An Object-Graph Planning Algorithm

R.M.Simpson and T. L. McCluskey
Department of Computing Science
University of Huddersfield, UK

September 1999

Abstract

In this paper we describe the results of our initial attempt to integrate 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. 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.

1 Introduction

This document 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 fairly simple planning-oriented domain modelling languages known as OCL [11]. 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-centered language **OCL**, and more recently the hierarchical version $\mathbf{OCL_h}$ [8, 9], have their roots in the 'sort abstraction' ideas used in the domain pre-processing work of references [13, 10]. 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 is 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 [12] 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 plan generation algorithm called Graphplan [1]. Graphplan has been used as the basis for many experimental planning systems, and was the predominant technology used in the AIPS-98 planning competition. This paper describes our initial investigation into the use of an *object-centred plan graph* in a Graphplan-like planning algorithm. Parallel work [6, 5] 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, in section 4 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, although in this initial work we have restricted the OCL language to fit in with the planner input language in Blum and Furst's algorithm [1]. In particular 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) makes graph generation more efficient: for example operator mutexes are generated in N steps, where N is the number of objects in the application (iii) makes graph searching more efficient: for example the operator mutexes generated are n-ary relations rather than binary (i.e. they are generally 'multi-mutexes' in the jargon of reference [2]), allowing speedier identification of inconsistent operator sets. Finally, our initial implementation using tests from two benchmark domains suggest a speed up of between 4 and 10 times in plan generation when comparing Graphplan with and without an OCL encoding.

2 Foundations of OCL

2.1 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, and we believe provides opportunities to improve on existing planning algorithms by adapting them to operate at the object level rather than the propositional level.

2.2 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 pair $(\mathbf{i}, \mathbf{ss})$, where **i** is the object's identifier, 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 Rocket World, as this was the original example used for Graphplan in reference [1]. Note that, however, this does not illustrate the full benefits of an OCL encoding as the rocket world is structurally trivial (e.g. there are no static predicates). Dynamic objects in a rocket world could be of sort rocket (identifiers r1,r2,..) or of sort cargo (identifiers c1,c2,..), and static objects may be of sort location (identifiers Jfk, Ln, Ps ..) or fuel_status (identifiers full, empty). Two examples of situated objects are (r1, {at(r1,paris),fuel(r1,empty)}) and (c1,{in(c1,r1)}).

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 cargo and rocket in the rocket world are:

substate_classes(cargo,[[at(cargo,location)], [in(cargo,rocket)]])
substate_classes(rocket,[[pos(rocket,location), fuel(rocket,fuel_status)]])

meaning that cargo can only be either at a location or in a rocket, and a rocket must be positioned at a location and have some fuel status. If **i** is a variable or an object identifier of sort **s**, and **se** is a set of predicates, then $(\mathbf{i}, \mathbf{se})$ is called an **object expression** if there is a legal substitution **t** such that $\mathbf{i_t} = \mathbf{j}$ and $\mathbf{se_t} \subseteq \mathbf{ss}$, for at least one situated object $(\mathbf{j}, \mathbf{ss})$. The second component of an object expression is thus called a substate expression. A **planning task** is normally defined by an initial state and a goal; in OCL an initial state is any well-formed world state, and a goal is any legal mapping of object identifiers to substate expressions i.e. a goal is a set of object expressions with distinct objects identifiers.

2.3 Operator Representation

An **object transition** is an expression of the form $(\mathbf{i}, \mathbf{se} \Rightarrow \mathbf{ssc})$ 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**, the 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**, and 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 rocket world we have operators load, unload and move. load can be specified as follows:

```
operator(load(C,R,A),
     % prevail
     [ se(R,[pos(R,A)]) ],
     % necessary
     [ ssc(C,[at(C,A)] => [in(C,R)]) ],
     % 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 load.Pre = [se(R,[pos(R,A)]), se(C,[at(C,A)])] If O is ground we we can define O.Rhs to be the set of **substates** in the right hand sides of **O.nec**. For the rest of the paper we restrict ourselves to models with no conditional operators.

3 The Graphplan System

Graphplan [1] 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. [4]. 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 rocket world the rocket 'r1' cannot be at Paris, Jfk and at London. Likewise in a link layer actions may be mutually exclusive. The actions of moving rocket 'r1' to Paris and the action of moving the rocket to Jfk 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. 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.

4 The Object Graph

4.1 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 [1]. In particular, OCL operator schemas are translated to a ground set, and do not contain a conditional component. 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. Some recent work has been aimed at extending Graphplan to cope with more expressive input languages (e.g. [7]) but we leave corresponding extensions for future work.

4.2 Building Up the Graph

We will 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-ids** and the partitioned set of all possible substates for that object:

level(n) : O-ids $\Rightarrow$ 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 at each time step.

(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 rocket world, the set {in(c1,r1), at(c1,paris)} 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 an object expression (c1, {in(c1,r1), at(c1,paris)} would not be allowed as c1's substate expression is not well formed (it is not a specialisation of either one of rocket'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 [1]) and add new substates to level(n+2)'s range if they are created by operator application at level(n+1). Consider the trivial rocket world with only two cities involved (London (Ln) and Paris (Ps)) with the initial state of one rocket r1 at London, and two packages c1,c2 at London. Then

level(3)[c1] = { partition 1: [in(c1,r1)], partition 2: [at(c1,Ln)] }
level(3)[c2] = { partition 1: [in(c2,r1)], partition 2: [at(c2,Ln)] }
level(3)[r1] = { partition 1: [pos(r1,Ln), fuel(r1,full)], [pos(r1,Ps),fuel(r1,empty)] }

as the operators applicable at level 2 are load(c1,r1,Ln), load(c2,r1,Ln), and move(r1,Ln,Ps).

4.3 Links

We define $\textbf{contains}(\textbf{level}(\textbf{n}), \textbf{SE})$, 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 an 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),[pos(r1,Ln)]) is true.

If operator O is applicable at level(n), and level(n+2)[i] contains ss, then a link '$\textbf{lk}(\textbf{O}, \textbf{i}, \textbf{ss}, \textbf{mode})$' 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]. Here mode is either 'change', 'prevail' or 'no-op' depending on each of the caes (a) - (c). In the running example we therefore store the following:

level(2) =

{ lk(no-op-1,c1,[at(c1,Ln)],no-op), lk(load(c1,r1,Ln),c1,[in(c1,r1)],change),
lk(no-op-2,c2,[at(c2,Ln)],no-op), lk(load(c2,r1,Ln),c2,[in(c2,r1)],change),
lk(no-op-3,r1,[pos(r1,Ln), fuel(r1,full)],no-op),
lk(move(r1,Ln,Ps),r1,[pos(r1,Ps),fuel(r1,empty)],change),
lk(load(c1,r1,Ln),r1,[pos(r1,Ln), fuel(r1,full)],prevail),

lk(load(c2,r1,Ln),r1,[pos(r1,Ln), fuel(r1,full)],prevail) }

4.4 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 ([1], 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.' 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 (these are referred to as multi-mutex relations which are more powerful than binary relations according to Fox and Long [2]). 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, **one** exception 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.

The mutually exclusive sets of operators can be stored in N steps, where N is the number of objects in the initial state. We read off the operators supporting each object's substates into a mutually exclusive set, stored in a set of mutexes at the link level n+1 (noting the exception above which would split up the set). Employing this method to the example above, the mutexes turn out to be:

mutex(2) = {
{ no-op-1, load(c1,r1,Ln)}, { no-op-2, load(c2,r1,Ln)}, { move(r1,Ln,Ps), no-op-3 },
{ move(r1,Ln,Ps), load(c1,r1,Ln) }, { move(r1,Ln,Ps), load(c2,r1,Ln) } }

Note that the exception to the rule collapses the mutex formed by considering r1 to, in this example, 3 binary mutexes. Using the set of mutexes, we can now define 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.

Mutual exclusion conditions on object levels: In the original Graphplan description, two propositions were mutual exclusive if all operators creating proposition p1 were exclusive

```
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] = a set containing I[i] in the appropriate partition
2. n := 1;
3. ACHIEVE(G,1, P) ;
4. while (P = null) do
5.    level(n+2) := level(n); level(n+1) := { }; mutex(n+1) = { };
6.    ∀ i ∈ O-ids: ∀ ss ∈ level(n+2)[i] :
7.        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(P = null) & not MUTEX(Pre,n) then
11.             ∀ (i,ss) ∈ O.Rhs: add ss to level(n+2)[i], add lk(O,i,ss,change) to level(n+1);
12.             ∀ (i,se) ∈ O.prev: if se ⊆ ss & ss ∈ level(n+2)[i]
13.                                  then add lk(O,i,ss,prevail) to level(n+1);
14.           end if
15.        end if
16.   end for;
17.   ∀ i ∈ O-ids: add {O : lk(O,i,ss,mode) ∈ level(n+1)} to mutex(n+1) and deal with exceptions;
18.   n := n+2;
19.   if contains(level(n),G) then ACHIEVE(G, n, P);
20. end while
21. end.
```

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

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. Otherwise, we can check (and possibly 'memoize' the set of object expressions and the result if required) for this kind of mutual exclusion as in the original Graphplan algorithm.

5 An OCL Planning Algorithm based on the Plan Graph

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. In Lines 7 the no-ops links are added to the structure (note each no-op is given a unique identifier no-op-X). In the main loop (Lines 8 to 16) the operators are applied to the previous level, with appropriate links (lines 11,12 and 13). After this loop, operator mutex sets are built up in Line 17.

Figure 2 details the (similar) definitions of 'ACHIEVE' and 'MUTEX'. The latter algorithm is

```
procedure ACHIEVE(SE : set of substate expressions, n : odd integer, P : plan );
Global levels, mutexes
Out a parallel plan P;
1. if n = 1 & contains(level(1), SE) then P = { }
2. else if n = 1 and not(I contains SE) then P = null
3. else if inconsistent(SE) then P = null
4. else
5.      P' = null;
6.      repeat
7.              choose Y := a set of operators that achieve a set of substates containing SE;
8.              Y' := union of all the operators' preconditions in Y;
9.              if ∃ M ∈ mutex(n-1) : | M ∩ Y | > 1 then call ACHIEVE(Y',n-2,P');
10.     until there are no choices left or not(P'=null);
11.     if not(P'=null) then P = append(P',Y) else P = null
12. end if
13. end.
function MUTEX(SE : set of substate expressions, n : odd integer): boolean
Global levels, mutexes
1. if n = 1 then MUTEX := false
2. else if ∃ Y, a set of operators that achieve a set of substates containing SE, and
3.         not( ∃ M ∈ mutex(n-1) : | M ∩ Y | > 1 ) then MUTEX := false
4. else MUTEX := true
5. end.
```

Figure 2: Auxiliary Procedures for the Object-Graph Algotithm

used to check groups of substate expressions. It does the checking very simply, by trying to find a set of consistent operators at the level below which add these substate expressions. Given G, the set of substate expressions making up a goal, the forward search halts when it reaches an object level level(n) such that contains(level(n),G) (Line 19 in Figure 1). ACHIEVE searches for a consistent operator set Y to achieve G, and if it finds one recursively calls itself at level(n-2) with the set of preconditions of Y as the new goals to achieve. Finally, the definition of inconsistent in Line 3 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. The current OCL-graph implementation does not memoize substate mutexes, but this is a subject for on-going research.

6 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.

6.1 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 where more than one object will be involved in the inconsistency.

7 Empirical Results

Tests have been carried out on a number of the standard 'toy' domains, such as the Rocket World and the Robot World. The tests have involved comparing times of the vanilla version of Graphplan against the OCL version. The restriction on the domains has risen partly due to the ease of automatically deducing the STRIPS operators from the OCL versions of these two domains.

The results of our tests would indicate a speed up of the algorithm in a range from as much as a factor of ten down to a factor of four. To give an indication of the improvement the following table shows the result of running the two versions on five problems in the Rocket world and three from the Robot world. In these experiments the code was run in SWI Prolog

vanilla	**ocl**
0.13	0.02
2.9	0.26
3.05	0.27
3.1	0.26
3.3	0.25
0.26	0.08
1.54	0.56
10.57	2.49

Table 1: Rocket - Robot World Timings

hosted on a Linux box. The times refer to cputime measured in seconds. Individual times for particular runs of the same task seemed to vary by plus or minus 10 percent. The figures given represent a fairly representative sample.

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, and the number of mutual exclusion relations recorded at a particular level. The figures presented in the latter tables are for a single problem in the rocket domain involving a problem solvable using only four proposition or object layers.

	vanilla		**ocl**	
Levels	**propositions**	**links**	**Substates**	**links**
1	6	n/a	4	n/a
3	16	14	12	12
5	16	28	12	24
7	20	36	16	32

Table 2: Rocket World Levels

	vanilla		**ocl**
Levels	**mutex Act**	**mutex prop**	**mutext**
3	24	36	8
5	84	10	16
7	132	10	32

Table 3: Rocket World Mutex Relations

8 Analysis

The first point to be made is to warn against putting too much stress on the timings provided. Timings are a notoriously poor way of trying to measure efficiency and may be distorted by all sorts of extraneous factors. Even though the two algorithms share a great deal of code and structure it still may be the case that a particular element of one of the algorithms is poorly implemented while the corresponding element in the algorithm we were more particularly interested in has been very efficiently coded, but these differences may not be a reflection of the underlying algorithms. Similarly the choice of examples may be biased in favour of one of the implementations. Certainly the Rocket world provided better results than the Robot world.

The test results are encouraging and suggest that there is an efficiency improvement. This is despite the fact that in some cases the graph built may 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 operators. But even in those cases occurring in the Robot world the OCL version was faster. The efficiency gain we conjecture results from our more powerful *mutexes* and a less dense graph providing fewer opportunities for backtracking and in the forward phases results from the fact that that many of the *mutexes* are implicit and hence do not require computational effort to memoize.

9 Conclusions

In this paper we have illustrated how a graph-based algorithm can benefit from an object-centred representation based on OCL. 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. Our initial experimental results appear to support this claim.

They are many avenues for future work. Most pressing is to extend our experimental results with our current domains, 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, 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 [3] 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

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

[2] M. Fox and D. Long. The Automatic Inference of State Invariants in TIM. *JAIR vol. 9*, pages 367–421, 1997.

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

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

[5] D. E. Kitchin. *Object-Centred Generative Planning*. PhD thesis, School of Computing and Mathematics, University of Huddersfield, forthcoming,1999.

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

[7] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending Planning Graphs to an ADL Subset. In *Proceedings of the 4th European Conference on Planning*, 1997.

[8] D. Liu. The OCL Language Manual. Technical report, Department of Computing Science, University of Huddersfield , 1999.

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

[10] T. L. McCluskey and J. M. Porteous. The Use of Sort Abstraction In Planning Domain Theories. In *Planning and Learning: On to Real Applications. Papers from the 1994 AAAI Fall Symposium*, number FS-94-01. Published by AAAI Press, American Association for Artificial Intelligence, ISBN 0-929280-75-X, 1995.

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

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

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