# Object-Centred Planning: Lifting Classical Planning from the Literal Level to the Object Level

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

D.E.Kitchin
scomdk@zeus.hud.ac.uk

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

The School of Computing and Mathematics,
The University of Huddersfield,
Huddersfield, UK

## Abstract

*A great deal of emphasis in classical AI planning research has been placed on search-control issues in plan generation, while the issue of knowledge representation and acquisition of models for use with classical planning engines has been largely ignored. Work in knowledge-based planning, on the other hand, is often associated with 'scruffy' AI, there being no standard representation languages with associated formal semantics for encoding domain models. In this paper we describe a method to create a planning domain model which preserves the domain independence, generality and 'clean' properties of generative planners to which the model can be attached. Our method is based on lifting the level of domain representation from the literal-centred, to the object-centred. This object-centred method has the advantage that it naturally allows for the creation of a supporting tools environment to help in (i) the creation and validation of a precise planning model, and (ii) the speed-up of plan generation.*

## 1 Introduction

Research into classical planning systems in Artificial Intelligence has for decades concentrated on theoretical issues of planning algorithms. Recent work has concentrated on, for example, the relative performance of total and partial order planners [2], the inherent computational complexity of plan generation [3], extending the expressiveness of the classical model [6], and general, theoretical frameworks for planning engines [11]. This research is dominated, however, by the use of the literal or proposition as the basic level of representation, and operators representing actions, as the basic knowledge structure. Although *environmental* assumptions such as default persistence, instantaneous deterministic actions and the closed world characterise the classical approach, in general little commitment to a more elaborate *knowledge structure* is made.

Knowledge-based planning is often associated with those systems such as SIPE [23] and OPLAN [5] which have usefully been applied to realistic applications. The approach to the encoding of knowledge in these systems, however, appears to have been less than systematic. In [5], for example, there is little material on how the domain models were engineered or debugged. Modelling the complex requirements of a planning task is akin to requirements acquisition in software engineering, and it appears that few of the issues surrounding this have been tackled. From another perspective, there is also the need to fill the gap between theoretically clean research on the one hand, and practical applications of planning on the other [18, pp 13], [7]. We see development of knowledge representational frameworks in which to site 'clean' planners essential to this development.

There has been some attempts to take an 'object view' of planning, but these have been aimed at the architectural rather than the conceptual level [4, 8]. An exception is in the work of Jonsson and Backstrom [10], who point out that many types of planning problem are highly structured and planners should be able to take advantage of this to improve their efficiency. Their work exploits the inherent structure in problems using an analysis of structural restrictions in a state-transition graph induced by the operators. Although the authors exploit domain structure to make planning tractable, their work is not aimed at domain engineering issues, and their representation proposition-centred rather than object-centred.

Here we describe our method to engineer planning domains that has been tested in a range of classical domains. It draws on ideas from other areas of computing, and is inspired particularly from our work in requirements capture

[17] and formal methods [21]. The method's benefits include (i) it can be used to *systematically* create models for classical planners, (ii) it produces a *tight* model (the property of a valid state is well defined, and operators can be checked for their operational consistency), (iii) it forces attention on the semantic level of object rather than the literal (iv) we have empirical evidence that the model compilation tools associated with the method produce a huge speed-up in plan generation.

## 2 A tool-supported method for encoding classical planning domains

The method we propose for encoding domains for classical planners is described as a series of steps in the subsections below. We use a non-trivial multi-robot world as a motivating example, which should be familiar to the reader as it is an elaborate form of the much-quoted STRIPS-worlds [20]. The actual (imagined) reality is denoted $DR^3$, and the symbolic model we create denoted $R^3$.

Note that the divisions between the method's steps are quite loose and at any step, should new requirements be discovered or bugs in the initial domain encoding uncovered, the process would loop back to an earlier step. At each step, tools to help in construction or consistency cross-checking can be used. In the final step compilation tools are used to provide a greater degree of validation and to produce an efficient planning application. It is only after the final step that the planning domain would be ready to attach to a planner for dynamic testing involving plan generation.

### 2.1 Step 1: Initial Domain Description

Initial requirements for the domain are described, in natural language and diagrammatic form. These should outline the main features (central abstractions) of the domain, the kind of problems to be solved and the ways that this can be brought about. A first attempt at a natural language description of $DR^3$ is given below:
*$DR^3$, a multi robot world:*
*'There exists a world, illustrated in figure 1, with seven rooms connected by various doors, with three robots (tom, dick and harry) capable of opening doors and pushing any of six boxes through doors, into rooms and next to other objects. Doors can be locked or unlocked, and keys must be carried around to unlock locked doors or to lock unlocked doors. Each room has a light which must be turned on by a robot before keys can be found in it, and keys are colour coded to determine which doors they open. Changes to the world are brought about only by the actions of the three robots. The planner is used to generate controlling plans so that all robots contribute to producing a desired state of the world from an initial state.'*

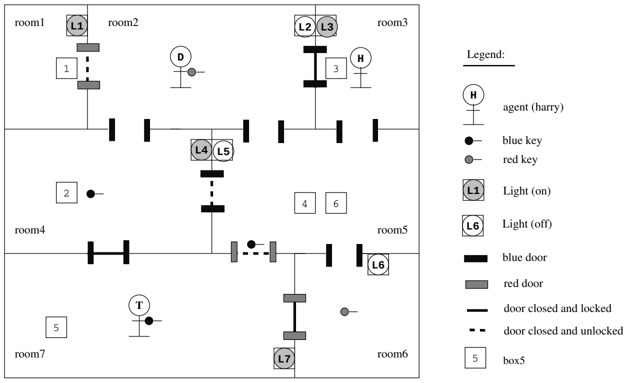


**Figure 1. $DR^3$: A multi-Robot World**

An important issue is to focus on the *use* of the model and to analyse the portion of reality that is being modelled at an appropriate level of detail. For example, in $DR^3$ we are interested in plans for moving boxes and robots to various locations within the world, and moving the colour coded keys between rooms so that doors can be locked, unlocked or opened and closed. A useful strategy at this stage (which is supplied amongst the guidelines for modelling domains for the PRODIGY planning system [9]) is to try and formulate example problems and descriptions of their possible solutions and to consider how you would teach someone to perform the task.

Translating such an outline requirements specification into a precise model will throw up many questions regarding the model, and these will need to be answered and explicitly encoded as part of the domain model.

### 2.2 Step 2: Sort Identification

The domain modeller(s) identifies the set of object classes that reflect the 'natural' groupings in the domain, and that seem appropriate for knowledge elicitation and validation. We term these object classes *sorts*. From $DR^3$ we identified the following sorts:

$$\{Room, Door, Box, Robot, Arm, Key, Light, Colour\}$$

By convention, sort names start with a capital letter. Each of the 8 sorts contained objects instances:

$$\begin{aligned} Robot &= \{tom, dick, harry\} \\ Room &= \{room1, room2, room3, room4, room5, \\ &\qquad room6, room7\} \\ Box &= \{box1, box2, box3, box4, box5, box6\} \\ Door &= \{door12, door23, door24, ...\} \end{aligned}$$

and so on, totaling 41 object instances altogether.

The main characteristic of sorts, on which our methodology is based, is that each instance of a dynamic sort (for example $box1$, $tom$, $door23$) has its own *local state*, defined by a set of predicates, which can be changed by actions in the domain. To distinguish a local state for some sort from a planning state, we will refer to these local states as *substates*. A complete, valid 'planning state' is then a conjunction of every object's substate which conforms to the invariants acquired by the step in section 2.5 (precise definitions of our framework appear in [15]).

### 2.3 Step 3: Identification of Relationships and Properties

At this stage the domain modeller specifies predicate descriptors for the verbs and verb phrases that describe relationships and properties in the domain that appear relevant to the planning task. A necessary condition on the language is that for every task that the planner might be asked to carry out, it should be possible to describe it in terms of these relationships and properties. In $R^3$ for the sort *Door* the following properties are suggested:

$$opened(Door), closed(Door),$$
$$locked(Door), unlocked(Door)$$

where we have used the sort name to give a 'type' to slots of the predicates. The sort *Door* may also be included in a number of relationships, for example it appears in the static predicate $connect(Room, Room, Door)$. All predicate descriptors are *owned* by exactly one Sort. Ownership of a predicate is a matter of deciding which is the 'key' sort in the predicate. Sometimes this is obvious, but in general it is not always possible until we consider state transitions of sorts (section 2.4). We model ownership as a mapping between predicate names and Sorts, e.g.

$$Owns^{-1}(Door) = \{opened, closed, locked, unlocked\}$$

Altogether, $R^3$ contains 24 predicate structures amounting to potentially 2035 predicate instances.

### 2.4 Step 4: State Transitions

For each sort in the domain that has a changeable state under the effect of actions, the range of substates that actual instances of that sort can occupy are specified. The task of the domain modeller at this step is then to specify a state transition diagram for each sort in the domain, where each node in the diagram represents a substate class - a disjoint set of substates.

Consideration of the *Door* sort suggests a range of three different substate classes that any door can be in, as follows:

$$\{open(Door), unlocked(Door)\},$$

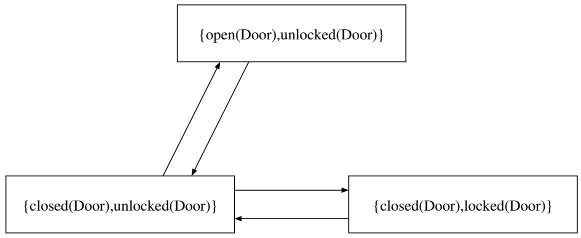


**Figure 2. State Transition Diagram for sort** *Door*

$$\{closed(Door), unlocked(Door)\},$$
$$\{closed(Door), locked(Door)\}$$

reflecting the fact that a door can be closed and locked or closed and unlocked or open and unlocked, and implicitly recording that a door may not be open and locked at the same time.

The state transition diagram for the sort *Door* in $R^3$ is shown in figure 2. Nodes in the figure represent the substate classes that instances of sort *Door* can occupy, and each node records a necessary condition for objects of sort *Door* to be in a valid substate. The diagram also tells us that in any valid state of the planning domain, each instance of the sort *Door* must be in *exactly* one of the substates.

The arcs between nodes in the diagram represent the ways that the substate of the sort can change. For example, there is no direct way to change the substate of a *Door* directly from opened and unlocked to closed and locked, the only 'route' is via the intermediate state of closed and unlocked.

The final specification of sort ownership and the division of sorts into static and dynamic can now be made:

- a sort which has no changing state, and therefore has a trivial transition diagram, is considered *static*. All other sorts are *dynamic*.
- a sort of interest *owns* all predicates that (a) make up the descriptions of substates of a typical object of the sort, and (b) change their truth value between some state transitions (these we call *dynamic* predicates).

In other words, dynamic predicates must be used in exactly one sort's state transition diagram, and a static sort owns no predicates whose truth value may change (similar distinctions are made in the work of Lifschitz [13] and Jonsson and Backstrom [10]).

In $R^3$ we built 6 state transition diagrams for the following dynamic sorts $\{Door, Box, Robot, Arm, Key, Light\}$, the other two sorts being static: $\{Room, Colour\}$. The transition diagram for the *Box* sort is slightly more complex than figure 2 and is shown in figure 3.

## 2.5 Step 5: State Invariant Construction

The modeller next considers the ways that different sorts can interact in order to build up a set of logical invariants for the model. These state invariants are analogous to those used in model-based formal specifications of software (e.g. as in VDM [21]). The availability of invariants promotes the effectiveness of tools to support in validation and plan generation speed-up. In our method the types of invariants are:

- atomic invariants: these are instances of predicates which are deemed always true in the model. All other instances of these static predicates are assumed false.
- intra-sort invariants: these are formulae containing predicates owned by one sort only;
- inter-sort invariants: these are formulae containing predicates owned by more than one sort;
- negative invariants: these are formulae that must evaluate to *false* in every valid state of the world;
- positive invariants: these are formulae that must evaluate to *true* in every valid state of the planning world.

Of the 2035 predicate instances in $R^3$, about half are static, of which 42 instances were deemed always true and hence form the atomic invariants (containing for example instances of the *connect* predicate). For each sort, the positive intra-sort invariant should correspond exactly to the transition diagrams that were identified in step 4. Thus each predicate expression describing a node in the diagram makes up each exclusive clause in the positive invariant.

A negative intra-sort invariant (negative invariants are also termed 'inconsistency constraints') for sort *Arm*, which was not explicit in the initial requirements, is the constraint that no two arms can hold the same key:

$$\forall A_1, A_2 \in Arm, \forall K \in Key : \\ inconsistent\_constraint(arm\_used(A_1, K), \\ arm\_used(A_2, K), A_1 \neq A_2).$$

A negative inter sort invariant that records that a robot must be in the same room as a box it is near, is as follows:

$$\forall A \in Robot, \forall B \in Box, \forall R_1, R_2 \in Room : \\ inconsistent\_constraint(robot\_in(A, R_1), \\ robot\_next\_box(A, B), box\_in(B, R_2), R_1 \neq R_2)$$

This is an 'inter' because the first two predicates are owned by the *Robot* sort whereas the *box_in* predicate is owned by the *Box* sort. Finally, the modeller considers the ways that different predicates in the domain are logically

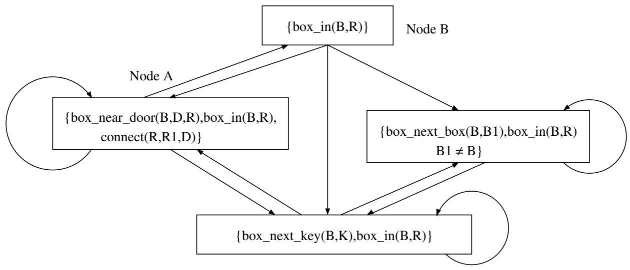


**Figure 3. State Transition Diagram for sort** *Box*

related and formulates a positive inter-sort invariant for the domain, in terms of a set of rules that logically relate predicates.

An important question is when does the domain modeller know that the invariant contains 'enough' information? For the positive intra-sort invariant this is straightforward since for each dynamic sort that has been identified the invariant must specify all of the substates that instances of that sort can occupy. However, we know of no such completeness criteria for negative invariants, although a level of security is reached when the tools described below check and compile the model without errors.

## 2.6 Step 6: Operator Specification

In this step parameterised operators that model the effect of actions are specified in terms of the way they affect classes of substates. In the natural language description of $DR^3$ verbs and verb phrases not only suggest relationships and properties of the domain, and possible substates of sorts, *but also* ways in which these substates can be changed. For example, the possible ways of changing the substate of an object of sort *Door* are to perform any of the following: *lock, unlock, open, close*. Operators are declaratively-defined but in contrast to classical planning they are specified in terms of the substate transitions of sorts. We identify which operators are needed using the state transition diagrams from the step above. The label of an arc can be used as an operator's identifier, and the description of the nodes joined by the arc can initiate the state change definition of the operator.

Clearly operators change the substate of the sort whose state transition diagram they appear on as arc labels, but the domain modeller needs to also consider any other sorts that the operator may effect, along with any other sorts that are involved. Operators are decomposed into the following components:

1. **Prevail Conditions**: conditions on substates that need to be true before an action can execute, *and* are unchanged by it.

**Name**: pushthrudoor
**Prevail conditions** for objects: $Door_1, Arm_1$

$\{open(Door_1), unlocked(Door_1)\}$
$\{arm\_empty(Arm_1, Robot_1)\}$

**Necessary state changes** for objects: $Box_1, Room_1$

$\{box\_near\_door(Box_1, Door_1, Room_1),$
$box\_in(Box_1, Room_1), connect(Room_1, Room_2, Door_2)\},$
$TO \quad \{box\_in(Box_1, Room_2)\}$
$\{robot\_in(Robot_1, Room_1), robot\_next\_box(Robot_1, Box_1)\}$
$TO \quad \{robot\_in(Robot_1, Room_2)\}$

**Conditional state changes** depending on the affected objects actually being in the state specified (for any objects $Box_2, Key_1, Robot_2, Box_2 \neq Box_1, Robot_1 \neq Robot_2$):

$\{key\_next\_box(Key_1, Box_1), key\_in(Key_1, Room_1)\}$
$TO \quad \{key\_in(Key_1, Room_1)\}$
$\{box\_next\_box(Box_2, Box_1), box\_in(Box_2, Room_1)\}$
$TO \quad \{box\_in(Box_1, Room_1)\}$
$\{robot\_in(Robot_2, Room_1), robot\_next\_box(Robot_2, Box_1)\}$
$TO \quad \{robot\_in(Robot, Room_1)\}$

**Figure 4. The operator Pushthrudoor**

2. **Necessary Effects**: conditions on substates that need to be true before an action can execute and are *necessarily* changed by it.
3. **Conditional Effects**: conditions on substates that *if* they were true before an action executes then they will be changed by it.

An operator *pushthrudoor*, connecting node A to node B in figure 3, is shown in figure 4. To systematically check for conditional effects, the modeller looks through each node in the sorts' transition graphs, or equivalently all the positive intra-sort invariants, which might refer to necessarily effected objects ($Box_1$, $Robot_1$) as 'foreign' sorts (i.e, all references to a Box or Robot which are not key references). At each one, the modeller must decide whether the substate can co-exist in a world state with the substates already recorded in the necessary and prevail conditions – the inconsistency constraints may be consulted to help this. If this is possible, and an object's substate may change as a side effect, then the modeller decides to which substate, and records it in the conditional effects. This kind of systematicity actually threw up an error in our initial encoding - we had left out the effect that some other robot (apart from the one pushing) underwent a state change conditional on it being next to the box being pushed.

An advantage of our method is that it delivers a tight domain model *before* on-line testing. We can use it to reason about the properties of the operators so constructed, in particular we can ask if the operator is *consistent* – if an operator is applied in a valid state of the world, is the resulting state valid? or if it is *deterministic* – if an operator is applied in a valid state of the world, then is the resulting state unique? Both these properties follow fairly easily for the example from the way we have constructed it – systematically, and using the already-defined substate definitions as a guide (a proof that *pushthrudoor* is consistent in given in [14]).

## 2.7 Tool Support

In summary, the result of using the method is to produce a *precise* specification with the following structure: a set of SORTS; a set of OBJECTS for each sort; a set of PREDICATES with slots belonging to a sort; the OWNS mapping between predicates and sorts; the ATOMIC INVARIANTS; the POSITIVE INTRA-SORT invariants defining the range of substates in each sort; the NEGATIVE invariants; the POSITIVE INTER-SORT invariants; and the OPERATORS defined in terms of substate transitions.

The specification will contain more declarative details than the initially and informally stated requirements, as the method and its tool support mechanisms uncover incompleteness. In particular for $R^3$, we had to encode what amounted to naive spatial axioms in the invariants. A full specification of $R^3$ is available via anonymous ftp from `prometheus.hud.ac.uk` in directory `pub/julie` as file `r3_world`.

We see the major benefit of formalising planning domains as the ability to support the whole process with tools. Tools all help to some degree in the crucial aspect of *validation and verification*, and include:
1. syntax and sort cross-checking of predicates
2. checking operator consistency
3. graphical editing, for example in the construction of state transition diagrams
4. partial construction of operators from diagrams
5. generating inconsistency constraints from operator definitions
6. generating random valid world states and goal conditions, and hence random planning tasks
7. tools for use in compiling the final model into a more efficient or operational form. These include tools for macro generation [14] and goal-order generation [16].

At present our environment contains tools of type 1,5,6 and 7. The functions of tools 2, 3 and 4 are performed by hand.

The compilation tools that are used in the method (forming 'step 7') are described in a little more detail in the next section.

## 3 Evaluation of the Method

We now discuss the results of engineering the $R^3$ model using our tool-supported, object-centred approach. In [15] we showed some preliminary results with an extended STRIPS-worlds and tyre-change world and we are soon to produce results using a warehouse world [12].

**Size of $R^3$:** The specification of $R^3$ (used in the tests below) contains 25 operator schema, equating to approximately 1,000 ground operator instantiations. It contains 80 negative invariants. 8 sorts, containing 41 objects in total, were used, the 6 dynamic sorts containing 32 of these objects. There were 20 substate classes in total with each sort's transition diagram having between two and five nodes. The number of world states is difficult to calculate accurately, given the number of invariants, but with 32 objects with varying substates being describable, the resulting number of permutations would be very large indeed.

**Development of $R^3$:** We looped through the steps in the method several times. For example, our initial hand encoding, and use of tool 5, produced 55 negative invariants. Use of the goal-order generation tool in the compilation step subsequently led to the discovery of several new invariants, as the production of cyclic orders indicated a lack of an invariant *or* an error in the encoding of the operator set. Inclusion of the new invariants (or debugged operators as appropriate) led to more cyclic orders output from the goal-order generation tool - and so on until our model settled on a final figure of 80 negative invariants. Checking operators preserve invariants also throws up errors, as remarked above in section 2.6.

**Use of Compilation Tools with $R^3$:** Whereas the discussion of $R^3$'s development might be considered anecdotal, we can give more objective evidence of the efficacy of a certain type of tool - those that compile the domain model. The premise of using tools to compile domain models is that the input model is not in an *operational* form, but written in a language designed to satisfy criteria to do with readability and naturalness of presentation. The function then, of compilation, as we see it, is twofold: (i) to improve the efficiency of the representation of the model, when used with a planning engine, and (ii) to throw up any inconsistencies uncovered between components of the model.

The same planner is used in each one of our tests: a conceptually simple, backward-chaining total-order planner called FMD. Steps 1 - 6 produce a model that could in fact be used with a range of classical planning methods although they would need to be adapted to take advantage of the object-level approach. FMD works on goals independently, advancing the plan state after partial solutions are found. So that we could conduct comparative experiments with uncompiled models, we coded the planner with loop detection heuristics in both goal achievement and goal back-chaining. This makes little difference when the domains we chose are input compiled but in contrast it has to be included to make at least some tasks tractable when the planner uses the uncompiled models. For our planning configuration, the compilation tools used are:

- a goal-order generation tool. This tool analyses the domain model, in particular using the negative state invariants, and creates orders on domain predicates which can be used to sequence conjunctive goals. For example, consider the goal conjunction:

  $$(\{robot\_in(dick, room1), robot\_next\_key(dick, key2)\}$$
  $$\{box\_next\_door(box2, door56, room5),$$
  $$box\_in(box2, room5)\}, \{locked(door45)\})$$

  which represents the conjunction of three object's substates (dick, box2 and door45). Goal orders are automatically produced which determine the order in which the object substates should be achieved without introducing the possibility of goal interactions (in this case *box*2's goal would be attempted first). The tool works by analysing domain operators and invariants, and building a partial order on predicates (our approach to this is detailed in [16]).

- a macro creation tool. This uses the object-centred approach to build a macro table (macros are valid sequences of operators) before the planner is used. The macros span the space of every pair of substate classes. During planning, therefore, the usual idea of achieving a literal at a time, by backchaining (as typified in the kind of framework summarised by Weld [22]) if an operator cannot be executed to solve a goal, is replaced by planning at the *object* level. From a set of outstanding goals, the goal ordering tool is used to choose which object's substate ($o_g$) to solve first. This done, the object's initial substate ($o_i$) is retrieved from the current state of the planning world. Finally, a macro(s) is retrieved from the table indexed by ($o_i$, $o_g$). This technique originated from the PROPOSE tool detailed in [19].

Here we show the planning speed-up by comparing the compiled $R^3$ (i.e. after tools have created a macro table and a goal ordering list) with an uncompiled version. Compilation of $R^3$ produced 70 goal orders and 2385 macros (compilation of this model took around an hour on a Sun Sparc workstation).

Figures 5 and 6 show the effects of planning speed-up, in the form of graphs showing the number of problems solved against CPU limit or node limit. Nodes are counted as the number of 'back-chainings' performed in goal directed

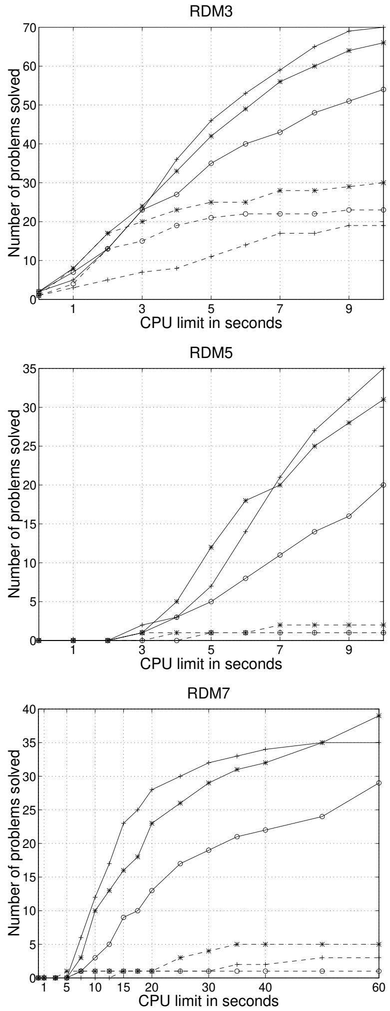


**Figure 5. CPU: $R^3$ world**

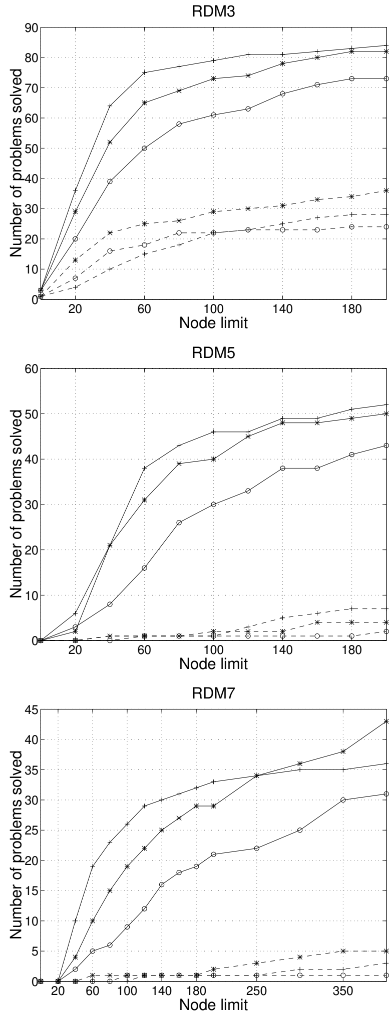


**Figure 6. NODES: $R^3$ world**

search. Three different search strategies were used: **DS**: a depth first search strategy, **BS**: a breadth first search strategy, and **HS**: a heuristic search strategy that favoured the expansion of nodes with singleton goal sets and non-empty partial solutions.

Using tool 6. above, 3 lots of 100 randomly generated goal sets were input to the planner: RDM3 contains sets of three predicates, RDM5 sets of five and RDM7 sets of seven. The predicate sets did not always form unique object substates, which meant that macro usage was restricted in the compiled model. Nevertheless the results are strikingly superior with the compiled models. An interesting outcome with the $R^3$ tests was that our heuristic search (HS) strategy, which was highly successful in other domain models, performed relatively poorly in all the configurations. This, of course, is to be expected of a 'weak' heuristic, in contrast to compilation tools, which can be viewed as producing 'strong' heuristics for each domain model that is input to them.

## 4 Conclusions and Future Work

We have described a method to create planning domain models which is based around an object-centred approach to encoding classical planning worlds. While preserving the domain independence of classical planning, it encourages the use of a tool-based, systematic approach to model creation and debugging. It is the notion of sort and the presence of the state invariant which supplies the basis for effective support tools. We have shown how this method has been used to encode a non-trivial classical planning domain model, and finally, we have described results which suggest that compilation tools deliver a sustained plan generation speed-up over uncompiled models.

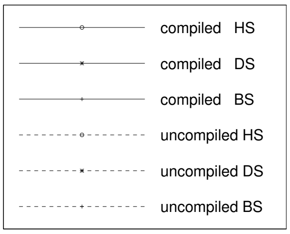


**Figure 7. Legend for Results Graphs**

We have deliberately scoped our planning model language to be able to capture the kinds of domain used in classical planning. To keep the presentation simple, we have sidestepped some issues such as *sort hierarchies*, and *exceptions* in sorts. A natural extension of our work is to loosen the strong typing on predicates slots and allow slots to be typed by 'super-sorts' - such as 'physical-sort' or 'pushable-sort'. In the future work we hope also to investigate another extension to non-determinism by relaxing the constraint on valid states that every object must be in exactly *one* of its substates.

## References

[1] P. M. Allen and L. T. Semmens. Integration of structured methods and formal notations. In *Proc. of the 4th International Conference on Information Systems Development (ISD 1994)*, 1994.

[2] A. Barrett and D. S. Weld. Partial-Order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67, 1994.

[3] T. Bylander. Complexity Results for Planning. In *Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[4] A. Chang, P. Kannan, and B. Wong. Design of an object-oriented system for manufacturing planning and control. In *Proceedings of the Rensselaer's 2nd International Conference on Computer Integrated Manufacturing*, 1991.

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

[6] O. Etzioni, K. Golden, and D. Weld. Tractable Closed World Reasoning with Updates. In *Proceedings Knowledge Representation Conference*, 1994.

[7] Y. Gil, M. Veloso, S. Chien, D. McDermott, and D. Nau. Symposium Preface. 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.

[8] D. G. Green and M. Todd. Object-oriented approach to robotic motion. In *IEEE SOUTHEASTCON*, 1993.

[9] T. P. R. Group. PRODIGY 4.0: The Manual and Tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.

[10] P. Jonsson and C. Bäckström. Tractable Planning with State Variables by Exploiting Structural Restrictions. In *Twelfth National Conference on Artificial Intelligence*, 1994.

[11] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as Refinement Search: A Unified framework for evaluating design tradeoffs in partial order planning. *Artificial Intelligence*, 76, 1995.

[12] D. E. Kitchin. Use of an object-centred approach for the creation and validation of a warehouse planning domain model. Technical Report In Preparation, School of Computing and Maths, University of Huddersfield, 1996.

[13] V. Lifschitz. On the Semantics of STRIPS. In *Proc. 1986 workshop: Reasoning about actions and plans*, 1986.

[14] T. L. McCluskey and J. M. Porteous. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. Technical Report RR9606, School of Computing and Maths, University of Huddersfield, 1996.

[15] T. L. McCluskey and J. M. Porteous. Planning Speed-up via Domain Model Compilation. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 233–244 . IOS Press, 1996.

[16] T. L. McCluskey and J. M. Porteous. On Extracting Goal Structure from Planning Domain Specifications. In *Proceedings of the 14th Workshop of the UK Planning SIG*, Nov 1995.

[17] T. L. McCluskey, J. M. Porteous, Y. Naik, D. N. Taylor, and S. Jones. A Requirements Capture Method and its use in an Air Traffic Control Application. *Software–Practice and Experience*, 25, January 1995.

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

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

[20] E. Sacerdoti. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence*, 5, 1974.

[21] J. G. Turner and T. L. McCluskey. *The Construction of Formal Specifications: an Introduction to the Model-Based and Algebraic Approaches*. McGraw-Hill Software Engineering Series, London. ISBN 0-07-707735-0, 1994.

[22] D. S. Weld. An Introduction to Least Commitment Planning. *AI Magazine*, 1994.

[23] D. Wilkins. *Practical Planning: Extending the Classical A.I. Paradigm*. Addison-Wesley, 1988.