of nodes, which may be of any of the four types listed above. (Note, this means a filter operator may be introduced into a plan as a result of an earlier filter operator, but we can always trace back to a method operator.) Unlike a method operator, a filter operator has no substate transition index. Whereas the purpose of a method operator is to specify a major substate transition for a particular object, the filter operator's purpose is to effect subordinate or associated transitions, required for the higher level operation. Static predicates, to preserve required conditions, must remain true throughout expansion of a method operator, and, consequently, any filter operators.

A filter operator for moving a traincar by train from location O to location D is as follows:

$$
\begin{aligned}
&\mathit{filter}(\mathit{move\_vehicle}(V, O, D, \mathit{Train}),\\
&[\mathit{traincar}(V), \mathit{connects}(R1, K, O), \mathit{connects}(R2, O, D),\\
&\mathit{is\_of\_sort}(R1, \mathit{rail\_route}), \mathit{is\_of\_sort}(R2, \mathit{rail\_route}),\\
&\mathit{is\_of\_sort}(\mathit{Train}, \mathit{train})],\\
&[\mathit{before}(1, 2), \mathit{before}(2, 3), \mathit{before}(3, 4), \mathit{before}(4, 5)],\\
&[1 : \mathit{commission}(\mathit{Train}, P),\\
&2 : \mathit{move\_vehicle}(\mathit{Train}, K, O, R1),\\
&3 : \mathit{attach\_traincar}(\mathit{Train}, V, O),\\
&4 : \mathit{move\_vehicle}(\mathit{Train}, O, D, R2),\\
&5 : \mathit{detach\_traincar}(\mathit{Train}, V, D)])
\end{aligned}
$$

Example 11.

This operator would occur as one of the nodes in the developing plan of a higher level filter operator, carry_direct(P,O,D), which would itself occur as one of the nodes of a method operator's expansion such as carry(P,O,D).

In what follows we will refer to the name, index, constraints and nodes components (where appropriate) of any primitive, filter or method operator m using the 'dot' notation - e.g. if m is the method in Example 10, then

```
m.name = transport(P,O,D), and
m.constraints = {is_of_sort(P,package), not(O=D),
                 before(1,2),before(2,3)}
m.nodes = {achieve([certified(P)]), carry(P,O,D),
           deliver(P,D)}
m.index = at(P,O) => (at(P,D), delivered(P))
```

4 Transparent HTN models in OCL_h

A general integrity rule for OC primitive models (i.e. models containing primitive operators only), called the *operator complete* property, was defined in [11]. Roughly, a model is operator complete if it has well defined object sorts and substate class expressions, and if every operator is consistent, i.e. it can be shown that the execution of any applicable operator on a well-formed state always results in a well-formed state. It is used as part of the validation process, and forms part of a systematic method for creating precondition planning domain models.

Extra problems arise when creating HTN models to do with their opacity: while pre- and post conditions of primitive operators are supposed to be self-contained, abstract operators are necessarily not so. The problem of hierarchical interference has been noticed by many authors. For example, Fox explains it as the problem of relating the effects asserted in the abstract operator with those of the primitive operators that implement it [7]. One has to ensure that the decomposition, under any condition, achieves the effects of the abstract operators. This is not particularly easy when the form of the effects are unrestricted or do not appear in a regular way. The object-centred framework allows us to define properties to help alleviate this problem; to define these properties we will first need to explain the intended procedural semantics of network expansion with OCL_h.

4.1 Object transitions and transition sequences

The space of valid transitions involving one object is defined as the set of all pairs of ground substate class expressions belonging to the object's sort. In [11] an algorithm was defined which generated *macros* which spanned the space of substate transitions, and each macro was *indexed* by the substate transition that the macro sequence achieved. In HTN planning, however, abstract operators are written to span a small proportion of this potential task space, but the tasks captured are normally very complex and correspond to the most important substate transitions (such as transporting an object from one place to another in OC-Translog).

Let $n > 0$, and i range from 1 to n. A transition sequence for object o of primitive sort s is a sequence:

$$[I\ ;\ \ I_1 \Rightarrow I'_1\ ;\ \ I_2 \Rightarrow I'_2\ ;\ \ \ldots\ ;\ \ I'_n \Rightarrow I_n\ ;\ \ F]$$

such that $I_i \Rightarrow I'_i$ are specifications of substate transitions of sort s, and I and F are (possibly partial) specifications of substate class expressions. A transition sequence is *linearly sound* if each I_i and F are necessarily achieved, using the I'_i and I as 'effects'. Example 12 shows a linearly sound transition sequence for object 'pkg_1' of sort package, where 'City_1' and 'City_2' are variables of sort *city*. This soundness idea is familiar in planning, for example our definition can be viewed as an abstracted form of Knoblock's 'justified sequence of operators' definition [10].

$$\begin{aligned}&[[at(pkg\_1, City\_1), uncertified(pkg\_1), is\_not\_insured(pkg\_1),\\&\quad not(City\_1 = City\_2)];\\&\quad [is\_not\_insured(pkg\_1)] \Rightarrow [is\_insured(pkg\_1)];\\&\quad [uncertified(pkg\_1)] \Rightarrow [certified(pkg\_1)];\\&\quad [at(pkg\_1, City\_1)] \Rightarrow [at(pkg\_1, City\_2)];\\&\quad [at(pkg\_1, City\_2), certified(pkg\_1)]]\end{aligned}$$

Example 12.

4.2 Expanding method operators into networks

A method m forms a partial plan network $n = net(m.name, m.index, m.constraints \cup \{b\}, m.nodes)$ when used to achieve a goal which matches the RHS of the method's index, under binding b. We *reduce* n to network n' when any method or filter node in n is replaced by the nodes of an operator op of the same name, or a node of type $achieve(G)$ is replaced by the name of a primitive or the nodes in a method operator which necessarily achieves a substate satisfying G. The replacement can only take place if $op.constraints$ are consistent with respect to the OCL's model's invariants. For example $n = net(name, index, constraints, nodes)$ reduces to $n' = net(name, index, constraints \cup op.constraints, (nodes - \{o\}) \cup op.nodes)$.

Thus for the transport method operator shown above and in Example 10 its first reduction might involve the replacement of $achieve([certified(P)])$ with one of the primitive $pay\_fees(P)$ operators, where its static predicates are true, according to the type of package being delivered. For example, where the method was being used to transport a hazardous package, then an instance of this operator would be required:

```
operator( pay_fees(P),
   [ [has_permit(P), is_of_sort(P,hazard_p)] ],
   [[ [uncertified(P)],
   [waiting(P),certified(P)] ]],
   [ ]).
```

Example 13.

A network n' is an *expansion* of n if n' was generated from n using one or more sequential reductions of the nodes in n. n' is a *normal* expansion of n if n' contains *no* filter operators. n' is a *primitive* expansion of n if n''s nodes are all primitive operators.

Any network can be expanded to a normal expansion, since if there is a name of a filter operator in the network, we can find an appropriate name's body and reduce, until no filter operator remains (this process must terminate as non-primitive operators are rooted in primitive ones).

All nodes in a network n, except for those naming filter operators, can be *labelled* by transitions. A method operator is labelled by its index "$S \Rightarrow T$", an $achieve(G)$ node is labelled "$\Rightarrow G$", and a primitive operator is labelled by all the necessary or conditional transitions it specifies. Finally, we say that a network n is *sort-abstracted* with respect to s, if we ignore all transitions in the labels which do not refer to a transition of an object of sort s, and call the resulting network n_s.

4.3 Model properties

Assume a method m is indexed by a transition of an object of sort s.

Soundness: m is sound if, for every sort-abstracted primitive expansion n_s of m, every legal linear ordering of the nodes' labels in n_s is linearly sound.

Here *legal* linear ordering means one that conforms to the node's temporal constraints. The soundness property captures the intuition that every solution admitted by a method which is labelled by $S \Rightarrow T$ specifying a transition of an object o actually does achieve T when its primitive operators are executed and operate on o. This property, however, does not concern intermediate levels in a network's expansion. The following property is more useful:

Transparency: m is transparent if, for every sort-abstracted, normal expansion n_s of m, every legal linear ordering of the nodes' labels in n_s is linearly sound.

A model in OCL_h is sound (transparent) if all its method operators are sound (transparent). It follows immediately from the definitions that if m is transparent it is also sound, since any primitive expansion of m is also a

Algorithm: Expand Network for Transparency Property Checking
In: m : method operator
Out: error report (*or* property certification if no errors)
Global Data: *Store*: set of networks, *Model* : an OCL_h model

1. $initialise\_Store(make\_method\_into\_network(m))$;
2. while non_empty_Store do
3. $\quad remove\_from\_Store(n)$;
4. $\quad$ if $no\_filter\_ops\_in(n)$ then
5. $\quad\quad n' := sort\_abstract(n)$;
6. $\quad\quad check\_transparency(n')$;
7. $\quad\quad$ if $not\_all\_primitive\_ops\_in(n)$ then
8. $\quad\quad\quad expand\_all\_nodes\_in(n)$
9. $\quad\quad$ end if
10. $\quad$ else
11. $\quad\quad expand\_all\_filter\_nodes\_in(n)$
12. $\quad$ end if
13. end while
14. end.

procedure $expand\_all\_nodes\_in(net(name, index, constraints, nodes))$
3.1 for all $n \in nodes$ do
3.2 $\quad$ if $is\_a\_method\_op(n) \vee is\_a\_filter\_op(n)$ then
3.3 $\quad\quad S := \{op : op \in Model\text{'s operators} \wedge op.name = n\}$
3.4 $\quad\quad$ for each $op \in S$ do
3.5 $\quad\quad\quad constraints' := constraints + op.constraints$;
3.6 $\quad\quad\quad nodes' := op.nodes \cup (nodes - n))$;
3.7 $\quad\quad\quad$ if $consistent(constraints', Model\text{'s invariants})$ then
3.8 $\quad\quad\quad\quad add\_Store(net(name, index, constraints', nodes'))$;
3.9 $\quad\quad$ end for
3.10 $\quad$ end if
3.12 end for
end procedure.

procedure $check\_transparency(net(name, index, constraints, nodes))$
5.1 $L := \{l : l$ is a legal linear ordering of $nodes\}$;
5.2 for all $l \in L$ do
5.3 $\quad$ if $\neg$ $linearly\_sound(l)$ then
5.4 $\quad\quad$ print error report
5.5 $\quad$ end if
5.6 end for
end procedure.

Figure 1. Outline Algorithm for Network Expansion and Transparency Checking

normal expansion. The transparency property is a useful constraint to follow when engineering a model, and can be operationalised into a tool as shown below.

We have used the transparency property in particular in the development of OC-Translog, and here we show how it verifies the main 'transport' method (see Example 10) which is indexed by:

```
[at(P,O)] => [at(P,D), delivered(P)]
```

A normal expansion of the method operator *transport*, in Example 10, would reduce 1. by replacing it with one of the *pay_fees* operators, as describe above, thus achieving $[certified(P)]$. The *carry* method in 2. has an index $[at(P,O), certified(P)] \Rightarrow [at(P,D), certified(P)]$ and the reduction of 3 by the primitive *deliver* operator achieves $[delivered(P)]$.

Hence we see the expansion of 1, 2 and 3 gives this transition sequence:

$[[at(P,O)]$;
$[] \Rightarrow [certified(P)]$; `%achieve goal`
$[at(P,O), certified(P)] => [at(P,D), certified(P)]$; `%carry method`
$[certified(P)] => [delivered(P)]$; `%deliver method`
$[at(P,D), delivered(P)]]$

Example 14.

This is clearly a sound transition sequence, as the LHS of the arrows and the final substate are necessarily achieved by expressions earlier in the sequence.

4.4 The tool supported approach to engineering HTN models

Our object-centred approach to designing domain models for input to precondition planning (as detailed in [12]) has been extended to HTN models as described, with a similar range of tools which support consistency and cross checking, as well as the implementation of a tool which helps check transparency. Models are engineered in two parts, firstly in a bottom-up fashion to construct the dynamic sort hierarchy. This involves identifying the primitive sorts, by trying to collect all objects which go through identical state changes into groups. Substate classes are thus defined, by writing a description of each possible state of a typical object of each sort. Any state features which are shared between primitive sorts (e.g. a tanker truck and a flatbed truck share a fuel level feature) can be factored out and used to specify substates at a more abstract level in the hierarchy.

In parallel to this is a top-down path, where the requirements of the domain are used to identify the main abstract tasks required (e.g. transport a package). The tasks are represented as methods specifying transitions between abstract substates, or as filter operators when the kinds of solutions to the abstract tasks need to be constrained to contain orderings of operators under certain conditions.

The transparency property is implemented as part of our tool support according to the outline algorithm in figure 1. The main algorithm (lines 1-14) outlines the way networks are expanded exhaustively for testing. Lines 3.1 - 3.12 outline the procedure for expanding any node (the definition of *expand_all_filter_nodes_in* is similar). A new network is added to the Store (line 3.8) only if the constraints in that network are self-consistent and with respect to the invariants given in the model (checked in line 3.7). Lines 5.1 - 5.6 outline the procedure for transparency checking a sort-abstracted network. The error report indicates the transition(s) that cannot be achieved in the linear sequence. Example 15 is actual output from the transparency tool - when changing the definition of substates for our OC-Translog model the tool was able to help us find that the 'waiting' predicate which we had recently added leads to an unsound linear sequence as shown:

```
[at(_457,_458)] =>
----achieve([certified(_457)])----
      => [certified(_457)]
----carry(_457,_458,_459)----
[at(_457,_458),certified(_457)] =>
[at(_457,_459),certified(_457)]
----deliver(_457,_459)----
[waiting(_457),certified(_457)] =>
   [delivered(_457)]  =>
       [at(_457,_459),delivered(_457)]
```

Example 15.

5 Discussion

5.1 Consequences of using the hierarchical OC approach

Properties such as transparency resemble *proof obligations*, which are carried out in engineering to obtain a clearer understanding of a model, detect bugs in a model, and improve confidence in the validity of the model. Note that the transparency property is a way of implicitly checking filter operators also, as each one must have at least one method operator as an ancestor in an expansion.

The Translog domain was encoded into OCL_h following the guidelines set out in the paper. It contains 40 sorts, (over half are primitive sorts), 43 substate class expressions in 15 dynamic sorts; it has 30 dynamic predicates, 80 instances of static predicates, 8 method operators, 16 filter operators and 45 primitive operators. The encoding of OC-Translog suggests that OCL_h does not overly restrict expression. The benefits of encoding a domain into OCL_h are summarised:

1. The approach gives a rationale for the use of effects in abstract operators i.e. effects are used in a method operator to specify the substate transition that it achieves; no explicit effects are allowed in filter operators. This restriction alone imposes a strong regularity on the model, and eliminates the 'sloppy' use of effects.

2. Method operators specify complex state transitions of objects, and must be written in such a way that any expansion path leads to a sound transition sequence. The transparency property gives the modeller static, planner-independent checks to analyse the effectiveness of method operators, and we believe that using the property as a heuristic for domain construction reduces the possible causes of interference between methods for different sorts.

3. Constraints on the persistence of facts (e.g. predicate *p* must remain true throughout the sequence of nodes) are represented not by meta-predicates but explicitly in the substate. For example, in the UM Translog encoding an airplane ramp may be available or not, and a metapredicate 'between' is used to stop the ramp being removed while the loading procedure is in progress. In the OC model the substate classes of the primitive sort[3] are 'available', 'needed & unavailable', and 'unavailable'. The 'needed' predicate records the causal link between the loading procedure and the ramp: a transition of the ramp substate from the 'needed & unavailable' substate can only take place as a side effect of the ending of the loading procedure.

Problems occurred, however, in the encoding of the Translog dynamic sort hierarchy. The requirements on vehicle subsorts led us to create a 'land_carrier' supersort merging traincars and trucks, as each had the same kinds of shapes (tankers, hoppers and so on). This meant that an extra static predicate had to be defined to distinguish between trucks and traincars. A further problem to do with OCL_h not allowing multiple inheritance is more serious. We stipulate that every object or sort has a unique inheritance path. For the sake of keeping a simple semantics, this means that some parts of an encoding can be awkward. One particular case was the encoding of 'special subtypes' of vehicles. Flatbed, regular and tanker trucks and traincars may be equipped to carry valuable, hazardous or refrigerated packages, and each combination (e.g. a 'hazardous package-carrying tanker') is possible. Rather than listing the substate classes of all the possibilities, we overcame this problem by allowing *primitive* sorts to have *subsorts*, effectively meaning that objects of a *primitive* sort could have variant substate classes. In the event very few subsorts were used, but the consequence of this extension is that one has to define the substate classes for each subsort in the same way as for the primitive sort and the rest of its sort hierarchy. An alternative solution would be to allow multiple inheritance, letting a land carrier, for example, in-

[3]The ramp also inherits a location as it is a physical object

herit state from a hazardous vehicle supersort, and from its flatbed sort.

5.2 Transparency of macros

It is interesting to compare an OCL_h domain model with a primitive model augmented with automatically generated macros specified in section 4 of [11]. Each macro is indexed by a substate transition $S \Rightarrow T$, and implements the transition of an object o of sort s. Macros are generated in a sort-abstracted domain model which considers only predicates from one sort's substate classes, and their bodies are sequences of primitive operators. Thus a sequence in a macro's body forms a sound transition sequence affecting o, the object of interest in the index. Hence a macro is in fact a simple kind of method operator which expands in one reduction to a sequence of nodes representing primitive operators. Further, as the macro-generation technique is sound, a model augmented with these kind of macros will always satisfy the transparency property.

5.3 Related work

Given the additional complexity of a HTN domain model, there is surprisingly little attention given to the topic of domain modelling in planning literature. A notable exception is Chien's work on the development and verification of the VICAR system [3]. Our approach is also similar in spirit to the combination of a planning technique with a KADS model of expertise, as proposed in [2]. A KADS model of expertise describes the problem solving method, linking together a planning technique and a structured KADS approach to knowledge acquisition and representation.

The domain modelling problem is acknowledged by [15], who are currently working on the further development of Task Formalism (TF), which currently seems only to offer a guidance framework, rather than a systematic method, and which has no tool-support. Similarly, within his work on the formalization of HTN planning, [5], Erol touches on the subject of how to encode domains for HTN planners. He outlines some simple steps in writing a domain specification, but gives no detailed method.

Some work on the formalisation of HTN planners has been done, for example by Kambhampati [8], which nevertheless has a bearing on the kind of domain models encoded. He points out that if we have a strict hierarchy among tasks we can use this to 'turn a filter condition from a non-monotic auxiliary filter into a monotonic one'. Once such a constraint is violated by the partial plan, the plan cannot be further refined to restore the constraint, and thus the plan can be removed from the search space. From this we can derive domain modelling heuristics. This is analogous to the method transparency property used with OCL_h, which holds if the main methods do not interfere with each other.

Previously, an object-centred model of action has been defined by [1], which is similar to our own in that it models the world in terms of classes of objects and their state transitions, and identifies a world state as a mapping from instances of objects to their states, as given in the 'state graph'. The authors stress the *computational* benefits of using such real-world constraints to avoid planning complexity, which is interesting in that our OC method has its roots in speed-up of classical plan generation (from [13]).

6 Conclusions

In this paper we have described an extension to the object-centred method and associated tool-support which can be used to encode HTN planning domain models. It encourages the development of 'clean' models by (i) the requirement to develop a precise structure representing the hierarchical substate classes of each object sort (ii) the form of abstract operators, which is such that effects only appear in the context of a substate transition (iii) the use of properties such as transparency and operator completeness. Further development and debugging of the planning model is alleviated, as domain invariants and substate definitions allow strong cross checking to carried out. We have encoded the the full *Translog* domain this way, which suggests that the regularities imposed by the OCL_h is not at the cost of expression. In the future we plan to attempt to exploit the regularity brought about by the object-centred encoding in HTN plan generation, and investigate the considerable scope for extensions to OCL_h to model other types of planning.

References

[1] Philip E Agre and Ian Horswill. Cultural support for improvisation. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 363–368, San Jose, CA, 1992. AAAI Press/MIT Press.

[2] J. S. Aitken and N. Shadbolt. Knowledge Level Planning. In C. Bäckström and E. Sandewall, editor, *Current Trends in AI Planning*. IOS Press, 1994.

[3] S.A. Chien. Towards an Intelligent Planning Knowledge Base Development Environment. In *Planning and Learning: On to Real Applications. Papers from the 1994 AAAI Fall Symposium*, number FS-94-01. AAAI Press, 1995.

[4] M. Drummond. On precondition achievement and the computational economics of automatic planning.

In C. Bäckström and E. Sandewall, editor, *Current Trends in AI Planning*. IOS Press, 1994.

[5] K. Erol. *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation*. PhD thesis, Department of Computer Science, University of Maryland, 1995.

[6] K. Erol, J. Hendler, and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proc. AIPS*. Morgan Kaufman, 1994.

[7] M. Fox. Natural Hierarchical Planning using Operator Decomposition. In *Proceedings of the Fourth European Conference on Planning* , 1997.

[8] S. Kambhampati. Comparing Partial Order Planning and Task Reduction Planning: A preliminary report. Technical Report TR 94-001, Arizona State University, Dept. of Computer Science and and Engineering, 1994.

[9] D.E. Kitchin. *Object-Centred Generative Planning*. PhD thesis, School of Computing and Mathematics, University of Huddersfield, forthcoming,1998.

[10] C. A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

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

[12] T.L. McCluskey, D.E. Kitchin, and J.M. Porteous. Object-Centred Planning: Lifting Classical Planning from the Literal Level to the Object Level. In *Proceedings of 8th IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[13] J. M. Porteous. *Compilation-Based Performance Improvement for Generative Planners*. PhD thesis, Department of Computer Science, The City University, 1993.

[14] K. Erol S. Andrews, B. Kettler and J. Hendler. Translog: A Planning Domain for the Development and Benchmarking of Planning Systems. Technical Report CS-TR-3487 (UMIACS TR-95-69), University of Maryland, Dept. of Computer Science, 1995.

[15] A. Tate, S. T. Polyak, and P. Jarvis. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh, 1998.

[16] Q. Yang. Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, 6, 1990.