Model-Based Planning for Object-Rearrangement Problems*

Max Garagnani and **Yucheng Ding**
Department of Computing
The Open University
Walton Hall, Milton Keynes - MK7 6AA
{M.Garagnani,Y.Ding}@open.ac.uk

Abstract

We describe a model-based planning representation, aimed at capturing more efficiently the basic *topological* and *structural* properties of a domain. We specify the syntax of a domain-modelling language based on the proposed representation. We report the experimental results obtained with a prototype system (called PMP, Pattern-Matching Planner) able to represent and solve planning problems expressed in this language. The performances of PMP on a set of five domains are compared with those of a second planner, adopting the *same* search algorithm but using a classical STRIPS propositional language. Preliminary results show a superior performance of PMP on *all* of the chosen domains.

Introduction

During the past few years, the planning community has put a significant effort into developing systems able to exploit *domain-specific* knowledge to carry out a more 'informed' search (e.g., knowledge about the generic type and structure of a domain (Fox & Long 2001; 2002), control knowledge and structure of desirable solutions (Bacchus & Kabanza 2000)(Nau *et al.* 1999), heuristics (Hoffmann & Nebel 2001)(Haslum & Geffner 2000), problem constraints and domain invariants (Kautz & Selman 1999)(Gerevini & Schubert 1998)). In spite of the leap in the scale and complexity of the problems solved that this effort has produced, current applications are still limited to narrow, well-defined domains, and do not exhibit the flexibility and adaptability that characterise human planners (Wilkins 1997)(Wilkins & desJardins 2001). To a large extent, the cause of this limitation is the fact that, in addition to *domain-specific* knowledge, planning in the real world requires using *common-sense* knowledge and reasoning (including reasoning by analogy, abstraction, learning, and dealing with uncertainty and incomplete knowledge), a type of inference which has proven particularly hard to automate in all areas of AI.

This work is guided by the hypothesis that one of the main factors preventing modern planning systems from carrying out fast and effective common-sense reasoning (and, hence, from scaling well to realistic domains) lies in their adoption of *inefficient problem representations*. In particular, most current planners rely on '*propositional*' domain descriptions languages (e.g., STRIPS, ADL, PDDL and descendants). Such formalisms are not always appropriate for modelling real-world problems, particularly when these require a substantial amount of common-sense reasoning about *spatial* and *topological* relations between objects. Indeed, even the most recent versions of PDDL (Fox & Long 2003) require the basic physical properties and *constraints* of the world (e.g., the fact that an object cannot be simultaneously in two different places) to be declared and/or dealt with *explicitly*. The adequate encoding and exploitation of such constraints turns out to be crucial for achieving good performances in large and realistically complex problems (e.g., see (Kautz & Selman 1998)).

An alternative to adopting propositional (or 'sentential') formalisms consists of using *model-based* (or 'analogical') domain descriptions. In a model-based representation, the world state is encoded as a data structure which is *isomorphic* to (i.e., a model of) the *semantics* of the problem domain. For example, in their seminal work, Halpern and Vardi proposed the adoption of a Kripke structure to model the 'possible-worlds' knowledge of a group of interacting agents (Halpern & Vardi 1991). Because of their isomorphism with the world state, a key feature of model-based representations is their ability to *implicitly* embody constraints that other representations must make explicit, and, hence, to improve the efficiency of the reasoning process (Myers & Konolige 1992). On the other hand, model-based formalisms tend to be less expressive and more limited in scope than propositional languages.

In this paper, we propose a model-based planning representation, able to capture implicitly, more efficiently and naturally the basic, common-sense *structural* and *topological* constraints (expressing spatial and 'containment' relationships, respectively) of a domain. Although model-based, the representation is sufficiently expressive to allow the encoding of a significant set of domains, in which the planning performances are notably improved.

The rest of the paper is organised as follows: first of all, we delimit the class of domains included within the scope of this investigation and describe the general features of the new representation. Secondly, we specify the syntax of a description language, which allows encoding domains using a simplified version of the general representation proposed.

*This work was partially supported by the UK Engineering and Physical Sciences Research Council, grant no. GR/R53432/01

Thirdly, we discuss preliminary results obtained with a prototype planner on a set of five domains, and conclude by pointing out advantages, limitations and possible extensions of the proposed approach.

A Model-Based Representation

The planning representation that we describe here has been developed to allow the efficient and natural encoding of object-rearrangement (or, simply, *move*) domains. These can be defined as problems that require planning the changes of *position* (location) of a finite set of objects on the basis of their spatial and topological relations, subject to a set of constraints. The Tower of Hanoi (ToH) represents a prototypical example of this class, in which the positions of a set of objects (disks) have to be changed according to a set of rules (constraining the movement of the disks). Other examples of this class are the Briefcase domain, Gripper, Blocksworld (BW), Grid, Logistics and Eight-puzzle. Notice, however, that although not explicitly of a 'move' nature, some domains are isomorphic to (and can be treated as) object-relocation problems. For example, if *activities* are represented as objects, and locations denote *time points* or *intervals*, then the problem of scheduling a number of tasks over a given time period can be seen as that of re-assigning to each 'object' (activity) an appropriate 'location' (start/end time point), subject to various constraints. More in general, any *state* change of an object can be modelled as a change of *position*, given an appropriate reformulation of the domain.

The basic entities of our representation are '*nodes*', '*places*' and '*edges*'. 'Nodes' represent instances of the types of (mobile) objects present in the domain (e.g., physical objects, agents, resources, etc.). 'Places' denote different locations of the domain, and can be thought of as *qualitatively* distinct areas of space containing sets of objects. A place can contain nodes, other '*sub*-places', or both. 'Edges' are pairs of places and nodes, and express spatial and topological relationships between them. An edge may 'connect' two places, two nodes, or a place and a node. Nodes, places and edges can be associated to unique labels.

The sub-places of a place are places themselves, and can be used to define the *internal structure* of a place. A place may be defined so that it is subject to specific restrictions, limiting, for example, the *type* and *number* of nodes that it can contain. The sub-places of a place may also be connected by edges. A place containing no connected sub-places will be called '*unstructured*'.[1] Nodes, places and edges are defined using three separate type hierarchies, in which the properties of a type are inherited by all of its instances and sub-types.

Figure 1.(a) shows an example of node and place hierarchies for the well-known Briefcase domain. The types "OBJECT" and "PLACE" lie at the roots of the two hierarchies. The place hierarchy specifies that a "Location" place will be allowed to contain any number of nodes of type "OBJECT" (i.e., instances of "Portable" or "Mobile"). A place of type "Briefcase" can only contain "Portable" nodes.

[1] In general, one can see nodes as places required to be always empty, or places as nodes which contain other nodes.

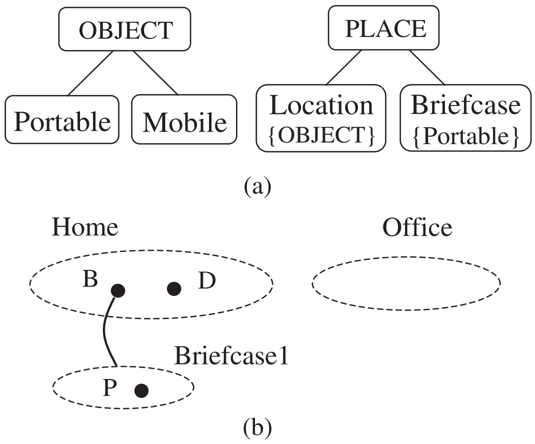

Figure 1: Briefcase world: (a) type hierarchies, and (b) initial state for the "Get-paid" problem

Figure 1.(b) contains a graphical representation of a possible encoding of a state in terms of places, nodes and edges. Nodes are represented as (labelled) filled circles, places are denoted by dashed ellipses, and edges are depicted as bold arcs (in this example, the edge hierarchy can be assumed to contain only the root class "EDGE"). The state represented in Figure 1.(b) corresponds to the set of propositions {`at(Home,B), at(Home,D), in(P), at(Home,P)`}, the initial state for the "Get-paid" problem. Nodes 'P' and 'D' are "Portable" objects (the types of the various instances are not shown in the figure), whereas 'B' (the briefcase) is an instance of "Mobile". Notice also the presence of different types of places: 'Home' and 'Office' are instances of "Location", while 'Briefcase1' is a place of type "Briefcase". The state contains a single edge, connecting node 'B' with place 'Briefcase1' and encoding the *association* between the briefcase node and its contents.

As a second example, Figure 2 models the Blocksworld domain. Part (a) of the figure describes node and place hierarchies, according to which a "Cell" place is allowed to contain up to one ('[1]') node of type "OBJECT", while a "Stack" place can contain any number of "Cells" as sub-places. Part (b) shows a graphical representation of a three-block problem. This example demonstrates the use of *structured* places and edges as place-to-place connectors. In particular, each of the three "Stacks" S_1-S_3 contains four connected "Cells", some of which contain a node. Although in this example they are not connected, these three places could, in turn, be linked by (possibly labelled) edges. The nodes labelled 'A', 'B' and 'C' are instances of the type "Block", while the three nodes labelled 'T' are of type "Table". Notice that such 'T' nodes refer to distinct objects of the current state which do not need to be *discriminated* at this level of the representation, and that have been assigned the same label. We will refer to this kind of objects as *generic instances* of a type, in that they cannot be distinguished from each other, but can still be discerned from other entities (even of the same type) having a unique name. In this example, none of the internal sub-places or edges have been associated to a unique label.

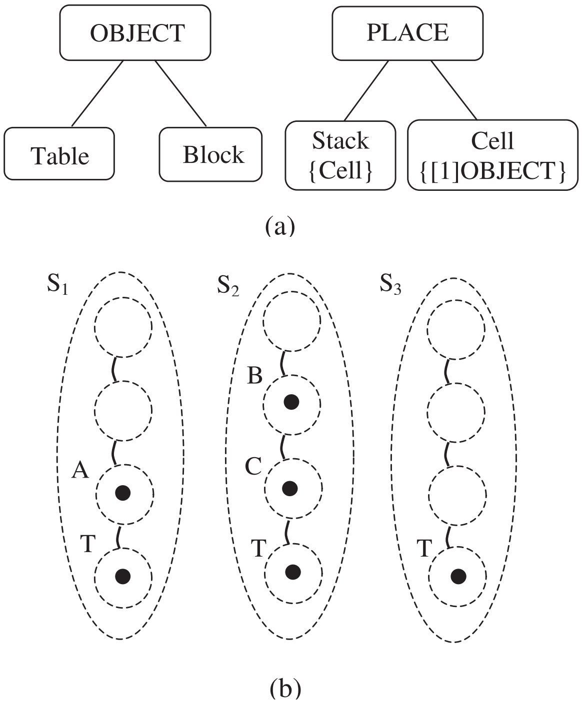

Figure 2: Blocksworld: (a) type hierarchies; (b) graphical representation of the state {`On(A,Table), On(B,C), On(C,Table), Clear(A), Clear(B)`}

Representing Change

Having described the basic elements of the state representation, let us move on to the definition of the formalism for specifying the set of possible *action schemata*, necessary for identifying the legal transformations of the state. In this representation, the world state is a collection of places, nodes and edges. In general, any of these elements will be allowed to be moved from their current position, or even to be *added* to or *removed* from the state. However, nodes are often the only *mobile* objects of the domain, while places and connecting edges cannot be affected during plan execution and can be regarded as forming an underlying *stationary* structure. For example, in Blocksworld, the internal cells of the three stacks can be regarded as 'fixed' places, while the blocks and the 'table' nodes can be moved from one place to another (subject to appropriate constraints).

In this analysis, we assumed that the possible transformations of the state are limited to the *movement* of nodes and places, while the edges connecting such entities remains unchanged.[2] When a node (or a place) moves, all edges connecting it to other entities remain 'attached' to it, and all contents of a place move with it. In addition, the movement of places and nodes is restricted by the general constraints of the domain (e.g., number and types of nodes allowed in a place) specified by the type hierarchies.

Consider, for example, the action schema '*Put-in*' for the Briefcase domain, illustrated graphically in Figure 3.(a). The left-hand side (*preconditions*) of the schema specifies the situation holding in the two relevant ('loosely' connected) places before the execution of the action. (Notice that the absence of nodes in one of the places should not be interpreted here as requiring such place to be empty – this is clarified below). The right-hand side (*effects*) depicts the same set of places and nodes after the action has produced a new node arrangement.[3]

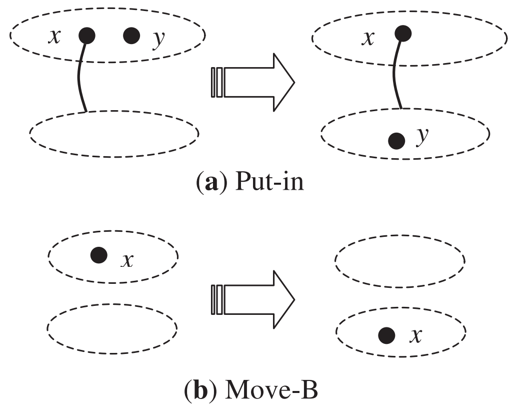

Figure 3: Briefcase domain: action schemata

In general, the preconditions of an action may contain a *conjunction* of 'groups' of connected places and nodes. An action schema can be applied in a state S when *each and every one of the elements* – places, nodes and edges – *present in its preconditions* 'matches' *(can be bound to) a distinct*[4] *element of* S, so that each precondition group is bound to a (distinct) group in S having the same *topological structure* ('pattern' of places and edges) and node arrangement. The definition of '*match*' is as follows: two types match *iff* one is sub-type of the other; two instances match *iff* they are the same instance (i.e., they have the same name); an instance x and a type T match *iff* x is an instance of T, or x is an instance of any of T's sub-types. A precondition group consisting of a place containing n nodes matches any place (of the appropriate type) containing *at least* n nodes (of the appropriate type). For two edges to match, they must connect matching entities. In summary, for an action to be applicable, each of its precondition groups must 'overlap' with a distinct part of the current state. In the previous example, the preconditions of the *Put-in* operator contain only one group, which is easily mapped to corresponding elements of the state shown in Figure 1.(b).

Figure 3.(b) contains the graphical representation of the more interesting action schema '*Move-B*', which allows the movement of the briefcase from a location to another. The preconditions of this action contain *two* groups, which must be bound to distinct places of the state. For the action schema to be applied correctly, node '`x`' must be bound to

[2]This implies that the initial number of entities – nodes, places and edges – remains *constant* throughout plan execution.

[3]The *Take-out* action schema can be obtained simply by reading the *Put-in* operator 'backwards'.

[4]Notice that *generic* instances of a type still represent distinct elements of the state.

an instance of "Mobile", and the two places to (distinct) instances of "Location". If these constraints on the type of the nodes and places were not enforced, these elements could be bound to incorrect instances and lead to illegal moves (such as moving a "Portable" node like 'D' directly across locations, or a "Mobile" (briefcase) node inside a "Briefcase" place). Notice that the *Put-in* operator (and its mirror-image *Take-out*) will also be subject to appropriate type requirements, although in this case the topological structure of the preconditions and effects is sufficient to guarantee that the application of the schema (in both directions) to a semantically correct state will always produce a correct state, even when multiple briefcases are present.

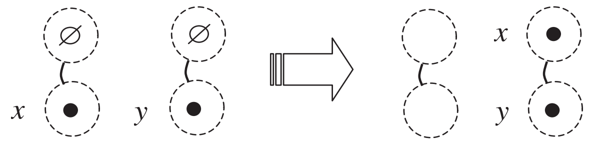

Figure 4: Blocksworld: '*Put-On*' action schema

Finally, consider the BW *Put-on* action schema, represented graphically in Figure 4. Let us assume the types of the nodes 'x' and 'y' to be, respectively, "Block" and "OBJECT", and all places to be of type "Cell". Notice the use of empty-set symbols in the preconditions, requiring the two top cells to be empty. With reference to the state depicted in Figure 2.(b), the first group of the preconditions would match the cell containing 'A' (and the empty cell immediately above), or, alternatively, the cell containing 'B' (and the one just above). Similarly, the second group would match the same cells, and, in addition, the two cells at the bottom of 'S_3' ('y' is of type "OBJECT"). Hence, this schema could be applied in the current state to move 'A' onto 'B' or vice versa, or either of the two blocks on the table. Notice that for this action schema to be used in a 'backward' search, it would be necessary to require an extra empty cell on top of the second group, in order to guarantee that node 'x' (being removed from the top of 'y') is 'clear'.

In general, the nodes, places and edges present in the initial state of a problem are not necessarily *preserved* throughout the plan execution: new objects can be dynamically 'produced', existing objects can change their *state* or be 'consumed', and relations between places might change as a result of some of the actions. For example, in a 'house construction' domain in which places represent the locations that the workers can currently reach, the *addition* of a new floor or of a new scaffolding will change the set of places and the connections between them. Although such 'non-conservative' state transformations have not been considered here, the model could be easily extended to include a set of *primitive* actions which allow the addition and removal of nodes, places or edges (subject to appropriate preconditions), enabling the representation of this type of dynamics.

A Prototype Language

In order to assess the effectiveness of the representation, a prototype planner has been implemented. The domain-description language developed for the planner consists of a restricted, 'diagrammatic' version of the formalism described in the previous section. In particular, in the implementation, the *internal* structure of places is confined to be a one- or two-dimensional *grid* of sub-places. Hence, places can either be unstructured, or have their internal structure equivalent to that of a matrix (or vector), in which cells are considered as a collection of *adjacent* locations and are allowed to contain up to *one* object (node).[5] Since nodes have been represented simply as labels, a place is a (structured or unstructured) collection of strings.

A further simplification of the language implemented consists of not allowing the use of connecting *edges*, which encode topological and spatial relationships. This restriction is compensated in part by the possibility of places to have a predefined, fixed internal structure (which can be augmented with a set of dedicated spatial relationship – this is discussed below in more details), and in part by the possible use of *identical names* across hierarchies, which allows, for example, a node and a place to be assigned the same label (i.e., to be somewhat connected). As mentioned before, dynamic changes of the underlying topological structure of the domain have not been allowed.

In spite of the limited expressiveness, this language still contains most of the fundamental characteristics of the general model, and allows a preliminary assessment of the validity of its basic assumptions. The language developed is sufficiently flexible to be able to model naturally and efficiently a small subset of the benchmark domains (including BW, Briefcase, Miconic, Gripper and Eight-puzzle), which we used to carry out a set of experiments. The results of such experiments are reported in the final part of this section. In what follows, we illustrate briefly the syntax of the language and its semantics, exemplifying the description with declarations taken from the BW domain.

Type Hierarchies

The type hierarchies are declared using a syntax similar to that of PDDL. To describe the syntax, we adopt the extended BNF (EBNF) formalism, used in the original PDDL definition (McDermott *et al.* 1998):

<types-def> ::= (:ObjectTypes <typed-list (name)>)
<types-def> ::= (:PlaceTypes <place-type>$^+$)
<typed-list (x)> ::= x^* | x^+- <type> <typed-list (x)>
<place-type> ::= <name> {<type>[::<dimension>]}
<type> ::= <name> | (either <type>$^+$)
<dimension> ::= 1 | 2

The pipe character ('|') indicates disjunction (e.g., <dimension> can be either '1' or '2'). The category <name> can be any string of characters, not necessarily beginning with a letter. The 'typed list' is a parameterised production that generates a (possibly empty) list of object-type names and 'IS-A' relations, using the minus sign ("$-$") as in PDDL to indicate a 'parent' type. '`object`' and '`place`' are both predefined types, and '`object`' is used as default

[5]Notice that, for simplicity, the notion of 'adjacency' in two-dimensions is restricted to *vertical* and *horizontal* pairs of cells.

terminator of any typed list. The optional parameter in the place type declaration (surrounded by square brackets) allows the specification of the internal structure of a type of place (array of one or two dimensions)[6]. In absence of such parameter, the place type is assumed by default to be unstructured. As in PDDL, the 'either' construct allows an object type to be defined as the union of several types. Below is an example of type declaration for the BW domain:

```
(define (domain blocksworld)
   ...
   (:ObjectTypes block table)
   (:PlaceTypes stack {object::1})
   ...
)
```

Types '`block`' and '`table`' are declared (by default) as sub-types of '`object`'. Places of type '`stack`' will be *one*-dimensional arrays ("::1") of '`object`'s. Notice that this syntax does not allow the specification of an upper limit on the number of elements contained in an unstructured place, or on the number of 'cells' composing a structured place. That is, a '`stack`' place (vector) could, at this point, contain any number of cells, each one containing up to one '`object`'. The upper limit on the number of elements (or cells) contained in a place will be determined at 'run time', by the specific problem instance.

Action Schemata

The following EBNF productions specify the syntax for the declaration of an action schema:

```
<action-def>  ::= (:action <name>
                     :parameters (<typed-list (variable)>)
                     <body-def>)
<body-def>    ::= :pre (<place>*)
                  :post (<place>*)
<place>       ::= <name> { [<relation>] <object>+}
<object>      ::= <variable> | <emptySpace>
<variable>    ::= <name>
<relation>    ::= ⋆ | / | ↔ | ↕
<emptySpace>::= _
```

The action declaration consists of a unique name, a list of parameters, and lists of 'pre'- and 'post'-conditions (effects). The parameters are names of variables, coupled with the respective types. The pre- and post-conditions consist of lists of place types, each containing as argument a list of parameters and, possibly, 'empty spaces'. The empty-space symbol ('_') is used in the preconditions to require the presence of empty cells in structured places, or to require the availability of 'space' for an object in non-structured ones.

For an action to be applicable, all the elements present in the preconditions must be bound to appropriate elements of the current state, as explained earlier on. Notice that *all* the places listed in the preconditions must be in a *1-1* mapping with those listed in the postconditions, following the *order* in which they are listed. In addition, the number of objects present in each place must remain constant[7]. Below is an example of '*Put-on*' action schema in Blocksworld:

```
(:action Put-on
    :parameters (x y - block)
    :pre  (stack {x _ } stack {y _ })
    :post (stack {_ _ } stack {y x })
)
```

This declaration can be easily mapped to the graphical representation of Figure 4. By default, the elements listed inside a *structured* place are interpreted as being required to be *adjacent* (i.e., to occupy adjacent cells). Thus, blocks '`x`' and '`y`' are guaranteed to be 'clear' by the presence of an empty cell adjacent to them.[8] The two '`stack`' places listed in the preconditions are mapped to the two places listed in the effects, in the order specified. The content of each cell of a *structured* place listed in the preconditions (identified by a parameter or an empty space) is *replaced* with the object occupying the *same position* in the postconditions, following the *order* in which the objects are listed. For example, the cell containing the object that gets bound to parameter '`x`' will end up containing an empty-space ('_'), and the (currently empty) cell adjacent to '`y`' will contain '`x`'. Elements specified in a non-structured place cannot be required to belong in any specific spatial relationship.

In order to require more complex spatial relationship to hold between elements contained in a structured place, the language has been endowed with a set of (optional) <relation> symbols, representing a limited version of the more general feature of labelled *edge*. These symbols can be used inside structured places to require that a certain spatial relationship, different from that of adjacency, hold between the specified elements. For example, '⋆' indicates that the elements that follow can be (with respect to each other) "anywhere within the place"; '↔' means "anywhere on the same row", '↕' means "anywhere in the same column", and '/' requires that the two following elements be located in adjacent cells of the same column (the last two relationships are only needed for two-dimensional arrays). The chosen set of relationships is by no means complete, and can be easily extended with other, more complex, ones. An example of usage of these relationships is demonstrated by the Miconic (elevator) domain description, reported in Appendix A.

Initial State and Goal

We use an actual initial state and goal declarations for a specific BW problem (the 'Sussman anomaly') to illustrate informally the syntax adopted for describing initial state and goal. The correct formalisation can be easily inferred from this example and the EBNF rules used earlier for the types and actions declarations:

[6] Notice that unlike square brackets, which are meta-symbols, *curly* brackets are terminal symbols of the language being defined.

[7] An 'empty space' is treated as a special type of object.

[8] This assumes that all the '`block`' objects will be arranged in the '`stack`'s so as to have always at most *one* empty cell adjacent to them, which represents the space on 'top' of them – e.g., see Figure 2.(b).

```
(define (problem Sussman)
   (:domain blocksworld)
   ...
   (:Objects A B C - block T - table)
   (:Places s1 s2 s3 - stack)
   (:init
          s1 [T A C _ ]
          s2 [T B _ _ ]
          s3 [T _ _ _ ])
   (:goal
          stack {C B A})
)
```

Notice the similarity between the declaration of the initial state and the graphical representation shown in Figure 2.(b). In the declaration of the initial state, the contents of a *structured* place are delimited by square brackets, indicating the actual 'start' and 'end' of the array. Hence, the declaration implicitly specifies the *maximum* number of required cells, and the *exact position* of all the objects and empty spaces ('_') within them. For unstructured places, the contents will be delimited by *curly* brackets (as for normal sets), and the declaration will specify the contents and the maximum number of objects that a place is able to contain.

A *goal* is syntactically equivalent to a precondition list. It consists of a 'conjunction' of places required to contain specific sets of objects (nodes), possibly subject to specific spatial relationships. The conditions for achieving a goal are analogous to those required for the application of an action schema: a goal g is *achieved* in a state S when all groups specified in g can be bound to distinct groups of S, such that, for each place of g, all the elements contained in it match distinct elements in the corresponding place of S (which satisfy the same spatial relationships, if appropriate).

In goals – like in action preconditions – the objects listed inside a *structured* place are required to occupy consecutive cells of the place, in the same order specified. However, the 'pattern' of objects specified can be placed *anywhere* within the place. More formally, if L is a type of structured place, and $x_1, x_2, \ldots, x_n$ are objects, the goal $L\{x_1 x_2 \ldots x_n\}$ requires an instance of a place of type L to contain the listed objects in *any* n consecutive cells, such that (x_1, x_2), (x_2, x_3), (x_3, x_4), ..., (x_{n-1}, x_n) are all pairs of adjacent elements. Thus, for example, for a state S to achieve the goal of the 'Sussman' problem presented earlier, it must be the case that S contains a place of type 'stack' in which ('C','B'), and ('B','A') are pairs of adjacent elements. Two examples of stacks of four cells that satisfy such requirements are '[T C B A]' and '[C B A _]'.

Experimental results

Ideally, the evaluation of the efficiency gain (or loss) resulting from the adoption of a new domain representation – let us call it 'α' – with respect to another (propositional) representation, 'β', could involve the following four steps: (1) measuring the performance of a state-of-the art planning system adopting β a on a select set of problems; (2) 'switching' the domain representation to α; (3) measuring the performance of the system on the same set of problems; (4) comparing the results. However, this method of assessment presents several drawbacks. The first one concerns the fairness of the evaluation in itself. After more than three decades of research based almost exclusively on propositional domain-modelling languages, planning algorithms have become more and more sophisticated and geared towards purely sentential descriptions. In fact, one could say that their efficiency results mainly from their ability to exploit some of the inherent properties of such representations. Taking a state-of-the-art planning system, designed specifically for propositional descriptions, and simply 'switching' its representation into a completely different one (assuming that this is possible) would not produce a system 'comparable' with the original one. Indeed, the new formalism might be incapable of replicating some of the particular techniques upon which the propositional planner might rely for achieving good performances. At the same time, new, different reasoning modes may become available in the new representation, which could lead to gains in performance. However, modifying the algorithm to take advantage specifically of such features would lead to the implementation of an entirely different system, not comparable with the original one.

The second methodological issue concerns the actual value and feasibility of the assessment. Even assuming that a specific system can be identified such that all of its sophisticated search mechanisms can be correctly 'translated' in the new representation, what would be the significance of an evaluation carried out on such a specific case? The results would not apply to all planning algorithms, not even to those adopting the same propositional language. In addition, replicating correctly all of the various features of the planner in the new representation would require a considerable effort, and would still leave a margin of uncertainty on the soundness of the outcome.

The above considerations suggest that a fair evaluation should be based on a very simple, 'primitive' planning algorithm, in which the few mechanisms at the basis of the search can be encoded in a straightforward manner in both representations. One possible candidate that immediately springs to mind is the original STRIPS planner. However, although old and inefficient, STRIPS is quite a complex system; obtaining, understanding and modifying the original software, written more than thirty years ago, did not seem a very practical approach.

Therefore, in order to assess the value of the new representation, we decided to build two simple prototype planners, adopting exactly the *same search engine* (a traditional, uninformed, breadth-first forward state-space search), but differing in the way they represent domains and problems. The first, propositional planner (which we called 'PP') adopts a classical (typed) STRIPS representation. The second planner (which we called 'PMP', Pattern-Matching Planner) adopts the 'diagrammatic', simplified version of the general model-based representation described in the previous section. The two planners were developed using the same programming environment and language (Java), and were run on the same machine. We tested the planners on the same problems for the five chosen domains (BW, Gripper, Miconic, Eight-puzzle and Briefcase). In carrying out these

Table 1: Time taken (in sec.) by PP and PMP for solving Blocksworld problems with four, five and six blocks.

	BW-n-problem instance (n = no. of blocks)			
Planner	BW-4-0	BW-4-1	BW-5-0	BW-6-0
PP	0.52	1.50	7.43	144.42
PMP	0.08	0.44	0.34	1.60
PMP(b)	0.19	0.54	2.13	36.23

experiments we hoped to demonstrate that: (1) PMP actually produced correct solutions; (2) its performances were at least 'comparable' with those of PP. The very first results that we obtained were encouraging: in the four BW problem instances used, PMP was not only producing correct solutions, but also performing much better than PP, with a speed-up factor varying from more than three to as much as *ninety* times faster (see the first two rows of Table 1).

We reasoned that such difference in performance had to do with the fact that the specific representation of BW in PMP is implicitly *constrained*. For example, in PP the space available on the table is not limited; hence, at any point of the search, any block on top of another can be put on the table. By contrast, in PMP the problem representation is limited to *three* stacks.[9] Hence, many of the moves which are possible in the standard propositional representation are no longer available in PMP, which can avoid exploring them. This significantly prunes the number of possible paths in the search space. In order to evaluate the performance of PMP without this 'inherent' advantage, we added to the description of the problems a number of redundant empty stacks, so that the total number of stacks equalled the total number of blocks. This 'evened up' the number of possible paths in the two representations, as the space available on the table in PMP was now sufficient to allow the same number of moves as in the propositional representation. We repeated the same experiments, adding five extra problems to the set of tests (all taken from the benchmark problems used in the AIPS'00 International Planning Competition). The new results for PMP on the original four experiments are reported in the last row of Table 1 (labelled 'PMP*(b)*'), while the results for the complete set of instances are plotted in Figure 5.

As expected, the presence of the redundant stacks caused a loss in performance, evident from the comparisons of the PMP and PMP*(b)* data. However, PMP still took significantly less than PP to find the same (shortest) plan in *all* of the problems. Figure 5 shows that the increasing difficulty of the problems produces (exponentially) bigger differences in performance in favour of PMP. One possible explanation for this speed-up may be that the problem representation in PMP is still more constrained than the propositional one. In other words, many of the search paths that are considered in PP may be redundant or non-feasible, and are completely ignored in PMP. For example, the ground action-schemata in PP may contain duplicate or unadmissable instances (e.g., move a block on top of itself) that are *implicitly* avoided by PMP's object-based representation.

[9] The problems had been chosen so that three stacks were sufficient for finding the same optimal solution found by PP.

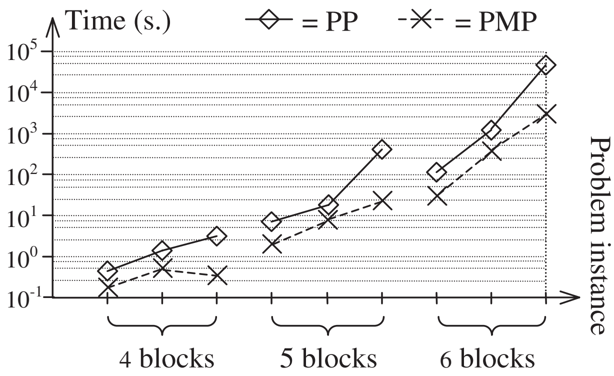

Figure 5: Blocksworld: results for PP and PMP planners.

In order to further investigate the effectiveness of the representation, we carried out experiments on four other domains – namely, Gripper, Miconic, Briefcase and Eight-puzzle. The results are plotted in Figures 6, 7, 8 and 9, respectively. In Figures 6 and 8 the problems have been ordered by increasing PP-time; notice that the *sum* of the two numbers identifying each problem increases monotonically.

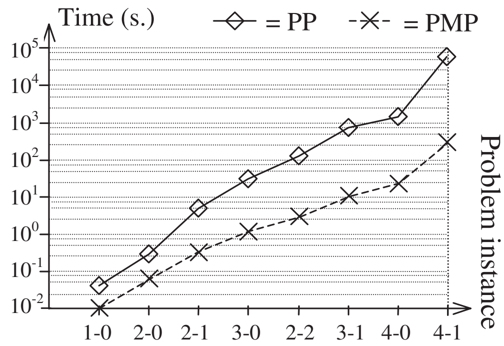

Figure 6: Gripper domain. Problem instance 'm-n' consists of two rooms, containing respectively m and n balls.

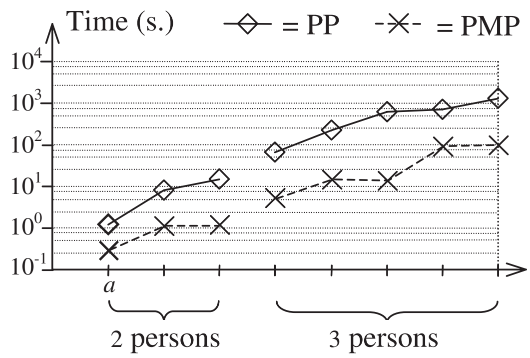

Figure 7: Results for the Miconic domain. All problems contain four floors (except for problem *(a)*, which has three).

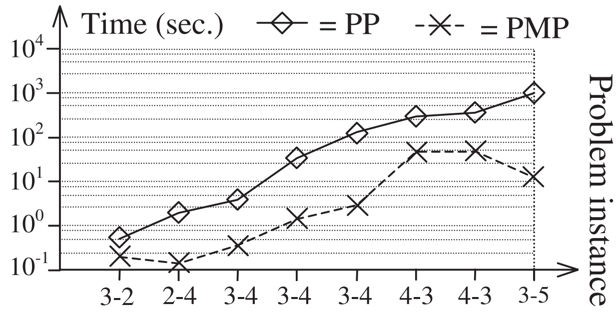

Figure 8: Briefcase domain. Problem instance 'm-n' contains one briefcase, m locations and n portable objects.

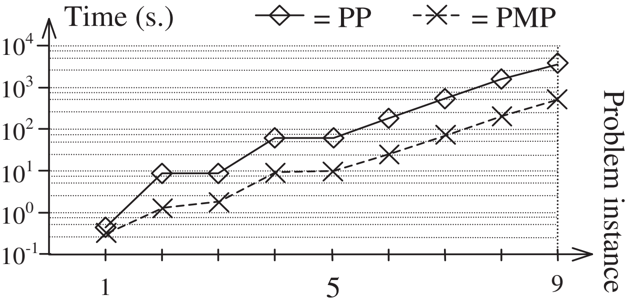

Figure 9: Eight-puzzle domain.

Being confined to a small set of domains not *randomly* chosen from the entire population, these results cannot be assumed to form a representative sample from which general conclusions can be drawn. However, albeit preliminary, they demonstrate the effectiveness of the proposed approach, and give a clear indication of the performance improvements that the new representation can yield (with respect to a standard propositional representation) in the context of *move* problems.

Related work

One of the few recent works adopting a model-based view to planning is that of Giunchiglia and Traverso (Giunchiglia & Traverso 1999), who treat planning as a model-checking problem. Their approach is more abstract and general than the one proposed here: the representation is not based on a model structurally isomorphic to the world, but on a graph (FSM) representing the possible states in which the world can be and the possible transitions between them. Although more general, this representation fails to *implicitly* capture the spatial and topological constraints of the domain.

In the work of Cesta and Oddi (Cesta & Oddi 1996), planning is seen as the problem of deciding the behavior of a dynamic domain described as a set of state variables subject to constraints. This view can be seen as complementary to the approach taken in this work, as it is based on the idea of representing a domain as a set of possible state *changes*, instead of possible object *moves*. However, once again, the representation is not capable to *implicitly* embody the topological and structural constraints of the problem.

The object-centred domain-description language (OCL) developed by Liu and McCluskey (Liu & McCluskey 2000) allows modelling a domain as a set of *objects* subject to various constraints. The idea of an object-based representation is, in many respects, analogous to that of a model-based approach. However, OCL does not allow the specification of the spatial features of the domain; hence, some of the constraints which are implicit in the proposed model still have to be explicitly declared in OCL (e.g., the fact that if an agent holds an object, the location of the object must coincide with that of the agent).

The work of Long and Fox (Long & Fox 2000) on the abstract structure of domains and generic types is also relevant in this context. Long and Fox have developed domain analysis techniques which allow the automatic detection and exploitation of generic types of objects (such as mobiles and portables) in a domain, given its propositional description. In essence, the *move* problems considered here can be seen as generalizing and combining two of the abstract classes identified by Long and Fox (namely, those of 'transportation' and 'construction' domains).

An earlier example of work integrating model-based and propositional representations is the hybrid problem-solving system of Myers and Konolige (Myers & Konolige 1992). Myers and Konolige proposed a formal framework for combining analogical and deductive reasoning. The system they implemented could reason about 'diagrammatic' structures isomorphic to the current world state. However, the system did not allow the model to undergo any *change* during the reasoning process, and, hence, could not be used to solve *planning* problems. In addition, the efficiency of the representation was not evaluated against that of a propositional language through a comparative study, as it has been done here. The latter approach and other related ones fall within the area of 'diagrammatic reasoning' (Glasgow, Narayanan, & Chandrasekaran 1995), and are particularly relevant to the simplified, diagrammatic version of the representation which was used in the experiments.

Discussion

In this paper we have introduced and assessed a new planning representation, able to capture more efficiently the basic *topological* (containment relationships between places and objects, or objects and objects) and *structural* (spatial relationships between places, or internal structure of a place) properties of a domain. In addition, we have described the syntax of a specific domain-description language, representing a simplified version of the general model proposed. The performance of a prototype system (PMP) on a subset of five domains has been compared with that of an equivalent propositional planner (PP), adopting an *identical* planning algorithm but using a STRIPS domain-description language. The results show that PMP is superior to PP on *all* of the chosen instances of problems. We believe that the speed-up of the PMP planner is the result of the ability of the new representation to (1) capture naturally and *implicitly* some of the basic, common-sense physical constraints of the domain, and to (2) *organize* in appropriate structures the relevant entities of the domain, allowing more efficient reasoning about the object-rearrangement aspects of the problem.

The adoption of a simpler, model-based, spatially-structured representation that allows effective commonsense reasoning also enables new reasoning modes, which can support easier and more efficient *learning*, *heuristic-extraction* and *abstraction* techniques. For example, it is easy to see that the graphical description of the action schemata shown in Figures 3 and 4 could be *inferred* automatically using machine-learning and image-processing techniques. In addition, the use of edges as node-to-place connections can easily support abstraction: a group of objects contained in the same place (or attached to the same object) can be represented as a *single* entity, allowing the system to abstract away from the details of the parts and enabling abstraction to take place at different levels.

The use of a model which replicates the spatial and topological aspects of the real world also provides the planner with an implicit guidance on the ordering of subgoals. Consider, for example, the 'Sussman anomaly' problem in BW, presented earlier. A propositional description of the problem does not provide *a priori* information on the order in which the two subgoals $\{$ `(On A B),(On B C)` $\}$ must be achieved. By contrast, in the proposed representation, the goal-pattern $G =$ `stack`$\{$`C B A`$\}$ allows regressing a new goal G' in which 'A' has been picked up from the top of 'B' (by reversing step *Put-on*(A,B)), but does *not* permit the regression of a situation in which 'B' has been removed from the top of 'C' (via step *Put-on*(B,C)). In fact, consider the action schema *Put-on*(x,y): its effects include leaving block x 'clear'. While goal G makes no requirements on the content of the cell to the immediate right (read 'top') of 'A' (which may thus be assumed empty), the cell to the right of 'B' is currently occupied by 'A'. This prevents x from being bound to 'B'. Hence, a backchaining planner would be (correctly) forced to begin with addressing subgoal `(On A B)` – i.e., to execute *Put-on*(A,B) as final step.

Finally, a domain-modelling language (such as the one described earlier) based on the proposed representation has a straightforward graphical interpretation, and, hence, should result more intuitive and easier to use for the non-experts; this, together with better performances, is expected to facilitate the take-up of AI planning technology and its wider application to the real world.

The proposed representation, however, is also limited in several ways. Perhaps its most obvious weakness is the fact that it does not offer any means for describing the non spatial or topological aspects of a domain (such as time or metric quantities), which, on the other hand, could be modelled using a propositional language. However, the approach described here should be seen as representing the extreme of a *continuum* of possible domain-modelling languages, in which propositional and analogical aspects of a domain can be present in different degrees. One of the future directions of this work consists of extending the representation to allow the use of propositional expressions. A more expressive version, for example, could allow nodes to consist of complex objects, having an internal (static or dynamic) structure and properties. In such an extended model, the action preconditions could require the *attributes* of the specified objects (such as size, color, shape, etc.) to satisfy specific (qualitative or quantitative) constraints.

In conclusion, the results of this investigation demonstrate the potential benefits of the proposed representation, and motivate further work on model-based planning formalisms and on their use in conjunction with current propositional paradigms. This paper lays the foundations for future developments in this direction.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Cesta, A., and Oddi, A. 1996. DDL.1: A formal description of a constraint representation language for physical domains. In Ghallab, M., and Milani, A., eds., *New Directions in AI Planning*. IOS Press (Amsterdam). 341–352. (Proceedings of the 3rd European Workshop on Planning (EWSP95), Assisi, Italy, September 1995).

Fox, M., and Long, D. P. 2001. STAN4: A Hybrid Planning Strategy Based on Sub-problem Abstraction. *AI Magazine* 22(4).

Fox, M., and Long, D. 2002. Extending the exploitation of symmetry analysis in planning. In *Proceedings of the 5th International Conference on AI Planning and Scheduling*. Toulouse, France: AAAI press.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research – Special issue on the 3rd International Planning Competition*. (forthcoming).

Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of the 15th National Conference on Artificial Intelligence*, 905–912. Madison, WI: AAAI Press.

Giunchiglia, F., and Traverso, P. 1999. Planning as model checking. In Biundo, S., and Fox, M., eds., *Proc. of the 5th European Conference on Planning (ECP-99)*, 1–20.

Glasgow, J.; Narayanan, N.; and Chandrasekaran, B., eds. 1995. *Diagrammatic Reasoning*. Cambridge, MA: MIT Press.

Halpern, J. Y., and Vardi, M. Y. 1991. Model Checking vs. Theorem Proving: A Manifesto. In Allen, J. A.; Fikes, R.; and Sandewall, E., eds., *Principles of Knoweldge representation and Reasoning: Proceedings of the 2nd International Conference (KR91)*, 325–332.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proceedings of the 5th International Conference of AI Planning Systems (AIPS 2000)*, 140–149. Breckenridge, Colorado: AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Kautz, H., and Selman, B. 1998. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the 1998 International Conference on AI Planning Systems (AIPS-98)*, 181–189. Pittsburgh, PA: AAAI Press, Menlo Park, CA.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and Graph-based planning. In *Proceedings of the 16th International Joint Conference on AI (IJCAI-99)*. Stockholm, Sweden: Morgan Kaufmann.

Liu, D., and McCluskey, T. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield (UK).

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Proceedings of the 5th International Conference on AI Planning and Scheduling Systems (AIPS'00)*, 196–205. Breckenridge, CO: AAAI Press.

McDermott, D.; Knoblock, C.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Version 1.7. Technical report, Department of Computer Science, Yale University (CT). (available at www.cs.yale.edu/homes/dvm/).

Myers, K., and Konolige, K. 1992. Reasoning with analogical representations. In Nebel, B.; Rich, C.; and Swartout, W., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR92)*, 189–200. Morgan Kaufmann Publishers Inc., San Mateo, CA.

Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on AI (IJCAI-99)*, 968–973. Stockholm, Sweden: Morgan Kaufmann.

Wilkins, D., and desJardins, M. 2001. A call for knowledge-based planning. *AI Magazine* 22(1):99–115.

Wilkins, D. 1997. That's something I could not allow to happen: HAL and Planning. In Stork, D., ed., *HAL's Legacy: 2001's computers as dream and reality*. Cambridge, MA: MIT Press. chapter 14, 305–331.

Appendix A

A.1 – Miconic domain description

```
(define (domain miconic)
  (:ObjectTypes person lift)
  (:PlaceTypes building {object::2}
               lift {person})

  (:action board
     :parameters (P - person L - lift)
     :pre (building{↔ L P} lift{ _ })
     :post (building{ L _ } lift{ P })
  )

  (:action depart
     :parameters (P - person L - lift)
     :pre (building{↔ L _ } lift{ P })
     :post (building{ L P } lift{ _ })
  )

  (:action move-up
     :parameters (L - lift)
     :pre (building{/ L _ })
     :post (building{/ _ L})
  )

  (:action move-down
     :parameters (L - lift)
     :pre (building{/ _ L })
     :post (building{/ L _ })
  )
)
```

A.2 – Miconic problem instance

```
(define (problem mic-01)
   (:domain miconic)

   (:Objects A B C D E - person
             lf - lift)
   (:Places bd - building lf - lift)
   (:init bd [ [_  A B _ ]
               [lf D _ _ ]
               [_  _ C _ ] ]
          lf { _ _ }
   )
   (:goal bd [ [lf A _ _ ]
               [_  D _ C ]
               [_  B _ _ ] ]
   )
)
```