OCLGraph : Exploiting Object Structure in a Plan Graph Algorithm

R. M. Simpson and **T. L. McCluskey**
Department of Computing Science
University of Huddersfield, UK
r.m.simpson@hud.ac.uk
t.l.mccluskey@hud.ac.uk

Abstract

In this paper we describe the results of integrating two strands of planning research - that of using plan graphs to speed up planning, and that of using object representations to better represent planning domain models. To this end we have designed and implemented OCL-graph, a plan generator which builds and searches an object-centred plan graph, extended to deal with conditional effects. Our initial design and experimental results appear to confirm our conjectures that the extra information and structure of OCL benefits plan generation efficiency and algorithmic clarity.

Introduction

This paper describes work that is part of a continuing effort to evaluate the impact of modelling planning domains in an object-centred way, using a family of planning-oriented domain modelling languages known as OCL (McCluskey & Porteous 1997). The benefit is seen as twofold: (a) to improve the planning knowledge acquisition and validation process (b) to improve and clarify the plan generation process in planning systems. With regard to (b), it is our belief that certain obstacles and problems that researchers into planning algorithms encounter can be alleviated or overcome using a rich, planning-oriented knowledge representation language.

The object-centred language *OCL*, and more recently the hierarchical version OCL_h (Liu 1999; McCluskey & Kitchin 1998), have their roots in the 'sort abstraction' ideas used in the domain pre-processing work of (Porteous 1993). OCL is primarily aimed as a high level language for planning domain modelling, the main feature distinguishing it from STRIPS-languages being that models are structured in terms of objects, rather than literals. It aims to allow modellers to more easily capture and reason about planner domain encodings independent of planning architecture, and to help in the validation and maintenance of domain models. On the other hand, OCL retains all the flexibility of a STRIPS-like encoding. The rationale behind OCL has been sustained by the experience of those applying planning technology. For example, the developers of the planner aboard Deep Space 1 (N. Muscettola & Williams 1998) stress the need to develop clean, planner-independent languages that can be used to build and statically validate domain models.

In this paper we seek to tie up the advantages in creating a domain model in OCL with the use of a particularly successful form of plan generation using a plan graph algorithm called Graphplan (Blum & Furst 1997). The plan graph has been used as the basis for many experimental planning systems, and was the the basis of most of the planners in the AIPS-98 planning competition. This paper describes our investigation into the use of an *object-centred plan graph* in a Graphplan-like planning algorithm. Parallel work (Kitchin & McCluskey 1996; Kitchin forthcoming1999) is investigating the use of OCL in traditional goal directed plan-space search algorithms. The current effort is therefore part of a larger project to implement many of the best regarded planning algorithms in a manner both to process planning problems expressed in OCL and to develop the algorithms in a manner to take advantage where possible of the additional information content of OCL models.

After introducing the reader to OCL and Graphplan, we detail the design and implementation of a planner which draws from Graphplan in algorithmic details, and from OCL for its representation. We argue that the 'object-graph' algorithm embedded in OCL-graph is conceptually simpler than the corresponding literal-based algorithm. Also we have extended the algorithm to deal with conditional effects using a strategy that is similar to the factored expansion described by (Anderson, Smith, & Weld 1998). In our empirical experiments, however, we have restricted the OCL language to fit in with the planner input language in Blum and Furst's algorithm (Blum & Furst 1997).

Our results suggest that the use of OCL (i) simplifies the plan graph: proposition levels become object levels where it is implicit that an object can only be in one 'substate' at one time (ii) simplifies the detection of 'mutex' relations and (iii) provides a surprisingly natural way of dealing with conditional effects. Finally, our initial implementation using tests from benchmark domains suggest a potential speed up of up to 100 times in plan generation when comparing Graphplan with and without an OCL encoding.

Foundations of OCL

Overview

In OCL the world is populated with objects each of which exists in one of a well defined set of states (called 'substates'), where these substates are characterised by predicates. On this view an operator may, if the objects in the problem domain are in some minimal set of substates, bring about changes to the objects in the problem domain. The application of an operator will result in some of the objects in the domain moving from one substate to another. In addition to describing the operators in the problem domain OCL provides information on the objects, their object class hierarchy and the permissible states that the objects may be in. The main advantage of the OCL conception of planning problems to algorithms is that they do not need to treat propositions as fully independent entities rather they now belong to collections that can be manipulated as a whole. So instead of dealing with propositions the algorithms deal with objects (typically fewer objects than propositions). This is a type of abstraction which we believe most naturally co-insides with domain structure. It provides opportunities to improve on existing planning algorithms by adapting them to operate at the object level rather than the propositional level.

Basic Formulation

A domain modeller using OCL aims to construct a model of the domain in terms of objects, a sort hierarchy, predicate definitions, substate class definitions, invariants, and operators. Predicates and objects are classed as dynamic or static as appropriate - dynamic predicates are those which may have a changing truth value throughout the course of plan execution, and dynamic objects (grouped into dynamic sorts) are each associated with a changeable state. Each object belongs to a unique *primitive sort* s, where members of s all behave the same under operator application. In what follows we will explain those parts of OCL sufficient for the rest of the paper, the interested reader is referred to the bibliography for more information.

A 'situated object' in a planning world is specified by a triple (i, s, ss), where i is the object's identifier, s is the objects *primitive sort* and ss is its *substate* - a set of ground dynamic predicates which all *refer* to i. All predicates in ss are asserted to be true under a locally closed world assumption.

As a running example we will use a version of the Briefcase World, as this is simple and has been used in (Anderson, Smith, & Weld 1998) as the basis of their discussion on the implementation of conditional effects in 'Graph Plan'. Note that, however, this does not illustrate the full benefits of an OCL encoding as the briefcase world is structurally simple. Dynamic objects in a briefcase world could be of sort bag (identifiers briefcase,suitcase,..) or of sort thing (identifiers cheque,dictionary,suit,..), and static objects may be of sort location (identifiers home,office ..). Two examples of situated objects are

```
(cheque,[at_thing(cheque,home),
         inside(cheque,briefcase),
         fits_in(cheque,briefcase)])
(briefcase,[at_bag(briefcase,home)}).
```

A **world state** is a complete set of situated objects for all the dynamic objects in the planning application, and is usefully viewed as a total mapping between object identifiers and their corresponding substates, as an identifier is allowed to be associated with exactly one substate. States are constrained by **invariants**. These define the truth value of static predicates and the relationships between dynamic predicates. In particular they are used to record inconsistency constraints. A world state that satisfies the invariants is called well-formed.

For each sort s, the domain modeller groups a sort's substates together, specifying each group with a set of predicates called a **substate class definition**. They form a complete, disjoint covering of the space of substates for objects of s. When fully ground, a substate class definition forms a legal substate. To ensure that *any* legal ground instantiation of a substate class definition gives a legal substate, definitions usually contain static predicates. The substate class definitions for the dynamic sorts *thing* and *bag* in the briefcase world are:

```
substate_classes(thing,
   [at_thing(Thing,Location),inside(Thing,Bag),
    fits_in(Thing,Bag)],
   [at_thing(Thing,Location),outside(Thing)])
substate_classes(bag,
   [at_bag(Bag,Location)])
```

meaning that a thing can only be either at a location and in a bag that it fits into or that it is at a location but is not in any bag, and a bag must be positioned at a location. If i is a variable or an object identifier of sort s, and se is a set of predicates, then (i, se) is called an **object expression** if there is a legal substitution t such that $i_t = j$ and $se_t \subseteq ss$, for at least one situated object (j, ss). The second component of an object expression is thus called a substate expression. A **planning task** is defined by a well-formed world state, and a goal consisting of any legal mapping of object identifiers to substate expressions i.e. a goal is a set of object expressions with distinct objects identifiers.

Operator Representation

An **object transition** is an expression of the form $(i, se \Rightarrow ssc)$ where i is a dynamic object identifier or a variable of sort s, se is a substate expression describing i, and ssc is an expression describing i that if legally instantiated in any way would form a substate of i.

An action in a domain is represented by operator schema O with the following components: $O.id$, an operator's identifier; $O.prev$, the prevail condition consisting of a set of object expressions; $O.nec$, the set of necessary object transitions; and $O.cond$, the set of (conditional) object transitions. Each expression in $O.prev$

must be true before execution of O, and will remain true throughout operator execution. In the briefcase world we have operators put_in, take_out and move. The put_in operator will have a prevail section which allows us to specify that the bag is at a location L but this does not change as a result of applying the operator. The necessary section specifies that the thing must be at the same location as the bag and must be outside all containers prior to the application of the operator but as a result of applying the operator the thing will now be inside the bag but still at the same location. The operator can be specified as follows:

```
operator(put_in(T,B),
     % prevail
     [se(bag,B,[at_bag(B,L)])],
     % necessary
     [ssc(thing,T,[at_thing(T,L),outside(T)] =>
       [at_thing(T,L),inside(T,B),fits_in(T,B)])],
     % conditional
     [ ])
```

We define O.Pre to be the preconditions of O, i.e. the set of object expressions in *O.prev* and the set of left hand sides of *O.nec*. Hence put_in.Pre is [at_bag(B,L), at_thing(T,L), outside(T)]. If O is ground we can define O.Rhs to be the set of *substates* in the right hand sides of *O.nec*.

The definition of the move operator illustrates the specification of a conditional effect. In the example the conditional clause asserts that if the thing is at the same location as the bag (A) and is inside the bag then the thing changes state to being at location (B) the new location of the bag and is inside the bag. Where there is more than one clause in a conditional section they form a disjunction. The move operator is defined as follows:

```
operator(move(X,A,B),
     % prevail
     [],
     % necessary
     [ssc(bag,X,[at_bag(X,A),ne(A,B)]
     =>
     [at_bag(X,B)]) ],
     % conditional
     [ssc(thing,T,[at_thing(T,A), inside(T,X),
      fits_in(T,X)]
     =>
     [at_thing(T,B), inside(T,X), fits_in(T,X)])]
      ).
```

We define *O.cond.lhs* to be the set of substate expressions forming the left hand side of the conditional clauses. Similarly we define *O.cond.rhs* to be the set of sub states defined in the right hand side of the conditional clauses.

The Graphplan System

Graphplan (Blum & Furst 1997) has proved to be one of the fastest plan generation algorithms working with a traditional STRIPS-like planning representation. Since its introduction a number of authors have proposed amendments with a view to improving the efficiency of the algorithm further e.g. (Kambhampati, Parker, & Lambrecht 1997). Here we give only a very brief review of the algorithm, given the amount of published literature already using it. Graphplan works by building a plan-graph representing all possible plans creatable from the initial state by application of the available operators. If we consider the set of propositions true in the initial state as being at level 1 in our plan-graph then at level 2 will exist the set of all operations that are applicable, i.e. have their preconditions fulfilled by the propositions of level 1. At level 3 will be the set of propositions made true by the application of the operators of level 2. This process continues by developing the graph in exactly the same manner to additional levels. In the developing graph we record the application of operators as links that connect the propositions of the adjacent odd numbered levels. This process of moving from one level of propositions to the next supported by the application of operators is augmented by the application to every proposition at level n with a special operator *no-op* that renders the proposition true at level $n + 2$. This forward development of the graph faces a problem in that clearly in all proposition levels other than level 1 there may be propositions that cannot be jointly true. In the briefcase world the bag 'briefcase' cannot be at home and at the office. Likewise in a link layer actions may be mutually exclusive. The actions of moving the briefcase home and the action of moving it to the office cannot be simultaneously undertaken. We think of each proposition level as recording what potentially might be true at the same instant. We think of each link layer as recording the operations that might consistently be applied in parallel or where no commitment to ordering is required. The inconsistencies within a layer are recorded within Graphplan by augmenting the graph further by noting these mutually exclusive relations both between operations in the link layers and by recording mutually exclusive relations at the proposition layers. The development of the graph in this way from one proposition layer to the next mediated by a link layer constitutes the forwards phase of Graphplan.

To complete Graphplan a backwards search phase is required to find if a legal plan that satisfies the goal condition has been generated. This backwards phase is undertaken after the generation of each proposition layer, and starts by first searching the new proposition layer to see if all the propositions of the goal state are supported at this level. If they are not then the backward phase can be terminated and the next forwards phase started. If the goals are all present then the goal propositions must be checked to ensure that there are no recorded mutual exclusions between any of them. The backwards phase continues finding a set of operations that support these propositions and are themselves mutually consistent then recursively checking the preconditions of those operations in the same manner at the level two below. This process continues until we have regressed to the propositions of level 1 which by definition must be consistent with one another. If at any layer we find that the chosen set of operators are not mutually consistent then we must backtrack and see if an alternative set of operations can be chosen to support the same set of propositions. In this way Graphplan will continue interleaving its forwards and backwards phases to find an optimally parallel short legal plan, if one exists.

Conditional Effects in Graphplan

Since the original description of Graphplan a number of authors have described algorithms to extend Graphplan to allow the processing of conditional effects (Anderson, Smith, & Weld 1998). In their paper Anderson Smith and Weld argue that the relatively simple approach of expanding the conditional effects section into all combinations of possible groundings is not feasible in cases dealing with significant numbers of possible groundings. They propose instead what they call a 'factored expansion approach'. Their approach requires that a operator with conditional effects be composed of clauses, one for the non-conditional component of the STRIPS operator and one for each grounding of the conditional clause conjoined with the non conditional element. The resulting $move - briefcase$ operator with the cheque and the dictionary is as follows:

```
move-briefcase (?loc ?new)
  :effect
    (when (and (at briefcase ?loc)(location ?new)
               (not (= ?loc ?new)))
          (and (at briefcase ?new)
               (not (at briefcase ?loc))))
    (when (and (at briefcase ?loc)(location ?new)
               (not (= ?loc ?new))
               (in cheque briefcase))
          (and (at cheque ?new)
               (not (at cheque ?loc))))
    (when (and (at briefcase ?loc)(location ?new)
               (not (= ?loc ?new))
               (in dictionary briefcase))
          (and (at dictionary ?new)
               (not (at dictionary ?loc))))
```

A consequence of this approach is that each of the elements becomes a semi-independent rule which can be fired separately which results in a requirement for more complex processing of mutex relations during the search phases of the Graphplan algorithm. The approach we take in OCL-Graph is similar in that when we ground the operators the result will have one clause in the conditional effects section for each object for which the grounding of the conditional effects clause is consistent with the necessary and prevailing sections of the operator. The growth of the number of clauses in the conditional effects section as a result of grounding is linear. It is bounded by the number of objects in the problem domain of the correct object sort. We will delay further discussion until we have presented the OCL-Graph algorithm.

The Object Graph

OCL Input

We will assume that the domain model is input using a restricted from of OCL to coincide with the input language specified in reference (Blum & Furst 1997), but extended to deal with conditional effects. In particular, OCL operator schemas are translated to a ground set. The conditional element is expanded to include all consistent groundings of the conditional element. During the grounding which is done as a preprocessor step, static predicates are used to ensure consistent groundings. For example the static information about which objects fit in the briefcase and which objects fit in the suitcase is used to ensure that a conditional clause for moving the 'suit' which does not fit in the briefcase is not generated. The ground operators to move the briefcase in a world containing a cheque a dictionary and a suit from home to the office expands to:

```
operator(move(briefcase, home, office),
  % Prevail
  [],
  % Necessary
  [ssc(bag,briefcase,(
         [at_bag(briefcase, home)]
         =>
         [at_bag(briefcase, office)]))],
   % Conditional
   [
   ssc(thing,cheque,(
         [at_thing(cheque, home),
          inside(cheque, briefcase)]
         =>
         [at_thing(cheque, office),
          inside(cheque, briefcase)])),
   ssc(thing,dictionary,(
         [at_thing(dictionary, home),
          inside(dictionary, briefcase)]
         =>
         [at_thing(dictionary, office),
          inside(dictionary, briefcase)]))]).
```

The initial state is a total mapping between object identifiers and substates, and a goal condition is a mapping between object identifiers and ground substate expressions.

Building Up the Graph

We build an 'OCL-graph' in the spirit of Graphplan by first substituting the idea of a proposition level with what we call an 'object level', defined as a (total) mapping (called $level(n)$ where n is odd) between the set of object identifiers $O\text{-}ids$ and the partitioned set of all possible substates for that object:

$$\text{level(n)} : \text{O-ids} \Rightarrow \text{Table}$$

where Table is a set of substates partitioned by the substate class definitions. The intuitive idea is that if an object situation (i,ss) is potentially reachable at level n through the execution of operators then ss will be somewhere in the (partitioned) set 'level(n)[i]'.

Two immediate consequences of this representation are that:

(a) The size of every object level in a plan graph is always fixed as the number of objects in the initial state, although the size of the range sets of this map generally increases to the point where all legal substates for the objects, as defined in the substate class definition, are in the range.

(b) In a literal-based Graphplan *any* subset of the propositions at each propositional level can form a goal set which is potentially satisfiable. For example in the briefcase world, the set {in_thing(cheque,briefcase), at_thing(cheque,home),outside(cheque)} would be acceptable in principle, but would be found to be inconsistent through operator back chaining. OCL restricts goal sets to a set of *legal* object expressions - hence the above expression would not be allowed as the cheque's substate expression is not well formed (it is not a specialisation of either one of *thing's* two substate classes).

To create level(n+2) from level(n), we copy over the old mapping (this parallels the use of 'no-ops' in reference (Blum & Furst 1997)) and add new substates to level(n+2)'s range if they are created by operator application at level(n+1). Consider the briefcase world with only two locations involved (home (h) and office (o)) with the initial state of the briefcase (b) at home, and only two things the cheque (c) which is at home inside the briefcase and the dictionary (d) which is outside the briefcase. Then the development form the initial state in level 1 to level 3 is as follows:

```
level(1)[c] =
     {partition 1: [at_thing(c,h),inside(c,b)}
level(1)[d] =
     {partition 1: [at_thing(d,h),outside(d)]}
level(1)[b] =
     {partition 1: [at_bag(b,h)]}

level(3)[c] =
     {partition 1: [at_thing(c,h),inside(c,b)],
                   [at_thing(c,o),inside(c,b)],
      partition 2: [at_thing(c,h),outside(c)]}
level(3)[d] =
     {partition 1: [at_thing(d,h),outside(d)],
      partition 2: [at_thing(d,h),inside(d,b)]}
level(3)[b] =
     {partition 1: [at_bag(b,h)],
                   [at_bag(b,o)]}
```

The operators applicable at level 2 are take_out(c,b), put_in(d,b), and move(b,h,o), with the conditional effect of moving the cheque from home to the office.

Links

We define $contains(level(n), SE)$, where SE is a set of ground object expressions, and n is odd, as being true if for each (i,se) in SE, there is a substate ss $\in$ level(n)[i] such that se $\subseteq$ ss. An operator is applicable to level(n) if contains(level(n),O.Pre) is true, where O.Pre are the operator's preconditions as defined above. For example, contains(level(3),[at_bag(b,o)]) is true. Note that O.Pre excludes any elements for the operators conditional effects.

If operator O is applicable at level(n), and level(n+2)[i] contains ss, then a link $lk(O, i, ss, mode)$ is stored in level(n+1). If (a) O *changes* i's substate to ss or (b) (i,se) $\in$ O.prev and se $\subseteq$ ss or (c) O is a no-op preserving ss from level(n)[i] to level(n+2)[i] then we record *mode* as either 'change', 'prevail' or 'no-op' depending on each of the cases (a) - (c). In the running example we therefore store the following:

```
level(3)[c] =
     {partition 1: [at_thing(c,h),outside(c)],
      partition 2: [at_thing(c,h),inside(c,b)]}
level(3)[d] =
     {partition 1: [at_thing(d,h),outside(d)],
      partition 2: [at_thing(d,h),inside(d,b)]}
level(3)[b] =
     {partition 1: [at_bag(b,h)],
                   [at_bag(b,o)]}

level(2) =
{lk(no-op-1,c,[at_thing(c,h),inside(c,b)],
              no-op),
 lk(take_out(c,b),c,[at_thing(c,h),outside(c)],
              change),
 lk(take_out(c,b),b,[at_bag(b,h)],prevail),
 lk(no-op-2,d,[at_thing(d,h),outside(d)],
              no-op),
 lk(put_in(d,b),d,[at_thing(d,h),inside(d,b)],
              change),
 lk(put_in(d,b),b,[at_bag(b,h)],prevail),
 lk(no-op-3,b,[at_bag(b,h)],no-op),
 lk(move(b,h,o),b,[at_bag(b,o)],change)}
```

To process the conditional effects in the forwards phase of the algorithm, new links and object substates at level $n + 2$ are added as follows: for each conditional effect clause *ssc* in the applicable operators O at level $n + 1$ if $contains(level(n), O.cond[ssc].lhs)$ then add if not already present $O.cond[ssc].rhs$ to level $n+2$ and add a link from O to $O.cond[ssc].rhs$ and label the link 'cond'. For the briefcase this adds a new substate to $level(3)[c]$ and adds a new link to record the application of the effect as follows:

```
level(3)[c] =
     {partition 1: [at_thing(c,h),outside(c)],
      partition 2: [at_thing(c,h),inside(c,b)],
                   [at_thing(c,o),inside(c,b)]}

lk(move(b,h,o),c,[at_thing(c,o),inside(c,b)],cond)}.
```

We have applied one of the conditional elements in the 'move' operator. In applying such conditional elements we only consider operators that have already been applied, that is operators that have their prevailing and necessary preconditions fulfilled at that level, these operators already have their necessary effects and links recorded as described above. The notation $O.cond[n].lhs$ is to be read as the left hand side (precondition) of the nth clause of the conditional effects clauses of operator O.

Mutual Exclusions in OCL-Graph

The forward development of the plan graph spreads in the manner described above. It is checked, however, by the use of mutual exclusion conditions on both operators and substates in the object levels. Blum and Furst's 'Interference' statement ((Blum & Furst 1997), section 2.2) is paraphrased as follows: 'If either of actions O1 and O2 deletes a precondition or Add-Effect of the other, they are mutually exclusive at that level. Secondly if actions 01 and 02 have preconditions which are recorded as mutually exclusive then they are mutually exclusive' The idea is then to check each operator at each level against all the others, resulting in a set of binary mutual exclusions (which are not transitive).

We exploit the structure of OCL to give the following definition:

> For each object identifier i in the object level(n+2), the set
> $\{$ O : lk(O,i,ss,mode) $\in$ level(n+1) $\}$
> forms an N-ary mutual exclusion relation (where N is the size of the set).

In other words, all operators that support the same object form a set whose members are mutually exclusive to one another. The rationale is as follows: if operators O1 and O2 change or rely on the same object being in a particular substate, then they would in general interfere with each other. There is, however, *two* exceptions to the general rule above. If lk(O1,i,ss,prevail) and lk(O2,i,ss,prevail) are in level(n+1), or lk(O1,i,ss,prevail) and lk(O2,i,ss,no-op) are in level(n+1), then it does not follow that O1 and O2 are mutually exclusive. In practice we say O1 and O2 conflict if there is an reference to different states of the same object either in the preconditions of the two operators or in the operators necessary effects. Secondly if any operation has a conditional effect lk(O3,i,ss,cond) that has fired we do not add the operator to any mutual exclusion set as a result of conflicts between *ss* and other states of *i*. We do not add the conflict at this stage as the conditional effect may not be used in the final plan even though the operator is. We do not in the forwards development of the graph detect if the firing of one element in an operator will force the firing of another. This is contrary to the practice of (Anderson, Smith, & Weld 1998).

The case made by (Anderson, Smith, & Weld 1998) for the need to recording such induced mutexes derives from two cases.

- If two components of an operator are such that the preconditions of one of the components cannot be logically met without meeting the preconditions of the second component then we need to record that component one will be mutexed with all the operators component two is mutex with.
- The second case is harder to paraphrase but essentially if component one can fire and due to absence of other information the only way component two could fail to fire is if component one did not fire then again we can deduce that one forces two and should be mutexed with the operators two is mutexed with.

In OCL the first of the cases cannot arise as each element of an operator will refer to a different object and hence the preconditions for a conditional clause to fire cannot be contained in the other elements of the operator. The second case can arise. For example in the briefcase world if at level one the cheque is inside the briefcase, and we move it, then the cheque will also move. There is no other possibility as no other possible state of the cheque is recorded at this level. At later levels other states of the cheque will also be recorded and hence there will not be the same guarantee that moving the briefcase moves the cheque.

We could search for such cases but they are just a special case of a conditional effect being forced as a result of the interplay of the preconditions of a set of operators at a given level. We could not deal with the general case in the forward phase of graph development as the set of operators will be dependent on the choices made in identifying a candidate valid plan. We therefore leave the backwards search phase of the planner to take care of potential conflicts arising from such conditional effects.

Employing this method to the example above, the mutexes turn out to be:

```
mutex(2) = {
{ no-op-1, take_out(c,b)},
{ no-op-2, put_in(d,b)},
{ move(b,h,o), no-op-3 },
{ move(b,h,o), take_out(c,b) },
{ move(b,h,o), put_in(d,b) } }
```

The mutex that we miss by delaying consideration of conditional effects is `{move(b,h,o), no-op-1}` . That is we cannot move the briefcase from home to the office with the cheque inside and simultaneous leave the cheque inside the briefcase at home. Note that the exceptions to the general mutex rule rule collapses the mutex formed by considering the 'briefcase' to binary mutexes.

Using the set of mutexes, we can now define the concept of consistent operator sets, which will be used in the algorithm below:

> A set of operators Y, applicable at level(n), is *consistent* if
> $\neg\ \exists$ M $\in$ mutex(n+1) : $|$ M $\cap$ Y $| > 1$

In other words, a set of operators is consistent if it does not contain 2 or more operators in the mutexes at level n. This however says nothing about possible conflicts arising out of the use of conditional effects.

Mutual exclusion conditions on object levels: In the original Graphplan description, two propositions p1 and p2 were mutually exclusive if all operators creating proposition p1 were exclusive of operators for creating p2. In the OCL formulation, mutual exclusion of object expressions (i,se1) and (j,se2) is immediately true if i = j and se1 and se2 fall into different substate classes. If i <> j then the two states se1 and se2 are mutually exclusive if all operations that support se1 are mutually exclusive of all operations that support se2, which can be checked using the stored mutex relations.

The OCL-graph Algorithm

```
algorithm OCL-graph
In O-ids : Object identifiers; I : O-ids ⇒ Substates,
Ops : Ground Operators, G : Goals
Out P : Parallel Plan
Types level(n) (n odd) is a map O-ids ⇒ Table, level(n)
(n even) is a set of links
Types mutex(n) is a set of operator sets
1. ∀ i ∈ O-ids: level(1)[i] = {I[i]}
2. n := 1;
3. ACHIEVE(G,1, P) ;
4. while (P = null) do
5.    level(n+2) := level(n);
6.    links(n+1) := { };mutex(n+1) = { };
7.    ∀ i ∈ O-ids: ∀ ss ∈ level(n+2)[i] :
        add lk(no-op-X, i, ss, no-op) to level(n+1);
8.    ∀ O ∈ Ops do:
9.      if contains(level(n), O.Pre) then
10.       if not MUTEX(O.Pre,n) then
11.         ∀(i,ss) ∈ O.RHS: add ss to level(n+2)[i],
12.           add lk(O,i,ss,change) to level(n+1);
13.         ∀ (i,se) ∈ O.prev:
14.           if se ⊆ ss & ss ∈ level(n+2)[i]
15.           then add lk(O,i,ss,prevail) to level(n+1);
16.         ∀ ssc ∈ O.cond:
17.           if contains(level(n),ssc.LHS) &
                not MUTEX(Pre ∩ ssc.LHS)
18.           then add ssc.RHS to level(n+2)[ssc.i];
19.           add lk(O,ssc.i,ssc.RHS,cond), to level(n+1);
20.           end if
21.       end if
22.     end if
23.   end for;
24.   ∀ i ∈ O-ids:
        add {O : lk(O,i,ss,mode) ∈ level(n+1)} to
        mutex(n+1) and deal with exceptions;
25.   n := n+2;
26.   if contains(G,level(n)) then ACHIEVE(G, n, P);
27. end while
28. end.
```

Figure 1: An Outline of the Object-Graph Planning Algorithm

Forwards Phase

Figure 1 shows the overall algorithm. Line 1 initialises the first level in the plan graph using the initial state. If the goals are not trivially achieved (Line 3), the algorithm builds two new levels, a new object level (n+2) and a link level (n+1) First in Line 7 the object states of level n are copied to level n+2 and the no-ops links added (note each no-op is given a unique identifier no-op-X). Following the addition of the no-ops, the code in the internal loop (Lines 8 to 23) applies the domain operators initially without reference to their conditional effects and the new object level is augmented and appropriate links added (lines 11,to 15). Following the application of an operator each state change clause of the operator's conditional effects is considered and if its preconditions are met and do not conflict with the preconditions of the prevail and necessary section it is applied and the appropriate substates and links added to the corresponding levels. (lines 16 to 20) After the loop adding all new substates to level n+2 and all links to level n+1 completes, operator mutex sets are built and added to level $n + 1$ in Line 24.

```
procedure ACHIEVE(SS : set of substate expressions,
n : odd integer, P : plan );
Global levels, mutexes
Out a parallel plan P;
1. if n = 1 & contains(level(1), SS) then P = { }
2. elseif n = 1 and not contains(level(1),SS) then
3.   P = null
4. else
5.   P' = null;
6.   choose Y := a consistent set of operators that
     achieve a set of substates containing SS;
7.   while(Y <> null & P' = null) do
8.     Y' := union of all the operators necessary
       and prevailing preconditions in Y;
9.     Y" := {};
10.    while(Y" <> null & P' = null) do
11.      Y" := COND_PRECONDITIONS(Y,n)
12.      ifY" <> null then
13.        ACHIEVE({Y'∪Y"},n-2,P')
14.    end while
15.    if not(P' = null) then
16.      P := append(P',Y)
17.    else
18.      systematically generate another choice for Y
19.    end if
20.  end while
21. end if
22. end.
```

Figure 2: Achieve Procedure for the Object-Graph Algorithm

Backwards Phase

Figure 2 details the definitions of 'ACHIEVE' which has overall control of the backwards search for a valid plan. ACHIEVE searches for a consistent operator set Y to achieve G, and if it finds one first calls COND_PRECONDITIONS to determine which conditional effects of the operators in set Y are required to achieve G and adds the preconditions of those elements to the necessary and prevailing preconditions of the operators Y. ACHIEVE then recursively calls itself at level(n-2) with the set of preconditions of Y as the new goals to achieve. The definition of consistent in Line 6 and 18 is left open ended, and depends on whether mutexes are stored concerning substates, as well as checking to see whether a goal expression is well formed with respect to the object class definitions. As a minimum the set of operators Y must not contain a subset of cardinality greater than one that is contained in the *mutex* data structure for that level. The current OCL-graph implementation does not memoize substate mutexes, but this is a subject for on-going research.

```
function COND_PRECONDITIONS(O : operator set,
n : odd integer): set of substate expressions
Global levels,mutexes

1. SS' := {SS - {se : se ∈ O.RHS}};
2. Required := a set of conditional elements from
      O.COND that achieve a set of substates
      containing SS';
3. while Required <> null do
4.    Spare := {O.COND - Required};
5.    ifnotMUTEX({O.Pre ∪ Required.lhs},n) &
6.      ∀ ssc ∈ Spare
7.          if ssc.lhs satisfied in {O.Pre ∪ Required.lhs} then
              not ssc.rhs conflicts with
              {O.RHS ∪ Required.RHS}
8.    then
9.      return Required;
10.   else
11.     Required := choose new set from O.COND
          that achieves the set of substates containing SS'
12.   end if
13. end while
14. return null;
15. end.
```

Figure 3: Selection Conditional Effect Elements for Plan Inclusion

The strategy for selecting conditional effects is shown in Figure 3. In line 1 we determine which substate expressions of the Goal state have not been supported by the necessary or $no-op$ effects of the chosen operator set O, these are the substate expressions that must be supported by the conditional effects. Line 2 selects a set of those substate change clauses from the conditional effects of the operators O that satisfies the unfulfilled goals SS'. The procedure then iterates on the selected set of clauses if any to check their consistency. To check the consistency of a selection we first determine those conditional effect clauses contained in the operator set O which are not needed to support the goal (Spare). We then check that the preconditions of each of the 'Required' clauses is consistent with the main preconditions of the selected operator set O and that none of the Spare conditional effects would if they are fired by the preconditions already required conflict with the outcomes of the operators selected. If these conditions are met we have successfully chosen the conditional effects needed and simple return them otherwise we must see if an alternative set of conditional effect elements can be generated to meet the requirement.

```
function MUTEX(SS : set of substate expressions, n :
odd integer): boolean
Global levels, mutexes

if n = 1 & contains(level(1), SS) then
   MUTEX := false
else if n = 1 and not(I contains SS) then
   MUTEX := true
else if ∃ Y, a set of operators that achieve a set
   of substates containing SS, and
   not( ∃ M ∈ mutex(n-1) : | M ∩ Y | > 1 ) then
   MUTEX := false
else MUTEX := true
end.
```

Figure 4: Detecting mutex relations in a set of Object Substates

The primary method for determining that a set of object states are consistent is the function 'MUTEX'. Figure 4 It does the checking very simply, by trying to find a set of consistent operators at the level below which add these substate expressions. Operators are consistent if no subset containing two or more operators is stored in the *mutex* structure at the corresponding level.

Implementations

To try and establish the benefits of using OCL in a Graphplan like algorithm two separate implementations were created. The first, though it could process OCL descriptions of planning domains, made no attempt to benefit from the structure. Rather it was used to simply extract the elements of the standard STRIPS style operators. Essentially operators were still conceived of as possessing a list of propositions which formed the preconditions to an action and two lists of propositions, the add list and the delete list. The add list contained the new propositions made true as a result of the application of the operator and the delete list contained those propositions made false by the application of the operator. In particular no attempt has been made to utilise the grouping of atomic propositions by the object they relate to. Similarly the internal data structures of this implementation of Graphplan do not utilise 'objects'. The graph is conceived of as made from proposition layers i.e. the propositions potentially true at an instant and links connecting the propositions in successive layers where each such link corresponds to the application of a single operator. The graph also contains edges between individual links to show that they are mutually exclusive and edges between propositions in the same layer to establish that they are mutually exclusive. The implementation, though done in Prolog tries to be faithful to the description of Graphplan provided above. This implementation is designed to form our base measure for conducting experiments in an attempt to investigate the advantages in utilising the structures inherent in OCL. We will refer to this implementation of Graphplan as 'vanilla' Graphplan. The strategy we adopt to translate OCL operators into STRIPS operators is not fully general. In some domains the connections between predicates is not automatically detected. Rather than hand craft the STRIPS operators we have only run comparisons where there is a fairly close resemblance between the operators in the two representations.

OCL-Graph data structures

The primary innovation in our second implementation of Graphplan is to replace the proposition layers in the graph with object layers. To assist in the searching of these layers the map structure defined in the abstract algorithm was flattened to allow easier searches for specific substates of a given object. Also to aid referencing these states in operator links we introduced identifiers for each such substate of an object at a given level. The size of this map generally grows from one level to another but it has an upper bound determined by the number of objects in the problem domain and the number of substates of each object.

In our implementation of OCL-graph the action links that join two adjacent object levels are stored in a structure that identifies the operation performed, the object states that jointly form the preconditions of the action and an element to identify the substates of objects resulting from the application of the action. The backwards references to object substates forming the preconditions of an action assist in the backwards search undertaken during the *achieve* phases. This search is also assisted by our ability to store the references both to preconditions and to supported object substates using the identifiers mentioned previously.

The remaining data-structure that constitutes the graph stores the mutual exclusions between operations. In the implementation we do not explicitly record *mutex* relations between different substates of the same object though they are found in the attempt to *achieve* or apply an operator. The only *mutexes* stored relate to links. Finally to aid efficient searching for mutexes we store them in a form of adjacency list.

Empirical Results

Tests have been carried out on a number of the standard 'toy' domains, such as the Rocket World and the Robot World and the Briefcase world for conditional effects. The tests have involved comparing times of the vanilla version of Graphplan against the OCL version. The restriction on the domains has risen due to the ease of automatically deducing the STRIPS operators from the OCL versions of these domains. The comparisons have also been restricted by the fact that 'vanilla' does not deal with conditional effects.

The results of our tests would indicate a speed up of the algorithm by a factor of over 100 times. To give an indication of the improvement the following table shows the average result of running the two versions on problems in the Rocket world over ten different problems. In these experiments the code was run in compiled Sicstus Prolog hosted on a Sun Ultra 5. The times refer to cputime measured in minutes.

domain	vanilla	ocl
rocket	3.08	0.03

Table 1: Rocket - Robot World Timings

In addition to timing the algorithms several other measurements were taken from the sample runs. We were particularly interested in the relative sizes of the graphs created by the two algorithms. To do this we have compared the number of propositions at a given level of the graph with the number of object states created by running the same problem. We also compared the number of operator links at a given level, The figures presented in the latter tables are for problems in the rocket domain which has been extended by a refuel operation to allow for some interesting depth.

	vanilla		ocl	
Levels	**propositions**	**links**	**Substates**	**links**
1	7	n/a	5	n/a
3	21	19	17	17
5	30	77	19	37
7	30	104	30	69
9	30	104	30	116
11	n/a	n/a	30	116

Table 2: Rocket World Levels

The difference in the numbers of levels in the plans resulted from cases where refuel operations occurred. OCL-Graph mutexed them with load or unload operations and hence produced longer plans than vanilla Graphplan. Otherwise in the fairly typical example shown here there does not appear to be a significant variation in the sizes of the graph built by the two algorithms.

Analysis

The first point to be made is that despite the promising results we recognise the limitations of the experiment. Timings are a notoriously poor way of trying to measure efficiency and may be distorted by all sorts of extraneous factors.

The test results are very encouraging and suggest that there is a worthwhile efficiency improvement. This is despite the fact that in our test cases the graph built by the OCL-Graph may in some cases be larger than the equivalent graph in the vanilla version of Graphplan. The larger graph can be a result of both the larger number of object substates as compared to propositions and the greater number of operator instantiations to object substates than ground propositions. But even in those cases the OCL version was faster. The efficiency gain we conjecture results from the *implicit mutexes* inherent in storing object substates allowing both a faster construction of explicit mutex relations and fewer opportunities for backtracking.

Conclusions

In this paper we have illustrated how a graph-based algorithm can be extended to more structured representations of planning domains. We have also shown that there is a high probability that such an approach will yield worthwhile efficiency gains. Our design of the Object-Graph algorithm has thrown up various ways in which the extra information content of OCL can be used to make the graph-based algorithm more efficient.

They are many avenues for future work. First we would like to extend the experimental base to cover cases with a greater diversity of graph sizes, and to experiment with more interesting domains possessing more structure. Secondly, there is a need to attempt to analyse the computational complexity of the OCL-based algorithm in greater depth, and compare it with the original. Thirdly, we need to extend the algorithm to be able to accept the full OCL language, and to improve the algorithm so that it uses the extra information given in an OCL model. For example, domain invariants typically found in an OCL model often read as mutex constraints on a pair of substates. Finally, improvements to the basic algorithm such as dependency directed backtracking (Kambhampati 1998) have not been implemented but there is no reason to expect that they would not be equally applicable to our version of the algorithm.

References

Anderson, C. R.; Smith, D. E.; and Weld, D. S. 1998. Conditional Effects in Graphplan. In *Fourth International Conference on Artificial Intelligence Planning Systems.*

Blum, A. L., and Furst, M. L. 1997. Fast planning through Planning Graph Analysis. *Artificial Intelligence* 90:281–300.

Kambhampati, S.; Parker, E.; and Lambrecht, E. 1997. Understanding and Extending Graphplan. In *Proceedings of the 4th European Conference on Planning.*

Kambhampati, S. 1998. On the relations between intelligent backtracking and explanation-based learning in planning and constraint satisfactions. *Artificial Intelligence* 105.

Kitchin, D. E., and McCluskey, T. L. 1996. Object-centred planning. In *Proceedings of the 15th Workshop of the UK Planning SIG.*

Kitchin, D. E. forthcoming,1999. *Object-Centred Generative Planning.* Ph.D. Dissertation, School of Computing and Mathematics, University of Huddersfield.

Liu, D. 1999. The OCL Language Manual. 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. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence* 95:1–65.

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.

Porteous, J. M. 1993. *Compilation-Based Performance Improvement for Generative Planners.* Ph.D. Dissertation, Department of Computer Science, The City University.