The Use of Sort Abstraction In Planning Domain Theories

T. L. McCluskey
lee@zeus.hud.ac.uk

J. M. Porteous
julie@zeus.hud.ac.uk

The School of Computing and Mathematics,
The University of Huddersfield,
Queensgate, Huddersfield HD1 3DH, UK

Abstract

This paper introduces a new planning algorithm which relies on a set of iterative macros *and* a set of necessary goal ordering constraints that are generated during a compilation phase prior to planning. The main contribution of domain compilation is to speed up planning and is achieved at the price of a well specified domain model that has been engineered into a *sort abstracted* form. This means that object classes have been identified and object states specified by means of state invariants. A typical planning state corresponds therefore to a conjunction of separate object states. Rather than being a burden for real applications, the use of a structured domain specification, and domain processing tools, is consistent with proper validation of the planning model when viewed from a software engineering perspective.

Introduction

Ideally the design of a language for encoding planning domain theories (such as TF in O-Plan (Currie & Tate 1991)) should be based solely on the needs of the domain modeller to facilitate model building, model validation and maintenance. It is well known that languages based on such design criteria are at odds with the efficiency of the resulting application, however, and the thrust of our recent research has been to investigate knowledge compilation and learning techniques to pre-process these domain theories.

Assuming that the domain theory has a finite closure one can imagine an extreme of compilation as the a priori generation of all possible problem-solution pairs. Standard toy planning worlds such as the STRIPS robot worlds (Fikes & Nilsson 1971) have around a million valid states, and therefore, depending on the form of a goal expression, up to 10^{12} problems are posable. Even if the pre-calculation of all the solutions was conceivable, it is difficult to envisage that the storage and maintenance of the resulting object file would be possible. At the other extreme of the compilation dimension lies a domain theory which is the result of some requirements capture. For the reasons described above, the domain theory tends to be in an implicit, declarative form that is not the most efficient form for input into a conventional planning engine.

This paper introduces a new planning algorithm which relies on an intermediate expression for the theory that lies between these two extremes. Generation of the new representation must be reasonably quick and storage should be equally restrained. On the one hand, the new expression should greatly improve planning performance, while on the other, the compilation process should help in the validation of the initial domain theory. To meet these requirements we have created two knowledge compilation techniques that:

(1) generate macro-operators that span the problem space of each object class in the domain theory;

(2) linearise a conjunctive goal problem by creating necessary orders among domain predicates.

These techniques help overcome two main sources of search explosion in classical planners: goal interaction and iterative application of operator sequences. The compiler assumes that the domain theory consists of an object-centred domain, where objects correspond to things being acted upon in a domain. Each object class, which we term *sort*, may have its own *state* which forms a part of the main planning state, and a set of predicate descriptors attached to it. For each sort, the compiler generates iterative macros which span the local problem space (i.e. every pair of possible *states* for an arbitrary object in the sort). During planning, the goal interaction rules generate an order for a set of outstanding goals. Macros, which are indexed according to initial and goal states for objects of each sort, are then retrieved to sequentially solve the ordered goals.

In this paper we start by outlining the language in which the planner's domain theory is posed. In the two sections following we briefly describe the two compilation techniques (the interested reader is referred to (Porteous 1993) for full details). After summarising the results of an earlier evaluation that demonstrated their potential for improving the performance of a conventional planner, we specify a novel linear planner which is specially designed to take advantage of compiled domain theories. Finally we illustrate the planner's operation with an example problem.

Domain Theory Presentation

In this paper we adhere to the classical planning paradigm, planning states being based on the closed world assumption, and planning actions based on default persistence. The planning domain is modelled by first identifying a set of sorts. A STRIPS-world, which we will use for illustration, may contain the sorts *Door*, *Box*, *Robot* and *Room*. In general, sorts consist of objects which can undergo a state change, for example a box undergoes a state change when it is moved from one room to another.

The planning domain is then specified by:

(a) a set of immutable facts relating objects of the same and different sorts (e.g. objects $room1$ and $room2$ are connected by $door12$, object $box1$ fits through $door12$);

(b) a set of state axioms for each sort. Each axiom is a list of clauses joined by an exclusive "$\vee$", defining the alternative generalised states possible for an object of the sort, and is called the *positive state invariant* for that sort. The positive state invariant for sort *Box* in our example STRIPS-world is (where *XOR* denotes exclusive $\vee$):

$$\begin{aligned}&\forall\, Bx \in Box, \exists\, Rm \in Room,\\ &\exists\, Dr \in Door, \exists\, Bx1 \in Box\\ &in\_room(Bx, Rm)\ XOR\\ &(in\_room(Bx, Rm) \wedge at\_door(Bx, Dr, Rm))\ XOR\\ &(in\_room(Bx, Rm) \wedge next\_to(Bx, Bx1))\end{aligned}$$

This asserts that what is true of an object of sort *Box* in any planning state is exactly one of the following: it can be in a *Room* and at a *Door*, in a *Room* and next to another *Box* or simply in a *Room*. A necessary condition for a planning state to be valid therefore is that every object in the state (all the boxes, robots, doors, rooms in a STRIPS-world) must be described by predicates satisfying exactly one of the clauses in its sort's invariant.

A final constraint on the domain description is that each domain predicate that is not immutable (i.e. not axiomatic of all planning states) must appear in exactly one sort's positive state invariant. Given a ground predicate, therefore, this allows us to determine both the sort and the object of that sort to which it "belongs". Hence $in\_room(box1, room1)$ belongs to sort *Box* and describes object $box1$. Conversely, given an object xs of some sort, we can *sort abstract* a planning state with respect to xs, which means returning the part of that state which describes the object's state. Generalised state expressions such as an operator's preconditions (see (d) below) can also be sort abstracted. In this case sort abstraction returns the predicates in an expression that belong to that sort.

(c) two sets of axioms which define: what cannot be true of any state, called *negative state invariants*; and logical relations between predicates. An example of a negative state invariant, linking predicates belonging to sorts *Box* and *Robot*, is:

$$\begin{aligned}&\forall\, Bx \in Box, \forall\, Rm, Rm1 \in Room\\ &\neg\,(robot\_in(Rm) \wedge robot\_near(Bx)\\ &\qquad \wedge\, in\_room(Bx, Rm1) \wedge \neg\,(Rm = Rm1))\end{aligned}$$

(d) a set of STRIPS-type operators which are defined in terms of sets of predicates representing preconditions and effects. These define the way objects change state (within the confines of the state invariants for that domain). Operators are not attached to any particular sort, however, and may affect the state of more than one sort. For example, pushing a box through a door affects the state of the box and the robot.

To sum up, a sort is a set of objects where the sort of an object defines the "behaviour" of that object (or the set of possible states of that object). Each object inherits its behaviour from its sort type, and its state may change between different instantiations of the positive state invariant's disjuncts by application of the operator schemas. This behavioural inheritance of sorts is useful because it allows us to abstract away detail by reasoning about the behaviour of a sort rather than all the objects of that sort.

We end the section by re-iterating our belief that a rich and tight domain theory such as this has two major advantages: first, it allows knowledge compilation techniques, such as those described in the next two sections, to transform the domain into an operational form. Secondly, it promotes validation of the domain theory, something that is desirable in real applications. For example, the carrying out of *proof obligations* when constructing a formal specification in software development (as detailed in (Turner & McCluskey 1994)) is analogous to checking that planning operators do not inviolate the planning invariant.

Macro Operator Generation

PROPOSE is a technique for generating macros from domain theories conforming to the constraints described above. It consists of two stages:

(1) the generation of generalised, iterative macros for each of the sorts in a planner's domain model;

(2) the unwinding of macros given specific instances of imported objects to that sort.

A specification of the first stage was given in (McCluskey & Porteous 1993), while the whole process is documented in (Porteous 1993). Each generalised macro is a sequence of operators and/or iterative operator subsequences, and is indexed by a (generalised) *task configuration* (i, g), where i and g each coincide with exactly one of the disjuncts in the positive state invariant for a particular sort. One of the 9 possible task configurations for sort *Box* is:

$(in\_room(Bx, Rm),$
$in\_room(Bx, Rm1) \wedge at\_door(Bx, Dr, Rm1))$

A macro is indexed by (i, g) if it abstractly links the two object states i and g. Such a macro for the example task configuration above is:

$[pushtodoor(Bx, Dr, Rm), [pushthrudoor($
$Bx, Dr, Rm1), pushtodoor(Bx, Dr1, Rm1)]^*]$

where the "*" means that the sub-sequence may be repeated 0,1 or more times. Each macro, indexed by (i, g) and of sort s, has 4 defining properties:

- The sort-abstracted preconditions of the first operator coincide with i. In other words, the set of precondition predicates of the first operator in the macro which belong to sort s contains the same predicates (up to variable re-naming) as i.
- The sort-abstracted positive effects of the last operator coincide with g. In other words, the set of predicates in the positive effects of the last operator of the macro which belong to sort s contain the same predicates (up to variable re-naming) as g.
- all iterative parts of macros satisfy a *loop invariant* and are *effective*. The loop invariant asserts that the positive effects of sort s of the last operator in the iterative part of the macro must coincide with the predicates of sort s in the first operators' preconditions. In other words each application of the iterative part achieves its own preconditions of sort s. The iterative component must be *effective* in the sense that it must change the truth values of predicate instances of sort s.
- the positive effects of sort s for any operator in the macro coincide with the preconditions of sort s of the next operator in the macro.

Macros are unwound by instantiating them with imported object values prior to planning, then further fixed during planning when applied to a specific object's state.

Given a specific task configuration, say

$(in\_room(box1, room6),$
$at\_door(box1, door12, room2) \wedge$
$in\_room(box1, room2))$

the set (a singleton in this case) of instantiated, unwound macros that is indexed by this task configuration in our STRIPS-world example is:

$\{[pushtodoor(box1, door56, room6), pushthrudoor($
$box1, door56, room5), pushtodoor(box1, door25,$
$room5), pushthrudoor(box1, door25, room2),$
$pushtodoor(box1, door12, room2)]\}$

Goal Order Generation

Although the macro-operator generation technique can be used in isolation as a speed-up technique for planning (as illustrated in (Porteous 1993)), its combination with a goal order generation technique is much more fruitful. PRECEDE is a process that returns a set of *necessary* orders for pairs of predicates in a given domain theory that meets the presentation criteria described above. The algorithm is based on the following observation: if the preconditions of every operator which achieves a predicate p form an inconsistent predicate set with a predicate q, then p must be achieved before q, written below as $b(p, q)$. In its most general form PRECEDE compares every pair of predicates in the domain, producing different orders taking into account different variable bindings. For the typical STRIPS-world in (Porteous 1993), the orders generated using this method are (where $Bx1$,$Rm4$ etc are all quantified over their respective sorts):

$b(in\_room(Bx1, Rm4), at\_door(Bx1, Dr2, Rm4)).$
$b(in\_room(Bx1, Rm4), next\_to(Bx1, Bx2)).$
$b(in\_room(Bx1, Rm4), next\_to(Bx2, Bx1)).$
$b(in\_room(Bx1, Rm4), robot\_at(Dr1, Rm2)).$
$b(in\_room(Bx1, Rm4), robot\_near(Bx2)).$
$b(in\_room(Bx1, Rm4), robot\_in(Rm3)).$
$b(next\_to(Bx2, Bx3), robot\_at(Dr1, Rm2)).$
$b(next\_to(Bx2, Bx3), robot\_near(Bx1)).$
$b(robot\_in(Rm3), robot\_at(Dr1, Rm3)).$
$b(robot\_in(Rm3), robot\_near(Bx2)).$
$b(open(Dr1), robot\_near(Bx2)).$
$b(closed(Dr1), robot\_near(Bx2)).$
$b(at\_door(Bx2, Dr1, Rm2), robot\_at(Dr2, Rm4)).$
$b(open(Dr1), robot\_at(Dr2, Rm2))$
$\quad \leftarrow not(Dr1 = Dr2).$
$b(closed(Dr1), robot\_at(Dr2, Rm2))$
$\quad \leftarrow not(Dr1 = Dr2).$
$b(at\_door(Bx2, Dr1, Rm2), next\_to(Bx3, Bx2))$
$\quad \leftarrow not(Bx2 = Bx3).$
$b(at\_door(Bx2, Dr1, Rm2), robot\_near(Bx3))$
$\quad \leftarrow not(Bx2 = Bx3).$
$b(at\_door(Bx2, Dr1, Rm2), robot\_in(Rm3))$
$\quad \leftarrow not(Rm2 = Rm3).$

Interestingly, the repeated generation of macros and goal orders using these techniques caused us to incrementally refine our domain theory, as early generations showed that our operators were not as required or our invariants were not complete. For example, circular PRECEDE orders led to the discovery of some subtle negative state invariants such as the following:

$\forall\ Bx, Bx1, Bx2 \in Box$
$\neg\ (next\_to(Bx2, Bx)$
$\wedge\ next(Bx2, Bx1) \wedge \neg\ (Bx = Bx1))$

that is, with our current operator set it was not possible to form a row of boxes.

Evaluation of Knowledge Compilation Techniques

PROPOSE and PRECEDE have been implemented and used to compile domain theories such as our example STRIPS-world, and also more complex theories such as a multi-agent STRIPS-world and a warehouse world. The combined compilation time (within a typical compiled Prolog environment) is less than five minutes for all domains, and the number of unwound macros generated runs typically to several hundred. The compiled domain theories were input to an adapted conventional planner, and the macros and goal orders were used to determine planning choices in preference to weak heuristics. In summary, the two techniques caused a speed-up over the basic planner averaging at between 2 and 5 times, depending on the domain. These results are fully detailed in (Porteous 1993). Kinds of domains for which our compilation techniques are *not* suitable are those where non-serialisable goal conjunctions are common, or where there is no sort structure to exploit (as in the common formulations of the Eight Puzzle (Korf 1985), for example).

A Planning Algorithm that exploits Sort Abstraction

Although the empirical results referred to above were encouraging, it was realised that a simpler planning algorithm that exploits the knowledge compilation process to the full would be more appropriate. Below we outline such a recursive, non-deterministic algorithm called **plan**, and to demonstrate its use of sort abstraction we follow it with a worked example.

plan(G,I, $SOLN$)

Input: G, the goal state, and I, a valid initial state, both sets representing conjunctions of ground predicates.
Output: $SOLN$, a sequence of operators which achieve G from I.

1. IF $G = \{\ \}$ THEN exit with $SOLN := [\,]$;
2. Sort G into a linear order which conforms to the PRECEDE-derived necessary orders;
3. Determine the sort s to which the first goal predicate in G belongs, and the object xs it describes;
4. Sort abstract I and G with respect to s and xs to obtain the abstracted initial state i and abstracted goal g;
5. Form M, the set of instantiated PROPOSE-generated macros which are indexed by task configuration (i,g);
6. IF there exists a macro m in M which is applicable to I THEN
   (a) apply m to I to obtain $Iadvanced$;
   (b) $Gnew := G - Iadvanced$;
   (c) plan($Gnew$, $Iadvanced$, $SOLN1$);
   (d) $SOLN := m \mathbin{++} SOLN1$;
   ELSE
   (e) Choose a macro m from M;
   (f) Wp := weakest precondition of m with respect to g ;
   (g) Gnew := Wp – I;
   (h) plan($Gnew$, I, $SOLN1$);
   (i) $Iadvanced$:= apply $SOLN1$ to I;
   (j) $Iadvanced1$:= apply m to $Iadvanced$;
   (k) $Gnew1 := G - Iadvanced1$;
   (l) plan($Gnew1$,$Iadvanced1$, $SOLN2$);
   (m) $SOLN := SOLN1 \mathbin{++} m \mathbin{++} SOLN2$;

end plan

Key "A ++ B" appends list A with list B while "S – T" returns the set difference of S and T.

Example To illustrate the algorithm we will work through an example planning problem with planning goal G as:

$$\{robot\_near(box3, room6), at\_door(box1, door12, room2)\}$$

and the initial state I as:

$$\{in\_room(box1, room6), robot\_in(room6), robot\_at(door56, room6), open(door25), closed(door56), \ldots\}$$

Step 2 of *plan* linearises G in an order that conforms to the PRECEDE orders, to form:

$$[at\_door(box1, door12, room2), robot\_near(box3, room6)]$$

Step 3 determines s (the sort of the first goal in G) and xs (the object which it describes) as Box and $box1$ respectively. Then step 4 uses s and xs to obtain the abstracted goal state g and the abstracted initial state i for the first goal in G. These are:

$$g : at\_door(box1, door12, room2) \wedge in\_room(box1, room2)$$
$$i : in\_room(box1, room6)$$

Step 5 instantiates the macros M that transform i into g (these were used earlier in the paper to illustrate macro operator generation). There is only one macro in M and it cannot be "applied" to I (for example precondition $open(door56)$ is not achieved for $pushthrudoor(box1, door56, room5)$), so the next step of **plan** is 6(f), which identifies the weakest precondition of the macro as:

$$\{robot\_in(room6), in\_room(box1, room6), robot\_near(box1), open(door56), open(door25)\}$$

which in turn are used to identify $Gnew$ as:

$$\{robot\_near(box1), open(door56)\}$$

Step 6(h) is a recursive call to **plan** with $Gnew$ and initial state I. Within this recursive call to **plan**, step 2 orders $open(door56)$ before $robot\_near(box1)$. Then s and xs are identified as $Door$ and $door56$ respectively for $open(door56)$ and the sort abstracted goal state g and initial state i are identified as:

$g : open(door56)$
$i : closed(door56)$

The only macro for this task configuration is $[open(door56)]$ and this is applied to I to form $Iadvanced$:

$$\{in\_room(box1, room6), robot\_in(room6), robot\_at(door56, room6), open(door25), open(door56), \ldots\}$$

The difference between the input set of goals and the new advanced state is $robot\_near(box1)$ and this goal along with $Iadvanced$ forms the input to a second recursive call to **plan**. This call exits with $SOLN1$ as $[goto(box1)]$ and hence the first recursive call of **plan** exits with $SOLN$ as:

$$[open(door56), goto(box1)]$$

Step 6(i) applies this solution to I and then step 6(j) applies the unwound macro m to get $Iadvanced1$ as follows:

$$\{robot\_near(box1), open(door25), open(door56), in\_room(box1, room2), at\_door(box1, door12, room2), ..\}$$

Finally, in step 6(l), **plan** is invoked again with the remaining top-level goal and the advanced planning state. This goal can be directly achieved using the following PROPOSE-generated macro:

$$[gotodoor(door25, room2), gothrudoor(door25, room5), gotodoor(door56, room5), gothrudoor(door56, room6), goto(box3, room6)]$$

and hence the solution to the original problem is formed by appending these solutions together.

Conclusions and Future Work

In this paper we have argued that a well specified planning domain model is important for real planning domains since it (a) allows knowledge compilation approaches to planning speed-up to re-represent the domain at a more efficient level (b) promotes the validation of the domain by the domain modeller. We have introduced a new idea, called sort abstraction, which essentially encourages the definition of domains in terms of different sorts of objects, such that a planning state is in fact an amalgam of object states. Given this type of domain model, we outlined the PROPOSE technique which produces macro operators that are solutions to sort abstracted problems. Given a compilation method which produces a linearization of goals such as PRECEDE, we finally presented an algorithm which exploits these assumptions.

Our future work will centre around the empirical testing of our planning algorithm. Early tests point to much larger speed-ups than we previously discovered when using the output of a compilation process with an adapted planner, but they have also shown up some problems:

(1) Exceptional objects in a sort cause problems. For example, in our STRIPS-world we have an odd box called big_box which does not fit through every door, and in its case retrieved macros fail.

(2) Our algorithm does not provide an optimal solution for non-linear goal problems. In this case, macros would have to be merged in non-trivial ways. We expect future developments of our algorithm to include plan merging techniques such as those in (Yang, Nau, & Hendler 1990).

References

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

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2.

Korf, R. E. 1985. Macro-operators: A weak method for learning. *Artificial Intelligence* 26(1).

McCluskey, T. L., and Porteous, J. M. 1993. Two Complementary Techniques in Knowledge Compilation for Planning. In *Proceedings of the ML93 Workshop on Knowledge Compilation and Speedup Learning.*

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

Turner, J., and McCluskey, T. L. 1994. *The Construction of Formal Specifications: An Introduction to the Model Based and Algebraic Approaches.* McGraw-Hill, London.

Yang, Q.; Nau, D. S.; and Hendler, J. 1990. Merging Separately Generated Plans with Restricted Interactions. *Computational Intelligence* 8:648 – 676.