An Operator Induction Tool Supporting Knowledge Engineering in Planning

Nona Elizabeth Richardson

School of Computing and Engineering

The University of Huddersfield

Queensgate

Huddersfield

HD1 3DH

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

The University of Huddersfield supported by EPSRC

July 2008

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Within the field of artificial intelligence are many disciplines, one of which is planning. Planning seeks to find a suitable sequence of actions to carry out a task specified as a set of initial states for the objects involved in the actions and a required goal state. To do this the system has to have enough knowledge about the 'world' in the form of a planning domain model.

The process of constructing a planning domain model requires knowledge engineering. The structuring of the knowledge is important and hand-coding a domain model is a tedious and error-prone process. Static knowledge in the domain requires little update but the same cannot be said for the dynamic knowledge. The most difficult area of engineering planning domain models is the acquisition of operator schema, which contain descriptions of all the primitive actions (operators). In hierarchical models where actions may be modelled as sub-tasks, construction of these actions (methods) is particularly difficult and error-prone.

We argue for a system whereby dynamic knowledge can be generated for every planning eventuality. The major contribution of the thesis is a method for inducing primitive and hierarchical actions from several solution examples, without the benefit of intermediate state information. In our system, *opmaker*, actions are generated from relatively short action sequences, indicated by a user, who has simply to name an action and identify associated objects as being affected or unaffected by the action. To complete a planning task the system uses the static domain knowledge, the initial and goal states from the planning task, and the action sequences. Using these it first deduces possible state-change pathways which, using the right heuristics, may be unique (or at most have only a handful of pathways), and uses these to induce all the actions it needs. These actions can then be learned or regenerated at will.

We show that these induced actions compare to hand-crafted versions and can be used in planning. We can demonstrate that the hierarchical methods offer greater efficiency in planning times when compared to domains where previously no methods were offered.

The motivation for these ideas comes from the need to extend the development of GIPO, an integrated package for the construction of domain models in the form of a graphical user interface. We show how *opmaker*1 has already become a tool within GIPO for flat domains and argue for the inclusion of its successor, *opmaker*2.

Descriptions of domain construction and results using *opmaker* with several example domains are given. We analyse how, in general, this work contributes to and supports knowledge within the field, and the thesis concludes with suggestions for future work and discusses the particular contributions offered by *opmaker* to planning and knowledge engineering.

ACKNOWLEDGEMENTS

I would particularly like to thank my supervisor, Lee McCluskey, for his advice encouragement and 'pep' talks when the going got rough during my time as a PhD student. I am indebted to Lee for his constructive advice on the content of this thesis and for always being able to see the 'bigger picture'. To Margaret West I would like to extend my gratitude for all the patient hours helping me with Prolog and keeping me on track. I would also like to thank both my second supervisors, Margaret West and Diane Kitchin, for their advice on the readability of this thesis. My thanks also go to Stephen Cresswell who has been invaluable for the discussion of ideas, arguing about invariants and algorithms, and sharing the excitement of good results with cautious optimism.

I would like to express my thanks to my family for their unstinting support of this venture, for their encouragement and belief in me and, in the case of my husband Mike, for his sheer determination when I had lost mine!

Finally I would like to dedicate this thesis to my father, Frank Douthwaite, who sadly died six months before its completion, but who would have been proud to see his daughter succeed.

DECLARATION

I grant powers of discretion to the University Librarian to allow this thesis to be copied in whole or in part without further reference to me. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Chapter 1

INTRODUCTION

The trends generally in computing are towards imaginative and intelligent use of computers, and away from the idea that their use is mainly in banking and billing. Information storage and retrieval are part of these trends and the last couple of decades have seen an increase in the use of huge databases. Now databases can be imaginatively searched and data retrieval and storage is as much a part of daily life as the office typist and the filing cabinet were twenty years ago.

The definition of data for a large database has become a science in its own right, along with its structure and linking in the database. Special algebras have been developed to aid database design and information retrieval and these have greatly increased the capacity and complexity of databases. With their capacity to store vast amounts of structured data, databases have become essential to the science of artificial intelligence, and database search techniques have been developed to achieve quick and accurate results. Artificial intelligence has developed as a science because some computer problems do not lend themselves to being solved by a straightforward program which gives a rigid, structured, single result. They require the same degree of flexibility that the human mind has - the ability to react to changes in input, to make choices and to search and recall information. Problems like these can only be resolved effectively if the system has the ability to learn from mistakes and to improve and extend overall knowledge, to become more expert as that knowledge increases and to use improved knowledge more effectively and efficiently. One area of artificial intelligence, the subject of this thesis, is planning. Planning involves searching a

database of knowledge about potential actions in order to find a sequence of actions to complete a task.

1.1 Artificial Intelligence

One way of resolving such issues is the use of artificial intelligence, [definition 1.1].

Definition 1.1 *Artificial Intelligence (AI) may be defined as the branch of computer science that is concerned with the automation of intelligent behaviour [41].*

The field of artificial intelligence has expanded rapidly in recent years alongside the general increase in the use of computing in almost every aspect of life. The use of artificial intelligence (AI) allows a system to learn and to respond more flexibly to many different circumstances. However intelligent systems require an extensive knowledge base which has to be carefully constructed using knowledge engineering.

1.2 Knowledge Engineering

Our definition of knowledge engineering (KE) in the context of an object centred world in planning is given below.

Definition 1.2 *Knowledge engineering in planning is the process by which a conceptual theory about an object based world is represented by being translated into code. Objects and relations between them are identified and translated into the logic code which includes all possible states of the objects and any invariant facts. Sets of actions are defined and desirable tasks can be added.*

In general, where an object based system is not required, KE is the process of capturing and implementing the expertise of some specialist into an effective and

seemingly intelligent representation. Once the knowledge is captured it can be refined by use of the results from some practice examples until it reaches some desired level of performance [41].

KE for planning is a tedious business because defining the actions in terms of transitions of objects from one state to another has to be accurate. It is largely an iterative process by which domain knowledge is stipulated by a domain expert and this knowledge is represented in some modelling language by the knowledge engineer. At all times the requirements of the end user are paramount - in planning such a person would be the one who requires to do the actual planning.

The main substance of this research is aimed at improving techniques to capture, code and refine knowledge so that planning engines can act upon it effectively and efficiently and knowledge engineering no longer requires an expert.

1.3 Planning

Planning is one area of AI which is particularly dependent upon an accurate and up-to-date specialised knowledge base. Informally the idea in planning is a simple one in which a knowledge base (database) is searched in a particular way. (However there are other approaches such as logic, reasoning, negotiation etc. [69].) In planning the knowledge base contains some facts about a problem scenario. Amongst the facts are allowable actions in the scenario and knowledge about objects involved in those actions. In particular there will be data on the states of the objects. (As a simple example consider a book, a table and a reader. If the reader picks up the book the book changes state from being on the table to being held by the reader.) The idea in planning is to be able to use computing to *define* problems to solve and to *solve* them. A typical problem *definition* would consist of listing starting states for the objects and desirable goal states for them, whilst a *solution* would be an ordered

sequence of actions which would allow the goal state to be achieved. It becomes a challenging problem computationally because of the very large number of potential orderings of the actions. At its most basic, where there are n available actions, there are n! sequences of actions. A more formal definition of a planning problem is given in definition 1.15.

1.4 Planning Domain Modelling

Definition 1.3 A non-case-based planning domain model *is a knowledge base which contains a set of logical axioms within a formal system which, together with rules of inference and any required heuristic control rules, is composed to accurately define and model a problem scenario, and forms the basis of a logical deductive system. In planning, such a theory is known as a domain theory.*

When it is written according to the special syntax of some particular computer modelling language the knowledge base becomes known as a planning domain model [see definition 1.3]. It is desirable that such a knowledge base contains the descriptive information required to accurately record and reflect a particular 'world' and whilst it is largely a complete picture of that world, the system should also have the ability to add to the dynamic knowledge it contains by experience. In this work we will be describing the processes of acquiring knowledge bases with the object-centred language *OCL* [45], and of adding to dynamic planning domain knowledge by the induction of operators (actions).

Definition 1.4 Theory revision *is a method of refining a theory or knowledge base by the system automatically finding better changes to the theory to improve its inferential capability.*

Definition 1.5 An operator *in planning is a written description of some action in the domain modelling language in which dynamic objects feature either because their initial states prevail or because they undergo state transitions which the operator details as pre-transition states and post-transition states.*

1.5 The Knowledge Acquisition Bottleneck

In the context of planning and this research, machine learning is desirable in part to alleviate what Feigenbaum and McCorduck have referred to as the 'knowledge engineering bottleneck', a major obstacle to the more widespread use of many AI systems [18]. Any information system needs a lot of development time and AI systems with their large and detailed knowledge bases are costly to develop. In terms of time, operators (allowable actions) are particularly costly to develop. Time could be saved using machine learning to acquire accurate actions automatically and track the changing states of objects manipulated by them. The aim of this research was to take steps towards relieving this knowledge acquisition bottleneck by introducing a system which can induce [definition 1.6] and learn new operators [definition 1.5] using a process of theory revision [definition 1.4].

1.6 Some Further Definitions

This research has its roots in the artificial intelligence branch of computer science, in particular it concerns the knowledge engineering [definition 1.2] of planning domain models [definition 1.3] constructed to form the knowledge base of planning problems [definition 1.15] and to assist with their solution. We also draw on the related computer science topics of induction [definition 1.6], inductive logic programming [definition 1.8], machine learning [definition 1.7] and theory revision [definition 1.4].

Definition 1.6 Induction *is the process by which learning involves generalisation from experience [41]. A learner may have to learn from only a few examples but still acquire knowledge that will generalise correctly though not necessarily optimally.*

Definition 1.7 Machine learning (ML) *is the ability of a system to avoid some of the computation involved if some calculation has to be repeated. Instead the system draws on items added inductively to its knowledge base, by storage of previous results, use of examples or analogy, previous experience of a successful outcome, use of probability or because it has been instructed by an expert (human).*

Definition 1.8 Inductive logic programming (ILP) *is that branch of computer science concerned with programming to produce additional facts from a set of examples which may be positive or negative. These additional facts should be generalised by machine learning [definition 1.7], but may not be optimal.*

Definition 1.9 A dynamic object *is considered to be an object whose state can be changed by some action in the domain. Dynamic objects in OCL always have choices of state sets listed in the domain as substates.*

Definition 1.10 A static object *cannot be relocated, changed or given different values and so does not any states listed in the substates. In fact the best examples of static objects are locations themselves.*

Definition 1.11 A method *is a hierarchical operator in a more complex planning domain model and is therefore part of an hierarchical domain [see definition 1.12]. Besides stating the transitions for the objects it lists preconditions, states effects and stipulates an ordering of the methods and primitive operators it calls. Being itself part of an hierarchical structure it can be called by other methods higher in the structure and call on other methods and operators at a lower level. A method may call a different selection of operators to perform similar tasks depending on the objects and circumstances involved.*

Definition 1.12 A hierarchical domain *is one in which there may be an hierarchical sort structure, or an hierarchical method structure, or both.*

Definition 1.13 Scheduling, *at its best, finds a close to optimal schedule (or plan) given a set of actions, a set of resources and a set of constraints (often time related). It seeks to arrange actions according to when the resources are available and to satisfy all the constraints in the process. In some applications, such as university timetabling, just finding a feasible schedule is considered a good solution.*

Definition 1.14 *In a* closed world model *there are boundaries beyond which nothing is stated in the model. Often these are physical restrictions on the model as in Blocks World where the table top defines the extent of the world. Only objects and states specifically used in the model are described.*

1.7 The Scope of this Research

The aim of this research is to alleviate the knowledge acquisition bottleneck by using the part of the planning domain that is less time consuming to construct to induce planning operators and methods (compound multi-stage operators) [see also definition 1.11] which normally take a long time to construct by hand. Further details and the aim statement can be found in Section 1.9. The scope of the research is detailed in the sections below.

1.7.1 Induction

In any new domain the knowledge engineer will have some idea which actions might be required for the sort of problems to be solved. At the very least he might have a name for such an action and he might also have ideas about the objects involved in each action. We shall show that by taking a partially constructed domain with only these outline ideas in the form of a sequence of operator names and a list of the static and dynamic objects involved (the parameters), we can use the induction process to model operators and methods. This will be done by giving consideration to the initial states of objects in the domain and formulating goal states to be achieved in planning. We plan, initially, to retain the new operators as they are induced, using them to revise the original domain theory and learn a full set of generalised operators to complete a new planning domain model [definition 1.3]. In this respect the examples for induction come from the initial sequence and the initial and goal states for the objects named in the sequence.

1.7.2 Planning Problems

Within the context of this thesis planning problems are typified by being closed world problems. They contain sets of objects, mostly physical, to be manipulated. These physical objects do different things, and so can exist in different states. For example

a book can be on a shelf or held in a hand and to get from one state to the other a transition has to take place. Thus actions such as picking up a book are defined in terms of a transition from the book's being on the shelf to being held. Actions are initiated and represented descriptively by operators. Some objects modelled are not dynamic. A common thread in this thesis is the use of static objects to model the location of an action or to model a move of objects from one location to another. The locations are thus modelled as static because they never change unlike the objects associated with the move.

1.7.3 Types of Actions

Sequences of operators can be constructed in order to model complex activities but at all times the system must be able to keep track of the state change paths of the objects in the model. Actions can be modelled as either instantaneous transitions or, more recently, can be modelled as durative actions [21].

Durative Actions

The kinds of actions best suited to modelling with duration are those in which the time factor is a natural element of the action, those where there is concurrency, and those whose durations affect the start time of further actions which may depend upon the effects of previous actions to produce their start conditions. A simple example from the world of athletics would be the running of a relay race in which the second runner may not begin running until the first has almost completed his leg of the race.

Instantaneous Actions

In this research we shall be considering only actions assumed to be instantaneous. In these actions time is not a factor provided that actions are performed in the correct sequence.

1.7.4 A Typical Planning Problem

A definition of a typical planning problem is given below.

Definition 1.15 A planning problem. *Given a knowledge base containing certain objects and states and a set of possible actions and knowing the initial states of the objects, find a sequence of actions which will produce some pre-conceived goal state.*

Actions in a world being modelled are captured in its operators [definition 1.5].

1.7.5 Modelling Methods

Knowledge engineers who are domain experts find that specifying operator descriptions for a planning domain model, is a slow and painstaking process. (For detailed descriptions of complex domain construction the reader is referred to the RAX and VICAR system [17, 37].) When procedural knowledge is captured in a declarative way [47] it requires a language (such as *OCL* or Planning Domain Definition Language, *PDDL* [21]), selected because it is designed to be compatible with the construction of planning engines. The problem of accurate representation of planning domains is acute if non-planning experts are undertaking this task, or the operators are complex or hierarchical.

There are different methodologies for domain creation and in his journal paper Tate gives a good selection [73]. In our work the starting point is the creation of the representation of objects in the domain whereas using a different system like STRIPS [20] the starting point might be the actions in the world being modelled.

The Object Centred Approach

Our preferred method is the object centred approach in which actions are modelled in terms of how they affect some of the objects declared for the domain. Objects undergo

state transitions under the effect of actions which record the initial state of the object and its final state. In our earlier work we have developed a domain description language for the purpose. Using this language an expert engineering a planning domain model by hand would take as his normal starting point the specification of the objects in the domain. He would arrange these in a tree structure of different sorts.

Definition 1.16 Sort *is the name given to a particular type of object. For example apple and orange are particular instances of the sort fruit. Many sorts may be specified for a domain.*

Definition 1.17 Flat domain. *We refer to a single level sort tree where 'sorts' is at the root, all the domain sorts are at the first level and have object leaves, as a 'flat' domain structure because the sorts are all at one level.*

Definition 1.18 An hierarchical domain. *A more complex tree with several different levels of sorts would be referred to as an 'hierarchical' sort structure and the domain containing it would become an hierarchical domain. Whenever the sort structure is not flat there is the potential for predicate inheritance. This occurs when predicates apply to sorts at different levels of the tree and is explained in section 3.5.3.*

The Action Centred Approach

Domain modellers using an action centred approach begin by specifying allowable actions in the world they are trying to represent. Objects are important only in how they are affected by these actions.

1.7.6 Domain Building

In our object centred approach the objects are modelled first. The domain expert would next specify the relationships between the domain objects with a set of predicates (or statements) which can be arranged in groups to describe potential sets of

alternative states for the objects. Next the expert would describe any constraints and assumptions and any unchanging details in a set of invariants. A set of operators would then be constructed and finally any tasks to be performed could be added. Appendix A contains the coded version of a simple domain. The knowledge engineer has the task of capturing the knowledge from the domain expert and structuring it in the best way to be effective with the planner. As the domain is built and used it is refined using input from the domain expert until both are satisfied with the model produced.

1.7.7 Opmaker

The normal process for constructing operators is to hand-craft them with reference to the desired set of actions for the domain and using the state sets to determine pre and post-action states for all the objects. One recent extension of our work on knowledge engineering is that operator construction can be eased by the use of an implemented algorithm, *opmaker*. To use *opmaker* the domain engineer builds the planning domain model to the point where operator construction is the next step. He then conceives some task, the achievement of which requires a desirable example sequence of operations. By giving the *opmaker* system a set of initial states for the objects, the partially constructed domain, and the example sequence it can then induce a set of operators to achieve the pre-conceived goal. There are other mechanisms for learning operator sets and these will be covered in Chapter 5.

1.7.8 GIPO

To further address the bottleneck problem we have been developing an assisted method by which the domain expert specifies the declarative structure of the domain interactively. The domain engineered in this way shares the same structure as a hand-written one.

A domain building tool has been developed in a parallel research project where we have developed a Graphical User Interface (GUI) tool, GIPO (Graphical Interface for Planning with Objects) [25, 70, 44, 71], which is a tools package designed to make the construction of domains easier for the non-expert. *Opmaker* has recently been implemented within GIPO making it a powerful construction tool for the knowledge engineer and the non-expert alike.

Currently GIPO enables the non-domain expert to construct a domain by abstracting away much of the detail of the domain description language and allowing him or her to focus on the objects in the domain and the states in which they can exist. It does this by providing a GUI allowing a visual representation of the object types in a sort [definition 1.16] tree (see Figure 1.1) and other construction, validation and planning tools to construct the states and constraints. Operators can be con-

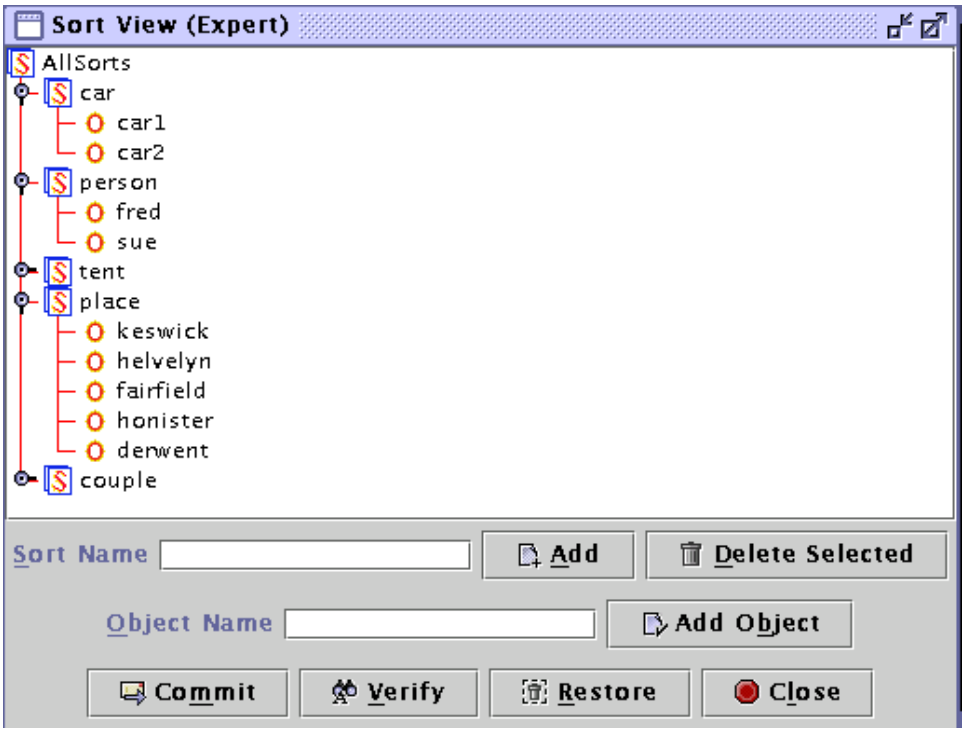


Figure 1.1: The Sort-Tree View Using GIPO

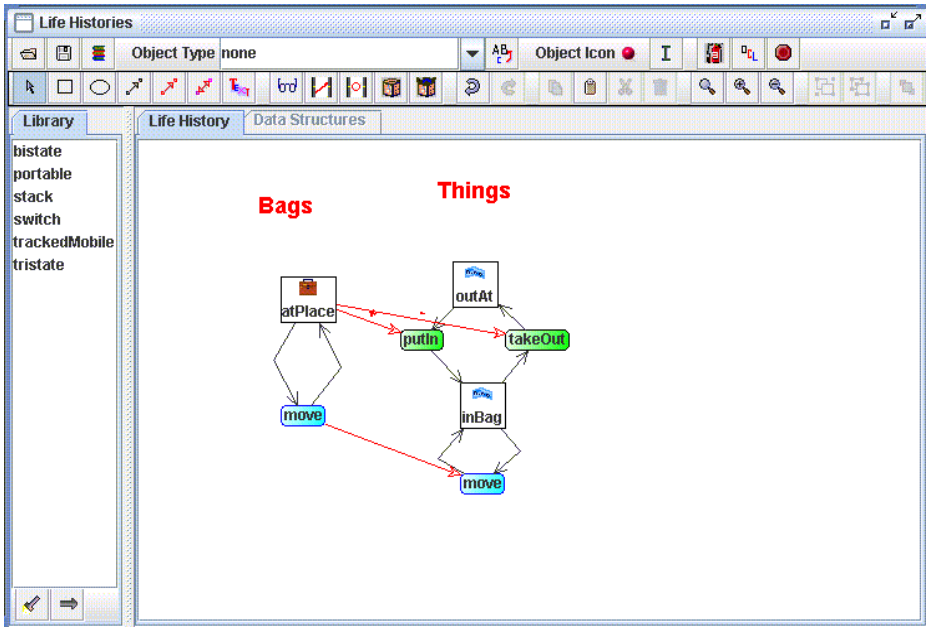


Figure 1.2: The Graphical Life-History Editor in GIPO

structed by hand using a graphical tool but using this method still requires a greater level of expertise from the user. Implementation of *opmaker* into GIPO has meant that it is now easier for the non-expert to completely describe a domain. We suggest that one area for further work should be an evaluation of the ease with which the non-expert can create operators for a domain using *opmaker* as opposed to creating the operators by hand and creating them using the operator construction tool also offered in the GIPO system.

In further developments of GIPO a graphical life history editor allows inexpert students of planning, after a little instruction, to build planning domains by constructing a diagram of objects, relationships and other properties, as shown in Figure 1.2.

1.7.9 Actions in a Planning Domain

Actions in a planning domain may be simple or compound. If we think of the simplest single action then we model this with an operator, to which we might refer as a 'primitive operator'. If we consider a group of single actions, put together in order to achieve a simple task, then we refer to that task as a 'method operator'. Since this method operator uses only primitive operators it is necessarily at a lower level in any method hierarchy than a method which uses other methods and primitive operators.

1.7.10 Hierarchies

Since domains may have method hierarchies and sort hierarchies there are different ways in which we can describe them as 'hierarchical'. We shall try to be consistent in this thesis to describe in which way(s) a domain is hierarchical.

1.7.11 Tasks in a Planning Domain

When we consider tasks we think of the compound actions we want to perform. If the domain has a hierarchical method structure then each method models a 'task'. These are effectively pre-determined tasks that are often required, so they become part of the available actions in the domain. Over and above that we are also able to define and declare tasks separately. In this way we have defined problems for the planner to solve, and they can be viewed as additions to the domains. The reader is referred to Appendix E where a number of tasks have been declared for that domain and placed at the end, after the operators and methods.

1.7.12 Induction of Methods

In an informal idea of human planning we would begin by taking an overview of the entire task to be achieved, then start splitting the task down into manageable chunks. Each of these chunks might split into further chunks and so on until finally the whole

task can be broken up into single actions. In planning we call the single actions 'operators' whilst the chunks are 'methods'. The branching tree of task, subtasks and single actions is a 'hierarchical task network' (HTN) and planning using such a network is called HTN planning. Within a single method there may be several single actions, or several methods or a mixture of both. Lowest level methods will just contain a series of single actions. The method contains information about the objects involved in it - in particular it considers the state sets for those objects. If the objects change state in one of the actions a 'transition' describes how the object states change. A method will contain several transitions and one obvious difference from an operator is that there may be several transitions *for one object*. Methods rely on the existence of operators and other methods and 'call' them. In the fact that these are called in a logical order to complete the task being modelled, methods resemble 'mini-plans'. Having methods in a planning system reduces the search space and so leads to time saved finding a suitable plan. A simple example of operators could be those actions needed to send a letter by post. These might be

1. write the letter
2. put the letter in the post box
3. stamp the envelope
4. put the letter in the envelope
5. take the letter to the post box.

The plan would be these items in the correct order e.g. 1, 4, 3, 5 and 2, whilst a method prepare letter for post would be the ordering 1, 4 and 3.

Development of *opmaker* has continued and it now induces operators and methods for hierarchical domains but at the time of writing this has not all been implemented in GIPO.

The logic of an example sequence may not include all the operators required for the particular domain in mind. For example if the world was to do with sending a letter you might put together a sequence that involved sticking on the stamp, walking to the post box and posting the letter. You may also have writing the letter and putting it in an envelope as possible actions which have not been included in this sequence. This can mean that a full set of useful operators is not generated at one go. This can be resolved by making the example sequence slightly artificial, by which we mean that in order to ensure all desirable operators are induced we add on extra tasks to the sequence but have to induce some of the operators twice. The example below illustrates this point.

We developed a new planning domain, called the hiking domain, for the purpose of testing the algorithm, *opmaker*. In this domain world a couple do a long walk over several legs in the English Lake District, between such places as Keswick, Derwent, Honister and Fairfield. Using 2 cars to transport themselves and their equipment to the start of every leg of the journey, Fred and Sue rise in the morning, fit after a night's sleep and take down their tent. The tent is taken by car and erected at the end point of the leg before returning to start the walk. On reaching the destination, tired, they sleep overnight and awake fit again for the following day. Operators required in the handwritten version of this domain were:-

```
put_down, put_up, drive, walk_together, load, unload, getin, getout
```

The operators could be induced by using the following example sequence. This sequence describes a complete day in the walk and has been made long enough to generate all the required operators but several are included more than once. For example there are several drive operators.

```
put_down(tent1,sue,keswick),
load(fred,tent1,car2,keswick),
```

```
getin(sue,keswick,car2),
getin(fred,keswick,car1),
drive(sue,tent1,keswick,helvelyn,car2),*
drive(fred,keswick,helvelyn,car1),*
getout(sue,helvelyn,car2),
unload(sue,tent1,car2,helvelyn),
getout(fred,helvelyn,car1),
putup(tent1,fred,helvelyn),
getin(sue,helvelyn,car2),
getin(fred,helvelyn,car2),
drive(sue,fred,helvelyn,keswick,car2),*
getout(sue,keswick,car2),
getout(fred,keswick,car2),
walk_together(sue,fred,couple1,keswick,helvelyn),
sleepintent(sue,fred,tent1,helvelyn),
getin(sue,helvelyn,car1),
getin(fred,helvelyn,car1),
drive(sue,fred,helvelyn,keswick,car1),*
getout(fred,keswick,car1),
getin(fred,keswick,car2),
drive(fred,car2,keswick,helvelyn),*
drive(sue,car1,keswick,helvelyn),*
getout(sue,car1,helvelyn),
getout(fred,car2,helvelyn)
```

(* These 'drive' operators are different and have different parameters. This is allowed in the *OCL* language which allows us to model conditional operators as we see here. Another way to model these as would be to name each differently such as 'drive', 'drive_tent' and 'drive_passenger'.)

A problem with this method of inducing operators is the repetition. It is not desirable to have several different versions of one operator in any domain and we require a comparison and decision process to help us retain only the most useful

version of any single operator.

Another problem that comes with inducing all the operators in one go is that the method which is induced is not very meaningful. The idea of methods is to use them to break down a planning task into smaller subtasks. Each of these can be broken down again until the smallest subtasks can be solved with a short sequence of operators. This strongly resembles the way humans plan by splitting a problem into chunks recursively until achievable tasks are obtained. A better way to use induction, therefore, is to induce only those operators at any one go that will create a sensible method, but to create enough of these methods to enable hierarchical planning to take place. The subject of this research is the extension and implementation of this algorithm to perform the following tasks.

1. Induce a complete set of operators for a flat domain
2. Induce methods [definition 1.11] and operators in hierarchical domains
3. Detect when an operator has already been induced
4. Use theory revision to sequentially improve domain theories.

Completion of this research allows for the provision of a method operator induction tool for hierarchical domains to enhance the package of design and validation tools in GIPO. This work is significantly different from other research in the area because it concentrates on adding an important construction and validation tool to an overarching system, already in use, and designed intentionally to make the construction of planning domains an easier and more exact process. In this respect this research should be seen as an add-on package to an existing system, although the coding will allow for operator induction independent of the GIPO system.

1.8 The Remainder of this Thesis

Chapter 2 in this thesis describes in detail the object centred domain language and how domains are built using it. The building of the hiking domain, briefly mentioned in this chapter, is described both by hand and also using GIPO. In Chapter 3 operator induction is introduced as we look at the development of *opmaker*1 and we begin to see why a further phase was required. Some new versions of classical domains, altered in order to demonstrate and test sections of code within *opmaker*1, are introduced. Chapter 4, continues the discussion and highlights why it was necessary to move on to *opmaker*2. The action of this new version is described and some results shown which have been obtained using it on some classical domains. Chapter 5 contains a literature survey of recent relevant work in the planning and knowledge engineering research sector. Chapter 6 is a short chapter discussing the way forward. This research can be viewed as steps towards much more fully automated domain construction allowing planning to be integrated into remote agents so that they can plan effectively and autonomously.

1.9 The Aim of this Research

The aim of the work represented in this thesis is to offer a tool which takes the effective realisation of efficient knowledge based systems as a motive. It aims to show that efficiency is addressed by:-

1. making fuller use of knowledge already captured (in states and invariants) saving expert time

2. adding to that knowledge with the capture of operators and methods, further saving expert hand-coding time

3. offering the potential for faster planning times when a good selection of operators and methods is included

4. using an object-based system with strong capture of static knowledge

5. using a system which tracks its own dynamic knowledge as it is built, via the intermediate states found.

1.10 Contributions

We adopt Zimmerman and Kambhampati's definition [89] of an agent: 'a computer program with learning capabilities (means) we can say that learning takes place as a result of the interaction of the agent and the world, and observation, by the agent, of its own decision-making processes'. Given this definition and the aims detailed in Section1.9, this work contributes to the area of research into knowledge engineering of planning domains in the following ways.

1. **Induction of Hierarchical Models**. Where an agent's knowledge is complex and systems include many objects and potential actions, our system can acquire operators (actions) and methods (macros) consisting of models for the completion of whole tasks.

2. **Evidence of Efficiency of Hierarchical Models**. Hierarchical planning has always born the cost of development time for the operators and methods. Our work addresses this issue by automatically acquiring operators and methods, and supports the hypothesis that where domains are complex, planning time is saved by the use of these. In line with the aims in Section 1.9, we argue that the complexity of the domain model in terms of its states and invariants ultimately promotes the efficiency of the process. Since this static knowledge requires little (if any) update, the amount of domain expert time required to

encode it is minimised. Using automated induction, system development costs should be significantly less, offering planning systems as an option where once they would have been discounted on grounds of cost.

3. **Towards True Agent Autonomy**. Whereas our first work (*opmaker*1.1) required some user interaction, *opmaker*2 lays the foundation for autonomous learning. Once the agent has a sufficient set of expert sequences it should be possible to induce sets of operators and methods without further intervention from the expert.

4. **New Versions of Experimental Domain Knowledge**. This work contributes to planning research by the introduction of new benchmark versions of old favourite domains. The extended Tyre Domain increases the complexity of its predecessor whilst the hierarchical Briefcase Domain offers a very simple hierarchical version with the additional challenge of the potential for 'double' conditional operators. Both these domains are available on the website [25].

Chapter 2

DOMAIN MODEL CONSTRUCTION

A planning domain is a knowledge base for a planning application modelled in some planning language. It contains all the predicates describing the relations between objects featured in the domain, all the objects themselves and the states in which those objects can exist, a set of operators and all the constraints needed to make logical sense of the world being modelled by the knowledge engineer. Whilst the problems reflected in these domains are inspired by real-world problems we are essentially reducing these to closed-world planning models. This chapter looks at different ways of representing a planning domain and the different domain modelling languages including our choice of an object-centred representation. Using two representative domains modelled in *OCL* we consider GIPO (developed in a separate project) and the arguments in favour of its use, showing how a domain can be built step by step. For clarity several diagrams are included, mainly snapshots from GIPO in stages of domain construction. Operator construction is shown using both GIPO and our recent addition, the *opmaker* tool, now embedded in GIPO. The chapter concludes with some comments about ease of use and accuracy of the construction tools. A final comment to be made here is the fact that there is another easier way to construct a flat domain using GIPO's life history editor. Figure 1.2 gives a flavour of this tool which is not described for two reasons:

1. It is the subject of a parallel research project

2. The language is completely unseen by the user until the domain is complete. This is a real advantage for the non-expert but we are looking at the language

description of parts of the domain.

2.1 Planning Domain Representation Languages

There are many different ways of representing domain knowledge and different languages have been developed for the purpose. In one useful survey the authors discuss the relationship between agent architectures and representation languages [85]. We mention two representation languages here. PDDL (Planning Domain Definition Language) is the most frequently used language in the planning community. It is action-centred and a domain written in PDDL contains all the operators, which are expressed in a STRIPS-like manner, together with a planning problem description. An example operator in PDDL, from the 'hiking' domain, is shown in Figure 2.1.

```
(:action put_down
        :parameters ( ?x1 - tent ?x2 - person ?x3 - place)
        :precondition (and (fit ?x2 ?x3)(up ?x1 ?x3))
        :effect (and (down ?x1 ?x3)(not (up ?x1 ?x3))))
```

Figure 2.1: The Operator, *put_down*, in *PDDL*.

Here a tent is taken down after a night's sleep in which the person becomes fit (by sleeping). The action is called put_down and is intended to represent the taking down of a tent. Before the action put_down happens, the declared parameters stipulate the objects concerned with this action. These are tent, person and place, of which only tent and person are regarded as dynamic and therefore capable of change. The precondition states that the tent must be up and the person fit and the effects are that the tent is taken down and is no longer 'up'. All of this happens at one place,

as shown by the inclusion of 'place' in the 'fit' precondition, and the person remains fit after the action.

As a result of the 2002 planning competition PDDL has been extended to PDDL 2.1 [21] which allows for a time factor to be taken into account in the planning process. Further developments to the PDDL language are represented in PDDL2.2 and PDDL3 [15, 23]. Previous to this, in classical planning, actions were assumed to be instantaneous. If a problem could not be modeled with instant actions then it was assumed to be a scheduling problem. With these new versions of PDDL there is a renewed interest in temporal planning.

2.1.1 The Argument in Favour of the Object Centred Approach

By contrast *OCL* (Object Centred Language), which was developed at The University of Huddersfield, [40, 43, 45] takes the objects rather than the operators as its focus. This seems more natural and enables a richly expressive domain structure. The Object-Centred Language and its associated development method forms a rigorous approach to capture the functional requirements of classical planning domains.

We can best justify the development of the object-centred approach and the *OCL* language by acknowledging the need to develop a precise domain model. This is best done based on a language offering a formal framework, which will be described in detail later in this chapter. This framework allows for analysis and checking as the domain is built and tools have been developed for this. Using the object-centred approach a domain model has the advantage of a very structured development method together with efficiency of planning algorithms [35].

The completed model of the planning world together with valid states and operator schemas offer the knowledge based system community a bridge between the conceptual models of informal knowledge acquisition methods (such as KADS [1, 76])

and implementations of knowledge-based systems [38], as well as being important in the verification and validation of KBS [51]. In planning, the construction and validation of a domain model is therefore recognised as an essential stage in the construction of a final system [45].

A more detailed description of the *OCL* operator construction will be given in Section 2.4.5 but some explanation is necessary here. An *OCL* operator consists of four components as shown below.

```
operator(name(parameter1, parameter2, ...,parameterN)
prevail clauses list
necessary transitions list
conditional transitions list).
```

The first of these states that an operator is being described, gives its name and declares the objects (parameter1, parameter2 etc) included in the description. The *OCL* representation of the same operator, instantiated, is shown in Figure 2.2. The operator's name is put_down[1] and the parameters are Tent1, Fred and Keswick. The next component lists states which prevail throughout the action. In Figure 2.2 'se' (a state expression) indicates such a prevail - here Fred is fit in Keswick both before and after putting down the tent. The third component shows the necessary transitions which must occur when the action happens and, in Figure 2.2, where 'sc' (state change) indicates that this is a necessary transition, these are shown as a list of state(s) for the tent to the left of the $\Rightarrow$ prior to action, whilst the right hand side shows the state after the action. The final component shown in Figure 2.2 in this particular operator shows an empty list but is used to show any conditional transitions in the same format as the necessary transitions of the third line. We could think, perhaps, of a flag on top of the tent which would change from being up and flying to being down conditional upon the tent being up or down. (See also Section 2.4.5, Figure 2.12.)

---

[1] This is a 'user' supplied operator name corresponding to take_down in Appendix A

```
%name and params
        operator(put_down(Tent1,Fred,Keswick),
        %prevails
        [se(person,Fred,[fit(Fred,Keswick)])],
        %necessary
        [sc(tent,Tent1,[up(Tent1,Keswick)] => [down(Tent1,Keswick)])],
        %conditional
        []
        ).
```

Figure 2.2: The Operator, *put_down*, in *OCL*.

The remainder of this chapter refers to aspects of domain construction using the *OCL* language and a hierarchical version of *OCL* named OCL_h. For a detailed description of the construction of a domain using *OCL* the reader should consult the *OCL* Manual [40] and the GIPO on-line manual [25] but the main points of domain construction will be summarised in this chapter.

2.2 Some Example Domains

There are a number of classic domains in existance which would serve to illustrate the process of domain construction. Blocks World is one such, which has many versions and was used to illustrate the STRIPS methodology. Mostly versions consist of a set of blocks, a table top and a gripper arm which can manipulate the blocks, altering their relative configuration. We have chosen a different flat domain, the hiking world [46], which is described in more detail in Section 2.2.1 and which has the following features:

- It models a real situation.
- It contains enough detail to make the construction and planning interesting.

- It requires several operators, some of which have conditional clauses.
- It contains both static and dynamic objects.

We have chosen the University of Maryland UM Translog Planning Domain [2] as an example of a much more complex hierarchically structured domain. It has a rich set of entities, attributes, actions and conditions, which makes for lengthy plans with many alternatives, and we have several versions of this domain available for use which have been translated into OCL_h. Whilst the work in this thesis is based on the full version of this domain some diagrams show a simpler version to illustrate a point without too much of the detail.

2.2.1 The Hiking Domain

The Hiking Domain [58] describes a hiking holiday in the English Lake District. It models a couple doing a long circular walk over several days between such places as Keswick, Helvelyn, Fairfield, Honister and Derwent. A place is next to another if the second can be reached from the first by a day's walking which is always in the same direction (clockwise or anti-clockwise). Fred and Sue walk a leg each day to arrive at the night's stop-over tired but with the welcome sight of their tent ready and waiting for them. They achieve this by using two cars to move their equipment around and to transport themselves to the start point of each day's walk. In a typical day they would need to

1. use one of the cars to fetch the other from the previous stop-over
2. take down the tent and drive it and both cars to the day's destination
3. erect the tent there and, leaving one car, return in the other to the start of the day's leg

4. leave the remaining car and walk the journey leg, arriving tired

5. sleep overnight in the tent to awake fit the following morning

6. repeat the process until the walk is completed.

A full listing of the Hiking Domain is given in Appendix A.

2.2.2 The UM Translog Domain

The version of the UM Translog Domain used for this work has been translated into OCL_h and so to distinguish the OCL_h version I shall refer to it as simply the Translog Domain. This domain was contrived to model a transport logistics problem and, whilst still a model-in-miniature of a real transport problem, it is nevertheless a large domain with many and varied alternatives and thus a good test of a planner. It models the transportation of a variety of different 'packages' between three cities. Cargo may go by road, rail or air so long as a route can be found. 'Packages' may be large bulky parcels, liquid, grain, cars or livestock and any package may be valuable, requiring guards to accompany the package, or hazardous, requiring decontamination procedures before the next package can be transported. There are an assortment of locations within the cities such as the railway station, the post-office, an airport and a city-location. Also there is a variety of equipment to load the packages such as cranes, ramps for livestock, bulky packages or cars to be loaded into the relevant transport, hoppers for grain and hoses for liquid. Packages have to be 'certified' by paying a fee before they can be transported. This domain is large enough to have many operators both primitives (non-hierarchical) and methods, and also has many constraints. Constraints include atomic invariants that state, for example, which vehicle object is suitable for which cargo, or which locations belong to which city. It has sorts arranged in a tree structure with several levels, and the methods are

arranged in a method hierarchy. The sort structure for this domain is shown in Figure 2.5 whilst Figure 2.6 shows the methods diagrammatically. Whilst methods are named in this diagram we show, in Figures 2.3 and 2.4, two random methods chosen to be representative of all the many methods available. (In these figures 'ss' (substate) indicates substates for the objects 'Package' and 'Train' respectively.) For a full listing of the OCL_h the reader is referred to [58].

```
method(transport(Package,Org,Dest),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(package,Package,[uncertified(Package),at(Package,Org)]=>[delivered(Package),at(Package,D
    % Static
    [
     in_region(Org,Region),
     in_region(Dest,Region)],
    % Temporal Constraints
    [
     before(1,2),
     before(2,3)],
    % Decomposition
    [
     achieve(ss(package,Package,[waiting(Package),certified(Package),at(Package,Org)])),
     carry_direct(Package,Org,Dest),
     deliver(Package,Dest)]
).
```

Figure 2.3: The Method, *transport*, in OCL_h.

```
method(move_traincar(V,O,L),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(traincar,V,[at(V,O)]=>[at(V,L)])],
    % Static
    [
     is_of_sort(V,traincar),
     connects(R2,O,L),
     is_of_sort(R2,rail_route),
     is_of_sort(Train,train)],
    % Temporal Constraints
    [
     before(1,2),
     before(2,3),
     before(3,4)],
    % Decomposition
    [
     achieve(ss(train,Train,[at(Train,O)])),
     attach_traincar(Train,O,V),
     pull_traincar(Train,O,V,R2,L),
     detach_traincar(Train,V)]
).
```

Figure 2.4: The Method, *move_traincar*, in OCL_h.

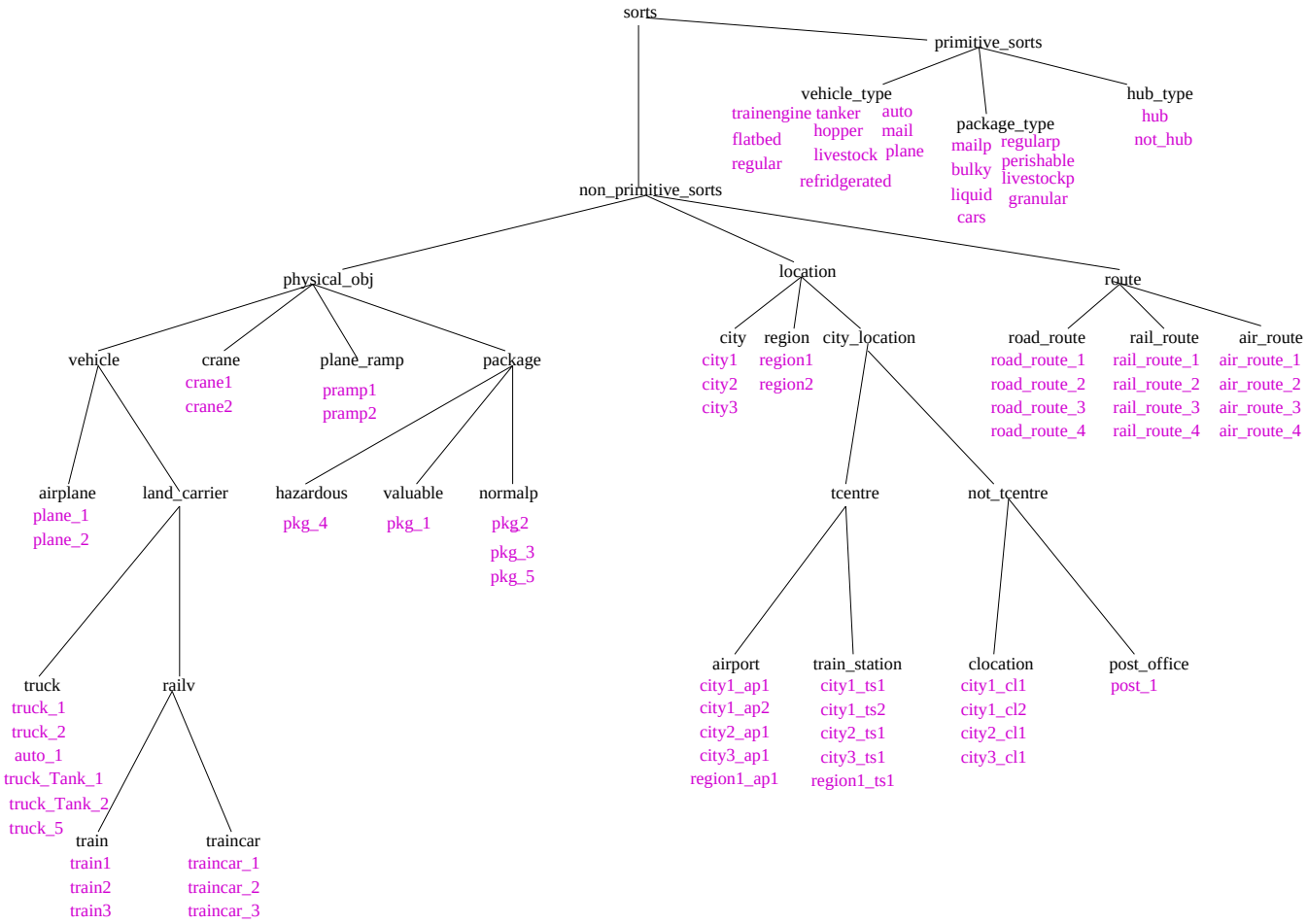


Figure 2.5: The Sort-Tree Showing Objects for the Translog Domain

2.3 GIPO

In this section we look at GIPO itself, argue the reasons for its development and show how *this* research fits into GIPO's bigger picture. As planners and planning applications become larger, the problems of engineering planning domain models become more acute. Engineering platforms are required that allow a domain expert to enter domain knowledge at a high level of abstraction, and to facilitate the gluing together of planning tools to help in domain modelling [58, 74, 38]. In particular, if AI planning is to provide a solution for end-user problems then a system of construction

Figure 2.6: The Method and Operator Structure for the Translog Domain

of detailed domains is required. The argument for the development of GIPO has already been raised in relation to the difficulties of domain construction and operator construction in particular. Another problem existing in the planning community concerns the general difficulty of planning and the time it takes to learn enough about the field to use the technology that planning offers on a wider scale. The ideal solution is a sort of abstraction allowing model building to be separated from coding in much the same ways as Windows-style environments allow many novice users to manipulate data without knowing an operating system language like DOS, or in modern integrated software development environments where CASE tools shield the user from the languages underneath. This abstraction would allow domain modelling to be speeded up and become more cost-effective. In this way it should be possible for the wider community to embrace planning and find uses for it in many applications.

For these reasons a tools package has been recently developed and offered to the community at an experimental level. The package, GIPO, [70, 44, 25] a Graphical Interface for Planning with Objects, is an experimental Graphical User Interface (GUI) and tools environment for building classical planning domain models. It provides an interface that abstracts away much of the syntactic detail of encoding domains, and embodies validation checks to help the user remove errors early in domain development. GIPO has a series of editors for each stage in domain development and we concentrate on these in the next section. These editors allow complete domains to be constructed, either flat or hierarchical without the user knowing *OCL*. GIPO also integrates a range of planning tools - plan generators, a stepper, an animator, a random task generator, a reachability analysis tool - all to help the user explore the domain encoding, eliminate errors, and determine the kind of planner that may be suitable to use with the domain.

This project has contributed towards the GUI with the introduction of an algorithm *opmaker* [46] [47] that induces operator descriptions from a user given example

sequence. Essentially, the user supplies examples of action sequences by describing all the objects that these operations affect. Where there is a choice of the target state for a dynamic object in an operation, the algorithm requires the user to point and click on that state. The whole process helps the user abstract away from the particular syntax and consequential errors, and in particular having to encode operator schema using a symbolic language with subtle uses of parameters. A more detailed description of the *opmaker* tool can be found in Section 2.4.5.

The present version of GIPO allows for hierarchical domains to be constructed by having an hierarchical transition editor, but the version of the *opmaker* tool implemented is for flat domains only. *Opmaker*2, not yet embedded in GIPO, will induce a set of operators and a method. But as methods are built some operators are repeated in different sequences. A systematic approach to inducing operators was required which aims to reduce repetition and this research shows a way in which operators can be induced gradually as the required methods are built thus minimising the repetition of induced operators.

2.4 Object Centred Language

The object centred domain modelling method provides a tool-supported language for the capture and implementation of planning domain models. The structured language leads to the sectional development of the whole model with validation and support tools available to the developer. Once operational, the object-centred representation has advantages in the development and resulting efficiency of planning algorithms. In the following sections we see how a natural English description of a domain for planning such as that given in Section 2.2.1 can be coded into the *OCL* representation in a step-wise manner. We use, as a running example, the building of the hiking domain as we follow through the steps of its construction.

2.4.1 Sorts and Objects

Sort structure hand coded

Reading through the natural English description in Section 2.2.1 allows a set of objects to be identified and classified into sorts. In the hiking domain the sorts are person, place, car, tent and couple. These, because this domain is a flat structure, are classified as primitive sorts. The code for these is below.

```
sorts(primitive_sorts,[car,person,tent,place,couple]).
```

Objects are of two types, static and dynamic. Sorts of dynamic objects are person, car, tent and couple, whilst place is static, and this fits in with our intuitive ideas about these sorts. We shall see later that only dynamic objects are capable of state change which becomes important in operators.

```
% Sorts
sorts(primitive_sorts,[car,person,tent,place,couple]).

% Objects
objects(car,[car1,car2]).
objects(tent,[tent1]).
objects(person,[sue,fred]).
objects(couple,[couple1]).
objects(place,[keswick,helvelyn,fairfield,honister,derwent]).
```

Figure 2.7: Part of the Coding of the Hiking Domain Showing the Sort Structure

Next specific objects belonging to the sorts are identified. These will be the cars, named here car1 and car2, the people, named fred and sue, the places, named keswick,

derwent etc, the couple's name (couple1) and tent (tent1). When these are identified and coded by hand the domain now looks as shown in Figure 2.7.

Figure 2.8: The Sort-Tree View Using GIPO and Showing OCL Implementation

Sort structure coded using GIPO

A domain developer using GIPO would not have to deal directly with the coding. Using GIPO's sort editor, he uses dialog boxes to name the sorts and adds objects to the resulting sorts tree structure by clicking on the sorts, Figure 2.8. Unseen by the developer unless requested, GIPO generates the same code that appears in Figure 2.7. The code can be viewed at any time by clicking the view option when it is displayed in *OCL*. It can also be viewed in *PDDL*, making GIPO a useful tool for the planning community at large.

2.4.2 Predicates

As can be seen in Figure 2.9 a set of predicates expressing relationships between the sorts is the next thing to be modelled. Logically these cannot be modelled until the sorts have been expressed and the predicates are required to express the various states in the model, so their construction is the next ordered step in domain construction. Of course if the domain is being built using GIPO then it is possible to revise the sorts after the predicates have been constructed, although the predicates would also need revision. The predicates themselves stem from the natural language description of the world though at first glance some may not seem obvious. For example the predicate

```
next(place,place).
```

does not seem intuitive but stems from the part of the description that states the walk is circular and the couple walk the next leg of the journey (the second argument 'place'). So here we make specific that places can be next to one another. Note that here we are just establishing the predicates in general terms, any object of sort place could be substituted for place in the predicate. Later we shall see in the atomic invariants, Section 2.4.4, that we can specify exactly which places are next to each other.

Another interesting feature here is that a simple 'at' predicate is not allowed by GIPO because of the clash between the sorts of the subject of the predicate. Instead we need 'at_tent', 'at_person' and 'at_car' for the subjects tent, person and car. *OCL* does not allow for a hierarchy of predicates but does allow use of a hierarchical sort structure. If this had been used then it would be possible to have tent, person and car belonging to the same sort, say 'thing' and the 'at' predicate applied at the level of 'thing' would then allow all the subjects individually to be used in the same 'at' predicate.

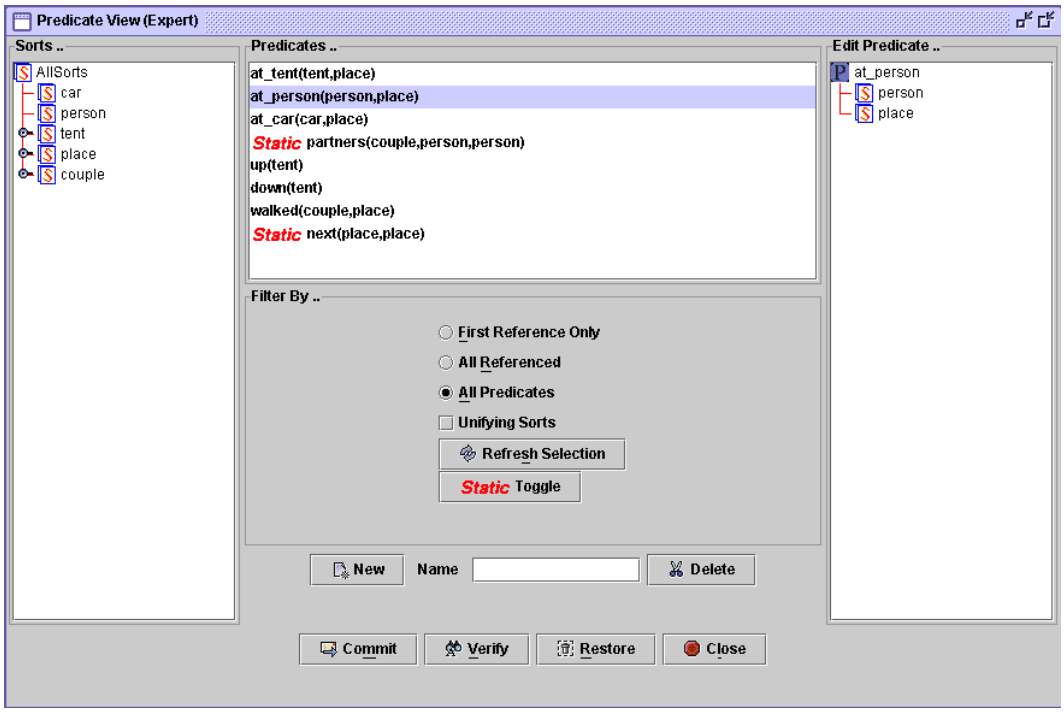


Figure 2.9: The Predicate Editor View Using GIPO

All predicates, once declared, keep the same arity, and this can be used as further validation. Some predicates such as up(tent), and down(tent) have an arity of one. The partners predicate has arity three and states that a couple consists of a person and a person. So does that mean that the same person twice could be a couple? If we refer to the atomic invariants, in Section 2.4.4, we find the statement

```
partners(couple1,sue,fred)
```

which constrains person and person to be two different people, a particular couple comprising a particular combination of the object person. Again this is a built-in validation feature of the GIPO software which reflects the completeness of the *OCL* language. This particular predicate is one of the static predicates in this domain - as such it is always true. The other static predicate

```
next(place,place).
```

reflects that places do not move. Other predicates in this domain are dynamic and remain true, in their instantiated format when used in the substate expressions (see Section 2.4.3), only until they undergo a change of state in an operator.

2.4.3 States

Once a set of predicates is complete the states in which the objects can exist are the next thing to be constructed. These are lists of substates which are put together to reflect the different states for the objects concerned. For example, if the object concerned is the tent then, in the predicates, we have already constructed three predicates of which tent is the subject. These are

```
at_tent(tent,place).
up(tent).
down(tent).
```

giving potentially six sets of substates (at_tent, up, down, at_tent and up, at_tent and down, and at_tent and up and down). Not all of these make sense or are required and careful thought in the hand-coded model is needed to correctly identify the relevant substates. The GIPO states editor, which is shown in Figure 2.10, assists in this decision process because it groups potential substates in the right window, separating them by a line. As can be seen in the figure only two of the six potential substates are required. To express the idea that a tent must always be at a place and that it can be either up or down, *OCL* allows us to unite two predicates into one substate set so the only substates required are those shown in Figure 2.10. In our later discussion of *opmaker*2 we shall see that these substate sets, which are effectively constraints, can be used to reduce the search space of potential state change pathways. The mechanism for input of the substate sets into GIPO is by point and click. For the example shown in Figure 2.10, the user would select the object tent in the left hand window. All the predicates are listed in the predicate window and from these the user can see those which feature the tent. Each predicate he clicks is added to the

'Editable State' window so by selecting the first pair of predicates shown in the 'State Definitions' window of Figure 2.10 he has put them together as a pair. He then clicks the 'Add' button at the bottom to commit the pair to the 'State Definitions'. Figure 2.10 shows that the user has just clicked 'Add' to complete editing the substates for tent. The substates are a very useful mechanism for omitting combinations that would not make sense of the world being modelled so, for example, there is no substate set where the tent can be both up and down.

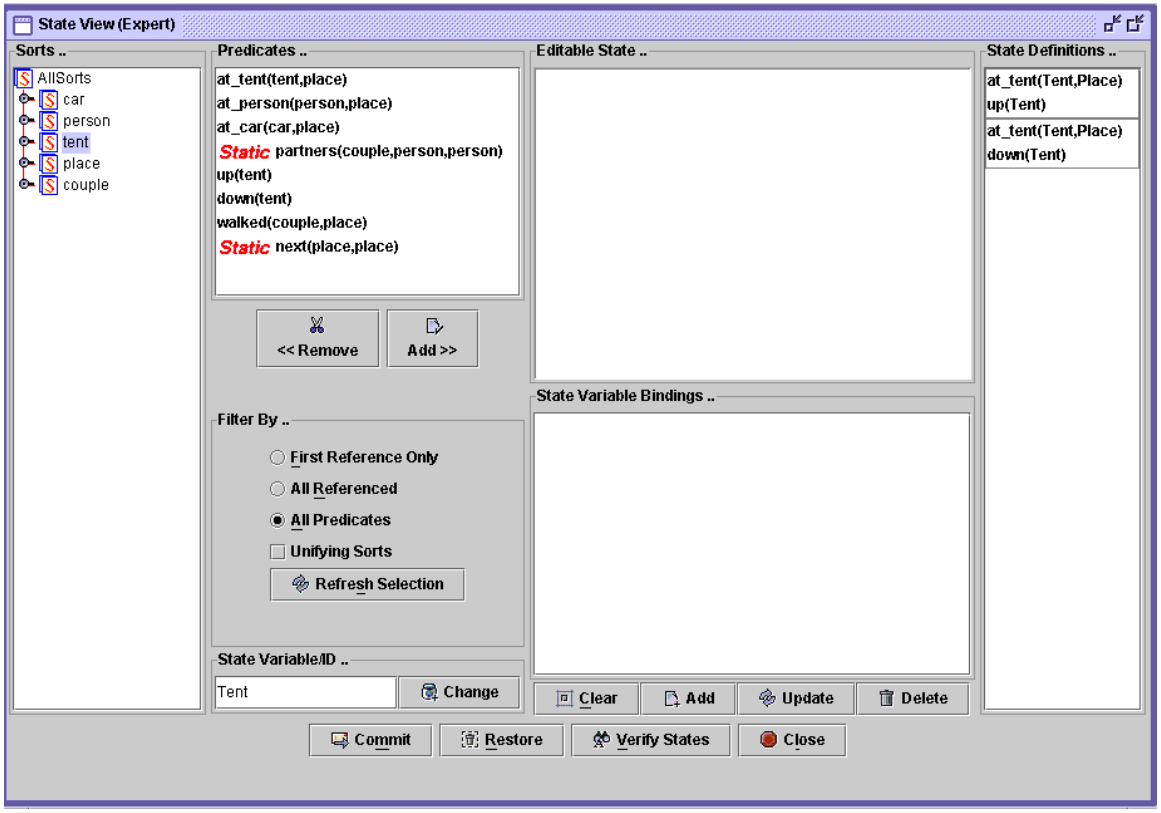


Figure 2.10: The States Editor View Using GIPO

The hand-coded definition of the substates is shown below. The reader should note that the substate sets listed are mutually exclusive so, for example, a person may be fit at a place or he may be tired at a place but he cannot be both. Thus no constraint is needed to state this explicitly.

```
% Object Class Definitions
substate_classes(person,Person,[
         [at_person(Person,Place),fit(Person)],
         [at_person(Person,Place),tired(Person)]).
substate_classes(couple,Couple,[
         [walked(Couple,Place),partners(Couple,Person1,Person2)]]).
substate_classes(tent,Tent,[
         [at_tent(Tent,Place),up(Tent)],
         [at_tent(Tent,Place),down(Tent)]]).
substate_classes(car,Car,[
         [at_car(Car,Place)]]).
```

2.4.4 Invariants

The domain modeller defines facts that make explicit any assumptions about the model. The atomic invariants set the boundaries of compatibility between objects in the domain. The inconsistent constraints state what cannot be true in objects' substates and the implied invariants state explicitly what is implied by existing substates.

Atomic invariants

Atomic invariants list the specific instances of some of the predicates. So for the predicate next(place,place) they state exactly which place is next to which other place. The full set of atomic invariants for the hiking domain is shown below. Here it is explicitly stated that couple1 consists of the sue and fred partnership, and that keswick is next to helvelyn etc.

```
  % Atomic Invariants
atomic_invariants([
         partners(couple1,sue,fred),
         next(keswick,helvelyn),
         next(helvelyn,fairfield),
         next(fairfield,honister),
```

```
next(honister,derwent)]).
```

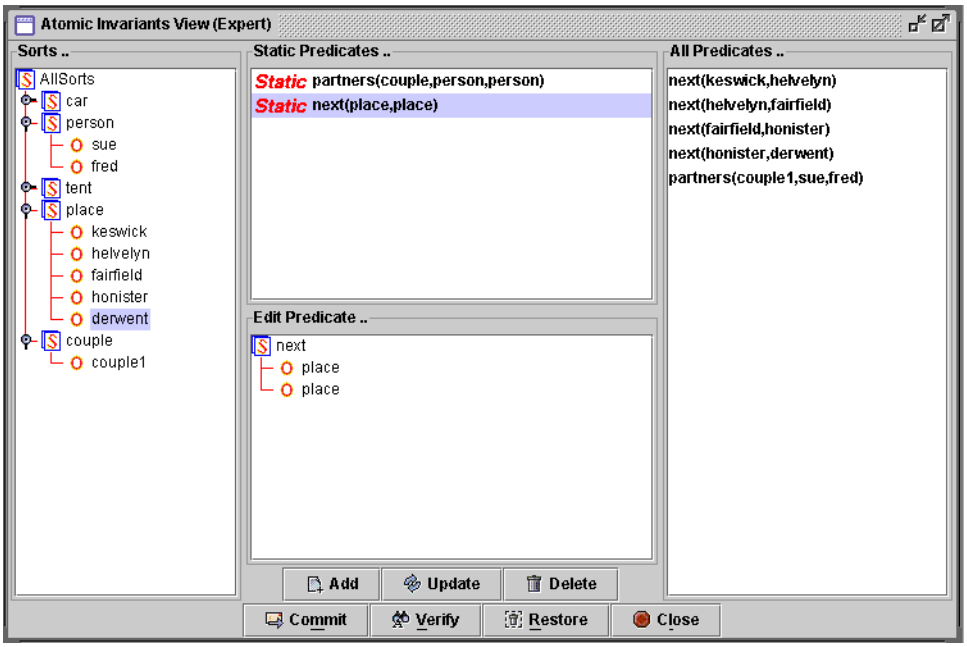


Figure 2.11: The Atomic Invariants Editor View Using GIPO

GIPO has a simple to use atomic invariant editor found under the edit menu. Using this editor, shown in Figure 2.11, static predicates can be edited by highlighting the predicate, expanding the sort tree, dragging the objects onto the sorts displayed in the edit window and adding the edited invariant to the all predicates window. The figure shows the editing of the predicate

```
next(place,place).
```

Implied invariants

Implied invariants state explicitly what is implied by existing substates. The hiking domain has no implied invariants. An implied invariant from the translog domain is...

```
implied_invariant([loaded(P,V)],[at(P,L),at(V,L)]).
```

To explain this invariant if P is a package, V is a vehicle and L is a location then this invariant says, 'If a package, P, is loaded in a vehicle, V, then both the package, P, and the vehicle, V, are at the same place, L'.

Inconsistent constraints

Inconsistent constraints describe things that may seem obvious but should be stated explicitly and make a useful debugging tool. They always record incompatibilities. There are no inconsistent constraints in the hiking domain but we can see one in the translog domain, where P is a package.

```
inconsistent_constraint([certified(P), not_insured(P)]).
```

Here the constraint is concerned with the way payment for transportation is made. When a package is to be transported a fee must be paid which covers transportation and insurance. Once the fee is paid the package becomes certified. This constraint is saying that an uninsured package is not compatible with being certified or, removing the negatives, to be certified is to be insured.

2.4.5 Operators

As has been previously stated, the hand-coding of operators is the hardest part of domain construction, requiring much careful thought and time. As a novice to planning at the beginning of this project, the writer can confirm this statement! In this section we consider the hand-coding of an operator from the hiking domain and we look at two tools offered by GIPO to make this coding easier. We begin by examining the structure of an operator in greater detail. The operator we consider here is the drive operator from a version of the hiking domain which allows conditional operators, and is shown in Figure 2.12. This domain has slightly different predicates from the version discussed so far and in particular contains the predicate

```
in(person,car,place)
```

to express that a person is in a car and both the person and the car are at a certain place. In this operator the list of parameters after the operator name in its first line, shows us that not only is a car being driven by a person from place to place, but also a second person and a tent are featured.

```
operator(drive(Person,Car,Place,Place2,Person2,Tent),
    % prevail
    [],
    % necessary
    [
     sc(person,Person,[in(Person,Car,Place),fit(Person)]=>
      [in(Person,Car,Place2),fit(Person)]),
     sc(car,Car,[at(Car,Place)]=>[at(Car,Place2)])
     ],
    % conditional
    [
     sc(person,Person2,[in(Person2,Car,Place),fit(Person)]=>
      [in(Person2,Car,Place2),fit(Person)]),
     sc(tent, Tent, [loaded(Tent,Car,Place)]=>[loaded(Tent,Car,Place2)])
     ]
     ).
```

Figure 2.12: The *OCL* Drive Operator from the Hiking Domain (Conditional Version).

In Figures 2.12 and 2.2 the reader will note that the operator has four sections of code separated by three comments which are preceded by the % symbol and simply give a heading to the state transition lines. The first line of code begins with 'operator' and then gives the operator name, 'drive' or, in Figure 2.2, 'put_down'. Next follow the parameters in braces which have initial capitals showing that these are variables. When constructing this operator the domain modeller has to give thought to the

objects she aims to manipulate with it and these objects are represented by the variables.

Having identified the objects the domain modeller next turns her attention to the substate lists for those objects. For each dynamic object she needs to consider what states exist for the object before the action takes place and what states it will assume after the action takes place. These states will all be drawn from the substate lists already constructed. She will also have to consider if any of the dynamic objects does not change state. For this operator she wishes to model the action that takes a car, driven by one person with another as passenger from one place to a different place whilst carrying a tent. To model the prevail states she has to consider which objects from the parameter list will not change state. The substates for this conditional version of the hiking domain are shown below.

```
substate_classes(person,Person,[
    [in(Person,Car,Place),fit(Person)],
         [at_person(Person,Place),fit(Person)],
         [at_person(Person,Place),tired(Person)]).
substate_classes(couple,Couple,[
         [walked(Couple,Place),partners(Couple,Person1,Person2)]]).
substate_classes(tent,Tent,[
         [at_tent(Tent,Place),up(Tent)],
         [at_tent(Tent,Place),down(Tent)],
         [loaded(Tent,Car,Place)]]).
substate_classes(car,Car,[
         [at_car(Car,Place)]]).
```

Taking the objects one at a time, she would decide in this case that all the dynamic objects must change state since at least one state of the possible state sets changes, namely all the dynamic objects begin by being at(*Object*,Place) and end by being at(*Object*,Place2). Strictly speaking, for this model, the objects are either 'at' themselves (the car) or 'in' a car which is 'at' (person or tent), but other models are possible. Since all the dynamic objects change state, the prevail states, i.e. the states

which don't change, is an empty set, and this is shown in the second line of the code for the operator.

The next consideration in modelling this operator are the necessary and conditional changes. Here our modeller must think carefully about which are essential parameters and which are the 'extras'. Clearly cars do not yet drive themselves so she reflects in her modelling that one of the objects of sort person must be required and so will undergo a necessary change whilst the other person, as passenger (or back-seat driver) undergoes a change that is conditional upon the car being driven. The same conditional change is true for the tent, whilst the transition for car is clearly necessary, so the modeller can decide to put transitions for the car and the driver under the necessary changes whilst those for the passenger and the tent are put under the conditional changes. In order to accurately reflect those transitions the modeller considers preconditions and effects for each object. For example for the driver of the car, 'Person', there is only one possible state to consider. Initially the driver is fit and in a car which is at 'Place' and the effect of the operator is to put the driver, fit, in a car at 'Place2'. This transition is represented in the first line of code under the '% necessary' heading of Figure 2.12 which shows the precondition to the left of the $\Rightarrow$ symbol and the effect to the right of the $\Rightarrow$ symbol. The second line under this heading reflects the similar necessary transition for the car. Of course the modeller has the responsibility to match all the correct before and after parts since it would make no sense for the driver and tent to move from A to B whilst the passenger and car move from B to A! Conditional transitions are often an empty set in an operator but where they exist, as in this case, they are modelled exactly like the necessary transitions.

The domain modeller can build a full set of operators in this way or they can be built using the tools that GIPO offers. We now look at the first of these, the transition constructor, which allows similar thought processes but requires no expertise in *OCL*.

The transition constructor

Figure 2.13 shows the transition editor when it is first opened. There are five window areas in the editor one of which displays the sorts constructed in the sort editor, Figure 2.8.

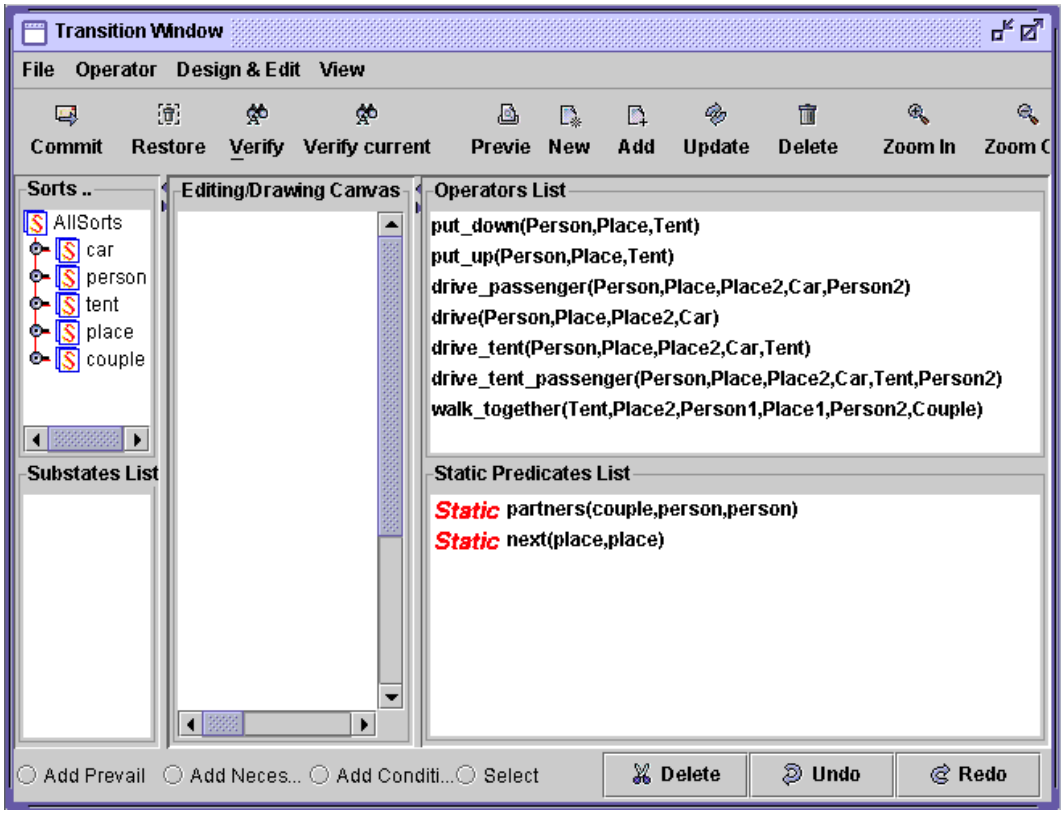


Figure 2.13: The Transition Editor View Using GIPO

Clicking on a sort in this window displays the lists of substates for that sort in the the relevant window. The Editing/Drawing Canvas is for the graphics and will be expanded in the next figure to show a graphical representation of a newly constructed operator. The Operators List shows the list of operators already constructed along with their parameters. The Static Predicates window reminds the user about the static predicates which have already been constructed and which cannot be changed

in an operator.

By clicking on the New button the user can select a name for the operator he wants to construct. When he confirms the name the drawing canvas is activated, displaying a graphical version of the name. This window can be expanded and a view of the left and central windows with the drive operator under construction is shown in Figure 2.14.

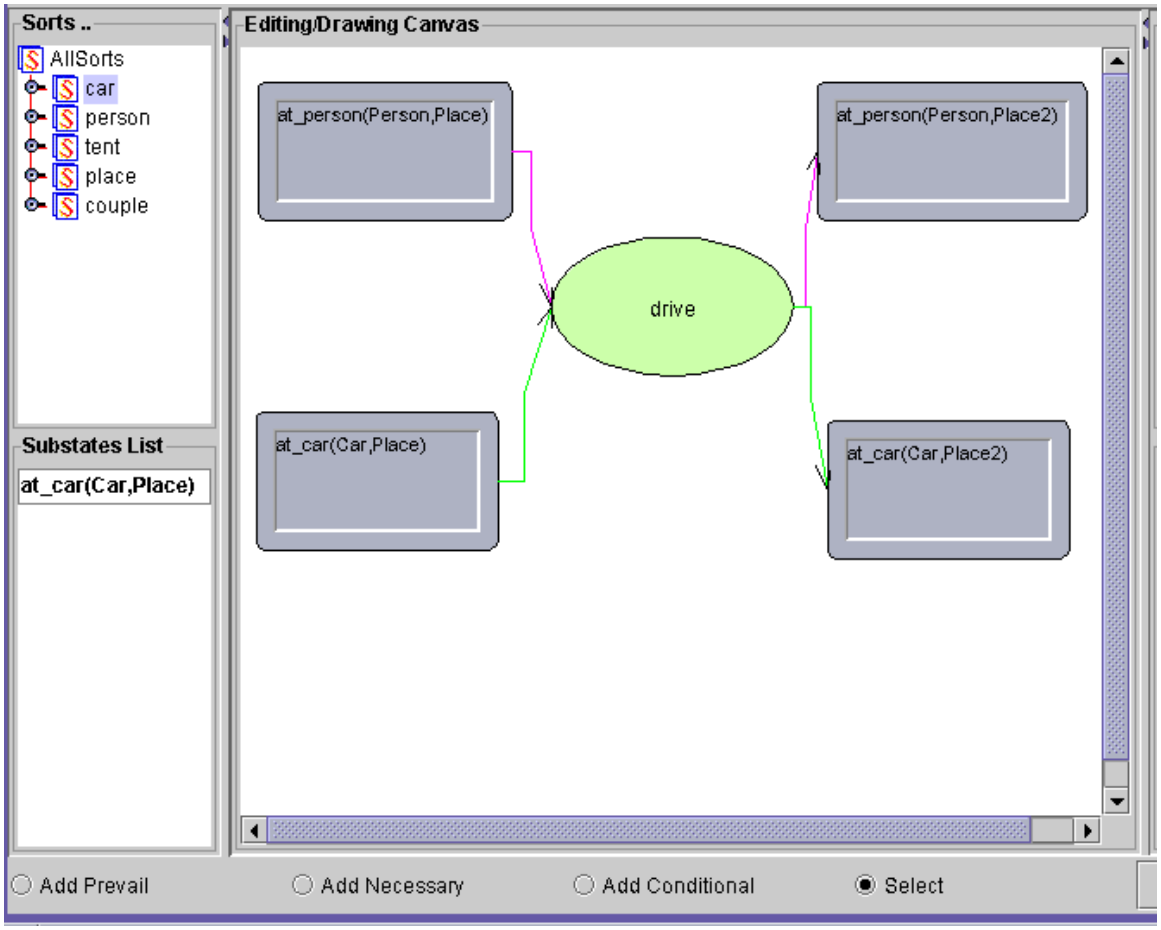


Figure 2.14: The Graphical Representation of an Operator using GIPO

The user has named the operator which appears in the oval-shaped area and has completed the transition for the object person. We can now track what he does to

add the transition for the object car. In the sorts window he selects car. As he does this the possible substates for car appear in the substates window. He selects the only choice (at_car(Car,Place)) and clicks necessary at the bottom to add the transition to the operator. Any ambiguity in the object of the transition is cleared up by GIPO with a pop-up dialog box and then GIPO draws in the LHS part of the transition and prompts the user to select a substate for the RHS. In this case there is only one choice which the user selects, giving the graphical output shown in Figure 2.14 which shows the graphical view after it has been edited. By clicking in the edit tick box (not shown in Figure 2.14) the user is able to edit the RHS of her transition. The original transition would reflect the substate for car and would read (at_car(Car,Place)) on both sides. This transition must be edited so that the user can indicate that 'Place' on the LHS is not the same as 'Place' on the RHS. After this edit the RHS of the car transition shows 'Place2' indicating that a different place to the initial place must be an effect of the drive operator. Once the user is satisfied with the new operator she clicks an add button (not shown) and the operator name and parameters appear in the 'Operators List' window where completed operators are listed whilst the description of the operator is added to the code, viewable at any time by clicking the view menu.

2.4.6 The opmaker tool

Hand-coding operators requires a domain expert who has a firm grasp of the *OCL* language and a large amount of time. Use of the GIPO system using the transition editor is quicker and knowledge of *OCL* is not required, but a novice would struggle even with this tool until they grasped the concepts of forming the transitions. Undergraduate students studying a module on Artificial Intelligence have built domains using GIPO and their results can be viewed on-line at [72]. If the aim of the planning community is to bring planning within the grasp of the world of software developers on a large scale then even more abstraction is needed. This is partly provided by

the *opmaker* tool. At the time of writing the *opmaker* tool embedded in GIPO is *opmaker*1 which still requires user input. This is described in more detail in the next chapter. It was anticipated that the follow up version, *opmaker*2, would allow operators to be induced fully automatically and we describe work towards this in a later chapter.

For the following discussion of the use of *opmaker*, we assume the domain developer has managed to develop her domain to the point where everything up to the operators has been developed. In the hiking world this would be at the stage where sorts, predicates, substates, and invariants have been declared. Further than this the developer should have thought about the actions she requires to be able to use. She should know, for example, about any 'moves' the objects should make, and which objects would be involved.

To see how the *opmaker* tool works let us imagine a typical situation in the hiking domain. Sue has taken down the tent at Keswick in the morning and driven with it to Helvelyn. Here she puts up the tent ready to sleep overnight after the walk. They need to leave one car at Helvelyn so Fred must pick her up. Effectively the task to perform here is for Fred to drive the second car to Helvelyn and to drive it back to Keswich with Sue as passenger. To stipulate such a task to a planner it would be necessary to declare the initial states and target states of all the objects. In this example initial states are that Fred and car2 are at Keswick, Sue and car1 are at Helvelyn and the tent is up at Helvelyn, whilst goal states are that Fred, Sue and car2 are at Keswick whilst car1 is at Helvelyn and the tent remains up at Helvelyn. So we can say, then, that a task consists of a set of initial states and a set of goal states for the domain objects whilst a plan is the achievement of reaching the goal states from the initial states. In the following series of figures we see how GIPO is used to construct this task and *opmaker* to build the operators required.

In Figure 2.15 we see the required task under construction. The 'Initial State' and

'Goal State' windows show that construction is almost finished. The initial states for Sue, Fred, the tent and the cars have been completed. The user wants to declare the goal state for car2 and has clicked on car2 in the 'Sorts' window. This has placed the state for the car in the 'States' window but it needs to be edited so that 'Place' can be selected. In the 'Edit States for Task' window the user is able to bring up a submenu of available places and will select 'keswick' for the goal state of car2. Clicking the 'Goal' button then commits this last state to the task which can then be verified and saved.

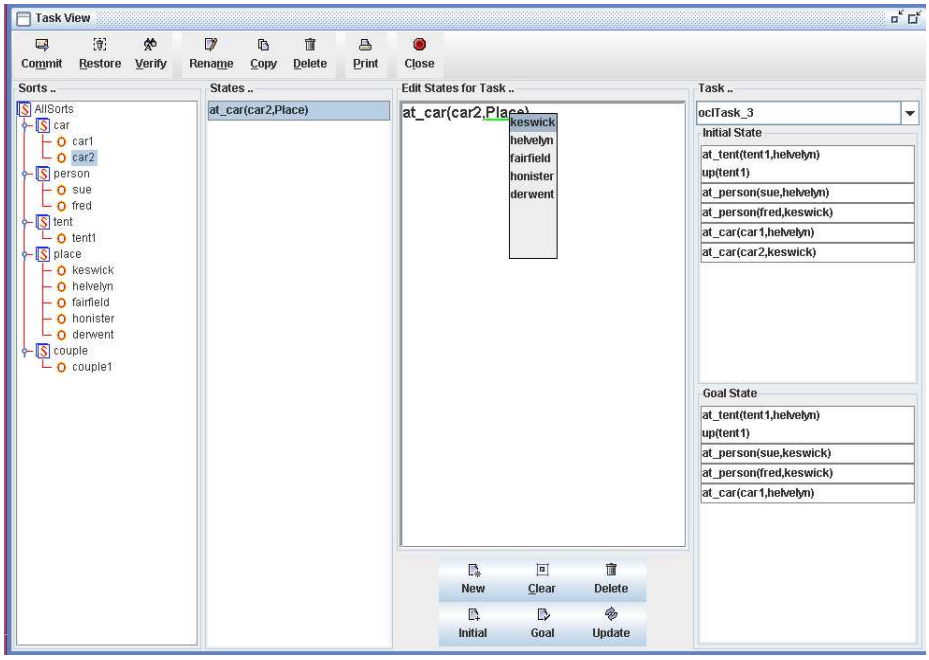


Figure 2.15: A Task is Constructed Using GIPO's Task Editor

Using *opmaker* the first thing to be done is to declare the initial sequence. Figure 2.16 shows this process at the stage where the first action has been declared and the

second is also complete. The user has typed into the 'Action Name' box a suitable name for the second action (here 'drive_passenger' has been chosen). This action name has been placed in the 'Edit Action' window and by clicking and dragging objects from the object tree the parameters have been added. At the same time the action builds up in the 'Action Sequence' window. Since this short sequence is now complete the user will click the 'Generate Operators' button, and this action starts a dialogue process with the user, part of which is shown in Figure 2.17.

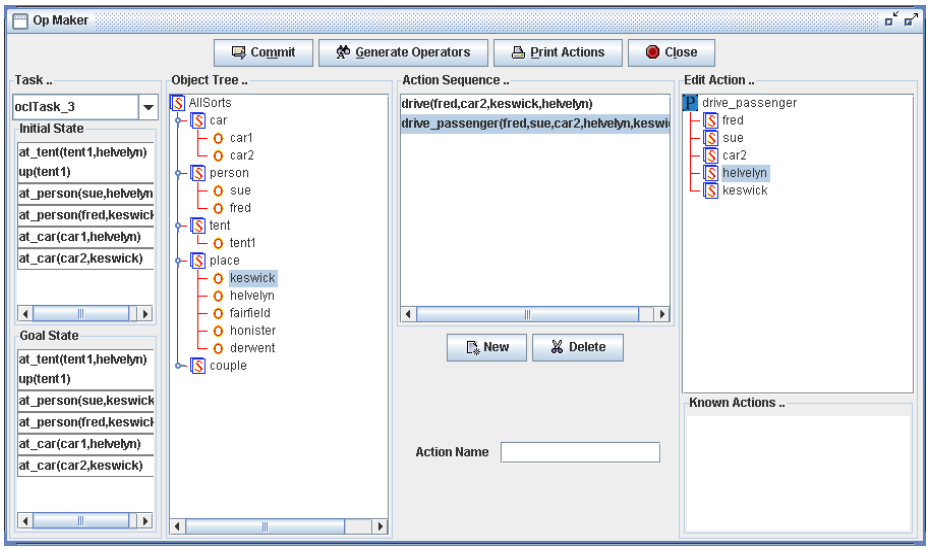


Figure 2.16: Constructing a Sequence Using GIPO

When the snapshot shown in Figure 2.17 was taken the user would have already answered several questions entering intermediate or goal states for objects. We see that the question relates to the second operator and asks the user to specify where 'fred' is after he has driven his passenger. Similar questions will appear for 'sue' and 'car2' until, in Figure 2.18, *Opmaker* has enough details to construct the operators

for the chosen task (shown in the 'Task' window as 'oclTask_3' and constucted as shown in Figure 2.15) and flags up the 'Generation Complete' message.

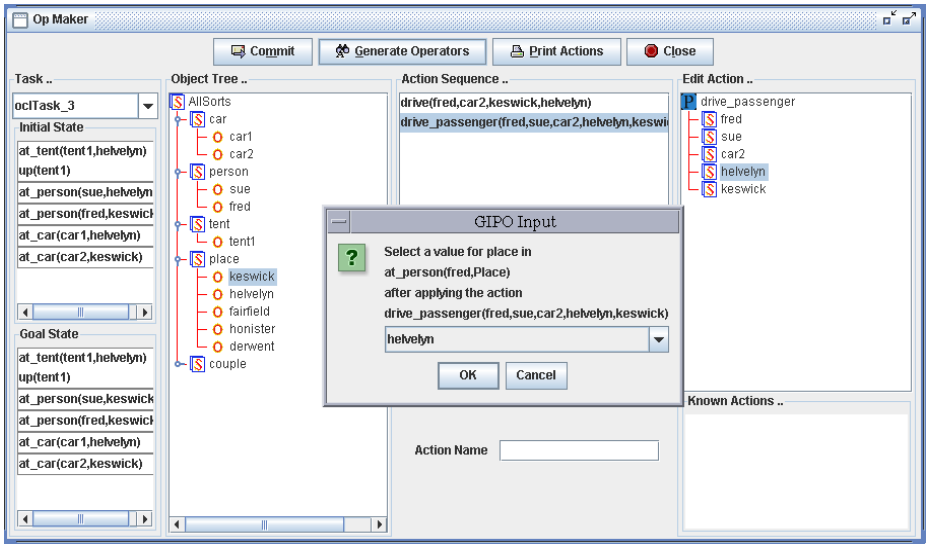


Figure 2.17: Opmaker Consulting the User

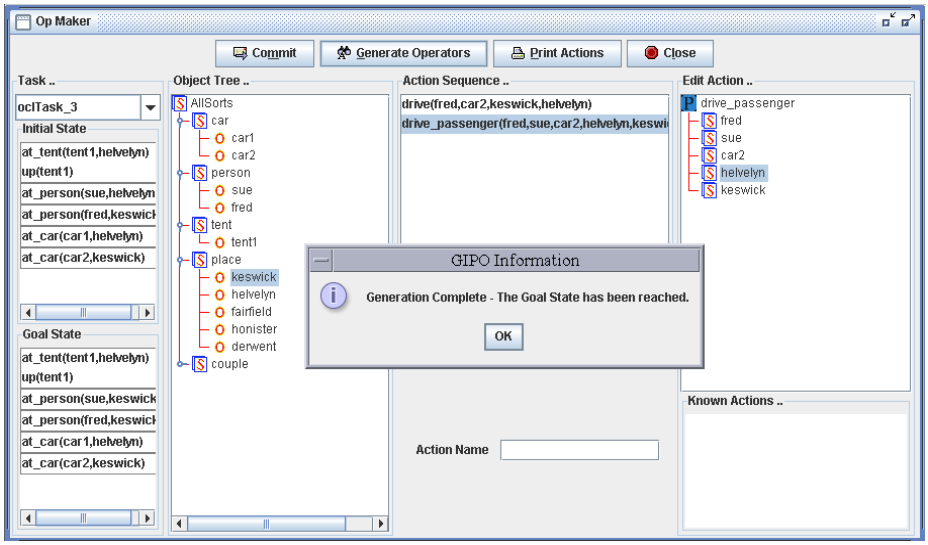


Figure 2.18: Sufficient Operators have been Generated to Complete the Task

Finally, as shown in Figure 2.19, we see that the user has used the 'Commit' button and the new operators have been added to the domain. Their headings are shown in the 'Known Actions' window and they have been generalised so that their reuse is possible in different situations. Figure 2.20 shows GIPO's 'View' option which contains the *OCL* code for the new operators.

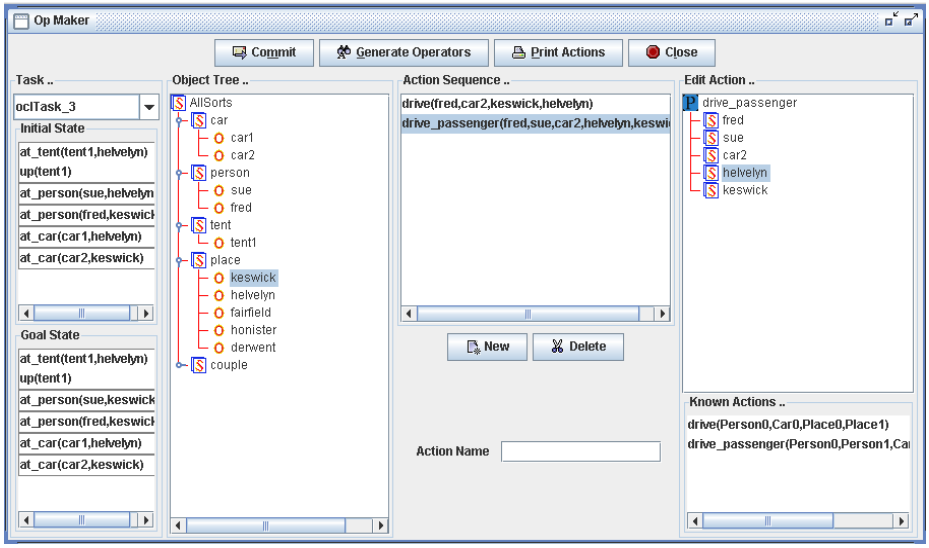


Figure 2.19: Newly Constructed Generalised Operator Headings Shown

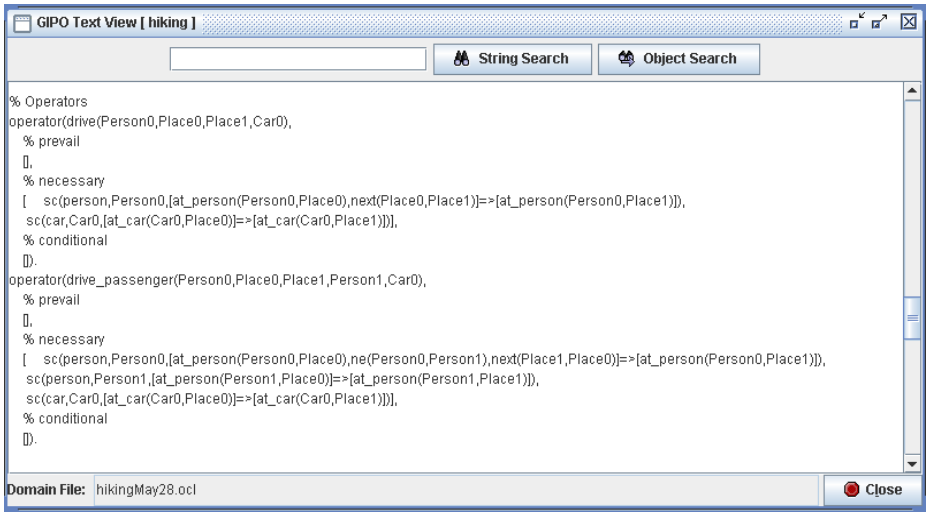


Figure 2.20: GIPO's View Option Showing Newly Formed Operators in OCL

A full version of the hiking domain was constructed using GIPO and in particular the Op Maker tool. The resulting set of operators from the first sequence is shown below, in which 'se' (state expression) denotes a prevail clause and 'sc' (state change) denotes a necessary transition while for a full listing of the hiking domain the reader is referred to Appendix A.

```
% Operators
operator(take_down(Person0,Tent0,Place0),
    % prevail
    [     se(person,Person0,[at_person(Person0,Place0),fit(Person0)])],
    % necessary
    [   sc(tent,Tent0,[at_tent(Tent0,Place0),up(Tent0)]=>
                      [at_tent(Tent0,Place0),down(Tent0)])],
    % conditional
    []).

operator(drive_tent(Person0,Tent0,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [     sc(person,Person0,[at_person(Person0,Place0),fit(Person0),
          next(Place0,Place1)]=>
        [at_person(Person0,Place1),fit(Person0)]),
        sc(tent,Tent0,[at_tent(Tent0,Place0),down(Tent0),
       next(Place0,Place1)]=>
      [at_tent(Tent0,Place1),down(Tent0)]),
       sc(car,Car0,[at_car(Car0,Place0),next(Place0,Place1)]=>
           [at_car(Car0,Place1)])],
    % conditional
    []).

operator(drive(Person0,Place0,Place1,Car0),
```

```
    % prevail
    [],
    % necessary
    [     sc(person,Person0,[at_person(Person0,Place0),fit(Person0),
          next(Place0,Place1)]=>
         [at_person(Person0,Place1),fit(Person0)]),
        sc(car,Car0,[at_car(Car0,Place0),next(Place0,Place1)]=>
            [at_car(Car0,Place1)])],
    % conditional
    []).

operator(put_up(Person0,Tent0,Place0),
    % prevail
    [     se(person,Person0,[at_person(Person0,Place0),fit(Person0)])],
    % necessary
    [     sc(tent,Tent0,[at_tent(Tent0,Place0),down(Tent0)]=>
     [at_tent(Tent0,Place0),up(Tent0)])],
    % conditional
    []).

operator(drive_passenger(Person0,Person1,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [     sc(person,Person0,[at_person(Person0,Place0),fit(Person0),
          ne(Person0,Person1),next(Place1,Place0)]=>
         [at_person(Person0,Place1),fit(Person0)]),
        sc(person,Person1,[at_person(Person1,Place0),fit(Person1),
           next(Place1,Place0)]=>
          [at_person(Person1,Place1),fit(Person1)]),
        sc(car,Car0,[at_car(Car0,Place0),next(Place1,Place0)]=>
          [at_car(Car0,Place1)])],
    % conditional
```

```
    []).

operator(walk_together(Person0,Person1,Tent0,Couple0,Place0,Place1),
    % prevail
    [     se(tent,Tent0,[at_tent(Tent0,Place1),up(Tent0)])],
    % necessary
    [     sc(person,Person0,[at_person(Person0,Place0),fit(Person0),
          ne(Person0,Person1),next(Place0,Place1)]=>
         [at_person(Person0,Place1),tired(Person0)]),
        sc(person,Person1,[at_person(Person1,Place0),fit(Person1),
           next(Place0,Place1)]=>
          [at_person(Person1,Place1),tired(Person1)]),
        sc(couple,Couple0,[walked(Couple0,Place0),next(Place0,Place1)]=>
          [walked(Couple0,Place1)])],
    % conditional
    []).
```

2.5 Conclusion

Having shown the essential parts of domain development in this chapter we will look at operator induction in the next. The reader should be aware that there is more to domain construction than the elements discussed. For example we have not described the use of many of the tools available in GIPO nor have we shown how plans can be executed. Nevertheless the contents of this chapter should be adequate for understanding the issues raised in this thesis and the reader is referred to the GIPO website [25] and the GIPO manual and tutorial for further details of tools not described. The essential point here is that the required parts of the domains needed for induction of operators have been described.

Chapter 3

THE DEVELOPMENT OF *OPMAKER* VERSION ONE

In this chapter we look at the development of *opmaker*, the algorithm for the automatic induction of operators. The first section considers a new experimental domain created to be a useful example to illustrate some aspects of operator induction. This domain is referred to in the early part of the chapter and used extensively in later sections. In the next section we consider the implications of a domain being hierarchical. The third section describes how the original version of *opmaker* worked whilst the fourth discusses further requirements. The fifth section describes the inheritance problem in detail and the work done to correct this problem. In the sixth section we discuss testing criteria and results from the induction process.

3.1 The Briefcase Domains

A good example of an hierarchical domain is the Translog Domain. The main problem with this domain is shown by Figures 2.5 and 2.6 which demonstrate its complicated sort tree and method hierarchy. We felt it would not be easy to follow through examples based on such a complex domain so we started to look for a less complex domain that had the two features of interest that Translog has, namely the hierarchical sort structure and the hierarchical method structure. It proved difficult to find a suitable simple domain so we decided to adapt another well-known domain, the Briefcase Domain.

3.1.1 The Briefcase Domain (BC)

The briefcase (BC) domain is extremely simple, having very few objects in a basic sort structure. This version of the traditional domain allows for conditional clauses in its operators and in BC relevant items are moved between two locations by using a choice of container, such as one of the two bags. In BC the three items, a pay cheque, a dictionary and a business suit, can be transported between home and the office. There are two bags, a briefcase and a suitcase, and whilst all the items can be transported in the suitcase, only the dictionary and the cheque can be carried in the briefcase. There is a simple sort and object structure which is flat [*Definition* 1.17] in the sense that every sort exists at the same level as every other and objects belong to one of the sorts. This sort structure is shown below.

```
% Sorts
sorts(primitive_sorts,[bag,thing,place]).

% Objects
objects(bag,[briefcase,suitcase]).
objects(thing,[cheque,suit,dictionary]).
objects(place,[home,office]).
```

3.1.2 The Hierarchical Briefcase Domain (HBC)

For experimental purposes it was desirable to find a domain with few objects and operators but with a hierarchical structure. To achieve this BC was adapted by adding some objects and structuring the sorts into a hierarchy, enabling the addition of some extra operators and methods and allowing the methods to be structured into a hierarchy [*Definition* 1.18]. So the idea of the Hierarchical Briefcase Domain (HBC) arose. HBC contains the additional items a pencil, some sandwiches, a lunch_box, a pencil_box and containers such as box, bag and carrier. The sort structure for HBC has different tiers or levels. The new sort 'carrier' is one of three top level sorts, whilst 'thing' and 'place' are the others. Carrier is different to the others in the sense that

it is not a primitive sort (one which has no more sorts below it in the tree). To ease future discussion about sort trees we shall refer to a sort at a higher level in the tree than another as a *supersort* whilst a sort at a level below another will be referred to as a *subsort*. The sort structure for HBC is shown below and is diagrammatically represented in Figure 3.1.

```
option(hierarchical).

% Sorts
sorts(primitive_sorts,[briefcase,suitcase,lunch_box,pencil_box,thing,place]).
sorts(carrier,[bag,box]).
sorts(bag,[briefcase,suitcase]).
sorts(box,[lunch_box,pencil_box]).

% Objects
objects(briefcase,[bc1]).
objects(suitcase,[sc1]).
objects(lunch_box,[lb1]).
objects(pencil_box,[pb1]).
objects(thing,[cheque,suit,dictionary,sandwiches,pencil]).
objects(place,[home,office]).
```

As can be seen carrier was the supersort of bag and box, whilst bag and box were, respectively, the supersorts of briefcase and suitcase, and of lunch_box and pencil_box.

With this new sort structure the HBC could represent such ideas as an object being placed in a box which, in turn is placed in a bag. For example a pencil in a pencil_box could be put in a briefcase. This introduced an unfamiliar idea of 'conditional conditionals' in which the pencil in the box moved if the box moved and, if the box was placed in a bag, then the pencil moved if the bag moved.

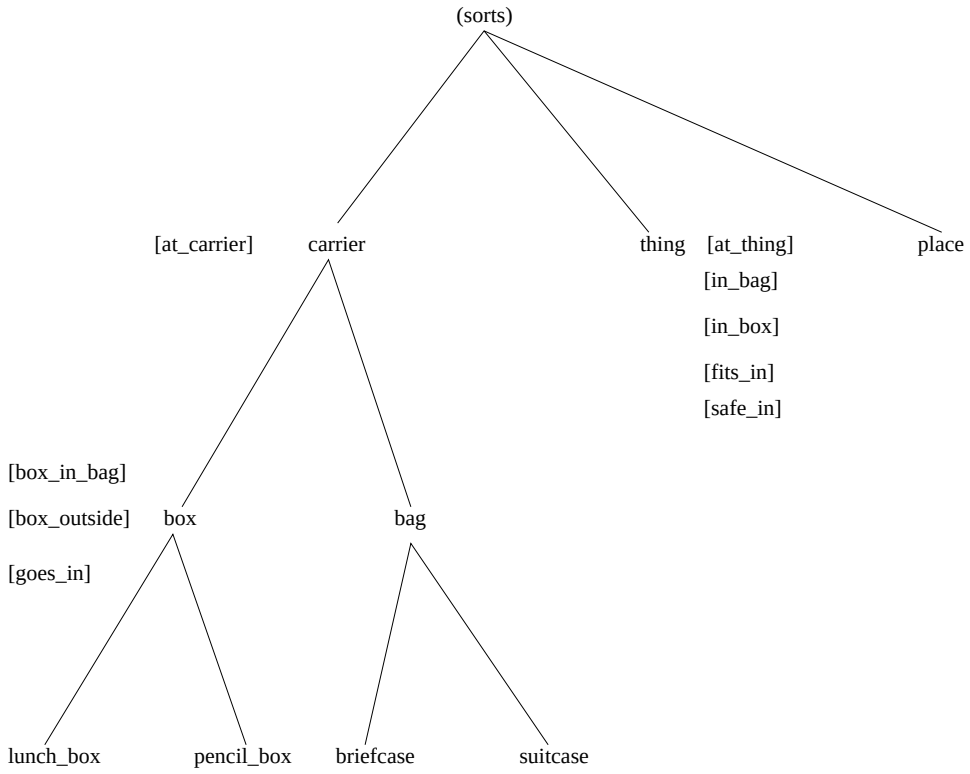


Figure 3.1: The Sort-Tree Showing the Levels at which Predicates Apply in HBC

With HBC there was the concept of packing your lunch ready for the journey to work - a two stage process of putting the sandwiches into the lunch_box and the lunch_box into the briefcase.

The example domain was created using GIPO and declared as option(hierarchical) and is listed in Appendix C. Operators and methods were constructed using GIPO's tools but not using *opmaker* which can only be used if domains are not declared as hierarchical. Tasks had to be constructed before the methods could be built. The process was fairly time consuming but we needed a benchmark against which we could compare any induced methods and operators. Having arrived at this benchmark we would be using only the first few sections of the new domain and a 'stand-alone' version of *opmaker* in order to build operators and methods, but it was important

to have the previously constructed ones for comparison. It is useful to explain here that, whilst the *opmaker* tool embedded in GIPO works only with domains *declared* as non-hierarchical, the stand-alone version does not have that restriction.

3.2 Hierarchical Domains

[See definition 1.12.] For a domain to be hierarchical we must consider two aspects, sort hierarchies and method hierarchies. These can occur in combinations:

1. Flat sort structure and no method hierarchy
2. Flat sort structure but with a method hierarchy
3. Hierarchical sort structure but no method hierarchy
4. Hierarchical sort structure and hierarchical method structure.

Any of combinations 2, 3 or 4 can describe an hierarchical domain. In choosing domains to illustrate points in this chapter we have selected the hiking domain as an example of combination 1, a flat domain, and HBC as an example of combination 4, hierarchical in both respects. Those domains with method hierarchies are built using a bottom up approach. For example Table 3.1 shows a hypothetical situation in which the Hiking Domain has groups of operators built into first level methods. We can imagine the first-level methods shown in the table could be built up into second-level methods. Perhaps one such could be prepare_to_walk, which could be composed of move_camp and collect_partner.

Task	*Possible Method Name*	*Primitive Operators*
1	move_camp (Person,Tent,Car1,Place1,Place2)	take_down(Person,Tent,Place1) load(Person,Tent,Car1,Place1) getin(Person,Car1,Place1) drive_tent(Person,Tent,Car1,Place1,Place2) unload(Person,Car1,Tent,Place2) put_up(Person,Tent,Place2)
2	collect_partner (Person1,Person2,Car)	drive(Person,Car,Place1,Place2) drive_passenger(Person1,Person2, Car,Place2,Place1)
3	collect_car (Person1,Person2,Car1,Car2)	drive_passenger(Person1,Person2, Car1,Place2,Place1) drive(Person1,Car1,Place1,Place2) drive(Person2,Car2,Place1,Place2)

Table 3.1: A Possible Designation of Methods and Operators

Another could be complete_a_leg consisting of methods collect_car, move_camp and collect_partner and operators walk_together and sleep_couple (these last two operators can be found in the full listing of the Hiking Domain in Appendix A).

3.3 Opmaker Phase One

In this section we consider the algorithm for the automatic induction of operators, which was called *opmaker*. This could be used to generate full sets of operators and a single method for domains with no sort or method hierarchy. This had two distinct developmental stages:

- development of the working algorithm for flat domains and integration into GIPO for the building of non-hierarchical domains

- development of the extended version to induce operators and methods for hierarchical domains.

For descriptive purposes this section is split up as follows:

1. Input to *opmaker*
2. Output from *opmaker*
3. The algorithm and how *opmaker* works
4. An example from the Hiking Domain showing operator and method induction
5. Incorporation of *opmaker* into GIPO.

3.3.1 Input

Phase one of the development of *opmaker* required input of the following items for induction to take place.

- A partial domain model without operators. This would be a sequentially constructed domain consisting of sorts, objects, predicates, substates and invariants.
- For some desired goal state, an ordered sequence of action names together with the objects to be manipulated that will achieve that goal state - i.e. a plan trace.
- A set of the initial states for all of the objects mentioned in the action sequence.
- A set of user defined intermediate and goal states which stipulate post-action states for all the objects featured in the initial sequence. These are numbered to match the order of the actions in the initial sequence so that, if some of the

objects have more than one state change, these changes happen in the correct order. This input has become known as 'example material'.

The last two of these are effectively a description of the task to be achieved. If a full domain were the starting point it would include operators and task descriptions. Task descriptions are, for flat domains, initial and goal state descriptions and these can be constructed using GIPO's task editor as we saw in Figure 2.15.

3.3.2 Output

Output from *opmaker* is

- A full set of induced operators - one for every named action.
- A single method operator, i.e. a macro or skeletal plan, which is a combination of all the operators induced. This does not just repeat the initial sequence of actions but contains precondition stipulations, an overall transition for each dynamic changing object (i.e. one that gives initial states on the LHS and final states on the RHS but ignores states in between), a record of any statics involved, an ordering (temporal constraints) of the actions and a listing of the actions, ordered to fit with the temporal constraints. Examples of two methods from the Translog Domain can be viewed in Figures 2.3 and 2.4 whilst a method from the Hiking Domain can be viewed in the results section of Appendix B.

With this first version of *opmaker* this method of operator induction only gave accurate operators for domains with flat sort structures and the reasons for this will be discussed more fully in a later section. Knowledge engineers accept that it is not often possible to identify a 'complete' set of operators for a domain, and we generally choose a set which is comprehensive enough to cover the main tasks planned for the domain. But operators can always be added. Indeed, using *opmaker*, this is no

longer a time consuming exercise. Whilst there is rarely a definitive complete set of operators for a domain, using this system the user could induce a working full set of operators at one go by having a suitable action sequence. This works well if the intention is to have a set of operators quickly but the system also induces a method which, because it consists of all the operators induced ordered into a single method, is probably not the most useful method to induce. The point about methods is that they break the task down into chunks that, hopefully, are logical, stand-alone mini-plans, and by designating a logical order the bigger problem can be solved by calling a selection of methods and operators.

A Significant Finding A better way to use *opmaker* is to compose shorter action sequences with the end method in mind.

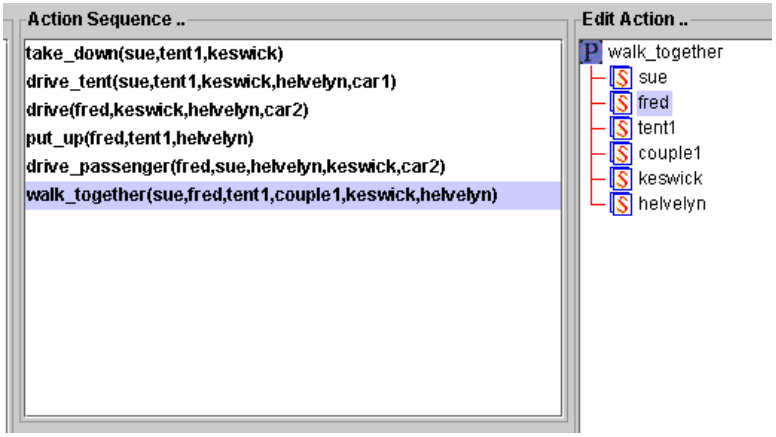


Figure 3.2: An Action Sequence Composed Using GIPO

For Example A leg of the walk in the Hiking Domain is shown in Figure 3.2 and using the sequence specified here six operators will be induced and a single method which does everything except collect the car for the start of the next leg.

But a different way to think of this task is to consider that three compound actions are involved. If, firstly, we call these

```
move_camp, collect_partner
```

and

```
collect_car
```

then, secondly, we decide the steps required for each, we can see that it would be better to use *opmaker* three times to induce, ultimately, the same set of operators but three methods. Table 3.1 shows a similar idea but different operators used to achieve it.

This Has Importance because it changed the way we used *opmaker* to be in line with the way in which humans plan. Humans plan by chunking actions into overall tasks. A system that plans in this way is closer to human intelligence and is more in tune with planning ideas.

It Highlights a Task Modelling Problem because a domain may be more general than its tasks. For example the Hiking Domain could be used to model a camping holiday where any walks taken would be circular and need no car. Cars could still be necessary if larger numbers of people were on holiday, but could be used differently for transportation, trips out and fetching provisions etc. In this case tasks would be modelled differently, ending up with different chunking of the actions.

program *opmaker*(OS: training sequence)

op = operator

RHS = right hand side of transition

LHS = left hand side of transition

In partial domain model

Out parameterised operator descriptions

1. for each *op* in OS do
2. form name and parameter list P;
3. for each dynamic O of sort S in P do
4. get RHS from user input
5. induce necessary substate class LHS
6. form transition $T = (S, O, LHS \Rightarrow RHS)$
7. match free vars in T with those in P
8. end for
9. for all conditional transitions
10. get LHS from user input
11. get RHS from user input
12. form '$\forall\, O \in S, (S, O, LHS \Rightarrow RHS)$'
13. end for
14. end for

procedure match free vars in T with those in P

1. repeat
2. for each parameter X in transition T, $X \neq O$,
3. choose a parameter Y in P to match with
4. X such that $Y \neq O, sort(X) = sort(Y)$,
5. end for
6. until parameter match set is consistent
7. end

Figure 3.3: Outline Design of the *opmaker* Algorithm

3.3.3 What Opmaker Does

We begin with an overview of the *opmaker* system given the input and output detailed in Sections 3.3.1 and 3.3.2.

- For every named operator
  - For every dynamic object
    - A transition is formed taking
      - LHS from the list of initial states or from the RHS of some previous operator
      - RHS from the given example material
  - For static objects (locations)
    - The order listed is important: if there are two
      - The first goes to LHS
      - The second goes to the RHS
  - (*Opmaker*1.1) Inheritance in the sort structure is preserved
- A method is formed from all the operators

In greater detail *Opmaker* works with the objects which are divided into dynamic objects [*Definition* 1.9] and static objects [*Definition* 1.10]. The formal algorithm is given in Figure 3.3 but we see what it does in the case of a flat domain - the Hiking Domain.

The operator heading - line2 of the algorithm Using the sequence of operator names and the object lists supplied by the user *opmaker forms an operator heading* - the first line of the operator is completed.

Formation of prevail and necessary transitions - lines 3 to 8 *Opmaker deals sequentially with the list of objects given.* If an object is dynamic it has sub-states and can change state by the action of some operator. By contrast a

static object has no sub-states. A dynamic object can undergo a transition and in an operator the left hand side of the transition is formed from either the object's initial state, if it has not been used before in the sequence, or the state in which some prior operator in the sequence left it. The right hand side of the transition is formed from the goal state described in the example material (the user defined intermediate and goal states) with the number matching the ordering in the sequence. If the example material contains the word null this indicates to the system that the object does not change state and a prevail clause is formed in the operator being built.

Handling of more than one static objects *If there are static objects in the operator heading* then *opmaker* has heuristics for dealing with them shown in the final procedure. Often they are locations to be used in expressing a move from one location to the other. If this is the case then *opmaker* ensures that a different location goes to the right of the transition than was in the left. The implementation currently in GIPO asks the user to indicate which location precedes the action and which is the consequent location.

If there is only one static object Transitions are *not formed for static objects* so they are never the subject of the transition but represent the attribute values of some of the dynamic objects, so they feature only when it is required to state locations of objects.

3.3.4 An Example from the Hiking Domain

We can now relate these ideas to a particular domain. Clearly there were many test files to be written and it was possible to end up with mistakes in the operators resulting from erroneous input material. Indeed, since the user has choice of operator names, these could be different from the model given previously as we see in the first operator named 'put_down' instead of 'take_down' which appears in Appendix A.

As was noted in the previous chapter, an induced operator takes its name from the user-supplied initial sequence and can, by this means, have a different name but all the same parameters and actions as the original domain model. A typical test file is shown in Appendix B. This is a variant of the domain version considered so far. In this version a tent can be 'loaded' in a car as a third possible state for tent, and Sue and Fred are 'tired' or 'fit' depending on whether they have just walked or slept. We see that the initial user given sequence of actions is:

1. `putdown(tent1,fred,keswick),`
2. `load(fred,tent1,car1,keswick),`
3. `getin(sue,keswick,car1),`
4. `drive(sue,car1,tent1,keswick,helvelyn).`

For the first operator in the sequence the left hand side of the transitions will come from the set of initial states:

```
ss(car,car1,[at(car1,keswick)]),
ss(car,car2,[at(car2,keswick)]),
ss(couple,couple1,[walked(couple1,keswick)]),
ss(person,sue,[fit(sue,keswick)]),
ss(person,fred,[fit(fred,keswick)]),
ss(tent,tent1,[up(tent1,keswick)]).
```

The first operator When composing the first operator *opmaker* uses the first action in the sequence (putdown) to form the heading.

- The initial state of the first dynamic object, 'tent1', is checked - up(tent1, keswick). This state is destined for the LHS of a transition.
- The example inputs are checked relating the correct set of numbered input material to the position in the sequence. These contain intermediate or goal state information and are given by the user. Actual example input is given below.

  ```
  % putdown(tent1,fred,keswick)
  ```

```
input(1,tent1,sclass(Tent,tent,[down(Tent,Place)])).
input(1,fred,null).

% load(fred,tent1,car1,keswick),
input(2,fred,null).
input(2,tent1,sclass(Tent,tent,[loaded(Tent,Car,Place)])).
input(2,car1,null).

% getin(sue,keswick,car1),
input(3,sue,sclass(Person,person,[in(Person,Car,Place)])).
input(3,car1,null).

% drive(sue,car1,tent1,keswick,helvelyn),
input(4,sue,sclass(Person,person,[in(Person,Car,Place)])).
input(4,car1,sclass(Car,car,[at(Car,Place)])).
input(4,tent1,sclass(Tent,tent,[loaded(Tent,Car,Place)])).
```

The input for example 1 shows that the goal state for 'tent1' is down(Tent, Place). The variables Tent and Place are matched to options in the operator heading to form the final goal state down(tent1,keswick).

- *Opmaker* then matches the given initial state for the first object with the goal state from the example material and, for each object, forms a transition which, because the sides are different, is a necessary transition and would form one transition in the third section of the operator.
- The second dynamic object from the first operator heading is treated in a similar manner. Thus 'fred', the second object, is checked against the initial states and is found to be fit(fred,keswick). Then the example material for the first operator is checked. Here we find input(1,fred,null). The word null indicates that fred's state does not change so now a different type of transition is formed. This transition is a prevail transition and is used whenever the state of an object remains the same. It is listed in the

operator's second section.

- Finally the last object for the first operator is checked and found to be a static object. It has already been used to describe the location of the tent and 'fred' and there is no other static object so the operator is complete. (The last section of an operator is earmarked for conditional transitions and there are none for this operator.)

The output for this operator is:

```
operator(putdown(Tent1,Fred,Keswick),
[se(person,Fred,[fit(Fred,Keswick)])],
[sc(tent,Tent1,[up(Tent1,Keswick)] => [down(Tent1,Keswick)])],
[]
).
```

The second operator For the second action in the sequence *opmaker* makes an operator heading of load(fred,tent1,car1,keswick).

- The list of objects involved begins with 'fred'. *Opmaker* recognises that 'fred' has already featured in the sequence so it 'knows' the final state of 'fred' from the last operator that featured him. This state is taken to be the initial state for the LHS of the first transition in the second operator and since the example inputs again have the word null for 'fred' we get a prevail transition for 'fred'.
- For the second object 'tent1' the initial state (LHS) comes from the RHS of the previous operator formed. It is down(Tent1,Keswick). The example material supplies the goal state for the RHS of loaded(Tent,Car,Place). Since this differs from the initial state a necessary transition is formed for 'tent1'.

- One further dynamic object remains - 'car1'. This did not feature in the first operator so the initial state comes from the initial state set. The null in the example material indicates that 'car1' does not change so another prevail clause is added and then the new operator is given.

```
operator(load(Fred,Tent1,Car1,Keswick),
[se(person,Fred,[fit(Fred,Keswick)]),
se(car,Car1,[at(Car1,Keswick)])],
[sc(tent,Tent1,[down(Tent1,Keswick)] => [loaded(Tent1,Car1,Keswick)])],
[]
).
```

The third operator is formed in a similar way. Object 'sue' has not featured in the sequence so her initial state and the LHS of her transition comes from the initial states, the object state from the example material and since they are different a necessary transition is formed. The LHS for 'car1' comes from the RHS of the previous operator and the null in the example material indicates a prevail transition for car.The third operator formed is;

```
operator(getin(Sue,Keswick,Car1),
[se(car,Car1,[at(Car1,Keswick)])],
[sc(person,Sue,[fit(Sue,Keswick)] => [in(Sue,Car1,Keswick)])],
[]
).
```

The fourth operator has a slightly different feature in that its list of parameters includes two static objects, 'keswick' and 'helvelyn'. Following the European convention, these are taken in order using the first for the LHS and the second for the RHS of the transitions for 'sue', 'tent1' and 'car1'. Note that the

partial domain shown in the full listing for this test file, Appendix B, contains a predicate next(place,place) which is used to describe the order of the places to be visited in the walk. It does this by detailing, in the atomic invariants, which specific static object is next to which other as in next(keswick,helvelyn). This is necessary information for an operator and needs to be addressed but static objects do not have transitions of their own. Therefore *opmaker* adds the clause to the LHS of the first transition where both places feature in the parameters. The final operator is:

```
operator(drive(Sue,Car1,Tent1,Keswick,Helvelyn),
[],
[sc(person,Sue,[in(Sue,Car1,Keswick),next(Keswick,Helvelyn)] =>
[in(Sue,Car1,Helvelyn)]),
sc(car,Car1,[at(Car1,Keswick)] => [at(Car1,Helvelyn)]),
sc(tent,Tent1,[loaded(Tent1,Car1,Keswick)] => [loaded(Tent1,Car1,Helvelyn)])],
[]
).
```

3.3.5 Induction of Methods

When *opmaker* reaches the end of the sequence of actions it finally gives a single method which is composed from the individual operators induced. The method has the following features:

- A method name and set of parameters which features just the dynamic objects affected by the method
- A set of dynamic constraints found from the initial states - e.g. 'fred' has to be fit and in the right place which must be the same as the tent
- A list of necessary transitions that take place between the start of the first action and the end of the last action. These do not necessarily reflect individual

necessary transitions in the induced operators but do reflect the overall transition for each object. For example in the operator sequence 'sue' has three states fit(Sue,Keswick) → in(Sue,Car1,Keswick) → in(Sue,Car1,Helvelyn) but in the method only fit(Sue,Keswick) → in(Sue,Car1,Helvelyn) appears

- A list of any static constraints
- A decomposition of the tasks to be undertaken which is a list of the operator names to be called. (If this were a higher level method then you could expect to see this list contain the names of other methods as well as operators.)
- A list of temporal constraints which stipulates the order in which the decomposition is applied taking the listed order of the operators as being numbered (in this case 1 - 4). Effectively this gives us the equivalent of a hierarchical *macro* (a macro that could also contain other macros).

The method induced is given below.

```
method(move_tent(Fred,Sue,Car1,Tent1),
% dynamic constraints
[se(person,Fred,[fit(Fred,Keswick)]),
se(person,Fred,[fit(Fred,Keswick)])],
% list of necessary transitions
[sc(person,Sue,[fit(Sue,Keswick)] => [in(Sue,Car1,Helvelyn)]),
sc(car,Car1,[at(Car1,Keswick)] => [at(Car1,Helvelyn)]),
sc(tent,Tent1,[up(Tent1,Keswick)] => [loaded(Tent1,Car1,Helvelyn)])],
% static constraints
[  next(Keswick,Helvelyn)],
% temporal  constraints
[before(1,2),before(2,3),before(3,4)],
%  decomposition
[  putdown(Tent1,Fred,Keswick),
```

```
  load(Fred,Tent1,Car1,Keswick),
  getin(Sue,Keswick,Car1),
  drive(Sue,Car1,Tent1,Keswick,Helvelyn)]
).
```

3.3.6 Incorporation of Opmaker into GIPO

The original version of *opmaker* discussed in the previous sections has now become one of many tools available in GIPO. Details of this can be found in Section 2.4.6.

This is not the whole story, however, because planning problems are becoming more challenging and are applied to a range of real world problems. Therefore *there is a need to consider how humans do planning by chunking a task into subtasks then planning these at a more basic level.* There is a good argument for replicating this idea in AI planning and to do this we need to use hierarchical planning. Hierarchical planning has been seen as a big challenge for the knowledge engineer because of the difficulty of constructing methods and method networks. We turn now, in the next section, to hierarchical planning, and examine how our later version of *opmaker*, *opmaker*2, will help to ease the burden on knowledge engineers.

3.4 Opmaker - Further Requirements for Hierarchical Domains

Version one of *opmaker* did not go far enough in addressing all the issues. For example, it would generate operators but the method it was capable of producing was dependent on the type of domain under construction. We saw the use of GIPO for domain construction and in Section 2.4.1 we saw how the sort structure for the hiking domain was constructed by hand and by using GIPO's sorts editor. We saw that, by comparison to the sort structure for the Translog Domain shown in Figure 2.5, the structure of the Hiking Domain was just a flat structure. Branching of the sort structure for the Translog Domain has a depth of 7. Using GIPO it is easy enough to construct such a sort tree but having predicates which apply at different

levels in the tree adds an extra problem for *opmaker* which will be shown in Section 3.5.3.

3.5 The Problem of Inheritance

In planning, where domains have an hierarchical sort structure, inheritance generally operates.

3.5.1 What Is Inheritance?

In Figure 3.1 we can see the sort tree for HBC. The figure shows that there are predicates associated with different levels of the tree. For example, 'carrier' is one of the three top level sorts and has the predicate at_carrier(carrier,place) associated with it. Inheritance works in the sort tree so that any subsort of carrier also inherits the at_carrier predicate. Further down the sort tree there are three predicates associated with box, one of which is goes_in(box,carrier). This predicate is inherited by lunch_box and pencil_box, but inheritance does not work up the tree, or across it, so carrier and bag do not inherit this predicate.

In the following three sections we detail

- exactly what the problem was
- how the inheritance problem was recognised
- what was done to correct the problem.

3.5.2 The Inheritance Problem

In order to identify objects that could be used in transitions, *opmaker* used the partial domain model to identify a list of the dynamic sorts featured in the domain. If a predicate such as at_carrier(carrier,place) was to feature in a transition, then any

object which inherited this predicate was a candidate for the list, but *opmaker* only delivered the dynamic objects that it found at the same level in the sort tree that the predicate applied to. This meant that transitions for objects like briefcase or box could not be formed correctly. Thus it was a fundamental problem for any domain with an hierarchical sort structure.

3.5.3 Finding the Inheritance Problem

We found that inheritance was causing a problem for opmaker because use of domains with a sort structure did not always give a correct set of operators and often the system hung or did not produce a method. An example of faulty output from HBC is given below.

For the following sequence of actions

```
put_thing_in_bag(sc1,home,suit),
move(sc1,home,office,suit),
take_out(suit,sc1,office)
```

the following output of operators and methods was produced.

```
operator(put_thing_in_bag(Sc1,Home,Suit),
[],
[sc(thing,Suit,[outside(Suit),at_thing(Suit,Home),fits_in(Suit,Sc1)] =>
[in_bag(Suit,Sc1),at_thing(Suit,Home)])],
[]
).


operator(move(Sc1,Home,Office,Suit),
[],
[sc(thing,Suit,[in_bag(Suit,Sc1),at_thing(Suit,Home),fits_in(Suit,Sc1)] =>
[in_bag(Suit,Sc1),at_thing(Suit,Office)])],
[]
).
```

```
operator(take_out(Suit,Sc1,Office),
[],
[sc(thing,Suit,[in_bag(Suit,Sc1),at_thing(Suit,Office),fits_in(Suit,Sc1)] =>
[outside(Suit),at_thing(Suit,Office)])],
[]
).

% name
method(take_suit_to_work(Suit),
% dynamic constraints
[],
% list of necessary transitions
[sc(thing,Suit,[outside(Suit),at_thing(Suit,Home)] =>
[outside(Suit),at_thing(Suit,Office)])],
% static constraints
[  fits_in(Suit,Sc1),
],
% temporal  constraints
[before(1,2),before(2,3)],
%  decomposition
[  put_thing_in_bag(Sc1,Home,Suit),
  move(Sc1,Home,Office,Suit),
  take_out(Suit,Sc1,Office),
]
).
```

We can see that the first operator has an accurate transition for the suit and the atomic invariant fits_in(Suit,Sc1) is also correctly represented. However there should be a prevail transition for the suitcase, sc1, which logically should remain at home whilst the suit is packed. Example material for this transition was

```
% put_thing_in_bag(sc1,home,suit)
input(1,sc1,null).
input(1,suit,sclass(Thing,thing,[in_bag(Thing,Bag),at_thing(Thing,Home)])).
```

and at_carrier(sc1,home) had been included in the initial states but no prevail transition had been formed. Examining the two other operators showed a similar problem - there were no transitions for the suitcase, sc1. The method produced carried the same problem and in fact the suitcase did not appear to be necessary to the transportation of the suit to the office! However, the decomposition of the method made it appear that the suitcase *did* feature so this anomaly was a problem.

After producing several test files and studying the results it seemed that the problem arose as a result of the sort structure which related the predicate

```
at_carrier(carrier,place)
```

to the level of the sort tree *carrier* and we were using it at the level *suitcase*, two levels lower on the tree. After more checking with other domains with sort structures this turned out to be a universal problem and we were able to identify that inheritance was not being preserved by *opmaker*.

To make description of the *opmaker* process easier in terms of versions, it now becomes necessary to define these. In future discussion the version we have discussed so far will be referred to as *opmaker*1.0, whilst the version updated to compensate for the inheritance problem will be *opmaker*1.1. The version in the next chapter will then become *opmaker*2.0.

Because only objects of dynamic sorts can be the subject of transitions (see Definition 1.9), when *opmaker*1.0 makes transitions it checks that each parameter is a dynamic sort (with a listed substate or substate set). Looking at the listing for the substate classes below

```
substate_classes([

carrier(C,
[
  [at_carrier(C,L)]
```

```
]),

thing(T,
[
  [outside(T),at_thing(T,L)],
  [in_bag(T,Bag),at_thing(T,L)],
  [in_box(T,Box),at_thing(T,L)]
]),

box(Box,
[
  [box_in_bag(Box,Bag)],
  [box_outside(Box)]
])
]).
```

we can see that, from this, *opmaker*1.0 can make the following list of dynamic sorts: [carrier, thing, box].

However, looking at Figure 3.1 and referring to the HBC code listed in Appendix C, will show that other dynamic sorts can be found which are subsorts of these items. We would thus gain lunch_box, pencil_box, suitcase, briefcase and bag, all of which inherit a substate from their supersort. The problem was that *opmaker*1.0 was missing these extra sorts.

3.5.4 Rectifying the Problem

A revised implementation was written according to the outline algorithm shown in Figure 3.4. In outline lines 1-4 isolate and list the dynamic sorts from the substate class list. At this stage these are [carrier, thing, box]. Lines 5-8 iterate through this list and find all the subsorts of each. In the case of 'carrier' these are [box, bag, suitcase, briefcase, lunch_box, pencil_box]. For 'thing' there are no subsorts and

program *get_dynamic_sorts*(X, Y)
In the single substate class list ($SSCL = X$)
Out a full list of dynamic sorts ($DSL = Y$)
1.for each *sort* in *SSCL* do
2. extract the sort name S
3. add S to sort list *DSL*
4. end for
5. for each sort S in *DSL* do
6. find all S's subsorts (*Subs*)
7. add *Subs* to *DSL*
8. end for
9. remove duplicates from *DSL*
10. end

Figure 3.4: Outline Design to Obtain all the Dynamic Sorts from a Hierarchy

those of 'box' are [lunch_box, pencil_box]. These lists are then combined with the original giving a list with duplicates which are removed by line 9.

This leaves the full list of dynamic sorts which is [carrier, bag, briefcase, suitcase, box, pencil_box, lunch_box, thing].

*Opmaker*1.1 would now be able to resolve the problem of inheritance since sorts below a node on the sort tree were identified as inheriting the predicate attached to the node of the supersort.

Returning to the initial problem example, which was the lack of a prevail transition for suitcase shown in the first faulty operator, we can see that suitcase would now be identified as a dynamic sort, inheritance of the at_carrier predicate would be assured and the operator would correctly have a prevail transition for suitcase.

3.6 ***Testing and Results from*** *Opmaker*1.1

Testing of *opmaker*1.1 was done on a range of domains. The new implementation was also tested on various random sort trees created using GIPO.

3.6.1 Success Criteria

Successful testing would be judged on:-

1. Accurate identification of full dynamic sorts lists from varied starting levels in the sort trees.

2. Use with domains without a hierarchical sort structure should give the same results as with *opmaker*1.0.

3. Use with the Translog Domain should give accurate operators.

4. Use with other domains should give either improved performance or the same accurate results as with use of *opmaker*1.0.

3.6.2 Results Measured Against these Criteria

1. All sort trees tested gave accurate sort lists of the subsorts for a named sort.

2. Non-hierarchical domains tested gave the same results as previously obtained using *opmaker*1.0.

3. The performance of *opmaker*1.1 with the Translog Domain gave a significant improvement in operator induction across a range of test files. Not only were operators now accurate but methods could now be induced on this domain comparable to hand-crafted ones.

4. For every domain tested *opmaker*1.1 gave the same results as *opmaker*1.0 where the domain had a flat sort structure, and improved accurate operators and methods where the domain had an hierarchical sort structure.

3.6.3 Testing and Results Using HBC

Input to *opmaker*1.1 consisted of test files of which one typical test file for HBC using *opmaker*1.1 is shown in Appendix D. Output is listed at the end of the appendix and shows that the transitions for the operators are accurate.

One negative finding was that conditional operators were not correctly induced. We decided this was due to the use of a double conditional domain (a pencil in a box, and the box in a bag means the pencil is doubly conditional on the box and the bag). This is the only example of this type of domain in *OCL* and we conjectured that a bug in *opmaker*1.1 (and *opmaker*1.0) is responsible. When this problem was identified we re-wrote the initial sequences for induction. They now included all previously conditional objects in the action parameter lists. We renamed 'move' actions according to what was moved, e.g. move_bag, move_box_in_bag, move_pencil_in_box_in_bag etc. and were then able to induce operators.

Building Methods using Induction

In order to induce methods we considered which actions were likely to be needed to be repeated most often. We argued that whilst all the objects would need to be transported regularly, taking lunch to work might be regarded as the most frequent of all actions. We could build each method using different initial sequences like the one in the test file shown in Appendix D. Using *opmaker*1.1 we induced the following methods and operators.

- the method pack_lunch(sandwiches,home,lunch_box,briefcase), which comprised the operators
  put_in_box(sandwiches,home,lunch_box) and
  put_box_in_bag(lunch_box,briefcase,home)

- the operator move(sandwiches,lunch_box,briefcase,home,office)

- the method unpack_lunch(sandwiches,office,lunch_box,briefcase), which comprised the operators
  take_out_box(lunch_box,briefcase,sandwiches,office) and
  empty_box(sandwiches,lunch_box,office)

- the method take_lunch_to_work(sandwiches,lunch_box,home,office,briefcase), which comprised the following
  - the method pack_lunch(sandwiches,home,lunch_box,briefcase)
  - the operator move(sandwiches,lunch_box,briefcase,home,office)
  - the method unpack_lunch(sandwiches,office,lunch_box,briefcase).

As can be seen above, the final method in this group could not be constructed until the other methods and operator existed but, since this final method used the others, it was at a higher level in the hierarchy of methods.

There were several other possibilities for making the method hierarchy more complex. One such is packing the suit into the suitcase and taking it to work at the same time as the briefcase containing the lunch. This would allow for a method such as take_lunch_and_suit_to_work at a third level in the method hierarchy. (A theoretical fouth level would be to put both the suitcase and the briefcase in the car and driving to work.)

The Important points Were...

1. **The hierarchy is built bottom up.** *Opmaker*1.1 uses operators to build lowest level methods and higher levels are built from the operators and methods already induced.

2. **Careful thought needs to be exercised** in the construction of a sensible method hierarchy. Methods should not duplicate part of the function of other methods.

3. **Using induction to build methods** makes their construction easier and quicker.

4. **Time is saved** building a method hierarchy.

3.6.4 Further Experimentation and findings

We built a version of HBC using just induced methods and operators and were able to use it with HyHTN [44], the hierarchical planner available in GIPO. We devised tasks similar to those in the GIPO built domain in Appendix C. A snapshot of the goal of the task devised to replicate the method take_lunch_to_work(sandwiches, lunch_box, home, office, briefcase) is shown in Figure 3.5. In this the left hand box shows the representation of the task. In GIPO the methods are shown in lilac whilst the operators are shown in green. The sequence of their application is demonstrated by the arrows and the right hand boxes list the available operators and methods - i.e. those already constructed.

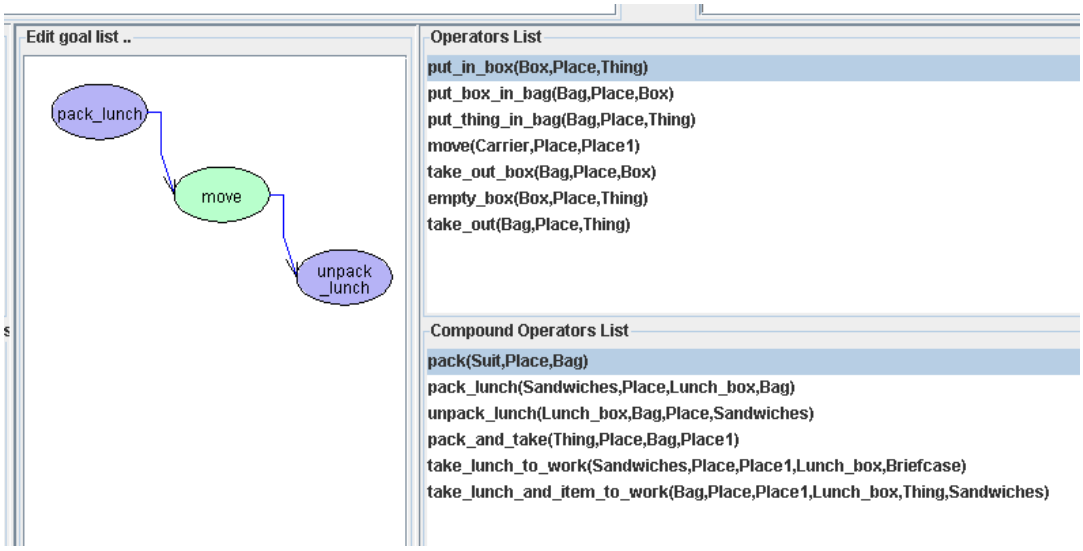


Figure 3.5: The Task Goal Construction Window in GIPO Showing the take_lunch_to_work Method under Construction

We used the planner to solve the tasks in two ways either by using just the induced operators or by using the induced methods as well as the operators. We found that the time taken by the planner to solve one task using just operators was shorter, at 0.36 seconds, than the time taken using methods, at 0.86 seconds. Figure 3.6 shows the solution and part of the rationale to the task shown in Figure 3.5. In it we can see a list of the operators used to reach the solution. Other tests produced similar results showing that the overall time taken using the hierarchical planner to solve tasks was longer if methods were used.

Figure 3.6: The Planner Window in GIPO Showing the Solution to the Task in Figure 3.5

These results were disappointing because we had expected to confirm that hierarchical planning was more efficient. However we realised that HBC was a very simple domain for which, normally, researchers would not use hierarchical planning. Hierarchical operators can be regarded as ready-to-use plan chunks which save the time taken to search for a sub-plan and thus save on overall planning time. On the other hand, before *opmaker*1.0, the time taken to construct methods would have to be taken into account and for simple domains would not be cost-effective. In the case of HBC, which has only a few operators, search times were unlikely to be very long. We had constructed a more complex domain in terms of operators and methods

than was required and it seemed that the complexity of the domain was slowing the planning time down. The same planner, HyHTN [44], was used for plans using just operators and plans using methods and the extra time taken was still under 1 second so there were arguments in favour of continuing the work. Firstly, hierarchical planning is more like the way humans plan by chunking tasks to be performed and AI aims to emulate human thinking using computational methods. Secondly, an *OCL*-expressed domain had a richer language structure and the methods encapsulated the plan chunking effectively. Thirdly, we wanted to continue but to use a more complex domain with more objects and operations to perform to see if we found the same slower plan times. We hoped there would be a crossover point where the number of searches needed using just operators would slow down the plan time more than the extra time required by using methods.

The following chapter describes the further work done in this area and demonstrates more promising results. In particular we introduce *opmaker*2, an improved system of induction which needs no intermediate state information.

Chapter 4

THE DEVELOPMENT OF INDUCTION TOOLS WITHOUT INTERMEDIATE STATE USER INPUT

In this chapter we investigate techniques for inducing operators without the requirement of hand-crafted intermediate and goal state descriptions. We are working towards the goal of automated planning in which an agent (system) has full or partial domain knowledge but is able to use the level of knowledge it has to infer and induce the rest.

To aid description we need to introduce another domain which was extended from an existing version to provide more of a challenge to the hierarchical planner. In the first section we describe the alterations made to the well-known tyre world [65] to create the Extended Tyre Domain. In the second section we see how experiments with this new domain compare to those on HBC. In the third section we introduce ideas for improving operator induction by automating the process of obtaining the example input sets giving goal states for operator transitions. The fourth section describes the implementation of the new process with a diagram and gives an algorithm for this, along with a walk-through description of the stages involved in induction without hand-crafted example material. Finally, the fifth section discusses the aims for experimentation with the process output and success criteria. Results are given for three different domains.

4.1 The Extended Tyre Domain

As we noted in the last chapter, HBC was not challenging to the planner, though it had helped with the testing of the inheritance problem. With its 11 objects, 9 sorts and 7 operators the conditional version of HBC was very primitive, but was effective for testing purposes once the inheritance problem had been resolved. One further difficulty with this domain, as we briefly noted, was that conditional operators were not well represented appearing in the necessary clauses of the operators instead of the conditional clauses. We conjectured that, since a version that contained no conditional operators and consequently contained more operators was still slower when planning using methods than when using operators, we should try a more challenging domain.

We aimed to find a domain with more objects and more operators. We decided to use a domain with a flat sort structure since we knew how to solve the inheritance problem. We wanted to use *opmaker*1.1 to induce all its operators and methods. A domain with a lot more objects was difficult to find because many domains for experimental purposes produce quick results from few objects and operators. *One of our main aims was to show that for longer plans our induced methods would give faster plan times than by just using operators.* After all if we could decrease the time taken for domain construction and, at the same time, reduce the time taken to arrive at accurate plans we would have made a significant contribution to planning. We considered the traditional tyre domain which did have more operators. We decided to extend this domain to make it more of a challenge to the planner by maximising the number of operators and objects it contained. In this section we describe the alterations to this flat domain to give it a full set of induced operators and a selection of useful methods.

4.1.1 The Original Tyre Domain

The original tyre domain was built to model the steps required in the changing of a wheel on a single wheel hub. Sorts and objects in the domain were those necessary to the wheel change but limited to two wheels and one each of other objects except the wheel nuts which are not numbered but simply declared as 'nuts', and thus effectively a single object.

The Sorts and Objects in the Original Tyre Domain

The flat sort and object declarations are shown below.

```
domain_name(tyre).

% Sorts
sorts(primitive_sorts,[container,nuts,hub,pump,wheel,wrench,jack]).

% Objects
objects(container,[boot]).
objects(nuts,[nuts_1]).
objects(hub,[hub0]).
objects(pump,[pump0]).
objects(wheel,[wheel1,wheel2]).
objects(wrench,[wrench0]).
objects(jack,[jack0]).
```

In total the original Tyre Domain contained 8 objects and 7 primitive sorts which made it less complex than HBC in this respect.

Operators in the Original Tyre Domain

There were more operators in the Tyre World, so in this sense it was more challenging to a planner than HBC. Also these operators had an obvious sequence which allow

the reader to discern the intentions of the full plan of changing the wheel. We shall list here just the names of these operators.

```
open_container(Boot)
fetch_jack(Boot,Jack)
fetch_wheel(Boot,Wheel2)
fetch_wrench(Boot,Wrench)
fetch_pump(Boot,Pump)
loosen(Wrench,Hub,Nuts)
jack_up(Hub,Jack)
undo(Wrench,Hub,Jack,Nuts)
remove_wheel(Wheel1,Hub,Jack)
put_on_wheel(Wheel2,Hub,Jack)
do_up(Wrench,Hub,Jack,Nuts)
jack_down(Hub,Jack)
tighten(Wrench,Hub,Nuts)
putaway_wheel(Boot,Wheel1)
putaway_wrench(Boot,Wrench)
putaway_jack(Boot,Jack)
putaway_pump(Boot,Pump)
close_container(Boot)
```

One item, the pump, was not actually used by any operators other than the two responsible for getting it out and putting it away. Whilst this was a comprehensive set of operators, the few objects in the domain did not allow for a full model unless it was of a unicycle with a boot.

4.1.2 The Tyre Domain Extension

The decision was taken to extend this domain so that it modelled the numbers of objects used in changing the wheel of the average family saloon car. Thereafter, if required, the domain could be extended further to cover a large commercial vehicle. We assumed the average family saloon car would have four wheels and a spare, with

normal tyres before the invention of run-flat tyres. With this model we assumed a fairly down-market vehicle without alloy wheels but with wheel trims which could be applied and removed. This gave us two extra operators. Also it now had tyres on the wheels which could be 'full' if they were fully inflated with air, 'flat' if they were discovered to be flat but could be fully inflated, or 'punctured' if use of the pump was ineffective. These alterations gave a use for the pump which had to be used when checking for punctures and created two further operators, inflate_tyre and discover_puncture.

The most notable thing about this extension was the increase in the number of objects - an increase over the original domain of 18. There were two additional sorts and 4 additional operators in total. A larger number of objects would increase the search time quite significantly when instantiating operators to find a correct plan. The extended object and sort structure is shown below, and a comparison of the objects and operators in the two versions of the Tyre Domain can be seen in Table 4.1.

```
domain_name(tyre_extended).

option(hierarchical).

% Sorts
sorts(primitive_sorts,[container,nuts,hub,pump,wheel,wrench,jack,wheel_trim,tyre
]).

% Objects
objects(container,[boot]).
objects(nuts,[nuts1,nuts2,nuts3,nuts4]).
objects(hub,[hub1,hub2,hub3,hub4]).
objects(pump,[pump0]).
objects(wheel,[wheel1,wheel2,wheel3,wheel4,wheel5]).
```

Tyre Domain	*Extended Tyre Domain*
Objects	*Objects*
1 boot	1 boot
1 set of wheel nuts	4 sets of wheel nuts
1 hub	4 hubs
1 pump	1 pump
2 wheels	5 wheels
1 wheel wrench	1 wheel wrench
1 jack	1 jack
0 wheel trims	4 wheel trims
0 tyres	5 tyres
Total Objects	*Total Objects*
8	26
Total Operators	*Total Operators*
18	22

Table 4.1: Comparison of Two Versions of the Tyre Domain

```
objects(wrench,[wrench0]).
objects(jack,[jack0]).
objects(wheel_trim,[trim1,trim2,trim3,trim4]).
objects(tyre,[tyre1,tyre2,tyre3,tyre4,tyre5]).
```

The additional operators were:

```
apply_trim(Hub,Wheel_trim,Wheel)
remove_trim(Hub,Wheel_trim,Wheel)
inflate_tyre(Pump,Tyre)
find_puncture(Pump,Tyre)
```

The complete version of the Extended Tyre Domain which was extended using

GIPO can be viewed in Appendix E.

4.1.3 Substates in the Extended Tyre Domain

Later sections of this chapter will show that it would be important to consider the substates declared in the partial domain input to *opmaker* much more closely that previously. The substates direct what the possible transitions for the objects might be. For example, looking at the substates for jack,

```
substate_classes(jack,J,[
    [have_jack(J)],
    [jack_in_use(J,H)],
    [jack_in(J,C)]]).
```

if we are given an initial state for the object jack as have_jack(Jack) then if the jack undergoes a transition we can see what the possible transitions might be:

- have_jack(Jack) → have_jack(Jack) (a prevail transition)
- have_jack(Jack) → jack_in_use(Jack,Hub) (one possible necessary transition)
- have_jack(Jack) → jack_in(Jack,Boot) (another possible necessary transition).

The substate classes for the Extended Tyre Domain contain a lot of possible states for the objects and are themselves another way in which this domain is more complex than HBC where the substates appear shorter because of inheritance and because there are few substate choices. Further discussion on substate classes can be found in Section 4.3.3.

4.2 Experiments with the More Complex Domain

We built the new version of the Extended Tyre Domain (ETD) using GIPO. The complete version of ETD included tasks and all the operators and methods which we

aimed to replicate using *opmaker*1.1 induction. This domain is included in Appendix E. This example ETD was validated by GIPO and tested by using the planner to find solutions to the tasks. This step was necessary so that we had an example correct domain to which we could compare induced operators and methods.

4.2.1 Aims of Experimentation

As with previous domains we then took the partial domain (without the operators and methods) and added the initial sequence, initial states and example intermediate states, having separate test files for each desired method. The aims in experimentation with the new domain were to:-

1. Build a full and accurate set of operators

2. Compare these operators to the ones constructed using GIPO, which had been validated by the system

3. Use logical short sequences of actions to induce meaningful methods which accurately described a complete task in the tyre change process

4. Add the induced methods and operators to the partial domain

5. Construct tasks that would test all of the operators and methods forming increasing lengths of plans

6. Use the new version for planning where only the *induced operators* were available for planning

7. Use the new version for planning using *operators and methods*

8. Compare the times taken for 6 and 7 above and compare both against times taken for hand-crafted operators and methods

4.2.2 The Full Planning Problem

At this stage we define the full planning problem for ETD. This is modified from the full problem for the original domain to take account of the extra objects and operators available. It is defined as:-

The vehicle is found to have two flat tyres. The motorist must open the boot and using the pump discover if either of the tyres is simply flat. One tyre is flat and is inflated using the pump, but the other is found to be punctured. The motorist must then remove the wheel trim from the punctured wheel, use the wrench (fetched from the boot) to loosen the nuts and the jack to jack up the hub. This done he removes the nuts, exchanges the wheel for the spare and does up the nuts. After jacking down he tightens the nuts, re-applies the trim and puts all items away in the boot before closing it.

4.2.3 Decisions on the Potential Methods

We needed to give consideration to the chunks of the full planning problem of changing the wheel from start to finish. This follows on from the argument in Section 3.6 where we state that it is at this stage that careful thought has to be given to the method structure. Since induction of methods has eased their individual construction it is better to be aware of what appears to be a sensible choice for the methods. We decided on a sensible choice of methods for ETD and these are shown in Figure 4.1.

Each of the desired methods shown in Figure 4.1 was made into a test file to be used with *opmaker*1.1. A test file for the method discover_puncture is shown in Appendix F. Results from running this file can be seen at the end of the test file.

1. A method called fix_flat in which, on spotting a flat tyre, the motorist opens the boot, takes out the pump, uses it to inflate the tyre, puts away the pump and closes the boot.

2. A method called discover_puncture which is similar to the first. Here the motorist opens the boot, takes out the pump, finds he can't inflate the tyre (so discovers he has a puncture) and puts away the pump, leaving the boot open because he knows he will need to get out more equipment.

3. A method called fetch_tools in which the motorist fetches the jack and the wrench from the open boot.

4. A method called putaway_tools in which the jack and wrench are returned to the boot.

5. A method called unfasten_hub in which the wheel trim is removed, the wrench is used to loosen the nuts, the jack is used to jack up the hub and the nuts are undone and removed.

6. A method called fasten_hub which reverses the process in point 5 above. Here the nuts are put on and done up, the hub is jacked down, the nuts are tightened and the wheeltrim is replaced.

7. A method called change_wheel in which the spare wheel with its tyre is fetched from the boot, the punctured tyre is lifted from the hub and put into the boot and the spare is positioned on the hub.

Figure 4.1: A Sensible Choice of Methods for the Extended Tyre Domain

Starting with a complete domain without operators, each of the desired method files was run with *opmaker*1.1 and the resulting new operators and method were added to the new version of the domain. Eventually all the operators that were in the GIPO-constructed domain were replicated in the new version of the domain, which then contained the seven methods suggested in Figure 4.1.

4.2.4 Results of the Testing

The results from the eight aims of the testing are listed below:-

1. All the operators were built successfully

2. All operators were similar to the example domain

3. Meaningful methods did describe chunks of the full tyre change problem

4. Induced operators and methods were added to a partial domain to be later tested further

5. The previously devised tasks were copied into the new version to complete the full induced version of ETD

6. The full induced version was used for planning when the methods were unavailable and plan times were noted

7. The full induced version was used for planning when the methods and operators were available and again plan times were noted

8. The times taken in item 6 and 7 were compared. Results of this comparison can be seen in Table 4.2 in which task 11 is the full planning problem.

Task No.	*Steps in Plan*	*Plan Correct?*	*Time for Operators*	*Time for Methods*
1	8	Yes	1.01 secs	1.09 secs
2	7	Yes	1.43 secs	2.25 secs
3	6	Yes	0.85 secs	1.25 secs
4	1	Yes	0.00 secs	0.01 secs
5	3	Yes	0.01 secs	0.01 secs
6	3	Yes	0.02 secs	0.02 secs
7	3	Yes	0.01 secs	0.05 secs
8	3	Yes	0.01 secs	0.07 secs
9	4	Yes	0.15 secs	0.24 secs
10	16	Yes	2.67 secs	4.53 secs
11	24	Yes for methods	no plan found	0.2 secs

Table 4.2: Comparison of Plan Times Using Operators and Methods

9. Improved planning efficiency using methods is demonstrated in Table 4.3. It shows that replacing operators by methods as the plan steps increase keeps plan times at around 0.02 seconds. Table 4.2 shows that plan times for just operators increase and as the steps increase, and some plans cannot be solved with operators alone.

Tasks for Testing

Tasks were devised to test that each section was working correctly and these were constructed in two ways. The first way was to assume the domain was flat and only allow the use of operators in the task. The second was to assume the domain was hierarchical and allow the use of the method. Each of these was run with the planner. The resulting times taken to reach a plan were recorded and are shown in Table 4.2.

Task No.	*Steps in Plan*	*Time (Secs)*	*Operators Used*	*Methods Used*
1	4	0.00	0	1
2	4	0.00	0	1
3	2	0.00	0	1
4	6	0.00	0	2
5	10	0.01	0	3
6	14	0.02	4	3
7	18	0.02	4	4
8	20	0.01	4	5
9	21	0.01	5	5
10	24	0.02	8	5

Table 4.3: HyHTN Plan Times Using Operators and Methods

Once each potential method had been tested as above, combinations of operators and methods were tested to achieve increasingly large sections of the full problem. Table 4.3 shows the ten different tasks that were tested on the Extended Tyre Domain. The second column lists the number of plan steps in terms of operators used and needs to be viewed in conjunction with the fifth column which lists the number of methods used because each method consists of ordered sets of operators. The third column lists the times taken to reach a successful plan using the HyHTN planner in every case.

The Full Tyre-Change Problem

So far none of the tests performed tested the domain for the full tyre change problem. As can be seen in Table 4.2 our results so far were showing that it was slower to use a method than to use operators for plans with a small number of steps. These results

compared with what we found using HBC. As yet we had not tried to use the task containing the full planning problem defined in section 4.2.2. Tasks for this full problem were coded up in the two ways using for the first just operators and for the second mostly methods and a few operators.

4.2.5 Results for the Full Problem

Using Just Operators

Using just operators was challenging because of the number of operator choices and the number of objects available to instantiate those chosen operators. The task was left running on a Sun Blade 100 overnight. After 20 hours no plan was found.

Using Operators and Methods

Each method, by the fact that it decomposes into an operator sequence, stipulates a chunk of the plan to be achieved. Our task used 5 methods and 8 operators and the planner found the plan shown below in 0.02 seconds.

```
SOLUTION
open_container(boot)
fetch_pump(boot,pump0)
find_puncture(pump0,tyre1)
putaway_pump(boot,pump0)
fetch_wrench(boot,wrench0)
fetch_jack(boot,jack0)
remove_trim(hub1,trim1,wheel1)
loosen(wrench0,hub1,trim1,nuts1)
jack_up(nuts1,hub1,jack0)
undo(wrench0,trim1,hub1,jack0,nuts1)
remove_wheel(trim1,wheel1,hub1,jack0)
fetch_wheel(boot,wheel5)
putaway_wheel(boot,wheel1)
put_on_wheel(trim1,wheel5,hub1,jack0)
```

```
do_up(wrench0,trim1,hub1,jack0,nuts1)
jack_down(nuts1,hub1,jack0)
tighten(wrench0,hub1,trim1,nuts1)
apply_trim(hub1,trim1,_108198)
putaway_wrench(boot,wrench0)
putaway_jack(boot,jack0)
fetch_pump(boot,pump0)
inflate_tyre(pump0,tyre2)
putaway_pump(boot,pump0)
close_container(boot)
END FILE
```

Results of Testing the Full Problem

Only one operator, apply_trim, was not fully instantiated and this did not affect correct plan formation. We had shown that, for a more complex domain, the use of hierarchical methods was a much more efficient way of planning.

4.2.6 Ideas for Improvements on the Opmaker System

It could well be argued that the efficiency of hierarchical planning comes at large cost in terms of time to construct methods and tasks. Indeed when we were constructing the example version of ETD we found this to be the case even using *GIPO* for some of the task editing. Even when using *opmaker*1.1 to induce methods and operators and even though initial action sequences can be rapidly achieved by 'point and click' means, it takes time to construct the example input material from which *opmaker*1.1 obtains the intermediate and goal states for the transitions as shown in chapter 3. Whilst our *opmaker*1.1 system saved significant time spent on operator and method construction, it needed to be more automated and less dependent on a knowledge engineer's input. We needed a system to speed up the induction process. The areas that could potentially be speeded up were the construction of the action sequence

and the compiling of the example material. We argued that time could not be saved constructing initial and goal states as they were necessary to the planning process anyway and were already represented in a planning task. The editor provided by the present *opmaker*1.1 system embedded in GIPO allows rapid construction of an initial action sequence. Hence we decided to concentrate our efforts on the example input material. In Section 4.3 of this chapter we discuss how the example input material was generated automatically by the next phase of *opmaker*, speeding up the process of induction.

4.3 Automatic Induction Without Intermediate State Information

We begin this section by recalling why we need example material for operator induction then we consider the argument for automatic generation of that example material and briefly state why this line was adopted. We consider the ways that we can get all the choices for the right hand side of state transitions for the operators and then we discuss ways of narrowing those choices to obtain good quality input example sets. Finally we show some resulting sets of example material.

4.3.1 The Need for Example Material

We recall that four main items of input were needed for *opmaker*1.0 and *opmaker*1.1 to induce operators. These were:-

- A training sequence of actions
- A set of initial states for all the objects in the training sequence
- A partial domain (without operators and methods)
- Numbered sets of example material, i.e. the user input.

When *opmaker* builds operators it obtains the post-transition states of the objects from the example material and these are reflected in the right hand sides of the operator transitions. We remind ourselves that some transitions are prevail transitions in which the post-transition state is the same as the initial state and that the example material we have met so far indicated the presence of a prevail transition to *opmaker* by use of the 'null' clause.

For a typical initial sequence of

```
do_up(wrench0,hub1,jack0,nuts1,trim1),
jack_down(hub1,jack0),
tighten(wrench0,hub1,nuts1,trim1),
apply_trim(hub1,trim1,wheel5)
```

a suitable set of example material could be

```
% do_up(wrench0,hub1,jack0,nuts1,trim1)
input(1,wrench0,null).
input(1,hub1,sclass(Hub1,hub,[jacked_up(Hub1,Jack0),fastened(Hub1)])).
input(1,jack0,null).
input(1,trim1,null).
input(1,nuts1,sclass(Nuts1,nuts,[loose(Nuts1,Hub1)])).

% jack_down(hub1,jack0)
input(2,hub1,sclass(Hub1,hub,[on_ground(Hub1),fastened(Hub1)])).
input(2,jack0,sclass(Jack0,jack,[have_jack(Jack0)])).

% tighten(wrench0,hub1,nuts1,trim1)
input(3,wrench0,null).
input(3,hub1,null).
input(3,trim1,null).
input(3,nuts1,sclass(Nuts1,nuts,[tight(Nuts1,Hub1)])).

% apply_trim(hub1,trim1,wheel5)
```

```
input(4,hub1,null).
input(4,trim1,sclass(Trim1,wheel_trim,[trim_on(Trim1,Wheel5)])).
input(4,wheel5,null).
```

and this example material was coded by hand so prone to errors. When *opmaker* was embedded into GIPO a series of questions were posed to the user who was effectively being asked to supply the end state for each transition. Again this process depended on the user making correct choices.

4.3.2 The Argument for Automatic Generation Without Intermediate State Information

Consistency and accuracy. An effective automisation of the process of forming the example input material would be a good way to ensure consistency and accuracy of the input material.

Time saved. Furthermore this would speed up the process of induction by saving the time taken to code and correct the examples by hand or by using GIPO.

Increase of abstraction. A further argument takes the line that automating the process of generating the examples increases the abstraction of domain building and removes more of the need for domain builders to be knowledge engineering experts.

Autonomous Learning. An agent (system) should be able to learn and plan without direct human intervention so that it can be of use in remote areas or those where it is unsafe for the human to go. If an agent is given action sequences as training material it can learn the knowledge it needs to plan safely and efficiently.

Having given due consideration to these arguments it seemed clear that a good way forward was to try to find some way of generating the example material automatically.

4.3.3 Generation of Examples

In later parts of this chapter we consider the process as a whole, so we view the whole picture as being one of generating the operators and methods. Internal to the system, however, the idea of having example material is useful to hang onto for descriptive purposes. This material is generated automatically and used by *opmaker*2.0 to generate the operators and methods.

The domain substate classes in any *OCL* domain contain templates for sets of states, and objects must be in one of these states at any one time. For example, in Section 4.1.3, three states for a jack are listed. At any one time the jack must be in one of those states and one only. If jack is in the state have_jack(J) a transition can do one of two things - either leave jack's state unchanged or change it to one of the other states. The idea behind automatic generation of operators is to consider possible alternative state combinations for the objects involved.

As a simple example suppose the initial sequence was

```
open_container(boot),
fetch_pump(boot,pump0),
```

and the initial states for boot and pump0 were

```
ss(container,boot,[closed(boot)]),
ss(pump,pump0,[pump_in(pump0,boot)])
```

then we can have the following possible example states:-

1. closed(boot) and pump_in(pump0,boot),

2. open(boot) and pump_in(pump0,boot),

3. open(boot) and have_pump(pump0),

4. closed(boot) and have_pump(pump0).

The first of these is effectively the prevail state where neither object changes state, whilst the other three are potential outcomes, even though actually only item 2 can be achieved in a single action.

Since each item in this sequence has only two possible states the possible combinations for example material sets are 2 x 2 = 4. For a longer sequence of actions such as that shown in Section 4.3.1 where more objects are involved the possible number of potential example sets, even for this short sequence of four actions, is 2 654 208!

4.3.4 Heuristics to Reduce Choice

This number can be drastically reduced by using information we already have or can easily obtain by:-

1. Making sure the initial sequence of actions includes information about which objects will not change at each action. Figure 4.2 shows one system of tagging objects that will not change in an example sequence.

2. Including heuristics that handle information in 1 above, for example if an object is labeled as 'changing' (untagged), then its pre-action state is not available as its post-action state.

3. Allowing goal states specified in the tasks to be used to narrow the search.

4. Using all the information in the substate classes to narrow the search - the only allowable states are listed there.

```
do_up(@wrench0,hub1,@jack0,nuts1,@trim1),
jack_down(hub1,jack0),
tighten(@wrench0,@hub1,nuts1,@trim1),
apply_trim(@hub1,trim1,wheel5)
```

Figure 4.2: The Initial Sequence Tagged (with '@') to Indicate Unchanging Objects

5. Using additional domain information, not previously used, to be declared in the invariants where it is not already implied by the substate classes.

6. Where potential operators are seen as direct opposites of one another these are declared e.g. do_up and undo.

4.3.5 Changes to Input to Indicate Unchanging Objects

We give, in Figure 4.2 a short sequence of actions as an example of typical user input. The user has only to compose the sequence and decide whether objects will be changed by the named action. In this sequence the '@' sign has been used by the user to indicate *unchanging* objects.

4.3.6 Calculating Paths Through State Space

We now consider how items 1 - 4 listed in Section 4.3.4 above reduce the possible paths through the state space and then go on to discuss the contribution made by invariants. For the sequence in Figure 4.2 the first action has five objects. Three of these, wrench0, jack0 and trim1 are tagged so only the initial state can be selected. Of the others, which *must* change state, hub1 has three potential different states whilst nuts1 has two. Hence the different variants for the first action total 1 x 3 x 1 x 2 x 1 = 6. Similarly we can calculate that the second action has 3 x 2 = 6 variants and the third has 1 x 1 x 2 x 1 = 2 variants. The fourth action with objects hub1, trim1 and wheel5 looks as though there is only one state each for hub1 and trim1 but three

states for wheel5. States for wheel are state sets which include a value for trim and the initial have_trim(trim1) state must change. The wheel state must be consistent with the trim state so the only choice available is wheel_on(wheel5,hub1),trim_on(wheel5, trim1). Hence our number of variants for the last action is 1 x 1 x 1 = 1. The combined number of different example sets for this action sequence is therefore 6 x 6 x 2 x 1 = 72.

4.3.7 Initial Results from Automatic Generation of Paths

Using experimental input and heuristics we found that tagged sequences, initial states and partial domains could be used to generate several different paths. These were counted and confirmed the number of 72 shown in the previous section.

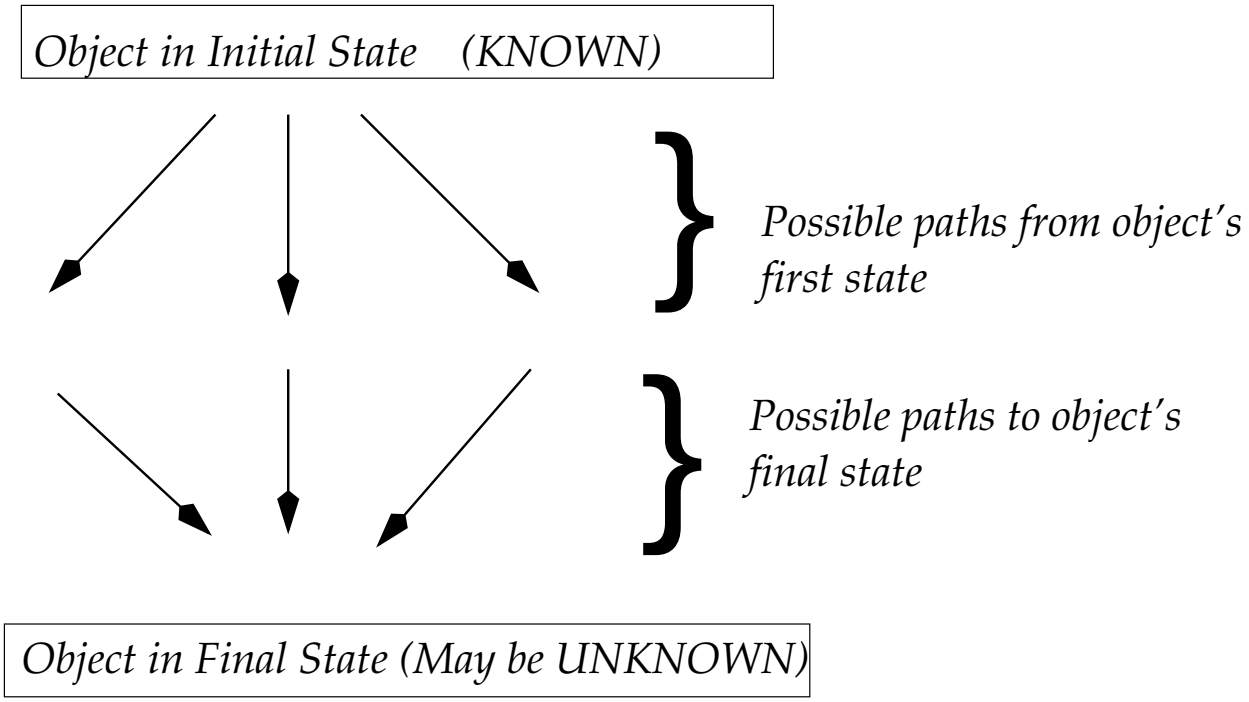
This could still mean that, for longer action sequences, there would be very large numbers of example sets. Also we had not taken account of any other restrictions in the domain, for example the invariants. We decided, on this evidence that we needed to use the information contained in the invariants.

To show how the different paths were reached we can see in Figure 4.3 that firstly an object has a known initial state. Information about the final state may be missing. The search for a path through the state space of potential interim states aims to find a final state and fill in the details each step of the way. In the second diagram we see the object's trace tracked. In Figure 4.4 we consider the object hub1 in the first action from the sequence shown in Figure 4.2. We note that it is not tagged which means it must change. In Figure 4.4 we see the potential paths one of which is not available because it represents a prevail transition.

Object	*No. of Substates*	*Substates*	*Valid?*	*No. Available*
Wrench	2	have_wrench(Wr)	Yes	1
		wrench_in(Wr,C)	Yes	
Hub	4	on_ground(H),fastened(H)	No	3
		jacked_up(H,J),fastened(H)	Yes	
		free(H),jacked_up(H,J),unfastened(H)	Yes	
		unfastened(H),jacked_up(H,J)	Yes	
Jack	3	have_jack(J)	Yes	2
		jack_in_use(J,H)	Yes	
		jack_in(J,C)	Yes	
Nuts	3	tight(N,H)	Yes	2
		loose(N,H)	Yes	
		have_nuts(N)	Yes	
Trim	2	trim_on_wheel(WT,Wh)	Yes	1
		have_trim(WT)	Yes	
Wheel	4	have_wheel(Wh),trim_off(Wh)	No	1
		wheel_in(Wh,C),trim_off(Wh)	No	
		wheel_on(Wh,H),trim_off(Wh)	Yes	
		wheel_on(Wh,H),trim_on(Wh,WT)	Yes	

Table 4.4: Table Showing Total States and States Available for the Action Sequence in Figure 4.2 if States *Must* Change

1. *For a Changing Object*



2. *Tracking the Path of Object Changes*

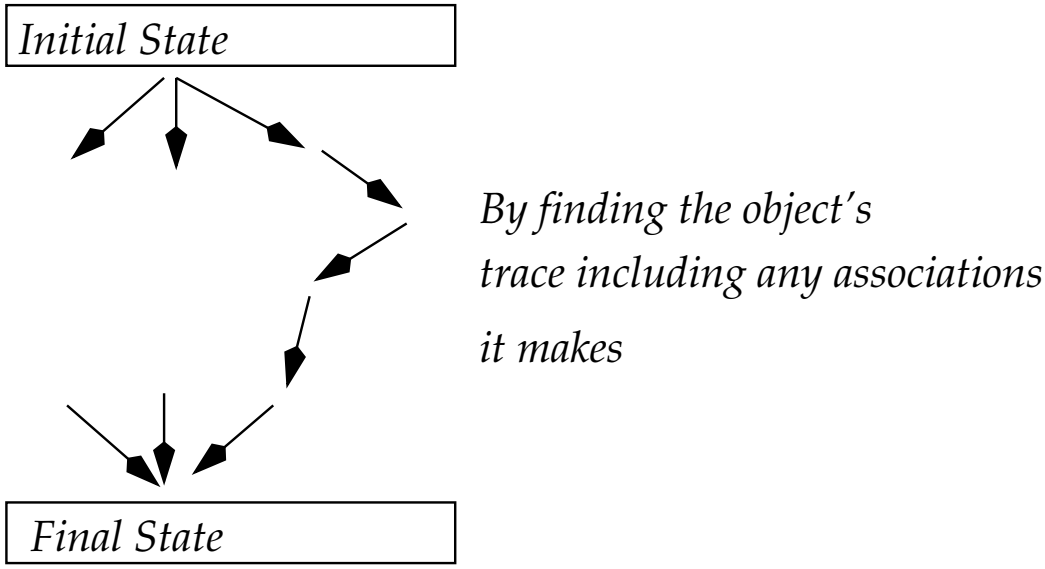


Figure 4.3: Changing States of an Object in a Sequence

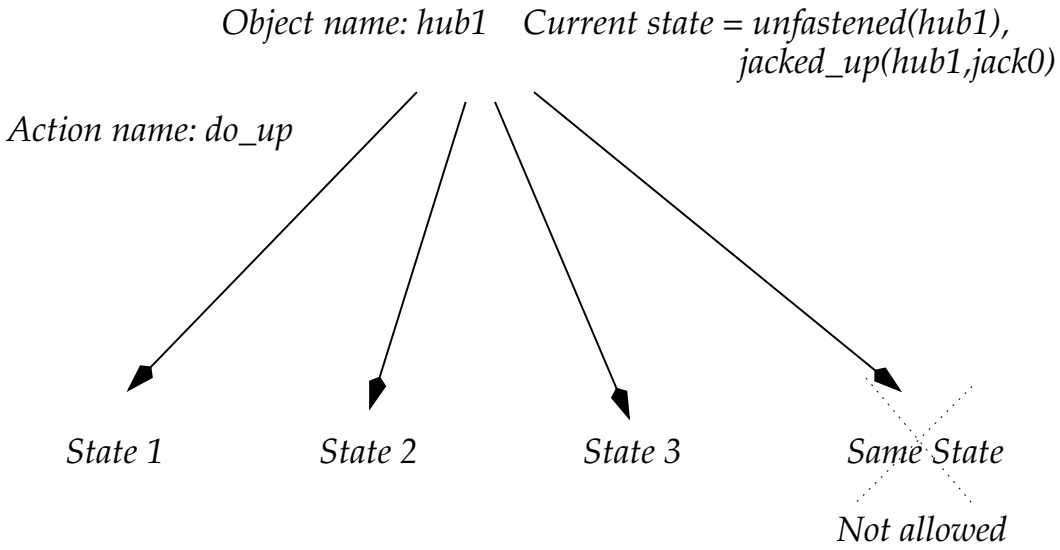


Figure 4.4: Changing States of hub1 in a Sequence

Method Name	*Number of Examples in Set*	*Accuracy*
fix_flat	2	1 accurate 1 not accurate
fetch_tools	1	accurate
discover_puncture	2	1 accurate 1 not accurate
unfasten_hub	1	accurate
change_wheel	1	accurate
attach_hub	1	accurate
putaway_tools	1	accurate

Table 4.5: Table Relating Numbers of Example Sets to Methods

4.3.8 The Use of the Invariants to Reduce the Search Space

Previously, using *opmaker*1.0, the partial domains used for induction had contained invariants in the form of atomic invariants, implied invariants and inconsistent constraints. The substate classes also act like invariants in this respect by controlling the transitions available. Of the invariants, only the atomic invariants were used in the induction process. We needed to accurately define and use other knowledge contained in the invariants as a further search heuristic.

Invariants were added where the constraints they represented were not already represented in the substate classes and atomic invariants. Their function is to guide the state space search for intermediate and goal states. The logic of the invariants is shown in Figure 4.5.

At the same time the *opmaker* code was altered and extended in the following ways.

- It took account of the invariants written as shown.
- All objects in the operators had an initial substate.
- There was a testing procedure on the invariants so that if any were incorrect they could quickly be adjusted.

4.3.9 Results Using the Invariants

When using *opmaker*1.1 we needed to give numbered sets of examples to the system as part of the input. The first phase of *opmaker*2.0 replicates these sets which can be compared to the old example sets for accuracy and can also be counted to show how accurate the state space search is. Test files were written to generate example sets for each of the 7 desired methods for the Extended Tyre Domain shown in Figure 4.1. These were run in turn with the extended *opmaker* and the results are shown

1. Equivalence between hub *fastened* and nuts *tight/loose* on hub.

$$\forall\, H{:}hub\,.\,[fastened(H) \Longleftrightarrow \exists\, N{:}nuts\,.\,(tight(N, H) \vee loose(N, H))]$$

2. Equivalence between *jack_in_use* and *jacked_up*.

$$\forall\, H{:}hub\,.\,\forall\, J{:}jack\,.\,[jack\_in\_use(J, H) \Longleftrightarrow jacked\_up(H, J)]$$

3. Equivalence between hub not *free* and *wheel_on* hub.

$$\forall\, H{:}hub\,.\,[\neg free(H) \Longleftrightarrow \exists\, W{:}wheel\,.\,wheel\_on(W, H)]$$

4. Equivalence between *trim_on_wheel* and *trim_on*.

$$\forall\, T{:}wheel\_trim\,.\,\forall\, W{:}wheel\,.\,[trim\_on\_wheel(T, W) \Longleftrightarrow trim\_on(W, T)]$$

5. Only a single set of nuts can be on a hub.

$$\forall\, H{:}hub\,.\,\forall\, N_1{:}nuts\,.\,\forall\, N_2{:}nuts\,.\left[\left(\begin{array}{c}(tight(N_1, H) \vee loose(N_1, H))\\ \wedge\\ (tight(N_2, H) \vee loose(N_2, H))\end{array}\right) \Rightarrow (N_1 = N_2)\right]$$

6. Only a single wheel can be on a hub.

$$\forall\, H{:}hub\,.\,\forall\, W_1{:}wheel\,.\,\forall\, W_2{:}wheel\,.\left[\left(\begin{array}{c}wheel\_on(W_1, H)\\ \wedge\\ wheel\_on(W_2, H)\end{array}\right) \Rightarrow (W_1 = W_2)\right]$$

7. Domain constraint: If nuts are tight on a hub then the hub must be on the ground.

$$\forall\, H{:}hub\,.\,[(\exists\, N{:}nuts\,.\,tight(N, H)) \Rightarrow on\_ground(H)]$$

8. Domain constraint: if a trim is on a wheel, then the wheel is on a hub and the nuts are tight.

$$\forall\, W{:}wheel\,.\,\exists\, T{:}wheel\_trim\,.\left[\begin{array}{c}trim\_on\_wheel(T, W) \Rightarrow\\ (\exists\, H{:}hub\,.\,wheel\_on(W, H)) \wedge (\exists\, N{:}nuts\,.\,tight(N, H))\end{array}\right]$$

Figure 4.5: Invariants encoded in the Extended Tyre World

in Table 4.4. The accuracy of the recorded example sets was also noted. For each method there was exactly one accurate example set for this domain. There were two inaccurate example sets which relate to the methods fix_flat and discover_puncture. In these cases the sense of the method was either that the tyre could be pumped up to change it from flat to full or that it could not be filled with air and so was designated punctured. However, in each case there were two other states for tyre apart from flat, and, with no invariant guidance, we got two example sets. One of these gave the final state for tyre as punctured(Ty) and the other gave it as full(Ty). These were relatively trivial inaccuracies and could be resolved interactively with the user selecting the correct version for the sense of the method.

4.4 How Opmaker2 Learns

In this section we show diagrammatically how the system is designed and we give a formal algorithm for the *opmaker*2 system. The section concludes with a descriptive 'walk-through' of the algorithm using a chosen action sequence. This is similar to the example given in a recent paper, [50].

4.4.1 A Diagrammatic Representation of the Opmaker2 System

Figure 4.6 gives an indication of the design of the system for inducing operators and methods in *opmaker*2. In figure 4.6 the boxes at the top of the diagram indicate the input to the *opmaker*2 system, whilst those at the bottom indicate output from the system. The large ellipse is the *opmaker*2 system which includes two further processes - 'Generate' and *opmaker*1 (described in Section 3.3). Generate is the new process which uses the input of the partial domain, the invariants, the initial states and optionally also the goal states, and the action sequence. Output from Generate is two-fold. The example states represent the automatically generated intermediate and

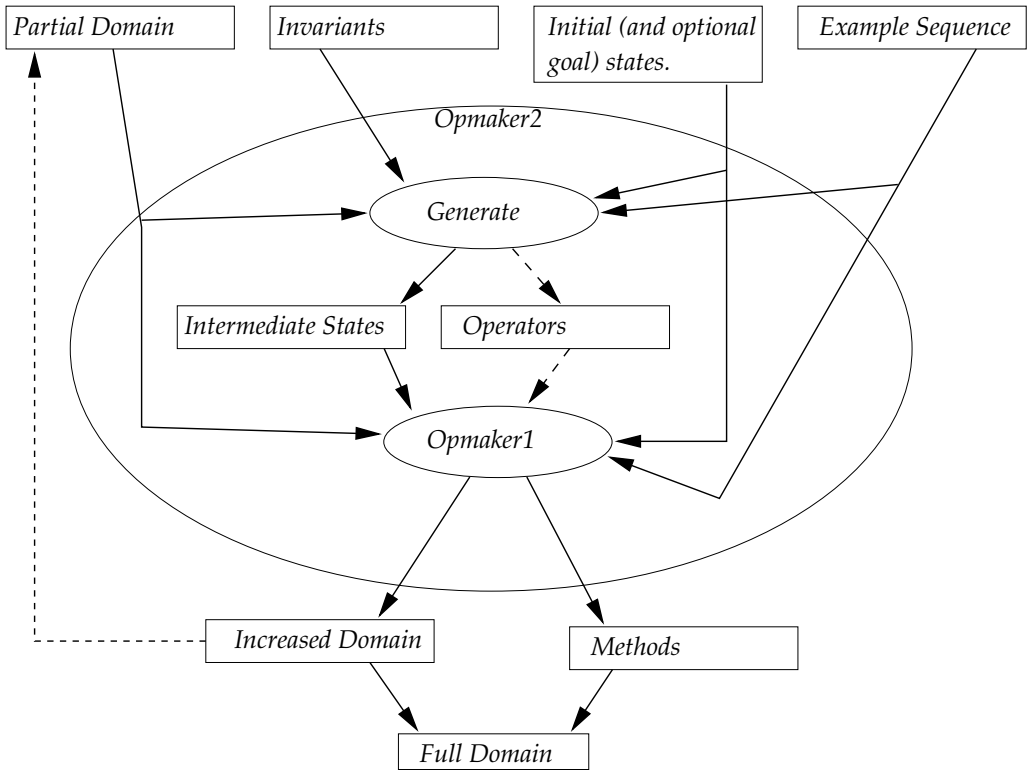


Figure 4.6: Diagrammatic Representation of the Opmaker2 System

end state sets which *opmaker*1 required as input to induce operators. Additionally and optionally Generate will generate operators but not methods. *Opmaker*1 is still required for the generation of methods and will generate a method for each input action sequence and is able to use most of the input; namely the partial domain, initial states and action sequence. In addition *opmaker*1 uses the example sets output from Generate and any operators from Generate to finally produce the more complete domain with some of the operators and a method. Each iteration takes a fresh action sequence so that a full set of operators and methods is built up and the final output is the full domain.

4.4.2 Outline Design of the Opmaker2 Algorithm

In the following algorithm variables are shown in uppercase. Assignments and storage of operators is undone on backtracking.

program **opmaker2**

In: a set of sorts,

a set of valid substate classes for each sort,

a set of objects,

initial and (optionally) final substates of objects,

state invariant conditions,

example sequence of actions.

1. Definitions:

   Obj.sort – sort of object *Obj*

   Obj.name – name of object *Obj*

   Obj.substate(*i*) - ground substate of object *Obj* at step *i*

   init – identifier for initial state

   final – identifier for final state

   Obj.substate(*init*) – initial state of *Obj*

   Obj.substate(*final*) – final state of *Obj*

   Act.prevail – set of objects which are unchanged by action *Act*

   Act.changing – set of objects which are changed by action *Act*

   Sort.substate_classes– set of substate classes for Sort

2. current:=init
3. next:=successor(init)
4. for each action *Act* in training sequence
5. for each object *Obj* in *Act.prevail*
6. *Obj.substate*(*next*) := *Obj.substate*(*current*)

7. end for
8. for each object Obj in $Act.changing$
9. if Obj doesn't change again in rest of sequence
10. and final state of Obj is known,
11. $Obj.substate(next) := object.substate(final)$
12. else
13. choose $Substate\_class \in (Obj.sort).substate\_classes$
14. $Obj.substate$:=**bind**$(Substate\_class, Obj, Act.prevail \cup Act.changing)$
15. if $Obj.substate(current)$=$Obj.substate(next)$
16. backtrack to previous choice point
17. end if
18. end if
19. end for
20. if the global state $\bigcup_{Obj} Obj.substate(next)$
21. is inconsistent with some invariant condition
22. then backtrack to last choice point
23. end if
24. Derive implied definition of operator and store with name $Act.name$
25. if operator is inconsistent with any previous definition then
26. backtrack to previous choice point
27. end if
28. current:=next
29. next:=successor(current)
30. end for
31. produce a method from the sequences of actions and states as in Opmaker

function **bind**$(Substate\_class, Obj0, Params)$

32. Let $ObjVar0$ be the variable placeholder for the object described by $Substate\_class$
33. Let $ObjVars$ be the the set of other free variables in $Substate\_class$
34. form $Bindings$, a set such that
35. $\quad \langle ObjVar0, Obj0 \rangle \in Bindings$ and
36. $\quad$ for each $ObjVar \in ObjVars$, there is a pair $\langle ObjVar, Obj \rangle \in Bindings$
37. $\quad\quad$ with choice of $Obj \in Params$ such that $V.sort = Obj.sort$
38. form ground substate S from $Substate\_class$ with $Bindings$
39. if no disequality constraints in S are broken
40. $\quad$ return S
41. else
42. $\quad$ backtrack to previous choice point
43. end if

4.4.3 A Description and Walk-Through of the Algorithm

The learning method is specified by the algorithm description at the start of Section 4.4.2. In outline, the method is:

1. Use a set of heuristics and inference to track the changing states of each object referred to within a training example, taking advantage of the static, object-state information and invariants within the domain model. Infer full details of object transitions for each dynamic object.

2. Use the techniques of the original Opmaker algorithm [47] shown in Figure 3.3 to generalise object references and create parameterised operator schema from the specific object transitions extracted in 1 from the training examples.

To illustrate the main innovations of the algorithm in Section 4.4.2, we will use an example walk-though taken from the extended tyre domain. In this sequence a changed wheel is secured on the hub and the vehicle is made ready for use. Assume a training sequence is input into *opmaker*2 and this has components as follows:

name: do_up; **prevail:** wrench0, jack0, trim1; **changing:** hub1, nuts1
name: jack_down; **changing:** hub1, jack0
name: tighten; **prevail:** wrench0, hub1, trim1; **changing:** nuts1
name: apply_trim; **prevail:** hub1; **changing:** trim1, wheel5.

Prevail objects have to be present in a particular state during execution of the action, but remain unaffected (wrench0 is available, jack0 is jacking up the wheel, trim1 is hub1's wheel trim and has to have been removed).

The 'changing' objects *must* change state (hub1 becomes 'fastened' and nuts1 are 'done_up').

Line 4 takes the first action in the training sequence shown above and identifies the objects.

Line 5 identifies the prevail (unchanging) objects as wrench0, jack0 and trim1. It iterates through these and **Line 6** makes prevail transitions from them.

Line 8 identifies the changing objects (hub1 and nuts1) and takes them in turn - the first of these is hub1.

Line 9 looks ahead to see if hub1 will change again in a subsequent action and finds that it does in the second action in the sequence. In the case of nuts1 they also change again in the third action when they are tightened. If we had chosen an example where neither changing object changes again after the first action then **Lines 10 and 11** execute and a transition to the object's final state is made.

We are still considering hub1 and now **Line 13** matches a suitable substate class for the sort of the object (hub) and **bind**s the variable to the object unless, **Line 16**, it selects the objects next substate to be the current one. (This rules out making a prevail transition for a changing object).

The current state of hub1 is [unfastened(hub1), jacked_up(hub1)] and there are four potential substate classes to select, which we name below as S1-4:

S1 = [on_ground(h),fastened(h)],
S2 = [jacked_up(h,j),fastened(h)],
S3 = [free(h),jacked_up(h,j),unfastened(h)],
S4 = [unfastened(h),jacked_up(h,j)]

Since hub1's current state is not necessarily its final one but we know it must change, there are 3 potential transitions.

(hub, h, [unfastened(h), jacked_up(h,j)] → [on_ground(h),fastened(h)])
(hub, h, [unfastened(h), jacked_up(h,j)] → [free(h),jacked_up(h,j),unfastened(h)])

(hub, h, [unfastened(h), jacked_up(h,j)] → [jacked_up(h,j),fastened(h)])

Lines 20-23 of the algorithm check these transitions with the invariants, derive potential end states and form a transition. This repeats for the other changing object, nuts1.

Line 24 stores the definition of the operator and checks it against any previous definition.

Example sets are formed by the stage of line 23. Operator descriptions are delivered by use of the whole algorithm.

Finally, a hierarchical method is generated by combining the 4 action schema generated from the action sequence in a similar fashion to the original *Opmaker* system [47].

4.5 Experiments and Results

In this section we discuss our testing process used on three different domains: the Extended Tyre Domain, the Hiking Domain and a 7-block version of the well- known Blocks World Domain [83]. We give details of the full planning problem devised for each domain. We set down the aims of our testing and the criteria for success. Next we list the results for each domain and discuss the implications of these. Finally we draw our conclusions on these results.

4.5.1 The Extended Tyre Domain (ETD)

Details of this domain are given in section 4.1.2 where there is a statement of the full planning problem. This domain had several invariants which have been implemented in the search for example intermediate states. Sets of example material were generated and these were tested using the test criteria detailed in section 4.5.4.

4.5.2 The Hiking Domain

Details of this domain are given in section 2.2.1. We identified the full hiking problem for the purposes of this test as being the completion of one complete leg of the walk, including the movement of the cars and personnel ready for the next day's walk. Again sets of example material were generated to be measured against our test criteria in section 4.5.4.

4.5.3 The Blocks World Domain

We have made little reference to this frequently used and quoted domain so far. Our *OCL* version of this domain has, where B, B1, B2 etc represent different blocks and G represents the gripper, the following operators:-

1. grip_from_blocks(B,B1,G,B2) (a block is gripped from a pile of blocks)

2. grip_from_one_block(B,B1,G) (a block is gripped from one other block)

3. grip_from_table(B,G) (a block is gripped from the table)

4. put_on_blocks(B,G,B1,B2) (a block is put on a pile of more than one block)

5. put_on_one_block(B,G,B1) (a block is put on another)

6. put_on_table(B,G) (a block is put on the table)

The task we devised for this domain to use as a motivation for inducing the operators is defined as follows:-

Beginning with a pile of 7 ordered blocks (block 1 on the bottom and 7 on the top) the task is to move the blocks so that the three even numbered blocks form one pile in order (bottom to top) of block 6, block 2 and block 4 whilst the four odd numbered blocks form the pile ordered (bottom to top) of block 5, block 1, block 7 and block 3. The overall problem splits into 7 sections.

1. 'position first even' aims to place block 6 on the table.

2. 'position first odd' aims to place block 5 on the table.

3. 'position second even' aims to place block 2 on block 6.

4. 'position second odd' aims to place block 1 on block 5.

5. 'position last even' aims to place block 4 on blocks 6 and 2.

6. 'position third odd' aims to place block 7 on blocks 5 and 1.

7. 'position last odd' aims to place block 3 on blocks 5, 1 and 7.

Splitting this planning problem into seven methods is only one way to solve it but will be sufficient to demonstrate that *opmaker*2 can be applied to other problems.

4.5.4 The Testing Criteria

Since induction sequences deliver several actions and a single method, initial sequences would be tailored to produce a meaningful method, and sufficient initial sequences would be composed to cover all the major sub-tasks that could be required by the domain. In each case the agent would begin with domain knowledge but have sketchy knowledge about its potential actions.

The Aims of Testing

1. To produce example sets automatically from the domains given initial and goal states (already declared in the task for the domain that is equivalent to that contained in the initial sequence), a partial domain and the handcrafted (or GIPO-crafted) action sequence.

2. To judge the efficiency of the system based on the number of example sets obtained and the accuracy of their content.

3. To use the example material obtained and *opmaker*2 to to induce operators and a method for each action sequence.

4. Using standard planners, to compare efficiency of planning using the new operators as opposed to the handcrafted ones.

Success Criteria

Success would be judged based on:-

1. Low numbers of example sets produced (ideally a single set) and accuracy of their content.

2. The ability to use the example material to produce operators and methods.

3. The results of comparison between the operators and methods produced and the hand-crafted versions.

4. Planning using the operator and method output of *opmaker*2 should be comparable in accuracy and efficiency to planning with the hand-crafted version. (Since the aim of this work is efficiency in knowledge engineering, planning with the new output does not need to be faster.)

5. Low numbers of invariants required. (Since these take time to construct there is a trade off between time saved constructing examples and time spent constructing invariants.)

4.5.5 Results for the Extended Tyre Domain

This domain was used for the testing of the initial stages of the development of *opmaker*2. Gradually during development extra tools were added which took account of goal states, tagging of unchanging items in transitions, use of invariants and use of 'opposite actions'. In the case of this last item we had noticed that sometimes extra example sets were generated where actions were the opposites of others. When we rectified this bug and declared actions to be opposite actions of others the numbers of example sets reduced further. Examples of opposite actions could be jack_up and jack_down, or apply_trim and remove_trim.

Findings

- By using the eight invariants shown in Figure 4.5 we obtained unique example sets for all method sequences in ETD.
- In each case the example set for a method matched the hand-coded example sets we had used with the *opmaker*1 system.
- The example sets delivered a full set of operators and methods using the *opmaker*2 system.
- The operators and methods matched the hand-crafted versions.
- Planning using the induced operators and methods from *opmaker*2 matched results for *opmaker*1, with the full plan using methods and operators taking a similar time to complete an accurate plan.
- With this large domain there were eight invariants. This was more than were required for the other domains we tested but the size of this domain was significantly larger too.

Apart from the large number of invariants required we judged the tests using this domain to be a success.

4.5.6 Results for the Hiking Domain

This domain differed from ETD by containing static objects. These were in the form of locations modelled as static objects in *OCL*. During testing we found that they could be identified as prevail objects in each action in the sequence, allowing the sense that a tent could be 'up' at Keswick and 'down' at Keswick to be expressed. Places possible were the hiking legs Keswick, Derwent, Helvelyn etc. and the alteration

meant that it was acceptable to have a transition from state 1 to state 1 so long as the variable bindings changed.

Findings

- Low numbers (1 - 4) of example sets were produced for 4 of the potential methods. The fifth was complicated by the fact that actions were repeated frequently and gave large numbers of examples. This did not affect planning since operators could be used in its place.
- A set of 4 methods and all operators could be reproduced.
- Operators and methods matched the hand-crafted ones.
- Planning using induced methods and operators produced similar results to use of the hand-crafted alternatives.
- Only one invariant was required.

One difficulty was the final method which gave initially 168 example sets. However the remainder of the domain gave good results although there were only two sets of unique examples. One advantage was the low number of invariants required - for this domain a single invariant was needed.

4.5.7 Results from the Blocks World Domain

Section 4.5.3 gives details of the operators to be replicated and the defined task. As before test files were built and in each case, for this domain, a single solution set was obtained. An example, where the gripper is represented by the name tom, is shown below.

```
%% grip_from_blocks(block7,block6,@block5,tom)
input(1,block7,sclass(Block7,block,[gripped(Block7,Tom)])).
input(1,block6,sclass(Block6,block,[on_block(Block6,Block5),clear(Block6),
ne(Block6,Block5)])).
input(1,block5,null).
input(1,tom,sclass(Tom,gripper,[busy(Tom)])).

%% put_on_table(block7,tom)
input(2,block7,sclass(Block7,block,[on_table(Block7),clear(Block7)])).
input(2,tom,sclass(Tom,gripper,[free(Tom)])).

%% grip_from_blocks(block6,block5,@block4,tom)
input(3,block6,sclass(Block6,block,[gripped(Block6,Tom)])).
input(3,block5,sclass(Block5,block,[on_block(Block5,Block4),clear(Block5),
ne(Block5,Block4)])).
input(3,block4,null).
input(3,tom,sclass(Tom,gripper,[busy(Tom)])).

%% put_on_table(block6,tom)
input(4,block6,sclass(Block6,block,[on_table(Block6),clear(Block6)])).
input(4,tom,sclass(Tom,gripper,[free(Tom)])).
Solutions = 1 ?
```

With this particular domain we noted that an 'ne' clause (not equal) appeared in the example material as can be seen in input 1 and input 3 above.

The 'ne' clause is part of the structure of the *OCL* language and is used to show that the arguments are not the same, so that, as above,

```
ne(Block6,Block5)
```

means that Block6 and Block5 are not equal (ne), i.e. they are different blocks.

This device is only used in domains where there are two objects of one sort listed in a predicate as in

```
on_block(Block6,Block5)
```

making the available substates for the object block to be

```
[gripped(B,G)],
[on_block(B,B1),clear(B),ne(B,B1)],
[on_block(B,B1),ne(B,B1)],
[on_table(B),clear(B)],
[on_table(B)]
```

.

In English these say, 'A block may be gripped, or it may be on a block with no block on top of it (i.e. clear), or it may be on a block with another on it, or it may be on the table with no other block on it, or it may be on the table with another block on it'.

Findings

- In every case a unique set of examples was given.
- Operators and methods were generated.
- Operators grip_from_table and put_on_table were accurate. Operators involving more than one block in their descriptions were not accurate. The variable names for the blocks were not necessarily the ones intended.
- Due to a lack of accurate induced operators planning was not attempted.
- No invariants were required.

4.5.8 Our Conclusions From These Results

This little 'ne' clause was a difficulty when it came to using these example inputs to induce the operators. Those operators which contained the 'ne' clause were incorrect

and so a method could not be formed. For now we have consigned the removal of this bug to 'Future Work' and the reader is referred to Chapter 6 where ideas for this are mentioned.

Despite this bug we have tried three different domains and been able to induce example sets for input into the operator induction process from them all.

From the results obtained so far we can conclude that an agent, given a 'working stock' of potential action sequences, and having domain knowledge and a 'belief' about the states of objects it 'knows' about will be able to generate its own examples and use them to supply itself with parameterised actions to suit every possible object combination. Since methods can be formed from the action sequences the agent should be able to plan efficiently and autonomously and does so for most of our results, so long as the number of operators is above about 12. For domains with smaller numbers of operators there seems to be no advantage using methods.

4.5.9 Training Sets for Opmaker

Both versions of *opmaker* require training sets. For both versions it is possible to induce a set of operators that replicate **all** the hand-crafted ones at one go. To do this requires:

- for *opmaker*1 - a full sequence containing at least one copy of all the original operator names (or new equivalents) and parameters, plus user responses to GIPO's efforts to choose correct final states

- for *opmaker*2 - a full sequence as above.

So in this way a full set of operators can be induced with a single set of training examples. However, since for each action sequence a method is produced, it is better

to use several small sequences which lead to a desirable set of methods. Training sequences are very quick to construct, especially using GIPO. It is possible to construct a series of these for the solution to the full problem in less time than it would take to hand-craft two or three operators - thus the system **is** effective at relieving the 'knowledge acquisition bottleneck'.

Chapter 5

RELATED WORK

In this chapter work closely related to ours is reviewed and analysed. The chapter begins with an historical overview of machine learning in the last thirty to forty years. Subsequent sections look more closely at machine learning of rules in Section 5.2 and explanation based learning in Section 5.3. A brief review of theory revision in Section 5.6 is succeeded by the longer Section 5.7 on operator induction. The chapter concludes with a survey of some very recent, relevant work and a summary of entries in the recent ICAPS competitions, Section 5.8. Whilst reviewing the related literature, an analysis of where and why *opmaker* fits into the picture is discussed.

5.1 Machine Learning - Historic

Some of today's applications of Machine learning include predictive text, word processing from speech, search engines, medical applications including diagnosis, detection of credit card fraud, stock market analysis, forensic applications, handwriting recognition, game playing and robotics. There are good reasons why the ability of machines to learn is desirable. In the context of artificial intelligence, it could be argued that the ability to learn is a prerequisite for intelligence whether it be human, animal or mechanical. The arguments for human learning are clear enough; humans use learning to progress through stages of life with increasing abilities such as acquiring new knowledge, learning new skills and improvement through practice. If machines learn like this, then, when the machine is switched off, a good percentage of that learning should be retained, but in many cases that is not quite the case. In

many systems that learn there is a training period by which the system becomes more expert, followed by a period of use where the results are significantly better than in the early stages of training. On the next use the same training is required and the system goes through the same learning process.

Machine learning has evolved over the last three or four decades. In the 1970's research into machine learning became more active and researchers were able to demonstrate learning in a number of areas. Some of these achievements are given in the paragraphs below.

P. H. Winston

One such researcher, Winston, demonstrated a supervised learning task where the challenge to the learner was to learn descriptions of structures such as an arch [84]. The learner was given a positive example in the form of two equal-sized upright blocks (not touching), supporting a horizontal cross-member. A sequence of positive and negative examples was given to the learner. One negative example was having the two uprights touching, another had the cross-member below the uprights, whilst a positive example was having a different shaped cross-member with only one long flat side. After 'observing' the first example the learner formed a hypothesis about a correct description of an arch. With subsequent examples this hypothesis was refined. For each example a set of differences was identified between the example and the current hypothesis, and the differences were used to either generalise the hypothesis, if the example was positive, or to specialise the hypothesis, if the example were negative. Winston concluded from this work that the best training sequence was a series of 'near misses' where only one concept was changed at once.

META-DENDRAL

Meta-DENDRAL [8] was an expert system that helped chemists determine how much the mass spectrometric fragmentation depended upon substructural features. To do this it discovered fragmentation rules for given classes of molecules. Meta-DENDRAL derived these rules by using training instances consisting of sets of molecules with known 3-D structures and known mass spectra. The system then used a Candidate Elimination Algorithm. It first generated a set of highly specific rules which account for a single fragmentation in a particular molecule. Then it used the training examples to generalize those rules.

AQ11

Machine learning was applied to diagnosis of soybean disease in 1978 [52]. Agriculturalists were questioned about disease symptoms of soybean plants. The project was to discover a set of classification rules for the diagnosis of the diseases. Classification rules were learned from training instances and consisted of sample patterns and their correct classification derived from the experts. The researcher, Michalski, represented discrimination rules as a modified version of propositional logic, which includes conjunction, disjunction, and set-membership operators.

ID3

Originally devised by J. R. Quinlan in 1983, the ID3 algorithm spawned many decision-tree methods for machine learning. The basic decision tree process looks at the various attributes of the positive and negative examples. For example if the decision was to do with people then attributes such as height, with instances tall or short, hair-colour, with instances red, blonde and dark, and eye-colour, with instances blue or brown, might be considered. An attribute is chosen at random to form the root of the tree and a decision tree is built recursively on the root. Differ-

ent trees could be produced for the same set of examples by making different initial and subsequent choices for the order in which the attributes were considered, and it was found that some produced deep trees whilst others produced shallow but more branching ones, and these shallow trees were said to be more efficient since decisions were reached more quickly because they were at a higher level. The contribution made by ID3 was the discovery that there were ways to compute a better initial choice of attribute for the root of the tree. The proportion of positive to negative examples for each attribute was to be calculated, the attribute with the highest ratio was selected, and the tree was built recursively reapplying the same principles. There was also a system called windowing to be used if there were very large numbers of examples of training data. This selected a subset of training data to build the initial decision tree. Remaining examples were then classified using the tree. If the tree gave correct classifications for these examples then the initial tree was accepted as the classification tree for all examples. If examples were found that could not be classified by the initial tree then a subset of these were used along with the original subset to compute a new tree and the process repeated until a tree was found to cover all the examples. Quinlan applied this theory to the problem of learning end-of-game chess rules [61].

MACROPS

Another form of learning, MACROPS, is demonstrated by the work of Fikes, Hart and Nilsson [19]. We have previously seen that learning can occur when a system is offered many examples of training data. In future sections of this chapter we shall see that learning is possible from few examples, but Fikes et al were, in a sense, ahead of their time, being able to demonstrate that learning could occur using just a single example in the well-known blocks world. Operators [*Definition* 1.5] in the world define a single action and four such operators are *pick-up-from-table, pick-up-from-block, put-down-on-table* and *put-down-on-block*.

Initial state:	*block B on block C, block A on table*
operator	**blocks**
pick-up-from-block	*B from C*
put-down-on-table	*B*
pick-up-from-table	*C*
put-down-on-block	*C on A*
Goal state:	*block C on block A, block B on table*

Table 5.1: Four Operators from Blocks World Showing the Completion of a Task

For example, if given three blocks A, B and C and a starting condition of block B on top of block C with block A resting on the table, and a goal condition of block C on top of block A, with B resting on the table, a single example for learning achieves the goal state and consists of the actions shown in table 5.1.

The idea was that these actions could be clumped together to form one single procedure as a kind of macro-operator which could be used whenever that particular manoeuvre was required. So the process was to build a plan then learn it. The problem was that learning that particular plan would not help to solve a similar situation with blocks D, E and F. The learned details would be specific to the named blocks in an identical configuration to the initial state. The system in MACROPS was to generalise a successful macro by replacing the constants with suitable variables and defining the preconditions and postconditions of the operation. The definition of the preconditions would say when the macro could be applied, whilst the comparison of the macro's postconditions to any required goal state would determine whether the macro was applicable in the given situation. Thus the system learned from a single example. Other examples in incremental learning can be seen in the work of Wang [80] and Gil [24].

5.2 Machine Learning and Induction of Rules

The machine learning required for the induction of operators is more specific than the generalised topic of machine learning. Two theses written in 1996 by Tim Grant [26], and Xuemei Wang [80], give very good overviews of these specific areas of machine learning.

In [26], pages 50-52, Grant defines machine learning and induction generally, in terms of the available literature and his own Planning Operator Induction (POI) system. He opts for Carbonell's definition of 1989 [9]:

". . . learning can be defined operationally to mean the ability to perform new tasks that could not be performed before or [to] perform old tasks better (faster, more accurately, etc) as a result of changes produced by the learning process."

Grant argues that this definition fits well with his own induction system and since *opmaker* is able to induce methods, and we have shown that in some cases the use of methods produced solutions to planning problems that could not be solved just using operators, we have to agree that this is also a good basic definition for the learning in our *opmaker* system.

In Chapter 7 of Wang's thesis [80] she gives an excellent review of rule learning in structural domains. She categorises four types of learning systems in structural domains as

1. The type of learned knowledge

2. The source of learning

3. The learning algorithm

4. Initial knowledge

and describes her own system, OBSERVER, under these headings. Using this classification we can say, of *opmaker*:

1. *Opmaker* is a concept learning system, and the concepts learned are operators and methods for hierarchical domains.

2. *Opmaker* learns from a partial domain which contains static knowledge about objects in the sorts, predicates, substates and invariants, but no dynamic knowledge in terms of operators. Also required is a named sequence of operators together with the objects they will manipulate (equivalent to a plan-trace in some other systems). Additionally *opmaker*1 requires user given positive examples which determine the intermediate and goal states for the new operators. *Opmaker*2 effectively deduces its own positive examples from the domain.

3. The learning algorithm is a deductive and inductive process for *opmaker*2. *Opmaker*1 was purely inductive.

4. Initial knowledge for learning assumes correct partial domain knowledge including substates, tasks and invariants but no operators or methods.

A good overview of machine learning, and induction in particular, can be found in Shavlik and Dietterich, 1990 [68]. More recently some of the text books on machine learning cover data mining ideas and are not particularly relevant to the subject matter in this thesis, but the reader is also referred to two text books which help to place machine learning in the context of other issues in artificial intelligence [41, 75].

In their survey [89], Zimmerman and Kambhampati present an analysis of research work done in the last thirty years on machine learning, as it relates to planning. They aim to develop a broad perspective of the work done in order to build a projection of

potentially profitable areas for further research. As such this is a useful starting point for new researchers in the field. They identify three phases where learning techniques might be applied: (i) learning and improving a given domain theory (ii) learning during the process of finding a valid plan and (iii) learning during the execution of a plan. For *opmaker*2, which is essentially a mixed initiative approach, the goal is to learn or improve domain theory (phase (i)).

Langley and Simon [36] have classified machine learning into five basic paradigms. One of these is rule induction of certain condition-action rules. Arguably operator induction is a good example of rule induction. They submit that theory revision combines the emphasis, on background knowledge, of analytical methods with the emphasis, on heuristic search, of rule induction. When considering operator induction itself there is little literature available and what there is will be discussed in a later section. The reason for this appears to be the fact that in many knowledge bases for planning, the operators together with the conditions under which they may be applied *are* the knowledge base and, whilst application rules may be induced, the operators must exist first. The case for object centred planning has already been presented in section 2.1.1.

In [22] Garland, Ryall and Rich show, in their Collagen system, that learning task models can be achieved by training examples and support from a domain expert. Their 'task models' are similar to OCL_h methods in the following ways:

- they show a complete recipe to achieve some task
- they show orderings of the steps to achieve the task
- they are developing a graphical user interface to aid construction
- the orderings of the steps contain primitive and non-primitive stages

- they list constraints that apply to the various steps
- user/expert guidance is required for the detail.

The notable differences between the Collagen and GIPO are in what is to be achieved. Collagen develops a task model by first defining a list of actions, though this is not essential, and the list may change over time. A major part of the task model development is defining the hierarchy of the actions. Whilst they agree that in all non-trivial domains identifying the correct set of abstract non-primitive actions is challenging, they do not actually have to go so far as as specifying the operators' details. This is where the learning occurs, using *opmaker* in GIPO. The need in task modelling is only to determine appropriate actions and having done this the expert can train the system by adding to the model definition any variations which would also produce different valid models. By contrast GIPO makes no assumption that actions exist and the user defines, interactively, a sequence of desirable actions to be constructed into both primitive and non-primitive, hierarchical operators.

Collagen assumes operators exist and learns by using positive evidence and the lack of negative evidence in building its task models. *Opmaker* learns by generalising operators from previously induced examples.

5.2.1 Learning from Examples

The usual mechanism for machine learning is the use of positive and negative examples. Whilst it was thought to be advantageous to use both kinds of example, many systems use only one kind of example. In particular Vere's [78] Maximal Unifying Generalisation (1987), and Wang's [80] OBSERVER system learn from just positive examples, whilst Grant's [26] POI system learns from positive examples and uses a default rule to provide negative information which boosts the positive training instances. *Opmaker*2 is similar. It uses positive examples in the initial sequence and

it deduces further examples from the domain substates, tasks and invariants, which serve to restrict the positive examples of the intermediate and final-states for the operators.

5.2.2 Heuristics

Porter and Kibler [59] use Experimental Goal Regression (EGR). With prompts from a user the learning system uses these to form heuristic rules to guide subsequent problems. As in our *opmaker*2 system, the learner can generate examples, but, whereas *opmaker*2 uses domain knowledge and invariants, EGR makes small changes to a training instance supplied by a teacher. Once the learner is able to solve the problems it will then classify the examples as positive and negative.

5.3 Explanation Based Learning (EBL)

In his thesis, Grant [26] differentiates between systems that learn

- Control knowledge e.g. rules
- Domain knowledge e.g. operators
- Plans and plan segments e.g. schemas and subgoal sequences

.

EBL approaches have been used in the above, in particular in learning control knowledge and plans. Notable amongst these are:-

Prodigy [10] Carbonell and Gil's system learns from failures, successes and interactions to derive rules for these.

GERRY [90] learns from constraints in scheduling problems to produce variable ordering heuristics.

Theo-Agent [3] learns rules from planning problems and operators.

Stepping Stone [64] which uses EBL and induction to learn subgoal sequences from goals and operators.

BAGGER [67] which learns schemas or generic plans from operators and a planning problem.

ARMS 1988 [66] which learns by observing a single task performed by an expert and constructing the explanation based on domain theory and the solution. The generalisation of this produces new operator schema. In the sense that learning is from a single example, *opmaker* resembles ARMS.

Nejati, Langley and Konik [56] have a system similar to ARMS [1988] which learns from observing sequences of operators which are experts problem solutions. However in their system hierarchical task networks are learned.

Planning in real-world situations can require necessarily complex planning operators which can be considered as control loops in a much larger system. In their work, Levine and DeJong [39] describe how EBL is applied to the aquisition of operator descriptions in such a planning domain. An 'operator design module' is used to collect information from background knowledge, and the authors illustrate the method with a complex flight simulator domain, where the focus is the aircraft 'takeoff' operator.

5.3.1 Other Techniques

In addition to EBL systems we mention here some other techniques which are relevant to our work. Most learn operators whilst two notable ones learn invariants. Learning invariants could become an idea for future work on our system, since invariants have proved to be useful for some domains in reducing the output of erroneous

operators. Puget's LIFE system, [60], outputs invariants by examining states. When our invariants were constructed (see Figure 4.5), the states and substate sets had to be examined to discover which eventualities were covered by the substates. Those not covered had to be handcrafted accordingly. In their work on invariants, the authors of [54] show that operators need not necessarily be the source from which invariants are derived. They argue that where knowledge does not include the operators they can be 'discovered' by analysing reachable states, rather in the way that generate tracks available substates in *opmaker*2. However their system is different in that it computes factors measuring the potential of states to make a good invariant.

The other systems under this heading learn operators. Two of these systems do so by using the before and after states in much the same way as *opmaker*1, where the initial states were given input, whilst the goal states were prescribed by the domain expert in the examples given to the system. Of these two systems, Diffy-S [34] used constructive induction, whilst THOTH [78] used generalisation, to achieve their results.

One further system, Operator Learner [13], learns from a set of partial operators that are edited by experts. Using induction the system supplies as much of the missing data as possible, e.g. preconditions, and evaluates and monitors user's planning choices. By contrast, *opmaker* begins with no operators but, like Operator Learner, it uses induction.

5.4 A View on Domain Theories

Huffman, Pearson and Laird [32] present a very useful analysis of pre 1992 Explanation-Based Learning systems. Their analysis shows that one of the major problems with EBL systems is the requirement of a complete and correct domain theory for learning to occur. They argue that the construction of any nontrivial domain to this degree

of completeness and correctness is a near impossible task due to the frame problem. Even in the closed-world system, by which everything not specifically stated as true is assumed to be false, construction of domains is notoriously difficult.

They define two complementary tasks performed by EBL systems as analysis and generation tasks.

Analysis tasks involve explaining or understanding some observed example as in Wang "Learning by observation and practice" [79]. In analysis tasks, the system observes a sequence of states ending at a goal and reasons about the observation to infer that a consistent sequence of operations like the one observed will bring about that goal. It will then use such a sequence of states as part of its planning if the same goal is required.

Generation tasks involve constructing (as opposed to observing) a plan. Here the system is given initial and goal states. It may also have to be given any additional details necessary in order to perform planning. The system then plans a sequence of operations leading from initial to goal state.

Both require knowledge of the performable operations in the domain, their applicability and their effects. Thus the analysis task can be seen as a more constrained version of the generation task and the concentration of their argument is on the generation tasks.

The domain theory is seen as simply the system's knowledge of operators, their preconditions and their effects. Each time the system makes an inference it can be viewed as learning an operator. Any additional knowledge within the domain is used to guide the search through the states of the problem space. This additional material, they argue, is not regarded as part of the domain but as part of the search control.

The recent thrust of the research here at Huddersfield has been directed towards

the construction of better domain theories by the development of a tools package aimed at removing most of the difficulty and uncertainty of domain construction. By comparison, in this system, not the operators but the *objects* are the focus of the system's knowledge.

Using GIPO the resulting domains are nearer to the targets of completeness and accuracy, described in *OCL*, an object centred language which can be translated to other description languages. *OCL* concentrates on describing the 'world' from the point of view of the objects in it and the states in which these objects can exist, rather than focusing on the operators as in traditional KB systems.

An Example

An operator causes an action to happen - e.g. "Put the red button in the blue box". In a knowledge base that 'knows' about operators there will be the knowledge that one operator puts the button in the box, plus a range of preconditions such as the button should be red, should be being held, should fit in the box which should be blue etc. There will also be a range of effects held in the knowledge base such as the hand becomes empty or the box becomes full. By contrast in a system that 'knows' about objects, the system knows that the button can be in different states, e.g. it can be red, it can be in the blue box, it can be held, it can be on the table, it can be attached to a coat - the number of different states being, at least in part, determined by other objects in the system like box, hand, table, coat and even non-concrete objects like colour. The operators are induced when an object changes its state - in this case the button changes from the state of being held to the state of being in the box, but the state of being red remains unchanged for the button whilst the box state remains blue. The box changes from the state of being empty to the state of being full.

Table 5.2 shows a comparison between the operator centred and the object centred

approach.

This 'object first' approach allows for the construction of more accurate and complete domain theories in a closed world situation, as it is often easier to envisage the concrete states of objects than the potential results of some action.

Huffman, Pearson and Lairds' view model of pre-1992 Explanation Based Learning systems	Aspects of similarity or difference	Huddersfield University's present system of domain knowledge acquisition
Centred around the operators, their preconditions and effects. Always incomplete due to the frame problem	Knowledge Base	Centred around the objects and their states of existance. Contains some operators and can induce others from the knowledge base. Allows for a more complete knowledge base
Difficult to refine without complete knowledge.	Refinement	Should be easier to refine because the theory is virtually complete - however any 'bugs' may be more difficult to detect.
Difficult to learn from an incomplete domain.	Learning	Yet to be researched - should be easier.
Almost impossible to build accurately and completely.	Domains	Easy to build using tools packages within GIPO.
Huffman, Pearson and Lairds' goal: to develop a universal weak method for domain theory refinement.	Goals	Use induction to produce operators. Use these to refine the domain theory.
Analysis of the refinement task	Required	Analysis of the refinement task is the starting point.
Search control knowledge not regarded as part of the knowledge base since it affects efficiency not correctness. Without good search control more of the search space is involved to solve the problem	Knowledge additional to the knowledge base	Search control not in the knowledge base. Different algorithms can be used to search depending on whether speed or efficiency are most desirable. Without good search control more of the search space is involved to solve the problem
Can affect the accuracy of the results of search so are required in the knowledge base.	Constraints	Can affect the accuracy of the results of search so are required in the knowledge base.

Table 5.2: A Comparison of the Operator Centred and the Object Centred Approach

Types of domain theory imperfections	What our system does
Overgeneral Preconditions. An operator precondition is missing, or an overgeneral test is used (e.g. "fruit" instead of "banana").	Objects are described as existing in certain states or have been left in those states by the action of some previous operator. Lists of allowable 'substates' for all objects are defined in the domain building process and these form the preconditions for the operators.
Overspecific Preconditions An extra, unnecessary precondition is present, overly restricting the set of situations in which the operator is applicable.	Preconditions should always be accurate under the terms of how the domain is defined. If a substate has been described to be over specific then the domain has not been correctly modelled.
Incomplete Postconditions The planner is unaware of some effect of an operator.	The effect of some operator is to change the substate of the object primarily concerned in the operator. However other objects may be less directly concerned and their substates may change too as a result of the operator. Their potential substates should be recorded within the domain.
Extraneous Postconditions The planner incorrectly believes that an operator will produce some effect which it does not.	The substates allowable should always be in the goal state of the operator so effects are predictable from a subset of the allowable substates
Missing operators An entire operator is absent from the domain theory.	Missing operators can be induced from the domain theory given some example input from the user

Table 5.3: Domain Theory Imperfections and the Object Centred Approach

Again an example will help to illustrate the point. Possible effects of the action of closing the door could be causing a temporary draught, it bangs, it breaks, it closes, it hits someone coming through it etc. By considering the states of the door it is easier to narrow the choices - the door can be open or closed.

5.5 An Analysis of the Types of Domain Theory Imperfections

Huffman, Pearson and Laird [32] present an analysis of past research in the area of correcting and extending domain theories by presenting a characterisation of the problem based on the types of knowledge deficiencies present in theories and the types of performance errors that can result.

Table 5.3 contains, on the left, their analysis, and on the right how our system reacts to these problems.

This work had implications for our later work when we concentrated on the refinement of domain theory extended with operators induced by the *opmaker* algorithm.

5.6 Theory Revision

A related work, a PhD thesis by Douglas Pearson[57], details one system of correcting operator preconditions and changing the domain knowledge. This uses techniques similar to the work of Wang [79], Gil [24] and DesJardins [13]. For a review of this topic including EBL in the planning area the reader is also referred to Minton and Zweben [53].

Research has been published from The University of Huddersfield on theory revision using a different context. In this work, "The Automated Refinement of a Requirements Domain Theory", a large air-traffic domain was studied and interpreted into the modelling language. When theory revision was applied to the domain model

several 'bugs' were identified and eliminated in a potentially safety-critical domain [48] [49]. There are differences between this work and ours in that theory revision was mainly numeric whilst our work adds or revises operators - state transitions of objects.

5.7 Induction of Operators

We can track the progress of *opmaker* through our publications at various stages. In [46] we see *opmaker* beginning to induce operators for both flat and hierarchical domains. At this stage some of the operators for hierarchical domains would be wrong because of the inheritance problem. A later publication [47] details how *opmaker*1 was integrated into GIPO. We can see that ideas for merging operator descriptions were emerging in [62] these ideas allow opmaker to handle repetitions of operators. In the next published work [44], we see the introduction of the hybrid hierarchical planner HyHTN, and its integration into GIPO, whilst in [63] the idea that induction sequences need to be carefully chosen by the domain engineer to model separate tasks, begins to emerge. This work also introduced HBC and suggested how hierarchies of methods could be built. The work in [50] details the introduction of *opmaker*2 and includes work detailed in Chapter 4.

For a very detailed and useful survey on the whole topic of operator induction the reader is referred to Xuemei Wang's thesis [80].

Induction of operators is not the only aspect of knowledge engineering when it is applied to planning. Various other types of knowledge can be acquired, such as knowledge of the current state of a domain, knowledge of suitable goal states for the domain and knowledge about the domain model. Operator induction belongs to the latter category.

Until recent years knowledge acquisition has been done manually, but starting with GIPO there has been a growth in systems that acquire the knowledge in a structured shift towards automation. Certainly, it could be argued that the ICAPS Knowledge Engineering Competition has added fuel to this. (The reader will see more on the ICAPS competition in Section 5.8.) *Opmaker*, being a mixed initiative system, has benefited from and contributed to this shift in thinking.

Whilst operator induction is the subject of this thesis, other methods have been applied to learning operators, as we showed earlier in the section on EBL, Section 5.3. Genetic algorithms have also been tried.

We turn our attention again to the closely related research by Tim Grant [26]. In his PhD thesis Grant takes as his motivating theme the simple question, "Where do planning operators come from?".

Under this theme, [26] page 14, he raises the related questions

- How does a developer formulate a set of planning operators for a new domain? Is this difficult? Is it a transferable skill?

- How can the developers (and the users) be sure that the set of planning operators is complete, correct and precise?

- What are the consequences of having incomplete, incorrect, or imprecise domain knowledge? In particular, what happens if the domain knowledge is distributed over multiple agents?

These questions are very relevant to our work and neatly summarise our objectives.

The stance taken by Huffman, Pearson and Laird [32] is that most domains employ an explanation-based learning system that requires a domain knowledge based around

the operators. The operators have to be part of the domain and then plans can be formed and learning can take place. This uses a system of generating 'explanations' for planning attempts then using these to update the knowledge base, but it assumes the presence of the operators in the knowledge base.

By contrast, by asking his question, Tim Grant [26] takes the stance that the operators need not necessarily exist. He cites the example of expert systems where there has been progress in automating the formulation of production rules by induction from examples, and argues that it should also be possible to induce planning operators from examples. This would save the domain development specialist long hours of tedious work.

Grant has developed a system based on Chen's [11] entity relationship modelling notation, which describes the states of the objects around which it is centred and uses the constraints that arise out of this form of modelling to induce operators. The 'examples' used to generate the operators are the observed states.

Our work compares and contrasts with Grant's in the following ways.

- our work is done in the context of an existing knowledge engineering tools environment and benefits from and contributes to this.

- constraints were not present in the sample material given to *opmaker*1. In *opmaker*2 constraints are present in the form of invariants only if they are required.

- examples come from the user using an interactive tool (GIPO) for development in the form of clicking, typing a name and dragging and dropping objects into the newly named operator shell. These 'shells' are built up into an example sequence of desirable operators.

- the development language, OCL, differs from the language Grant developed but remains object based. Differences are minor and are documented in [27].

- operators will be used to refine the domain theory e.g. by the addition of constraints. Here the constraints will come last whereas Grant begins with them to build his operators.

- Grant's work also included inheritance; see his thesis [26] page 105 and section 4.10, pages 155-8.

- using OCL allows for the extension of our system to cover hierarchical domains.

Further work by Tim Grant can be seen later in Section 5.9.1.

Another very useful research was done by Xuemei Wang [79, 80] whose system builds operators incrementally from experimentation and practice. Unlike our system which accepts examples from a non-expert domain builder, in Wang's system the sample material comes from observation of an expert. Whilst training takes place the domain knowledge can be incorrect or incomplete in the ways described by Huffman, Pearson and Laird [32] and therefore planning must be able to take place on this incomplete domain so a system of integrating planner repair, learning and execution is required.

Wang's system [79, 82, 81, 80] OBSERVER, takes, as input, the domain description language and expert solution traces and some initial operators are generated and learned. These can be refined by further observation until the refinement is sufficient to allow planning to take place. OBSERVER is then given practice problems similar to our OCL 'Tasks', and initial plans can be formed using the learned operators. Plans formed are run and result in either successful executions or unsuccessful executions. These can be considered to be the necessary positive and negative examples

required in theory refinement. Unsuccessful executions can be used to repair plans and also used along with successful executions to refine operators. Refined operators are used to build a more complete system of operators and these operators are the eventual output of the system.

There are some obvious similarities to our system, the main difference being that using *opmaker* integrated with GIPO accepts example material from non-experts and uses the restraints of a question and response system to allow non-domain experts to create domains and make and use operators as part of a complete package of domain creation tools. Whilst Wang was limited to flat domains, our recent improvement of the algorithm to generate operators for hierarchical domains makes *opmaker* a useful and credible tool for building domains of any degree of complexity.

5.8 ICAPS 2005 and 2007 Competitions on Knowledge Engineering for Planning and Scheduling

In 2005 the competition was introduced to the International Conference on Automated Planning and Scheduling (ICAPS). This competition had the following aims:

- Promote the knowledge-based and domain modeling aspects of planning and scheduling
- Accelerate knowledge engineering research in AI planning and scheduling
- Encourage the development and sharing of prototype tools or software platforms that promise more rapid, accessible, and effective ways to construct reliable and efficient planning and scheduling systems.

It is interesting to see and compare the work of contemporaries in this field. There were seven entries in each of the 2005 and 2007 competitions, including GIPO itself

in 2005, and in this section we can explore the 2005 entries and one from 2007 in some detail.

1. The argument used by Borrajo et al [6] to support their work is that whilst domain independent planners can be given increased efficiency by the use of heuristics (which may fail in some domains), domain dependent planners require additional control knowledge to be added to the domain to make planning more efficient. This control knowledge can be acquired automatically using a tool that is able to learn search control knowledge and formulate it into control rules. The authors appear to claim that the result of learning this control knowledge builds up macro-operators by using both an EBL system and an inductive-deductive approach. The system uses an explanation trace to find search routes leading to good solutions and these are then formulated into control rules. This is similar to Minton and Zweben [53] although the latter's approach was purely EBL.

   The deductive side of this approach depends on retaining learned rules from all decisions that were effectively the second and third alternatives tried on each occasion and therefore more specific than the first. Each retained rule was then ascribed a utility factor calculated from the time saved by using the rule, the probability that the rule would match, and the matching time cost. An acceptable utility factor could be predefined by the user and rules with a lower utility factor could then be discarded.

2. Like the GIPO system the Tailor tool featured in the work by Jim Blythe and Varun Ratnaker [4], allows for interaction with users who are not KE experts. Tailor allows for end users to modify procedure definitions by offering a set of plausible modifications and demonstrates these in use so that the user may

backtrack if required. The system will then analyse the effect of these modifications and will offer warnings of potential problems and advice for fixing them. However Tailor is a fundamentally different system from GIPO in that it relies on modification of an existing domain rather than having the potential to build a new one. The system was tested by novices using sets of instructions found on the web. However these instructions had to be translated by experts into the correct format for Tailor to accept. Thus we conclude that a novice cannot use Tailor to build a new domain from scratch as is the case with GIPO.

3. NASA contributed to the competition with Patrick Daley et al's work on a debugging system, PlanWorks, for constraint based planning systems [12]. Besides being developed for debugging, PlanWorks also has potential as a knowledge capture system and can also be used as an end-user operations tool. PlanWorks was developed for the constraint based planning system EUROPA$_2$ (Extensible Universal Remote Operations Planning Architecture), developed at NASA. As the planner makes decisions each activity is logged and the unique key is passed down to PlanWorks which uses a MySQL database system back-end to log and keep track of relationships and produce ER-type diagrams enabling users to see the relationships between entities in the plan. The user can use this and other facilities for debugging during planning. These ideas are similar to the use of the stepper in GIPO where the user can manually step through a set task to test the validity of the domain under construction. There are other similarities to GIPO since PlanWorks allows for many different views of the domain under consideration, but the essential difference is that PlanWorks is not primarily a domain construction tool but rather a tool for debugging a domain. However the NASA team claim that the present system is a precursor to a visual model building tool planned for the near future.

4. Stefan Edelkamp and Tilman Mehler [16] have adopted a very similar argument for their development of their planning *ModPlan* workbench to the one used to develop GIPO. Differences lie in the approach as the authors claim that developing a domain model is an iterative process and have designed their acquisition tool around this fact, whereas using GIPO to develop a domain is a largely serial process so long as the modeller is reasonably competent. However the theory revision of the generated operators is still iterative. The other main difference between the two systems is the initial approach since only GIPO is object oriented. A further difference lies in the fact that the final '*product*' has different emphasis. The workbench system is aimed at producing plans by learning domains along the way whereas GIPO produces a largely complete domain 'oven-ready' for planning. *ModPlan* is an integrated environment for domain modelling, static analysis, plan finding and validation and plan visualisation producing domains in a temporal version of PDDL.

5. Vaquero et al's itSIMPLE tool [77] concentrates on producing a system that ultimately creates real world domains that are portable. The authors have spotted an anomaly with the modelling languages used amongst different research groups and industry and tried to redress this using Unified Modelling Language (UML) [14] to build an original model of the domain to be represented. The code is then stored in XML (Extensible Markup Language) files [7] and can be made compatible with PDDL 2.1 [21]. UML is an object oriented modelling language generally represented in diagrammatic form. It allows a visual representation and includes invariant modelling. It is interesting that despite using an object oriented model the authors still concentrate on modelling operators in the STRIPS style, with add and delete lists [[20]]. This is probably because they are aiming to produce models in the main in PDDL so this model only partly bridges the gap between other planning model languages. However since

OCL is translated into PDDL by GIPO this anomaly can be overlooked. There is an argument that the use of XML gives a more modern feel and approach to planning and may give some protection from redundancy of the system as it ages. It certainly allows access by a web browser and many of the most recently developed languages will be able to read its files thus greatly increasing its portability. The itSIMPLE tool will also incorporate the use of Petri Nets [55] which provide a tree model for checking and selecting planning heuristics. This addition will increase compatibility with current manufacturing processes and allow it to be a tool for the next generation of planning. At present the itSIMPLE system is still experimental and is incomplete unlike GIPO which has already reached its third release but the basic ideals of a universal planning language with complete portability might indicate a way toward greater commercial use of planning.

6. Kangheng Wu et al [86] have taken planning on into the realms of automatically discovering action models. The system finds macro-operators for plan completion by using observation of previously successful plans. Their ARMS system learns from gathered knowledge on the statistical distribution of frequent sets of actions in a set of example plans. Using a model built as a propositional satisfiability problem it then uses a SAT solver to learn the action models. In the first phase, the ARMS algorithm finds frequent action sets from plans that share common parameters and it also finds predicate-action relations giving a start point for preconditions and STRIPS style add and delete lists. These can then be used to devise an initial set of constraints which ensure correct planning. In the second phase, the ARMS algorithm converts the constraints into a weighted SAT representation [5]. The solution of this SAT problem produces an action model. Further action models are produced and refined by the iterations of the process.

There are some similarities with the *opmaker* system of incrementally refining operators which make this system interesting. Whereas with *opmaker* we concentrate on refining short operator sequences, here the energies go into refining operator sets. This is comparable to *opmaker*2.0 allowing hierarchical operators to be defined for more complex domains. Of course there are differences too, not least the fact that our operators are produced from complete knowledge and are object oriented rather than in STRIPS format.

7. The final entry to the competition was the Huddersfield University paper on GIPO [71]. In this the author describes the GIPO environment and suggests that GIPO should be useful as a modelling tool irrespective of the final modelling language. The object oriented approach is a very visual way of representing the domain and GIPO as a tool exploits this by offering the knowledge engineer different ways of visualising what she is building. The paper describes the function of many of the tools within GIPO including, new for the third release, an object life history editor. This allows the user to draw state machines and the argument for its inclusion has some similarities to the use of UML in the itSIMPLE tool [77] detailed above in paragraph 5. Also described in this work is the application of *opmaker* for inducing operators which has now been encoded into GIPO. Using *opmaker* the knowledge engineer has to supply a list of names and parameters for actions required to complete a predefined task. Then GIPO uses a series of questions to remove any ambiguities and produces a complete set of operators for the specific task.

   This new version of GIPO also includes tools for engineering HTN planning. There are special tools for validation and planning including a stepper which allows a user to step through an action sequence and to correct any errors to the newly constructed domain. A typical example may be a missing invariant

which allows a faulty operator to be constructed.

Overall this was a very strong field for the competition and the competition itself has promoted interest in this field of research. The winner was the GIPO system developed at Huddersfield University by Ron Simpson.

In the 2007 Planning Competition one author, [29], argues for a change of direction for the competition itself. He defines a 'tough nut' as a domain which can be addressed, language-wise, by existing planning techniques, but cannot be solved efficiently. The argument is for adding a track to the competition in which awards are made

- for the 'tough nut' which survives the longest
- for the first technique which solves a problem

and would keep track of these challenges and solutions. Perhaps this would be a means for resolving problems with our version of Blocks World?

5.9 Further Work in Knowledge Engineering

Model-lite

In their paper on Model-lite Planning [88] the authors detail a system for planning where the domain model is incomplete. Whilst their aims are quite different to ours there are some similarities to our approach. They aim to produce action sequences (plans) by use of a probabilistic model for planning using incomplete domains. Thus both their system and ours begin with lack of operator knowledge. However, in their case the aim is to achieve valid operators from existing incomplete domain models whereas we begin with no operators.

The other interesting facet of Model-lite Planning is the use of invariants to model the domain constraints. The authors use invariants to supply the detail lacking in their operators and these invariants thus enable them to control planning. In our case we have been constructing and using invariants to prune potential intermediate substates for transitions induced by *opmaker*2.

Learning Recursive HTN-Method Structures for Planning

In their machine learning paper the authors [87] tackle the issue of learning HTN methods by looking at three problems:-

- acquiring logical relationships between high-level tasks and low level actions (i.e. task descriptions and available operators)
- learning pre and post conditions of primitives in STRIPS or PDDL action models
- acquiring decomposition structures of HTN methods from observed action traces from input of multiple action sequences for multiple tasks.

The authors focus on matching sub-sequences to tasks assuming no knowledge of observed states achieved by low-level actions. The output consists of pairs of action sequences and the high-level tasks achieved by them. As with our system they begin with action sequences and defined tasks (ours are defined in terms of initial states and goal states). Unlike ourselves they do not use lists of potentially available substates. The first phase is matching sub-sequences of actions from examples to tasks, using probability and hand tuning. In the second phase sets of recursive methods are learnt from these with the focus on learning the decomposition of the methods. They assess their success by comparing learnt methods to hand-crafted ones. The system only required about 40 training sets. Once learned they were fine tuned by domain

experts by hand. By contrast our system induces both primitives and a method using substate lists and invariants and requires only an initial action sequence. The induced primitives form the decomposition of the method.

CaMeL

The authors of Learning Preconditions for Planning [33] describe their *CaMeL* system which aims to aid domain building by learning the preconditions from HTN methods. They present a proof theorem about *CaMeL*'s soundness, completeness and convergence, and show some empirical results. These claim that *CaMeL* converges fastest on HTN's that are needed most often - thus enabling it to be useful even before convergence.

In their paper [31] the authors share our argument that the major hurdle for HTN planning use is the acquisition of the HTN domain descriptions. They present a compelling argument that encoding 'task models', which contain knowledge about how to decompose tasks into subtasks is really difficult and time consuming. Their solution is $HTN - MAKER$ (Hierarchical Task Networks with Minimal Additional Knowledge Engineering Required) an off-line incremental algorithm for learning task models. Learning is based on the input of a STRIPS domain model, a STRIPS plan and task definition. In an upward manner as variables are incrementally substituted, it becomes possible to learn a set of methods, where the first encapsulates the previous operator, the next the two previous operators and so on. In this way the methods that are shortest and simplest are learned first and these can then be learned and used as subtasks in methods that encapsulate longer sections of the plan. This idea has some similarities with our own thinking. These are

- The CaMeL team agree that acquisition of HTN planning domain descriptions is a major hurdle for HTN planning.

- They build methods with a bottom up approach.

Semantics for High Level Actions

In their paper [42] Marthi et al argue for a system of high-level actions (HLA's) which can be refined downwards into simple action sequences in many ways, following the lines of human mind planning where a high level action would be the first choice and as planning proceeds the details of simpler actions begin to emerge.

Their starting point seems to be the knowledge of simple (primitive) actions and some high-level 'desires'. This system works in a top-down forward search way, searching for a suitable action sequence to define the HLA. Our system, by contrast, begins with no actions but a human designed action sequence which translates into primitive action schema and a single high level action which has, as its decomposition, the initial action sequence. Their system is illustrated by an interesting version of blocks world which used a variant of a STRIPS-like symbolism. The description of the workings of the domain suggest a 1 dimensional table the width of a block and of set length upon which the blocks are stacked by a suspended gripper. The gripper can only pick up a block from the right or left sides and to effect a pick-up the gripper must be facing the block. The blocks are stacked on the table and on each other at various levels. However, the gripper can only be turned if it is above the level of the blocks. Thus this version of the blocks world has many actions associated with manipulating the gripper such as raising, lowering, moving, turning right or left, and picking up and putting down the blocks. These additions allow for much longer plans to be formed for relatively few objects and the authors include an apparently simple three blocks problem which requires 50 steps to solve. Further work on our system could well include the ideas in this domain which will extend plan steps quite quickly to large numbers.

Using their implemented algorithm the authors are able to demonstrate consistent improvement in running time as stages in the process are added for problems requiring up to 90 steps in planning.

5.9.1 Very Recent Publications

Two very recent publications (December 2007) support our work directly and are very closely allied to it. In the first of these [28] we look at another interactive domain editing and planning system for building domains which delivers domain descriptions in PDDL. In the second [27] a system of inducing operators by relying on the use of invariants is the subject of some interesting work in a multi-agent system.

VLEPpO

The *VLEPpO* (Visual Language for Enhanced Planning problem Orchestration) domain modelling system [28] was developed in response to a need to have a similar system to GIPO for building domains modelled in PDDL. The authors express the view that a visual tool similar to the graphical life history editor in GIPO is needed to help domain builders build domains in the latest versions of PDDL without the need for familiarity with either PDDL or OCL and without having to translate between them. There are fundamental differences in the structure of the languages which mean that direct translation is effective only into the more basic versions of PDDL from OCL which does not yet handle temporal planning for example. *VLEPpO* also allows the user to plan by using web services. In a later stage of development it is planned to develop an extention into HTN planning. We welcome the introduction of *VLEPpO* and will watch with interest how it develops.

POI

In his work in the past [26] Grant has shown how he can induce operators from a knowledge of inconsistent constraints. In this latest work [27] he shows how this system, Planning Operator Induction (POI), extends to a multi-agent system. The work is based on representations of operators and constraints which between them model the domains so the modelling process is fundamentally different from ours. In this work we assume there is complete domain knowledge and known initial and goal states. Planning is, however, made more complex by the fact that the agents initially do not have complete knowledge and have to share parts of their own knowledge with other agents before planning can take place effectively. In this context planning requires the application of machine learning techniques to the acquisition of planning operators. It also includes an element of Model-lite planning [88], and requires knowledge sharing concepts. The author presents a good assessment and diagrammatic model of planning in this context where an initially complete domain model is shown to be capable of receiving and assimilating sensory feedback. Because this initial domain model is distributed across several agents who, as a set, have complete knowledge, individual agents will have only partial knowledge and must share this knowledge for planning to be successful. The emphasis in [27] is on how the recipient agent assimilates the knowledge another agent has given it into its own knowledge.

5.10 Summary

All this recent literature shows many attempts to make computers simulate tasks the human brain is already good at. In their work [31] the authors show how humans *learn* by tackling simple tasks and gradually building up to move onto more complex ones. By contrast, when *planning* the authors of [42], Semantics for High-level Actions, say that humans plan top down, by looking at the overall task first, which is split down into simpler tasks and eventually into simple operations. *Opmaker* builds operators

first then constructs methods from the newly induced operators. In planning the HyHTN [44] planning algorithm in GIPO uses methods, where they are available, in preference to operators, so priority is given to chunking in the way that humans plan.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Limitations of this Research

This work contains a number of assumptions and limitations to restrict the scope and scale of the research which have been necessary to identify the specific project. These are:-

1. The work has only been applied to planning domains constructed in the *OCL* planning language which enforces an object based approach.

2. The work includes action modelling of 'instantaneous actions' only. Durative actions or those depending on resource availability have not been modelled.

3. Planners used have been restricted to Hoffmann's FF [30] and HyHTN [44].

4. Objects modelled have been concrete items rather than abstract concepts such as jobs or marital status.

5. The work reported depends on the availability of at least a partial domain with which to work. Often we begin with a full model, in which case operators contained in the full model can be used for comparison purposes.

6. We assume that the space of states is restricted in that objects are pre-conceived to be a fixed set of plausible states.

7. Some knowledge engineering problems remain - more work is required to use the acquired methods in an HTN network for larger domains.

6.2 Summary

This thesis has studied how knowledge engineering is used to construct knowledge bases for planning. Different planning domain languages are used for this and up till now the knowledge engineer has needed to be an expert in the particular language she uses. However, this in itself is a barrier to the wider use of planning since it takes a long time to become an expert. Additionally, in the past, planning domains have taken a lot of time to construct because of the high degree of accuracy required especially when constructing the operators. The aim of this work, specified in Section 1.9, has been to increase the efficiency of domain construction by the automatic induction of planning domain operators using an object based knowledge capture system, and making particular use of the knowledge already contained.

We have shown that *opmaker*1.0 was capable of inducing operators for domains with flat sort structures and could, additionally, induce a single method for every input sequence. Initially this system was error-prone because the examples required for the induction were hand coded and erroneous material could be used to create meaningless operators. For example before the inheritance problem was solved it was possible to induce operators with empty prevail or necessary transitions. Our system of checking induced operators relied on people who themselves can make errors.

One of the advantages of this system being embedded in GIPO was that the constraints of working with an editor reduce the chance of introducing erroneous data. When *opmaker*1 was embedded into GIPO this represented a big step forward because GIPO contained many validation checks to screen out most errors. At that stage, however, GIPO still required some knowledge of planning from the engineer. Now GIPO has a graphical 'Life History' editor which allows for much greater abstraction and students of artificial intelligence can, with a little instruction, create new domains without ever seeing the *OCL* language. Basically this means that GIPO is getting

towards its target of allowing non-experts to use planning as a tool.

With the new *opmaker*2.0 system, operator creation becomes one step easier too. The hand-coding of the example material was an area likely to introduce bugs into the system. *Opmaker*2.0, with its automatic generation of the intermediate states, ruled out these kinds of errors. Some engineering problems remain. The system still needs to be integrated into GIPO. There are a few factors that need further consideration. Whilst we can choose the initial sequences carefully to induce meaningful methods, more work is required to learn and use those methods when building up the HTN network for larger domains. We need to implement *opmaker*2.0 in the Life History editor of GIPO which, as yet, only engineers flat domains.

Despite these problems, we have shown the great power of a system able to create, learn and use its own planning operators. More than this when considering automatic operator induction using GIPO there is the potential to engineer, validate and use HTN planning without the expert knowledge that was once mandatory for the engineer. An autonomous system that plans and creates its own operators, perhaps working in some remote area, is close to reality.

This thesis began by examining the area of artificial intelligence into which this work fits. We saw, in Chapter 1, how knowledge engineering is essential to the field of AI by creating the database on which it is based. Planning is seen as a challenging problem and accurate representation of the planning domain and in particular the operators is essential to enable planning to be accurate. In the discussion of the 'knowledge acquisition bottleneck' [18], we saw the difficulty of hand-constructing operators to reasonable time-scales. The chapter also gives several definitions which would be required throughout the thesis. There was a discussion of the process of induction of operators, which model actions in the domains. *Opmaker* was introduced. The end of this chapter showed the aim of this research and discussed the contributions it makes to planning knowledge.

Chapter 2 was concerned with explaining to the reader how a detailed domain is constructed in the *OCL* language. We saw, in various steps, how the static knowledge for the knowledge base is engineered both by hand and by using GIPO. Several domains were introduced in the chapter and we saw how to use some of the construction tools in GIPO to construct domain parts like sort trees, atomic invariants and operators. Later in the same chapter we saw how to use *opmaker*1 in GIPO to construct an operator sequence and we saw the dialogue that GIPO begins with the user to resolve any intermediate state conflicts. In later chapters we would show that *opmaker*2 was capable of resolving these conflicts by drawing on domain knowledge.

In Chapters 3 and 4 we saw the two distinct stages of *opmaker* development. Chapter 3 gives an overview of the *opmaker* algorithm and also the algorithm in more detail. It explains how well it worked and gives some experimental results before going on to explain how these were tarnished, for domains with hierarchical sort structures, by the inheritance problem. In Chapter 4 we discuss the reasons for developing *opmaker* further and again give an algorithm. We show how this might work with reference to a tyre domain which serves as an example throughout the chapter. Finally we discuss experimental results.

Chapter 5 discusses literature and work done in this area of planning. It begins with some of the recent 'historical' work showing how induction of operators arose. Later it shows how we can categorise our system of operator induction in terms of the other work which has been done. The chapter concludes with some very recent work and reports on the achievements of those taking part in the bi-annual Planning Competition.

6.3 Contributions

Contributions are detailed in Chapter 1, Section 1.10. Here we summarise where these contributions are to be found bearing in mind the aims of this research stated in Section 1.9.

1. **Contribution 1 - Induction of Hierarchical Models** is demonstrated in Chapters 3 and 4 where we show how methods can be induced from the initial sequences given to the system. We show that a sensible choice of action sequences helps to build hierarchical operators which can be used effectively in planning for the completion of whole tasks.

2. **Contribution 2 - Evidence of Efficiency of Hierarchical Models** is demonstrated in Chapters 3 and 4. We show that development time for operators and methods is minimised by the induction process and this is particularly the case when the *opmaker*2.0 system is used and user input is greatly reduced. We see, in Chapter 3, that hierarchical systems do not always produce faster plan times but in Chapter 4, Table 4.3 we show that as the plans get longer and the number of objects in the domains increase then they become more efficient.

3. **Contribution 3 - Towards True Agent Autonomy** is demonstrated in the second half of Chapter 4 where the idea of complete autonomy is discussed in a system which generates its own intermediate states using heuristics and invariants, and uses its knowledge to output accurate HTN methods for each novice or expert defined input sequence. The desirability of a remote agent acquiring its own planning knowledge with user input reduced to a minimum is also discussed.

4. **Contribution 4 - New Versions of Experimental Domain Knowledge** are shown in Chapters 2, 3 and 4 where we discuss new versions of older domains

- the Hierarchical Briefcase Domain (Chapter 3) and the Extended Tyre Domain (Chapter 4). Chapter 2 also discusses a relatively new experimental domain in the Hiking Domain, introduced in 2001 [46].

6.4 Further Work

Suggestions for further work include the following.

1. Since a set of operators can be induced to fit the demands of a chosen task, then in that sense we can say they are a complete set for that particular problem. However another task may require some different operators for its completion. We could argue that the space of unknown tasks is unknown and, therefore, there is no such thing as a complete set of operators for a domain. However, the beauty of a system such as *opmaker*2 is that so long as the new task can be modelled, and the expert can compose a sequence to complete the task, then a further set of operators for that task can be induced. This argument could be strengthened if we could say that any necessary invariants could be constructed and added dynamically to the static domain knowledge. This idea gives a pointer for a future direction.

2. When using HyHTN, the hierarchical OCL planner, there seems to be a crossover point (about 12 steps) beyond which hierarchical operators produce faster plan times than non-hierarchical ones, whether or not the domains used contained induced operators and methods as opposed to hand-crafted ones. Currently our evidence is that hierarchical domains do produce faster plan times, but this area requires further investigation and could make a useful project for research.

3. This work could be extended to capturing domains with durative actions, or other, more expressive formulations for action.

4. The theoretical limitations on the content of the methods that are created from Opmaker2, as compared to hand-crafted HTN operators, should be investigated.

5. The Opmaker2 system should be extended to deal with model maintenance, so that old operator schema can be refined in the presence of new example solution sequences.

6. A rigorous investigation should be conducted into the resilience of our approach in the face of errors in training tasks or in the partial domain model.

7. Investigations should be conducted into whether

   - This work will extend to the operator centred rather than the object centred approach.
   - This work can be replicated using other forms of machine learning.
   - The work offers a forum for planning with uncertainty or incomplete information.
   - The chunking of operators can be automated.

8. The evaluation of *opmaker*1, currently available in GIPO, should be done using groups of non-experts (students) to create flat domains in different ways, such as:-

   - By hand
   - Using the operator constructor in GIPO
   - Using the *opmaker* tool embedded in GIPO
   - Using the graphical life-history editor in GIPO

9. The embedding of the hierarchical induction process into GIPO is a clear requirement for the completion of the GIPO tool set.

10. The induction of methods for the Translog Domain including testing and evaluation would give useful confirmation of current results.

11. A collaborative system where two agents induce methods and act together to solve a single plan would offer a challenge to teams taking the work further.

12. Further analysis of results using *opmaker*2 is required. These results could be shown graphically or in a table similar to Table 4.2.

13. Analysis of which sorts of domain are suitable to be HTN should be attempted, bearing in mind we have shown conditional domains to be problematic and domains where there are many instances of the same sort in an operator as in blocks world.

14. It would be of interest to use GIPO for the construction and use of the unusual version of blocks detailed in Section 5.9 [42], i.e. in a very complex domain. *Opmaker*2 could then induce more complex operators and methods and results of experimentation on this domain would make a useful addition to planning.

Appendix A

A FULL CODING OF THE VERSION OF THE HIKING DOMAIN BUILT USING GIPO

Automatically generated OCl Domain from GIPO Version 1.0

Author: scomner Institution: University of Huddersfield Date created: Tue Jul 22 15:01:56 BST 2003 Date last modified: Description:GIPO constructed domain for thesis using fit and tired.

```
domain_name(newhiking).

% Sorts
sorts(primitive_sorts,[car,person,tent,place,couple]).

% Objects
objects(car,[car1,car2]).
objects(person,[sue,fred]).
objects(tent,[tent1]).
objects(place,[keswick,helvelyn,fairfield,honister,derwent]).
objects(couple,[couple1]).

% Predicates
predicates([
```

```
    at_tent(tent,place),
    at_person(person,place),
    at_car(car,place),
    partners(couple,person,person),
    tired(person),
    fit(person),
    up(tent),
    down(tent),
    walked(couple,place),
    next(place,place)]).

% Object Class Definitions
substate_classes(car,Car,[
    [at_car(Car,Place)]]).
substate_classes(tent,Tent,[
    [at_tent(Tent,Place),up(Tent)],
    [at_tent(Tent,Place),down(Tent)]]).
substate_classes(person,Person,[
    [at_person(Person,Place),fit(Person)],
    [at_person(Person,Place),tired(Person)]]).
substate_classes(couple,Couple,[
    [walked(Couple,Place)]]).

% Atomic Invariants
atomic_invariants([
    next(keswick,helvelyn),
    next(helvelyn,fairfield),
    next(fairfield,honister),
    next(honister,derwent),
    partners(couple1,sue,fred)]).

% Implied Invariants
```

```
% Inconsistent Constraints

% Operators
operator(take_down(Person0,Tent0,Place0),
    % prevail
    [     se(person,Person0,[at_person(Person0,Place0),fit(Person0)])],
    % necessary
    [
sc(tent,Tent0,[at_tent(Tent0,Place0),up(Tent0)]=>[at_tent(Tent0,Place0),down(Ten
t0)])],
    % conditional
    []).
operator(drive_tent(Person0,Tent0,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),fit(Person0),next(Place0,Place1)]=>
[at_person(Person0,Place1),fit(Person0)]),

sc(tent,Tent0,[at_tent(Tent0,Place0),down(Tent0),next(Place0,Place1)]=>[at_tent(
Tent0,Place1),down(Tent0)]),

sc(car,Car0,[at_car(Car0,Place0),next(Place0,Place1)]=>[at_car(Car0,Place1)])],
    % conditional
    []).
operator(drive(Person0,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),fit(Person0),next(Place0,Place1)]=>
[at_person(Person0,Place1),fit(Person0)]),
```

```
sc(car,Car0,[at_car(Car0,Place0),next(Place0,Place1)]=>[at_car(Car0,Place1)])],
    % conditional
    []).
operator(put_up(Person0,Tent0,Place0),
    % prevail
    [     se(person,Person0,[at_person(Person0,Place0),fit(Person0)])],
    % necessary
    [
sc(tent,Tent0,[at_tent(Tent0,Place0),down(Tent0)]=>[at_tent(Tent0,Place0),up(Ten
t0)])],
    % conditional
    []).
operator(drive_passenger(Person0,Person1,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),fit(Person0),ne(Person0,Person1),ne
xt(Place1,Place0)]=>[at_person(Person0,Place1),fit(Person0)]),

sc(person,Person1,[at_person(Person1,Place0),fit(Person1),next(Place1,Place0)]=>
[at_person(Person1,Place1),fit(Person1)]),

sc(car,Car0,[at_car(Car0,Place0),next(Place1,Place0)]=>[at_car(Car0,Place1)])],
    % conditional
    []).
operator(walk_together(Person0,Person1,Tent0,Couple0,Place0,Place1),
    % prevail
    [     se(tent,Tent0,[at_tent(Tent0,Place1),up(Tent0)])],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),fit(Person0),ne(Person0,Person1),ne
```

```
xt(Place0,Place1)]=>[at_person(Person0,Place1),tired(Person0)]),

sc(person,Person1,[at_person(Person1,Place0),fit(Person1),next(Place0,Place1)]=>
[at_person(Person1,Place1),tired(Person1)]),

sc(couple,Couple0,[walked(Couple0,Place0),next(Place0,Place1)]=>[walked(Couple0,
Place1)])],
    % conditional
    []).
operator(sleep_couple(Person0,Person1,Tent0,Place0),
    % prevail
    [     se(tent,Tent0,[at_tent(Tent0,Place0),up(Tent0)])],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),tired(Person0),ne(Person0,Person1)]
=>[at_person(Person0,Place0),fit(Person0)]),

sc(person,Person1,[at_person(Person1,Place0),tired(Person1)]=>[at_person(Person1
,Place0),fit(Person1)])],
    % conditional
    []).
operator(drive_tent_passenger(Person0,Person1,Tent0,Place0,Place1,Car0),
    % prevail
    [],
    % necessary
    [
sc(person,Person0,[at_person(Person0,Place0),fit(Person0),ne(Person0,Person1),ne
xt(Place1,Place0)]=>[at_person(Person0,Place1),fit(Person0)]),

sc(person,Person1,[at_person(Person1,Place0),fit(Person1),next(Place1,Place0)]=>
[at_person(Person1,Place1),fit(Person1)]),

sc(tent,Tent0,[at_tent(Tent0,Place0),down(Tent0),next(Place1,Place0)]=>[at_tent(
```

```
Tent0,Place1),down(Tent0)]),

sc(car,Car0,[at_car(Car0,Place0),next(Place1,Place0)]=>[at_car(Car0,Place1)])],
    % conditional
    []).

% Methods

% Domain Tasks
planner_task(1,
    % Goals
    [
     se(couple,couple1,[walked(couple1,helvelyn)]),
     se(car,car1,[at_car(car1,helvelyn)]),
     se(tent,tent1,[at_tent(tent1,helvelyn),up(tent1)]),
     se(car,car2,[at_car(car2,keswick)]),
     se(person,sue,[at_person(sue,helvelyn),tired(sue)]),
     se(person,fred,[at_person(fred,helvelyn),tired(fred)])],
    % INIT States
    [
     ss(car,car1,[at_car(car1,keswick)]),
     ss(car,car2,[at_car(car2,keswick)]),
     ss(person,sue,[at_person(sue,keswick),fit(sue)]),
     ss(person,fred,[at_person(fred,keswick),fit(fred)]),
     ss(tent,tent1,[at_tent(tent1,keswick),up(tent1)]),
     ss(couple,couple1,[walked(couple1,keswick)])]).
planner_task(2,
    % Goals
    [
     se(couple,couple1,[walked(couple1,helvelyn)]),
     se(car,car1,[at_car(car1,helvelyn)]),
     se(car,car2,[at_car(car2,helvelyn)]),
     se(tent,tent1,[at_tent(tent1,helvelyn),down(tent1)]),
```

```
 se(person,sue,[at_person(sue,helvelyn),fit(sue)]),
 se(person,fred,[at_person(fred,helvelyn),fit(fred)])],
% INIT States
[
 ss(couple,couple1,[walked(couple1,helvelyn)]),
 ss(car,car1,[at_car(car1,helvelyn)]),
 ss(tent,tent1,[at_tent(tent1,helvelyn),up(tent1)]),
 ss(car,car2,[at_car(car2,keswick)]),
 ss(person,sue,[at_person(sue,helvelyn),tired(sue)]),
 ss(person,fred,[at_person(fred,helvelyn),tired(fred)])]).
```

Appendix B

A TEST FILE FROM THE HIKING DOMAIN

```
% This code is a typical test file designed to run with the induction code which
% includes the $opmaker$ algorithm. It consists of an action sequence, a set of
% initial states for the objects, example training material and a partial
% domain in which the substates are in single set format. Added to the end is
% the output given by the induction process.

:- multifile input/3.
:- dynamic input/3.
:- multifile planner_task/3.
:- dynamic planner_task/3.

% ACTION SEQUENCE

t1 :- sequops([
putdown(tent1,fred,keswick),
load(fred,tent1,car1,keswick),
getin(sue,keswick,car1),
drive(sue,car1,tent1,keswick,helvelyn)
]).

htn(move_tent).

% INITIAL STATES

planner_task(_,_,
  [
```

```
     ss(car,car1,[at(car1,keswick)]),
     ss(car,car2,[at(car2,keswick)]),
     ss(couple,couple1,[walked(couple1,keswick)]),
     ss(person,sue,[fit(sue,keswick)]),
     ss(person,fred,[fit(fred,keswick)]),
     ss(tent,tent1,[up(tent1,keswick)])
]).

% EXAMPLE INPUTS

% putdown(tent1,fred,keswick)
input(1,tent1,sclass(Tent,tent,[down(Tent,Place)])).
input(1,fred,null).

% load(fred,tent1,car1,keswick),
input(2,fred,null).
input(2,tent1,sclass(Tent,tent,[loaded(Tent,Car,Place)])).
input(2,car1,null).

% getin(sue,keswick,car1),
input(3,sue,sclass(Person,person,[in(Person,Car,Place)])).
input(3,car1,null).

% drive(sue,car1,tent1,keswick,helvelyn),
input(4,sue,sclass(Person,person,[in(Person,Car,Place)])).
input(4,car1,sclass(Car,car,[at(Car,Place)])).
input(4,tent1,sclass(Tent,tent,[loaded(Tent,Car,Place)])).

% PARTIAL DOMAIN DESCRIPTION

domain_name(hiking).
```

```
% Sorts
sorts(primitive_sorts,[car,person,tent,place,couple]).

% Objects
objects(car,[car1,car2]).
objects(tent,[tent1]).
objects(person,[sue,fred]).
objects(couple,[couple1]).
objects(place,[keswick,helvelyn,fairfield,honister,derwent]).


% Predicates
predicates([
    up(tent,place),
    down(tent,place),
    loaded(tent,car,place),
    in(person,car,place),
    fit(person,place),
    tired(person,place),
    at(car,place),
    partners(couple,person,person),
    walked(couple,place),
    next(place,place)]).

% Object Class Definitions
substate_classes([

person(Person,
[
     [tired(Person,Place)],
     [fit(Person,Place)],
      [in(Person,Car,Place)]
  ]),
```

```
couple(Couple,
[
[walked(Couple,Place),
 partners(Couple,Person1,Person2)]
]),

tent(Tent,
[
[up(Tent,Place)],
     [down(Tent,Place)],
     [loaded(Tent,Car,Place)]
     ]),

car(Car,
[
     [at(Car,Place)]
   ])
]).

% Atomic Invariants
atomic_invariants([
    partners(couple1,sue,fred),
    next(keswick,helvelyn),
    next(helvelyn,fairfield),
    next(fairfield,honister),
    next(honister,derwent)]).

% OUTPUT GIVEN BY INDUCTION PROCESS
/*
states created..

operator(putdown(Tent1,Fred,Keswick),
```

```
[se(person,Fred,[fit(Fred,Keswick)])],
[sc(tent,Tent1,[up(Tent1,Keswick)] => [down(Tent1,Keswick)])],
[]
).

operator(load(Fred,Tent1,Car1,Keswick),
[se(person,Fred,[fit(Fred,Keswick)]),
se(car,Car1,[at(Car1,Keswick)])],
[sc(tent,Tent1,[down(Tent1,Keswick)] => [loaded(Tent1,Car1,Keswick)])],
[]
).

operator(getin(Sue,Keswick,Car1),
[se(car,Car1,[at(Car1,Keswick)])],
[sc(person,Sue,[fit(Sue,Keswick)] => [in(Sue,Car1,Keswick)])],
[]
).

operator(drive(Sue,Car1,Tent1,Keswick,Helvelyn),
[],
[sc(person,Sue,[in(Sue,Car1,Keswick),next(Keswick,Helvelyn)] =>
[in(Sue,Car1,Helvelyn)]),
sc(car,Car1,[at(Car1,Keswick)] => [at(Car1,Helvelyn)]),
sc(tent,Tent1,[loaded(Tent1,Car1,Keswick)] => [loaded(Tent1,Car1,Helvelyn)])],
[]
).


% name
method(move_tent(Fred,Sue,Car1,Tent1),
% dynamic constraints
[se(person,Fred,[fit(Fred,Keswick)]),
se(person,Fred,[fit(Fred,Keswick)])],
```

```
% list of necessary transitions
[sc(person,Sue,[fit(Sue,Keswick)] => [in(Sue,Car1,Helvelyn)]),
sc(car,Car1,[at(Car1,Keswick)] => [at(Car1,Helvelyn)]),
sc(tent,Tent1,[up(Tent1,Keswick)] => [loaded(Tent1,Car1,Helvelyn)])],
% static constraints
[  next(Keswick,Helvelyn)],
% temporal  constraints
[before(1,2),before(2,3),before(3,4)],
%  decomposition
[  putdown(Tent1,Fred,Keswick),
  load(Fred,Tent1,Car1,Keswick),
  getin(Sue,Keswick,Car1),
  drive(Sue,Car1,Tent1,Keswick,Helvelyn)]
).
*/
```

Appendix C

FULL LISTING OF THE HIERARCHICAL BRIEFCASE DOMAIN (HBC)AS DEVELOPED USING GIPO

```
/**
 *  All rights reserved. Use of this software is permitted for non-commercial
 *  research purposes, and it may be copied only for that use.  All copies must
 *  include this copyright message.  This software is made available AS IS, and
 *  neither the GIPO team nor the University of Huddersfield make any warranty
 *  about the software or its performance.
 *
 *  Automatically generated OCL Domain from  GIPO Version 2.0
 *
 *  Author: Beth Richardson
 *  Institution: University of Huddersfield
 *  Date created: Fri Oct 07 14:19:51 BoxST 2005
 *  Date last modified: 2006/10/04 at 12:46:18 PM BST
 *  Description:
 *    Briefcase world has additional containers - a lunch box and a pencil box.
 *    These fit in the briefcase and themselves can contain, respectively,
 *    sandwiches and a pencil. Additional activities include
 *    pack and unpack lunch, pack and take an object to work and pack suit.
 */

domain_name(hier_briefcase).

option(hierarchical).

% Sorts
```

```
sorts(primitive_sorts,[briefcase,suitcase,lunch_box,pencil_box,thing,place]).
sorts(carrier,[bag,box]).
sorts(bag,[briefcase,suitcase]).
sorts(box,[lunch_box,pencil_box]).

% Objects
objects(briefcase,[bc1]).
objects(suitcase,[sc1]).
objects(lunch_box,[lb1]).
objects(pencil_box,[pb1]).
objects(thing,[cheque,suit,dictionary,sandwiches,pencil]).
objects(place,[home,office]).

% Predicates
predicates([
    at_thing(thing,place),
    outside(thing),
    at_carrier(carrier,place),
    in_bag(thing,bag),
    in_box(thing,box),
    box_in_bag(box,bag),
    box_outside(box),
    fits_in(thing,bag),
    safe_in(thing,box),
    goes_in(box,bag)]).

% Object Class Definitions
substate_classes(carrier,Carrier,[
    [at_carrier(Carrier,Place)]]).
substate_classes(thing,Thing,[
    [outside(Thing),at_thing(Thing,Place)],
    [in_bag(Thing,Bag),at_thing(Thing,Place)],
    [in_box(Thing,Box),at_thing(Thing,Place)]]).
```

```
substate_classes(box,Box,[
    [box_in_bag(Box,Bag)],
    [box_outside(Box)]]).

% Atomic Invariants
atomic_invariants([
    fits_in(cheque,bc1),
    fits_in(dictionary,bc1),
    fits_in(suit,sc1),
    fits_in(cheque,sc1),
    fits_in(dictionary,sc1),
    goes_in(lb1,bc1),
    goes_in(pb1,bc1),
    safe_in(pencil,pb1),
    safe_in(sandwiches,lb1)]).

% Implied Invariants
implied_invariant([in_box(Thing,Box),box_in_bag(Box,Bag),at_carrier(Bag,Place)],
[at_thing(Thing,Place),at_carrier(Box,Place)]).
implied_invariant([in_box(Thing,Box),box_in_bag(Box,Bag)],[in_bag(Thing,Bag)]).

% Inconsistent Constraints
inconsistent_constraint([at_thing(Thing,Place),at_thing(Thing,Place1),ne(Place1,
Place)]).
inconsistent_constraint([at_carrier(Carrier,Place),at_carrier(Carrier,Place1),ne
(Place1,Place)]).
inconsistent_constraint([in_bag(Thing,Bag),in_bag(Thing,Bag1),ne(Bag1,Bag)]).
inconsistent_constraint([in_box(Thing,Box),in_box(Thing,Box1),ne(Box1,Box)]).
inconsistent_constraint([in_bag(Thing,Bag),outside(Thing)]).
inconsistent_constraint([in_box(Thing,Box),outside(Thing)]).
inconsistent_constraint([in_bag(Thing,Bag),at_carrier(Carrier,Place),at_thing(Th
ing,Place1),ne(Place1,Place)]).
inconsistent_constraint([in_box(Thing,Box),at_carrier(Carrier,Place),at_thing(Th
```

```
ing,Place1),ne(Place1,Place)]).
inconsistent_constraint([outside(Thing),fits_in(Thing,Bag)]).
inconsistent_constraint([outside(Thing),safe_in(Thing,Box)]).
inconsistent_constraint([box_outside(Box),goes_in(Box,Bag)]).

% Operators
operator(put_in_box(Box,Place,Thing),
    % prevail
    [     se(box,Box,[box_outside(Box),at_carrier(Box,Place)])],
    % necessary
    [
sc(thing,Thing,[outside(Thing),at_thing(Thing,Place)]=>[in_box(Thing,Box),at_thi
ng(Thing,Place),safe_in(Thing,Box)])],
    % conditional
    []).

operator(put_box_in_bag(Bag,Place,Box),
    % prevail
    [     se(bag,Bag,[at_carrier(Bag,Place)])],
    % necessary
    [
sc(box,Box,[box_outside(Box),at_carrier(Box,Place)]=>[box_in_bag(Box,Bag),at_car
rier(Box,Place),goes_in(Box,Bag)])],
    % conditional
    [
sc(thing,Thing,[in_box(Thing,Box),at_thing(Thing,Place)]=>[in_box(Thing,Box),at_
thing(Thing,Place),safe_in(Thing,Box)])]).

operator(put_thing_in_bag(Bag,Place,Thing),
    % prevail
    [     se(carrier,Bag,[at_carrier(Bag,Place)])],
    % necessary
    [
```

```
sc(thing,Thing,[outside(Thing),at_thing(Thing,Place)]=>[in_bag(Thing,Bag),at_thi
ng(Thing,Place),fits_in(Thing,Bag)])],
    % conditional
    []).

operator(move(Carrier,Place,Place1),
    % prevail
    [],
    % necessary
    [
sc(carrier,Carrier,[at_carrier(Carrier,Place)]=>[at_carrier(Carrier,Place1),ne(P
lace1,Place)])],
    % conditional
    [
sc(box,Box,[box_in_bag(Box,Bag),at_carrier(Box,Place),goes_in(Box,Bag)]=>[box_in
_bag(Box,Bag),at_carrier(Box,Place1),goes_in(Box,Bag),ne(Place1,Place)]),

sc(thing,Thing,[at_thing(Thing,Place),fits_in(Thing,Bag),in_bag(Thing,Bag)]=>[at
_thing(Thing,Place1),fits_in(Thing,Bag),in_bag(Thing,Bag),ne(Place1,Place)]),

sc(thing,Thing1,[in_box(Thing1,Box),at_thing(Thing1,Place)]=>[in_box(Thing1,Box)
,at_thing(Thing1,Place1),ne(Place1,Place)])]).

operator(take_out_box(Bag,Place,Box),
    % prevail
    [     se(bag,Bag,[at_carrier(Bag,Place)])],
    % necessary
    [
sc(box,Box,[box_in_bag(Box,Bag),at_carrier(Box,Place),goes_in(Box,Bag)]=>[box_ou
tside(Box),at_carrier(Box,Place)])],
    % conditional
    [
sc(thing,Thing,[in_box(Thing,Box),at_thing(Thing,Place),safe_in(Thing,Box)]=>[in
```

```
_box(Thing,Box),at_thing(Thing,Place),safe_in(Thing,Box)])]).

operator(empty_box(Box,Place,Thing),
    % prevail
    [     se(box,Box,[box_outside(Box),at_carrier(Box,Place)])],
    % necessary
    [
sc(thing,Thing,[in_box(Thing,Box),at_thing(Thing,Place),safe_in(Thing,Box)]=>[ou
tside(Thing),at_thing(Thing,Place)])],
    % conditional
    []).

operator(take_out(Bag,Place,Thing),
    % prevail
    [     se(bag,Bag,[at_carrier(Bag,Place)])],
    % necessary
    [
sc(thing,Thing,[at_thing(Thing,Place),in_bag(Thing,Bag),fits_in(Thing,Bag)]=>[ou
tside(Thing),at_thing(Thing,Place)])],
    % conditional
    []).

% Methods
/****
 *
 */
method(pack(Suit,Place,Bag),
    % pre-condition
    [
],
    % Index Transitions
    [
```

```
sc(thing,Suit,[outside(Suit),at_thing(Suit,Place)]=>[in_bag(Suit,Bag),at_thing(S
uit,Place)])],
    % Static
    [
     fits_in(Suit,Bag)],
    % Temporal Constraints
    [
     before(1,2)],
    % Decomposition
    [
     achieve(ss(bag,Suitcase,[at_carrier(Suitcase,Place)])),
     put_thing_in_bag(Bag,Place,Thing)]
).
/****
 *
 */
method(pack_lunch(Sandwiches,Place,Lunch_box,Bag),
    % pre-condition
    [
],
    % Index Transitions
    [

sc(thing,Sandwiches,[outside(Sandwiches),at_thing(Sandwiches,Place)]=>[in_box(Sa
ndwiches,Lunch_box),at_thing(Sandwiches,Place)]),

sc(lunch_box,Lunch_box,[box_outside(Lunch_box),at_carrier(Lunch_box,Place)]=>[bo
x_in_bag(Lunch_box,Bag),at_carrier(Lunch_box,Place)])],
    % Static
    [
     safe_in(Thing,Lunch_box),
     goes_in(Lunch_box,Bag)],
    % Temporal Constraints
```

```
    [
     before(1,2)],
    % Decomposition
    [
     put_in_box(Box,Place,Thing),
     put_box_in_bag(Bag,Place,Box)]
).
/****
 *
 */
method(unpack_lunch(Lunch_box,Bag,Place,Sandwiches),
    % pre-condition
    [
],
    % Index Transitions
    [

sc(lunch_box,Lunch_box,[box_in_bag(Lunch_box,Bag),at_carrier(Lunch_box,Place)]=>
[box_outside(Lunch_box),at_carrier(Lunch_box,Place)]),

sc(thing,Sandwiches,[in_box(Sandwiches,Lunch_box),at_thing(Sandwiches,Place)]=>[
outside(Sandwiches),at_thing(Sandwiches,Place)])],
    % Static
    [
     safe_in(Thing,Lunch_box),
     goes_in(Lunch_box,Bag)],
    % Temporal Constraints
    [
     before(1,2)],
    % Decomposition
    [
     take_out_box(Bag,Place,Box),
     empty_box(Box,Place,Thing)]
```

```
).
/****
 *
 */
method(pack_and_take(Thing,Place,Bag,Place1),
    % pre-condition
    [
],
    % Index Transitions
    [

sc(thing,Thing,[outside(Thing),at_thing(Thing,Place)]=>[in_bag(Thing,Bag),at_thi
ng(Thing,Place1),ne(Place1,Place)])],
    % Static
    [
     fits_in(Thing,Bag)],
    % Temporal Constraints
    [
     before(1,2)],
    % Decomposition
    [
     pack(Suit,Place,Bag),
     move(Carrier,Place,Place1)]
).
/****
 *
 */
method(take_lunch_to_work(Sandwiches,Place,Place1,Lunch_box,Briefcase),
    % pre-condition
    [
],
    % Index Transitions
    [
```

```
sc(thing,Sandwiches,[outside(Sandwiches),at_thing(Sandwiches,Place)]=>[outside(S
andwiches),at_thing(Sandwiches,Place1),ne(Place1,Place)]),

sc(box,Lunch_box,[box_outside(Lunch_box),at_carrier(Lunch_box,Place)]=>[box_outs
ide(Lunch_box),at_carrier(Lunch_box,Place1),ne(Place1,Place)]),

sc(bag,Briefcase,[at_carrier(Briefcase,Place)]=>[at_carrier(Briefcase,Place1),ne
(Place1,Place)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(1,2),
     before(2,3)],
    % Decomposition
    [
     pack_lunch(Sandwiches,Place,Lunch_box,Bag),
     move(Carrier,Place,Place1),
     unpack_lunch(Lunch_box,Bag,Place,Sandwiches)]
).
/****
 *
 */
method(take_lunch_and_item_to_work(Bag,Place,Place1,Lunch_box,Thing,Sandwiches),
    % pre-condition
    [
],
    % Index Transitions
    [

sc(briefcase,Bag,[at_carrier(Bag,Place)]=>[at_carrier(Bag,Place1),ne(Place1,Plac
```

```
e)]),

sc(lunch_box,Lunch_box,[box_outside(Lunch_box),at_carrier(Lunch_box,Place)]=>[bo
x_outside(Lunch_box),at_carrier(Lunch_box,Place1),ne(Place1,Place)]),

sc(thing,Thing,[outside(Thing),at_thing(Thing,Place)]=>[outside(Thing),at_thing(
Thing,Place1),ne(Place1,Place)]),

sc(thing,Sandwiches,[outside(Sandwiches),at_thing(Sandwiches,Place)]=>[outside(S
andwiches),at_thing(Sandwiches,Place1),ne(Place1,Place)])],
    % Static
    [
     fits_in(Thing,Bag),
     safe_in(Sandwiches,Lunch_box),
     goes_in(Lunch_box,Bag)],
    % Temporal Constraints
    [
     before(1,3),
     before(2,3),
     before(3,4),
     before(3,5)],
    % Decomposition
    [
     put_thing_in_bag(Bag,Place,Thing),
     pack_lunch(Sandwiches,Place,Lunch_box,Bag),
     move(Carrier,Place,Place1),
     take_out(Bag,Place,Thing),
     unpack_lunch(Lunch_box,Bag,Place,Sandwiches)]
).

% Domain Tasks

% HTN Domain Tasks
```

```
htn_task(1,
    goal(
          [
            pack(Suit,Place,Bag)],
    % Temporal Constraints
          [
],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(thing,suit,[outside(suit),at_thing(suit,home)]),
     ss(suitcase,sc1,[at_carrier(sc1,home)])]).
htn_task(2,
    goal(
          [
            pack_lunch(Sandwiches,Place,Lunch_box,Bag)],
    % Temporal Constraints
          [
],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(thing,sandwiches,[outside(sandwiches),at_thing(sandwiches,home)]),
     ss(briefcase,bc1,[at_carrier(bc1,home)]),
     ss(lunch_box,lb1,[box_outside(lb1),at_carrier(lb1,home)])]).
htn_task(3,
    goal(
          [
            unpack_lunch(Lunch_box,Bag,Place,Sandwiches)],
```

```
    % Temporal Constraints
          [
],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,office)]),
     ss(lunch_box,lb1,[box_in_bag(lb1,bc1),at_carrier(lb1,office)]),

ss(thing,sandwiches,[in_box(sandwiches,lb1),at_thing(sandwiches,office)])]).
htn_task(4,
    goal(
          [
            pack_and_take(suit,home,sc1,office)],
    % Temporal Constraints
          [
],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(thing,suit,[outside(suit),at_thing(suit,home)]),
     ss(suitcase,sc1,[at_carrier(sc1,home)])]).
htn_task(5,
    goal(
          [
            move(bc1,home,office)],
    % Temporal Constraints
          [
],
```

```
    % Static constraints
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,home)])]).
htn_task(6,
    goal(
          [
            pack_lunch(sandwiches,home,lb1,bc1),
            move(bc1,home,office),
            unpack_lunch(lb1,bc1,office,sandwiches)],
    % Temporal Constraints
          [
            before(1,2),
            before(2,3)],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,home)]),
     ss(lunch_box,lb1,[box_outside(lb1),at_carrier(lb1,home)]),
     ss(thing,sandwiches,[outside(sandwiches),at_thing(sandwiches,home)])]).
htn_task(7,
    goal(
          [
            put_thing_in_bag(bc1,home,dictionary),
            take_out(bc1,office,dictionary),
            move(bc1,home,office),
            put_in_box(pb1,home,pencil),
            put_box_in_bag(bc1,home,pb1),
            take_out_box(bc1,office,pb1),
```

```
            empty_box(pb1,office,pencil)],
    % Temporal Constraints
          [
            before(1,3),
            before(3,2),
            before(5,3),
            before(3,6),
            before(4,5),
            before(6,7)],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,home)]),
     ss(thing,pencil,[outside(pencil),at_thing(pencil,home)]),
     ss(pencil_box,pb1,[box_outside(pb1),at_carrier(pb1,home)]),
     ss(thing,dictionary,[outside(dictionary),at_thing(dictionary,home)])]).
htn_task(8,
    goal(
          [
            put_thing_in_bag(bc1,home,cheque),
            pack_lunch(sandwiches,home,lb1,bc1),
            move(bc1,home,office),
            unpack_lunch(lb1,bc1,office,sandwiches),
            take_out(bc1,office,cheque)],
    % Temporal Constraints
          [
            before(1,3),
            before(2,3),
            before(3,5),
            before(3,4)],
    % Static constraints
```

```
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,home)]),
     ss(lunch_box,lb1,[box_outside(lb1),at_carrier(lb1,home)]),
     ss(thing,sandwiches,[outside(sandwiches),at_thing(sandwiches,home)]),
     ss(thing,cheque,[outside(cheque),at_thing(cheque,home)])]).
htn_task(9,
    goal(
          [
            put_thing_in_bag(bc1,home,cheque),
            put_in_box(lb1,home,sandwiches),
            put_box_in_bag(bc1,home,lb1),
            move(bc1,home,office),
            take_out_box(bc1,office,lb1),
            empty_box(lb1,office,sandwiches),
            take_out(bc1,office,cheque)],
    % Temporal Constraints
          [
            before(1,2),
            before(2,3),
            before(3,4),
            before(4,5),
            before(5,6),
            before(6,7)],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(briefcase,bc1,[at_carrier(bc1,home)]),
     ss(lunch_box,lb1,[box_outside(lb1),at_carrier(lb1,home)]),
```

```
ss(thing,sandwiches,[outside(sandwiches),at_thing(sandwiches,home)]),
ss(thing,cheque,[outside(cheque),at_thing(cheque,home)])]).
```

Appendix D

TEST FILE WITH RESULTS FROM HBC SHOWING THE SORT TREE CODE IS WORKING

/* THIS FILE REPLICATES THE TASK FOR PACK LUNCH AND TAKE TO WORK Task 6 in my original domain */

```
:- multifile input/3.
:- dynamic input/3.
:- multifile planner_task/3.
:- dynamic planner_task/3.

% Sequops below details the task to be achieved and names the operators.
% Arguably it is equivalent to the goal state (part of the task editor
% in GIPO)

t1 :- sequops([
put_in_box(sandwiches,lb1,home),
put_box_in_bag(bc1,home,sandwiches,lb1),
move(bc1,sandwiches,lb1,home,office),
take_out_box(lb1,sandwiches,bc1,office),
empty_box(lb1,sandwiches,office)
]).

% Labels the method produced
htn(pack_and_take_lunch_to_work).

% details initial states for all the dynamic objects equivalent to GIPO's
% initial states in the task editor
```

```
planner_task(_,_,
  [
   ss(thing,sandwiches, [outside(sandwiches),
        at_thing(sandwiches,home)]),
   ss(thing,pencil, [outside(pencil),
        at_thing(pencil,home)]),
   ss(thing,dictionary, [outside(dictionary),
        at_thing(dictionary,home)]),
   ss(thing,cheque, [outside(cheque),
        at_thing(cheque,home)]),
   ss(thing,suit, [outside(suit),
        at_thing(suit,home)]),
   ss(box,lb1, [at_carrier(lb1,home),box_outside(lb1)]),
   ss(box,pb1, [at_carrier(pb1,home),box_outside(pb1)]),
   ss(carrier,bc1, [at_carrier(bc1,home)]),
   ss(carrier,sc1, [at_carrier(sc1,home)])
  ]).

/* The example inputs are the equivalent of GIPO asking which state
an object is in after the action. These inputs are numbered, one for every
operator heading in sequops. For each dynamic object in the operator heading
there is an input line where, if the object does not change state, null
implies there is no state change and prompts a prevail clause.
At present you have to put any object in the operator heading which is
conditional and, because opmaker does not yet deal correctly with conditional
domains an extra necessary transition is created for it which gets round the
problem for now. */

% EXAMPLE INPUTS
%put_in_box(sandwiches,lb1,home)
input(1,sandwiches,sclass(Thing,thing,[in_box(Thing,Box),
at_thing(Thing,Place)])).
input(1,lb1,null).
```

```
%put_box_in_bag(bc1,home,sandwiches,lb1)
input(2,bc1,null).
input(2,sandwiches,null).
input(2,lb1,sclass(Box,box,[box_in_bag(Box,Bag)])).

%move(bc1,sandwiches,lb1,home,office)
input(3,bc1,sclass(Bag,carrier,[at_carrier(Bag,Office)])).
input(3,sandwiches,sclass(Thing,thing,[in_box(Thing,Box),
at_thing(Thing,Office)])).
input(3,lb1,sclass(Box,carrier,[at_carrier(Box,Office)])).
input(3,lb1,sclass(Box,box,[box_in_bag(Box,Bag)])).

%take_out_box(lb1,sandwiches,bc1,office)
input(4,lb1,sclass(Box,box,[box_outside(Box)])).
input(4,sandwiches,null).
input(4,bc1,null).

%empty_box(lb1,sandwiches,office)
input(5,lb1,null).
input(5,sandwiches,sclass(Thing,thing,[outside(Thing),
at_thing(Thing,Office)])).

/*************************************************************************
This section is a listing of the domain sorts, objects, predicates, statics
substate classes, invariants and constraints (axioms) and excluding methods,
operators and tasks.
*************************************************************************/
domain_name(hier_briefcase).

option(hierarchical).

% Sorts
```

```
sorts(non_primitive_sorts, [bag,box,carrier]).
sorts(primitive_sorts,[briefcase,suitcase,lunch_box,pencil_box,thing,place]).
sorts(carrier,[bag,box]).
sorts(bag,[briefcase,suitcase]).
sorts(box,[lunch_box,pencil_box]).

% Objects
objects(briefcase,[bc1]).
objects(suitcase,[sc1]).
objects(lunch_box,[lb1]).
objects(pencil_box,[pb1]).
objects(thing,[cheque,suit,dictionary,sandwiches,pencil]).
objects(place,[home,office]).

% Predicates
predicates([
% dynamic
    at_thing(thing,place),
    outside(thing),
    at_carrier(carrier,place),
    in_bag(thing,bag),
    in_box(thing,box),
    box_in_bag(box,bag),
    box_outside(box),
% static
    fits_in(thing,bag),
    safe_in(thing,box),
    goes_in(box,bag)]).

% Object Class Definitions
substate_classes([

carrier(C,
```

```
[
  [at_carrier(C,L)]
]),

thing(T,
[
  [outside(T),at_thing(T,L)],
  [in_bag(T,Bag),at_thing(T,L)],
  [in_box(T,Box),at_thing(T,L)]
]),

box(Box,
[
  [box_in_bag(Box,Bag)],
  [box_outside(Box)]
])

]).

/* to fit with the opmaker method the substate classes have to be in the format
above. This section shows how they would normally appear.

substate_classes(carrier,Carrier,[
    [at_carrier(Carrier,Place)]]).
substate_classes(thing,Thing,[
    [outside(Thing),at_thing(Thing,Place)],
    [in_bag(Thing,Bag),at_thing(Thing,Place)],
    [in_box(Thing,Box),at_thing(Thing,Place)]]).
substate_classes(box,Box,[
    [box_in_bag(Box,Bag)],
    [box_outside(Box)]]). */

% Atomic Invariants
```

```
atomic_invariants([
    fits_in(cheque,bc1),
    fits_in(dictionary,bc1),
    fits_in(suit,sc1),
    fits_in(cheque,sc1),
    fits_in(dictionary,sc1),
    goes_in(lb1,bc1),
    goes_in(pb1,bc1),
    safe_in(pencil,pb1),
    safe_in(sandwiches,lb1)]).

% Implied Invariants

% Inconsistent Constraints
inconsistent_constraint([at_thing(Thing,Place),at_thing(Thing,Place1),
ne(Place1,Place)]).
inconsistent_constraint([at_carrier(Carrier,Place),at_carrier(Carrier,Place1),
ne(Place1,Place)]).
inconsistent_constraint([in_bag(Thing,Bag),in_bag(Thing,Bag1),ne(Bag1,Bag)]).
inconsistent_constraint([in_box(Thing,Box),in_box(Thing,Box1),ne(Box1,Box)]).
inconsistent_constraint([in_bag(Thing,Bag),outside(Thing)]).
inconsistent_constraint([in_box(Thing,Box),outside(Thing)]).
inconsistent_constraint([in_bag(Thing,Bag),at_carrier(Carrier,Place),
at_thing(Thing,Place1),ne(Place1,Place)]).
inconsistent_constraint([in_box(Thing,Box),at_carrier(Carrier,Place),
at_thing(Thing,Place1),ne(Place1,Place)]).
inconsistent_constraint([outside(Thing),fits_in(Thing,Bag)]).
inconsistent_constraint([outside(Thing),safe_in(Thing,Box)]).
inconsistent_constraint([box_outside(Box),goes_in(Box,Bag)]).

/* RESULTS from running this test file with opmaker

states created..
```

```
operator(put_in_box(Sandwiches,Lb1,Home),
[se(lunch_box,Lb1,[box_outside(Lb1),at_carrier(Lb1,Home),safe_in(Sandwiches,Lb1)
])],
[sc(thing,Sandwiches,[outside(Sandwiches),at_thing(Sandwiches,Home)] =>
[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Home)])],
[]
).

operator(put_box_in_bag(Bc1,Home,Sandwiches,Lb1),
[se(briefcase,Bc1,[at_carrier(Bc1,Home),goes_in(Lb1,Bc1),safe_in(Sandwiches,Lb1)
]),
se(thing,Sandwiches,[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Home)])],
[sc(lunch_box,Lb1,[box_outside(Lb1)] => [box_in_bag(Lb1,Bc1)])],
[]
).

operator(move(Bc1,Sandwiches,Lb1,Home,Office),
[],
[sc(briefcase,Bc1,[at_carrier(Bc1,Home),goes_in(Lb1,Bc1),safe_in(Sandwiches,Lb1)
] => [at_carrier(Bc1,Office)]),
sc(thing,Sandwiches,[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Home)] =>
[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Office)]),
sc(lunch_box,Lb1,[at_carrier(Lb1,Home),box_in_bag(Lb1,Bc1)] =>
[at_carrier(Lb1,Office),box_in_bag(Lb1,Bc1)])],
[]
).

operator(take_out_box(Lb1,Sandwiches,Bc1,Office),
[se(thing,Sandwiches,[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Office),goes_in
(Lb1,Bc1),safe_in(Sandwiches,Lb1)]),
se(briefcase,Bc1,[at_carrier(Bc1,Office)])],
[sc(lunch_box,Lb1,[box_in_bag(Lb1,Bc1)] => [box_outside(Lb1)])],
```

```
[]
).

operator(empty_box(Lb1,Sandwiches,Office),
[se(lunch_box,Lb1,[box_outside(Lb1),at_carrier(Lb1,Office),safe_in(Sandwiches,Lb
1)])],
[sc(thing,Sandwiches,[in_box(Sandwiches,Lb1),at_thing(Sandwiches,Office)] =>
[outside(Sandwiches),at_thing(Sandwiches,Office)])],
[]
).


% name
method(pack_and_take_lunch_to_work(Sandwiches),
% dynamic constraints
[],
% list of necessary transitions
[sc(thing,Sandwiches,[outside(Sandwiches),at_thing(Sandwiches,Home)] =>
[outside(Sandwiches),at_thing(Sandwiches,Office)])],
% static constraints
[  goes_in(Lb1,Bc1),
  safe_in(Sandwiches,Lb1)],
% temporal  constraints
[before(1,2),before(2,3),before(3,4),before(4,5)],
%  decomposition
[  put_in_box(Sandwiches,Lb1,Home),
  put_box_in_bag(Bc1,Home,Sandwiches,Lb1),
  move(Bc1,Sandwiches,Lb1,Home,Office),
  take_out_box(Lb1,Sandwiches,Bc1,Office),
  empty_box(Lb1,Sandwiches,Office)]
).
*/
```

Appendix E

THE GIPO-CONSTRUCTED EXTENDED TYRE DOMAIN INCLUDING EXTRA TASKS

Automatically generated OCL Domain from GIPO Version 2.0

Author: Beth Richardson Institution: University of Huddersfield Date created: April 2000 Date last modified: 2006/11/01 at 03:16:13 PM GMT Description: This tyre domain has extra objects and actions. Now, instead of just two wheels there are five with five tyres, four wheel trims and four named hubs. There are also four sets of wheel nuts. The pump now has a use. If a tyre is low it may be 'flat' or 'punctured'. If it is flat then a prevail of inflate_tyre is have(pump).

```
domain_name(tyre_extended).

option(hierarchical).

% Sorts
sorts(primitive_sorts,[container,nuts,hub,pump,wheel,wrench,jack,wheel_trim,tyre]).

% Objects
objects(container,[boot]).
objects(nuts,[nuts1,nuts2,nuts3,nuts4]).
```

```
objects(hub,[hub1,hub2,hub3,hub4]).
objects(pump,[pump0]).
objects(wheel,[wheel1,wheel2,wheel3,wheel4,wheel5]).
objects(wrench,[wrench0]).
objects(jack,[jack0]).
objects(wheel_trim,[trim1,trim2,trim3,trim4]).
objects(tyre,[tyre1,tyre2,tyre3,tyre4,tyre5]).

% Predicates
predicates([
    closed(container),
    open(container),
    tight(nuts,hub),
    loose(nuts,hub),
    have_nuts(nuts),
    on_ground(hub),
    fastened(hub),
    jacked_up(hub,jack),
    free(hub),
    unfastened(hub),
    have_pump(pump),
    pump_in(pump,container),
    have_wheel(wheel),
    wheel_in(wheel,container),
    wheel_on(wheel,hub),
    have_wrench(wrench),
    wrench_in(wrench,container),
    have_jack(jack),
    jack_in_use(jack,hub),
    jack_in(jack,container),
    trim_on(wheel_trim,wheel),
    trim_off(wheel_trim),
    fits_on(tyre,wheel),
```

```
    full(tyre),
    flat(tyre),
    punctured(tyre)]).

% Object Class Definitions
substate_classes(container,C,[
    [closed(C)],
    [open(C)]]).
substate_classes(nuts,N,[
    [tight(N,H)],
    [loose(N,H)],
    [have_nuts(N)]]).
substate_classes(hub,H,[
    [on_ground(H),fastened(H)],
    [jacked_up(H,J),fastened(H)],
    [free(H),jacked_up(H,J),unfastened(H)],
    [unfastened(H),jacked_up(H,J)]]).
substate_classes(pump,Pu,[
    [have_pump(Pu)],
    [pump_in(Pu,C)]]).
substate_classes(wheel,Wh,[
    [have_wheel(Wh)],
    [wheel_in(Wh,C)],
    [wheel_on(Wh,H)]]).
substate_classes(wrench,Wr,[
    [have_wrench(Wr)],
    [wrench_in(Wr,C)]]).
substate_classes(jack,J,[
    [have_jack(J)],
    [jack_in_use(J,H)],
    [jack_in(J,C)]]).
substate_classes(wheel_trim,WT,[
    [trim_on(WT,Wh)],
```

```
    [trim_off(WT)]]).
substate_classes(tyre,Ty,[
    [fits_on(Ty,Wh)],
    [full(Ty)],
    [flat(Ty)],
    [punctured(Ty)]]).

% Atomic Invariants
atomic_invariants([
    fits_on(tyre1,wheel1),
    fits_on(tyre2,wheel2),
    fits_on(tyre3,wheel3),
    fits_on(tyre4,wheel4),
    fits_on(tyre5,wheel5)]).

% Implied Invariants

% Inconsistent Constraints
inconsistent_constraint([have_nuts(N),tight(N,_)]).
inconsistent_constraint([have_nuts(N),loose(N,_)]).
inconsistent_constraint([loose(_,H),tight(_,H)]).
inconsistent_constraint([unfastened(H),tight(_,H)]).
inconsistent_constraint([unfastened(H),loose(_,H)]).
inconsistent_constraint([wheel_in(Wh,_),wheel_on(Wh,_)]).
inconsistent_constraint([wheel_in(Wh,_),have_wheel(Wh)]).
inconsistent_constraint([jack_in(J,_),have_jack(J)]).
inconsistent_constraint([pump_in(Pu,_),have_pump(Pu)]).
inconsistent_constraint([wrench_in(Wr,_),have_wrench(Wr)]).
inconsistent_constraint([open(C),closed(C)]).
inconsistent_constraint([full(Ty),flat(Ty)]).
inconsistent_constraint([full(Ty),punctured(Ty)]).
inconsistent_constraint([flat(Ty),punctured(Ty)]).
inconsistent_constraint([fastened(H),unfastened(H)]).
```

```
inconsistent_constraint([jacked_up(H,J),on_ground(H)]).
inconsistent_constraint([free(H),wheel_on(_,H)]).
inconsistent_constraint([free(X),fastened(X)]).
inconsistent_constraint([free(X),tight(Nuts,X)]).
inconsistent_constraint([free(X),loose(Nuts,X)]).
inconsistent_constraint([wheel_on(W1,X),wheel_on(W2,X),ne(W1,W2)]).
inconsistent_constraint([wheel_on(W,H1),wheel_on(W,H2),ne(H1,H2)]).
inconsistent_constraint([wheel_on(W,H1),have_wheel(W)]).
inconsistent_constraint([jacked_up(H,J),jack_in(J,_)]).
inconsistent_constraint([jacked_up(H,J),have_jack(J)]).
inconsistent_constraint([fastened(H),have_nuts(N)]).
inconsistent_constraint([jack_in_use(J,_),jack_in(J,_)]).
inconsistent_constraint([jack_in_use(J,_),have_jack(J)]).
inconsistent_constraint([jack_in_use(J,H),on_ground(H)]).
inconsistent_constraint([trim_on(WT,Wh),loose(N,H)]).
inconsistent_constraint([trim_on(Wheel_trim,Wheel),jack_in_use(Jack,Hub)]).
inconsistent_constraint([trim_on(Wheel_trim,Wheel),have_nuts(Nuts)]).
inconsistent_constraint([trim_on(Wheel_trim,Wheel),free(Hub)]).
inconsistent_constraint([trim_on(Wheel_trim,Wheel),unfastened(Hub)]).

% Operators
operator(open_container(C),
    % prevail
    [],
    % necessary
    [     sc(container,C,[closed(C)]=>[open(C)])],
    % conditional
    []).
operator(close_container(C),
    % prevail
    [],
    % necessary
    [     sc(container,C,[open(C)]=>[closed(C)])],
```

```
    % conditional
    []).
operator(fetch_jack(C,J),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(jack,J,[jack_in(J,C)]=>[have_jack(J)])],
    % conditional
    []).
operator(fetch_wheel(C,Wh),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(wheel,Wh,[wheel_in(Wh,C)]=>[have_wheel(Wh)])],
    % conditional
    []).
operator(fetch_wrench(C,Wr),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(wrench,Wr,[wrench_in(Wr,C)]=>[have_wrench(Wr)])],
    % conditional
    []).
operator(fetch_pump(C,Pu),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(pump,Pu,[pump_in(Pu,C)]=>[have_pump(Pu)])],
    % conditional
    []).
operator(putaway_wheel(C,Wh),
    % prevail
    [     se(container,C,[open(C)])],
```

```
    % necessary
    [     sc(wheel,Wh,[have_wheel(Wh)]=>[wheel_in(Wh,C)])],
    % conditional
    []).
operator(putaway_wrench(C,Wr),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(wrench,Wr,[have_wrench(Wr)]=>[wrench_in(Wr,C)])],
    % conditional
    []).
operator(putaway_jack(C,J),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(jack,J,[have_jack(J)]=>[jack_in(J,C)])],
    % conditional
    []).
operator(putaway_pump(C,Pu),
    % prevail
    [     se(container,C,[open(C)])],
    % necessary
    [     sc(pump,Pu,[have_pump(Pu)]=>[pump_in(Pu,C)])],
    % conditional
    []).
operator(loosen(Wr,H,WT,N),
    % prevail
    [     se(wrench,Wr,[have_wrench(Wr)]),
     se(hub,H,[on_ground(H),fastened(H)]),
     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(nuts,N,[tight(N,H)]=>[loose(N,H)])],
    % conditional
```

```
    []).
operator(tighten(Wr,H,WT,N),
    % prevail
    [     se(wrench,Wr,[have_wrench(Wr)]),
     se(hub,H,[on_ground(H),fastened(H)]),
     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(nuts,N,[loose(N,H)]=>[tight(N,H)])],
    % conditional
    []).
operator(jack_up(N,H,J),
    % prevail
    [     se(nuts,N,[loose(N,H)])],
    % necessary
    [     sc(hub,H,[on_ground(H),fastened(H)]=>[jacked_up(H,J),fastened(H)]),
     sc(jack,J,[have_jack(J)]=>[jack_in_use(J,H)])],
    % conditional
    []).
operator(jack_down(N,H,J),
    % prevail
    [     se(nuts,N,[loose(N,H)])],
    % necessary
    [     sc(hub,H,[jacked_up(H,J),fastened(H)]=>[on_ground(H),fastened(H)]),
     sc(jack,J,[jack_in_use(J,H)]=>[have_jack(J)])],
    % conditional
    []).
operator(do_up(Wr,WT,H,J,N),
    % prevail
    [     se(wrench,Wr,[have_wrench(Wr)]),
     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(hub,H,[unfastened(H),jacked_up(H,J)]=>[jacked_up(H,J),fastened(H)]),
     sc(nuts,N,[have_nuts(N)]=>[loose(N,H)])],
```

```
    % conditional
    []).
operator(remove_wheel(WT,Wh,H,J),
    % prevail
    [     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(wheel,Wh,[wheel_on(Wh,H)]=>[have_wheel(Wh)]),
     sc(hub,H,[unfastened(H),jacked_up(H,J)]=>[free(H),jacked_up(H,J),unfastened(H)])],
    % conditional
    []).
operator(put_on_wheel(WT,Wh,H,J),
    % prevail
    [     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(wheel,Wh,[have_wheel(Wh)]=>[wheel_on(Wh,H)]),
     sc(hub,H,[free(H),jacked_up(H,J),unfastened(H)]=>[unfastened(H),jacked_up(H,J)])],
    % conditional
    []).
operator(undo(Wr,WT,H,J,N),
    % prevail
    [     se(wrench,Wr,[have_wrench(Wr)]),
     se(wheel_trim,WT,[trim_off(WT)])],
    % necessary
    [     sc(hub,H,[jacked_up(H,J),fastened(H)]=>[unfastened(H),jacked_up(H,J)]),
     sc(nuts,N,[loose(N,H)]=>[have_nuts(N)])],
    % conditional
    []).
operator(apply_trim(H,WT,Wh),
    % prevail
    [     se(hub,H,[on_ground(H),fastened(H)])],
    % necessary
    [     sc(wheel_trim,WT,[trim_off(WT)]=>[trim_on(WT,Wh)])],
    % conditional
```

```
    []).
operator(remove_trim(H,WT,Wh),
    % prevail
    [     se(hub,H,[on_ground(H),fastened(H)])],
    % necessary
    [     sc(wheel_trim,WT,[trim_on(WT,Wh)]=>[trim_off(WT)])],
    % conditional
    []).
operator(inflate_tyre(Pu,Ty),
    % prevail
    [     se(pump,Pu,[have_pump(Pu)])],
    % necessary
    [     sc(tyre,Ty,[flat(Ty)]=>[full(Ty)])],
    % conditional
    []).
operator(find_puncture(Pu,Ty),
    % prevail
    [     se(pump,Pu,[have_pump(Pu)])],
    % necessary
    [     sc(tyre,Ty,[flat(Ty)]=>[punctured(Ty)])],
    % conditional
    []).

% Methods
/****
 * The pump is used to re-inflate the tyre and then returned to the boot.
 */
method(fix_flat(Ty),
    % pre-condition
    [
],
    % Index Transitions
    [
```

```
     sc(tyre,Ty,[flat(Ty)]=>[full(Ty)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(1,2),
     before(2,3),
     before(3,4),
     before(4,5)],
    % Decomposition
    [
     open_container(C),
     fetch_pump(C,Pu),
     inflate_tyre(Pu,Ty),
     putaway_pump(C,Pu),
     close_container(C)]
).
/****
 * The pump is used to discover that the flat tyre won't inflate so it is
 classified as punctured. The pump is then returned to the boot.
 */
method(discover_puncture(Ty),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(tyre,Ty,[flat(Ty)]=>[punctured(Ty)])],
    % Static
    [
],
    % Temporal Constraints
```

```
    [
     before(1,2),
     before(2,3),
     before(3,4)],
    % Decomposition
    [
     achieve(ss(container,C,[open(C)])),
     fetch_pump(C,Pu),
     find_puncture(Pu,Ty),
     putaway_pump(C,Pu)]
).
/****
 * The jack and wrench needed to change the wheel are fetched.
 */
method(fetch_tools(Wr,C,J),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(wrench,Wr,[wrench_in(Wr,C)]=>[have_wrench(Wr)]),
     sc(jack,J,[jack_in(J,C)]=>[have_jack(J)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(1,2),
     before(1,3)],
    % Decomposition
    [
     achieve(ss(container,C,[open(C)])),
     fetch_jack(C,J),
```

```
     fetch_wrench(C,Wr)]
).
/****
 * The jack and wrench are put away in the boot.
 */
method(putaway_tools(Wr,C,J),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(wrench,Wr,[have_wrench(Wr)]=>[wrench_in(Wr,C)]),
     sc(jack,J,[have_jack(J)]=>[jack_in(J,C)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(1,2),
     before(1,3)],
    % Decomposition
    [
     achieve(ss(container,C,[open(C)])),
     putaway_wrench(C,Wr),
     putaway_jack(C,J)]
).
/****
 * The wheel trim is removed and the wheel nuts are loosened. After jacking
 up the car the wheel nuts are undone and removed.
 */
method(unfasten_hub(N,H,J,WT,Wh),
    % pre-condition
    [
```

```
     se(wrench,Wr,[have_wrench(Wr)])],
    % Index Transitions
    [
     sc(nuts,N,[tight(N,H)]=>[have_nuts(N)]),
     sc(hub,H,[on_ground(H),fastened(H)]=>[unfastened(H),jacked_up(H,J)]),
     sc(jack,J,[have_jack(J)]=>[jack_in_use(J,H)]),
     sc(wheel_trim,WT,[trim_on(WT,Wh)]=>[trim_off(WT)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(1,2),
     before(2,3),
     before(3,4),
     before(4,5)],
    % Decomposition
    [
     fetch_tools(Wr,C,J),
     remove_trim(H,WT,Wh),
     loosen(Wr,H,WT,N),
     jack_up(N,H,J),
     undo(Wr,WT,H,J,N)]
).
/****
 * The wheel nuts are applied and done up, then the jack is lowered and the
 wheel nuts tightened. Finally the wheel trim is replaced.
 */
method(fasten_hub(N,H,J,WT,Wh),
    % pre-condition
    [
     se(wrench,Wr,[have_wrench(Wr)])],
    % Index Transitions
```

```
    [
     sc(nuts,N,[have_nuts(N)]=>[tight(N,H)]),
     sc(hub,H,[unfastened(H),jacked_up(H,J)]=>[on_ground(H),fastened(H)]),
     sc(jack,J,[jack_in_use(J,H)]=>[have_jack(J)]),
     sc(wheel_trim,WT,[trim_off(WT)]=>[trim_on(WT,Wh)])],
    % Static
    [
],
    % Temporal Constraints
    [
     before(3,4),
     before(4,5),
     before(1,2),
     before(2,3)],
    % Decomposition
    [
     achieve(ss(hub,H,[unfastened(H),jacked_up(H,J)])),
     do_up(Wr,WT,H,J,N),
     jack_down(N,H,J),
     tighten(Wr,H,WT,N),
     apply_trim(H,WT,Wh)]
).
/****
 * The punctured wheel is removed and replaced by the spare.
 */
method(change_wheel(Wh1,H,Wh2),
    % pre-condition
    [
],
    % Index Transitions
    [
     sc(wheel,Wh1,[have_wheel(Wh1)]=>[wheel_on(Wh1,H)]),
     sc(wheel,Wh2,[wheel_on(Wh2,H),ne(Wh2,Wh1)]=>[have_wheel(Wh2),ne(Wh2,Wh1)])],
```

```
    % Static
    [
],
    % Temporal Constraints
    [
     before(2,3),
     before(1,2)],
    % Decomposition
    [
     achieve(ss(hub,H,[unfastened(H),jacked_up(H,J)])),
     remove_wheel(WT,Wh,H,J),
     put_on_wheel(WT,Wh,H,J)]
).


% Domain Tasks
planner_task(1,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     se(tyre,tyre1,[full(tyre1)]),
     se(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     se(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     se(pump,pump0,[pump_in(pump0,boot)]),
     se(wrench,wrench0,[wrench_in(wrench0,boot)]),
     se(jack,jack0,[jack_in(jack0,boot)])],
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
```

```
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)])]).
planner_task(2,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(wheel,wheel1,[wheel_in(wheel1,boot)]),
     se(wheel,wheel5,[wheel_on(wheel5,hub1)]),
     se(tyre,tyre1,[punctured(tyre1)]),
     se(wheel_trim,trim1,[trim_on(trim1,wheel5)]),
     se(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     se(pump,pump0,[pump_in(pump0,boot)]),
     se(wrench,wrench0,[wrench_in(wrench0,boot)]),
     se(jack,jack0,[jack_in(jack0,boot)]),
     se(nuts,nuts1,[tight(nuts1,hub1)])],
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
```

```
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)])]).
planner_task(3,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(tyre,tyre1,[full(tyre1)]),
     se(wheel,wheel1,[wheel_on(wheel1,hub4)]),
     se(wheel,wheel4,[wheel_in(wheel4,boot)]),
     se(wheel_trim,trim4,[trim_on(trim4,wheel1)])],
    % INIT States
    [
     ss(wheel,wheel1,[wheel_in(wheel1,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(tyre,tyre2,[full(tyre2)]),
     ss(wheel_trim,trim2,[trim_on(trim2,wheel2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(tyre,tyre3,[full(tyre3)]),
     ss(wheel_trim,trim3,[trim_on(trim3,wheel3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(tyre,tyre4,[punctured(tyre4)]),
     ss(wheel_trim,trim4,[trim_on(trim4,wheel4)]),
     ss(wheel,wheel5,[wheel_on(wheel5,hub1)]),
     ss(tyre,tyre5,[full(tyre5)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel5)]),
     ss(container,boot,[closed(boot)]),
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
```

```
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(hub,hub2,[on_ground(hub2),fastened(hub2)]),
     ss(hub,hub3,[on_ground(hub3),fastened(hub3)]),
     ss(hub,hub4,[on_ground(hub4),fastened(hub4)])]).
planner_task(4,
    % Goals
    [
     se(wheel,wheel5,[wheel_on(wheel5,hub3)]),
     se(hub,hub3,[on_ground(hub3),fastened(hub3)]),
     se(wheel_trim,trim3,[trim_on(trim3,wheel5)]),
     se(wheel,wheel3,[wheel_in(wheel3,boot)]),
     se(nuts,nuts3,[tight(nuts3,hub3)])],
    % INIT States
    [
     ss(wheel_trim,trim3,[trim_off(trim3)]),
     ss(tyre,tyre3,[punctured(tyre3)]),
     ss(hub,hub3,[on_ground(hub3),fastened(hub3)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(pump,pump0,[have_pump(pump0)]),
     ss(container,boot,[open(boot)]),
     ss(tyre,tyre5,[full(tyre5)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)])]).

% HTN Domain Tasks
htn_task(1,
```

```
    goal(
          [
            open_container(boot),
            fetch_pump(boot,pump0),
            inflate_tyre(pump0,Ty),
            putaway_pump(boot,pump0),
            close_container(boot)],
    % Temporal Constraints
          [
            before(1,2),
            before(2,3),
            before(3,4),
            before(4,5)],
    % Static constraints
          [
]),
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)])]).


/* Experimental set of domain tasks designed to cover the main tasks for which the methods were
```

```
% Domain Tasks
planner_task(1,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     se(tyre,tyre1,[full(tyre1)]),
     se(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     se(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     se(pump,pump0,[pump_in(pump0,boot)]),
     se(wrench,wrench0,[wrench_in(wrench0,boot)]),
     se(jack,jack0,[jack_in(jack0,boot)])],
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)])]).
planner_task(2,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(wheel,wheel1,[wheel_in(wheel1,boot)]),
     se(wheel,wheel5,[wheel_on(wheel5,hub1)]),
```

```
     se(tyre,tyre1,[punctured(tyre1)]),
     se(wheel_trim,trim1,[trim_on(trim1,wheel5)]),
     se(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     se(pump,pump0,[pump_in(pump0,boot)]),
     se(wrench,wrench0,[wrench_in(wrench0,boot)]),
     se(jack,jack0,[jack_in(jack0,boot)]),
     se(nuts,nuts1,[tight(nuts1,hub1)])],
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)])]).
planner_task(3,
    % Goals
    [
     se(container,boot,[closed(boot)]),
     se(tyre,tyre1,[full(tyre1)]),
     se(wheel,wheel1,[wheel_on(wheel1,hub4)]),
     se(wheel,wheel4,[wheel_in(wheel4,boot)]),
     se(wheel_trim,trim4,[trim_on(trim4,wheel1)])],
```

```
    % INIT States
    [
     ss(wheel,wheel1,[wheel_in(wheel1,boot)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(tyre,tyre2,[full(tyre2)]),
     ss(wheel_trim,trim2,[trim_on(trim2,wheel2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(tyre,tyre3,[full(tyre3)]),
     ss(wheel_trim,trim3,[trim_on(trim3,wheel3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(tyre,tyre4,[punctured(tyre4)]),
     ss(wheel_trim,trim4,[trim_on(trim4,wheel4)]),
     ss(wheel,wheel5,[wheel_on(wheel5,hub1)]),
     ss(tyre,tyre5,[full(tyre5)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel5)]),
     ss(container,boot,[closed(boot)]),
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(hub,hub2,[on_ground(hub2),fastened(hub2)]),
     ss(hub,hub3,[on_ground(hub3),fastened(hub3)]),
     ss(hub,hub4,[on_ground(hub4),fastened(hub4)])]).
planner_task(4,
    % Goals
    [
     se(wheel,wheel5,[wheel_on(wheel5,hub3)]),
     se(hub,hub3,[on_ground(hub3),fastened(hub3)]),
```

```
     se(wheel_trim,trim3,[trim_on(trim3,wheel5)]),
     se(wheel,wheel3,[wheel_in(wheel3,boot)]),
     se(nuts,nuts3,[tight(nuts3,hub3)])],
    % INIT States
    [
     ss(wheel_trim,trim3,[trim_off(trim3)]),
     ss(tyre,tyre3,[punctured(tyre3)]),
     ss(hub,hub3,[on_ground(hub3),fastened(hub3)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(pump,pump0,[have_pump(pump0)]),
     ss(container,boot,[open(boot)]),
     ss(tyre,tyre5,[full(tyre5)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)])]).
planner_task(5,
    % Goals
    [
     se(container,boot,[open(boot)]),
     se(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     se(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     se(tyre,tyre1,[punctured(tyre1)]),
     se(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     se(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     se(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     se(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     se(wheel,wheel5,[wheel_in(wheel5,boot)]),
     se(pump,pump0,[pump_in(pump0,boot)]),
     se(jack,jack0,[have_jack(jack0)]),
     se(wrench,wrench0,[have_wrench(wrench0)])],
    % INIT States
    [
```

```
     ss(container,boot,[closed(boot)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(tyre,tyre1,[flat(tyre1)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)])]).
planner_task(6,
    % Goals
    [
     se(nuts,nuts1,[have_nuts(nuts1)]),
     se(hub,hub1,[free(hub1),jacked_up(hub1,jack0),unfastened(hub1)]),
     se(wheel,wheel1,[have_wheel(wheel1)]),
     se(wheel_trim,trim1,[trim_off(trim1)])],
    % INIT States
    [
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(wrench,wrench0,[have_wrench(wrench0)]),
     ss(jack,jack0,[have_jack(jack0)]),
```

```
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
     ss(wheel_trim,trim2,[trim_on(trim2,wheel2)]),
     ss(wheel_trim,trim3,[trim_on(trim3,wheel3)]),
     ss(wheel_trim,trim4,[trim_on(trim4,wheel4)]),
     ss(tyre,tyre1,[punctured(tyre1)])]).
planner_task(7,
    % Goals
    [
     se(tyre,tyre1,[punctured(tyre1)]),
     se(wheel_trim,trim1,[trim_off(trim1)]),
     se(jack,jack0,[jack_in_use(jack0,hub1)]),
     se(wrench,wrench0,[have_wrench(wrench0)]),
     se(wheel,wheel1,[have_wheel(wheel1)]),
     se(hub,hub1,[free(hub1),jacked_up(hub1,jack0),unfastened(hub1)]),
     se(nuts,nuts1,[have_nuts(nuts1)]),
     se(container,boot,[open(boot)])],
    % INIT States
    [
     ss(container,boot,[closed(boot)]),
     ss(nuts,nuts1,[tight(nuts1,hub1)]),
     ss(nuts,nuts2,[tight(nuts2,hub2)]),
     ss(nuts,nuts3,[tight(nuts3,hub3)]),
     ss(nuts,nuts4,[tight(nuts4,hub4)]),
     ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
     ss(pump,pump0,[pump_in(pump0,boot)]),
     ss(wheel,wheel1,[wheel_on(wheel1,hub1)]),
     ss(wheel,wheel2,[wheel_on(wheel2,hub2)]),
     ss(wheel,wheel3,[wheel_on(wheel3,hub3)]),
     ss(wheel,wheel4,[wheel_on(wheel4,hub4)]),
     ss(wheel,wheel5,[wheel_in(wheel5,boot)]),
     ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
     ss(jack,jack0,[jack_in(jack0,boot)]),
     ss(wheel_trim,trim1,[trim_on(trim1,wheel1)]),
```

```
    ss(wheel_trim,trim2,[trim_on(trim2,wheel2)]),
    ss(wheel_trim,trim3,[trim_on(trim3,wheel3)]),
    ss(wheel_trim,trim4,[trim_on(trim4,wheel4)]),
    ss(tyre,tyre1,[flat(tyre1)])]).
*/
```

Appendix F

A TYPICAL TEST FILE TO GENERATE THE METHOD *DISCOVER_PUNCTURE* IN THE EXTENDED TYRE DOMAIN

```
/* Experimental file to produce a method for the actions
- open (boot)
- fetch_pump
- find_puncture
- putaway_pump

from tyre_extended world. The method will be called discover_puncture. */

:- multifile input/3.
:- dynamic input/3.
:- multifile planner_task/3.
:- dynamic planner_task/3.

sequence([
open_container(boot),
fetch_pump(@boot,pump0),
find_puncture(@pump0,tyre1),
putaway_pump(@boot,pump0)

]).


htn(discover_puncture).
```

```
opposites(_,_):-fail.

planner_task(_,
   % Goal
  [
   se(container,boot,[open(boot)]),
   se(pump,pump0,[pump_in(pump0,boot)]),
   se(tyre,tyre1,[punctured(tyre1)])
  ],
  % Initial state
  [
   ss(container,boot,[closed(boot)]),
   ss(wheel,wheel1,[wheel_on(wheel1,hub1),trim_on(wheel1,trim1)]),
   ss(wheel,wheel2,[wheel_on(wheel2,hub2),trim_on(wheel2,trim2)]),
   ss(wheel,wheel3,[wheel_on(wheel3,hub3),trim_on(wheel3,trim3)]),
   ss(wheel,wheel4,[wheel_on(wheel4,hub4),trim_on(wheel4,trim4)]),
   ss(wheel,wheel5,[wheel_in(wheel5,boot),trim_off(wheel5)]),
   ss(tyre,tyre2,[full(tyre2)]),
   ss(tyre,tyre1,[flat(tyre1)]),
   ss(wheel_trim,trim1,[trim_on_wheel(trim1,wheel1)]),
   ss(wheel_trim,trim2,[trim_on_wheel(trim2,wheel2)]),
   ss(wheel_trim,trim3,[trim_on_wheel(trim3,wheel3)]),
   ss(wheel_trim,trim4,[trim_on_wheel(trim4,wheel4)]),
   ss(nuts,nuts1,[tight(nuts1,hub1)]),
ss(nuts,nuts2,[tight(nuts2,hub2)]),
ss(nuts,nuts3,[tight(nuts3,hub3)]),
ss(nuts,nuts4,[tight(nuts4,hub4)]),
   ss(pump,pump0,[pump_in(pump0,boot)]),
   ss(jack,jack0,[jack_in(jack0,boot)]),
   ss(wrench,wrench0,[wrench_in(wrench0,boot)]),
   ss(hub,hub1,[on_ground(hub1),fastened(hub1)]),
   ss(hub,hub2,[on_ground(hub2),fastened(hub2)]),
```

```
ss(hub,hub3,[on_ground(hub3),fastened(hub3)]),
ss(hub,hub4,[on_ground(hub4),fastened(hub4)])
   ]).

/**
 *  All rights reserved. Use of this software is permitted for non-commercial
 *  research purposes, and it may be copied only for that use.  All copies must
 *  include this copyright message.  This software is made available AS IS, and
 *  neither the GIPO team nor the University of Huddersfield make any warranty
 *  about the software or its performance.
 *
 *  Automatically generated OCL Domain from  GIPO Version 2.0
 *
 *  Author: Beth Richardson
 *  Institution: University of Huddersfield
 *  Date created: April 2000
 *  Date last modified: 2006/10/25 at 03:52:11 PM BST
 *  Description:
 *    This tyre domain has extra objects and actions. Now, instead of just two
 *    wheels there are five with five tyres, four wheel trims and four named
 *    hubs. There are also four sets of wheel nuts. The pump now has a use. If
 *    a tyre is low it may be 'flat' or 'punctured'. If it is flat then a prevail of
 *    inflate_tyre is have(pump).
 */

domain_name(tyre_extended).

% Sorts
sorts(primitive_sorts,[container,nuts,hub,pump,wheel,wrench,jack,wheel_trim,tyre]).

% Objects
objects(container,[boot]).
objects(nuts,[nuts1,nuts2,nuts3,nuts4]).
```

```
objects(hub,[hub1,hub2,hub3,hub4]).
objects(pump,[pump0]).
objects(wheel,[wheel1,wheel2,wheel3,wheel4,wheel5]).
objects(wrench,[wrench0]).
objects(jack,[jack0]).
objects(wheel_trim,[trim1,trim2,trim3,trim4]).
objects(tyre,[tyre1,tyre2,tyre3,tyre4,tyre5]).

% Predicates
predicates([
    closed(container),
    open(container),
    tight(nuts,hub),
    loose(nuts,hub),
    have_nuts(nuts),
    on_ground(hub),
    fastened(hub),
    jacked_up(hub,jack),
    free(hub),
    unfastened(hub),
    have_pump(pump),
    pump_in(pump,container),
    have_wheel(wheel),
    wheel_in(wheel,container),
    wheel_on(wheel,hub),
    have_wrench(wrench),
    wrench_in(wrench,container),
    have_jack(jack),
    jack_in_use(jack,hub),
    jack_in(jack,container),
    trim_on(wheel,wheel_trim),
    trim_off(wheel),
    fits_on(tyre,wheel),
```

```
    full(tyre),
    flat(tyre),
    punctured(tyre),
    have_trim(wheel_trim),
    trim_on_wheel(wheel_trim,wheel)]).

% Object Class Definitions
substate_classes([
container(C,
   [
    [closed(C)],
    [open(C)]
   ]),
nuts(N,
   [
    [tight(N,H)],
    [loose(N,H)],
    [have_nuts(N)]
   ]),
hub(H,
   [
    [on_ground(H),fastened(H)],
    [jacked_up(H,J),fastened(H)],
    [free(H),jacked_up(H,J),unfastened(H)],
    [unfastened(H),jacked_up(H,J)]
   ]),
pump(Pu,
   [
    [have_pump(Pu)],
    [pump_in(Pu,C)]
   ]),
wheel(Wh,
   [
```

```
    [have_wheel(Wh),trim_off(Wh)],
    [wheel_in(Wh,C),trim_off(Wh)],
    [wheel_on(Wh,H),trim_off(Wh)],
    [wheel_on(Wh,H),trim_on(Wh,WT)]
   ]),
wrench(Wr,
   [
    [have_wrench(Wr)],
    [wrench_in(Wr,C)]
   ]),
jack(J,
   [
    [have_jack(J)],
    [jack_in_use(J,H)],
    [jack_in(J,C)]
   ]),
wheel_trim(WT,
   [
    [trim_on_wheel(WT,Wh)],
    [have_trim(WT)]
   ]),
tyre(Ty,
   [
    [full(Ty)],
    [flat(Ty)],
    [punctured(Ty)],
    [fits_on(Ty,Wh)]
   ])
]).

% Atomic Invariants
atomic_invariants([
    fits_on(tyre1,wheel1),
```

```
    fits_on(tyre2,wheel2),
    fits_on(tyre3,wheel3),
    fits_on(tyre4,wheel4),
    fits_on(tyre5,wheel5)]).

% Implied Invariants

% Inconsistent Constraints

% Equivalences between predicates
%  (note redundancy of predicates sometimes has useful side-effect of
%   of enforcing 1:1 relationships)
invariant( all(H:hub,fastened(H)<==>ex(N:nuts,tight(N,H)\/loose(N,H))) ).
invariant( all(H:hub,all(J:jack,jack_in_use(J,H)<==>jacked_up(H,J))) ).
invariant( all(H:hub,~free(H)<==>ex(W:wheel,wheel_on(W,H))) ).
invariant( all(T:wheel_trim,all(W:wheel,trim_on_wheel(T,W)<==>trim_on(W,T))) ).

% Hub may only have one set of nuts attached
invariant(
  all(H:hub,all(N1:nuts,all(N2:nuts,
         (tight(N1,H)\/loose(N1,H)) /\
         (tight(N2,H)\/loose(N2,H))
         ==>(N1=N2) ))) ).

% Hub may only have one wheel attached.
invariant( all(H:hub,all(W1:wheel,all(W2:wheel,
              wheel_on(W1,H)/\wheel_on(W2,H)==>(W1=W2) ))) ).

% If the nuts are tight then the hub must be on the ground.
invariant( all(H:hub, ex(N:nuts,tight(N,H)) ==> on_ground(H)) ).

% If a trim is on a wheel, then the wheel is on a hub and
% the nuts are tight.
```

```
invariant(
  all(W:wheel,ex(T:wheel_trim,trim_on_wheel(T,W))==>
     ex(H:hub,wheel_on(W,H)/\ex(N:nuts,tight(N,H)))) ).

/*
OUTPUT - the correct set of operators and a method


operator(fetch_pump(Container1,Pump2),
   [se(container,boot,[open(Container1)])
   ],
   [sc(pump,Pump2,[pump_in(Pump2,Container1)]=>[have_pump(Pump2)])
   ],
   []
).

operator(find_puncture(Pump1,Tyre2),
   [se(pump,pump0,[have_pump(Pump1)])
   ],
   [sc(tyre,Tyre2,[flat(Tyre2)]=>[punctured(Tyre2)])
   ],
   []
).

operator(open_container(Container1),
   [],
   [sc(container,Container1,[closed(Container1)]=>[open(Container1)])
   ],
   []
).

operator(putaway_pump(Container1,Pump2),
   [se(container,boot,[open(Container1)])
```

```
   ],
   [sc(pump,Pump2,[have_pump(Pump2)]=>[pump_in(Pump2,Container1)])
   ],
   []
).

*/



% name
method(discover_puncture(Tyre1,Boot,Pump0),
% dynamic constraints
[se(pump,Pump0,[pump_in(Pump0,Boot)])],
% list of necessary transitions
[sc(tyre,Tyre1,[flat(Tyre1)] => [punctured(Tyre1)]),
sc(container,Boot,[closed(Boot)] => [open(Boot)])],
% static constraints
[],
% temporal  constraints
[before(1,2),before(2,3),before(3,4)],
%  decomposition
[  open_container(Boot),
  fetch_pump(Boot,Pump0),
  find_puncture(Pump0,Tyre1),
  putaway_pump(Boot,Pump0)]
).
```

BIBLIOGRAPHY

[1] M. Aben, J. Balder, and F. van Harmelen. Support for the formalisation and validation of kads expertise model. Technical report, KADS-II/M2/UvA/DM2.6a/1.0, ESPRIT, 1994.

[2] Scott Andrews, Brian Kettler, Kutluhan Erol, and James Hendler. Um translog: A planning domain for the development and benchmarking of planning systems. Technical report, Dept. of Computer Science, University of Maryland, College Park, MD 20742, USA 301.405.1000, 1995.

[3] J. Blythe and T. M. Mitchell. On becoming reactive. *Segre*, pages 255–257, 1989.

[4] Jim Blythe and Varun Ratnaker. Helping end users modify procedures by instruction. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[5] B. Borchers and J. Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *Journal of Combinatorial Optimization 2*, 4:299 – 306, 1999.

[6] Daniel Borrajo, Susana Fernandez, Raquel Fuetetaja, and Juan D. Arias. Tool for automatically acquiring control knowledge for planning. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[7] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 - fourth edition world wide web consortium, recommendation rec-xml-20060816. http://www.w3.org/TR/REC-xml/, 2006.

[8] B. G. Buchanan and T. M. Mitchell. Model-directed learning of production rules. In *Pattern-Directed Inference Systems*. Academic Press, 1978.

[9] J. G. Carbonell. Introduction: Paradigms for machine learning. *Artificial Intelligence*, 40:1–9, 1989.

[10] Jaime G. Carbonell and Yolanda Gil. *Learning by experimentation: the operator refinement method*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[11] P. P.-S. Chen. The Entity-Relationship Model: Towards a Unified View of Data. In *ACM Transactions on Database Systems*, volume 1, pages 9–36. ACM Press, 1976.

[12] Patrick Daley, Jeremy Frank, Michael Iatauro, Conor McGann, and Will Taylor. Planworks: A debugging environment for constraint based planning systems. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[13] Marie desJardins. Knowledge development methods for planning systems. In *AAAi-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, New Orleans, LA, USA, 1994. AAAI.

[14] F.D. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, United States of America and Canada, 1999.

[15] S. Edelkamp and J. Hoffmann. Pddl2.2: The language for the classical part of the 4th international planning competition? In *Proceedings of the ICAPS*, 2004.

[16] Stefan Edelkamp and Tilman Mehler. Knowledge acquisition and knowledge engineering in the *ModPlan* workbench. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[17] D. Bernard et al. Remote Agent Experiment: Deep Space 1. Technical report, National Aeronautics and Space Administration, 2000.

[18] E. Feigenbaum and P. McCorduck. *The Fifth Generation.* Addison-Wesley, Reading, MA, 1983.

[19] R. Fikes, P. Hart, and N Nilsson. Learning and executing generalised robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[20] Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application for theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[21] M. Fox and D. Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61 – 124, 2003.

[22] Garland, Tyall, and Rich. Learning hierarchical task models by defining and refining examples. In *Proceedings of the First International Conference on Knowledge Capture*, 2001.

[23] A. Gerevini and D. Long. Plan constraints and preferences in pddl3: The language of the fifth international planning competition. Technical report, The University of Brescia, Italy, August 2005.

[24] Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

[25] GIPO. Graphical interface for planning with objects. http://compeng.hud.ac.uk/Artform/projects/planform/gipo, 2003.

[26] T. J. Grant. *Inductive Learning of Knowledge-Based Planning Operators.* PhD thesis, de Rijksuniversiteit Limburg te Maastricht, Netherlands, 1996.

[27] T.J. Grant. Assimilating planning domain knowledge from other agents. In *Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group*, 2007.

[28] O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas. Vleppo: A visual language for problem representation. In *Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group*, 2007.

[29] J. Hoffmann. A "tough nuts" track for the ipc. In *Proceedings of the ICAPS*, 2007.

[30] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[31] C. Hogg and H. Munoz-Avila. Learning hierarchical task networks from plan traces. In *Proceedings of the ICAPS'07 Workshop on Artificial Intelligence Planning and Learning*, 2007.

[32] S. B. Huffman, D. J. Pearson, and J. E. Laird. Correcting imperfect domain theories: A knowledge-level analysis. In S. Chipman and A. Meyrowitz, editors, Kluwer Academic Press., 1992.

[33] O. Ilghami, D.S. Nau, H. Munoz-Avila, and D.W. Aha. Learning preconditions for planning from plan traces and htn structure. *Computational Intelligence 21*, 4:88–143, 2005.

[34] C. M. Kadie. Diffy-s:learning robot operator schemata from examples. In *Proceedings of the 5th International Conference on Machine Learning*, San Mateo, California, USA, 1988. Morgan Kaufmann.

[35] D. E. Kitchin. *Object-centred Generative Planning.* PhD thesis, School of Computing and Mathematics, University of Huddersfield, UK, 1999.

[36] Pat Langley and Herbert A. Simon. Applications of machine learning and rule induction. *Communications of the ACM*, 38(11):54–64, 1995.

[37] S. LaVoie, D. Alexander, C. Avis, H. Mortensen, C. Stanley, and L. Wainio. Vicar user's guide, version 2, jpl internal document d– 41 86. Technical report, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1989.

[38] D.B. Lenat and R. Davis. *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill, New York, 1982.

[39] Geoffrey Levine and Gerald DeJong. Explanation-based acquisition of planning operators. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 152–161, California, 2006.

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

[41] G. F. Luger and W. A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving.* The Benjamin/Cummings Publishing Company, Inc., California, 1993.

[42] B. Marthi, J. Wolfe, and S. Russell. Semantics for high-level actions. In *Proceedings of the International Conference on Automated Planning and Scheduling, ICAPS 2007*, 2007.

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

[44] T. L. McCluskey, D. Liu, and R. Simpson. Gipo ii: Htn planning in a tool-supported knowledge engineering environment. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2003.

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

[46] T. L. McCluskey and N. E. Richardson. The induction of operator descriptions from examples and structural domain knowledge. In *Proceedings of the 20th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 181–192, 2001.

[47] T. L. McCluskey, N. E. Richardson, and R. M. Simpson. An Interactive Method for Inducing Operator Descriptions. In *Proceedings of the 7th International Con-*

ference on Artificial Intelligence Planning and Scheduling Systems (aips-2002), 2002.

[48] T. L. McCluskey and M. M. West. Towards the automated debugging and maintenance of logic-based requirements models. In *ASE '98: Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, 1998.

[49] T. L. McCluskey and M. M. West. The Automated Refinement of a Requirements Domain Theory. *Journal of Automated Software Engineering, Special Issue on Inductive Programming*, 6:195–218, May 2001.

[50] T.L. McCluskey, S.N. Cresswell, N.E. Richardson, and M.M. West. Opmaker2: Efficient action schema acquisition. In *Proceedings of the 26th Workshop of the UK Planning and Scheduling Special Interest Group*, 2007.

[51] P. Meseguer and A. D. Preece. Assessing the role of formal specifications in verification and validation of knowledge-based systems. In *Proceedings of the 3rd International Conference on Achieving Quality in Software*, pages 317–328, London, 1996. Chapman and Hall.

[52] R. S. Michalski. Pattern recognition as rule-guided inductive inference. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.

[53] S. Minton and M. Zweben. *Machine Learning Methods for Planning.* Morgan Kaufmann, San Francisco, California, 1993.

[54] Proshanto Mukherji and Lenhart K. Schubert. Discovering planning invariants as anomalies in state descriptions. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, Monterey, US, 2005.

[55] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE, 77*, 4:541 – 580, 1989.

[56] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 665–672, New York, NY, USA, 2006. ACM.

[57] D. J. Pearson. *Learning procedural planning knowledge in complex environments.* PhD thesis, Computer Science and Engineering, The University of Michigan, 1996.

[58] Planform. An open environment for building planners. http://compeng.hud.ac.uk/Artform/projects/planform/, 2007.

[59] Bruce W. Porter and Dennis F. Kibler. Experimental goal regression: A method for learning problem-solving heuristics. *Mach. Learn.*, 1(3):249–285, 1986.

[60] J-F. Puget. Learning invariants from explanations. *Segre*, pages 200–204, 1989.

[61] J. R. Quinlan. Learning efficient classification procedures and their application to chess end games. In *Machine Learning: An artificial intelligence approach*, Los Altos, CA:, 1983. Morgan Kaufmann.

[62] N. E. Richardson. Towards comparing and merging induced operator descriptions. In *Proceedings of the 21st Workshop of the Planning and Scheduling Special Interest Group*, University of Delft, Netherlands, 2002.

[63] N. E. Richardson, T. L. McCluskey, and M. M. West. Towards inducing htn domain models from examples. In *Proceedings of the 25th Workshop of the Planning and Scheduling Special Interest Group*, The University of Nottingham, UK, 2006.

[64] D. Ruby and D. Kibler. Learning to plan in complex domains. *Segre*, pages 180–182, 1989.

[65] S. Russell. Efficient memory-bounded search algo-rithms. In *Proceedings of the ECAI*, 1992.

[66] A. M. Segre. *Machine Learning of Robot Assembly Plans.* Kluwer Academic Publishers, Boston, MA, 1988.

[67] J. W. Shavlik. An empirical analysis of ebl approaches for learning plan schemata. *Segre*, pages 183–187, 1989.

[68] J. W. Shavlik and T. G. Dietterich. *Readings in Machine Learning.* Morgan Kaufmann Publishers Inc., San Mateo, CA, USA, 1990.

[69] Herbert A. Simon. Search and reasoning in problem solving. *Artificial Intelligence*, 21:7–29, 1983.

[70] R. M. Simpson, T. L. McCluskey, W. Zhao, R. S. Aylett, and C. Doniat. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*, 2001.

[71] R.M. Simpson. Gipo graphical interface for planning with objects. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[72] Artificial Intellegence Students. Gipo student domains. http://compeng.hud.ac.uk/planform/gipo/, 2006.

[73] A. Tate. Roots of spar - shared planning and activity representation. *The Knowledge Engineering Review, Special Issue on "Putting Ontologies to Use"*, 13(1):121–128, 1998.

[74] A. Tate, S. T. Polyak, and P. Jarvis. TF Method: An Initial Framework for Modelling and Analysing Planning Domains. Technical report, University of Edinburgh, UK, 1998.

[75] E. Turban and J. E. Aronson. *Decision Support Systems and Intelligent Systems*. Prentice-Hall Inc., Upper Saddle River, New Jersey, USA, 1998.

[76] F. van Harmelen and M. Aben. Structure-preserving specification languages for knowledge-based systems. *International Journal of HumanComputer Studies*, 44:187–212, 1996.

[77] Tiago Stegun Vaquero, Flavio Tonidandel, and Jose Reinaldo Silva. The it-simple tool for modeling planning domains. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[78] S. Vere. *In Pattern Directed Inference Systems*. Academic Press, New York, 1978.

[79] X. Wang. Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.

[80] X. Wang. *Learning Planning Operators by Observation and Practice*. PhD thesis, Computer Science Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsberg, PA 15213, 1996.

[81] X. Wang. Planning while learning operators. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinburgh, Scotland, 1996.

[82] X. Wang and M. Veloso. Learning planning knowledge by observation and practice. In *ARPA/ Rome Laboratory Knowledge-Based Planning and Scheduling Initiative*, Tucson, Arizona, 1994.

[83] T. Winograd. *Understanding Natural Language.* Academic Press, New York, USA, 1972.

[84] P. H. Winston. Learning structural descriptions from examples. In *The Psychology of Computer Vision*, pages 157 – 209, New York, 1975. McGraw-Hill.

[85] M. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. In *Proceedings of ECAI ATAL Workshop*, pages 1–39, 1994.

[86] Kangheng Wu, Qiang Yang, and Yunfei Jiang. Arms: Action-relation modelling system for learning action models. In *Proceedings of the International Conference for Knowledge Engineering in Planning and Scheduling*, Monterez, 2005.

[87] Q. Yang, R. Pan, and S. J. Pan. Learning recursive htn-method structures for planning. In *Proceedings of the ICAPS'07 Workshop on Artificial Intelligence Planning and Learning*, 2007.

[88] Sungwook Yoon and Subbarao Kamphampati. Towards model-lite planning: A proposal for learning and planning with incomplete domain models. In *Proceedings of the ICAPS'07 Workshop on Artificial Intelligence Planning and Learning*, 2007.

[89] Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: looking back, taking stock, going forward. *AI Mag.*, 24(2):73–96, 2003.

[90] M. Zweben, E. Davis, B. Daun, E. Draschler, M. Deale, and M. Eskey. Learning to improve constraint-based scheduling. *Artificial Intelligence*, 58:1–3, 1993.