Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency

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,
Huddersfield, UK

September 22, 2000

Abstract

This paper postulates a rigorous method for the construction of classical planning domain models. We describe, with the help of a non-trivial example, a tool supported method for encoding such models. The method results in an 'object-centred' specification of the domain that lifts the representation from the level of the literal to the level of the object. Thus, for example, operators are defined in terms of how they change the state of objects, and planning states are defined as amalgams of the objects' states. The method features two classes of tools: for initial capture and validation of the domain model; and for operationalising the domain model (a process we call compilation) for later planning. Here we focus on compilation tools used to generate macros and goal orders to be utilised at plan generation time. We describe them in depth, and evaluate empirically their combined benefits in plan-generation speed-up.

The method's main benefit is in helping the modeller to produce a tight, valid and operational domain model. It also has the potential benefits of (i) forcing a change of emphasis in classical planning research to encompass knowledge-based aspects of target planning domains in a systematic manner, (ii) helping to bridge the gap between the research area of theoretical but unrealistic planning on the one hand, and 'scruffy' but real-world planning on the other, (iii) a commitment to a knowledge representation form which allows powerful techniques for planning domain model validation and planning algorithm speed-up can be bound up into a tool-supported environment.

1 Introduction

1.1 The Problems of 'Knowledge Sparse' Planning

Research into classical planning 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 order vs partial order planners [3, 57, 44], the inherent computational complexity of plan generation [6, 23], extending the expressiveness of the classical model [25], and general, theoretical frameworks for planning engines [31]. This research has been dominated by the use of the literal or proposition as the basic level of representation, and operators representing actions, as the basic knowledge structure, containing formulae made up of these literals. Although *environmental* assumptions such as default persistence, instantaneous deterministic action and the closed world characterise the classical approach, in general little commitment to a more elaborate *knowledge structure* has been made. In this paper we argue that research issues in classical planning should not be considered in isolation of knowledge acquisition and representation and, to facilitate this, there should be a move away from the syntactic representational primitive of the literal, to the semantic level of the 'object'. Several lines of argument lead us to believe that knowledge representational issues have to be taken into account when making claims about Planning and Planners.

Firstly, consider the recent trend in AI Planning research to analyse the computational properties of different variations of the classical generative planner and to compare the efficiency tradeoffs between linear and partial-order planners. Initial results with systematic causal link partial-order planners suggested that they were more efficient than linear planners, in part as a result of reducing redundancy [39, 3]. But these results have been called into question, as fixed planning strategies can give wildly varying relative performance over a number of different planning domains. In a similar vein, some researchers have concluded that we are asking the wrong question: rather than ponder over which is preferable, total or partial-order, we should concentrate on where it is best to use a particular strategy. So rather than match different planners off against each other, research should focus on control strategies for hybrid planners [32] or different domain independent heuristics for planning in different problem domains [54]. We conclude that even in the range of domains open to classical planning, the choice of optimal research strategy appears domain-dependent, and so we need frameworks for domain classification and mechanisms to take advantage of domain encodings.

Secondly, there is a need to fill the gap between theoretically clean research and practical applications of planning [42], [26]. Researchers who are exploring the abstract features of planning necessarily use simple domains to facilitate reasoning about search intensive issues. On the other hand real-world planning requires teams of both knowledge acquisition as well as planning and software experts - a completely different 'ball game' to the research scenario, in that many non-functional requirements such as user factors (HCI, user training etc), hardware and software constraints, system response time, and reliability issues have to be considered. In making steps towards bridging this gap we must be careful that while moving away from the 'knowledge sparse' stance we keep within the spectrum of 'clean AI'. Hence we see a systematic approach to modelling planning domains, within a standard, broad representational framework, as a step forward.

Thirdly, work has shown how performance can vary by using a fixed planning strategy with different encodings of the *same* domain model. Some researchers have testified to the large differences in performance that can result from seemingly insignificant changes to a domain encoding. An example of this is given for the performance of the PRODIGY/EBL system in [24, p 916]. Even results favouring one planner over another using the same domain encoding may be flawed in that it is the particular encoding that is favouring a planner, rather than something intrinsic in the domain. Guidelines or frameworks for encoding domains would at least put this encoding problem into some context.

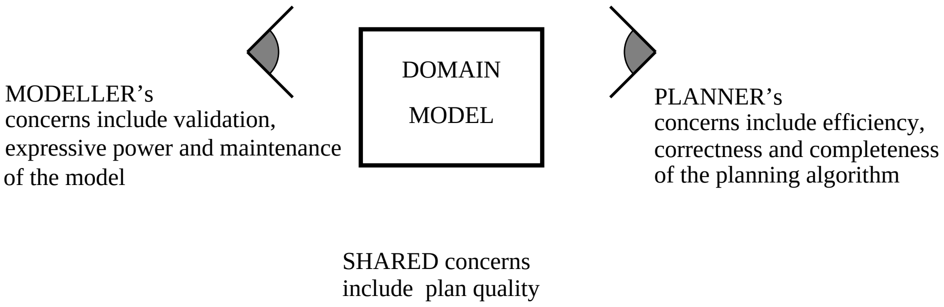

Figure 1: Views of the Domain Model

1.2 Implications of Introducing Knowledge Representation Issues

In introducing representational issues we have three phenomena to consider (i) the domain itself (a reality, or an imagined reality), (ii) the symbolic domain model, and (iii) the representation language used to encode the model. Initially creating (ii), and then debugging and validating it with respect to (i), is often as hard as debugging the planning software to be used with it. Whereas planning software is supposed to conform to a theoretical model, or an outline algorithm, the domain model is supposed to capture a piece of *reality* (block stacking, machine-shop scheduling, etc), and so, unlike software, relational criteria such as *correctness* are inappropriate for the purposes of acceptance testing. Using (iii) as a framework, one can devise guidelines and tests to capture levels of cohesion and self-consistency for domain models (rather like the concept and process of normalisation in relational databases) and in applications involving a real client one would need to have guidelines and metrics for this kind of construction. This is apparently not the case in AI planning models and their specification languages that have dominated in the literature – they appear sparse and underdeveloped[1]. For example, in [59], Weld equates a domain specification with a set of pre- and post condition operators, yet this form of definition alone leaves questions unanswered such as 'what constitutes a *valid* initial state in the domain?'. More generally, one might reasonably expect domain models (particularly those incorporating the closed world assumption) to include necessary and sufficient conditions on *any* valid state. Also, the way that domain models are constructed for the purposes of research evaluation appears ad hoc as the reasons for an encoding are rarely ever given.

In Figure 1, we show the typical concerns of the modeller as model validation, expressive power of the specification language, and ease of maintenance. Validation issues are also linked to understandability of the representation language used for interaction with the planner, a concern cited in [29]. From the planning algorithm's point of view the typical requirements of the input model are that it should lead to efficient and effective plan generation. This generally entails that it is as pre-compiled as possible, that it is in some standard, simple format, and any processing that can be separated out from the planning task has been done off line. Issues such as the understandability of the planner's input are not as important. On the other hand, although these issues can be considered in isolation from the planner, some types of requirements are difficult to separate out (plan quality, for example, relies on both

[1]There has been some work in developing 'realistic' planner representation languages (e.g. [9]) but not within the realms of AI planning, despite the move to more expressive languages such as ADL [45].

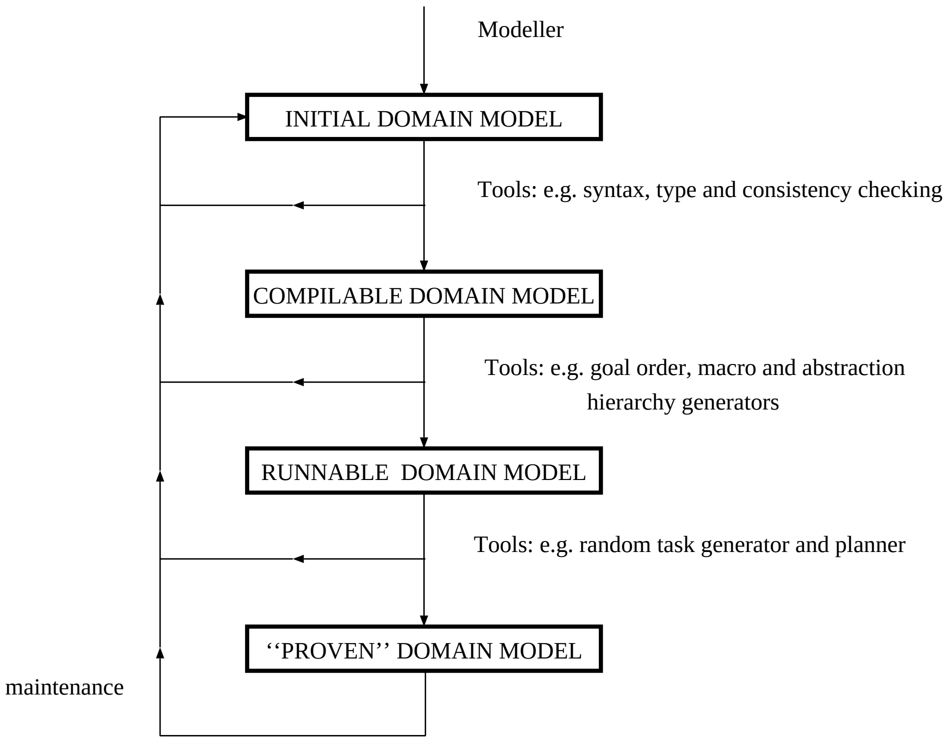

Figure 2: The development of a domain model using tool support

an accurate model and an effective planner).

Tool support associated with these activities is essential, and in Figure 2 we show where tools feature in the development of a model. At each stage, the domain model may have to be revised and tools re-applied to the changed model. In particular, tools should provide a measure of insulation between making the domain model *valid* and making the final planning system *efficient*. In this paper we emphasise the use of tool support as a form of model compilation.

Within planning, the idea of using domain engineering to tackle search problems is not new. For example, Korf's use of macro tables to solve Rubic's Cube problems features compilation producing macros, a form of domain engineering which 'unwinds' macros from the structure of the original domain encoding [37]. Also, there have been many instances of planning systems that use knowledge to prune search, but these have relied mainly on hand-coding of knowledge. The O-Plan system [16] has a rich domain specification language which features operators with causal information/precondition typing, but it is admitted by the authors that the 'actual coding up' (of operators) .. 'is a difficult job' [16, pp58]. This problem is also apparent in some hierarchical planners where, 'the designer of the problem space must manually engineer the appropriate abstractions' which is 'largely a black art' [36, pp1]. In other words, the initial domain model features domain specific heuristics encapsulated within the operator descriptions. This is certainly useful in reducing search during plan generation but the coding up and maintenance of these operators is difficult, a problem that is exacerbated by the lack of a standard representational form, and tool support for developing these operators.

1.3 A Way Forward

This paper contains the first steps towards a standard tool-supported method, and a representation language for classical planning domains, backed up by a set of standards for encodings. It shows that a rigorous method for capturing the functional requirements of classical planning domains within a domain model leads to many opportunities to improve the model with respect to its fit with the domain and its efficiency when used at plan time.

In the past, classical planning has been concerned with the level of the literal or proposition. Abstraction mechanisms have relied on them (Knoblock [36]), complexity classes rely to some extent on them (Bylander [6, 7]) and theoretical formulations rely on them (Chapman [11], Lifschitz [38]). The attraction of this is the apparent generality of the approach, yet syntactic restrictions (such as having function free terms or propositional terms) are often made. To us, the domains that classical planners are aimed at invariably involve objects which are manipulated or otherwise change their state through the process of plan execution. Further, domains tend to contain groups of objects which share common properties and behaviours: for example, STRIPS domains contain rooms, doors, boxes; job shop scheduling domains contain drills, lathes, components; warehouse worlds contain trucks, shelves and cranes etc.

To date there have been some attempts to take an 'object view' of planning but these have been largely aimed at the architectural rather than the conceptual level. For example, an object oriented approach has been used in robot planning, by Chang et al [10], Green and Todd [28] and others. In these systems the OO paradigm was used to abstract information about objects from a knowledge base into families and to pattern match unfamiliar objects with ones in the knowledge base. These approaches seem to concentrate on utilising object oriented knowledge bases rather than carrying out plan generation by exploiting object structure. In Kazi's proposed object-oriented reactive planning system [33] objects are to be represented in an increasingly specialised sequence of objects in an inheritance hierarchy and plan fragments are associated with objects. This object oriented knowledge base and the plan fragments are used in conjunction with the reactive planner. Again, this work does not seem to incorporate the object-centred approach into plan generation, but to restrict its use to the creation of abstractions and hierarchies in the knowledge base. In contrast, one of the central aims of our work is to exploit the regularity that results from adopting an object level approach to domain modelling in tackling the problem of *plan generation* in classical planning.

Overall, our approach could be summed up as providing a tool-supported method for domain modellers to engineer the capture of domain models and then use compilation tools to operationalise the model (that is translate it into a more efficient form for subsequent planning). In summary, the potential benefits of using the method are that:

- it allows domain models to be created in a systematic fashion
- tools can naturally be provided to support each step of the method
- it ensures the production of a *tight* domain model e.g. the modeller must define the property of a valid state for a given domain and this can then be used to check, for example, the operational consistency of action representations
- it forces the domain modeller to focus attention on the semantic level of the object rather than the literal

- the tools help in maintenance of the model since they can be re-run whenever the model is updated
- the method represents the first steps towards a standard object-centred representational language for classical planning domains, and this promises to provide a framework for analysis of the impact of variations of domain model representation on different plan generation strategies.

In addition we have empirical evidence that the model compilation tools associated with the method are capable of producing large speed-ups in plan generation. Finally, a more general benefit of the method may stem from the cross-fertilisation of research from other areas of computing – our approach is influenced by our own work in requirements capture [41] and formal methods in software engineering [56].

This paper is organised as follows. The use of the method is communicated using a non-trivial classical planning domain in section 2. Section 3 formalises concepts and properties underlying the method and the model it produces. Section 4 discusses a range of tool support for the method and for operationalising domain models captured using the method. This latter category of tools produces macros and goal orders and is the subject of sections 4.1 and 4.2 respectively. In section 5 we evaluate the method showing empirical evidence of plan generation speed-up using a number of planning domains including the one introduced in section 2. We present a review of related work in section 6, some extensions to our approach in section 7 before summarising our work in section 8.

2 A tool-supported method for encoding classical planning domains

In overview, the method we propose for encoding domain models for classical planners, is given below. Steps 1–6 produce a 'compilable' domain model, whilst step 7 produces a 'runnable' model (see Figure 2).

1. Initial requirements for the domain are described, in natural language and diagrammatic form.
2. The domain modeller identifies an object hierarchy that reflects the natural groupings of objects in the domain, and that appear appropriate for knowledge elicitation and validation. The hierarchy is based on object classes which we call *sorts*.
3. A set of predicate descriptions denoting properties and relationships in the domain are defined.
4. For each sort in the domain whose state can be changed by the effect of an action (these are called *dynamic* sorts), the range of states that actual objects of that sort can occupy are specified. Transition diagrams for each such dynamic sort are constructed.
5. A set of state invariants are constructed. These state invariants are analogous to those used in model based formal specifications of software (e.g. as in VDM [56]). The availability of an invariant for the domain promotes the effectiveness of tools to support the validation, development and compilation of the domain model.

6. Operators that model the effect of actions are specified in terms of the way they affect states of an object class. The consistency of the operators is checked.
7. Compilation tools are used to provide a further degree of validation and to produce an efficient planning application.

Note that the divisions between the different steps are fairly loose and at any step, should new requirements be discovered or bugs in the initial domain encoding be 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. It is only after the last step that the planning domain would be ready to attach to a planner for dynamic testing involving plan generation.

Each of these steps is explained below with the help of a non-trivial multi-robot example. The domain is an elaborate form of the much-quoted STRIPS-worlds [50], and so the basic terminology should be familiar to the reader. We will call the actual (imagined) reality DR^n, and the symbolic model we create R^n.

2.1 Initial Domain Description (step 1)

For the domain being captured, the initial requirements should outline its main features (central abstractions), 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^n is given below. A diagram representing a particular configuration of the domain, chosen for illustrative purposes, is shown in figure 3.

> *'DR^n, a multi robot domain, consists of a configuration of rooms, connected by various doors, wherein 'n' robots exist. Each robot has an arm which can be used for pushing or carrying. Robots are therefore capable of opening doors, pushing objects to doors, through doors and next to other objects. Doors can be locked or unlocked with a key, 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 objects can be found in it, and keys are colour coded to determine which doors they open. Changes to the domain are brought about only by the actions of the robots. The planner is used to generate sequential controlling plans so that all robots contribute to producing a desired state of the world from an initial state. Robots should contribute fairly to plans in order to keep overall running costs down.'*

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 with respect to the use of the model. For example, in the robots' world we will be interested in plans that typically involve moving boxes and robots to various locations, 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 [48]) 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.

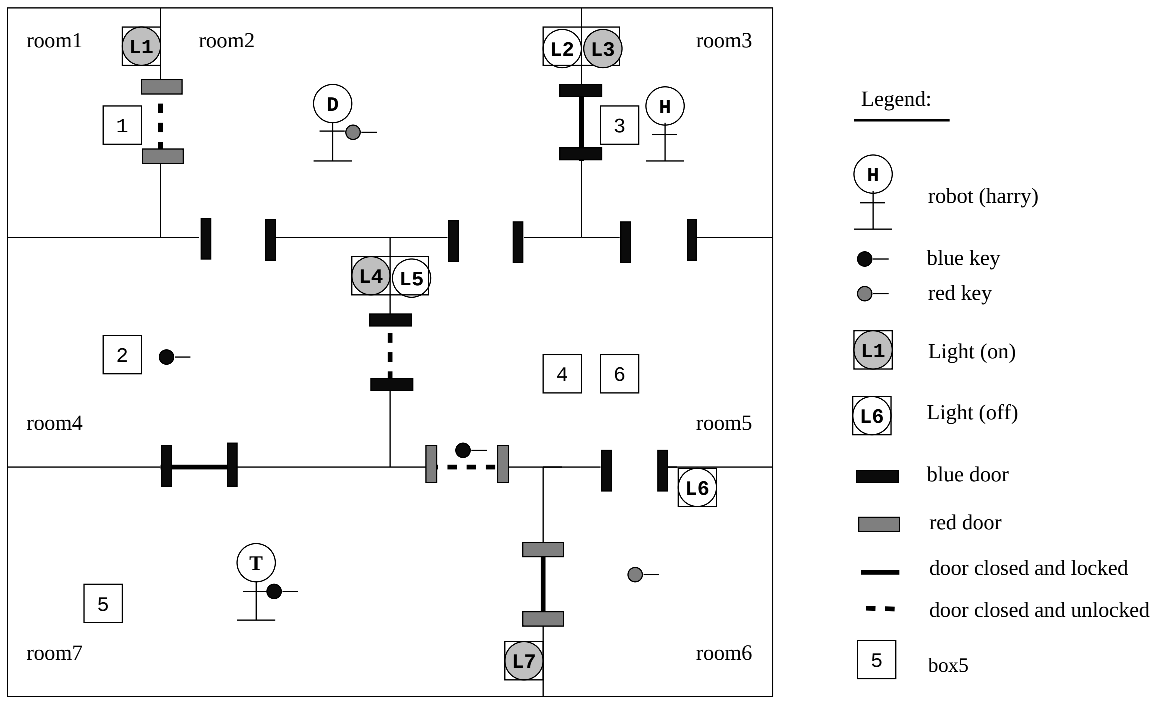

Figure 3: DR^3: An example configuration of a multi-robot domain model

Translating such an outline requirements specification into a precise model will reveal many questions regarding the model, and these will need to be answered and explicitly encoded as part of the domain model (in the operators and invariants).

2.2 Sort and Object Identification (step 2)

Nouns used in the natural language description may indicate the sorts in the domain model that is being captured. We regard them as the model's central abstractions - identifying sorts is similar to identifying classes of objects in object-oriented design methods in software engineering. Sorts themselves can be grouped hierarchically by collecting objects in primitive sorts together into a 'supersort'.

From DR^n we identified the following sorts:

$$sorts = \{Room, Door, Box, Robot, Arm, Key, Light, Colour, Movable\_obj, Physical\_obj\}$$

The first eight sorts are primitive, whereas the last two of the sorts are supersorts:

$$Movable\_obj = Box \cup Robot \cup Key$$
$$Physical\_obj = Door \cup Arm \cup Movable\_obj$$

After sorts have been identified, the object identifiers belonging to each primitive sort are recorded (for example $box1$, $key2$, $door23$). By convention, sort names and sort variables will start with a capital letter, and object identifiers will start with a lower case letter. The set of object identifiers belonging to a supersort is the union of its contained sorts. Primitive sorts are described as being *dynamic* or *static*, depending on whether their objects are deemed to change state by the effects of actions. In R^n, we chose *Room* and *Colour* to be static. All objects of a dynamic (static) sort are called *dynamic* (*static*) objects.

The main characteristic of sorts, on which our methodology is based, is that each object of a dynamic sort has its own *local state*, 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 a mapping between object identifiers and appropriate substates (described in step 4) which conforms to a set of invariants (described in step 5).

2.3 Identification of Relationships and Properties (step 3)

The verbs and verb phrases in the natural language description give some indication of the relationships and properties in the domain model that is being captured (they also suggest the states of the different sorts and the way that changes of state can be effected). 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 problem; a necessary condition on the language is that every problem that the planner might be asked to solve should be describable in terms of these relationships and properties. Considering our example domain R^n, for the sort *Door*, the following properties are suggested:

$$open, closed, locked, unlocked$$

and these may be used as predicate symbols with the single argument of sort *Door* to form the following predicate descriptors (we use the sort name to give a 'type' to slots of predicates):

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

Likewise, the following predicates give positional information on movable objects:

$$on\_floor(Movable\_obj, Room), next(Movable\_obj, Movable\_obj),$$
$$near\_door(Movable\_obj, Door, Room)$$

Predicates are thus strongly typed so that slots must take values (object identifiers) from the specified sort. We make a binary distinction between two types of predicates, those whose truth value may change during the course of planning are called *dynamic*, while the remaining predicates are called *static*. This idea is similar to Lifschitz's 'essential sentences' and 'non-essential sentences' distinction in reference [38]. It is also similar to Jonsson and Bäckström's static, reversible and irreversible distinction [30] for 'atoms' (as opposed to sorts).

Which slots are included in a predicate is to some degree determined by the objects related by the predicate, and the kinds of goals that are required to be solved by the planner. For example, the goal of getting $box1$ next to $box2$ may be specified by

$$next(box1, box2)$$

without being specific about the room they are in or whether the boxes are next to some other objects. On the other hand, a goal predicate such as:

$$near\_door(box1, door23, room2)$$

is specific in that it includes on which side of the door (that is which room) the box is to be placed. The choice of granularity of predicate appears to us to be dependent on the requirements of the target planning system.

2.4 Substates and Substate Transitions (step 4)

The state space of a planning domain is the set of all valid combinations of situations that the objects in the model can occupy. In any non-trivial model the size of the state space is astronomical, and choosing the most natural and effective state decomposition to ensure that the complexity brought about by this size is managed effectively, is at the heart of our method.

Whereas in a classical planner each state might be modelled as a predicate formula written as a set of asserted predicates under a closed world assumption, for example:

$$\{on\_floor(box1, room2), near\_door(box1, door23, room2), closed(door23), locked(door23), \\ on\_floor(harry, room3), next(harry, key3), \ldots\},$$

in our object-centred formulation a planning state is modelled as a mapping between object identifiers and *substates*. A substate is a set of ground, dynamic predicates[2] that describes the situation of the dynamic object which is mapped to it.

For example, the partial state above would be represented thus:

$$\{box1 \mapsto \{on\_floor(box1, room2), near\_door(box1, door23, room2)\}, \\ door23 \mapsto \{closed(door23), locked(door23)\}, \\ harry \mapsto \{on\_floor(harry, room3), next(harry, key3)\} \\ \ldots\}$$

Hence, the state space can be described as a space of mappings between dynamic object identifiers and all valid substates.

Determination of Substate Classes

An object identifier cannot be mapped to an arbitrary set of ground predicates; a valid substate for an object must be a member of one of the *substate classes* identified for that object's sort. A substate class is defined by a collection of predicate expressions: a substate belongs to a class if and only if it satisfies one of the expressions.

The method for designing substate classes for a primitive sort is as follows:

[2] In section 7 we explain how this representation can be naturally extended so that it can handle incomplete information about an object's substate.

1. A substate class transition diagram, made up of nodes and arcs, is drawn for that sort. Nodes represent susbstate classes, that is the *abstract* states that a typical object could be in, and arcs between nodes represent the ways that the abstract states of the sort change. The choice of nodes and arcs is determined by examining the verbs and verb phrases used in the natural language domain description, and sample diagrams of domain configurations such as figure 3.

2. Each node in the diagram is annotated with a predicate expression defining a substate class. To do this the modeller should consider a typical object, and (i) write down the conjunction of predicates that describe the object and which are true if the object occupies that substate. No dynamic predicates which primarily or exclusively describe objects of another dynamic sort should be included. (ii) Consider instantiations of the resulting predicate expression. Add static predicates, if necessary, to ensure every instantiation of the expression corresponds to a valid state of an object in the domain.

The principal role of substate classes is to group those substates together that behave the same way under a state transition brought about by an action. They also provide a means of checking the validity of a state, and later we will see how they can provide the basis for generation of useful macro-operators to help in planning speed-up.

Examples

The substate class transition diagram for the sort *Door* in R^n is shown in Figure 4. The predicate expressions annotating the nodes in Figure 4 define the three substate classes that any door object can be in, 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. Each node's annotation, therefore, represents a necessary condition that must be satisfied for objects of sort *Door* to be in a well-formed state; and in any well-formed state of the model, each object of the sort *Door* must be in a substate satisfying exactly one of the predicate expressions. The arcs record state transitions, and show that there is no direct way to change the substate of a *Door* directly from open and unlocked to closed and locked, the only 'route' is via the intermediate state of closed and unlocked.

For the sort *Box*, objects are deemed to be either on the floor of a *Room*, and not next to a *Box*, a *Key*, or near a *Door*; *or* on the floor of a *Room*, at a *Door*, and not next to a *Box* or *Key*; *or* on the floor of a *Room*, next to another *Box*, and not near a *Door* or next to a *Key*; *or* on the floor of a *Room*, next to a *Key*, and not next to a *Box* or near a *Door*. This gives us the nodes in the transition diagram for the *Box* sort, in Figure 5 on page 15. The cyclic arcs are used to indicate a change of substate where only bindings of the predicate expressions change (to indicate, for example, that a box may be moved from one door to another within a room).

From the diagram we can write down the substate class definitions for an object *Bx* of sort *Box* (here Rm, Rm_1 are variables of sort *Room*, Bx_1 is of sort *Box* and *Dr* is of sort *Door*. The *is_of_sort* predicate is used to restrict a variable of a non-primitive sort to a subsort of that sort):

$$\{on\_floor(Bx, Rm)\}$$
$$\{on\_floor(Bx, Rm), near\_door(Bx, Dr, Rm), connect(Rm, Rm_1, Dr)\}$$
$$\{on\_floor(Bx, Rm), next(Bx, Ky), Ky\ is\_of\_sort\ Key\}$$

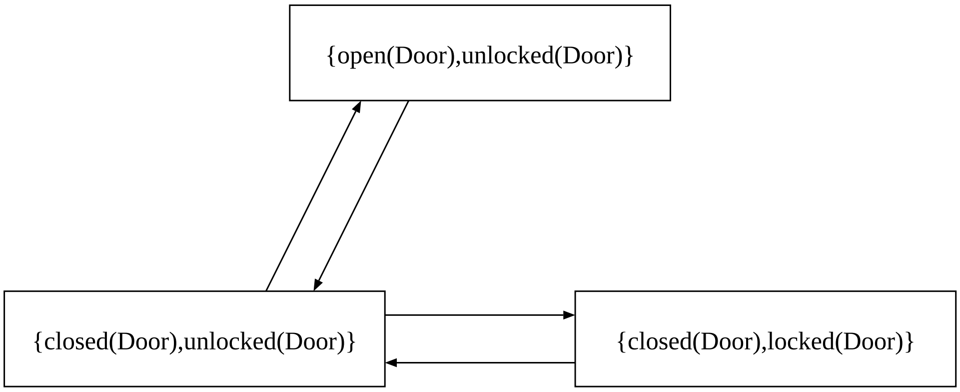

Figure 4: Substate Class Transition Diagram for Sort *Door*

$$\{on\_floor(Bx, Rm), next(Bx, Bx_1), Bx_1\ is\_of\_sort\ Box, Bx \neq Bx_1\}$$

We assume a *local* closed world assumption for dynamic predicates chosen to describe a box in the substate class definitions. For example, since in the first set it is not asserted that Bx is next to an object, then this is assumed false for any box object occupying this substate. Here the modeller has used the following static predicates to restrict values of the dynamic sorts:

$$connect(Room, Room, Door), Obj \neq Obj, Obj\ is\_of\_sort\ Sort$$

An object's substate satisfies a substate class definition if the substate matches with the dynamic predicates in the definition, and any static predicates in the definition evaluate to true unders the bindings caused by the match.

2.5 State Invariant Construction (step 5)

The next stage in engineering of the domain is to specify logical *state invariants*. Informally, invariants are axioms that rigorously define the conditions that must be true of a planning state. Invariants have been used before in planning (in for example WARPLAN [58]), but we have exploited them in various aspects of domain model engineering such as compilation, tool construction, validation and maintenance. In particular, invariants can be used for consistency checking, and to explicitly record the assumptions of the modeller. Using our method invariants are acquired both manually and automatically. They fall into the following classes:

- positive invariants: expressions that must be *true* in every planning state. This includes a set of *atomic* invariants, that is, those instances of static predicates that are always true. In the context of planning with substates, the substate class definitions are in fact positive invariants: a necessary condition for a planning state to be well-formed is that each object identifier maps to a substate belonging to one of that object's sort's substate classes.

- negative invariants: These are expressions that must be *false* in every planning state. This category implicitly includes all instances of static predicates that have not been declared as always true. Explicitly-declared negative invariants are termed 'inconsistency constraints'.

Examples of invariants

Positive atomic invariants are given primarily by a list of all predicate instances that never change during planning, for example:

$$connect(room6, room5, door56)$$
$$position(light4, room4, door45)$$
$$colour(door47, blue)$$

are instances of static predicates that are always true. We have already met two distinguished static predicates, '$\neq$' and '*is_of_sort*', each with the obvious meaning.

For an example of a negative invariant, consider in R^n part of a valid state describing the situation of robot *harry*:

$$harry \mapsto \{on\_floor(harry, room3), next(harry, box1)\}$$

Hence the substate of $box1$ must be an instantiation of the dynamic predicates in one of the following sets, such that the bindings of object identifiers to sort variables Dr, Rm_1, Ky, or Bx satisfies the associated static predicates:

$$\{on\_floor(box1, room3)\}$$
$$\{on\_floor(box1, room3), near\_door(box1, Dr, room3), connects(room3, Rm_1, Dr)\}$$
$$\{on\_floor(box1, room3), next(box1, Ky), Ky\ is\_of\_sort\ Key\}$$
$$\{on\_floor(box1, room3), next(box1, Bx), box1 \neq Bx, Bx\ is\_of\_sort\ Box\}$$

or in other words $box1$ must be in some substate in which it is in the same *Room* as the robot *harry*. This can be summed up using a generalised, negative invariant:

$$\exists A, B \in Movable\_obj, \exists R_1, R_2 \in Room : \\ on\_floor(A, R_1) \& next(A, B) \& on\_floor(B, R_2) \& R_1 \neq R_2$$

and/or it could be expressed as a positive invariant as follows:

$$\forall A, B \in Movable\_obj, \forall R \in Room : \\ next(A, B) \& on\_floor(A, R) \to on\_floor(B, R)$$

Maintaining two separate sets of invariants (positive and negative) in this way is required so that the tools that use them can work more efficiently.

As another example of an invariant, consider the substate class for sort *Arm* given by the expression:

$$\{arm\_used(Arm, Robot, Key), part\_of(Robot, Arm)\}$$

If it is assumed that in DR^n it is only possible for a key to be held by *one* robot arm, a planning state in which two different arms were both being used to hold the same key is not allowed. For example, a state containing the following two substates would be invalid:

$$\begin{aligned}&\{\ldots\\&harrys\_arm \mapsto \{arm\_used(harrys\_arm, harry, key3)\},\\&dicks\_arm \mapsto \{arm\_used(dicks\_arm, dick, key3)\},\\&\ldots\}\end{aligned}$$

A negative invariant for sort Arm is the constraint that no two arms can hold the same key:

$$\begin{aligned}&\exists\, A_1, A_2 \in Arm, \exists\, R_1, R_2 \in Robot, \exists\, K \in Key :\\&arm\_used(A_1, R_1, K) \& arm\_used(A_2, R_2, K) \& A_1 \neq A_2.\end{aligned}$$

In concrete syntax (used later) this constraint would be written:

$$inconsistent\_constraint(arm\_used(A_1, \_, K) \& arm\_used(A_2, \_, K) \& A_1 \neq A_2).$$

An important question is when does the domain modeller know that the invariant contains 'enough' information? For the positive invariants specifying substate classes this is straightforward since for each dynamic sort that has been identified the invariant must specify all of the abstracted states that objects of that sort can occupy. However, we know of no such 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 Operator Specification (step 6)

In the natural language description of DR^n verbs and verb phrases not only suggest relationships and properties of the domain, and possible states of sorts, but also ways in which states can be changed. For example, the description of the domain includes the following text *'Robots ... capable of ... pushing ... objects to doors, through doors and next to other objects'*, suggesting the possible ways of changing the state of a movable sort. We call these the *actions* in the planning domain because they affect objects of dynamic sorts in the model.

Within our framework, actions are represented by schema called *operators* that have pre- and post-conditions, which, in contrast to classical planning operators, are specified in terms of the substate transitions of sorts. The need for operators is identified using the transition diagrams from the step above. The node description can initiate the pre- and postconditions of the operator and for each arc that leads to a node a suitable operator name is selected that could be used to label the arc (for example, in Figure 5, *pushtodoor* could be used to label the arc from Node B to Node A, and *pushthrudoor* to label the arc from Node A to Node B).

Clearly operators change the substates of objects from the sort on whose transition diagram they appear, but the domain modeller needs to also consider any other objects that the

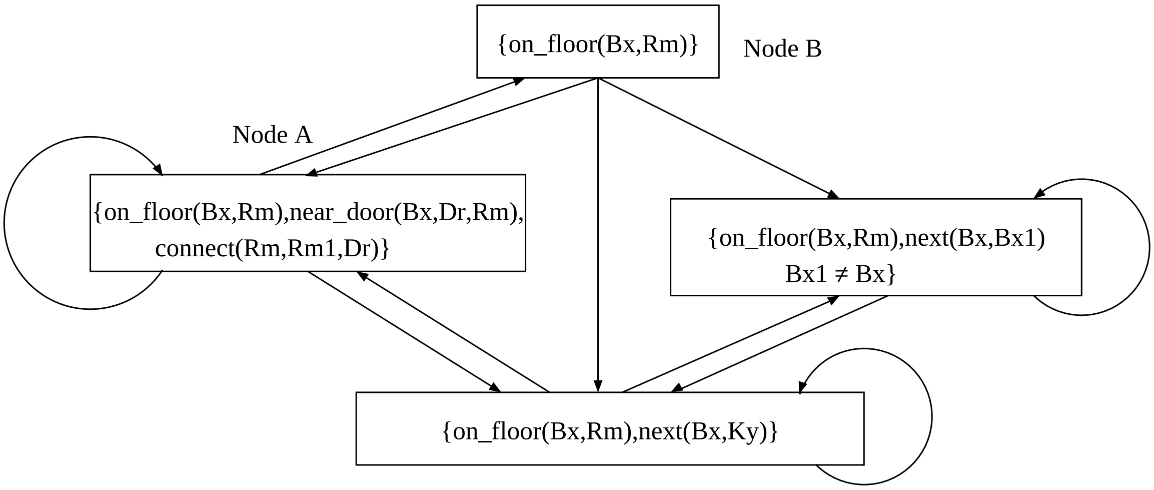

Figure 5: Substate Class Transition Diagram for sort *Box*

operator may affect, along with any other objects that are involved but not affected (that is conditions on an object that prevail). We will show how an operator representing the action of pushing a box through a door can be constructed using the sort *Box*'s transition diagram shown in Figure 5. Each node represents a substate class (as recorded in section 2.4), and each arc an abstracted state transition. Each operator will be designed by recording:

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

The term 'condition' in each of these operator components refers to a predicate expression which matches with one or more substate classes. In an operator's definition we precede each conditional expression by the sort name to which it is applicable. Hence the condition:

$$Box : \{on\_floor(Box_1, Room_1)\}$$

could be satisfied by a box object at any of the nodes shown in the transition diagram. On the other hand, the new substate of the affected object given in the Necessary and Conditional Effects slots is interpreted as identifying a unique substate class (or equivalently a node in the transition diagram), to ensure that the operator produces a well-formed output state.

Construction of Prevail Conditions

The operator that transforms node A to node B (figure 5), requires an object of sort *Door*, called $Door_1$, to be open, and the arm object of the robot, called Arm_1, to be empty (in this domain a robot cannot carry while its pushing), but neither of these conditions are changed by the operator. These two conditions are:

$Door : \{open(Door_1), unlocked(Door_1)\}$
$Arm : \{arm\_empty(Arm_1, Robot_1), part\_of(Robot_1, Arm_1)\}$

They must be specified in the preconditions of the operator, but will not appear in the effects.

Construction of Necessary Effects

The operator that pushes a *Box* through a *Door* changes the substate of *Box* from one satisfying:

$$\{near\_door(Box_1, Door_1, Room_1), on\_floor(Box_1, Room_1),$$
$$connect(Room_1, Room_2, Door_1)\},$$

to the substate:

$$\{on\_floor(Box_1, Room_2)\}$$

Now actions may affect other objects in certain ways. $Robot_1$ must 'push' the Box_1 through the door and hence the operator also changes the substate of $Robot_1$ from one satisfying:

$$\{on\_floor(Robot_1, Room_1), next(Robot_1, Box_1)\}$$

to the substate:

$$\{on\_floor(Robot_1, Room_2), next(Robot_1, Box_1), Box_1 \ is\_of\_sort \ Box\}$$

Although the scope of sort variables is global to the whole of the operator, substate expressions may be examined individually, hence we needed to place the '*is_of_sort*' restriction in the expression above.

Construction of Conditional Effects

There may be other objects that change depending on context e.g. it may be that other movable objects were next to the box about to be pushed. For any object, *Obj*, if its current substate satisfies:

$$\{next(Obj, Box_1), on\_floor(Obj, Room_1)\},$$

then it changes to the substate:

$$\{on\_floor(Obj, Room_1)\}$$

To systematically check for conditional effects, the modeller considers any dynamic objects that may be affected by the change in the substate of the necessarily affected objects. This may be done by searching through transition graphs and considering nodes which may refer to the necessarily affected objects (that is Box_1, $Robot_1$). At each one, the modeller must decide whether the substate can co-exist in a planning 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 may change as a side effect, then the modeller decides to which substate.

Name: pushthrudoor
Prevail conditions

$Door : \{open(Door_1), unlocked(Door_1)\}$
$Arm : \{arm\_empty(Arm_1, Robot_1), part\_of(Robot_1, Arm_1)\}$

Necessary state changes

$Box : \{near\_door(Box_1, Door_1, Room_1), on\_floor(Box_1, Room_1), connect(Room_1, Room_2, Door_2)\},$
$\quad CHANGES\ TO \quad \{on\_floor(Box_1, Room_2)\}$
$Robot : \{on\_floor(Robot_1, Room_1), next(Robot_1, Box_1)\}$
$\quad CHANGES\ TO \quad \{on\_floor(Robot_1, Room_2), next(Robot_1, Box_1), Box_1\ is\_of\_sort\ Box\}$

Conditional state changes

$Movable\_obj : \{next(Obj, Box_1), on\_floor(Obj, Room_1)\}$
$\quad CHANGES\ TO \quad \{on\_floor(Obj, Room_1)\}$

Figure 6: The operator Pushthrudoor

Operator Integrity

The operator design can now be encoded into a concrete syntax using add and delete lists etc, for the purpose of input to the target planning engine. We can also use the design to reason about its properties, in particular we can ask if the operator is:

1. *consistent* – if an operator is applied in a well-formed planning state, is the resulting state well-formed?
2. *deterministic* – if an operator is applied in a well-formed planning state, then is the resulting state unique?

Both these properties follow fairly trivially from the way we have constructed the operator. In the next section, after formally defining an operator's semantics, operator consistency and well-formedness of states, we will show how the modeller can prove the consistency of the 'pushthrudoor' operator.

3 A Formalisation of the Object-centred Framework

The result of using the method of the last section is the production of a specification of the model $\mathcal{M}$ of a domain of interest $\mathcal{D}$, which is composed of sets of:

- object identifiers: *Objs*
- sort definitions: *Sorts*
- predicate definitions: *Prds*

- substate class expressions: *Exps*
- positive and negative domain invariants: $Invs^{+}$ and $Invs^{-}$
- operators: *Ops*

In this section we formalise the object-centred framework. Some definitions from the previous section will be made more precise, some useful properties of domain models will be described, and some notation used in later sections will be introduced.

3.1 Loosely Sort-Abstracted (LSA) Models

Our starting point is to declare what we mean by a sort:

Definition 1 *A* sort *is a set of object identifiers in $\mathcal{M}$ denoting objects in $\mathcal{D}$ that share a common set of characteristics and behaviours.*

Objs is thus the union of all the sorts in $\mathcal{M}$. The characteristics and behaviours of a sort are modelled by the invariants and operators associated with that sort. Unless otherwise stated, in what follows the word 'object' will be used to mean 'object identifier'.

Definition 2 *Sorts are either primitive or non-primitive.* Non-primitive *sorts are defined as the union of objects from two or more other sorts. A sort is* primitive *if it is not defined in terms of other sorts.*

Each object is a member of exactly one primitive sort, and if an object is a member of two sorts s_1 and s_2 then s_1 must be a supersort of s_2 or vice-versa. Each argument of an element of *Prds* (predicates have at least one argument) is pre-defined in $\mathcal{M}$ as referring to objects of one sort, which may be primitive or non-primitive.

Let *Bindings* denote all sequences of legal bindings of object identifiers to variables within arguments of predicates. A binding of an object to a variable is legal if the object belongs to the same sort as the variable according to the predicate's definition. A *grounding* of a predicate (or predicate formulae) is a binding of all its sort variables to objects. We can thus define the set of all possible ground predicates $Prds_G$ as the set of all legal groundings of every predicate definition.

Definition 3 *The* Positive Atomic Invariant *is the subset of $Invs^{+}$ whose members are taken from $Prds_G$ and are interpreted as being always* true *in $\mathcal{M}$.*

All other instances of those predicates appearing in the Positive Atomic Invariant are interpreted as being always *false* in $\mathcal{M}$ (and form a *negative* atomic invariant). For example, in R^n, as the static ground predicate:

$$connect(room1, room6, door16)$$

does not appear in the positive invariants it is therefore a member of $Invs^{-}$. Static predicates such as '*is_of_sort*', are assumed to be defined explicitly in the Atomic Invariant.

The following two definitions are to do with the split in a model between objects and predicates that are affected by operators, and those that are unaffected.

Definition 4 *A predicate is* static *if one or more instances of it are declared true in the Atomic Invariant. Otherwise the predicate is considered* dynamic.

Definition 5 *A primitive sort is* dynamic *if its objects are deemed to change state in $\mathcal{M}$. Otherwise a sort is called* static. *Objects of a dynamic (static) sort are called dynamic (static).*

Below we let $Sorts_D$ and $Objs_D$ denote the dynamic sort names and the dynamic objects respectively. We use the phrase 'are deemed' in Definition 5, as the decision as to which sorts are considered dynamic in some cases is not straightforward. We could, for example, have declared *Room* as dynamic in R^n, the state of a room changing depending on which objects were in it. Sorts whose objects feature frequently in the atomic invariants are good candidates for static sorts.

Having reviewed the basic terminology for objects, sorts and predicates, we will formalise the substate idea. First, for each $s \in Sorts_D$, we assume that the modeller chooses an exclusive set of dynamic predicate definitions $Prds^s \subset Prds$ with which to describe the local state of objects of s (in our earlier publications we used the terminology '*s owns $Prds^s$*'). Further, we restrict the syntax of elements of $Prds^s$ so that the object whose state they refer to is given in their *first* argument. For example the three members of *Prds* chosen for primitive sort *Box* are:

$on\_floor(Movable\_obj, Room)$,
$near\_door(Movable\_obj, Door, Room)$,
$next(Movable\_obj, Movable\_obj)$

with the first argument of each predicate restricted to members of sort *Box*.

Definition 6 *Let $Prds^s_G$ be the set of all groundings of $Prds^s$. A* valid substate *of $o \in Objs_D$ of sort s, is a set $W \subset Prds^s_G$, describing a situation in $\mathcal{D}$ of the referent of o, such that*
(a) every predicate in W is deemed true in the interpretation given by $\mathcal{D}$,
(b) every predicate in $(Prds^s_G - W)$ that has its first argument equal to o is deemed false in the interpretation given by $\mathcal{D}$.

Let *Cjn* denote all conjunctions of predicates that can be made up from members of *Prds*. They have the property that they may or may not be satisfied by a subset of $Prds_G$ as follows:

Definition 7 *If $W \subset Prds_G$ and $\mathcal{C} \in Cjn$, then W* satisfies *$\mathcal{C}$ if and only if there is a grounding of $\mathcal{C}$ given by $\alpha \in Bindings$, call it $\mathcal{C}_\alpha$, such that $W \& Invs^+ \vdash \mathcal{C}_\alpha$.*

An important property of a member of *Cjn* is whether or not it can satisfy a negative invariant:

Definition 8 *For any $\mathcal{C} \in Cjn$,* inconsistent($\mathcal{C}$) *is true if and only if for all $\alpha \in Bindings$ which are groundings of $\mathcal{C}$, there is an $\mathcal{A} \in Invs^-$ such that $\mathcal{C}_\alpha$ satisfies $\mathcal{A}$.*

In practice one defines the set of possible substates of a dynamic object implicitly using the substate class expressions *Exps*. Elements of *Exps* that apply to a sort s are members of an important subclass of *Cjn* called Cjn^s. This is defined as those conjunctions composed of dynamic predicate(s) taken from $Prds^s$ only, and zero, one or more static predicates from *Prds*. As well as this constraint, substate class expressions are interpreted under a local closed world assumption when describing the substate of some object o. This means that all instances of any predicate in $Prds^s$, describing o but not included in the expression, are *false* in that substate.

The model's substate class expressions *Exps* are segregated into groups associated with each sort, and defined as follows:

Definition 9 *$Exps^s$ is a* valid set of substate class expressions *for sort s if*
(a) every valid substate of objects of sort s satisfies exactly one member of $Exps^s$
(b) for each expression $\mathcal{C}$ in $Exps^s$, if $\mathcal{C}_\alpha$ is a grounding of $\mathcal{C}$ under bindings α, and all static predicates in $\mathcal{C}_\alpha$ are true in $\mathcal{M}$, then the remaining formulae's dynamic predicates form a valid substate of an object of sort s.

Adequacy of substate descriptions, and validity of sets of substate class expressions for a sort, cannot be formally checked but have to be validated using the domain requirements.

We call the process of designing the planning model's state following this method '*sort abstraction*' since the sort and its substate classes are the main abstractions employed. We now turn to the operators in the model:

Definition 10 *An* operator *O is a schema having three components $O.P, O.E^n, O.E^c$ where $O.P$ is a set of pairs, and each pair x has components $x.s \in Sorts_D$ and $x.\mathcal{C} \in Cjn^{x.s}$; and $O.E^n$ and $O.E^c$ are sets with each member x having three components $x.s \in Sorts_D, x.\mathcal{C}^p \in Cjn^{x.s}$ and $x.\mathcal{C}^e \in Cjn^{x.s}$.*

In later sections we may take the liberty of referring to an operator simply by a name and a collection of objects or object variables as the name's parameters. These parameters are chosen to be the smallest set such that their instantiation grounds the preconditions of the operator.

Drawing on these definitions we can describe an important, necessary condition of model well-formedness:

Definition 11 (LSA Property) *A model $\mathcal{M}$ of a domain $\mathcal{D}$ is* loosely sort abstracted *if the following restrictions on the components of $\mathcal{M}$ are true:*

(a) for Objs: for every object in $\mathcal{D}$ that needs to be represented there exists a unique object identifier in Objs that refers to it;

(b) for Sorts: every object identifier in Objs is a member of exactly one primitive sort. All sorts (and therefore all objects) are deemed to be either dynamic or static;

(c) for Prds: Prds is defined such that each predicate's argument may refer to any object identifier from exactly one (primitive or non-primitive) sort.

(d) for Exps: a set of valid substate class expressions $Exps^s$ has been constructed for each dynamic sort s, according to Definition 9. All the dynamic predicates in Prds appear in at least one substate class description.

(e) for Invs: the invariant is assumed to define (at least) the truth values of the distinguished static predicates '$=$', '$\neq$' and 'is_of_sort' for all domain objects.

(f) for Ops: a set of operators have been defined according to Definition 10.

The LSA model would normally contain many invariants, although this part of the specification is somewhat open ended in a realistic application.

Given the LSA property, we can make the following central definition of planning states that represent the changeable part of a domain model.

Definition 12 *Let Cjn^s_G represent those members of Cjn^s which are grounded. Then the set of* well-formed states *in an LSA model $\mathcal{M}$ is the set of all the (total) maps*

$I : Objs_D \rightarrow Cjn_G$

such that, for any state I,

(a) $\forall\, o \in Objs_D$, if o is of sort s then $I(o) \in Cjn^s_G$ satisfies exactly one member of Exp^s

(b) range(I) (the conjunction of the range elements of every object in the domain of I) satisfies none of the negative invariants in $\mathcal{M}$.

3.2 Operator Complete Models

The operational semantics of an operator O is given by the next two definitions:

Definition 13 *O is* applicable *to a well-formed state I if there exists a binding α for all the variables in $O.P$ and $O.E^n$ such that:*

(a) $\forall\, x \in O.P, \exists\, o \in Objs_D$ of sort $x.s$ such that $I(o)$ satisfies $x.\mathcal{C}_\alpha$

(b) $\forall\, x \in O.E^n, \exists\, o \in Objs_D$ of sort $x.s$ such that $I(o)$ satisfies $x.\mathcal{C}^p_\alpha$.

Note that while the operator may be applicable to different objects depending on the binding chosen, given a binding and a state I the objects necessarily affected are determined.

Definition 14 *If O is applicable to a well-formed state I in $\mathcal{M}$ under a binding α, then new state $O(I)$ is equal to I with the following substate replacements given by (a) and (b):*

(a) for each member of $O.E^n$, we know from Definition 13 that there is an object o such that $I(o)$ satisfies $x.\mathcal{C}^p_\alpha$. Replace the maplet $(o \mapsto I(o))$ with the maplet $(o \mapsto x.\mathcal{C}^{new})$ where $x.\mathcal{C}^{new}$ is the set $x.\mathcal{C}^e_\alpha$ with static predicates removed.

(b) Let $Objs'$ be those dynamic objects unaffected by (a). Then $\forall\, x \in O.E^c, \forall\, o \in Objs'$, if $I(o)$ satisfies $x.\mathcal{C}^p_\alpha$ under some binding β, then replace the maplet $(o \mapsto I(o))$ with the maplet $(o \mapsto x.\mathcal{C}^{new})$ where $x.\mathcal{C}^{new}$ is the set $x.\mathcal{C}^e_{\alpha\beta}$ with all its static predicates removed.

These definitions extend naturally to the *sequential application* of a sequence of operators. Although we have defined operator applicability without the use of an explicit set of preconditions, we will find it useful to have a function which returns them:

Definition 15 *$precons : Ops \rightarrow Cjn$*
is a function such that $precons(O) = \mathcal{P}\&\mathcal{Q}$, where
$\mathcal{P}$ is the conjunction of all $x.\mathcal{C}$ such that $x \in O.P$
$\mathcal{Q}$ is the conjunction of all $x.\mathcal{C}^p$, such that $x \in O.E^n$.

precons returns the preconditions of an operator O, and it follows that O is applicable to any state I if $range(I)$ satisfies $precons(O)$.

The applicative semantics of operators and the well-formedness property of states allows us to define a notion of consistency for operators:

Definition 16 *An operator O in an LSA model is* consistent *if inconsistent(precons(O)) is false, and O's application to a well-formed state (assuming it is applicable to that state) transforms it into another well-formed state. An operator set is consistent if all its members are.*

Example The 'pushthrudoor' operator defined in section 2 is consistent.

The first part of the consistency definition is to check that there is at least one state in which the operator can be applied, and for this case it is straightforward to verefy that

$$\begin{aligned}&\mathit{inconsistent}(\{\mathit{open}(\mathit{Door}_1), \mathit{unlocked}(\mathit{Door}_1), \mathit{arm\_empty}(\mathit{Arm}_1, \mathit{Robot}_1),\\ &\quad \mathit{part\_of}(\mathit{Robot}_1, \mathit{Arm}_1), \mathit{near\_door}(\mathit{Box}_1, \mathit{Door}_1, \mathit{Room}_1), \mathit{on\_floor}(\mathit{Box}_1, \mathit{Room}_1),\\ &\quad \mathit{connect}(\mathit{Room}_1, \mathit{Room}_2, \mathit{Door}_2), \mathit{on\_floor}(\mathit{Robot}_1, \mathit{Room}_1), \mathit{next}(\mathit{Robot}_1, \mathit{Box}_1)\})\end{aligned}$$

is *false*. The second part was generally established by the systematic form of its construction, but we can check it in more detail by a two step process: (a) we check that each expression stated after '*CHANGES TO*' in the specification contains the same collection of predicates as one of the substate class definitions (b) we check the possible output states resulting from the operation against the set of negative invariants to make sure that they all remain unsatisfied.

(a) The expressions after '*CHANGES TO*' are:

$$\{\mathit{on\_floor}(\mathit{Box}_1, \mathit{Room}_2)\}$$
$$\{\mathit{on\_floor}(\mathit{Robot}_1, \mathit{Room}_2), \mathit{next}(\mathit{Robot}_1, \mathit{Box}_1), \mathit{Box}_1\ \mathit{is\_of\_sort}\ \mathit{Box}\}$$
$$\{\mathit{on\_floor}(\mathit{Obj}, \mathit{Room}_1)\}$$

The full definition of $\mathcal{M}$ can be used in a straightforward (and automated) manner to verify that these are equivalent to exactly one of the substate class definitions.
(b) We must check that the introduction of the new substates does not potentially satisfy one of the negative invariants. From the definition of operator application we can see that the output state has two changed substates resulting from the necessary condition, and zero, one or more other changed substates depending on the conditional effect. Since the input state is well-formed, we know that the unaffected substates are self-consistent. The definition of operator application shows that '$\mathit{Robot}_1 \neq \mathit{Obj}$' and '$\mathit{Box}_1 \neq \mathit{Obj}$', hence there is no possible inconsistency arising from the conjunction of the changed states. Finally, we have to check through the negative invariants which contain one or more of the predicates appearing in the new substates, to verify that they remain unsatisfied by the output state. Assume we *left out* the substate change to

$$\{\mathit{on\_floor}(\mathit{Obj}, \mathit{Room}_1)\}.$$

A search of the invariants would reveal:

$$inconsistent\_constraint(on\_floor(Obj, Rm)\&next(Obj, Obj1)\&$$
$$on\_floor(Obj1, Rm1)\&Rm \neq Rm1)$$

since predicates in this constraint are affected by the operation. To prevent this from being satisfied after the operation of pushthrudoor, the relevant conditional effect would be added as shown in Figure 6.

Putting Definitions 11 and 16 together gives us a standard for planning models which we call 'operator complete':

Definition 17 *A domain model $\mathcal{M}$ is* operator complete *if*

(a) $\mathcal{M}$ satisfies the LSA property,

(b) the operator set in $\mathcal{M}$ is consistent

(c) the operator set in $\mathcal{M}$ is complete relative to dynamic sorts, that is all required transitions in the transition diagrams have been represented by operators, and every dynamic predicate appears in the effects of some operator.

The concept of the 'weakest precondition' of a plan is important in knowledge compilation for planning, and we will need the following definition in later sections:

Definition 18 *Let $(Ops_G)^*$ represent the sequences of ground operators in $\mathcal{M}$. We define the* weakest precondition *of a sequence with respect to a conjunction of goal predicates to be the function:*

$$wp : (Ops_G)^* \times Cjn_G \rightarrow Cjn_G$$

If OS is sequentially applicable to at least one well-formed state, then $wp(OS, G)$ is the smallest conjunction such that given any state I where $range(I)$ satisfies $wp(OS, G)$, OS is applicable to I and will produce a new state whose range satisfies G. Otherwise $wp(OS, G) = false$.

3.3 Goal Conditions

Engineering domain models to be operator complete is itself a significant step away from a model consisting only of a set of literal-based operators. While operator completeness gives a level of well-formedness for object-centred domain models, it does, however, allow conditions in an operator's preconditions, or predicates in a goal, to be *underspecific* in that when instantiated they might not give a unique substate (as mentioned at the end of section 2.3). A tighter property that does not allow substates to be decomposed, is as follows:

Definition 19 (TSA property) *A domain model is* tightly sort abstracted *if*

1. *it is operator complete*
2. *every substate class expression consists of exactly one predicate.*

The apparent advantage of creating a TSA model is one of efficiency - here any conjunction of ground predicates is equivalent to a conjunction of objects' substates. If the domain is not TSA, then goal predicates may specify a *disjunctive* goal in terms of object states. On the other hand, refining a loose formulation into a TSA model loses the flexibility and expressiveness of the original.

In the classical formulation, goal conjunctions are often expressed as sets of literals. We interpret a goal conjunction as a map between objects and *sets of* substates, explicitly showing the conjunctive and disjunctive nature of goals.

For example, consider the following literal set representing a planning goal in the LSA model of R^n:

$\{near\_door(tom, door45, room4), next(box1, box2), \neg\ open(door12), on\_floor(tom, room4)\}$

This would be interpreted as describing the substates of 3 dynamic objects $tom, box1, door12$ (as these are the objects in the first slots of the predicates), and so it would be translated into a goal consisting of the conjunction of the three objects' substate expressions:

$$\begin{aligned}
&\{tom \mapsto \{\{on_floor(tom, room4), near_door(tom, door45, room4)\}\}\\
&door12 \mapsto \{\{closed(door12), unlocked(door12)\}, \{closed(door12), locked(door12)\}\}\\
&box1 \mapsto \{\{on_floor(box1, room1), next(box1, box2)\},\\
&\qquad\qquad \{on_floor(box1, room2), next(box1, box2)\} \ldots\}\}
\end{aligned}$$

The negated goal above evaluates to a set of goal substates that do not contain the predicate after the negation. This formulation has the effect of making implicit disjunction explicit, as a single literal goal may correspond to a set of object states (as in $box1$'s goal state). In what follows we will assume that planning goals are still input as literal sets, but we will also use the following logical definition of *goal conditions* within LSA models.

Definition 20 *The set of well-formed* goal conditions *in an LSA model $\mathcal{M}$ is the set of all maps:*

$G : Objs_D \to Cjn_G\text{-}set$

where each member of Cjn_G-set is a set of substates of one sort, and there exists at least one well-formed state $I \in \mathcal{M}$ such that $\forall\, c \in dom(G) : I(c) \in G(c)$.

Note that for any goal G, $dom(G) \subseteq dom(I)$, as goals can only be posed for known objects.

We end the section by defining the useful function *achieve* which returns true if and only if a goal condition is met in a state:

Definition 21 *For any well-formed state I and goal condition G in an LSA model $\mathcal{M}$*
$achieve(I, G) \Leftrightarrow \forall\, c \in domain(G) : I(c) \in G(c)$

4 Tool Support for a Particular Class of Architectures

In this section we will examine a major benefit of formalising planning domain models - the ability to support the whole process with tools. Their use can be interspersed throughout the whole cycle. Tools all help to some degree in the crucial aspect of *validation and verification* of a model, and here we classify them into two main categories:

1. Tools for use in model construction and maintenance, which might include:
   (a) syntax and sort cross-checking of a model's components
   (b) static analysis of operators: checking that operators are consistent and that dynamic goal predicates are achievable. MVP is an example of a planning system featuring tools that test whether certain goals are unachievable [13].
   (c) graphical editing, for example in the construction of substate class transition diagrams and operators. The SOCAP system features a range of knowledge development tools including a graphical operator editor which performs type and consistency checking of operators [18].
   (d) partial construction of operators from diagrams, similar to the kind of 'methods integration' tools used in software engineering [53]. For example, the modeller could use a graphical editor (featuring node-link diagrams) to enter operator pre and post-conditions and then the tool could automatically store these in the planner input format.
   (e) generating negative invariants from operator definitions. An early version of this features in the work of Dawson and Siklossy [17] where sets of 'assertions' (predicates) are deemed to be 'incompatible' (inconsistent) if they appear in both the positive and negative effects of an operator.
   (f) generating random well-formed planning states and sets of goal literals, and hence random planning problems (as planning states, in our formulation, are defined as amalgams of substates satisfying the invariants, the potential for randomly generating problems in a systematic manner is obvious).
2. Tools used primarily for compiling a domain model into a more efficient or operational form. These include:
   (a) macro generation (see section 4.1 below),
   (b) generation of various types of goal orderings (see section 4.2 below).
   (c) abstraction hierarchy generation (for example ALPINE in [35, 36]).

At present our environment contains a number of tools of category 1 (a, b, e and f) as follows:

- a tool that uses the substate class definitions to: check the syntax of the domain operator set; check that all substates that should be achievable are indeed achievable by operator action; help check that the operator set is consistent.
- a tool that identifies goals that are unachievable given the current operator set (this is described as a *Type (iii) Blocking* below).
- a tool that generates random planning problems for a particular domain using information from state invariants, substates and descriptions of well-formed states of the domain to ensure that generated problems are valid.
- a tool that generates negative invariants through static analysis of operator effects.

The function of the remaining category 1 tools are performed by hand although they are amenable to automation. In the remainder of this section we will concentrate on the tools that fall into category 2 above, the more planning-specific category. In section 4.1 we introduce a method for generating macro operators for planning through model analysis and then in section 4.2 we introduce a method for generating goal orderings. The effectiveness of these tools is dependent on the input of an operator complete model constructed using a systematic approach as described in section 2.

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: (a) to improve the efficiency of the representation of the model, when used with a planning engine, *without* any marked adverse effect on the quality of solutions and (b) to further validate and verify the model.

4.1 Macros in Planning

The overall effect of 'macro creation' in this context is to produce, during a once only compilation stage, a set S of partial solutions which can be used as building blocks for complete solutions of planning problems. There are two obvious extremes:

- S is empty. In this case the planner has no macros, and uses the basic operators as its building blocks.
- S is the set of macros representing every solution to every planning problem.

In the kind of problems we have in mind both the generation and maintenance of an exhaustive set of macros would be intractable. A good set of macros is one that satisfies (a) above, and that falls somewhere between these extremes. Given a domain model, a planner and a macro generation technique, the performance of a macro set can be predicted by consideration of the following factors:

1. the likelihood of some macro being usable at any step in solving any given planning problem
2. the amount of processing (search) a macro cuts down
3. the cost of searching for an applicable macro during planning
4. the cost (in terms of solution non-optimality) of using a macro
5. the cost of generating and maintaining the macro set

We would like a set of macros and a selection technique that scored highly on the first two points and minimised the cost related to the other three. For example, taking the extreme where S is an exhaustive set means that the probability of a macro being applicable is 1, and the amount of processing (in terms of search space) cut down is total. Unfortunately, in all but trivial domain models, the cost of 3. and 5. would be prohibitive. As another example, assume a planner receives random problems to solve. Then the strategy of storing solutions to previously solved problems would (initially) give a high 2. and a low 3., 4. and 5. Unless there was some strong bias in the problem generation, 1. is likely to be very low. Finding a good trade-off between factors 1. – 5. is not easy.

Work on macro creation was pioneered by Korf in [37], at least for domains with propositional encodings (in the work cited he assumed states could be modelled as feature vectors of propositions). Korf gave some explicit properties that his macro sets display: the number of macros is a small fraction of the number of possible states; the amount of time required to generate the macros is of the same order as that of solving a problem instance; and the worst case solution length of a problem is equal to the number of subgoals in the problem times their optimal solution. These criteria are implicit in 1. – 5. above.

The regularity used in Korf's domains was that of 'operator decomposability' - his domains had to be expressed so that 'the effect of an operator on a state can be decomposed into its effects on each individual component of the state, independent of the other components of the state' [37, p 59]. The regularity relied upon by our macro technique is not dissimilar. If we replace the phrase 'component of the state' (which in Korf's work seems to mean a slot of a feature vector) by our 'substate' then we claim that operators in an operator complete encoding are decomposable in the sense that each operator is defined in terms of the effect it has on substate classes. Further, LSA implies that any problem that could be posed in the planning domain (expressed in terms of an initial state and goal condition) will require the manipulation of one or more instances of dynamic objects in the domain, where manipulation refers to the change in the substates of the objects. Now, if a planning domain model has been engineered to be at least LSA then the classes of substates that objects of a sort can occupy are pre-defined. This means that for any object of a dynamic sort that features in a planning problem, its substate both in the initial state and the goal must be a grounding of a substate's class.

For each such pair of substate classes (the generalised start and end situation for an object of a sort) we can produce a macro that represents an abstract plan for solving the (generalised) problem represented by this generalised pair. This macro production can be performed for every dynamic sort in a domain model in a way that spans the space of all possible pairs of generalised start and end situations.

In summary: given that the description of any object's substate must be a ground instance of a sort's substate classes, we can aim to generate a macro set that will be exhaustive with respect to providing solutions to subtasks involving the 'transition' of one object.

4.1.1 Macro Generation Techniques

Let us consider two substate classes within sort Box, and explore possible macros and generation techniques:

$$\{on\_floor(Bx, Rm_1)\}$$
$$\{on\_floor(Bx, Rm_2), next(Bx, Bx_1), Bx_1 \; is\_of\_sort \; Box\}$$

Such a pair of substate classes is called a *task configuration* below. Generating an abstract plan (a macro) that 'spans' these two generalised substates (assuming that the first one is an initial substate and the second is a goal) is an abstract problem of transporting a box from one room to another room and putting it next to another box in that room. A solution for this sort abstracted task in the R^n model will consist of a series of operators where a *Robot* (Rb_1) successively pushes the *Box* (Bx) through a series of *Rooms* (Rm_1, Rm_2, ..), connected by doors (Dr_1, Dr_2, ..) using operators *pushtodoor* and *pushthrudoor*, and then pushes the *Box* to be next to the other *Box* (Bx) using an operator *pushnext*. This can be represented as follows:

$$[[pushtodoor(Rb_1, Bx, Dr_1, Rm_1), pushthrudoor(Rb_1, Bx, Dr_1, Rm_2)]^*,\\ pushnext(Rb_1, Bx, Bx_1)]$$

where the abstract solution consists of a sequence (0,1 or more) of applications of *pushtodoor* and *pushthrudoor* (denoted by the * notation), followed by application of *pushnext*. There are a number of conditions on these abstract macros, chiefly that they *join* (the post-conditions of one operator in the sequence satisfy the sort abstracted preconditions of its successor) and that they are effective (they change the state of some object), as specified in [40].

Our initial work [47] concentrated on generating such fully generalised iterative macros - that is they would be applicable for any instantiations of the sort parameters in the task configuration. The iterative property arose out of cycles in the transition diagrams. Being very general, the total number generated at compile time would be relatively small. On the other hand, these macros had to be 'unwound' at plan-time, adding a processing overhead which endangered their utility.

The opposite end of the scale would be to generate macros for every 'ground' task configuration - that is for every pair of substates (rather than classes) within a sort. This still would be much more abstract than generating all possible plans since we would only be interested in abstract[3] plans between substates of the same sort. An example of such a ground task configuration and macro, if generated from R^n, might be:

$$i = \{on\_floor(box1, room1)\}$$
$$g = \{next(box1, box3), on\_floor(box1, room4)\}$$
$$[pushtodoor(harry, box1, door12, room1), pushthrudoor(harry, box1, door12, room2),\\ pushtodoor(harry, box1, door24, room2), pushthrudoor(harry, box1, door24, room4),\\ pushnext(harry, box1, box3)]$$

Both of these approaches have disadvantages, however: unwinding of fully generalised macros at plan-time consumes on-line resources; and compilation of fully ground macros may prove prohibitive in terms of computation, storage and maintenance. Hence, we were interested in finding some useful mid point to use for abstraction in the macro generation and our basis for this came from the difference between static and dynamic predicates that are used to define a planning domain model.

We observed that for many domains in the planning literature solution plans tend to be made up of repeated sequences of operators that are iterated over static relationships in the domain. In R^n many planning problems require moving instances of sort *Robot*, *Box*, *Key* through series of *Room* and *Door*, where the relationship between different *Rooms* and *Doors* is that adjacent doors and rooms are 'connected' to each other unchangingly. Our

[3] By abstract we mean one which suppresses details of all other dynamic sorts.

strategy then was to explore binding variables (in operator slots and substates) whose value is determined uniquely by static relationships and to leave all other variables generalised. Then the generalised variables could be instantiated at run-time since their values are dependent on the actual input planning problem. For an intuitive example of this processing consider the situation in R^n where at this level of abstraction we would compile the set of 'paths' that *any* instance of sort *Box* could be pushed by *any* instance of sort *Robot* through the interconnected doors and rooms. These paths are unchanging and at run-time need only be looked up and instantiated with the appropriate instances of *Box* and *Robot*.

The idea is embodied in our second and more successful macro generation algorithm which is shown in Figure 7 (and was used in our experiments described in section 5). The algorithm *Generate_macros* accepts as input components of an operator complete model and builds up a macro table. Step 1 initialises the macro table, while Step 2 iterates through every dynamic sort in the model. Step 3 calls a procedure which returns all the pairs of generalised substates (the task configurations) that are going to be used to produce and index the macros. Step 4 calls a procedure that gathers the reduced set of 'sort abstracted' operators - the set *Ops* with operators removed which do not affect objects of sort s. The resulting operators are further reduced by removing any components which are not prefixed by sort s. Step 5 iterates for each task configuration, producing generalised plans (Step 6) and storing them in a table (Step 7), indexed by their task configuration.

In step 6 a planner (such as the one described in section 5.4) is used to produce abstract plans for each task configuration where in step 6.1 it (a) searches in the abstract plan space containing only static predicates and dynamic predicates owned by the sort s; (b) grounds the task configuration with 'typical' instances of sorts, if necessary. Step 6.2 generalises the abstract plan along similar lines to a standard explanation-based technique [55]: the weakest precondition of the abstract plan is assembled, and all instances of sorts that are unified with variables in the preconditions of operators during execution are carefully generalised to variables in the final plan.

Returning to our example above, we have the generalised task configuration:

$$(\{on\_floor(Bx_1, room1)\},$$
$$\{next(Bx_1, Bx_2), on\_floor(Bx_1, room4), Bx_1 \neq Bx_2\})$$

and an explanation-based generalisation of the solution would be:

$$[pushtodoor(Rb_1, Bx_1, door12, room1), pushthrudoor(Rb_1, Bx_1, door12, room2),$$
$$pushtodoor(Rb_1, Bx_1, door24, room2), pushthrudoor(Rb_1, Bx_1, door24, room4),$$
$$pushnext(Rb_1, Bx_1, Bx_2)]$$

4.2 Generation of Goal Orders

The main function of the 'goal ordering' stage is to produce rules which cut down the range of goal[4] choices a planner may have at each step of a planning algorithm, although such goal

[4]Although a goal could potentially mean a literal or a substate, in this section we keep the discussion at the general level of literals.

algorithm Generate_macros
In Operator complete model $\mathcal{M}(Sorts, Objs, Prds, Exps, Invs, Ops)$
Out MT: a macro table

1. $MT := \{\ \}$;
2. for each s in $Sorts_D$ do
3. $\quad$ $generate\_task\_configs(Exps^s,\ \ TC^s)$;
4. $\quad$ $abstract(s, Ops,\ \ Ops^s)$;
5. $\quad$ for each pair (i, g) in TC^s do
6. $\quad\quad$ $produce\_generalised\_plans(i, g, Ops^s,\ \ GP)$;
7. $\quad\quad$ $MT := MT \cup ((i, g), GP)$
8. $\quad$ end for;
9. end for

end.
procedure generate_task_configs(**in** $Exps^s$, **out** TC^s)
3.1. $ins^s := \{j_\alpha \mid j \in Exps^s \,\&\, inconsistent(j_\alpha)$ is false,
where α is a binding of all static variables in j $\}$;
3.2. $TC^s := \{(i, g) \mid i \in ins^s, g \in ins^s, i \neq g\}$;
end procedure.
procedure abstract(**in** s, Ops, **out** Ops^s)
4.1. $Ops\_reduced := \{O \in Ops \mid \exists\, x \in O.E^n : x.s = s\}$
4.2. $Ops^s := \{O' \mid O \in Ops\_reduced \,\&$
$O' = O$ with all its components not referring to sort s removed$\}$;
end procedure.
procedure produce_generalised_plans(**in** i, g, Ops^s, **out** GP)
6.1. $P := \{p \mid p \in (Ops^s_G)^*$, and for some fixed grounding α of (i, g),
p is an optimal solution of g_α from initial state i_α $\}$;
6.2. $GP := \{G(p, wp(p, g_\alpha)) \mid p \in P, G$ is an EBG operator $\}$;
end procedure.

Figure 7: Outline Algorithm for Macro Generation

ordering techniques are also useful in spotting possible problems in the domain encoding, as we shall see later. The work here originates from Porteous's thesis [47] and is influenced by the work of [17, 12, 36].

Static analysis of the inherent goal structure of models can reveal heuristic or definite orders in which to establish individual goals that make up conjunctive goal planning problems. We examine the goal structure of the model by analysing pairs of literals: our approach has been to map out the set of possible ways that a *goal pair* could be established (using a 'goal establishment' diagram). This diagram is used to formulate general conditions that lead to blocks between nodes, and hence order goals for planning. Some of the conditions that we have identified have been implemented in a tool called *PRECEDE* and this section ends with an outline of its algorithm and some illustrative examples.

4.2.1 Goal Establishment

In addition to the notation introduced in section 3, we assume the existence of the following functions:

Definition 22 *$establishes(A_\alpha, p)$ is true if and only if operator A establishes literal p under the unique most general binding α (A_α makes p_α true).*

For example in R^n:

$$\begin{aligned} &establishes(pushnext(Rb_1, Bx_1, Bx_2)_\alpha, next(box1, box3)) = True \\ &\qquad\qquad where\ \alpha = \{box1/Bx_1, box3/Bx_2\} \end{aligned}$$

Definition 23 *For operator A and literal q, $clobbers(A, q)$ is true if and only if A_α clobbers q_α (makes it false) for any $\alpha \in Bindings$.*

Note that we assume it is not possible for $establishes(A_\alpha, p)$ and $clobbers(A, p)$ to both evaluate to true for any assignment of A and p. If $establishes(A, p)$, for some ground operator A and ground predicate p, then from Definitions 15, 18 and 22, we can derive:

$precons(A) = wp([A], p)$.

As a framework for our static analysis the goal establishment diagram of Figure 8 will be used. A node marked $x\&y$ identifies the set of all well-formed states which satisfy the two ground literals x and y (we assume this set is non-empty); and links, labelled P1 through P6, between nodes represent the valid links (sequences of operators) between two states identified by two separate nodes. We define a *valid link* between two nodes as follows:

Definition 24 *A* valid link *X between two distinct nodes N_1 and N_2 in a goal establishment diagram, is a sequence of grounded operators such that:*

- *only the* last *operator in X changes the truth value of either or both of the two literals in N_1*
- *$inconsistent(wp(X, N_2)\&N_1)$ is false*

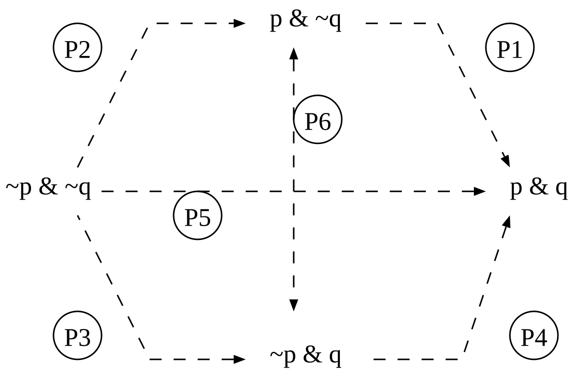

Figure 8: The goal establishment diagram for two ground literals

The effect of the first condition is to make sure a valid link does not go through other nodes in the goal establishment diagram. The second ensures that a link could form a viable plan. If a sequence X forms a valid link between two goal sets N_1 and N_2 it follows that there is a well-formed state that contains N_1 from which X can be applied sequentially to produce a state that asserts N_2.

There are five possible acyclic paths from any state asserting $\sim p$ & $\sim q$ to any state asserting p & q. The first two are defined by the node sequences:

1. [$\sim p$ & $\sim q$, p & $\sim q$, p & q]
2. [$\sim p$ & $\sim q$, $\sim p$ & q, p & q]

where in the first sequence p is established first, then q. The other three are 'odd' paths, involving an operator which affects the truth values of both p and q.

3. [$\sim p$ & $\sim q$, p & $\sim q$, $\sim p$ & q, p & q]
4. [$\sim p$ & $\sim q$, $\sim p$ & q, p & $\sim q$, p & q]
5. [$\sim p$ & $\sim q$, p & q]

For an example of a problem with a solution path such as 3., consider the problem of filling a fixed container and a movable jug of equal volume. Letting p = *'jug full'*, and q = *'container full'*, then the sequence of actions below corresponds to the path identified in 3.

[fill jug from tap, fill container from jug, fill jug from tap]

4.2.2 Identifying Blocked Links

For some goal sets certain paths may be unavailable because a link between two nodes is blocked (the operator set doesn't provide a way of changing between states that satisfy the two nodes). The main thrust of our analysis has been to use the goal establishment diagram to reason about 'necessarily' blocked paths between goal sets. We define a path between two nodes N_1 and N_2 to be *necessarily blocked* as follows:

Definition 25 *A path between two nodes N_1 and N_2 in a goal establishment diagram is necessarily blocked iff there exists no valid sequence of operators linking N_1 and N_2.*

From Figure 8, we can identify various distinct types of blocking between $\sim p \& \sim q$ and $p \& q$, including:

- type (i): only one link is blocked at P1 (or equivalently P4);
- type (ii): only one link is blocked at P2 (or equivalently P3);
- type (iii): there are no sequences of valid links joining $\sim p \& \sim q$ to $p \& q$. For example, when P1, P4 and P5 are blocked;

as well as various others (where P2, P3 and P5 are blocked, both P1 and P2 are blocked etc). We can formulate rules for ordering goals for planning using type (i) and (ii), whereas type (iii) can be used for debugging and consolidating domain models.

Type (i)

A computable condition for a blocked path at P1 is as follows: given any operator $A \in Ops$ that establishes $q \in Prds$, either A also changes the truth value of p, or the precondition of A is inconsistent when conjoined to p. This condition is formalised using Definitions 8,15, 22 and 23 as follows. Let $\alpha \in Bindings$, then:

$$\begin{aligned} &\forall A \in Ops : establishes(A_\alpha, q) \Rightarrow clobbers(A_\alpha, p) \vee \\ &\qquad inconsistent(precons(A)_\alpha \& p) \end{aligned} \qquad [i]$$

Theorem: [i] is a necessary and sufficient condition for P1 to be blocked.

Sketch Proof

That this is a sufficient condition is self evident – if it is the case, then it will never be possible to execute an establisher A and establish q from a state where p is true and q is false. To prove it is a necessary condition we will assume that the link between two nodes N_1 ($p \& \sim q$) and N_2 ($p \& q$) is blocked yet condition [i] is false, and obtain a contradiction.

Given [i] is false, we have that there exists an establisher A for q under some substitution α for which $precons(A)_\alpha \& p$ is consistent and A_α does not necessarily clobber p. Let us choose A to form this sequence, that is $X = [A_{\alpha\beta}]$ for some grounding β. We will show it forms a valid link between N_1 and N_2 as follows: firstly, only the last operator of X changes the value of any of the literals in N_1. Secondly, we have in this case:

$$wp(X, N_2) \& N_1 = wp([A_{\alpha_\beta}], p \& q) \& p \& \sim q$$

Now from our observation after Definition 23, and the fact that $establishes(A_{\alpha\beta}, q)$, we have

$$precons(A_{\alpha_\beta}) = wp([A_{\alpha\beta}], q)$$

and hence

$$wp([A_{\alpha\beta}], p \& q) \& p \& \sim q = precons(A)_{\alpha\beta} \& p \& \sim q$$

Now we are given that $precons(A)_\alpha \& p$ is consistent, and so it follows that

$$precons(A)_{\alpha\beta} \& p \& \sim q$$

is consistent, as A is an establisher for q. Hence $wp(X, N_2) \& N_1$ is consistent, and $[A_{\alpha\beta}]$ is a valid grounded sequence connecting N_1 and N_2. This gives the required contradiction.

Type (ii)

This arises when the path is blocked at position P2 only (and similarly if the only block is at P3 only). In this case we can form the necessary and sufficient condition using the functions defined above, for some $\alpha, \beta \in Bindings$, as follows:

$$\forall A \in Ops : establishes(A_\alpha, p) \Rightarrow$$
$$establishes(A_{\alpha\beta}, q) \vee inconsistent(precons(A)_\alpha \ \& \neg\, q) \qquad [ii]$$

Condition [ii] can be seen as the dual of [i]. The first disjunctive clause here corresponds to taking the path through P5 straight to the main goal, while the second corresponds to the block at P2.

Type (iii)

This arises when it is not possible to solve the conjunctive goal problem from any initial state (for example when P1, P4 and P5 are all blocked). We describe the pair $\{p, q\}$ as being *operationally inconsistent*. We have found this a major help in debugging and consolidating domain specifications, as pairs discovered this way may lead to bugs being discovered in the specification. The pair may be added to form part of the inconsistency constraints, and the compilation software re-run with this addition.

4.2.3 'Lifting' the Analysis

So far we have analysed the domain model assuming nodes are identified by ground literals. In R^3 there are over 1000 contingent, dynamic predicate instances, and hence around 10^6 pairs. Using the regularity brought by the object-centred approach – that objects of the same sort behave in the same way – we shall 'lift' the analysis from the level of the ground literal to the level of the sort variable. In R^3 there are only 20 dynamic predicate structures, resulting in less than 400 comparisons.

Definition 24 can be extended by defining a node $x \& y$, where x and y are lifted predicates, as identifying *all the well-formed states I such that $range(I)$ satisfies $(x \& y)_\alpha$, for any grounding* $\alpha \in Bindings$. Valid links between lifted nodes are therefore any valid links (in the original sense) that join the identified states. Blocked paths are defined in the same way, and when p and q do not have any sorts in common[5], conditions [i] and [ii] remain unchanged. When p and q do have one or more sorts in common, then a useful distinction can be made as to whether their sort variables are made to codesignate or are prevented from co-designating. As observed in a series of experiments with the STATIC system [46] comparing uninstantiated pairs of predicates means that all the possible co-designation relations between the

[5] Two literals have a sort in common when both literals have an argument of the same sort.

different predicates should be considered. So for example, if $p = on\_floor(Bx_i, Rm_i)$ and $q = near\_door(Bx_j, Dr, Rm_j)$ (where Bx_i and Bx_j are variables of sort Box, and Rm_i and Rm_j of sort $Room$, and Dr of sort $Door$) and these sorts have more than one instance, then they should be considered under each of the following co-designation constraints:

$$Bx_i = Bx_j \wedge Rm_i = Rm_j$$
$$Bx_i = Bx_j \wedge Rm_i \neq Rm_j$$
$$Bx_i \neq Bx_j \wedge Rm_i = Rm_j$$
$$Bx_i \neq Bx_j \wedge Rm_i \neq Rm_j$$

The number of constraints may get large if many slots are shared, but the theoretical limit is always less than (and in our experiments *much* less than) the total number of pairs of ground predicates.

4.2.4 Use of Goal Orders

The results of goal analysis can be used in different ways depending on the type of planning architecture to be used. Here we will give two examples of their use with a total order, goal directed planner.

Use of type (i) orders in planning

When a total order, goal directed planner has to establish a goal set containing the conjunctive goal $p\&q$ from a state where they are both false, a type (i) block (at P1 as shown on Figure 8) gives us the rule:

Rule (i): *'Establish q, then establish p from the advanced state containing q'*

This saves planning effort looking into the possibility of achieving q first, and avoids the on line use of goal analysis (conditions such as *inconsistent* are only executed during the compilation phase). We have also found that the rule helps towards the optimality of solutions (section 5 supplies some empirical evidence for this). Determination of type (i) and use of this rule forms the backbone of our earlier work on PRECEDE which has already been empirically validated as producing planning speed up across different domains [47, 40].

Use of type (ii) orders in planning

When wff [ii] is true (ie there is a type (ii) block at P2 on Figure 8), a rule for guiding planning to establish $p\&q$ from a state where they are both false is as follows:

Rule (ii): *'Establish p in a goal directed fashion (as any plan achieving p will also establish q)'*

Consider goals $p = next(Bx_1, Bx_2)$ and $q = next(Rb, Bx_3)$ from R^1, the example domain with only one robot, and with the binding constraint that $Bx_1 = Bx_3$. These predicates would display a type (ii) relationship as it would be impossible to establish p without at some point making q true also under the binding constraint. To establish both predicates in R^1 therefore, it is sufficient to form a plan aimed at $next(Bx_1, Bx_2)$ only.

algorithm Order_Goals
In Operator complete model $\mathcal{M}(Sorts, Objs, Prds, Exps, Invs, Ops)$
Out ODS: Goal Orders, CYC: Cyclic Goal Orders;

1. $ODS := \{\ \}$; $CYC := \{\ \}$;
2. for each $l \in Prds_D$ do
3.   for each $l' \in Prds_D$ do
4. $codesignation(l, l', B)$;
5.   for each $\mathcal{C}$ in B do
6. $Ops_l := \{O_\alpha \mid O \in Ops \,\&\, establishes(O_\alpha, l)\}$;
7.   if $\forall A \in Ops_l : (clobbers(A, l')$ under constraints $\mathcal{C}$ or $inconsistent(precons(A)\&l'\&\mathcal{C}))$ then
8.   if $order(l', l, \mathcal{C}) \notin ODS$ then
9. $ODS := ODS \cup \{(l, l', \mathcal{C})\}$
10.   else $CYC := CYC \cup \{(l, l', \mathcal{C})\}$; $ODS := ODS \backslash \{(l', l, \mathcal{C})\}$
11.   end if;
12.   end if;
13.   end for;
14.   end for;
15. end for;

end.

Figure 9: Outline Algorithm for generating Goal Orders

4.2.5 Outline Algorithm for Generating Goal Orders

Figure 9 shows an outline algorithm for generating goal orders. The algorithm takes as input the components of an operator complete domain model (refer to section 3 for their definitions). It outputs two sets of orders between pairs of predicates (with binding constraints if appropriate) one resulting from type (i) orders, and the other cyclical orders representing type (iii) blocks. The algorithm steps through each dynamic predicate and compares it against every other dynamic predicate (steps 2-3). This pair is 'lifted' and hence the next step is to find the set of codesignation constraints between variables of the same sort in the two predicates (step 4). If there are no variables of the same sort that are common to the two predicates then the set of codesignation constraints is empty. For each different set of codesignation constraints $\mathcal{C}$, steps 6-10 operationalise condition [i]. Firstly, a set of operators that establish the literal of interest l is constructed, then a check is made to see if all of these operators clobber l' or have preconditions inconsistent with l', in the context of the constraints $\mathcal{C}$. If it is the case that *all* establishers for l clobber l' or have inconsistent preconditions then an order $(l, l', \mathcal{C})$ has been identified. In step 9 any orders are added to the output ODS where appropriate. Type (iii) orders are generated after discovering there are orders between $(l, l', \mathcal{C})$ and $(l', l, \mathcal{C})$. This cycle is then added to CYC and the order $(l', l, \mathcal{C})$ is removed from ODS.

4.2.6 Possible Goal Orderings

The analysis using the goal establishment diagram can be extended in a natural way to model techniques based on *possible* orderings such as ABGEN [47]. These techniques are based on possible orderings between pairs of literals p and q as follows: if the establishment of p by a planner cannot possibly affect the truth value of q (given a goal directed search, say) then q can be established before p without the risk of a goal violation. This kind of heuristic was used originally by Knoblock for hierarchical planning - q would in this case appear in a higher level of abstraction than p. p would therefore be a 'detail' whose establishment could be safely left to a lower level of the abstraction space [36].

Consider the space of all valid paths that can be generated in a goal directed search to establish $p\ \&q$ from any state containing $\sim p\ \&q$ (P4 in Figure 8). If *none* of these paths traverse through states satisfying either of the other two nodes in the diagram then we can assert the heuristic that q can be established before p without the risk of it being violated when p is established. We return to the use of possible orderings in section 5.3.3.

5 Evaluation of the Method

The method has been used to construct object-centred specifications for a range of planning domains, some familiar and some novel. These include R^n, introduced earlier in this paper, Russell's Tyre World [49], STRIPS-worlds [50], a job shop scheduling world (similar to the PRODIGY test domain [48]) and a warehouse world [34]. We have used tools presented in section 4 to help acquire, validate and compile these domain models and have explored the effect upon planner performance of the operationalised models using various planning engines. The purpose of this section is to present some of the results of these activities and demonstrate the benefits of the use of the method in a number of planning domains. Not surprisingly we devote most space to the results of the dynamic tests of the compiled domain models, as an account of the benefits of the use of a development method using a small set of test domains tends to be anecdotal.

As a guide, the organisation of this section is as follows: in section 5.1 we introduce the 4 domains that will be used in testing. In 5.2 and 5.3 we evaluate the method's use in initially encoding the models, and compiling them to produce macros and goal orders. In 5.4 we evaluate the method empirically by analyising the results of random planning tests using compiled and uncompiled configurations, and finally in 5.5 we summarise the benefits of the method. Note that all the software used was implemented in Quintus Prolog running on a Sun IPX workstation, and all CPU measures are taken from this configuration.

5.1 The Four Test Domains

The most complex test domain in terms of structure, and number of distinct actions, was the **DR^3 World**, that is the domain described in section 2, with n = 3 (a state of this domain is illustrated in figure 3). The most complex domain in terms of goal interactions was the **Tyre World** [49], which concerns the ordering of repair actions on an automobile. The central problem in the domain, that of changing a wheel, as typically formulated could be termed 'laboriously serialisable' [3], since there are very few orders which permit each goal to be established and then preserved whilst the remaining goals are established. These two domains contrast well: with respect to *goal interactions*: the Tyre World abounds with potential goal interactions, whereas in the R^3 World the probability of two goals interacting is relatively low (though still significant); with respect to *structure*, the Tyre World has few static predicates and invariants, in contrast to R^3 which has many. In between these extremes lie the two other domains we used: the familiar **STRIPS-world** and the **Extended STRIPS-world**. The former is a variant of Sacerdoti's STRIPS-world [50], which consists of 7 rooms connected by 10 doors in which a robot is capable of opening doors and moving any of 3 different boxes. The latter is an elaboration of a 6 room STRIPS-world which we adapted from an example domain used to test the PRODIGY system [8]. It introduces a number of complications to the domain: doors that can be locked and unlocked; and keys must be moved around by the robot to unlock locked doors.

5.2 Initial Construction of Domain Models

Construction of operator complete models involved looping through the method's steps, described in section 2, several times. Old literal-based specifications gave us a partial description of the domains, and to an extent the encoding resembled a 'reverse engineering' task. The tool support revealed many errors in our initial encoding: from simple syntactic errors, shown up using predicate cross-checking, to the omission of conditional effects, shown up by checking that the operators always produce a consistent state when input with a consistent state. Often these errors would be hard to spot using dynamic testing alone.

For example with R^3, carrying out step 5 with the help of tool (e) in category 1 (cf section 4), resulted in 55 inconsistency constraints. Use of the goal order generation tool in step 7 of the method subsequently led to the discovery of several new constraints, as the production of cyclic orders indicated a lack of a constraint *or* an omission in the encoding of the operator set. Inclusion of the new constraints (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 by the end of step 7.

Analysis of random tasks generated by tool (f) in category 1 (cf section 4) for the domain models showed that the invariants built up in step 5 eliminated a great many potentially inconsistent or impossible goal sets. Nevertheless, the invariants used in the dynamic tests described later in this section were still incomplete: of the 100 hardest tasks randomly generated for R^3 (this set is called $RDM7$ below), 12 were found to amount to 'impossible' problems. In a batch of 100 that were generated *without* the use of invariants, however, 54 were found to be impossible problems.

As another example, we initially supplied the Tyre World with 15 inconsistency constraints. The following are two of the constraints that were immediately apparent:

$$inconsistent\_constraint(loose(Nuts, Hub) \& tight(Nuts, Hub))$$
$$inconsistent\_constraint(wheel\_in(Wheel, Hub) \& wheel\_on(Wheel, Hub))$$

but other constraints, such as the pair given below:

Model	*Objs*	*Sorts*	*Prds*	*Exps*	*Invs*	*Ops*
R^3	41	8	27	20	204	25
Tyre	8	7	22	20	36	17
$ext - STRIPS$	19	6	22	18	150	19
$STRIPS$	21	4	11	8	106	8

Table 1: Size of the Domain Models' Sort Abstracted Components

$$inconsistent\_constraint(have\_nuts(Nuts)\&tight(Nuts, Hub))$$
$$inconsistent\_constraint(free(Hub)\&loose(Nuts, Hub))$$

were not as obvious, but were revealed with the help of type (iii) blocks identified during goal order generation (cf section 4.2). In the final model there were 27 inconsistency constraints.

To give a feel for the size of each domain model we indicate the number of elements for each of the components in the model in Table 1. As the tool support was not available at the time, our models were all built and tested without the use of sort hierarchy. Thus all the sorts used were primitive and all the predicates' slots were restricted to a primitive sort. Use of the sort hierarchy mechanism would on the whole have made the components more compact, without affecting the number of *ground* elements of these components.

For R^3 especially, the number of ground elements in a component was very much larger than the figures in the table: here the approximate size of $Prds_G$ was 4,500, there were in the region of 1,000 operator instances (in fact many more if one were to consider variations dependent on conditional effects) and the number of distinct substates was around 1,700. The total size of the state space was thus very large indeed, although difficult to calculate accurately in the presence of state invariants. Consider the sort *Box*: there were 87 distinct substates for a given box, and with 6 boxes in this domain the number of possible states made up of box objects alone was of the order of 87^6.

5.3 Compilation of Domain Models

All the domain models were compiled in the sense that a set of macros, necessary and possible goal orders were generated for each one. The size of the sets and the approximate generation times are shown in Table 3. As predicted, the use of compilation tools was beneficial in detecting bugs in the domain encodings, tightening encodings by indicating new invariants, and delivering speed-up in plan generation (as demonstrated by the results in section 5.4). Additionally, this stage is useful in evaluating the utility of the choice of a model's components (e.g. what sorts and predicates are static and what are dynamic). One might make design choices based on, for example, the number and utility of the macro set so generated.

5.3.1 Construction of Macro Tables

Using the notation introduced in the algorithm of Figure 7, the number of *entries* in a macro table is bounded by the following formulae:

Model feature that increases	Effect on the size of the macro table
no. of dynamic objects	stays constant
no. of dynamic sorts	increases, bounded by $K\times \mid Sorts_D \mid$
no. of substate classes	increases, bounded by $K\times \mid Exps \mid^2$
no. of static objects	increases polynomially

Table 2: An indication of the space complexity of a macro table (K is some constant)

$$\sum_{s\in Sorts_D} (ins^s)^2 - ins^s$$

that is, the number of entries is of the order of the square of the number of elements that result from partially instantiating substate classes with static objects. The crucial point (and this is what circumvents the exponential bottleneck noticed in Ginsberg's critique of Universal Plans [27]) is that our approach leads to a separate macro table being built for each sort, as macros are indexed according to the substates of an object rather than the whole planning state. The theoretical growth of macros with model size, derived from the algorithmic description, is summarised in Table 2.

The results of macro compilation in our sample worlds are summarised in the first part of Table 3. The number given is the total number of macros generated for that model. The Tyre World has no static predicates apart from standard typing information (that is facts such as $wheel_1$ *is_of_sort wheel* etc) and so the macro set is small, and their potential for use in planning speed-up is low. It should be stressed, however, that macro generation in domains such as the Tyre World helps clarify goal structure and validate the choice of substate classes. Where a solution to a sort-abstracted planning problem fails as part of the macro generation process, it often uncovers an unreachable goal or throws doubt on the choice of substate class expressions.

The maximum time taken for the macro generation phase was for R^3 which took several hours. This figure is acceptable since it should be seen in the context of the overall time taken to create the domain model, because compilation need only be performed when the domain is captured (and occasionally when the domain model is changed). On the other hand there are potential complexity problems as macro generation involves search and consequently is inherently exponential. The central point is that this search occurs in a sort-abstracted planning space which suppresses the details of all but one of the dynamic sorts.

Another potential problem that our tests revealed was the question of how many distinct macros should be indexed by a task configuration. Given that it would be intractable to store every generalised solution (including non-optimal macros) to every task configuration, we allowed the macro generation algorithm to produce only the shortest solutions. Recognising that this offers only heuristic coverage, we would have to allow a planner recourse to the original operator set should the macro retrieved fail (it may be possible that a locally non-optimal macro is required to fit in with the rest of a plan).

Generation of:	Model	Number	CPU Time
Macros	R^3	2385	several hours
	Tyre	32	
	extended-STRIPS	1926	
	STRIPS	824	
Necessary Orders	R^3	98	several minutes
	Tyre	30	
	extended-STRIPS	105	
	STRIPS	18	
Possible Orders	R^3	56	several seconds
	Tyre	23	
	extended-STRIPS	65	
	STRIPS	19	

Table 3: Compilation Results

5.3.2 Generation of Necessary Goal Orderings

Of interest during compilation is the computational complexity of the algorithm in Figure 9, the number of goal orders that will be compiled by it, and an assessment of the potential utility of the goal orders on planning performance.

Let N = number of dynamic predicates in the domain model, and M = the number of grounded dynamic predicates. Then c, the number of comparisons that the algorithm in Figure 9 has to make is bounded as follows:

$$N(N-1) \leq c \leq M(M-1)$$

In practice c turns out to be much nearer N^2 than M^2. This is illustrated in the example section 4.2.3: while the number of constraints gives 4 comparisons, the number of grounded comparisons would be over 600.

The dominating component during each comparison is the evaluation of the *inconsistent* function. Checking for inconsistency in a first order theory is of course undecidable in general, although in our implementation the inconsistency constraints are limited to the form used for the examples in this paper. In practice, the inconsistent function (embedded in the PRECEDE tool) is used *after* the state invariants have been specified to check operator consistency. The tractability or otherwise of executing

$$inconsistent(precons(A))$$

for each operator 'A' in the model would then be established as it is a condition of operator consistency that this evaluate to false. If this were computationally prohibitive then this would indicate to the modeller at an early stage the need to re-design the domain model. On the other hand, if the execution time for this is acceptable then this would suggest that the execution time of

$$inconsistent(precons(A) \& p)$$

Figure 10: Graph of Type (i) Goal Orders for the Tyre World

which is at the heart of PRECEDE, would also be acceptable.

The test results of using the PRECEDE tool to generate necessary goal orders is summarised in table 3. The number is the total number of goal orders that were generated during this phase and the maximum time taken for generation was found to be of the order of minutes. It is interesting to note the difference in the number of necessary goal orders across models. For the extended STRIPS-world the relatively large number of goal orders reflects the higher degree of interaction between goals in this domain. In R^3 the introduction of the 3 robots greatly increases the average number of possible ground establishers per goal and this explains the lower number of necessary goal orderings identified for this model.

As for predicted utility of the PRECEDE goal orders, we would expect the impact of using goal orderings to be related to the number of orders identified relative to the size of the domain model. Hence we would expect the orders generated for our extended STRIPS-world to have more effect on planning performance than the orders generated for domain R^3.

Example

The declarative specification of the main problem in the Tyre World is posed using the following set, representing a conjunction of predicate goals:

$$\{closed(boot_1), jack\_in(jack_1, boot_1), pump\_in(pump_1, boot_1),$$
$$wheel\_in(wheel_1, boot_1), wrench\_in(wrench_1, boot_1), wheel\_on(wheel_2, hub_1),$$
$$tight(nuts_1, hub_1), inflated(wheel_2)\}$$

which can be paraphrased as: *'replace a flat tyre on a car and put all the tools away in the boot'* when the initial state asserts that $wheel_1$ has a flat tyre on hub_1, and all the appropriate tools are in $boot_1$. The operators that make up the application's model allow a wheel to be changed by jacking up the wheel, unbolting the wheel nuts and so on.

The PRECEDE tool generated 30 type (i) orders (as defined in section 4.2) between pairs of predicates, and these are shown abstractly (that is without details of variable bindings) in Figure 10. Each arrow corresponds to a type (i) order. Type (i) orders are *not* strictly associative, in the sense that if $a < b$ and $b < c$ then it may not necessarily be the case that to solve goal $\{a, c\}$, subgoal a has to be solved first in a solution sequence. If the goal is $\{a, b, c\}$, however, then their ordering is total. When applied to a set of predicates the orders reduce the set to a subset such that predicates in the subset can be established in any order, and subsequent solution of the complement of the subset from the resulting advanced state will not necessarily clobber any of the solved predicates. Using the diagram we can see that the orders reduce the eight goals to three:

$$\{wheel\_in(wheel_1, boot_1), wheel\_on(wheel_2, hub_1), inflated(wheel_2)\}$$

The necessary orders are used at every planning node to order goal establishment in the same fashion.

The diagram also provides an opportunity to check goal structure, and the validity of the operators from which the orders were generated. For example, there is no precondition to do with inflating a tyre *before* putting it on a hub in the operators, yet this might be a valid constraint.

5.3.3 Generation of Possible Goal Orderings

ABGEN is an implementation based on the ALPINE algorithm presented in [35, 36], which generates goal orders on the basis of possible interactions between goals (ABGEN was described in section 4.2). We included it in the tests as it complements necessary orders when choosing which goal to establish next during planning: given a set of goal predicates, necessary orders reduce goal choice by removing goals whose establishment would be undone by another goal's establishment, as shown in the Tyre World example above. Possible orders are then used on the reduced goal set to further limit choice. Any predicates whose establishment may *possibly* clobber some other goal in the reduced conjunction are removed from the goal set. The number of possible orders produced by the ABGEN tool for the tests domain models are shown in Table 3.

5.4 Using the Domain Models in Planning

5.4.1 The FMD Planner

The planner constructed to dynamically test the domain models was a goal directed, total order planner called FMD. Assuming goal conjunctions are serialisable, FMD synthesises the use of goal orderings, macros and sort information to solve a planning problem. By effectively trading optimality for speed of solution construction the FMD planner demonstrates very good planning performance and this was the main reason we selected it to dynamically test the models. Our rationale was that its relative plan generation efficiency would provide a good test for the effect of domain compilation on planner performance. Further, we predicted that the disadvantages of a total order planner (viz. the difficulty in solving interacting goals and the tendency to produce overlong solution sequences) would be ameliorated by the use of a processed domain model. We could equally have selected other planning algorithms for our test platform and some examples of these are discussed in section 7.5.

FMD works on goals independently, advancing the plan state after partial solutions are found. For simplicity, we start by describing an outline version of FMD shown in Figure 11. To illustrate the workings of the planner, and the benefits of the combined use of goal ordering rules and macros, we will consider FMD's behaviour at a typical planning node when input with a compiled operator complete domain model, an initial well-formed state, and a goal condition represented as a partial mapping between object identifiers and sets of substates.

The algorithm searches through a space of open nodes, where each node has the form:

$$node(N, A, I, G, OS, P)$$

and has fields N = identifier, A = parent's identifier, I = current state, G = current goal condition, OS = partial solution, P = the node's purpose (the macro or operator whose preconditions the node was established to solve is invariably the purpose of the node). The partial solution slot contains the operator sequence that already has been applied to the node's initial state to produce the node's current state I.

Step 1 initialises the set of open and closed nodes in the planner's search space. Using ODS (step 5 in figure 11), the orderings of goal predicates generated during the compilation process, a substate gs is picked that should be solved first. gs will be a member of one of the set of substates mapped to by some object c say, from the range of goal condition G. This process of gs's selection is computationally cheap as it corresponds to sorting using a pre-defined partial order. The predicates in $range(G)$ are ground, hence the matching process is straightforward. In step 6 the entry in the macro table whose index matches with (I_c, gs) is retrieved. This process is a simple look-up operation with linear time complexity as matching is one way and therefore does not lead to any increase in complexity order. A macro operator retrieved this way will be instantiated to one or more ground macros (set M) according to the goal node. If a member of M is applicable it will be applied to I, otherwise new nodes will be created to establish the weakest preconditions of each member of M in a backchaining fashion (steps 8 et al).

If a node is picked whose current state solves the goal G, then (steps 19 and 20) its solution OS is applied to N's parent's current state, followed by the purpose of the node's existence (typically a macro). Then the parent node is re-asserted as an open node (step 21) and any of its ancestors are closed.

In addition to the outline design, the algorithm was elaborated with some important extensions.

- firstly, it contained the admissible heuristic of *loop detection*: the planner was able to check for loops in both the head and the tail plan (that is by checking the developing plan OS in a node, and by checking the ancestry of a new node) and take appropriate action;
- in step 6 FMD uses Ops rather than MT, and resorts to normal operator backchaining, in two cases (i) if MT is empty as is the case where $\mathcal{M}$ is uncompiled; (ii) if $G(c) \neq \{gs\}$, that is if the range of c is a disjunction of two or more substates.
- in step 20 the combined application of P and OS may fail due to goal violations - a problem that is much more acute when FMD is input with an uncompiled model, that is without the benefit of goal orders. The planner was equipped to detect and deal with solution sequences that clobbered protected goals by allowing solutions to be developed to different orderings of conjunctive goals.

Finally, three variations of FMD were created by adjusting the choice strategy of step 24. The variations were called *HS*, *DS* and *BS*, and are defined as follows:

  - *HS*: a heuristic search strategy that used the following heuristic to evaluate which candidate planning nodes to expand: favour the expansion of open nodes with either a single goal or a partial solution before any other nodes (and default to breadth-first search).
  - *DS*: a depth first search strategy
  - *BS*: a breadth first search strategy.

```
algorithm FMD
In Operator complete model M(Sorts, Objs, Prds, Exps, Invs, Ops);
Init: a well-formed state in M;
Goal: a well-formed goal condition in M;
MT: a macro table for M;
ODS: a set of goal orders for M
Out SOLN: Operator sequence

O_Nodes := {node(root, null, Init, Goal, [ ], null)}; C_Nodes := { }
Remove node(N, A, I, G, OS, P) from O_Nodes;
while (N ≠ root or ¬ achieve(I, G)) do
      if ¬ achieve(I, G) then
         Determine a substate gs using ODS, where gs ∈ G(c) for c ∈ Objs and I(c) ∉ G(c);
         Pick m from MT such that (I(c), gs) satisfies the index of m_α for some α ∈ Bindings
         M := {m_αβ : β ∈ Bindings is a grounding of m_α}
         if there exists an m_g ∈ M applicable to I then
            I_advanced := Apply m_g to I;
            O_Nodes := O_Nodes ∪ {node(N, A, I_advanced, G, OS ++m_g, P)}
         else
            C_Nodes := C_Nodes ∪ {node(N, A, I, G, OS, P)};
            for each m_g ∈ M do
                 Wp := the weakest precondition component of m_g ;
                 O_Nodes := O_Nodes ∪ {node(New, N, I, Wp, [ ], m_g)}
            end for;
         end if;
      else
         Retrieve node(A, Ap, Ip, Gp, OSp, Pp) from C_Nodes;
         Ip_Advanced := Apply P to (Apply OS to Ip);
         O_Nodes := O_Nodes ∪ {node(A, Ap, Ip_Advanced, Gp, OSp ++OS ++P, Pp)};
         Move all nodes which have ancestor A from O_Nodes to C_Nodes
      end if;
      Choose and remove a node(N, A, I, G, OS, P) from O_Nodes;
end while;
SOLN := OS
end.
```

Figure 11: Outline Algorithm of FMD

Example: Assume we use FMD with a compiled form of R^3, and consider a goal condition G, which is a mapping between four objects *box2, tom, harry, door45* and and their sets of possible substates. Assuming G is input in literal-based form as follows:

$$\{on\_floor(harry, room1), on\_floor(box2, room5), next(tom, key2),$$
$$near\_door(box2, door56, room5), locked(door45)\}$$

then the goal orders ODS will identify *on_floor* as the first predicate to be solved, with $c = box2$ as the domain element of G. That is :

$$G(box2) = \{\{on\_floor(box2, room5), near\_door(box2, door56, room5)\}\}$$

and hence:

$$gs = \{on\_floor(box2, room5), near\_door(box2, door56, room5)\}$$

As $G(c)$ is a singleton, the next step is to retrieve the entry from the macro table whose index matches $(I(box2), gs)$. Assuming

$I(box2) = \{on\_floor(box2, room1)\}$

then the entry retrieved from the macro table, and instantiated with the instance $box2$ is:

$$[pushtodoor(Robot, box2, door12, room1),$$
$$pushthrudoor(Robot, box2, door12, room1, room2),$$
$$pushtodoor(Robot, box2, door25, room2),$$
$$pushthrudoor(Robot, box2, door25, room2, room5),$$
$$pushtodoor(Robot, box2, door56, room5)]$$

If this macro's preconditions are satisfied by I for some value of parameter $Robot$ then it will be applied, otherwise new nodes will be created to recursively establish them.

Overall, FMD's search space is reduced by the use of macros cutting out search during operator backchaining, and the use of goal ordering rules linearising sets of goal predicates that make up the main goal or a macro's preconditions. Use of macros and goal orderings at each node is computationally cheap, and the size of macro tables and goal ordering sets does not grow unmanageable with the rise in numbers of sorts or dynamic objects. Hence speed-up does not seem to suffer from the 'utility' problem as described in [43].

5.4.2 Planning Configuration

Twenty four planning configurations were constructed by 'bolting' the four domain models (R^3, Tyre, STRIPS, and extended STRIPS), in both compiled and uncompiled form, onto each of the three different variations of the FMD planner (HS, DS and BS). The compiled forms would therefore have use of the previously generated macros, necessary and possible goal orders.

To test these configurations we generated one thousand problems as follows:

- for each of R^3, the extended STRIPS-world and the STRIPS-world we generated three hundred random problems which were split into divisions of one hundred called RDM3, RDM5 and RDM7. RDM7, for each model, contained problems consisting of an initial state, and a goal set containing 7 randomly generated positive, ground literals. RDM5 and RDM3 were formed likewise, with their goal literal sets being of size 5 and 3 respectively. For example, problem goal set NO. 36 for RDM7 of R^3 was[6]:

  $$\{key\_on\_floor(key1, room7), status(light3, room3, on), unlocked(door35),$$
  $$robot\_in(tom, room3), key\_being\_held(key3, harry, room4),$$
  $$key\_next\_key(key1, key5), box\_near\_door(box1, door23, room3)\}$$

  which corresponds to achieving goal states for six objects

  $\{key1, light3, door35, tom, key3, box1\}$.

  Some of the objects referred to in a goal set have a unique goal substate (here key1, light3, key3, box1) while others (door35, tom) can be satisfied by a range of substates.

  Initial states and goal sets were all generated to be well-formed using the model's state invariants. Further, initial states were randomly generated, but from a reduced state space where in particular doors were not allowed to be locked (unless one of the goals of the corresponding literal set was to unlock the door). This was to attempt to overcome the problem of impossible tasks where, for example, a key is required but is inaccessibly locked away in a room.

- for the Tyre World a test file was generated consisting of 100 random permutations of the 8 goal predicates described in section 5.3. The initial state (the flat tyre!) in this case was kept fixed.

The results of the experimental tests are summarised in figures 13 through 20. For each domain there is a separate figure representing the results measured in terms of CPU (the number of seconds taken to generate a correct solution) and nodes (the number of new partial plans created using steps 13-16 of the FMD algorithm in Figure 11). For the STRIPS-world, extended STRIPS-world and R^3 worlds each figure contains 3 graphs which represent the results (in terms of CPU or nodes as shown on the legend) for each of the three different random problem samples, labelled $RDM3$, $RDM5$ and $RDM7$. For the Tyre World each graph contains one figure for the problem sample, called $RDM8$, consisting of 100 random permutations of the problem given above. Within each graph continuous and dashed lines have been used to plot the results for the six different versions of the domain model, as shown in the legend (figure 12).

Table 4 summarises the average solution length of the problems solved within the resource limit of 60 cpu seconds (120 cpu seconds for R^3). Note that the number of problems solved, if it is significantly less than 100, has the effect of reducing the average length (as the problems actually solved tend to be those with a shorter solution). Thus the average solution length of solutions from the uncompiled configurations in R^3 is small because only the simpler problems were solved.

[6] As previously mentioned the tests were encoded without use of a sort hierarchy and hence the predicate names used are more elaborate than those in our previous examples in R^n.

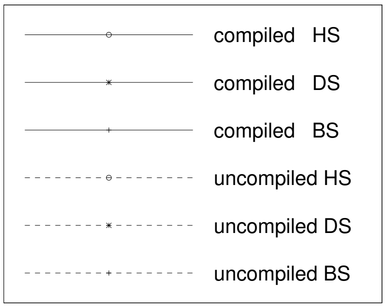

Figure 12: Legend for Results Graphs

5.4.3 Discussion of Results

STRIPS-world

The results for this domain model, measured in terms of CPU and number of nodes expanded are shown in Figures 13 and 14, These results show that for the compiled models the solution of conjunctions of goals is virtually trivial. Out of the 300 problems generated, the poorest compiled configuration (depth-first search) solved each problem in more than 85% of the batch in 3 seconds or under. Even the hardest 15 % were all solved within 16 cpu seconds. Compiled *HS* performed the best, solving each of the 300 problems in 7 seconds or less, with an average CPU expenditure of 1.7 seconds per solution. The uncompiled configurations were in contrast less successful, with the best configuration (*HS*) failing to solve 3 of the 300 problems within 60 seconds of CPU time. The results of the uncompiled configurations varied appreciably with choice of search strategy, and predictably the graphs show breadth-first search poorer in plan generation time and space usage, and the depth first search producing the poorest solutions.

An interesting observation is that the number of nodes expanded during plan generation, by all of the compiled planning configurations is consistently low. For example, for more than three quarters of the problems in sample $RDM7$, the average number of nodes expanded during planning is less than the length of the resulting solution for the problem (this can be seen by comparing the average solution lengths given in Table 4 with the graphs in Figure 14). The most striking contrast in the use of space is that while the compiled breadth-first search solved all 100 RDM7 problems within a space limit of 90 nodes, the same uncompiled configuration only managed to solve one problem out of the same 100 problems within this space limit.

One might expect solutions made up of macros to be overly long. Although FMD does not produce provably-optimal solutions we believe they are close to optimal since (i) we attempted to laboriously find optimal solutions to some tasks by hand and compared these with the generated solutions (ii) the optimal solution of one literal in this model can potentially take an operator sequence up to 20 in length, and our hand validated tests show an average of between 7 and 8 operators. This is in line with the average solution sizes of the problem set solutions as shown in the table.

Extended STRIPS-world

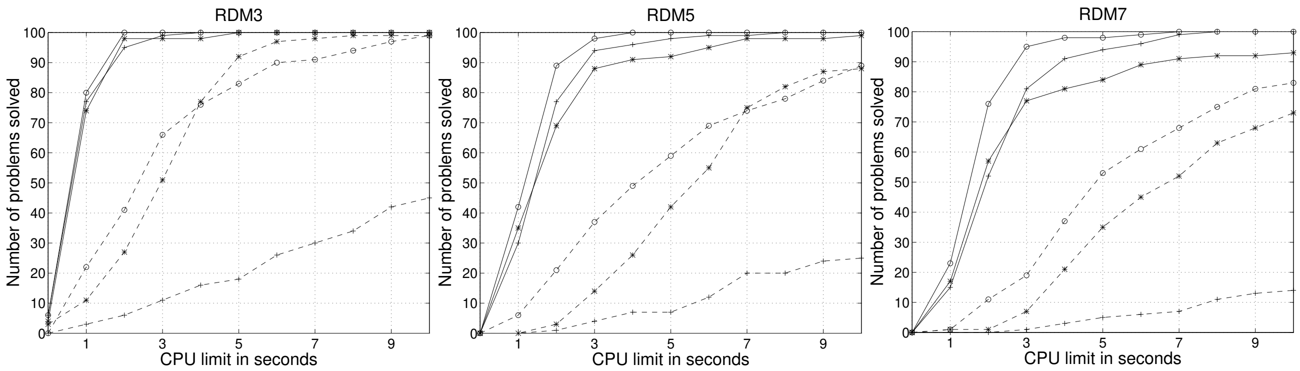

Figure 13: CPU: STRIPS world

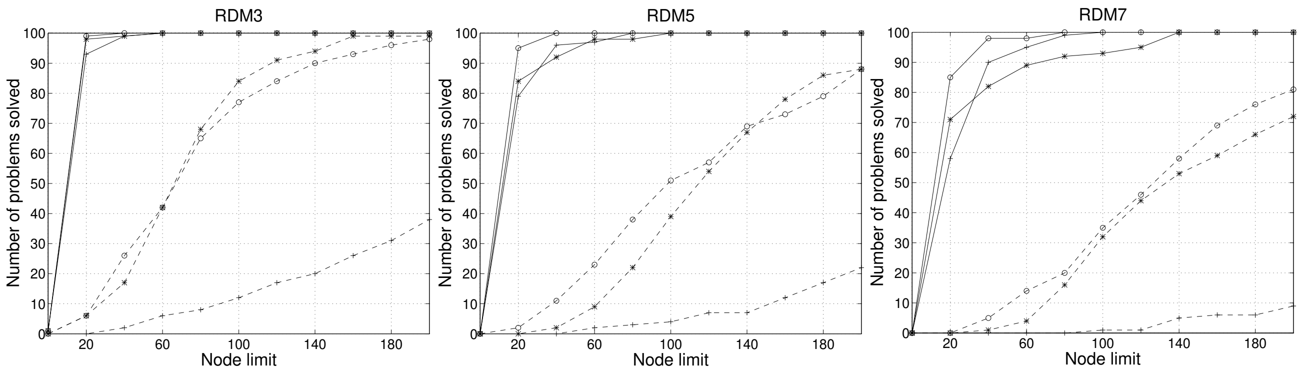

Figure 14: NODES: STRIPS world

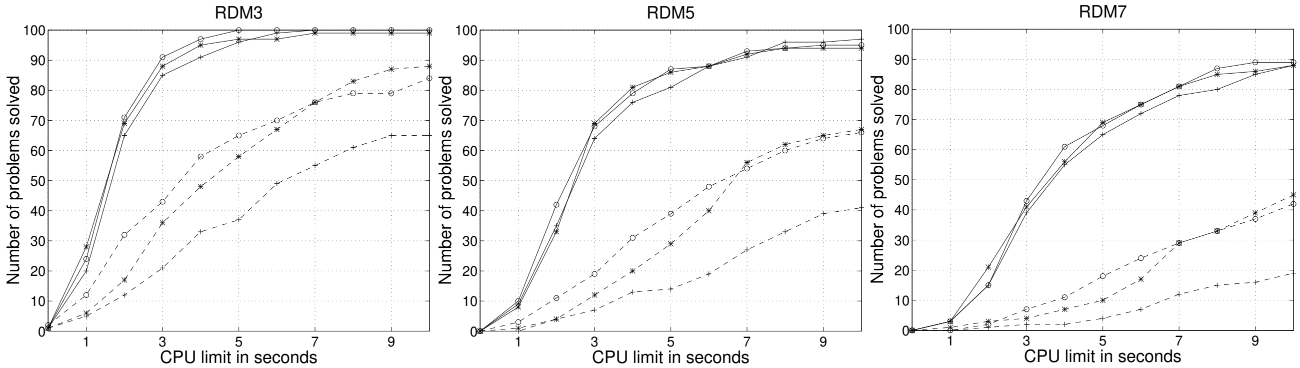

Figure 15: CPU: Extended STRIPS world

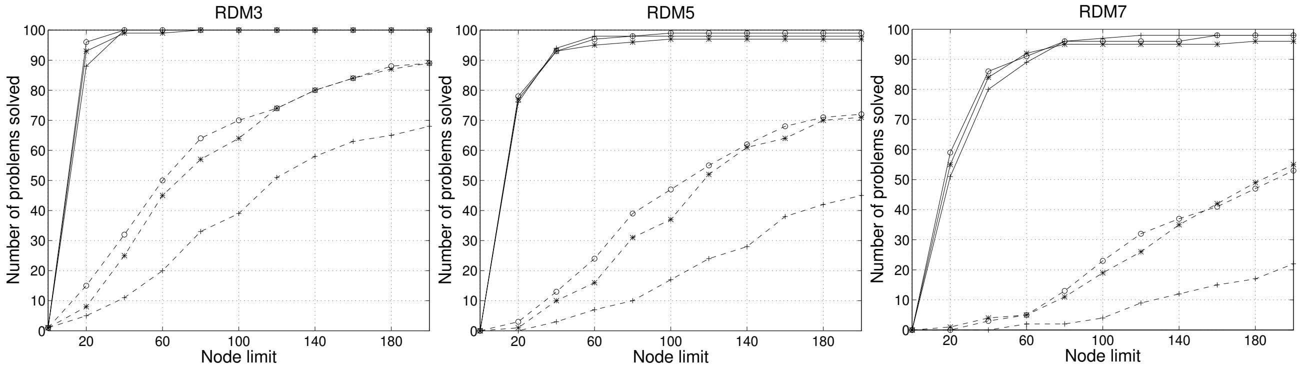

Figure 16: NODES: Extended STRIPS world

As with the results for the STRIPS-world, the results show that the overall effect of domain model compilation is a dramatic improvement in the solution of the problem sample. Consider again the results for the hardest problem sample, $RDM7$, where the problems are all goal sets of size seven. The results in Figures 15 show that around two thirds of the $RDM7$ sample problems are solved within 5 seconds, in which time the best uncompiled configurations have solved less than 20% of the problems. For the other problem samples $RDM5$ and $RDM3$, the performance improvements are as impressive.

Comparison of the results in Figure 16, which plots planning performance against the number of nodes expanded, and the average length of solution plans (shown in Table 4) reveals that the number of nodes expanded during plan generation by the compiled planning configurations in this planning domain is consistently low – across the 3 problem samples, for at least 80% of problems the number of nodes expanded is less than the average length (measured in terms of number of operators) of solution plans.

R^3 World

The results for this domain are represented graphically in Figures 17 and 18. The additional complexity in this multi-robot world means that problems are considerably harder than the those in the STRIPS-worlds. Also, in this model some problems set were impossible to solve despite passing the tests of the static invariants. As mentioned above, $RDM7$ was found to contain at least 12 impossible problems.

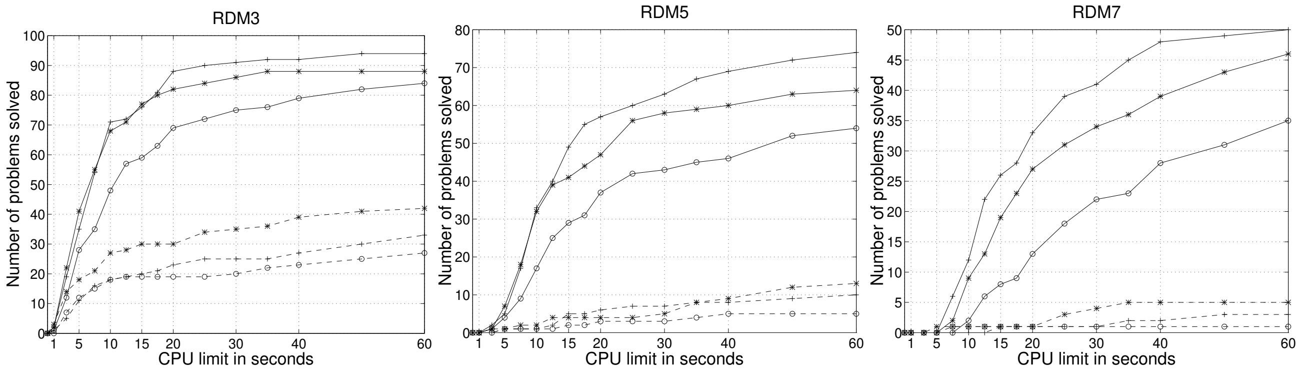

Figure 17: CPU: R^3 world

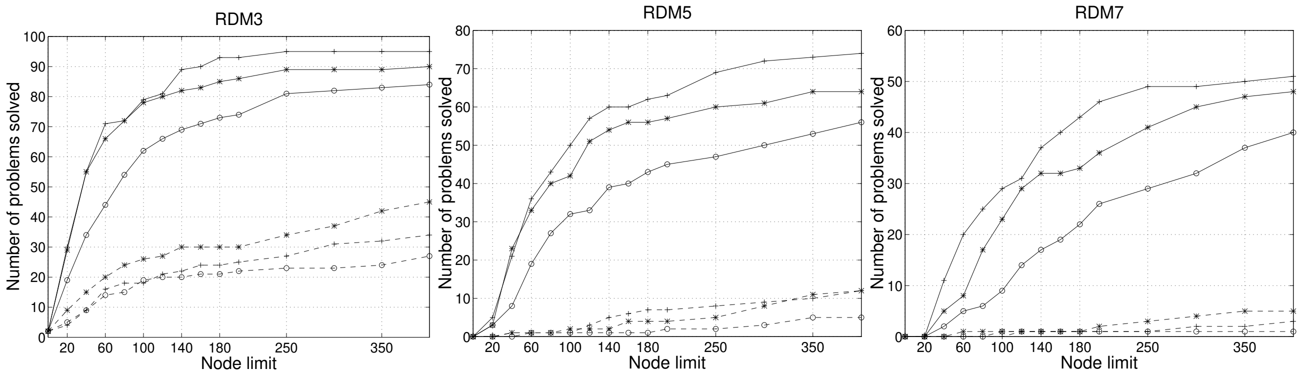

Figure 18: NODES: R^3 world

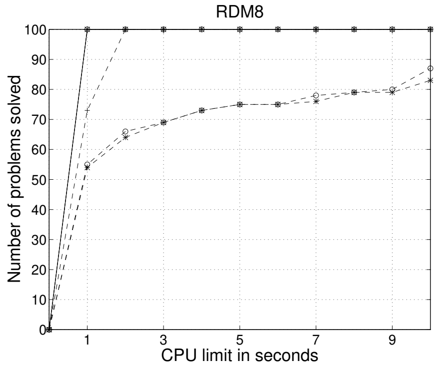

Figure 19: CPU: Tyre World

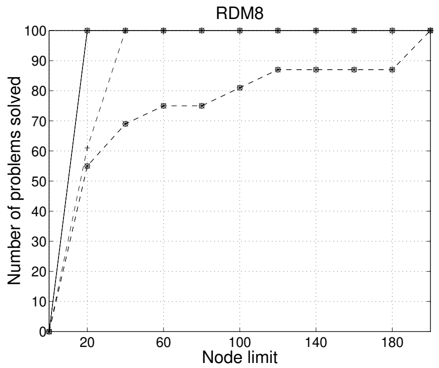

Figure 20: NODES: Tyre World

Again there is a marked difference in the solution rate between the compiled planning configurations and the uncompiled configurations, and solutions to the harder problems from the uncompiled configuration are particularly poor in that the gradient of the graph levels off - much more resource given to these problems is clearly not going to lead to many further successes.

Although the compiled planning configurations fair much better, the fact that in R^3 only two thirds of the problems were solved within the CPU limit suggests that with some problems the compilation techniques were not as useful. Also, the performance of *BS* is consistently poorer than *HS* and *DS* across the three problem samples for this domain. This is interesting since for all the other domain models the performance of the three compiled planning configurations is very similar – a result which suggests that the impact of model compilation is to some extent independent of the particular search method. The reason seems to lie with the amount of *macro usage*. In the STRIPS and extended-STRIPS worlds macro usage was around 80 % (i.e. in 20 % of nodes generated FMD reverted to its original operator set to establish a goal), whereas with the R^3 world macro usage had dropped to 40 %. Thus the compiled configuration R^3 is starting to exhibit the behaviour of an uncompiled configuration with *BS* faring worst.

Tyre World

The results of the tests in this domain are shown graphically in Figures 19 and 20 and

		DS				BS				HS			
		comp		uncmp		comp		uncmp		comp		uncmp	
		len	suc	len	suc	len	suc	len	suc	len	suc	len	suc
STRIPS	RDM3	19	100	36	99	19	100	21	100	19	100	20	100
	RDM5	32	100	64	100	29	100	28	88	29	100	31	100
	RDM7	42	100	76	94	36	100	35	82	35	100	38	97
Extended STRIPS	RDM3	22	100	36	99	20	100	22	95	20	100	23	98
	RDM5	34	97	55	92	31	98	32	81	35	99	34	88
	RDM7	49	96	76	86	42	98	44	72	42	98	46	81
R^3	RDM3	32	95	41	38	22	91	11	35	23	92	18	57
	RDM5	49	74	64	13	33	61	21	9	36	66	29	17
	RDM7	61	51	85	5	46	46	29	3	46	49	34	8
Tyre		21	100	33	100	21	100	31	100	21	100	31	100

Table 4: Solution Length (len) averaged over the number of Successes (suc) within the set resource limit

the average solution length is summarised in Table 4. Performance is remarkably similar for the three compiled planning configurations (so much so that the plots for the different configurations are superimposed on each other) and the average solution lengths are identical at 21 operators per problem. Somewhat surprisingly, given Russells results in his encoding of this domain [49], the uncompiled configuration *BS* solves 100% of the problems in an average of 2 seconds per problem. The relatively good performance for the uncompiled planning configurations can be explained by the fact that the total-order planning algorithm FMD can produce sub-optimal plans whereas Russell's planner only generated optimal solution plans. Optimal length plans for the problem sample *RDM8* contain 19 operators: the average length of solution plans generated by the compiled planning configurations was 21 operators, whereas the length of plans generated by the uncompiled planning configurations was on average 50 per cent longer (between 31 and 33 operators).

Barrett and Weld describe this domain as being 'fairly difficult', and quote a figure of 6 hours for solution time for early experiments in the domain [3, pp 99]. The best combination of their own algorithms takes 123 seconds to reach an optimal solution. Our results show that the problems were trivial for the compiled configurations, and that there was virtually no variation in solution time for each problem (at 1 second of CPU), supporting our contention that the goal ordering rules produced very useful orderings during planning. On the other hand, one should point out that to achieve these results we have to (a) create a full domain model with invariants, substate class expressions etc (b) compile the model (c) sacrifice the *guarantee* of optimality when using a planner such as FMD (although we found that the the introduction of further types of ordering devices produced an optimal solution for this problem - see section 5.5.2).

5.5 Evaluation Summary

5.5.1 Evidence from Dynamic Testing

The results indicate that the **compiled** configurations are relatively superior, producing generally shorter solutions using much less cpu time and much less space compared to the uncompiled configurations. It was impossible to distill an overall factor of improvement, as many problems solved by the compiled configuration were never solved using the uncompiled configurations. Also, the results for the Tyre World are very impressive compared to other results given in the literature.

The compiled configurations also rate well in a more fundamental sense. Firstly, the average number of nodes expanded during successful plan generation by the compiled planning configurations was consistently low, and was less than the resulting average length of solutions in the majority of tests (in contrast to theoretical results which point to the number of nodes rising exponentially with respect to the required length of a solution plan). Also, the distinctive effect of different search strategies over all the domain models was dampened by the effect of compilation, giving a consistent feel to the results. This suggests that the impact of the compiled domain model is to some extent independent of the particular search method in the planning algorithm.

5.5.2 Effectiveness of FMD + Goal Orders

In general goal orders lead FMD to produce shorter solutions and to shorten plan generation times. This conclusion is supported by the Tyre World tests as their use of macros in this world had a negligible effect. Although PRECEDE's generation of type (i) goal orders allowed FMD to obtain a sensible ordering of the Tyre World goals, the total order planner produces on average a sub-optimal solution of 21 rather than 19 operators. The cause was found to be that while working on top level goal $wheel\_in(wheel_1, boot_1)$, there was a lack of an ordering constraint between two sub-goals $p = unfastened(Hub)$ and $q = jacked\_up(Hub, Jack)$. Investigation of p and q here shows that we have a type (ii) block: it is impossible to establish p only, by a valid sequence of operators from a state which contains $\{\sim p, \sim q\}$ since all establishers for p have q amongst their preconditions. Type (ii) rules force the planner to consider $unfastened(Hub)$ only, and in their combination with type (i) would lead to an optimal order for this problem.

5.5.3 Effectiveness of FMD + Macros

We will use the factors introduced in section 4.1 to judge the effectiveness of a macro set, in the light of dynamic testing.

1. *the likelihood of some macro being usable at any step in solving any given problem:* Given that the macro table spans the whole set of task configurations, this likelihood would be a certainty for *TSA* domains. As the test models were not *TSA* FMD reverted to operator backchaining when a unique substate could not be found for the object described by a chosen goal predicate. Across all our results the use of macros, as a

percentage of all nodes processed, ranged between 40 % and 85 %. As predicted, the poorest result were obtained from R^3 with a macro usage averaging 40 %.

2. *the amount of processing (search) a macro cuts down:* The method of engineering domain models into sorts provides a means of decomposing problems into useful subproblems where this 'factors' out some of the search. This is supported by observations above on the small amounts of nodes required to obtain solutions in the results. In models where substate transitions are trivial because of the lack of static structure, such as in the Tyre World, the use of macros in the form we have described makes no significant difference (here goal ordering is the dominant factor in planning speed-up).
3. *cost of searching for an applicable macro during planning:* As the index used to search for a macro is a ground pair of substates, the cost of searching a macro table is not subject to exponential matching problems.
4. *cost, in terms of solution non-optimality, of using a macro:* Our experiments suggest that this cost is low as argued above. We believe that this is linked to the combined use of goal ordering techniques that order goal sets so that the establishment of one goal cannot undo another. As Korf states [37, pp 45] (himself paraphrasing Banerji [2]) 'macros that are useful .. leave all previously satisfied subgoals intact while satisfying an additional subgoal".
5. *the cost of generating and maintaining the macro set:* The average cost of generating a macro is generally a fraction of the average cost of solving a problem in the full domain model as macro generation abstracts out all dynamic sorts except the one in question. The parameters that determined a macro table's size were detailed above and in Table 2.

5.5.4 Summary of the Benefits of the Object-Centred Approach

The object-centred approach that we postulate here appears to form the glue that integrates our method and tools to help solve the problems associated with (a) the management of the complexity of the planning process, and (b) the validation and verification of the planning domain models. Both these problems are attacked by our engineering approach which focuses on enriching the model with knowledge *in a systematic way* that leads to a (compiled) representation in which the solutions to problems is much easier to find. Yet at the same time this enrichment creates more opportunities for checking our understanding of the domain and the accuracy of the resultant model.

6 Related Work

Up to now there has been little research into utilising an object-centred representation to manage the process of plan generation. A recent exception is the PLANRIK algorithm [20]. This is similar to plan generation at the object level in our framework as object states are used as the basis for plan generation. Linear plans are formed for each object in the domain by ignoring all other objects, and then *merging* them with a current global plan to produce a partially-ordered plan (this is an interesting alternative to the use of a total order planner although the merging operation may well introduce a processing overhead). PLANRIK differs from our framework in that the plans to move objects, and the precedence between object states, are all decided on-line during planning, rather than during a compilation stage; and the level of planning is at the *object* rather than the object class (sort) level.

An interesting comparison can be made with the work of Jonsson and Bäckström [30], 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 by studying 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 is proposition-centred rather than object-centred.

There is an increasing awareness of the need for structured approaches and associated tools to support the modelling of planning domains. For example, the KADS methodology in reference [1] was used to produce a model independent of planning control knowledge. KADS is a proven methodology for knowledge base system development, and its use contrasts sharply with our planning-oriented, custom-built method. The output of the work cited is quite different from our own, however, in that the model produced was a 'KADS inference structure' for use in hierarchical skeletal plan refinement. The importance of understanding the impact of modelling domains in particular ways was demonstrated by Collins and Pryor in [15]. They showed the potential problems of incompleteness that can result from including *filter conditions* in operator preconditions. In deductive planning, Biundo and Stephan recognise the need for modelling planning domains systematically [52]. They use temporal logic as a formal framework, and as well as taking a formal view of proving the consistency of models using invariants, they further engage other software engineering concepts such as abstract data types and model reuse.

An integral feature of our method is the emphasis on the use of tool support at all stages of domain model capture and validation which is is in part inspired by our work in formal specification and requirements capture [56, 41]. The need for tool support for domain engineering is increasingly being reflected in the literature, for example Des Jardins work with SOCAP emphasised the need for tools to help capture domain models [18] and Chien showed the importance of tools to validate the planning domain model for the Multimission VICAR planner [13]. As Chien states, some of the errors that such tools reveal might appear straightforward to catch but nevertheless can be 'painful to manually track down' [13, pp 26]. In a more recent publication [14] he takes the work further, introducing static analysis tools to analyse the achievability or otherwise of planning goals.

Another key aspect of our approach is the use of compilation tools to transform (or operationalise) a domain model into a more 'efficient' form. This reflects trends in other areas of Artificial Intelligence in general. For example, a common feature of knowledge and rule bases is that an initial model is captured in a declarative form that is best suited to the domain and the modeller(s) and then this is compiled into a procedural form that is more amenable to automated reasoning (see for example work on COLAB in [5]).

Work on macro generation was pioneered by Korf [37] who used macro tables to solve problems in simple domains such as the Rubic's Cube, and this was discussed in section 4. The macros generated by our compilation tool improve planning efficiency since the search needed to construct the 'plan fragments' (i.e macros) is not required at planning time. This is similar to a perceived advantage of hierarchical task reduction planners: task reduction schemas (non primitive operators) contain plan fragments, supplied by the domain modeller, that remove search during planning (this HTN framework, in the tradition of NOAH [51], has recently been formalised and compared to partial order planners by [22] and [4] for example). We argue that our macros serve the same function: they have the same 'hierarchical' flavour as task reduction schemas since they are sort abstracted; and unlike other examples of macro use they do not heavily increase the branching factor during planning because macro selection and instantiation is linked to the current state as well as the goal conditions.

A related work to our macro generation technique is STATIC [24], a method for acquiring control rules for operator and binding selection. STATIC can be viewed as a compilation tool, although an important difference is that our technique reasons about object-centred domain theories whereas STATIC analyses static interactions for each domain literal. In addition, the output of our method is sets of sort abstracted macros generated through a process of sort abstracted planning whereas STATIC analyses 'goal-stack cycles' to generate control rules. Static analyses of operators is also carried out in the work of Poet and Smith [19]. They show how operator graphs, which are similar to the transition graphs we used in section 2, can be generated to analyse threats in partial order planning.

There are a number of systems that appear in the literature concerned with the generation of goal orders for planning. The identification of interactions that forms part of our goal ordering technique PRECEDE has been used elsewhere. For instance, Dawson and Siklóssy's REFLECT system [17] compiled pairs of predicates ('incompatible assertions'), that cannot be simultaneously true in a consistent state of the planning domain and Drummond and Currie [21] used state invariants that were similar in their work on 'temporal coherence'. Both sets of researchers used these invariants (similar to our negative invariants) 'on the fly' to speed up planner performance, by selecting partial plans with temporally coherent outstanding preconditions. PRECEDE uses a similar state invariant to compile goal orders but a key difference with this and other work is that invariants are used to identify goal interactions and form goal orders during a compilation stage in a problem-independent manner.

7 Future Work

7.1 Extensions to Sorts

More research and development is needed to make our current framework more expressive, and less dependent on STRIPS-type assumptions. Also, it would be useful to integrate more ideas from the object oriented software design areas. For example, one might usefully introduce *aggregation* to our model - that is defining an object of one sort as an aggregation of other sorts, for example:

$Box = box(Box\_id, Shape, Colour)$

Further, we could introduce recursive sort definitions, for example:

$Stack = above(Box, Stack) \cup Box$.

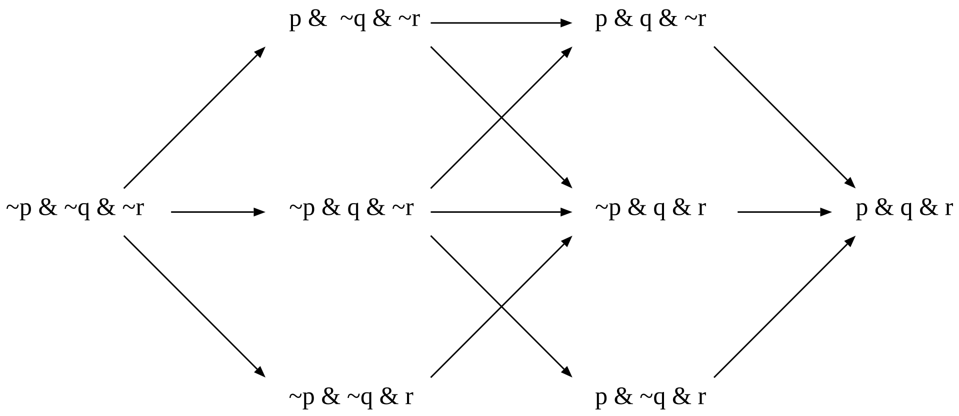

Figure 21: Goal establishment Diagram for 3 literals

Two object instances of sort Stack might be:

$box(box1, small, green)$
$above(box(box1, small, green), above(box(box2, medium, green), box(box3, large, red)))$

For such recursive sorts, substate class expressions would include an enumeration of the *forms* of the structure (in the same way that an equational specification of an abstract data type might define each type operation in terms of how it re-writes the different constructor forms of the type [56]).

7.2 Improvements to Goal Order Generation and Use

The goal ordering techniques that were presented in the paper generated goal orderings based on an analysis of necessary interactions between pairs of lifted predicates in a domain model. We see development progressing in three ways. Firstly, further ordering procedures resulting from our analysis need to be implemented and tested (the experimental results were based solely on type (i) necessary orders, and possible orders). Secondly, we could extend the technique so it tests for interactions between pairs of *substates* for some sort in a domain model. Thirdly, we could extend the analysis to consider larger groups of predicates for goal ordering. As an illustration, consider the goal establishment diagram shown in figure 21. This shows all the valid states that assert the 3 ground literals p, q and r. The relationship between this diagram and the diagram for 2 literals (figure 8) is that a single 'block' in the 2-diagram (for example between $p\&\sim q$ and $p\&q$) is equivalent to finding the following blocks in the 3-diagram:

$$[p\&\sim q\&\sim r, p\&q\&\sim r]$$
$$[p\&\sim q\&r, p\&q\&r]$$

So finding a block in the 3-diagram will lead to an increased number of 'finer grained' rules being identified, but where these rules will be less widely applicable.

7.3 Improvements to Macro Generation and Use

The regularity of the object-centred approach is crucial to the effectiveness of the procedure for creation and use of a Macro Table. We have not exploited the idea fully, however. Currently our planner makes use of macros in a limited manner, in the case where the chosen goal predicate defines a unique substate in the context of the current goal. We have yet to experiment with full use of Macro Tables in the disjunctive substate situation, or in applications involving TSA models. More development is also required on the form of a macro table. The size and flexibility of tables would be optimised by producing macros for supersorts (rather than for primitive sorts only), and by filling tables with partially-ordered rather than linear macros. t

7.4 Incomplete Information

A promising area for exploration is the use of sorts and substates to cleanly capture non-deterministic operators, and domains in which the exact substate of an object is unknown. To do this we remove the constraint that a well-formed state is a kind of vector of substates; instead a state is generalised to be a vector of **sets of substates**. Incomplete information about an object means that it is represented as a set of substates - that is the range of possible situations in the domain model that it could be in. Likewise, the constraints on operators can be loosened so that their effect causes an object to be in a set of possible states. This can be done, for example, by allowing the right hand sides of the substate change rules in operators to match on more than one substate class. When an operator is executed it will therefore leave the object in a set of substates. The attraction in using object-centred representation for this extension would be that the range of states (and hence uncertainty) is bounded by the range of possible substates specified for each object.

7.5 More Powerful Sort Abstracted Planners

To date the main platform for our empirical evaluation of sort engineering and compilation tools has been conventional 'literal-based' planners. Our future work will involve the construction of a fully sort engineered planner that reasons solely at the level of the *object* rather than at the level of the *literal* which can be used as a platform for future experimentation. This will be in contrast to our current platform which, although accepting a sort engineered domain theory, performs some reasoning at the level of the literal. It is anticipated that this sort engineered planner will be more efficient than our current system and that the continuity afforded by extending the representation level throughout the system will help further evaluate the merits of our approach.

Part of the development of this new planner will involve experimentation using sort engineered compiled domain models with other plan generation algorithms such as HTN, a hierarchical planner in the tradition of NOAH [51], which has recently been formalised and compared to partial order planners by [22] and [4]. This appears a promising area of investigation since the macros that are generated by our compilation tools can be seen as storing abstract planning solutions, and as such they could provide solutions for a hierarchical planner at levels organised by sort abstraction. Alternatively, the use of goal orderings that are output by our compilation tools could provide an advanced initial partial plan for use with a partial order planner with temporal constraints on the initial goals. Any other temporal point in an advanced partial plan that required the establishment of a conjunction of goals could likewise be augmented with temporal constraints if appropriate.

8 Conclusions

In this paper we have argued for a change in the emphasis in classical planning research to take into account knowledge-based aspects of domains. This can be done cleanly, retaining a measure of domain-independence, by creating an object-centred model. We have detailed a rigorous method, using a non-trivial example, which helps in the acquisition, validation and refinement of such a model. The method was supported by a set of definitions and properties which formalised the model, and a description of the tool support required for each step of the model's lifecycle. In particular we detailed two compilation tools that capitalised on the knowledge-based form of the model, and showed how their use in compiling the domain model improves plan generation performance, when used in conjunction with a total-order planner. Using compiled knowledge to cut down search in this way is evidently as powerful and yet as generally applicable as recent techniques for planning algorithm improvement, and to our knowledge, more effective than using other machine learning techniques for planning speed-up.

Finally, we see our work as the first step towards (a) providing a bridge between realistic, application-oriented planning and clean, theoretical planning research and (b) providing a set of standards for planning domain encodings so that models can be exchanged easily between research groups, and properties of these models can be universally understood.

9 Acknowledgements

Our thanks go to Diane Kitchin, Blaga Iordanova, Steve Scott and Malcolm Roome for reading over earlier drafts of this document, and to Colin Pink and Iain Anderson for help with production of the results graphs.

References

[1] J. S. Aitken and N. Shadbolt. Knowledge Level Planning. In C. Bäckström and E. Sandewall, editor, *Current Trends in AI Planning.* IOS Press, 1994.

[2] R. Banerji. GPS and the Psychology of the Rubik Cubist. In A. Elithorn and R. Banerji, editor, *Artificial and Human Intelligence.* North-Holland, Amsterdam, 1983.

[3] A. Barrett and D. S. Weld. Partial-Order planning: evaluating possible efficiency gains. *Artificial Intelligence*, 67:71–112, 1994.

[4] A. Barrett and D. S. Weld. Task-Decomposition via Plan Parsing. In *Twelfth National Conference on Artificial Intelligence*, 1994.

[5] H. Boley, P. Hanschke, K. Hinkelmann, and M. Meyer. COLAB: A Hybrid Knowledge Representation and Compilation Laboratory. Technical Report RR-93-08, The German Research Centre for Artificial Intelligence, 1993.

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

[7] T. Bylander. Complexity Results for Extended Planning. In *Artificial Intelligence Planning Systems: Proceedings of the First International Conference*, 1992.

[8] G. Carbonnel, C. A. Knoblock, and S. Minton. PRODIGY: An integrated architecture for planning and Learning. In K. Van Lehn, editor, *Architectures for Intelligence.* Morgan Kaufman, 1991.

[9] A. Cesta and A. Oddi. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 341–352. IOS Press, 1996.

[10] 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 Conferen ce on Computer Integrated Manufacturing*, 1991.

[11] D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, 32, 1987.

[12] J. Cheng and K. B. Irani. Ordering Problem Subgoals. In *Proc. IJCAI*, 1989.

[13] S.A. Chien. Towards an Intelligent Planning Knowledge Base Development Environment. In *Planning and Learning: On to Real Applications. Papers from the 1994 AAAI Fall Symposium*, number FS-94-01. AAAI Press, 1995.

[14] S.A. Chien. Static and Completion Analysis for Planning Knowledge Base Development and Verification. In *Proc. AIPS.* AAAI Press, 1996.

[15] G. Collins and L. Pryor. On the Misuse of Filter Conditions: A Critical Analysis. In C. Báckstróm and E. Sandewall, editors, *Current Trends in AI Planning*, pages 105 – 116. IOS Press, 1993.

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

[17] C. Dawson and L. Siklóssy. The Role of Preprocessing in Problem Solving Systems. In *Proc. IJCAI*, 1977.

[18] M. desJardins. Knowledge Development Methods for Planning Systems. 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.

[19] D.E.Smith and M.A.Poet. Postponing Threats in Partial Order Planning. In *Eleventh National Conference on Artificial Intelligence*, 1993.

[20] E. Diaz-Infante and C. Zozaya-Gorostiza. PLANRIK: A Hierarchical Nonlinear Planner based on Object States. In *Proceedings of FLAIRS The Florida AI Research Symposium*, 1996.

[21] M. Drummond and K. Currie. Goal Ordering in Partially Ordered Plans. In *Proc. IJCAI*, 1989.

[22] K. Erol, J. Hendler, and D. S. Nau. UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning. In *Proc. AIPS*. Morgan Kaufman, 1994.

[23] K. Erol, D. S. Nau, and V. S. Subrahmanian. On the complexity of domain-independent planning. In *Tenth National Conference on Artificial Intelligence*, 1992.

[24] O. Etzioni. Why PRODIGY/EBL Works. In *Proc. AAAI*, 1990.

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

[26] Y. Gil, M. Veloso, S.A. 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.

[27] M. Ginsberg. Universal Planning: An (Almost) Universally Bad Idea. *AI Magazine*, 3(4), 1989.

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

[29] J. Hertzberg. On Building a Planning Tool Box. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 3–18. IOS Press, 1996.

[30] P. Jonsson and C. Bäckström. Incremental Planning. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 79–90. IOS Press, 1996.

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

[32] S. Kambhampati and B. Srivastava. Universal Classical Planner: An Algorithm for Unifying State-Space and Plan-Space. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 61–75. IOS Press, 1996.

[33] Z. Kazi. Integrating Human-Computer Interaction with Reactive Planning for a Telerobotic System. Ph.D. Proposal,Department of Computer and Information Sciences, University of Delaware, DE, USA, 1994.

[34] D. E. Kitchin. Use of an object-centred approach for the creation and validation of a warehouse planning domain model. Technical Report in Preparation, The University of Huddersfield, School of Computing and Mathematics, 1997.

[35] C. A. Knoblock. Learning Abstraction Hierarchies for Problem Solving. In *Proc. AAAI*, 1990.

[36] C. A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

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

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

[39] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Ninth National Conference on Artificial Intelligence*. AAAI Press, 1991.

[40] T. L. McCluskey and J. M. Porteous. Two Complementary Techniques in Knowledge Compilation for Planning. In *Proceedings of the 3rd International Workshop on Knowledge Compilation and Speedup Learning*, 1993.

[41] T. L. McCluskey, J. M. Porteous, Y. Naik, C. N. Taylor, and S. Jones. A Requirements Capture Method and its use in an Air Traffic Control Application. *Software–Practice and Experience*, 25:47–41, January 1995.

[42] 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*, 76:1–16, 1995.

[43] S. Minton. Quantitative Results concerning the utility of Explanation-Based Learning. In *Seventh National Conference on Artificial Intelligence*, pages 564 – 569, 1988.

[44] S. Minton, J. Bresina, and M. Drummond. Commitment Strategies in Planning: A Comparative Analysis. In *Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[45] E. P. D. Pednault. Generalising Nonlinear Planning to Handle Complex Goals and Actions with Context-Dependent Effects. In *Twelfth International Joint Conference on Artificial Intelligence*, 1991.

[46] M. A. Perez and O. Ezioni. DYNAMIC: A new role for training problems in EBL. Technical Report CMU-CS-92-124, School of Computer Science, Carnegie Mellon University, 1992.

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

[48] The PRODIGY Research 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.

[49] S. Russell. Efficient memory-bounded search algorithms. In *Proc. ECAI*, 1992.

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

[51] E. Sacerdoti. The Nonlinear Nature of Plans. In *Proc. IJCAI*, 1975.

[52] S.Biundo and W.Stephan. Modeling Planning Domains Systematically. In *Proceedings of the 12th European Conference on Artificial Intelligence*, 1996.

[53] L. T. Semmens, R.B.France, and T.W.G. Docker. Integrating Structured Analysis and Formal Specification Techniques. *The Computer Journal*, 36:600 – 610, 1992.

[54] P. Stone, M. Veloso, and J. Blythe. The Need for Different Domain Independent Heuristics. In *Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems*. Morgan Kaufman, 1994.

[55] T. M. Mitchell and R. Keller and Kedar-Cabelli. Explanation–Based Learning: a unifying view. *Machine Learning*, 1, 1986.

[56] 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.

[57] M. Veloso and J. Blythe. Linkability: Examining Causal Link Commitments in Partial Order Planning. In *Proc. AIPS.* Morgan Kaufman, 1994.

[58] D. H. D. Warren. WARPLAN: A System for Generating Plans. Technical Report Department of Computational Logic memo 76, University of Edinburgh, 1977.

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