Object Transition Sequences: A New Form of Abstraction for HTN Planners

T. L. McCluskey
Department of Computing Science
University of Huddersfield,
West Yorkshire, UK
email: lee@zeus.hud.ac.uk

Abstract

This paper presents *EMS*, an implemented HTN planning algorithm using a novel form of abstraction. A plan is viewed as a set of dynamic objects taking part in sequences of transitions. *EMS* builds up plans using an *E*xpand then *M*ake-*S*ound cycle: a plan is built up by constructing and expanding a tree of networks at different levels of abstraction, where a network contains object transition sequences. A network is made sound by adjusting the pre and postconditions of all the object transition sequences it contains, and proving the sequence in between is sound. As new objects are discovered in the detailed levels of the hierarchically developing plan, the pre and postconditions of their transition sequences are passed up to a parent network in the hierarchy which must then be made sound. The main benefits of *EMS* are that it processes an expressive, declarative input language based on object hierarchies; it is efficient in its reasoning, in that object transition sequences can be manipulated independently of objects of unrelated sorts, and reasoning about condition achievement is performed locally in a network; and it provides a clear, sound algorithm with the potential of application to complex applications.

Introduction

One major goal of work in Hierarchical Task Network planning appears to be the creation of efficient general purpose algorithms that process expressive domain model languages, where both the algorithm and the domain modelling language are soundly based, yet have the potential to be applied to complex knowledge-based applications. In a sense this boils down to the problem of the split between elegant impractical algorithms, and ad hoc practical programs, as pointed out in reference (McDermott & Hendler 1995). Encoding non-trivial domains in HTN languages, and designing sound planning algorithms to efficiently solve tasks in such domains has long been recognised as an important yet complex task, not least because of the success of HTN planners in some fielded applications.

This paper presents *EMS*, an implemented HTN planning algorithm that inputs a domain model in an expressive, object-based language called OCL_h (McCluskey & Kitchin 1998). A model's main components are its *dynamic objects*, which go through sequences of transitions during execution of a plan. Object transition sequences offer another form of abstraction in addition to the abstraction offered by the hierarchy in HTN planning, in that reasoning about goal achievement can be limited to objects in related object classes (here called *sorts*). Also, information about transition sequences in detailed parts of the hierarchy can be reasoned with on a higher level using the pre and postconditions of the sequence alone.

EMS builds up hierarchical plans in terms of a tree of networks. It uses an *Expand then Make-Sound* cycle: it expands out a level of networks where expanding one network N means creating a new network for each non-primitive step that N contains. The algorithm then makes the new networks sound. This is done by reasoning about each object transition sequences in turn, and making it sound in the sense that any object which satisfies the precondition of the sequence, will after taking part in the transition sequence satisfy the postcondition (the pre and postconditions are called 'guards' of the sequence). As new objects are discovered in the detailed levels of the tree as a result of compound operator expansion, their transition sequences are made sound and their guards are passed up the tree, requiring some or all of the network's ancestors to be re-made sound. A hierarchical plan contains a solution when all its leaves are primitive operators, and every network has been made sound.

The main benefits of *EMS* are that (a) it processes an expressive, declarative input language based on the specification of dynamic object hierarchies which implicitly define the possible object transition sequences; (b) it is economical in its use of reasoning, in that reasoning about object transition sequences within a network N is restricted to objects of sorts related in the sort hierarchy; and reasoning about an object in other networks higher up the tree than N uses the pre and postconditions of that object's transitions. (c) it contains a clear algorithm which can be subject to analysis. In this paper we show the design of our algorithm to be sound but not complete.

The Input Language OCL_h

Objects in an OCL_h planning application are viewed as changing entities, going through sequences of transitions during plan execution. A domain modeller using OCL_h aims to construct a model in terms of objects, a sort hierarchy, predicate definitions, substate specifications, invariants, and operators. A tool-supported methodology has been designed to guide the development of models and is described in references (McCluskey & Porteous 1997; McCluskey & Kitchin 1998). Predicates and objects in a model are classed as dynamic or static as appropriate - dynamic predicates are those that may have a changing truth value throughout the course of plan execution, and dynamic objects are each associated with a changeable state. Below we define $\mathcal{P}$ as the set of all predicate structures, and $\mathcal{O}$ as the set of all objects in a domain model. Each object in $\mathcal{O}$ belongs to a unique *primitive sort*. We will use a transport logistics model based on Translog (Andrews *et al.* 1995) as a running example. A sort hierarchy is shown in Example 1, containing 4 dynamic primitive sorts *truck, package, train, traincar*.

$sorts(physical\text{-}obj, [vehicle, package])$
$sorts(vehicle, [railv, roadv])$
$sorts(roadv, [truck])$
$sorts(railv, [train, traincar])$
$sorts(location, [city\text{-}location, city])$
$sorts(city\text{-}location, [tcentre, not\text{-}tcentre])$
$sorts(tcentre, [train\text{-}station])$
$sorts(not\text{-}tcentre, [clocation, post\text{-}office])$
$sorts(route, [road\text{-}route, rail\text{-}route])$
$objects(train\text{-}station, [city1\text{-}ts1, city2\text{-}ts1, city3\text{-}ts1])$
$objects(clocation, [city1\text{-}cl1, city1\text{-}cl2, city2\text{-}cl1, city3\text{-}cl1])$
$objects(post\text{-}office, [post_1])$
$objects(city, [city_1, city_2, city_3])$
$objects(train, [train_1])$
$objects(traincar, [traincar_1])$
$objects(road\text{-}route, [road\text{-}route_1, road\text{-}route_2, road\text{-}route_3])$
$objects(rail\text{-}route, [rail\text{-}route_2, rail\text{-}route_3, rail\text{-}route_4])$
$objects(truck, [truck_1, truck_2, truck_3])$
$objects(package, [pk_1, pk_2])$

Example 1: a simple sort hierarchy

OCL_h is based on the assumption that the state of the world in a planning application can be decomposed into the state of each object in that world - these object states are called **substates** to emphasise the fact they represent a part of the total world state. A *ground, dynamic object description* is specified as a tuple (s, i, e), where i is the object's identifier, s is the primitive sort of i, and e is its substate, a set of ground dynamic predicates that all *refer* to i. All predicates in e are asserted to be true under a locally closed world assumption. For example in the logistics domain, the predicates that refer to a train may be *at(train,location), attached(train,traincar), unattached(train), in-service(train), available(train).* An object description is:

$$(train, train_1, [at(train_1, city1\text{-}ts1), \\ in\text{-}service(train_1), attached(train_1, traincar_1)])$$

Example 2: an object description

Here the local closed world assumption tells us that, for example, the train is not available, and it is not attached to other traincars apart from *traincar*-1.

Given an object identifier i, the set of legal substates that i might occupy is called $substates(i)$.

Object Expressions

Crucial to OCL_h is the idea of an **object expression**. Goals and operator preconditions are written as collections of object expressions. An object expression is a generalisation of a object description, and is specified using dynamic and possibly static predicates.

To define object expressions we need to introduce some notation that will be used throughout the paper.

- A legal substitution is a sequence of replacements, where each replacement substitutes a variable of sort s by a term which has s as either its primitive sort or its supersort.
- A set of static predicates are *consistent* if there is a legal substitution that instantiates them to facts asserted as true in the OCL_h domain model.
- If $p \subseteq \mathcal{P}$ then let $dyn(p)$ and $stc(p)$ be the dynamic and static predicates in p, respectively.

If $oe \subseteq \mathcal{P}$, then (s, i, oe) is called an **object expression** if there is an $ss \in substates(j)$ for some object identifier j of primitive sort s', and a legal substitution t such that

- $i_t = j$
- $dyn(oe)_t \subseteq ss$
- $s' = s$ or s' is a subsort of s
- $stc(oe)_t$ is consistent

In this case object (s', j, ss) is said to *satisfy* (s, i, oe). Since i could be a dynamic object identifier or variable, we refer to it as an *object term.* Example 3 is an object expression as it is satisfied by an least one object description - that of Example 2. (in Example 3 and the following examples we single capital letters inside predicates represent variables). In this case *train* is a subsort of *railv*, $t = [train_1/T, city1\text{-}ts1/Y]$ and $in\text{-}city(city1\text{-}ts1, city1)$ is consistent with the domain model (it is actually an atomic invariant).

$$(railv, T, [at(T, Y), in\text{-}service(T), in\text{-}city(Y, city1)])$$

Example 3: an object expression

Class Expressions

In OCL_h developers specify all the legal substates that a typical object of a sort may occupy at the same time as developing the operator set. This helps in the understanding and debugging of the domain model, as well as contributing to the efficiency of planning tools. The specification is written implicitly as a list of predicate expressions such that any legal ground substitution of one of the expressions will be a hierarchical component of a substate. The legal substates of identifier i are thus all ground expressions having a component from exactly one of the predicate expressions at each level in the hierarchy.

The substate in Example 2 actually has three hierarchical components - *at*, relating to physical objects, *attached* relating to rail vehicles, and *in-service*, relating specifically to trains. Objects of sort *train* are described by predicates through their primitive sort but they also inherit the dynamic predicates from supersorts *railv* and *physical_obj*.

$$(physical\_obj, T, [[at(T, L), is\text{-}of\text{-}sort(L, train\text{-}station)]])$$
$$(railv, T, [[unattached(T)], [attached(T, V1), is\text{-}of\text{-}sort(V1, traincar)]])$$
$$(train, T, [[out\text{-}of\text{-}service(T)], [in\text{-}service(T)], [in\text{-}service(T), available(T)]])$$

Example 4: hierarchical substate specification for trains

Example 4 implicitly specifies the substates of the train. If there are n stations, m traincars, and t trains then this implicitly defines all $t*(n*(m+1)*3)$ substates. Static predicates are used to capture the exact set required.

We define a **class expression** (s, i, ce) in much the same way as an object expression except that when ground ce must equate to one or more hierarchical components of an object's substate. Class expressions are important because we want to specify deterministically how classes of objects change using a parameterised notation. Hence the output of a parameterised operator (defined below) will be in terms of class expressions, which when instantiated will specify hierarchical components of an object description uniquely.

In a sense, an object description is a class expression that (a) is fully ground (b) contains a substate containing all hierarchical components.

Hence class expressions may *satisfy* object expressions under the same conditions that an object description satisfies an object expression.

Transitions

If (s, i, oe) and (s, i, ce) are an object expression and a class expression respectively, then $oe \Rightarrow ce$ is called an object **transition**. A transition of i is applicable to a class expression (s', i', ce') if under a legal substitution t ce' satisfies oe. The transition changes (s', i', ce') to (s', i', ce_t).

Predicates in the hierarchy of the substate specification that are not mentioned in the left hand side of the transition are assumed to persist when it is applied. For example, the transition of package P

$$[waiting(P), certified(P)] \Rightarrow [loaded(P, V), certified(P)]$$

changes the class expression:

$$(package, pk_1, [at(pk_1, L), waiting(pk_1), certified(pk_1)]$$

to:

$$(package, pk_1, [at(pk_1, L), loaded(pk_1, V), certified(pk_1)]$$

when this transition is applied. Here $t = [pk_1/P]$, and the predicate $at(pk_1, L)$ persists as it is specified at a higher level in package's hierarchy than the other predicates.

Transition Sequences

Let two sorts s, s' be *related* if either $s = s'$, s is a subsort of s' or s' is a subsort of s. A useful abstraction of a plan is to consider the transitions of a certain object independently of any objects of unrelated sorts. A guarded transition sequence (or *guarded trace*) for object term i of sort s is written

$$(s, i, oe_I, [oe_1 \Rightarrow ce_1, ..., oe_N \Rightarrow ce_N], ce_F)$$

where the expressions $oe_i \Rightarrow ce_i$ are transitions of any object term (including i) of a sort related to s. (s, i, oe_I) and (s, i, ce_F) are object and class expressions respectively (se_I and se_F can be thought of as the guards, or pre and postconditions of the sequence). For object term i, the transition sequence within this structure we call simply the $trace(i)$. A guarded trace extracted from our test runs in the transport logistics model is shown in Example 5. It shows the transitions of a *truck* as it moves from some city location C to $city3\text{-}cl1$, then is involved in transporting a package on to $city1\text{-}cl1$.

$$\begin{aligned}
&(truck, T, [at(T, C), movable(T), available(T)]\\
&[[movable(T), available(T)]) \Rightarrow\\
&\quad [movable(T), busy(T, pk_2)]\\
&[at(T, C), movable(T)] \Rightarrow\\
&\quad [at(T, city3\text{-}cl1)]\\
&[at(T, city3\text{-}cl1), movable(T)] \Rightarrow\\
&\quad [at(T, city3\text{-}cl1), movable(T), busy(T, pk_2)]\\
&[at(T, city3\text{-}cl1), movable(T)] \Rightarrow\\
&\quad [at(T, city1\text{-}cl1)]\\
&[at(T, city1\text{-}cl1), movable(T), busy(T, pk_2)] \Rightarrow\\
&\quad [at(T, city1\text{-}cl1), movable(T), available(T)]]\\
&[at(T, city1\text{-}cl1), movable(T), available(T)])
\end{aligned}$$

Example 5: a guarded transition sequence

A guarded trace(i) is **sound** if for any object description (s, i', ss) that satisfies (s, i, oe_I), the transitions can be applied sequentially on (s, i', ss) and result in an object (s, i', ss') that satisfies (s, i, ce_F). Likewise, a partially ordered set of transitions is called **linearly sound** if all sequences of transitions (being guarded by se_I and se_F) that conform to the partial order are sound. Partially ordered transitions can be constrained to be linearly sound in a way similar to that in partial order planning, by the addition of temporal constraints and binding constraints. What makes the abstraction useful is that if an object goes through a linearly sound trace in an abstracted plan, the trace will still be sound in the presence of all other transitions. Although such sequences are not independent of other dynamic objects (for instance pk_2 here), they are good abstractions that are central to the power of our hierarchical planner described below (and have been used previously in the specification of preprocessing tools as described in (McCluskey & Porteous 1996)).

Primitive Operators

An action in a domain model is represented by either a primitive or compound operator. Primitive operators specify under what conditions objects may go through single transitions; compound operators specify under what conditions objects go through whole transition sequences. A **primitive operator** schema O has components $(Nm, Prevail, Index, Conditionals, Statics)$, such that Nm is the operator's name followed by its parameters and $Prevail$ are the prevail conditions consisting of a set of object expressions that must be true before the operator can be executed and remain true during execution. $Index$ is a set of necessary object transitions, $Conditionals$ is a set of conditional transitions, and $Statics$ is a set of static predicates acting as constraints. A primitive operator specifying the movement of trucks between different cities is shown in Example 6.

Nm : $move(V, O, L, R)$,
$Prevail$: [ ],
$Index$:
$[[at(V, O), movable(V)] \Rightarrow [at(V, L)]]$,
$Conditionals$:
$[[loaded(P, V), at(P, O)] \Rightarrow [loaded(P, V), at(P, L)])]$,
$Statics$:
$[is\text{-}of\text{-}sort(R, road\text{-}route), in\text{-}city(O, City),$
$in\text{-}city(L, City1), City \neq City1,$
$connects(R, City, City1)]$

Example 6: a primitive operator

A primitive operator O can be applied to a state S if there is a grounding substitution t for $Index$ and $Prevail$ such that each transition in $Index_t$ can be applied to an object description in S, and each object expression in $Prevail_t$ is satisfied in S. Further, $Statics_t$ must be consistent. The new world state is S with

- the changes made to a set of objects as specified in the necessary transitions
- all *other* objects not affected by the necessary transitions, but which satisfy the LHS of a transition in $Conditionals$, changed according to that transition.

Compound and Method Operators

$(Nm, Pre, Index, Bodies)$ defines a **compound operator** C if Nm is the operator's name followed by its parameters, and Pre is a set of object expressions that must be true before C (unlike the prevail in a primitive operator objects in Pre may be affected by the operators in the expansion of C). $Index$ is a set of necessary state transitions (possibly null) similar to $Index$ in the primitive operator case. $Bodies$ is a list of n conditional expansions of the form $(Statics_i, Temps_i, Body_i)$, where $i = 1$ to n. Here $Statics_i$ and $Temps_i$ are static constraints on the parameters in the operator, and temporal constraints on the enumerated nodes in the $Body_i$, respectively. If a $Statics_i$ can be satisfied, the compound operator can be expanded into the network as specified in $Body_i$ (the $Statics_i$ need not be mutually exclusive). $Body_i$ contains nodes in the usual HTN fashion: a node is either the name of a primitive operator, the name of a compound operator, or an expression of the form 'achieve-goal(G)', where G is a class expression.

Compound operators can be split into n *methods* of the form $(Nn, Pre, Index, Statics_i, Temps_i, Body_i)$. Example 7 shows an example of a method for carrying a package from one location to another within the same city. Number n in a 'before' relation refers to the *nth* component of the method's body list.

Nm : $carry\text{-}direct(P, O, D)$,
Pre : [ ],
$Index$:
$[(package, P, [at(P, O), waiting(P), certified(P)] \Rightarrow$
$[at(P, D), waiting(P), certified(P)])]$
$Statics$:
$[is\text{-}of\text{-}sort(P, package), is\text{-}of\text{-}sort(V, truck),$
$in\text{-}city(O, CY), in\text{-}city(D, CY)]$,
$Temps$:
$[before(1, 3), before(2, 3), before(3, 4), before(4, 5)]$,
$Body$:
$[commission(V, P), achieve((truck, V, [at(V, O)])),$
$load\text{-}package(P, V, O), move(V, O, D, R1),$
$unload\text{-}package(P, V, D)]$

Example 7: a method operator

An OCL_h compound operator (in comparison with operators used in other HTN planner such as UMCP (Erol, Hendler, & Nau 1994)) has no achievable conditions on intermediate nodes in its body. This is because constraints on the persistence of facts (e.g. using a 'between' constraint) throughout a sequence of nodes in compound operator bodies can be represented within the substate specification of each sort, making the constraints implicit as discussed in reference (McCluskey & Kitchin 1998).

Whenever a method has been fully expanded to primitive operators, the resulting plan must effect the transitions given in the *Index*. A method may, however, change other objects in ways conditional on its expansion into more detailed task networks (such as the vehicle V in the example above). The property that methods decompose into networks that guarantee the transitions specified in a method's index we have called **transparency**. Earlier work discussed the design and use of a tool for checking the transparency of a domain model (McCluskey & Kitchin 1998).

The *EMS* Algorithm

EMS inputs a model in OCL_h containing transparent operators. It searches a space of **hierarchical plans** (*Networks*, *Statics*), where *Networks* is a set of networks (elements of *Networks* are structured as a tree which has networks as nodes and leaves). Object and class expressions in the elements of *Networks* contain only dynamic predicates - all the static predicates are collected together in *Statics*. A complete hierarchical plan is one in which all its networks are linearly sound, and all its leaf networks contain primitive operators. We will define these concepts in more detail below.

If (*Id*, *Pre*, *Post*, *Steps*, *Temps*) is a network, then *Id* is the network's identifier, and *Pre* and *Post* are the input and output conditions on objects necessarily changed by the steps in the network. If the network was created using a method C, then *Pre* would initially[1] be the set of object expressions formed from the object expressions in $C.Pre$ and the left hand side of all the transitions in $C.Index$ (we use the 'dot' notation here to refer to components of a tuple). Similarly, *Post* would initially be the set of class expressions formed from the right hand side of the transitions in $C.Index$.

Steps is a set of steps under temporal constraints in *Temps*. Each step has the form (*Id*, *Nm*, *Pre*, *Post*) where *Id* is the Step's identifier and *Nm* is the name of the originating operator or achieve-goal. *Pre* and *Post* have the same format and are formed in the same way as a network's *Pre* and *Post* conditions. For example, Figure 1 contains two abstract networks, $N1$ and $N3$. $N1$ can be represented as follows (although we have omitted the *names* of the steps):

$Id : N1,$

$Pre : \quad [A0, P0, B0],$

$Post : \quad [A5, P3],$

$Steps : \quad [(N2, \_, [B5, P1], [P2]), (N3, \_, [A1, B1], [A2]), (N4, \_, [A3], [A4])],$

$Temps : \quad [before(N3, N4)])$

[1]The *Pre* and *Post* of a network may change during planning as discussed later

$N1$ contains 3 steps, $N2, N3$ and $N4$. $A0, P0, B0, B5, P1, A1, B1, A3$ are object expressions, and $A2, P2, A4, A5, P3$ are class expressions.

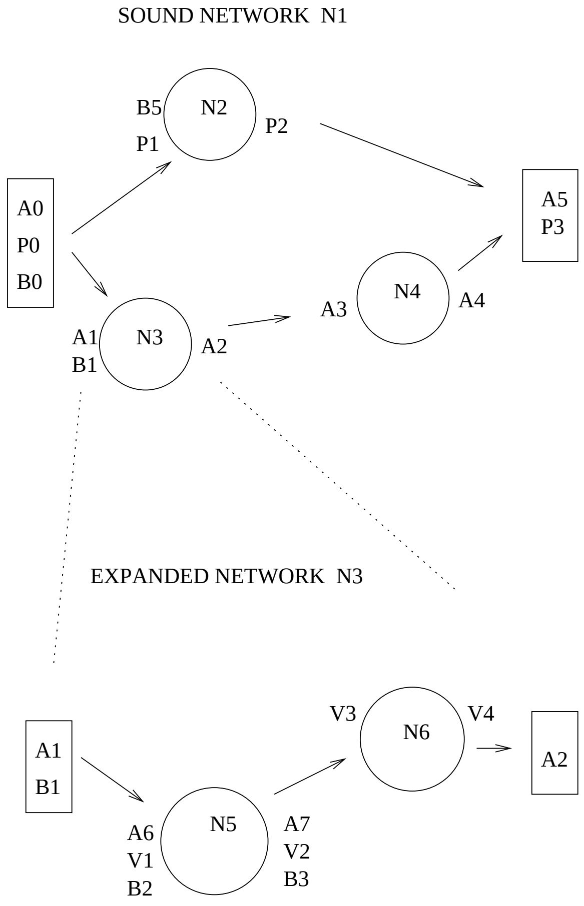

Figure 1: Expansion of a Network

Figure 2 details the top level of the *EMS* algorithm. In line 1, an initial plan is formed containing only a *root* network using the initial state and the task. A task is of the form (*Body*, *Temps*, *Statics*), where *Body* is a collection of operator names or achieve-goals, *Temps* is an ordering on the members of *Body*, and *Statics* is a set of binding constraints on variables with members of *Body*.

For example, the task 'transport package pk_1 from $city3\text{-}cl1$ to location $C1$ then transport package pk_2 from $city1\text{-}cl1$ to location $C2$, where $C1$ and $C2$ are in the same city' would be represented as follows:

$Body : \quad [transport(pk_1, city3\text{-}cl1, C1), transport(pk_2, city1\text{-}cl1, C2)],$

$Temps : \quad [before(1, 2)],$

$Statics : \quad [in\text{-}city(C1, X), in\text{-}city(C2, X)]$

The initial plan that *EMS* constructs for this task is shown below. It contains only one network (*root*). *root* contains three steps and empty pre and postconditions.

$$
\begin{array}{l}
\textit{Networks}: \\
\quad \textit{Id}: \quad \textit{root}, \\
\quad \textit{Pre}: \quad [\,], \\
\quad \textit{Post}: \quad [\,], \\
\quad \textit{Steps}: \\
\quad\quad [(n1, \textit{init}, [], [\textit{initial-state}]), \\
\quad\quad (n2, \textit{transport}(pk_1, \textit{city}3\text{-}\textit{cl}1, C1), \\
\quad\quad\quad [(\textit{package}, pk_1, [\textit{at}(pk_1, \textit{city}3\text{-}\textit{cl}1)])], \\
\quad\quad\quad [(\textit{package}, pk_1, [\textit{at}(pk_1, C1), \textit{delivered}(pk_1)])]), \\
\quad\quad (n3, \textit{transport}(pk_2, \textit{city}1\text{-}\textit{cl}1, C2), \\
\quad\quad\quad [\textit{package}, pk_2, [\textit{at}(pk_2, \textit{city}1\text{-}\textit{cl}1)])], \\
\quad\quad\quad [(\textit{package}, pk_2, [\textit{at}(pk_2, C2), \textit{delivered}(pk_2)])])], \\
\quad \textit{Temps}: \\
\quad\quad [\textit{before}(n1, n2), \textit{before}(n1, n3), \textit{before}(n2, n3)])], \\
\textit{Statics}: \\
\quad\quad [\textit{in-city}(C1, X), \textit{in-city}(C2, X)]
\end{array}
$$

Here *initial-state* represents the set of object descriptions in the initial state.

The main loop (line 2 to line 14) involves examining plans and expanding them out if they are 'sound' but not fully expanded, or making them sound if they are not so. In overview, the algorithm expands out a complete level of networks in the hierarchy. Then, if any of these new networks cannot be proved to be sound, conditions are passed back up to networks higher in the hierarchy and these networks must be re-made sound. This continues until all the networks are made sound, or this is found to be impossible, in which case the hierarchical plan is abandoned. In the next two sections we explain this technique in detail.

algorithm EMS
In OCL_h model, a task T
  initial state of dynamic objects I
Out A Solution to T
Types $Task = (Body, Temps, Statics)$
  $Plan = (Networks, Statics)$
  $Network = (Id, Pre, Post, Steps, Temps)$
  $Step = (Id, Name, Pre, Post)$

1. create and store initial plan using I and T;
2. REPEAT
3. remove a plan P from the store;
4. IF $\forall N \in P.Networks, sound(N) \wedge$
  $\exists N \in P.Networks : \neg\, expanded(N)$ THEN
5.   repeat for all unexpanded $N \in P.Networks$;
6.   expand($N, P, NewPs$);
7.   store $NewPs$;
8. ELSEIF $\exists N \in P.Networks : \neg\, sound(N)$
9.   repeat for all unsound $N \in P.Networks$
10.   make-sound($N, P, NewPs$);
11.   store $NewPs$
12.   end repeat
13. ELSE
14.   extract solution from P
15. END IF;
16. UNTIL a solution has been found,
  or there are no nodes left
17. end.

Figure 2: The *EMS* Algorithm

Network Expansion

The expansion of a network N_1 in plan P (line 6) is carried out by expanding all its non-primitive steps. For a non-primitive step S in the network, we have two cases to consider:

CASE A: This is the case where $S.Nm$ is the name of a compound operator occurring in plan P. Let *Op-Cons* be the set of constraints formed from all primitive operators in a method C:

$$Op\text{-}Cons = \{O.Statics \mid O \in C.Body, O \text{ is primitive}\}$$

We need to consider the constraints occurring in primitive operators at this stage since once they appear in a hierarchical plan, they will not be expanded further.

Consider a method C such that

(a) $C.Nm$ unifies with $S.Nm$ under substitution t;

(b) $(P.Statics \cup C.Statics \cup Op\text{-}Cons)_t$ is consistent.

When (a) and (b) are true, we can expand step S into a new network

$$N_2 = (S.Id, S.Pre, S.Post, X, C.Temp)_t$$

where X is a set of steps

$$(Id_j, M_j, Pre, Post)_t,\ n \geqslant j \geqslant 1$$

Here Id_j is a unique identifier and M_j is the *jth* of n node names in $C.Body$. If M_j is the name of an operator then $Pre = M.Pre$ and $Post = M.Post$. If M is of the form achieve-goal(G), then Pre = null and $Post = G$. The new hierarchical plan is then

$$(P.Networks \cup N_2, P.Statics \cup C.Statics \cup Op\text{-}Cons)_t$$

CASE B: We consider the case where S was formed from an achieve-goal G. In this case a method or primitive operator C is chosen from the model with a class expression (s, i, ce) in $C.Post$ such that $G_t \subseteq ce_t$ for some legal substitution t. S is transformed into a new step $(S.Id, C.Nm, C.Pre, C.Post)$, then expanded into a network using CASE A above. As a special case of this we can let C be the *no-op*, the identity operator. In this case $S.Pre = S.Post = G$.

Figure 1 gives an abstract example of network expansion. Here step N3 has been expanded into network N3 containing two steps N5 and N6.

Making Networks Sound

After a network has had all its steps expanded out into new networks, giving a new layer of detail, all of these networks must be made sound. We define the external objects of a network N to be $ext\text{-}objs(N)$, the set of dynamic object terms i of sort s such that (s, i, e) appears in $N.Pre$ or $N.Post$. Likewise, $int\text{-}objs(N)$ is the set of i's such that (s, i, e) appears *only* within the *Pre* and *Post* of the steps in N i.e. not in $N.Pre$ or $N.Post$. For the abstract networks in Figure 1, assume class or object expressions $A0, A1, ..$ etc describe object term a, those with B describe object b, and so on. Assume the sort of a is A etc. Then:

$int\text{-}objects(N1) = []$

$ext\text{-}objects(N1) = [a, p, b]$

$int\text{-}objects(N3) = [v]$

$ext\text{-}objects(N3) = [a, b]$

Next, we describe a structure (a network or step, for example) as being *sort abstracted* with respect to sort s, if that structure has been stripped of any object or class expressions referring to an object term of an unrelated sort to s. We use the notation X^s to denote any structure X that has been sort-abstracted. Informally, we prove a network sound by considering each object term i in turn. The network is sort abstracted so that it only contains objects in a related sort. For an external object term i, the steps in the network are viewed as transitions, and the object and class expressions in $N.Pre$ and $N.Post$ for i are the pre and postconditions of the trace(i). Hence, to make a network sound, we have to form the guarded transition sequence:

$(s, i, N.pre, trace(i), N.post)^s$

and for all internal and external objects i, make it linearly sound. Making a transition sequence sound involves a form of goal achievement and 'de-clobbering' similar to that found in conventional partial-order planning. The procedure is based on the truth criterion for object-centred planning is given in (McCluskey, Jarvis, & Kitchin 1999). Two object terms i and j may codesignate during this process if they are of related sorts - the resulting sort of the unified term being the most specialised of the two. Goal achievement is in a sense harder, however, than the 'literal-based' form in that each of the hierarchical components of an object expression need to be proved separately, and may have to have more than one achiever. On the other hand declobbering of the achievement of an object expression for i is made easier: consider, in the sort-abstracted network, there is another term $j \neq i$ that may codesignate with i. If the transitions of i and j affect different parts of the sort hierarchy, then the transitions of j will not affect the soundness of i. Otherwise, we must add constraints to either (a) use demotion/promotion so that the steps involved are temporally separated or (b) separate the objects with a single binding constraint $i \neq j$. There are some special cases:

(a) The algorithm may have to resort to conditional transitions to prove a trace is sound. If a conditional transition is required, then that transition is taken from the primitive operator's conditional slot and put into the step's *Pre* and *Post*. An example illustrating this occurs in our worked example below.

(b) For an external object i, $N.post$ may not contain a reference to i, as is the case for object b in $N1$ and $N3$ of Figure 1. For $N1$, this is fine as there are no class expressions in postconditions involving b (that is, object b is not *changed* within $N1$ at this level of abstraction). For $N3$, b is changed by $N5$, hence a postcondition needs to be formed before this net can be made sound.

(c) For an internal object i, suitable object expressions must be put into $N.Pre$ and $N.Post$ to form the guards of the trace. In many cases, the guards may simply be the beginning an end of the trace (for example in Figure 3 it may be that V0 = V1 and V4 = V5) but in practice the use of hierarchical objects entails that the trace may not be totally ordered and, for example, the postcondition may contain a class expression for i containing components achieved by more than one step.

Passing up altered pre and postconditions

In the example of Figure 3, let us assume that the following guarded traces have been made sound in N3:

$(A, a, [A1, A6 \Rightarrow A7, A2])$

$(B, b, [B1, B2 \Rightarrow B3, B4])$

$(V, v, [V0, V1 \Rightarrow V2, V3 \Rightarrow V4, V5])$

Assume we have a network N with a parent network N' (hence $N.Id$ is the name of a step in N'). As well as having the effect of adding constraints to a network, the process of making traces sound may have the effect of changing $N.Pre$ and $N.Post$. In this case step $N.Id$ in N' must have its *Pre* and *Post* conditions augmented, and N' must be **re-made sound**. This process we call 'passing up'. Assuming network $N1$ was made sound in Figure 1, then with the new conditions passed up from step $N3$ it must be re-made sound. As illustrated in Figure 3, this may have the effect of changing the parent network's external conditions, in which case $N1$'s parent must be re-made sound, and so on. Also internal constraints may be added such as the new temporal constraint between steps $N3$ and $N2$.

Notice that the augmented postcondition plays a part in that it may be used to satisfy preconditions. For example, in the transport problem when transporting several packages, the pre and postconditions of the carry steps will eventually include object expressions describing the the vehicle used. After all lower level networks

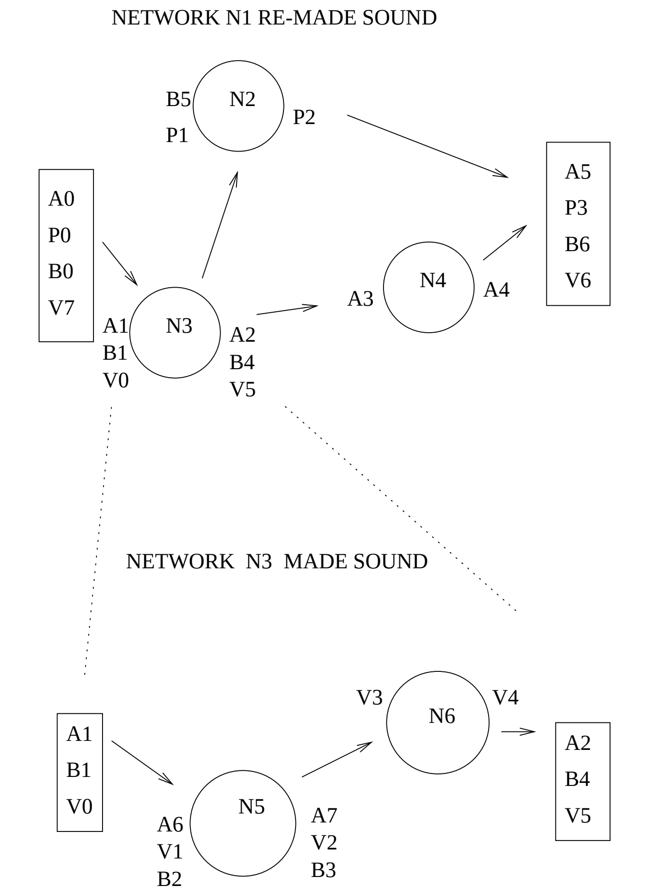

Figure 3: Making a Network Sound

are sound, re-making this network sound may then involve using the same truck to carry more than one package. In Figure 3, expansion of step N4 may entail that N4's pre and postconditions are augmented with expressions $V'1$ and $V'2$, describing object term v'. If v and v' are of related sorts, then they may unify, possibly allowing $V5$ to achieve $V'1$. In this case the postcondition of the network would have to be adjusted accordingly.

A Translog Example

To further illustrate the *EMS* algorithm, we will use parts of the workings on a transport-package task. Note that plans contain a set of constraints as well as networks - for brevity we leave out this component. Example 8 is an example of a network identified as $n6$. It was created through the expansion of step $n6$ in its parent network, using a carry-direct method operator similar to the one shown in Example 7.

$n6$ contains one internal dynamic object identifier - T. Calculating the guarded trace for T results in the expression:

$$(truck,\ T, n6.pre, trace(T), n6.post)^{truck}$$

where $trace(T)$ is identical to $trace(T)$ shown in Example 5. The guarded trace evaluates to $(truck,\ T, [\,], trace(T), [\,])$. The pre and postconditions of the trace are empty because no information is given on the desired status of the truck in the network's external conditions. The make-sound procedure in fact extracted the relevant pre and postconditions and produced the sounded guarded trace shown in Example 5. These pre and postconditions were passed up to $n6$'s parent, which was re-made sound.

$Id:\quad n6,$

$Pre:\quad [(package, pk_2, [at(pk_2, city3\text{-}cl1),$
$waiting(pk_2), certified(pk_2)])],$

$Post:\quad [(package, pk_2, [at(pk_2, city1\text{-}cl1),$
$waiting(pk_2), certified(pk_2)])],$

$Steps:$

$[(n17, commission(T, pk_2),$
$[(truck,\ T, [movable(T), available(T)])],$
$[(truck,\ T, [movable(T), busy(T, pk_2)])]),$

$(n18, move(T, C, city3\text{-}cl1, R),$
$[(truck,\ T, [at(T, C), movable(T)])],$
$[(truck,\ T, [at(T, city3\text{-}cl1)])]),$

$(n19, load\text{-}package(pk_2,\ T, city3\text{-}cl1),$
$[(truck,\ T, [at(T, city3\text{-}cl1), movable(T)]),$
$(package, pk_2, [at(pk_2, city3\text{-}cl1),$
$waiting(pk_2), certified(pk_2)])],$
$[(truck,\ T, [at(T, city3\text{-}cl1),$
$movable(T), busy(T, pk_2)]),$
$(package, pk_2, [at(pk_2, city3\text{-}cl1),$
$loaded(pk_2,\ T), certified(pk_2)])]),$

$(n20, move(T, city3\text{-}cl1, city1\text{-}cl1, R1),$
$[(truck,\ T, [at(T, city3\text{-}cl1), movable(T)])],$
$[(truck,\ T, [at(T, city1\text{-}cl1)])]),$

$(n21, unload\text{-}package(pk_2,\ T, city1\text{-}cl1),$
$[(truck,\ T, [at(T, city1\text{-}cl1),$
$movable(T), busy(T, pk_2)]),$
$(package, pk_2, [at(pk_2, city1\text{-}cl1),$
$loaded(pk_2,\ T), certified(pk_2)])],$
$[(truck,\ T, [at(T, city1\text{-}cl1), movable(T),$
$available(T)]),$
$(package, pk_2, [at(pk_2, city1\text{-}cl1),$
$waiting(pk_2), certified(pk_2)])])],$

$Temps:$

$[before(n17, n18), before(n18, n19),$
$before(n19, n20), before(n20, n21)]$

Example 8: network $n6$ containing steps $n17$-$n21$

$n6$ contains one external object pk_2. Analysis of its sort-abstracted trace shows that it is not sound in $n6$ also - $at(pk_2, city1\text{-}cl1)$ cannot be proved by any of the steps postconditions that occur before $n21$. The introduction of the conditional effects of the primitive operator *move* (shown in Example 6) into $n20$'s pre and postconditions in fact leads to the soundness of pk_2's guarded trace.

Solution Extraction

The final procedure in the algorithm, called in line 14, extracts a solution from a completed plan P. If all P's networks have been made sound, and all P's network's steps are primitive, then any sequence of primitive operators satisfying $P.Statics$ and the temporal constraints in each of the networks will be a solution.

Evaluation

Soundness

The soundness proof for *EMS* is given in two steps:

(a) If the trace of an object identifier i is sound in a sort-abstracted network, then it is sound in the network. This is because objects in unrelated sorts cannot directly affect the truth of object expressions in the trace(i). For example, detaching a train TR from a traincar TC affects the state of both these objects, although they are of unrelated sort (neither of their sorts is a supersort of the other). Hence this fact is recorded independently in the the train and traincar's substates, in that the traincar TC will go through a transition making $attached(TC, TR)$ false, and TR will go through a transition making $attached(TR, TC)$ false.

(b) In the final plan, all dynamic object terms changed by network N's steps will appear in expressions in $N.pre$ and $N.post$ (in particular, the root network will record all objects changed in the whole plan). Extracting a solution involves the extraction of all the primitive operators in the (sound) leaf networks of the plan tree. Consider an object expression for object term i that was achieved in some network, but not in the extracted solution. Then there must a 'clobbering' step specifying a transition of some object identifier j, where it is possible that $i = j$. But there must be some network in the tree (at least the root network) in which both i and j appear together - in which case, because they may co-designate, the (abstract) steps in which they appear would have been temporally separated, or the constraint $i \neq j$ added. Hence we have a contradiction to the hypothesis that i is not achieved in the extracted solution.

Completeness

With the deepening strategy described in Figure 2, *EMS* is *not* complete. The reason is as follows. Consider an unsound net N with an internal object i. During the process of making N sound, the object expression (s, i, oe) representing the precondition guard of trace(i) is extracted and put into $N.Pre$. If an achiever for (s, i, oe) is not found at a higher level in the hierarchy, then the node will be abandoned. Further expansion of the descendents of N may eventually uncover an internal object term j such that the derived postcondition of trace(j) would have achieved (s, i, oe). In this case $i = j$, and another precondition guard (s, i, oe') would have been generated, with the possibility that (s, i, oe') may be achieved in a higher network.

Model	*Objs(Dyn)*	*AtInvs*	*OpS*	*Sorts*
*Translog*1	31(15)	35	16	18
*Translog*2	73(38)	83	18	21
*Translog*3	109(57)	123	18	21
*Translog*4	145(76)	163	18	21

Table 1: Size of the Translog Models

Model	*Time*	*Plans*	*ASoln*	*MSoln*
*Translog*1	0.7	39.5	19.4	53
*Translog*2	15.4	200.0	24.8	56
*Translog*3	55.0	231.5	26.5	64
*Translog*4	88.0	247.0	27.2	62

Table 2: Running *EMS* with the Translog Models

The EMS Implementation

EMS has been implemented in Sicstus Prolog and inputs models written in OCL_h version 1.1 as described in reference (Liu & McCluskey 2000). All tests were run on a Sun Ultra 5 with 128mb of memory and a 333-MHz processor. We carried out experiments with OCL_h versions of Translog domains of varying complexity. This domain provides the minimum amount of detail to take advantage of our hierarchical, object-centred approach.

Table 1 gives a summary of the size of the models: *Objs*(*Dyn*) is the total number of objects, with the number of dynamic objects in brackets; *AtInvs* is the number of explicit atomic invariants; *OpS* is the number of operator schema, and *Sorts* the total number of sorts. *TranslogN* has N regions, with regions connected by air travel for $N > 1$.

Table 2 shows the results from running 80 tasks in total, 20 tasks in each of the four models. *Time* is the average CPU time in seconds used to solve each task; *Plans* is the average number of hierarchical plans expanded; *ASoln* is the average solution size in primitive operators; and *MSoln* is the largest solution in that batch. The results indicate a non-linear but manageable increase in resource consumption as the domain model size, in terms of objects and atomic invariants, increases. Test runs, sample models and test results are available from the resource section of the 'Planform' web site[2].

[2] http://helios.hud.ac.uk/planform

Related Work

Tsuneto et al point out the importance of calculating "external conditions" in HTN planning in reference (R. Tsuneto, J. Hendler, D. Nau 1998). Object expressions making up the preconditions of internal objects in a network are analogous to these "external conditions" in that they originate from conditions about objects which cannot be proved within the network.

SHOP (D. Nau & Munoz-Avila 1999) is a hierarchical planner which allows the modeller to create heuristic procedural knowledge by coding up the kind of hierarchical plans that will lead to efficiently generated solutions. This system is efficient, but is extreme in the amount of domain-dependent heuristics that requires encoding. Rather, *EMS* has been constructed from the point of view that a domain model needs to have a declarative semantics independent of domain heuristics or planning algorithm, which appears to be in line with Muscettola et al's conclusions of their experience paper describing a fielded planner, in reference (N. Muscettola & Williams 1998). The OCL_h language has been developed independently of the *EMS* algorithm, and is part of a general methodology which promotes the development of a declarative specification of substates and object transitions, in parallel with the development of the domain model operators. This gives opportunities for cross checking between these two forms of representation, and the developer gains a deeper understanding of the domain. The concepts of achieve-goals, index, etc have been compared to the 'Condition Types' of O-Plan in (McCluskey, Jarvis, & Kitchin 1999).

Conclusions

In this paper we have introduced object transition sequences, a novel form of abstraction for HTN planning. We have described an implemented HTN algorithm called *EMS* that exploits them, and performed analysis using its current design, showing it to be sound but not complete. The main innovation in the *EMS* algorithm is the efficient form of reasoning it uses to secure the soundness of its plans. This is because

- object transitions correspond, in general, to the sequential parts of a plan. This allows the pre and postconditions of transitions in a network to be efficiently computed, and these conditions can be used as guards in reasoning within higher-level networks.
- the sort-abstraction technique means that we only consider transitions of a related sort when reasoning about soundness.
- reasoning about the soundness of transition sequences is essential done *locally* - at no time does *EMS* have to consider the soundness of the whole plan at a global level.

Our future work will concentrate on knowledge engineering of real planning domains within the Planform collaborative project (described in the website referred to in footnote 2). Using the object-centred approach as a starting point, we aim to develop a high level platform for the systematic construction of planner domain models and abstract specifications of planning algorithms. Our ultimate aim is to create an environment that provides tool support for the synthesis of these specifications into sound, efficient planners.

Acknowledgements I would like to thank Ron Simpson of the Department of Computing Science at Huddersfield University for commenting on drafts of this paper, as well as the efforts of the anonymous reviewers.

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.

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

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proceedings of AIPS.*

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing Science, 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., and Porteous, J. M. 1996. Planning Speed-up via Domain Model Compilation. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning.* IOS Press. 233–244 .

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

McCluskey, T. L.; Jarvis, P.; and Kitchin, D. E. 1999. OCL_h: a sound and supportive planning domain modelling language. Technical report, Department of Computer Science, The University of Huddersfield.

McDermott, D., and Hendler, J. 1995. Planning: What it is, What it could be, An Introduction to the Special Issue on Planning and Scheduling. *Artificial Intelligence* 76:1–16.

N. Muscettola, P. P. Nayak, B. P., and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–48.

R. Tsuneto, J. Hendler, D. Nau. 1998. Analyzing External Conditions to Improve the Efficiency of HTN

Planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence.*