City Research Online

City St George's, University of London

Citation: McCluskey, T. L. (1988). Experience-driven heuristic acquisition in general problem solvers. (Unpublished Doctoral thesis, The City University)

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/36071/

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

The City University,

Department of Computer Science

"EXPERIENCE-DRIVEN HEURISTIC ACQUISITION IN GENERAL PROBLEM SOLVERS"

by

Thomas Leo McCluskey

Thesis submitted for the degree of Doctor of Philosophy,
to The City University, Northampton Square, London.

December 1988

CONTENTS

ABSTRACT ...5

INTRODUCTION ...6

1: THE TASK FRAMEWORK AND THE PERFORMANCE COMPONENTS

- 1.1 The FM Framework ...8
  - 1.11 State Representation ...8
  - 1.12 Operator Representation ...9
  - 1.13 Operational Semantics ...11
- 1.2 The Performance Components ...12
  - 1.21 FOR: The State Space Search ...12
  - 1.22 MEA: The Linear Goal Directed Search ...13
  - 1.23 NLP: The Non-linear Goal Directed Search ...14

2: THE BASIS OF CHUNK AND MACRO CREATION WITHIN FM

- 2.1 Background to Chunk creation ...21
- 2.2 Regression ...22
- 2.3 The Eight Puzzle Example ...25
- 2.4 Chunk Generalisation and Use ...25
- 2.5 Closed Macro Creation ...28

3: HEURISTIC ACQUISITION IN A LINEAR GOAL DIRECTED PLANNER

- 3.1 C-chunk Creation and Use ...31
  - 3.11 Introduction ...31
  - 3.12 Example Objectives ...32
  - 3.13 Basic C-chunk Creation ...35
  - 3.14 The Use of C-chunks ...41
  - 3.15 C-chunk Examples ...41
  - 3.16 Evolution of the Chunking Technique ...44
- 3.2 Chunk Refinement ...46
  - 3.21 The Form of Heuristic Rules ...46
  - 3.22 Rule Optimisation ...47
  - 3.23 Rule Repair ...49
- 3.3 Experimental Results ...53
  - 3.31 Introduction ...53
  - 3.32 The Robot World Experiments ...53
  - 3.33 The Warehouse World Experiments ...64
  - 3.34 Conclusions ...71

4: TOWARDS HEURISTIC ACQUISITION IN A NON-LINEAR PLANNER

- 4.1 Introduction ...73
- 4.2 The Search through Partial Plan Space ...73
- 4.3 Heuristic Acquisition using E.B.G. ...75
- 4.4 An Example Application ...78
- 4.5 Discussion and Future Work ...80

5: CONCLUSIONS

- 5.1 General Conclusions ...83
- 5.2 Comparisons ...85
- 5.3 Future Directions ...89

APPENDIX A: SAMPLE APPLICATIONS

A.1: A Robot World ...91
A.2: A Warehouse World ...95
A.3: The Eight Puzzle ..103
A.4: A Blocks World ..105
A.5: The Tower of Hanoi problem ..106
A.6: A Macbeth World ..108
A.7: A Boxes World ..110
A.8: The NLP Blocks World ..112

APPENDIX B: SAMPLE PROGRAM OUTPUT

B.1: Robot Results ..114
B.2: Warehouse Results ..117
B.3: Handcrafted Rules ..122

APPENDIX C: PROGRAM LISTINGS ..126

REFERENCES: ..192

AUTHOR'S CONTRIBUTING RESEARCH PAPERS (APPENDIX D)

D.1: "Machine learning of heuristics in general problem solvers" invited paper for the 11th Visegrad Winter School on Foundations of Intelligent Systems, Veszprem, Hungary, January 1987.

D.2: "The anatomy of a weak learning method for use in goal directed search", Proceedings of the Fourth International Workshop on Machine Learning, Morgan Kaufman, June 1987.

D.3: "Combining weak learning heuristics in general problem solvers", Proceedings of the 10th International Joint Conference on Artificial Intelligence, Morgan Kaufman, August 1987.

D.4: "A weak learning method for use in goal directed search", poster given to the Fourth International Workshop on Machine Learning, University of California at Irvine, June 1987, and technical report TCU/CS/87/15.

D.5: "Deriving a correct logic program from the formal specification of a non-linear planner" in Methodologies for Intelligent Systems, Volume 3, North Holland, October 1988.

D.6: "The FM problem solving environment user guide" technical report TCU/CS/88/30, October 1988.

LIST OF FIGURES:

figure 0/1: The FM Learning and Planning System ...8

figure 1/1: A blocks world ...11

figure 1/2: An Eight Puzzle state ...11

figure 1/3: Goal Node processing in MEA ...17

figure 2/1: Build-up of weakest preconditions ...24

figure 2/2: An initial state for the Eight Puzzle ...24

figure 3/1: A robot world fragment ...33

figure 3/2: Another robot world fragment ...33

figure 3/3: And/or tree ...36

figure 3/4: Generalisation hierarchy ...36

figure 3/5: The residual predicate sets ...36

figure 3/6: Strengthening algorithm A ...40

figure 3/7: Strengthening algorithm B ...40

figure 3/8: Chunks approximating a goal's target concept ...51

figure 3/9: Main types of chunk use ...51

figure 3/10: Robot world ...54

figure 3/11: Warehouse world ...65

figure 4/1: Partial Plan Space ...76

figure 5/1: The Prodigy System ...87

DECLARATION

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

ABSTRACT

This thesis investigates the addition of experience learning components to types of general problem solver which have been advocated by the Artificial Intelligence community for use in planning domains. The learning components considered preserve the general applicability of a problem solver while allowing for it to improve it's efficiency when applied to a particular domain. Various heuristic acquisition methods are presented, as well as three types of problem solver; together they have all been implemented in a large integrated system called "FM".

A specific aim is to demonstrate that a particular planning and learning configuration can significantly improve its efficiency by the automatic acquisition of strong heuristics, using a novel heuristic aquisition method. The body of the thesis concentrates on this particular configuration which proved successful in a range of planning applications.

INTRODUCTION

In early 1986, after being engaged for some time in a study of Machine Learning, I had concluded that most of the research carried out in the area lacked immediate evaluation because systems were not fixed to a performance component. For example, a majority of work published in Concept Induction did not include applications for the induced concept descriptions which could act as evaluators for the quality of learning.

On the other hand, in the general-problem-solving area, little work had been carried out on systems which could perform automatic search improvement through experience. Early work on the Soar project [Laird et al 84] indicated that it was possible for general problem solvers to improve their performance significantly in this way; at about the same time Korf [Korf 85] suggested that as well as implanting problem solvers with weak heuristics, to maintain their generality but improve their efficiency they should also be equipped with weak methods for generating strong heuristics (he introduced the phrase 'weak methods for learning' to describe this).

It seemed natural to attack both these issues with one system, and therefore I composed a research plan [McCluskey 86] with the following proposal as its main conclusion:

'[I propose] ..the construction of a "heuristic-learning planning shell" which would tackle the efficiency/generality/power trade-off problems in the following way: The shell would be applicable to a class of problem domains; when it attempts problems in a particular domain it starts by applying the traditional weak methods of general heuristic search but from experience develops a strong model with which to guide subsequent searches'.

The general aim of this thesis is therefore to investigate the usefulness of experience learning techniques when applied to general problem solving systems; specifically the thesis aims to demonstrate that a particular general planner configuration can significantly improve its efficiency by the automatic acquisition of strong heuristics (c-chunks), when supplied with an application domain. This work improves on previous research in that the c-chunks created are very general but useful heuristics, which are refined in the light of future problem solving. Being general, they minimise matching costs yet are more widely applicable than, for example, heuristics which would be obtained using pure Explanation Based Learning techniques of [Mitchell 86].

I followed the path of learning through experience because natural intelligence is inextricably bound to this type of learning; but also experience-based methods have focus. What is to be learned is determined by previous experience or use; systems should improve their performance with the particular type of

problem in which they have had experience. This contrasts with preprocessing methods which lack direction or bias. It is also consistent with the idea that the set of relevant problems should be much smaller that the set of possible problems for experience learning to be worthwhile ([Van der Velde 86], p.13, puts forward this idea in his work on the conversion from deep to shallow knowledge in a second generation expert system).

I assume the reader is familiar with basic concepts in the field of Machine Learning, and in particular the relationship between the sub-field studied here and other sub-fields: other works have adequately covered this, e.g. [Carbonell 83], [Mitchell 83]. I have restricted this thesis to learning from the problem solving trace which is generated in finding successful action sequences, or operator sequences, as we shall call them. As with natural learning about problem solving, this demands that simpler problems are posed initially. Apart from setting the problems, no other user intervention is required.

Contents Overview

Chapter 1 provides a gentle introduction to the idea of states and operators, and introduces the problem specification language ('task framework') for my FM problem solving system shown in figure 0/1, which has the Strips-assumptions at the heart of its operator representation. Then the three control strategies, which form the performance components, are described. These are the (quite standard) best-first state space search ('FOR'), goal directed linear search ('MEA'), and goal directed constraint posting non-linear search ('NLP'). The latter, being the most complex of these, is defined and implemented using a constructive formal specification, described in [McCluskey 88a] and appendix D.5.

Chapter 2 forms a basis for the succeeding chapter by defining basic chunks and macros. The established idea of using goal regression on declaratively specified operators to create heuristics is formally defined for any FM-specified operator schemas. A simple example is used to clarify this form of chunk and macro creation.

Chapter 3 is the core of this thesis: it describes a developed, reasonably successful attempt at meeting the proposal quoted above from [McCluskey 86], for a linear goal directed search strategy. The heuristic acquisition method described in 3.1 is novel: useful heuristics for an operator, being the last in a successful sub-sequence of operators, are approximated by the similarities between the weakest precondition of the sequence and the operator's precondition; then they are backed up by a background knowledge base, and strengthened by a discrimination technique which draws on both successful and failed operator instantiations from the problem solving trace.

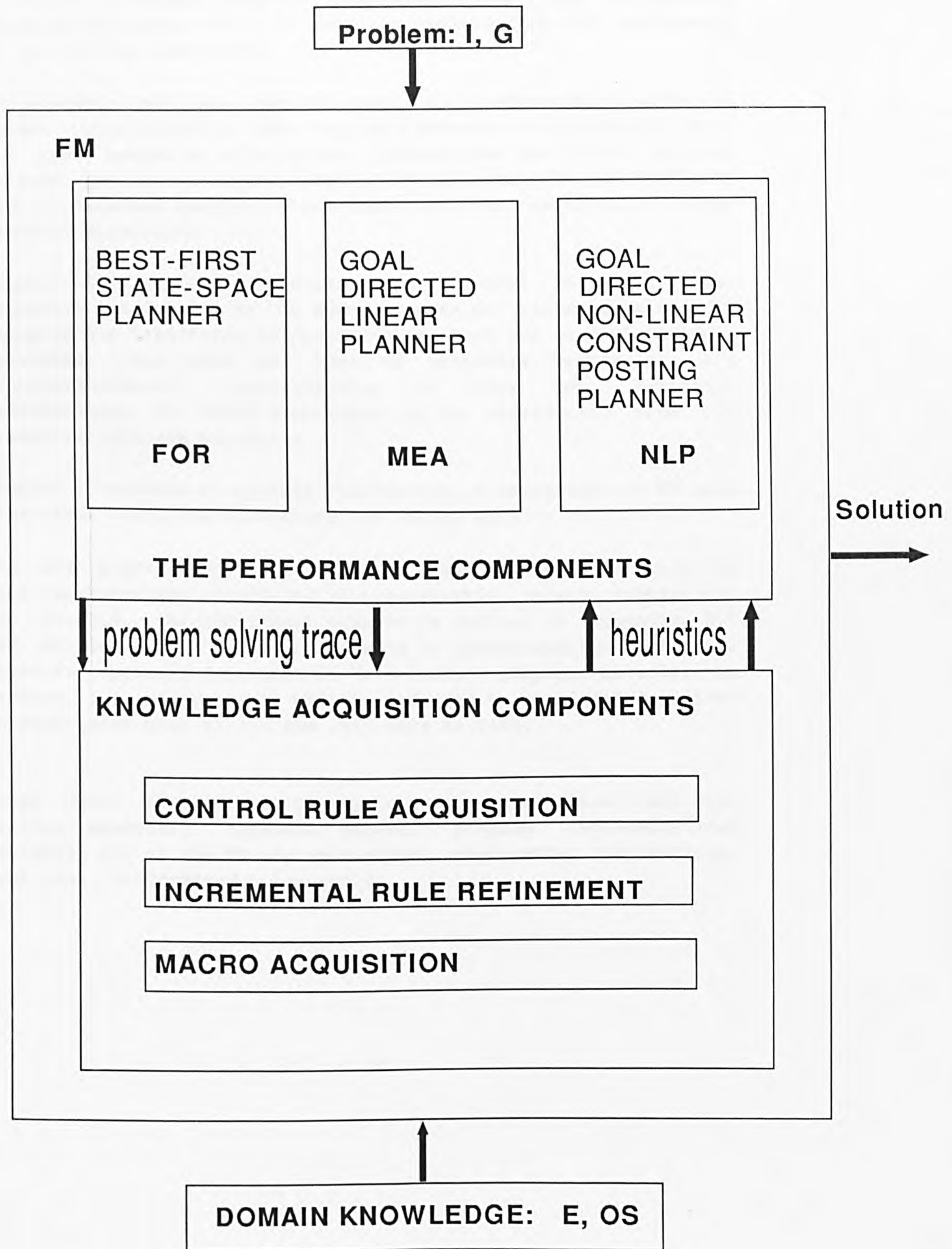

FIGURE 0/1: THE FM LEARNING AND PLANNING SYSTEM

The techniques in 3.1 produce heuristic preconditions for operator - goal combinations which must be refined and integrated. In 3.2 I define a general form for heuristic rules, and incremental learning techniques which perform the optimisation and refinement of the initial heuristics.

3.3 provides test runs from two sample applications which show the system significantly improving its performance in both the time and space needed to solve tasks. Comparisons are made between various planner configurations: with no heuristic acquisition, with handcrafted heuristics and with heuristic acquisition using chunking algorithms.

Chapter 4 is exploratory in its scope, in that it investigates heuristic acquisition in the more powerful non-linear planner. It explores how transforms in partial plan space can be declaratively specified, and uses the idea of transform regression and Explanation-Based Generalisation to show how heuristic preconditions for these transforms can be constructed from old transform solution sequences.

Chapter 5 contains my general conclusions, a comparison of FM with some rival work, and directions for future work.

The six papers comprising appendix D are an important part of this work and are considered to support this thesis submission. For example, the non-linear planner is defined in appendix D.5 and should be read as a prerequisite to understanding chapter 4. Likewise appendix D.6, the FM User Guide, supports material in section 1.1. Appendices will be referred to when needed either through their name or via the reference section.

Final Note: All of this thesis is my own work (i.e. all the written material, appended papers, program implementations including all of the FM implementation, application definitions, test runs, handcrafted rules etc.)

1. THE TASK FRAMEWORK AND PERFORMANCE COMPONENTS

1.1 The FM Framework

In this chapter I develop the representational framework for the FM problem solving and planning system. I will use the framework to define the three performance components for which heuristic acquisition will be discussed in chapters 2,3 and 4.

This particular section defines the task framework "(I,G,E,OS)", which will be referred to throughout the rest of the thesis. Apart from developing notation, the chapter contains introductory material, included for completeness, which can be skipped by those familiar with Strips-type frameworks.

1.11 State Representation

A STATE DESCRIPTION 'S' is defined to be a conjunction of ground, i.e. fully instantiated predicates. It models the changeable facts in some application, at a certain point in time. For example in the application of stacking blocks on a table, the state description that represents figure 1/1 could be:

```
on(a,b)&on(b,table)&clear(a)&clear(c)&
on(c,table)&clear(d)&on(d,table).                         -1(1)
```

Persistent facts such as 'clear(table)' may be omitted but made implicit in the action representation described in 1.2.

We supplement state descriptions with an ENVIRONMENT 'E' which defines the unchanging background facts of an application. In the blocks world this may trivially contain typing information, but in other applications this may be any persistent information deemed relevant. The environment for the block's world could be:

```
type_of(a,block)&type_of(b,block)&
type_of(c,block)&type_of(d,block)                         -1(2)
```

Both the environment and states may each be supplemented by set of rules, which are primarily for the use of FM's learning components. Thus we may split E and S:

E = E.f (facts) + E.r (rules)
S = S.f (facts) + S.r (rules)

but note that the problem solving components must be supplied with the full theory as ground facts, and so we assume:

Theory(E) = E.f and
Theory(S) = S.f,

that is the rules should only show the connections between facts already present (see appendix A for examples of domains with

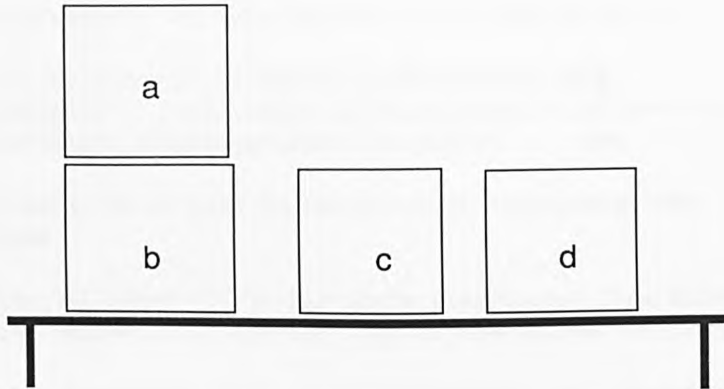

figure 1/1: A blocks world

p1 tile1	p2 tile2	p3 tile3
p4 tile4	p5 tile8	p6 tile6
p7 blank	p8 tile7	p9 tile9

figure 1/2: An Eight Puzzle state

rules, and for examples of rule syntax).

Another example application is the children's puzzle, commonly known as the Eight Puzzle; the state shown in figure 1/2 is represented by:

```
at(tile1,p1)&at(tile2,p2)&at(tile3,p3)&at(tile4,p4)&
at(tile8,p5)&at(tile6,p6)&at(blank,p7)&at(tile7,p8)&
at(tile9,p9).                                           -1(3)
```

The environment for this puzzle can be represented as:

```
next(p1,p2)&next(p1,p4)&next(p2,p3)&next(p2,p5)&
next(p3,p6)&.....type_of(p1,position)&type_of(p2,position)&...
type_of(tile1,tile)&type_of(tile2,tile)&...   etc       -1(4)
```

Of course it is easy to devise other representations for these problems.

An Initial State 'I' is any state description from which problem solving begins. 1(1) and 1(3) can be considered initial states.

A Goal Condition 'G' is a conjunction of ground predicates specifying a SET of state descriptions, to which problem solving must be directed (although G may contain existentially quantified variables for the performance component specified in 1.23).

Examples of goal conditions for the blocks and eight puzzle worlds are:

```
on(b,c)&on(a,b)                                        -1(5)

at(tile1,p1)&at(tile2,p2)&at(tile3,p3)&at(tile4,p4)&
at(blank,p5)&at(tile6,p6)&at(tile7,p7)&at(tile8,p8)&
at(tile9,p9)                                           -1(6)
```

The set of states specified are exactly those that contain the goal condition. Several state descriptions satisfy 1(5), but 1(6) specifies precisely one state - the usual goal for this puzzle.

1.12 Operator Representation

I will model actions in these worlds as operators that change state descriptions instantaneously. A most convenient way is by simply having an operator:

- delete predicates from the old state to which the action is applied;

- add predicates to "create" the new state.

Any unaffected predicates therefore remain true in the new state.

This form of operator side-steps a famous problem in A.I. called 'The Frame Problem', which is encountered when we try to model everything in a first order logic. In such a formulation, every operator would have to have attached to it a set of 'frame axioms' which mention each predicate that is unaffected - making operators of unmanageable size.

Since this representation is declarative, it also makes it easier for the system to reason about its operators. The most general form of an Operator in FM is a 7-tuple, having three preconditions and three postconditions:

```
(    name:     <name>(<parameters>),

     filter:   <filter preconditions>,

     check:    <environmental preconditions>,

     precon:   <state preconditions>,

     padd:     <add set>,

     add:      <side effects>,

     del:      <delete set>   )
```

The last six components are sets of predicates. Each component of an operator O can be referred to by its component name, or for brevity by a selector function: say O.n, O.e, O.f, O.p, O.a, O.s and O.d respectively.

O.e, O.f, and O.p are the preconditions for the operator. We stipulate that the filter predicates O.f must be identical to a subset of O.p (O.f will only be used in goal directed search, and performs a similar function to the preconditions in SIPE's operators [Wilkins 84]). O.a and O.s constitute the full add-set for the operator.

Parameters in operators are represented below with capital letters. They have SCOPE throughout the whole of the tuple; one may think of the operator as being represented as a logic term. The name must contain at least all those parameters that occur in the precondition, so that the instantiation of O.n yields a unique application of the operator. Two special predicates are recognised by the system: 'ne(X,Y)' meaning X is not equal to Y, and a 'type_of' predicate with the obvious meaning. For instance when FM learns new operators or rules it knows that parameters of different types can't be instantiated to the same constant, and so don't need any further binding restrictions.

Using this representation, two blocks world operators could be

defined as:

```
(    name:     unstack(Block1, Block2),
     check:    type_of(Block1,block)&type_of(Block2,block)&
               ne(Block1,Block2),
     filter:   nil,
     precon:   on(Block1,Block2)&clear(Block1),
     padd:     on(Block1,table)&clear(Block2),
     add:      nil,
     del:      on(Block1,Block2) )

(    name:     stack(Block1, Block2),
     check:    type_of(Block1,Block)&type_of(Block2,Block)&
               ne(Block1,Block2),
     filter:   nil,
     precon:   on(Block1,Object)&clear(Block1)&clear(Block2),
     padd:     on(Block1,Block2)&clear(Object),
     add:      nil,
     del:      on(Block1,Object)&clear(Block2) )               -1(7)
```

An operator for legally moving the tiles in the Eight Puzzle is:

```
(    name:     move(Tile,P1,P2),
     check:    next(P1,P2)&ne(Tile,blank)&type_of(P1,position)&
               type_of(P2,position)&type_of(Tile,tile),
     filter:   nil,
     precon:   at(blank,P2)&at(Tile,P1),
     padd:     at(blank,P1)&at(Tile,P2),
     add:      nil,
     del:      at(blank,P2)&at(Tile,P1)  )                     -1(8)
```

(see appendix A for more examples)

1.13 Operational Semantics

An OPERATOR APPLICATION of operator O on state description S with respect to environment E is possible if there exists a ground instance O' of O such that S contains O'.p and E contains O'.e, i.e. S satisfies the preconditions of O. In general there may be more than one such ground instance, but application of instance O' to S can be defined as the following state description (N.B. 'U' and '-' will be the symbols used for set union and difference, respectively):

O'[S] = { S - O'.d } U O'.a U O'.s

In other words O'[S] is the state produced by first removing the instantiated operator's delete set from S, then adding its add set and side effects to S.

A TASK is defined as a four-tuple (I,G,E,OS):

(initial state, goal condition, environment, operator set)

e.g. (1(1), 1(5), 1(2), 1(7)) is a task:

A task (I,G,E,OS) is ACHIEVED when a sequence of operator instances O'(1) ... O'(n-1), O'(n), taken from set OS, is found, such that O'(n)[O'(n-1)[... O'(1)[I] ...]] contains G.

Of course this expanded STRIPS-type framework is still a little 'loose'. It should be pointed out, for example, that a necessary condition for task achievement is that each predicate in the goal condition can unify with a predicate in some operator's add set. More details of this framework can be found in the FM user guide [McCluskey 88b] and in appendix D.6.

1.2 The Performance Components

I shall now describe the three planners that were used in the experimental work that lead to this thesis, using the notation of the task framework defined above. Their implementation is given in appendix C.

1.21 The State Space Search: FOR

FOR contains the simple strategy of best-first search through the space of states. It starts by generating all states from the initial state by every possible instantiation of operators from the operator set, then expanding each of the generated states likewise. Each state is actually represented within a node containing information such as the operator sequence required to reach it, and the cost of that particular partial solution. Optionally, the following features may be included in this search:

(1) The user can supply an operator's inverse so that the strategy will avoid applying an operator and then its inverse sequentially. In the environment E, the user states:

inverse(O.n, O'.n).

for example:

inverse(move(Tile,P1,P2), move(Tile,P2,P1)),

This will stop sub-sequences being generated which move a tile to a square then immediately move it back again. Of course, because of the declarative representation for operators, generation of inverses could be easily automated and executed during a pre-processing stage.

(2) The strategy can keep a list of all states expanded, and check through them to make sure an identical one is not re-expanded.

This heuristic can cause more matching overheads than its worth in some problem domains and so is optional.

So costly is this strategy that acquiring heuristics by experience (see chapter 2) only works well on domains such as the 8-puzzle!.

1.22 The Goal Directed Search: MEA

MEA is so named because it implements some of the principles of 'Means-Ends Analysis' in a similar fashion to STRIPS [Fikes et al 72]. Specifically it forms the difference between a goal state and initial state as a set of predicates; then it treats the preconditions of the operators that achieve one or more of the difference predicates as new goal conditions.

Hence the backward search of FM proceeds in a goal reduction manner, starting with the initial goal, through a space of goal nodes. This space is searched by storing 'open' nodes in a priority queue. Elements are given an initial priority depending on a weak heuristic, represented as a rule below, which depends on the initial state of that node and the goals it must solve.

In the version of MEA used in chapter 3, this was simply:

priority = large number -
2*(no. of goal predicates not solved by initial state)

To ensure fairness, the priorities are all incremented after each node expansion.

Each goal node can be modelled as a 5-tuple:

(Goal, Initial State, Ancestors, Purpose, Trace),

The Trace records attempts to solve the Goal; these are made up by combining operator subsequences together, which solved subsets of Goal. The Purpose records why the goal node was created, which can be of two types:

(a) to solve the unsatisfied preconditions of an operator;

(b) to solve a Goal from an advanced Initial State (not I in (I,G,E,OS)) in which one or more of Goal's predicates has been achieved.

The 'Ancestor' slot is simply a record where the node's ancestry is kept. One use of this slot is to allow MEA to identify nodes with recurring goals and delete them from the search.

Goals, expressed as conjunctions of predicates, are initially assumed to be decomposable: when a goal node is activated, operator instantiations which add goal predicates have their unsatisfied preconditions form another goal node, unless they are

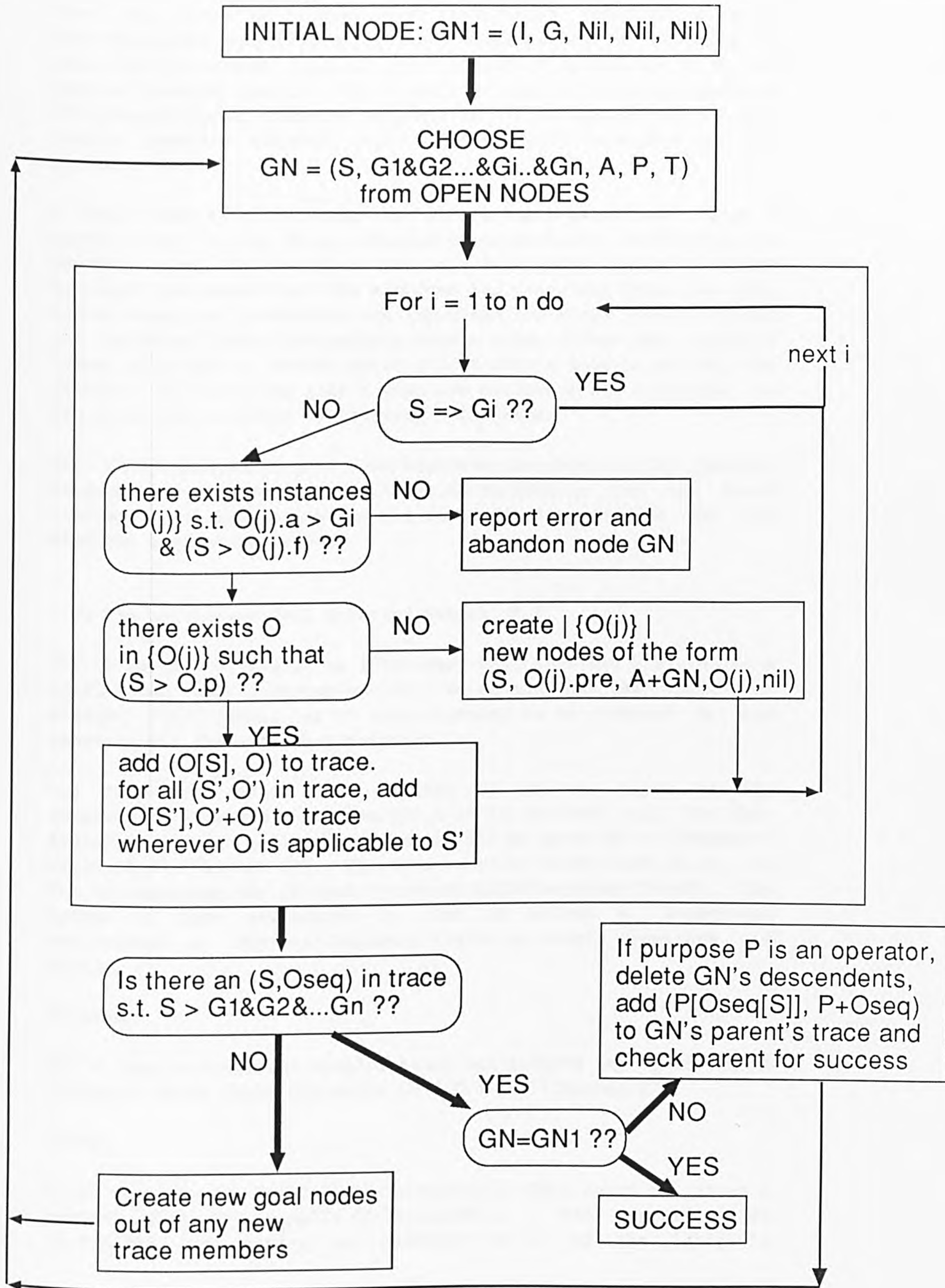

figure 1/3: Goal Node processing in MEA (">" means "contains")

already satisfied in which case those operators are applied to the initial state and the result recorded in the trace. If the parent's main goal is solved as a result, the process recurses.

When the trace of a goal node eventually contains a state satisfying its goal (via an operator sequence Os), we say that the goal node is solved, and all nodes which are successors of it are removed from the search. If it was activated to solve an operator O's preconditions, then the sequence Os + O is applied to the goal node's parent's initial state and the result recorded in the parent's trace.

A goal node's initial state may be the state inherited from a parent node, or may be an advanced state partially satisfying the parent's goal. The latter is the case when goals cannot be solved by simple decomposition; MEA examines the trace and forms new goal nodes whose goal predicates are inherited but whose initial states are selected from intermediate states taken from the parent's trace. This type of search ensures that when a node is solved, the attempts to solve the node's goal are declaratively available in the trace for scrutiny by learning components.

The whole process of goal node expansion is shown in the process diagram of figure 1/3. The level above this is the top level strategy: it simply chooses the next node to expand as the one with the highest priority.

1.23 The Non-Linear Goal Directed Search: NLP

NLP is defined precisely in [McCluskey 88a, appendix D.5], using a model-based specification method. We assume that the reader has studied this paper, as it is background to to chapter 4, and contains all the necessary notation.

One important detail that is left out of the paper is its comparison with the work from which it is derived, e.g. is the specification for goal achievement in NLP as powerful as Chapman's original in [Chapman 87]. The theorem below shows that it is. As far as comparing the FM task framework with Chapmans 'Tweak', the former is more structured in that it allows a background environment E, whereas Chapman's relied on simply operators and states.

Theorem:

NLP's goal achievement specification can achieve any goals that Chapman's Modal Truth Criterion (M.T.C.) can [Chapman 87].

Proof:

First we state the Modal Truth Criterion in FM's notation. Given a partial plan PP = pp(Os,Ts,Ps,As,Es), a task specification (I,G,E,OS), and taking an operator O to be of the (slightly

simplified) form (O.n,O.e,O.p,O.a,O.d), then if (P,O) is in Ps, A in Os:

```
P is achieved at operator O  in PP by A (called 'achieved(P,O,A)')
if there exists A in Os:
      [(there exists Q in A.a: P = Q) &
       before(A,O,Ts) &
       for all C in Os:
          [before(O,C,Ts) V
           for all Q in C.d:
               [not(unify(Q,P,Es) V
                there exists WK in Os:
                    [before(C,WK,Ts) & before(WK,O,Ts) &
                     there exists R in WK.a:
                         [P=Q => P=R] ]  ]  ]  ]
```

The modal operators (originally in Chapman's M.T.C.) are included implicitly in the definitions of predicates 'before' and 'unify', which are given in [McCluskey 88a, appendix D.5].

Lemma:

In whatever situation a white-night (WK) has to be used for goal achievement, it can be taken to be the asserter A.

Considering the diagram:

```
----> C ----> A ----> C ----> WK ----> (P) O ----> C ----> ......
```

C represents possible positions for clobberers - i.e. those operators whose delete list contains a predicate Q which could possibly unify with P. If a clobberer cannot be necessary before A or necessary after O, then the only other way of goal achievement for P is by WK: if Q unifies with P, then WK has a predicate in its add list which can be made identical with P.

But in every situation where this latter case follows, we may choose WK to be the asserter instead of A. At worst this causes more instantiation of the partial plan (as admitted by Chapman himself in [Chapman 87]), so it can be seen that this does not lessen the power of the planner.

So taking A = WK we have: achieved(P,O,A) =

```
there exists A in Os:
      [(there exists Q in A.a: P = Q) & before(A,O,Ts) &
       for all C in Os:
          [before(O,C,Ts) V
           for all Q in C.d:
               [not(unify(Q,P,Es) V
                there exists A in Os:
                    [before(C,A,Ts) & before(A,O,Ts) &
                     there exists R in A.a:
                         [P=Q => P=R] ]  ]  ]  ]
```

But P=Q in A.a in the first conjunction, and 'before(A,O,Ts)' is true by the second, hence:

```
achieved(P,O,A) =
there exists A in Os:
      [(there exists Q in A.a: P = Q) & before(A,O,Ts) &
       for all C in Os:
          [before(O,C,Ts) V
           for all Q in C.d:
               [not(unify(Q,P,Es) V
                there exists A in Os:
                    [before(C,A,Ts) &
                     true & true
                                        ] ] ] ]
```

and since before(C,A,Ts) does not depend on Q this is logically equivalent to

```
there exists A in Os:
          [(there exists Q in A.a: P = Q) &
          before(A,O,Ts) &
          for all C in Os:
             [ C = O V
               C = A V
               before(O,C,Ts) V
               before(C,A,Ts) V
               (for all Q in C.d: not(unify(Q,P,Es)) ]
```

which is my specification of goal achievement. QED.

This specification is an integral part of the constructive specification given in NLP, and it is verified that the design and implementation are faithful to it in [McCluskey 88a,appendix D.5].

2 THE BASIS OF CHUNK AND MACRO FORMATION WITHIN FM

2.1 Background to Chunk creation

In this chapter the chunking method developed is used exclusively in FOR, the forward state space search, but the other chunking techniques developed later are based on it and use the same notation. The material is mostly foundational, for those not familiar with heuristic/macro generation from goal regression, but as such contributes to the body of the thesis in chapter 3.

The theory of Explanation Based Generalisation (EBG) was postulated in [Mitchell 86] and has had considerable influence since then (see [Hirst 87], [DeJong and Mooney 87] for example). Since it is already a familiar theory to Machine Learning workers, we use it's framework to explain the basis of chunk and macro creation in FM. Essentially this corresponds with my original idea of "model based generalisation of a successful operator sequence's weakest precondition" [McCluskey 87b p.136], which means that the generalisation is justified by the operator and environment model supplied by the user.

Recall that there must be four components involved in EBG (following [Mitchell 86]):

(a) the target concept: what is to be learned;

(b) operationality criteria: the form in which the learned concept description must be encoded, the operational description of (a);

(c) the domain theory: a 'deep' body of knowledge containing a non-operational definition of (a);

(d) an example of (a). ..2(1)

The process of EBG is first to build a proof tree showing that (d) is an example of (a), (drawing from the theory in (c)) such that all leaves in the tree are in form (b). Then each leaf is generalised as much as possible without falsifying the proof, and finally the acquired heuristic is defined as

conjunction of generalised leaves --> concept ..2(2)

Mitchell et al, using their LEX2 system, exemplify how EBG is applied to heuristic acquisition in [Mitchell et al 86, p. 62-65]. We will use the same line of argument, but generalise to systems of FM's power (recall that unlike FM, LEX2 was tied to a specific application i.e. Symbolic Integration, which has a fixed goal, rewrite rules as operators, and a relatively simple generalisation space for operator preconditions). Consider the solution sequence to some task (I,G,E,OS), paired with the corresponding intermediate state descriptions:

[(O(1),S(1)), ... ,(O(i),S(i)), ... (O(n),S(n))],

where O(i)[S(i-1)] = S(i), S(n) contains G, S(0) = I.

Using 2(1) we have:

(a) is a heuristic precondition for O(i) defining the set

{ S : O(i) is the correct operator to apply
when in state S to achieve goal G }

(b) is a generalised state expression;

(c) is the set of general axioms for state space search, as in [Mitchell et al 86, p.62] as well as the specific regression theory developed in section 2.2 following;

(d) is [(O(1),S(1)), ... ,(O(i),S(i)), ... (O(n),S(n))],

Following [Mitchell et al 86, p63], the main operational 'leaf' in the proof tree is:

```
matches( (S(i-1) U E) ,
        regress(O(i),regress(O(i+1),.. regress(O(n),G).. )) )
                                                              ..2(3)
```

where 'regress' is defined in 2.2 below, and matches(S,C) means C is some condition that is satisfied by S; if C and S are simple sets of ground predicates, then this is equivalent to 'S contains C'. If C contains variables, then this is equivalent to 'there exists some instantiation of C which is contained by S'.

(S(i-1),G) is then regarded as an instance of the set of all (State, Goal condition) pairs under domain definition of E and OS, to which O(i) is best applied. From the proof tree it can be seen that we are using no other characteristic of S except this match, so any state that matches the regression expression will also do.

2.2 Regression

We define the regression of a conjunction of ground predicates G through an operator O as its weakest precondition written "wp(O,G)":

wp(O,G) = { G': O is applicable to G and O[G'] contains G}

This is analogous to the weakest precondition used in the program proving literature (e.g. [Gries 83]): wp(O,G) specifies the set of all states such that execution of command(s) O from any one of them will terminate in a state satisfying G.

Regression works well where there is a declarative definition for

operators ([Porter and Kibler 85] use the word "transparent" to describe appropriate operator schemas); in program proving the axiomatic semantics of control constructions are used. For FM, where O is a totally instantiated operator from OS, and S is a state expression:

```
O[S] = (S - O.d) U O.a U O.s
```

and since O.p U O.e are supposed to be the necessary and sufficient condition for O's application, it follows that the weakest precondition is the conjunction of ground predicates given by the set:

```
wp(O,G) = (G - (O.a U O.s)) U O.p  U O.e                        ..2(4)
```

Now consider the case of a sequence of operators [O(1), ..., O(n)]. Following [Gries 83, page 115] we can define:

```
wp([O(1), ..., O(n)],G)  =
      wp(O(1),wp(O(2), ... wp(O(n-1),wp(O(n),G)) ... ))
```

Letting:

```
P(0) = G
P(1) = wp(O(n),G)
P(2) = wp(O(n-1),wp(O(n),G))
P(3) = wp(O(n-2),wp(O(n-1),wp(O(n),G)))   etc,
```

and for any operator O, let O.as = O.a U O.s, then from 2(4):

```
P(1) = (P(0) - O(n).as) U O(n).p U O(n).e
P(2) = (P(1) - O(n-1).as) U O(n-1).p U O(n-1).e
P(3) = (P(2) - O(n-2).as) U O(n-2).p U O(n-2).e  etc, and hence
```

```
P(0) = G  and for  1 <= j <= n,
P(j) = (P(j-1) - O(n-j+1).as) U O(n-j+1).p U O(n-j+1).e,     ..2(5)
```

If we define wp([O(i), ... ,O(n)],G) = WP(i,n), then (see fig 2/1)

```
WP(i,n) = P(n-i+1) = (P(n-i) - O(i).as) U O(i).p U O(i).e    ..2(6)
```

In fact we will separate these out into two disjoint components:

```
WP(i,n) =  WPs(i,n) U WPe(i,n), where:
```

```
WPe(i,n) =  O(i).e U ... U  O(n).e
WPs(i,n) = P(n+i-1) - WPe(i,n)                                  ..2(7)
```

This construction is well defined because FM's operators' components must be restricted to conjunctions of predicates, avoiding the problems of obtaining disjunction through regression as mentioned in Porter and Kibler's critique of analytical goal regression [Porter and Kibler 85].

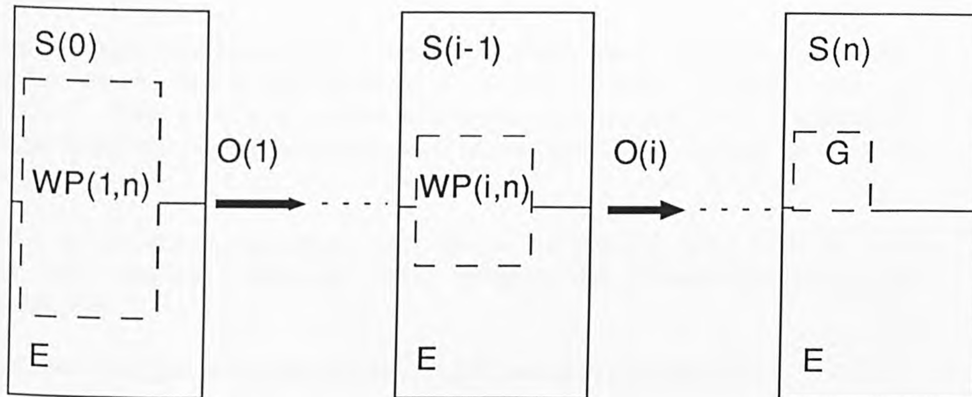

figure 2/1: Build-up of weakest preconditions

p1 tile6	p2 tile2	p3 tile3
p4 tile1	p5 blank	p6 tile4
p7 tile8	p8 tile7	p9 tile5

figure 2/2: An Initial State for the Eight Puzzle

2.3 The 8-Puzzle Example

We will use the 8-puzzle example introduced in chapter 1 to demonstrate the ideas of chunk creation. (The similarity to Soar's chunks for this type of strategy can be seen by referring to [Laird 86] which also uses the 8-puzzle problem as an example application domain).

Recall that the board has 9 numbered positions (p1,p2,..,p9) on which there are 8 numbered tiles (tile1,tile2,..,tile8) and a 'blank'. The idea is to find a sequence of moves (i.e. swapping a tile with the blank horizontally or vertically) linking a pair of states, (I,G).

Let I be as shown in figure 2/2, let G = at(tile1,p7), then FM with the FOR search strategy will output the solution sequence of operators:

[move(tile7,p8,p5),move(tile8,p7,p8),move(tile1,p4,p7)]

Using the equations 2(7) and the operator definitions in appendix A, we have:

WPs(1,3) =

at(tile7,p8)&at(blank,p5)&at(tile8,p7)&at(tile1,p4)

WPe(1,3) =

next(p4,p7)&next(p8,p5)&next(p7,P8)&ne(tile1,blank)&
ne(tile8,blank)&ne(Tile7,blank)

and hence an 'ungeneralised chunk' is formed as:

(O(i),G,WPs(i,n),WPe(i,n)) for n = 3 and i = 1,2 and 3.

In future problem solving, if the current state S contains WPs(i,n) and the current goal = G, then O(i) is the operator instantiation that should be chosen to continue the search (WPe(i,n) is superfluous in this ungeneralised version).

2.4 Chunk Generalisation and Use

To recap, our regression equation 2(5) can be applied to a solution sequence for any task (I,G,E,OS); for i, 1 =< i =< n, it will produce the subset of S(i-1)&E.f which is necessary and sufficient for O(i), ... ,O(n) to succeed, that is the expression WP(i,n).

This is a generalised expression in the sense that it specifies the set of all states which contain it. It can be generalised further: recall from section 1.13 that for an operator to be applicable, its preconditions much match the current state and

environment. Using this matching that proves each O(j) is applicable, for i =< j =< n, wherever a constant in S(j-1) was substituted for a variable in O(j).p or O(j).e, the constant can be generalised to a variable, without falsifying the proof. This is justified because any property of the particular identity of the constant used would be stated in O(j).e and will itself be generalised to the same variable, becoming a binding constraint. Whenever a constant matched a constant in the proof, however, this must stay in unchanged. Also the following restrictions are imposed:

-Identical constants must be generalised to the same variable instance only when they were both substituted for the same variable instance in some operator's preconditions;

-To perform 'careful' generalisation (as defined in [Kodratoff 84]), different variable instances from O(j) should have binding restrictions added to the target chunk, so that they may not be instantiated to the same constant later, when the chunk is in use. Since FM's variables should be given a type in the domain definition this restriction is not needed unless the variables are of the same type.

STRIP'S macrops system violated the second rule and over-generalised macro operators (as pointed out in [Fikes et al 72]). The generalisation operator we have just described will be called GP: it carefully generalises constants to variables in an ungeneralised chunk as long as the precondition proofs of the original solution sequence are not violated.

A simple chunk is then defined as the logical term:

GP(G,O,WPs(i,n),WPe(i,n)) for i = 1,2, .. n-1.

A chunk C can be used in future searches to expand a state space S generated in the search for a solution to some task (I,G,E,OS) as follows: Given a chunk C = (Gc',Oc',Sc',Ec'), If there exists a ground instantiation of C, say (Gc,Oc,Sc,Ec), such that

```
  G = Gc
& S => Sc
& E => Ec
then apply Oc.                                          ..2(8)
```

Alternatively, chunks may be used within future searches in test mode: all new states are created, and ones which are created through operators that satisfy 2(8) are chosen to expand next, the others discarded or given a lower priority.

Expand mode is more efficient in the sense that it avoids generating states in the first place; on the other hand generate and test mode is desirable if chunks are heuristics favouring expansions rather than considering them as dead certainties. The latter is true in the case of the C-chunks described in chapter 3.

2(8)'s first conjunction could be generalised to 'G contains Gc' which would allow chunks to apply to subsets of the main goal: in the context of other interfering goal predicates, however, operator instantiations recommended may not turn out to be the best ones.

Back to example 2.3. Assuming that names starting with capitals are variables, then

```
GP(G,O(1),WPs(1,3),WPe(1,3)) =

(ch1,move(Tile7,P8,P5),
     at(Tile1,P7),
     at(Tile7,P8)&at(blank,P5)&at(Tile8,P7)&at(Tile1,P4),
     next(P4,P7)&next(P8,P5)&next(P7,P8)&ne(Tile1,blank)&
       ne(Tile8,blank)&ne(Tile7,blank)&ne(P4,P7)&
       ne(P4,P8)&ne(P4,P5)&ne(Tile1,Tile8)&ne(Tile1,Tile7)&
       ne(P7,P8)&ne(p7,P5)&ne(Tile8,Tile7)&ne(P8,P5)  )


GP(G,O(2),WPs(2,3),WPe(2,3)) =

(ch2,move(Tile7,P8,P5),
     at(Tile1,P8),
     at(Tile7,P8)&at(blank,P5)&at(Tile1,p7),
     next(p7,P8)&ne(Tile1,blank)&next(P8,P5)&
     ne(Tile7,blank)&ne(p7,P8)&ne(p7,P5)&
     ne(Tile1,Tile7)&ne(P8,P5)    )
```

Here we see that constant 'blank' is not generalised since it appears as a constant in the operators. The 'ne' predicate adds the appropriate binding restrictions mentioned above. Note that this is a logical term in the sense that each component shares variable identifiers, and constants in the goal and operator slots are generalised with ones in the other two components.

For an example of chunk use, consider the task where I is shown in figure 2/2, G = at(tile2,p3) and E and OS are as in appendix A. In search FOR, from the initial state I, the operators generated would be:

```
{ move(tile4,p6,p5), move(tile2,p2,p5),
  move(tile7,p8,p5), move(tile1,p4,p5) }
```

Matching the operator move(tile4,p6,p5) and goal G with ch1 we have:

```
(ch1,move(tile4,p6,p5),
     at(tile2,p3),
     at(tile4,p6)&at(blank,p5)&at(Tile8,p3)&at(tile2,P4),
     next(P4,p3)&next(p6,p5)&next(p3,p6)&ne(tile2,blank)&
       ne(Tile8,blank)&ne(tile4,blank)&ne(P4,p3)&
```

```
ne(P4,p6)&ne(P4,p5)&ne(tile2,Tile8)&ne(tile2,tile4)&
ne(p3,p6)&ne(p3,p5)&ne(Tile8,tile4)&ne(p6,p5)  )
```

and since I matches the third component of ch1 with bindings tile3/Tile8 and p2/P4, making the expression

```
next(p2,p3)&next(p6,p5)&next(p3,p6)&ne(tile2,blank)&
  ne(tile2,blank)&ne(tile4,blank)&ne(p2,p3)&
  ne(p2,p6)&ne(p2,p5)&ne(tile2,tile3)&ne(tile2,tile4)&
  ne(p3,p6)&ne(p3,p5)&ne(tile3,tile4)&ne(p6,p5)
```

consistent with E. Since none of the other operator instances instantiate the chunk to give a consistent match with I or E, then the first operator will be chosen for application. ch2 will likewise choose the correct operator at the next step, and infact the goal will be solved with no search. This shows a simple example of chunking, taken directly from the FM implementation, demonstrating what is termed 'symmetrical transfer' in [Laird 86].

2.5 Closed Macro Creation

Since ch1 above held the weakest precondition of the sequence of operators with respect to the generalised goal, it could have put forward the rest of the sequence as the solution immediately. A structure in FM that can in fact do this is called a 'closed macro' (introduced in [McCluskey 87a] and [McCluskey 87c]). It is a compiled operator which takes the place of the sequence O(i), ..., O(n). Closed macros take a form similar to that of a primitive operator:

```
( macroN( <all variables occurring in WP(i,n) and G> )

  check:  WPe(i,n)

  macrop: <list of primitive operators>

  precon:  WPs(i,n)

  padd:   G

  add:    Sn - (S0 U G)

  del:    nil    )
```

For example, FM could form a macro analogous to ch1:

```
(  macro1(Tile7,P5,Tile8,P8,Tile1,P4,P7)

  check:  next(P4,P7)&ne(Tile1,blank)&next(P7,P8)&
          ne(Tile8,blank)&ne(Tile7,blank)&ne(P4,P7)&ne(P4,P8)&
          ne(P4,P5)&ne(Tile1,Tile8)&ne(Tile1,Tile7)&ne(P7,P8)&
```

```
        ne(P7,P5)&ne(Tile8,Tile7)&ne(P8,P5)

macrop: [move(Tile7,P8,P5),move(Tile8,P7,P8),move(Tile1,P4,P7)]

precon: at(Tile7,P8)&at(blank,P5)&at(Tile8,P7)&at(Tile1,P4)

padd:   at(Tile1,P7)

add:    at(blank,P4)&at(Tile8,P8)&at(Tile7,P5)

del:    nil   )
```

When FM creates a macro for the FOR or MEA strategies, it need not construct a delete set, since when an operator is applied, the primitive sequence in the 'macrop' slot is used. However, for the Non-linear planner NLP, a delete set WP(i,n).d is built using analogous regression equations to 2(5):

```
WP(i,n).d = Pd(n-i-1) = (Pd(n-i) - O(i).as ) U O(i).d  and
            Pd(1) = O(1).d
```

This is because NLP reasons about the temporal ordering of operators using all their components, and so needs a declarative version of the delete set. Macros thus defined can be used in two ways in future search:

(a) as chunks, in the sense that if the current goal-state-environment combination matches the macro's padd-precon-check components respectively, then the primitive sequence will be applied (or in the case of the goal directed searches, added to the current partial solution).

(b) just like a primitive operator, in any of the search paradigms.

Case (a) suits the FOR strategy and the 8-puzzle problem, but is generally a less flexible method of knowledge acquisition than chunking, for two reasons: firstly, chunking amasses preconditions for particular operators, which can be integrated (as will be seen in section 3.2), whereas the closed macro is too rigid; secondly, chunks may be generalised further and used in situations that do not demand the rest of the operator sequence to be added.

In case (b), macros change long searches with low branching factors, to shorter bushy searches, and if created and used indiscriminately soon cause a search explosion. Minton in [Minton 85] sensibly suggests that their formation must be selective; his MORRIS system stores successful operator sequences and creates macros from the most often occurring subsequences (called script macros). Although we presume MORRIS's macro construction was similar to ours, it is not quite clear how the purpose predicates (FM's 'padd' set) would be derived for these sequences.

Another reason that a macro would be built in the MORRIS system is

to accomplish some conjunction of goals that proved difficult to solve. This is a similar idea to that of Iba's system [Iba 85]. His simple precondition for macro creation was that the primitive operator sequence left most of the state invariant but changed some crucial aspect. An example of this would be in the creation of macros for the 8-Puzzle (as investigated in [Korf 85]). Good candidate operator sequences would be those that moved a tile from one square to another, leaving others that were already in 'goal' positions in place. This use of macros, although quite efficient for finding a solution to tasks with interacting goals, is prone to find solutions many times longer than an optimal solution (as found in Korf's Rubik's Cube example in [Korf 85]).

The problems mentioned above were encountered when using macro creation in FM on the application domains listed in appendix A: as a result of them the emphasis in the rest of this thesis will be placed on the more successful chunking techniques.

3. SEARCH CONTROL ACQUISITION IN LINEAR GOAL DIRECTED PLANNING

3.1 C-chunk Creation and Use

3.11 Introduction

The demise of the general problem solver in the 70's was partly due to the fact that weak methods alone were not sufficient to maintain an acceptably low level of search. There were attempts to combat the search problem using pre-processing techniques on the initial domain definition, e.g. the ABSTRIPS system [Sacerdoti 74] which generated predicate abstraction levels; or the REFLECT system [Dawson and Siklossy 77]: this looked for inconsistent goal predicate pairings, then during an exhaustive goal directed expansion, goals nodes containing such a pair would be discarded. These systems seemed to perform well but lacked focus: on the other hand there are convincing arguments for experience-learning systems in such works as [Carbonell 83] and [Van der Velde 86]. Carbonell describes a computational model which directs the experience of problem solving into a learning mechanism. This mechanism compiles and stores a problem's solution in a form suitable for re-use by analogy with new problems. Van de Velde neatly sums up his arguments for experience learning in 'second generation expert systems' ([Van de Velde 86] page 13) "the translation from deep to shallow knowledge is only worthwhile if there are far less relevant problems that possible ones" - the relevant ones will be found only by experience.

The intended function of the chunk described in chapter 2 was that it should form a heuristic precondition for a generalised goal and operator by matching on the 'current state' in the search space. In contrast linear goal directed searches are through spaces of 'goal nodes' of various types, each node consisting of (at least) such information as an unachieved goal set, an initial state and a sequence of operators; inevitably this does not allow the straightforward matching of the simple chunk, since search is through a space of goal nodes.

Hence the reason for my development of an experience-based, general method for learning heuristics to cut down goal directed search, one that would find operational reasons why one operator/ operator instantiation is to be preferred over another, and could do so for any domain defined by the task specification of chapter 1 (the term 'operational' is used here with a similar meaning to that used in the definition of E.B.G. in [Mitchell et al 86] p.51, i.e. an operational description is one which can be readily matched with some part of the working memory of a problem solver). My work developed through two stages: I devised b-chunk creation as a method of extending the use of equation 2(6) of chapter 2 to goal directed searches [McCluskey 87b], [McCluskey 87d]; and recently evolved this into a more efficient version, incorporating incremental rule repair. I shall call the latter version the 'c-

chunk' simply to distinguish it from the others.

In a typical goal directed search, a goal node is expanded, creating new goal nodes, by collecting all operator instantiations that solve one or more of the predicates in the set of unachieved goals (i.e. contain the predicate in their add-set). As stated in [Minton and Carbonell 87], there are four distinct choice points in this type of search:

1 -- Which node N to expand next?
2 -- Which goal predicate P in N to achieve?
3 -- Which operator O that achieves P, to add to N?
4 -- Which set of operator bindings should be applied to O?

Acquiring strong heuristics for Step 2 was originally addressed by the ABSTRIPS system mentioned above, where predicates acquired abstraction levels, and during search the next goal predicate was chosen according to which had the highest abstraction value. Creating 'goal structure' in this way may deal effectively with choice of goal, but not how to achieve it.

Thus I decided to concentrate on the crucial choices 3 & 4, where search control is badly needed, to cut down the number of new nodes created thus eventually eliminating choice 1. In 'concept learning' terms, I address the problem of learning an operational concept (one that matches on the initial task specification) of which is the best operator and instantiation for that operator, with respect to achieving a particular goal.

3.12 Example Objectives

Roughly, the intended purpose of our chunking mechanism is to capture the most general conditions under which a certain operator instance should be used to achieve a certain goal. This I will define as the target concept for the operator-goal combinations. I will give some simple examples of choice in goal directed search, and guess, by examining the domain definitions from appendix A, some operational reasons why a particular choice is the best. In this way I will attempt to hand craft a rule for particular goal predicates; later we shall see how the c-chunk mechanism automates this through experience, for domains which has been stated within the FM framework.

For example, consider our robot world domain defined in appendix A. It may be required to learn under what most general conditions a robot should choose a particular door D to push a box B through into rooom R, thus achieving 'in_room(B,R)'. Consider figure 3/1:

Here the goal is simply 'in_room(box1,room2)'. Of the three choices of operator instantiation, (shown by the arrows), 'pushthrudoor(box1,door25,room2)' is obviously the best, an operational reason we might venture, as being:

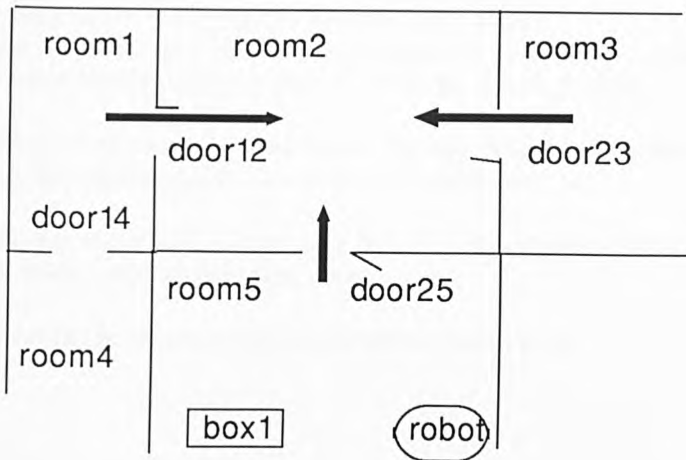

figure 3/1: A robot world fragment

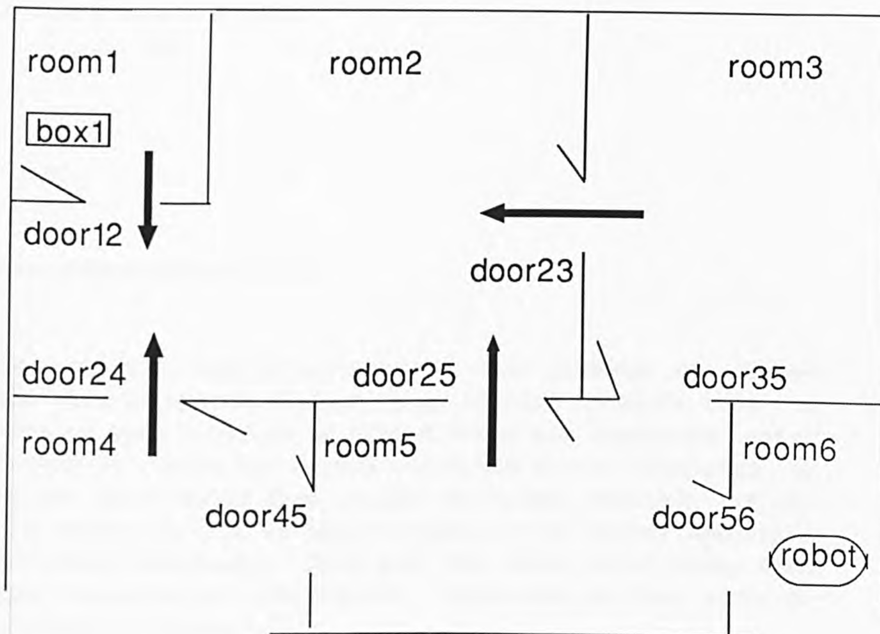

figure 3/2: Another robot world fragment

```
'in_room(box1,room5)&
 connect(room2,room5,door25)&
 fits_thru(box1,door25)'
```

On further reflection we might come up with a description of the target concept as:

D is the best door to push B through into R if
D can be reached, and is the nearest to B, by a path which includes only doors (including D) through which B fits.

For simplicity we define 'nearness' by equating each operator with a unit cost. An operational description might be:

```
to achieve in_room(B,R),  for any B,  D,  R, where type_of(B,box),
type_of(D,door) and type_of(R,room):

{     in_room(B,R1)&connect(R,R1,D)&fits_thru(B,D)

      V

      not( there exists D':
           in_room(B,R1)&connect(R,R1,D')&fits_thru(B,D') )&
      in_room(B,R1)&connect(R,R2,D)&fits_thru(B,D)&
      connect(R2,R1,D1)&fits_thru(B,D1)

      V

      not( there exists D', D":
            ...... etc

      V

      .... etc              }

=>    choose pushthrudoor(B,D,R)
```

Even this series is not quite correct - the problem of closed doors means that it may be cheaper to go through an extra room to avoid having to open a series of closed doors and therefore enter by a different D! Where the domain contained a room connected to another by two doors would also change the target concept: if one is open, it should be able to discriminate in its favour against a closed door where necessary. Thus even the robot world shows that these target concepts are non-trivial. Examining another scenario may help: consider figure 3/2.

Here the main goal is 'in_room(box1,room3)', whereas the subgoal to be solved is 'in_room(robot,room2)'.

Of the four choices of operator instantiation, (shown by the arrows), 'gothrudoor(door25,room2)' is obviously the best; an

operational approximation to the reason is:

```
'connect(room2,room5,door25)&
 connect(room6,room5,door56)&
 in_room(robot,room6)&
 not( there exist Z: connect(room2,room6,Z) )'
```

and the reader is left to draft a general disjunctive rule, similar to the 'pushthrudoor' example.

3.13 Basic C-chunk Creation

Consider operator O(i) (1<i=<N) taken from a minimal, successful operator sequence of size N, which has solved some task given to FM. O(i) must have been the correct operator choice from some set of operator instantiations OI = {O(j): j in 1..n & n>=1}, which were proposed at a node in goal directed search, to achieve one of the node's goal predicates, G. The heuristic creation processes which I shall describe search for the characteristics in the task specification which make O(i) the best choice from OI to achieve G, and so discriminate against the rest of OI.

This is in contrast to using pure EBL on the problem solving trace and searching for characteristics which match the goal node that explain why O(i) was the best choice (e.g. see chapter 4 or [Minton et al 87]). This latter approach is too easily bogged down by the complexity and peculiar representation of nodes, and is invariably over-specific; on the contrary I believe that learning algorithms should not so much concentrate on correctness (as also argued in [Van der Velde 88]) but on forming practical and usable rules, although they may need refining in the light of further experience.

The build up of useful chunks may be helped by the fact that in MEA, a solution sequence to any solved sub-task can be a candidate for chunk creation, and may cause the creation of up to N-1 chunks (recall from chapter 1 that each node in the goal directed search is itself a sub-task). For instance a situation can arise when a sub-goal solution which is not on the main solution path gives rise to chunk creation. Consider the and/or tree, representing part of a solution trace, in figure 3/3. Here a subgoal of node1 may be solved through nodes 4 and 5, and chunk(s) may be created, even though the solution path for the main task may consequently turn out to be along a different path.

At some nodes (e.g. node5), OI may have been a singleton, in which case a chunk is not made. This may be because the operators have been well refined by the user (e.g. 'gotodoor', 'close' of the robot world in appendix A only ever cause the generation of only one child node during goal directed search) or that chunks have already cut down the search. The simple rule is that where

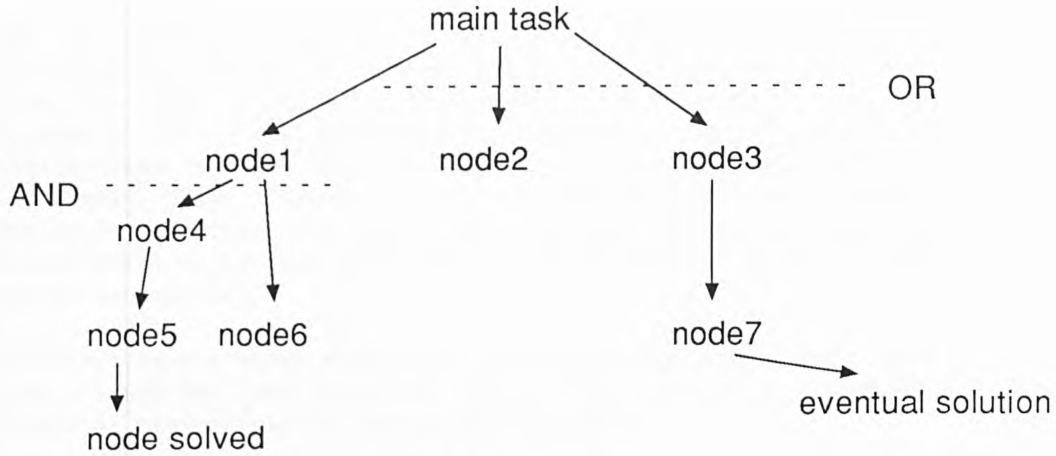

figure 3/3: and/or tree

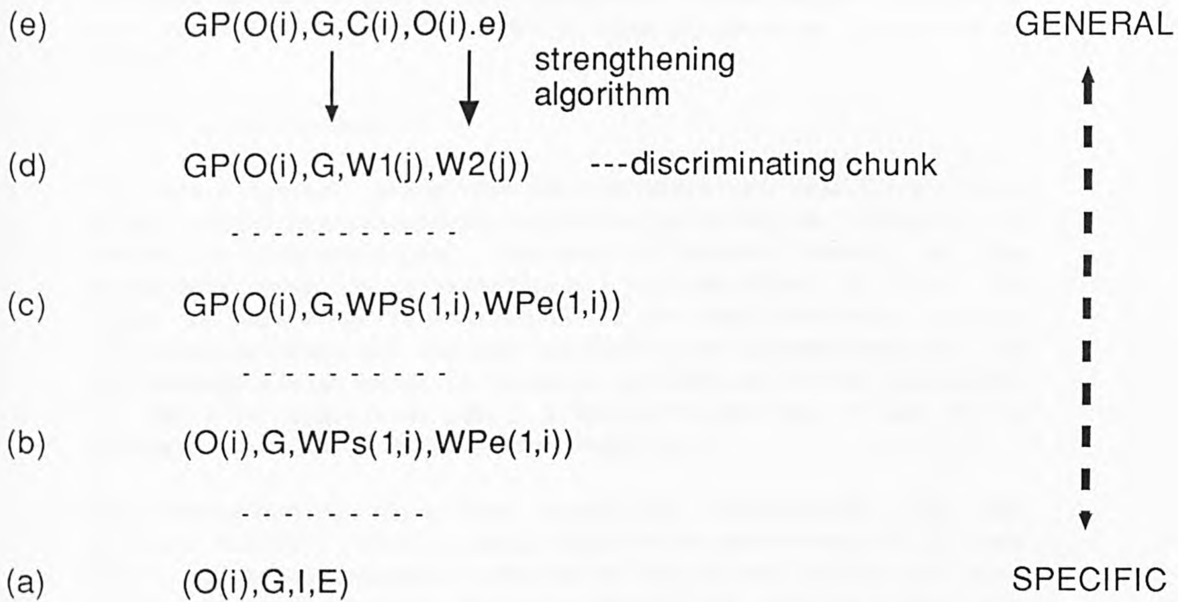

figure 3/4: generalisation hierarchy

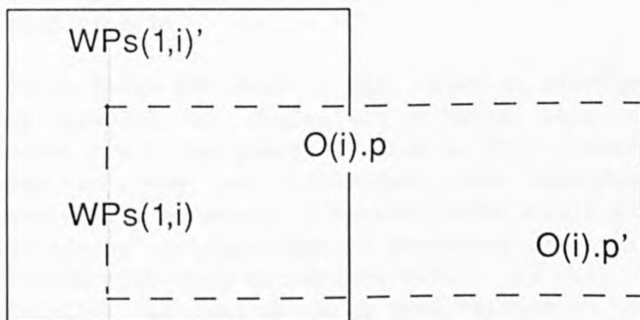

figure 3/5: The residual predicate sets

there was no branching in the solution trace, then no chunks are created. This causes the rate of chunk creation to be reduced, as they are effectively used.

The discriminant components which are derived by chunking are conditions on the initial state I and and environment E, and stem from the goal regression and generalisation processes used in chapter 2. For the b-chunk this discriminant is derived from the similarities between WPs(1,n) and WPs(i,n) (see 2(7) and consult [McCluskey 87d], figures 2 and 3), whereas for the c-chunk, WPs(1,i) and O(i).p are used. The final goal predicate used to derive WPs(1,i) is that predicate for which O(i) was added to the search to achieve.

Similarities are enhanced using a 'strengthening algorithm'; two such algorithms are detailed below. The core of a c-chunk's heuristic precondition on states is defined as:

C(i) = {P in WPs(1,i) : O(i).p&S.r => P} ..3(1)

To produce C(i), the present algorithm uses the rules in S.r to increase the set O(i).p, then intersects these ground predicates with WPs(1,i). A possible chunk is then proposed as (compare with 2(7)):

GP(O(i).n,G,C(i),O(i).e)

The core, however, along with the environmental conditions O(i).e, gives only a generalised approximation as to why an operator is needed to achieve a goal, and success depends heavily on the particular domain's representation, and the rules in S.r. The chunk at this stage is reminiscent of the weak heuristic 'choose the operator which has the most preconditions already achieved by the initial state' which is actually included as a weak heuristic in MEA. To improve on this, I devised algorithms A and B to strengthen the chunk's latter two components.

The strengthening algorithms specialise expressions C(i) and O(i).e (= W1(0), W2(0)) until they reach expressions W1(j) and W2(j) which discriminate in favour of the correct choice if used in exactly the same task. Following figure 3/4, The instance to be generalised is represented at point (a). At point (b) it has been generalised to include only those predicates that were necessary and sufficient for the operator sequence to be applied, up to and including O(i).

Systems based strictly on EBG, such as Prodigy, would presumably only produce an equivalent of point (c), when learning from success i.e. the generalisation of (b)'s constants to variables, where it does not invalidate the sequence O(1) .. O(i)'s precondition's proof ([Minton & Carbonell 87] p.231 show the acquisition of this type of heuristic for a blocks world example which we reproduce in section 5.2). In (c), however, there is generally information which just relates to the satisfaction of

intermediate operator preconditions, and is irrelevant to the final operator O(i). In fact, the last two components of (c) are identical to the state and environmental preconditions of a closed macro (see section 2.5 or [McCluskey 87c]).

My chunking techniques start with the expression at point (e) and specialise it, using a strengthening algorithm, until it discriminates against choices in the original solution at point (d). Although the algorithms are essentially general to specific, in some cases C(i) may already be too specific - for instance when it produces a conjunctive component of a disjunctive target concept. In fact the discriminating chunk at point (d) may still only be an approximation to the target concept - methods of chunk repair and refinement are explained in 3.2. An outline of my two 'general to specific' algorithms are given below, and their implementations are in appendix C.

Algorithm A

First let us define the following two 'residual' sets (see figure 3/5):

```
WPs(1,i)'  = WPs(1,i) - O(i).p
O(i).p'  = O(i).p - WPs(1,i)
```

This algorithm adds predicates from WPs(1,i)' to C(i), and adds environment relations to the last component of the chunk that connect O(i).p' and WPs(1,i)', finding relevant connections between them using association chains.

Thus the idea is to look for connections between features in WPs(1,i) and O(i).p that do not appear in the initial chunk. Following Vere in [Vere 77] (see also figure 3 in appendix D.4), an association chain between two predicates X and Y, with respect to a set of background facts F, is a sequence of predicates C1,C2, ... Cm, such that

```
     C1 = X, Cn = Y, Ci in F, 1 < i < m
```

and for all k, 1 < k < n, Ck-1,Ck,Ck+1 is such that:

```
     there exists terms x and y in Ck such that
          x is a term in Ck-1 &
          y is a term in Ck+1 &
          not(x = y).                                    3(2)
```

An example of a chain, with F = E.f in appendix A.1, is C1,C2,C3, where

```
               C1 = in_room(box1,room1)

               C2 = connects(room1,room2,door12)
```

C3 = at_door(box1,door23,room2)

and where x = room1 and y = room2.

Algorithm A is shown in figure 3/6: note the use of only a subset of the environment E for the association chain's background facts F - in fact those parts used in operator application proof before and including O(i).

The algorithm strengthens the initial chunk along two dimensions: the inner loop starts with F assigned to only O(i).e, and gradually increments this to the full sequence {O(i).e U .. U O(1).e}, if necessary. This reflects the fact that discriminating conditions will naturally be found 'near' i, and ensures that it will not be over-specialised. The outer loop lengthens the association chain allowed.

Algorithm B

As previously mentioned in [McCluskey 87b], the complexity problems involved in building association chains limited their size. To combat this, algorithm B was developed as A's successor. Specifically, B is computationally less expensive because extra features which strengthen the core (defined by 3(1)) are obtained directly from operator applications' preconditions, rather than using relational chains. Inside the single loop of B it can be seen that the most complex operations are intersection and term listing. Growing association chains, on the other hand, is in general an exponential problem, and only viable for small chains.

If we first define

TS(wff) = {t : t is a term appearing in wff},
C(i,k) = {P in WPs(1,i) : O(k).p&E => P}, 1 < k =< i,

then the basic idea is to specialise C(i) (= C(i,i)) towards WPs(1,i) by adding predicates of WPs(1,i) that appear in the preconditions of operators 'behind' O(i), i.e. adding C(i,i-1), then C(i,i-2) etc, until the chunk discriminates. Any predicates that are added to C(i) are supported by environmental predicates from operator preconditions that contain common terms (see figure 3/7).

The resultant C-chunk from either algorithm is defined as

GP(O,G,W1(j),W2(j)) ..3(3)

where GP is the generalisation operator defined in section 2.4.

To recap: a c-chunk is a 4-slot logical term; it is formed from a successful problem solving trace of a goal reduction search, and

```
procedure strengthen_A;

/* initialise chunk components */
W1(0):= C(i);
W2(0):= O(i).e;
j    := 0;

while  (GP(O,G,W1(j),W2(j)) is not a discriminating chunk)
        &  (j < complexity bound)   do:
        j := j+1;
        F := O(i).e;
        k := i;
        while    (GP(O,G,W1(j),W2(j)) is not a discriminating chunk)
                 &  (k > 1)   do:
                 k      := k-1
                 F      := F U O(k).e
                 (X, Y) := {x, y :  x is a predicate in WPs(1,i)',
                           and x is related to some predicate z in O(i).p' by an
                           association chain x,y,z of length j+2 in F };
                 W1(j)  := W1(j-1)&X;
                 W2(j)  := W2(j-1)&Y
        end while;
end while;

end strengthen_A.
```

figure 3/6: strengthening algorithm A

```
procedure strengthen_B;

/* initialise chunk components */
W1(0):= C(i);
W2(0):= {P :   P is in O(i).e  &
                ( TS(P) intersect TS(W1(0))  <> empty_set)   };
j    := 0;

while (GP(O,G,W1(j),W2(j)) is not a discriminating chunk & j<i) do
      j := j+1;
      X := C(i,i-j);
      Y := {P  :  P is in O(i-j).e  &
                ( (TS(P) intersect TS(X))  <> empty_set)    };
      W1(j)  := W1(j-1)&X;
      W2(j)  := W2(j-1)&Y
end while

end strengthen_B.
```

figure 3/7: strengthening algorithm B

used as a heuristic to guide future searches by matching its slots with components of a goal node.
Specifically, it is formed from the context of an operator application O which achieved some sub-goal G at the end of a solution sub-sequence of operators. The chunk's third and fourth components are extracted from the state and environmental parts of the weakest precondition of the sub-sequence; predicates extracted are ones which appear in operator preconditions towards the end of the sub-sequence, and which seem to be the reason why operator O was used to achieve goal G.

Chunk use will now be defined in the following section.

3.14 The Use of c-chunks

The meaning of 'a discriminating chunk' in 3/7 is made precise by our definition of the use of chunks. Consider the set of operator instantiations OI = {O(j): j in 1..n & n > 1 } which have been proposed in the goal directed expansion of some node with initial state S, to solve one of the node's goal predicates, G; also assume that none of the O(j)'s have preconditions satisfied by S (if an operator were applicable, then it would be applied and the advanced state would be stored in the node's trace, as explained in chapter 1). A chunk is defined as discriminating if it favours one and only one member of OI. A chunk (O',G',W1,W2) that matches the operator O(i) and the node's three components, in the following way, favours that instantiation:

```
there exists some variable binding set t:
(O(i) = [O']t) &
(G = [G']t) &
(S => [W1]t) &
(E => [W2]t)                                                ..3(4)
```

As defined, if O(i) is the only instantiation to be favoured, then the chunk is discriminating and O(i) will be the one chosen to continue goal directed search. Of course the same chunk, or others, may advise more one operator instantiation, in which case all favoured paths will be followed (the various combinations are elaborated later, in section 3.23).

3.15 C-chunk examples

We will use the scenario of figure 3/2 to clarify the chunk idea. If the initial state is as shown, and the goal is in_room(box1,room3), then using E and OS defined in appendix A.1, FM will output the following operator sequence as the solution:

{O(1), O(2), ... O(11)} =

{gotodoor(door56,room6), gothrudoor(door56,room5),

```
 gotodoor(door25,room5), gothrudoor(door25,room2),
 gotodoor(door12,room2), gothrudoor(door12,room1),
 goto(box1), pushtodoor(box1,door12),
 pushthrudoor(box1,door12,room2), pushtodoor(box1,door23,room2),
 pushthrudoor(door23,box1,room3) }.
```

Chunk formation will occur whenever there was a choice of operator/operator instantiation in the search generated for this solution. If there were no existing chunks, then in fact there would have been four choice points: we will show chunk creation for two of these - O(9) and O(11). For more examples consult section 3.3.

Chunk creation first uses equations 2(7) to construct the weakest preconditions:

```
WPs(1,9) =  in_room(box1,room1)&in_room(robot,room5)&
            open(door12)&open(door25)&open(door56),


WPe(1,9) =  connect(room1,room2,door12)&fits_thru(box1,door12)&
            connect(room2,room5,door25)&
            connect(room5,room6,door25)&type_of(box1,box)

WPs(1,11) = in_room(box1,room1)&in_room(robot,room5)&
            open(door23)&open(door12)&open(door25)&open(door56),

WPe(1,11) = connect(room1,room2,door12)&fits_thru(box1,door12)&
            connect(room2,room5,door25)&fits_thru(box1,door23)&
            connect(room2,room3,door23)&type_of(box1,box)&
            connect(room5,room6,door25)

Since O(9).p =
     at_door(robot,door12,room1)&next_to(robot,box1)
     &open(door12)&&in_room(box1,room1),
```

then from 3(1) and the robot-world environment E in appendix A,

```
C(9) = in_room(box1,room1)&open(door12)
```

and a discriminating c-chunk would be formed:

```
GP(pushthrudoor(box1,door12,room2),
              in_room(box1,room3),W1(0),W2(0)) =

ch(ch9,    pushthrudoor(B,D,R),
           in_room(B,R),
           open(D)&in_room(B,R1),
           connect(R1,R2,D)&type_of(B1,box)&fits_thru(B1,D))
```

This discriminates against the other candidates:

```
{ pushthrudoor(box1,door23,room2), pushthrudoor(box1,door25,room2),
  pushthrudoor(box1,door24,room2) }
```

This example produced a discriminating heuristic without the need for strengthening; if we examine C(11), however, this will not immediately lead to a good heuristic; without strengthening, this chunk will look like:

```
ch(ch11', pushthrudoor(B1,D23,R3),
          in_room(B1,R3),
          open(D23),
          connect(R2,R3,D23)&type_of(B1,box) )
```

which does not discriminate between the two operator instances

{pushthrudoor(box1,door34,room3), pushthrudoor(box1,door23,room3),

The strengthening algorithm B for example, would terminate for j=2 and output:

```
W1(2) = open(door23)&in_room(box1,room1)&open(door12)

W2(2) = connect(room1,room2,door12)&fits_thru(box1,door12)&
        fits_thru(box1,door23)&connect(room2,room3,door23)&
        type_of(box1,box).
```

which produce the discriminating chunk:

```
ch( ch11,      pushthrudoor(B1,D23,R3),
               in_room(B1,R3),
               open(D23)&in_room(B1,R1)&open(D12)
               connect(R1,R2,D12)&fits_thru(B1,D12)&
                    fits_thru(B1,D23)&connect(R2,R3,D23)&
                    ne(D12,D23)&ne(R1,R3)&type_of(B1,box))
```

Sometimes discriminating features are found which are only supportive or co-incidental to the target concept. Although occurrences are minimalised (since the algorithm is searching through predicates which are in the weakest precondition) section 3.2 shows how FM can refine imperfect rules.

We shall now give a simple example of the use of ch11, in future problem solving. Consider figure 3/1, with the goal of 'in_room(box1,room1)'. Then two operator instances would be immediately generated during search:

{pushthrudoor(box1,door14,room1), pushthrudoor(box1,door12,room1)}

ch11 discriminates between these two and favours the correct choice pushthrudoor(box1,door12,room1) because ch11 instantiates to:

```
ch(ch11,pushthrudoor(box1,door23,room1),
        in_room(box1,room1),
        open(door23)&in_room(box1,R1)&open(D12)
        connect(R1,R2,D12)&fits_thru(box1,D12)&
```

```
fits_thru(box1,door23)&connect(R2,room1,door23)&
ne(D12,door23)&ne(R1,room1)&type_of(box1,box))
```

and a consistent binding exists for the state component (R1 = room5, D12 = door25) and so for the further instantiated environment component (R2 = room2).

3.16 Evolution of the Chunking Technique

Before developing c-chunks any further, I will show how they have evolved from my previous work on b-chunks. I originally designed b-chunks [McCluskey 87a] as being analogous to chunks which aid forward search - they both rely on the same kernal, the 'WPs(i,n)' of section 2; they also both take into account the final goal. An advantage of this was thought to be that they could be used to propose operators to generate a skeleton solution to a problem before search began, similar to the ideas of Carbonell in [Carbonell 83]. This would change their test role in a 'generate and test' search to one of operator generation. As probably happened in Carbonell's line of research, I found that the combinatorics of search for the rest of the solution was as least as bad as starting from scratch - this also proved to be the case when using chunks to generate an advanced partial plan within a general non-linear planner (see chapter 4).

I will not describe the b-chunk in depth here, since the two forms are very similar, and the material is covered in [McCluskey 87b], [McCluskey 87d] which support this thesis. I will, however, list the important differences, using the terminology already developed for creating a chunk with respect to an operator O(i):

3.161: The 'goal slot' of the b-chunk was occupied by the main goal predicate(s) of the goal node's solution sequence (a goal node tree was shown in figure 3/3), not necessarily the one O(i) was invoked to achieve.

3.162: WPs(1,n) and WPe(1,n) were used instead of WPs(1,i) and WPe(1,i) respectively. This would include extra information from the second part of the solution sequence, after the action of O(i).

3.163: O(i).p was expanded to WPs(i,n).

The b-chunk was then created using the same processes as explained in section 3.13. For example the core of a b-chunk was:

C(i) = {P in WPs(1,n) : (WPs(i,n) & E) => P}

Referring to 3(4), a b-chunk favours an operator instantiation if:

there exists some binding set t:
(O = [O']t) &

```
(S => [W1]t) &
(E => [W2]t) &
(([G']t => G) V
 (there exists G1: ancestor_of(G1,G) & [G']t => [G1]) )
```

This is not a straightforward match (unlike 3(4)) because the b-chunk stores the 'final goal G', and uses it as an extra constraint. The final disjunction includes an 'ancestor' relation, which allows the chunk's goal to unify with any ancestor goal encountered in the search - making the chunk more generally applicable. Also there was the possibility that the final goal was a conjunction of predicates, hence the need for the implication in the last conjunction.

Early tests on several different 'toy' worlds, lead to the following conclusions about the b-chunk:

3.164: Although in some circumstances the correct choice of an operator/operator instantiation does depend on the main goal of the task, as well as on the rest of the specification, using the goal as a further constraint causes the target concept generalisation space to be too complex and disjoint - in fact it tends to multiply the chunks needed. A circumstance where the main goal may be important is when it consists of two interfering subgoals; this may be overcome, however, with separate heuristics to deal with goal ordering such as abstraction.

3.165: The constraints that are gathered from the successful operator sequence to form a b-chunk, for some operator O, consist of predicates used in application proofs both before and after O in the sequence. Those that are from after O tend to be irrelevant and make the chunk overspecific.

3.166: B-chunks don't record which goal predicate an operator was added to achieve in the original search. This causes no problem if a user is constrained to structure the operators so that each has only one non-side effect (e.g. only having one predicate on the 'padd' list of FM operators), but seems unreasonable in general, since the spirit of this work is for the system itself to make such operational decisions.

Finally, we recap the make-up of b-chunks:

Like c-chunks, they are 4-slot logical terms formed from a successful problem solving trace of a goal reduction search; they are used as heuristics to guide future searches by matching their slots with components of a goal node.

They are heuristics formed from the context of an operator application (say O) which occurred in some solution sequence of operators (say in solving a main goal G). As with c-chunks, their third and fourth components are extracted from this context, but in contrast they are made from similarities between:

--the generalised weakest precondition of the WHOLE sequence of operators that solved G; and

--the weakest precondition of the sub-sequence of operators after (and including) O.

3.2 Chunk Refinement

3.21 Introduction: the Form of Heuristic Rules

Refinement and acquisition control of chunks into a rule set will now be discussed, as any expanding rule base needs at least some form of redundancy control and a truth maintenance component.

It should be obvious from the examples above that chunking does not immediately acquire a perfect operational version of the target concept: recall the example in 3.15 that created ch11. If in the original initial state (in figure 3/2), door34 had been closed, then the weak chunk ch11' would have discriminated between the two operator instances. This is not too serious, however, since (following 3.15) if the chunk ch11' is applied to operators that solve goal 'in_room(box1,room1)' in figure 3/1, then the chunk's lack of discrimination in this case would send it to be specialised as described in 3.23 below.

I therefore view chunk refinement under the umbrella of incremental concept learning (it should follow chunk creation as suggested in [McCluskey 87b]) and feel that a planner should improve its performance both analytically and experimentally. This is because analytic goal regression only generates a sufficient condition for goal achievement, and is very often over-specific. To this end, the integration of concept learning and analytic goal regression has already been encouraged in both [Boswell 86] and [Porter and Kibler 85]. Let us first review the structure of the stored c-chunks. In the earlier part of this chapter we modelled a chunk (O,G,W,W') as a kind of decision rule of the form:

IF G & W & W' match their corresponding task components

then add O under the bindings of these matches ..3(5)

This is too primitive, because the heuristic store must be in a form suitable for change and repair as more successful sequences are analysed. In general, several chunks may be made for the same operator-goal pair, and would together form an evolving, disjunctive rule. Rules may also have to be repaired, in cases where they are too general. To accommodate this we draw on the idea of evolutionary, automated knowledge refinement as proposed in [Michalski 85]. He defines censored production rules of the form:

```
if        <premise>
then      <decision>
unless    <censor>.
```

Building on this idea of the censor, our evolving knowledge base is defined as a collection of heuristic rules, each of the form:

```
" ( if G then
    (if (W1,W1') unless <list of exceptions>  V
     if (W2,W2') unless <list of exceptions>  V
      ......                                  V
     if (Wn,Wn') unless <list of exceptions>  )
             then choose O     ) "
                                                                    ..3(6)
```

where it is implicit that G, Wi, Wi' match a node's goal, initial state and environment, respectively.

This representation allows the problem of 'overlapping rules' to be solved incrementally. For instance, following [Boswell 86,pp52-53], given two learned heuristic rules:

```
If P1 then apply operator O1;
If P2 then apply operator O2;
```

it is argued that if P1 and P2 are not disjoint and O1 leads to a shorter solution, then the rule set should be changed to:

```
If P1 then apply operator O1;
If P2&not(P1) then apply operator O2;
```

I will use the exception slot of 3(6) to deal with overlapping chunks in a similar way (see 3.23(c)). Given this more general form, we redefine the use of an individual chunk: it favours an operator instantiation if and only if it satisfies 3(4) AND none of the chunks in its exception list favour a different operator instantiation.

Incremental chunk creation and use will drive the repair or modification of heuristic rules of this form, and will help them to converge towards their target concepts. Basically, heuristic rules can evolve in three ways: they can be generalised by the addition of new chunks, specialised by chunk strengthening, or specialised by exception addition. Using the rule representation just discussed, this will be dealt with in section 3.23, while in the next section we will describe rule optimisation.

3.22 Rule Optimisation

The purpose of optimisation is to make heuristic rules more efficient while preserving their meaning. If R is a rule in 3(6) format then define:

```
Theory(R) = { S : S is  a state description and (S,E.f) is
                  matched by some pair (Wi,Wi') in R,
                  and none of its exceptions }
```

An optimisation operation T is thus axiomatised by the following invariant, for any rule R:

```
Theory(R) = Theory( T applied to R)
```

Some operations are given below, together with examples of their use:

T1: Using the rules specified in environment E.r, remove redundant predicates in each chunk.

For example consider the chunk ch1 which is taken from our results of 3.3:

```
ch(ch1,   gothrudoor(x(1),x(2)),   in_room(robot,x(2)),
          open(x(1))&in_room(robot,x(3)),
          ne(x(3),x(2))&ne(x(5),x(1))&ne(x(2),x(4))&ne(x(3),x(4))&
          connect(x(3),x(4),x(5))&type_of(x(4),room)&
          connect(x(4),x(2),x(1))&type_of(x(3),room)&
          type_of(x(2),room)&type_of(x(5),door)&
          connect(x(4),x(3),x(5))&connect(x(2),x(4),x(1))   )
```

the rules from the robot world in appendix A shorten this to:

```
ch(ch1,   gothrudoor(x(1),x(2)),   in_room(robot,x(2)),
          open(x(1))&in_room(robot,x(3)),
          ne(x(3),x(2))&ne(x(5),x(1))
          connect(x(4),x(3),x(5))&connect(x(2),x(4),x(1))   )
```

T2: Re-order the predicates in each chunk for efficiency.

Our chunk becomes (its final form is in section 3.3):

```
ch( ch1, gothrudoor(x(1),x(2)),   in_room(robot,x(2))
         open(x(1))&in_room(robot,x(3)),
         connect(x(3),x(4),x(5))&connect(x(4),x(2),x(1))&
         ne(x(3),x(2))&ne(x(5),x(1))   )
```

T3: Re-order the chunks within a rule for efficiency.

FM keeps a record of the heuristic worth of each chunk within a heuristic rule, and uses this to bias its use, i.e. the most used chunks, being the most probable to give a match, are tried first.

T4: Use background rules specified in environment E to merge chunks.

For larger implementations it would also be advisable to cut down the number of chunks using environment/state rules such as:

```
type_of(X,door) --> closed(X) V open(X).
```

Then given two chunks:

```
ch(  ch1, gothrudoor(x(1),x(2)),   in_room(robot,x(2))
          open(x(1))&in_room(robot,x(3)),
          connect(x(3),x(4),x(5))&connect(x(4),x(2),x(1))&
          ne(x(3),x(2))&ne(x(5),x(1))   )
ch(  ch2, gothrudoor(x(1),x(2)),   in_room(robot,x(2))
          closed(x(1)&in_room(robot,x(3)),
          connect(x(3),x(4),x(5))&connect(x(4),x(2),x(1))&
          ne(x(3),x(2))&ne(x(5),x(1))   )
```

these would simply merge to:

```
ch(  ch3, gothrudoor(x(1),x(2)),   in_room(robot,x(2))
          in_room(robot,x(3)),
          connect(x(3),x(4),x(5))&connect(x(4),x(2),x(1))&
          ne(x(3),x(2))&ne(x(5),x(1))   )
```

I define a predicate relation to be sparse if the number of instances of it in E is much smaller that the number of possible instances. For example 'connect' in appendix A is sparse - there are 18 instances of it whereas there are (no. of objects)**3 possible.

It is then obvious that T2 can make a significant amount of difference to the matching cost of rules by re-ordering chunks' predicates so that the sparse relations are matched first. Naturally, relations with the largest arity tend to be also the most sparse; this particular optimisation rule has had the most impact on the experimental results outlined below.

3.23 Rule Repair

Chunks are domain dependent heuristics created by a weak (general) learning method. They can be viewed as generalisations in 'operator x goal x state x environment' space. As the environment is a fixed body of facts, this component acts as a constraint on the state expression's variables. Figure 3/8 represents a simplification of the way some fictional chunks approximate a particular target concept (i.e. where the goal predicate name and operator name have been fixed).

Due mainly to FM's generality and more powerful operators, simple incremental induction methods, epitomised by the version space technique of LEX [Mitchell 83], cannot be used. For example, operator parameters, rather than operators alone, are chosen by chunks.

Related to the more general, powerful representational factor mentioned above is the disjunctive nature of target concepts. In the version space paradigm, an example inside the space allows the concept 'lower bound' to be generalised so as to cover it, or on the other hand, a counter example causes the specialisation of the concept's 'upper bound' (see [Mitchell 83] for more details of the version space method applied to problem solving). If this method were to be used in the scenario of figure 3/8, e3 should generalise ch-2 towards ch-1, but this clearly is not correct because of the disjunctive nature of the target concept.

A more drastic measure is rule subsumption: if ch-2 is created from example e2, after ch-1, then it can be seen from the diagram that ch-1 subsumes ch-2, i.e. ch-1 covers more ground instances than ch-2. In some of those ground instances, however, ch-1 may either offer more than one operator instantiation, or worse the wrong one, hence it is be wise to keep ch-2. On the other hand, deleting ch-1 and keeping ch-2 does not seem profitable, since ch-1 covers more of the target concept.

We will show how rule repair can give a partial answer to this problem, and deal with over-general or over-lapping rules. We have already seen how chunk creation and strengthening processes help to 'home in' on conjunctive parts of the target concept quickly (see figure 3/4), and how a heuristic rule may be disjunctive to match this.

First, consider figure 3/9 which shows the main cases where chunks are created or incremented when applied to search reduction. In the figure, 'node' represents an expanded node which has been encountered during goal directed search but is also on the solution path to some task; and each arc a choice of operator and instantiation that was generated at that node.

In case (a) no chunks have been used, whereas (b) is the desirable case where a chunk has indicated the correct path. In (c), a chunk has indicated the correct path, but other chunks C' and C" have fired for incorrect paths. In (d), the chunk is too general and has fired for more paths than the solution. Finally (e) represents a chunk C' which has incorrectly fired.

FM performs an analysis on the trace of a planning session and deals with these eventualities as follows:

(a) A chunk is made (cf. section 3.13), optimised (cf. section 3.22, T1 and T2), and augments the relevant heuristic rule.

(b) The heuristic worth of C is incremented (cf. section 3.22, T3).

(c) The heuristic worth of C is incremented and C is added to C' and C" 's <list of exceptions> slots (see expression 3(6)).

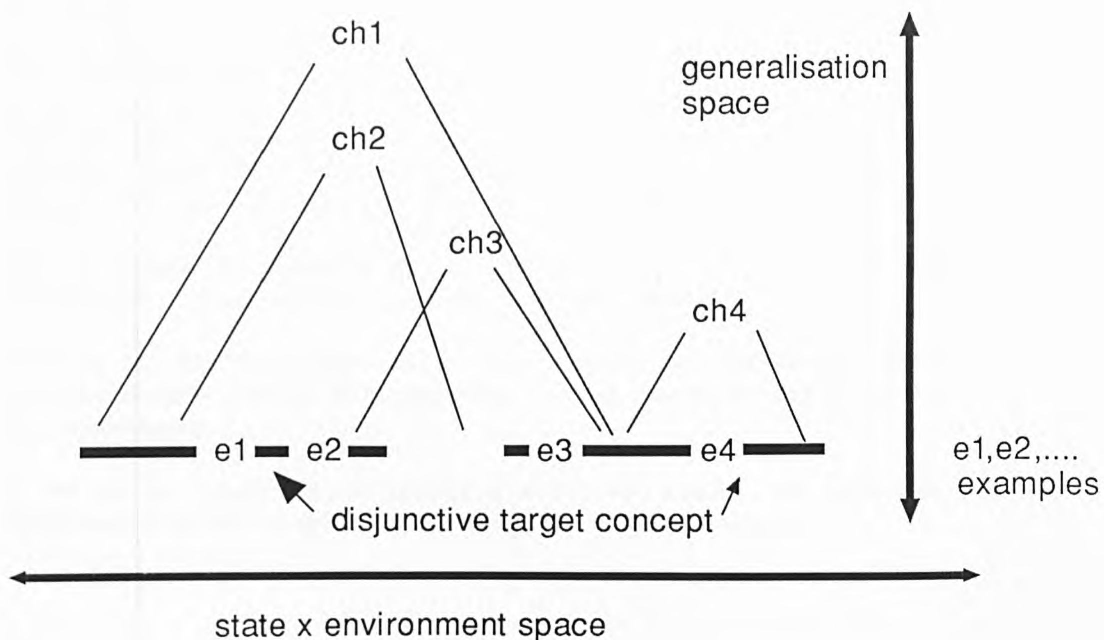

figure 3/8: chunks approximating a goal's target concept

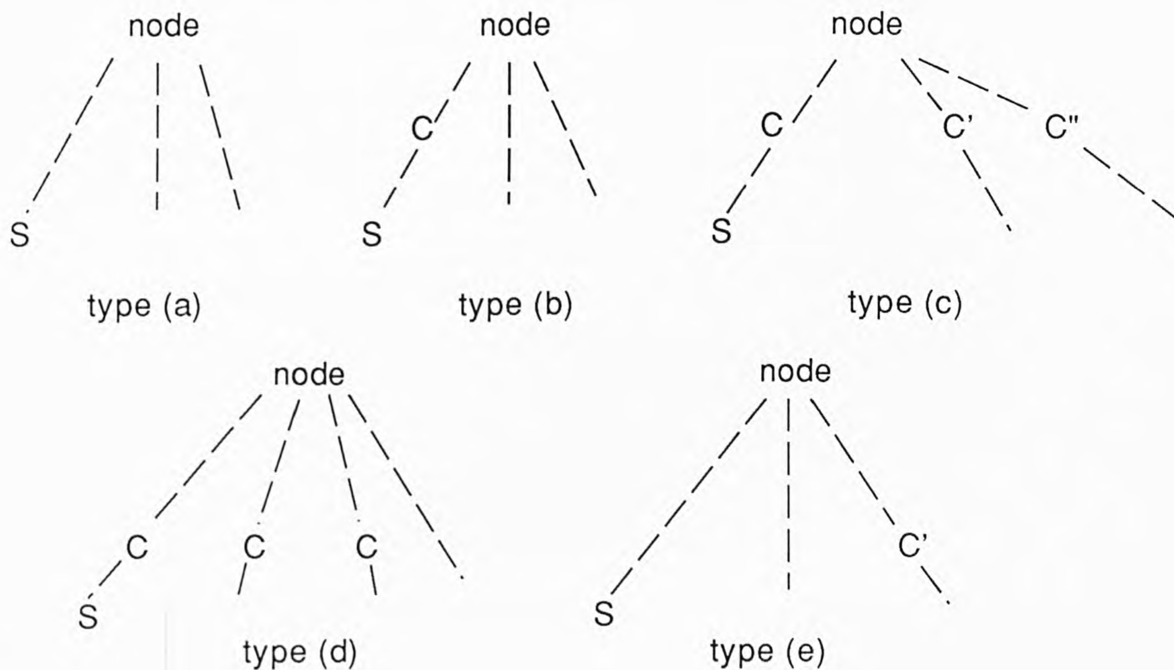

figure 3/9: main types of chunk use (in each case S marks the successful branch)

Occasionally, since the the rules acquired are generally incomplete, circular exceptions may be recorded. In this case FM will generalise both rules so as not to include either exception.

(d) Chunk C is passed to the strengthening algorithm, with the set of operator instantiations (OI in 3.1) restricted to those that the chunk has favoured (the first three branches in the diagram). C is specialised so that it would subsequently discriminate between the three branches (the exception to this is where the branches lead to a solution of the same length - this causes a dilemma for the learner, and is discussed later in 3.321)).

(e) A chunk is made as in (a) with C added to its list of exceptions. This option is only available when:

* FM is in 'learning mode' only - i.e. where the MEA search does not use chunks' advice but registers it and builds heuristic rules up accordingly.

* FM is in 'learning and planning mode' but chunks do not cut branches down but simply favour instances heuristically.

3.3 Experimental Results

3.31 Introduction

Although several other domains have been specified to FM (as listed in appendix A, these include Tower-of-Hanoi, 8-Puzzle, Blocks worlds, Macbeth World) we have chosen two particular ones to show the power of the c-chunk with the MEA strategy (of course these applications are simply 'micro-worlds', that is they make simplifying assumptions when compared with the more complex real world):

1. A robot-room world (figure 3/10), an augmented version of the one in [Sacerdoti 73]. This was chosen for several reasons-

-similar worlds appears several other times in the literature, making it somewhat of a 'bench-mark' (e.g. [Fikes et al 72], [Dawson & Siklossy 77]).

-the target concepts to be learned are not trivial (as we saw in section 3.1).

2. A warehouse world (figure 3/11); this involves tasks such as moving crates from one part of the warehouse to another, which means having to plan a route to a truck, plan the trucks route to the crate, pick up the crate, remove obstructions along the way etc. This was chosen because:

-like the robot world, it has a convenient spatial representation, and is therefore more intelligible.

-it is a more complex world than the robot worlds.

-it does not seem to have been modelled elsewhere in the literature, and is therefore somewhat 'fresh'!

3.32 The Robot World Experiments

Although many variations and representations of plan layout, initial state and environment are possible, I show a particular one which in detailed by the I,E,OS of appendix A.1, and graphically shown in figure 3/10. I have tried variations along the representational and arrangement dimensions of this domain, and obtained similarly encouraging results as presented below, because of the generality of the c-chunking method.

The following criteria were taken into account when selecting the sample tasks:

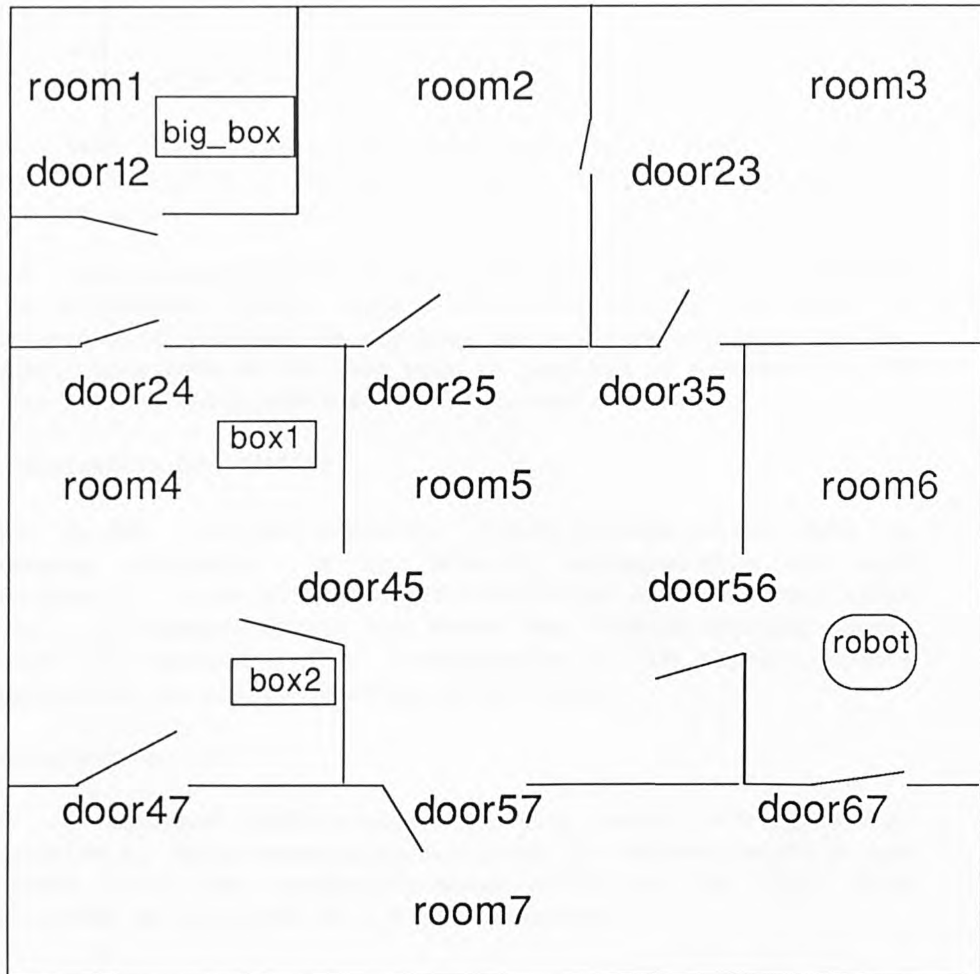

Figure 3/10: robot world

1. put less demanding tasks at the beginning of the session;

2. do not have tasks with complex, interfering multiple goals.

Criterion 1. was included since it seems more 'natural' for a learning system to be presented with less demanding tasks first.

Criterion 2. was included since the c-chunk in its present form is not aimed at advising on goal interactions, and MEA (unlike NLP) is not particularly adept at solving them.

The task lists presented below reflected these criteria, (especially list 1). The initial states, environment and operator sets are given in appendix A.

Each task consists of one or more goal predicates to be achieved from a 'current' state; once a solution is found, the state is incrementally updated by applying the solution sequence to it. Output from some of the test runs is supplied in appendix B. We tried various configurations with the sample tests:

Configuration NO-CHUNKING:

This is MEA, the goal directed, linear problem solver with no learning components. It is, however, equipped with two weak heuristics: 'nodes with most goals satisfied in the initial state should be expanded first' and 'nodes that contain circular goals should be deleted'. This configuration is the lowest common denominator for all the configurations below.

Configuration A:

MEA is equipped with c-chunk creation under strengthening algorithm A. Only association chains of the minimum length 3 are allowed (i.e. the complexity bound in 3/7 set to '2'). Rule refinement as specified in 3.2 is implemented.

Configuration B:

MEA is equipped with c-chunk creation under strengthening algorithm B, and rule refinement as specified in 3.2.

Configuration HAND-CRAFTED:

This is Configuration B but with an initial set of hand crafted c-chunks, obtained through studying the environment and operator set given in appendix A.1.

Note that the CPU times given in the results tables are the totals; this includes applying each problem's solution sequentially to the initial state to arrive at a final state which contains the goal; and also any heuristic acquisition, repair and use that might be taking place.

Running in interpreted Prolog on a Sun 3/50 workstation, with plenty of diagnostic reporting, the implementation speeds are obviously not for absolute but relative comparison. Also, the addition of other weak, possibly preprocessing techniques, such as hierarchical operator and goal structure, would speed up all the results uniformly.

Task List1:

```
1: in_room(robot,room2);
2: in_room(box1,room2);
3: in_room(box2,room3);
4: in_room(big_box,room3);
5: in_room(box1,room6)&closed(door56);
6: in_room(robot,room7);
7: in_room(box1,room3);
8: in_room(box2,room7);
9: next_to(box2,big_box);
10: in_room(box1,room6)&
    in_room(box2,room7)&
    in_room(box2,room7);
```

Task List2:

```
1:  in_room(box1,room3).
2:  in_room(box2,room6)&
    in_room(robot,room6)&
    closed(door56)&
    closed(door67).
3:  in_room(box1,room5)&
    in_room(big_box,room5).
4:  in_room(box2,room3)&
    closed(door23)&
    closed(door35).
5:  in_room(robot,room1)&
    closed(door12).
6:  next_to(box1,box2)&
    in_room(box2,room5).
7:  in_room(box1,room7)&
    in_room(box2,room7)&
    in_room(big_box,room6).
8:  in_room(big_box,room7)&
    next_to(big_box,box1)&
    in_room(box2,room6).
9:  in_room(box1,room1)&
    in_room(robot,room2)&
    closed(door12).
10: in_room(big_box,room1)&
    next_to(big_box,box1)&
    in_room(box2,room5).
```

RESULTS FOR TASK LIST1

TASK NO &SOLN SIZE	NO-CHUNKING CPU TIME USED	NO-CHUNKING NODES EXPANDED	CONFIG. A CPU TIME USED	CONFIG. A NODES EXPANDED	CONFIG. B CPU TIME USED	CONFIG. B NODES EXPANDED	HAND-CRAFTED CPU TIME USED	HAND-CRAFTED NODES EXPANDED
1: 5	59s	20	67s	20	65s	20	27s	5
2: 5	25s	5	27s	5	29s	5	27s	5
3: 7	75s	10	62s	9	63s	9	68s	7
4: 12	185s	28	200s	28	203s	28	87s	14
5: 9	102s	18	77s	15	78s	15	72s	14
6: 4	35s	12	19s	4	17s	4	22s	4
7: 9	485s	58	340s	45	51s	9	59s	9
8: 5	98s	14	99s	13	31s	5	36s	5
9: 5	96s	15	78s	12	33s	6	38s	5
10: 23	***	**	599s	106	407s	68	452s	63

Avge soln size: 8.4

(N.B. '***' means that program ran out of space, after at least 2000s of CPU time and 200 nodes expanded)

RESULTS FOR TASK LIST2

TASK &SOLN SIZE	NO-CHUNKING CPU TIME USED	NO-CHUNKING NODES EXPANDED	CONFIG. A CPU TIME USED	CONFIG. A NODES EXPANDED	CONFIG. B CPU TIME USED	CONFIG. B NODES EXPANDED	HAND-CRAFTED CPU TIME USED	HAND-CRAFTED NODES EXPANDED
1: 7	88s	25	101s	25	99s	25	40s	8
2: 11	131s	21	108s	17	104s	17	84s	15
3: 15	1056s	161	367s	68	367s	65	252s	42
4: 11	291s	39	82s	16	82s	16	82s	16
5: 7	45s	14	53s	13	51s	13	41s	11
6: 11	541s	71	96s	18	94s	18	100s	24
7: 12	772s	99	210s	43	208s	43	196s	40
8: 10	104s	19	86s	13	87s	13	59s	12
9: 14	332s	44	171s	29	170s	29	151s	26
10: 22	***	**	472s	78	480s	78	372s	64

Avge soln size: 12 operators

RESULTS FOR TASK LIST11 — RESULTS FOR TASK LIST21

TASK &SOLN SIZE	LIST11 CONFIG. A CPU TIME USED	LIST11 CONFIG. A NODES EXPANDED	LIST11 CONFIG. B CPU TIME USED	LIST11 CONFIG. B NODES EXPANDED	LIST21 CONFIG. A CPU TIME USED	LIST21 CONFIG. A NODES EXPANDED	LIST21 CONFIG. B CPU TIME USED	LIST21 CONFIG. B NODES EXPANDED
1: 5	26s	5	23s	5	55s	15	36s	5
2: 5	25s	5	24s	5	26s	5	30s	5
3: 7	83s	10	68s	9	47s	7	48s	7
4: 12	84s	14	165s	23	229s	28	228s	28
5: 9	84s	15	76s	15	81s	14	99s	14
6: 4	20s	4	17s	4	22s	4	31s	4
7: 9	357s	44	60s	10	74s	10	81s	10
8: 5	108s	13	31s	5	36s	5	35s	5
9: 5	41s	6	34s	6	39s	6	39s	5
10:23	471s	72	414s	68	588s	90	647s	86

RESULTS FOR TASK LIST 22 — RESULTS FOR TASK LIST12

TASK &SOLN SIZE	LIST 22 CONFIG. A CPU TIME USED	LIST 22 CONFIG. A NODES EXPANDED	LIST 22 CONFIG. B CPU TIME USED	LIST 22 CONFIG. B NODES EXPANDED	LIST12 CONFIG. A CPU TIME USED	LIST12 CONFIG. A NODES EXPANDED	LIST12 CONFIG. B CPU TIME USED	LIST12 CONFIG. B NODES EXPANDED
1: 7	85s	20	114s	18	43s	8	36s	8
2: 11	94s	15	120s	15	109s	17	79s	15
3: 15	304s	59	280s	42	273s	44	244s	42
4: 11	82s	16	82s	16	89s	16	79s	16
5: 7	45s	11	55s	11	41s	11	38s	11
6: 11	103s	18	127s	18	143s	32	124s	32
7: 12	192s	40	223s	40	257s	47	184s	40
8: 10	87s	13	87s	13	164s	28	58s	12
9: 14	182s	28	229s	28	187s	26	141s	26
10: 22	498s	78	566s	64	522s	78	425s	78

3.321 Discussion of Results

It should first be noted that my hand crafted heuristic rules (given in appendix A) cut down CPU time and expanded nodes by up to a factor of 6, averaging at over 3 for this level of task complexity (these factors, in fact, generally rise as the task complexity goes up). Occasionally, in simpler tasks, the rules may only give similar performance to the basic problem solver - this is where the implanted weak heuristics are sufficient to cut down search completely.

Chunk creation in configuration B is shown to dramatically cut down times also, the last five times of B for list1 even beating HAND-CRAFTED. This is because of the higher matching cost of a more comprehensive rule set held by the latter, and is an indication that (in more complex worlds) learning from experience of relevant problems is preferable than preprocessing techniques. Configuration A has shakey results as over-specific chunks were created initially, slowing down learning and causing more repair to have to take place towards the end of the task list.

More evidence that the acquired rules are well on the way to the target concepts is given in 3(7) below. These are Configuration B's rules after list 1; they should be compared to the the hand-crafted in appendix B (note that identifiers beginning with 'x' are variables).

```
IF GOAL = in_room(robot,x(1)) THEN
/2/  IF ( in_room(robot,x(2)),
          connect(x(2),x(1),x(3)) )          unless <>      OR
/1/  IF ( in_room(robot,x(2))&open(x(3)),
          connect(x(2),x(4),x(5))&connect(x(4),x(1),x(3))&
          ne(x(2),x(1))&ne(x(5),x(3)) )      unless </2/>
     THEN CHOOSE gothrudoor(x(3),x(1))

IF GOAL = in_room(x(1),x(3)) THEN
/3/  IF ( in_room(x(1),x(4))&open(x(2)),
          connect(x(4),x(3),x(2))&type_of(x(1),box)
          &fits_thru(x(1),x(2)) )            unless <>      OR
/5/  IF ( in_room(x(1),x(4))&open(x(2)),
          connect(x(4),x(5),x(6))&connect(x(5),x(3),x(2))&
          fits_thru(x(1),x(6))&type_of(x(1),box)&fits_thru(x(1),x(2))&
          ne(x(4),x(3))&ne(x(6),x(2)) )      unless </3/>   OR
/6/  IF ( in_room(x(1),x(4)),
          connect(x(4),x(3),x(2))&type_of(x(1),box)&
          fits_thru(x(1),x(2)) )             unless <>
     THEN CHOOSE pushthrudoor(x(1),x(2),x(3))               ..3(7)
```

Task list 2 was chosen to test heuristic acquisition with more difficult tasks of several goal conjunctions. The operator set and

environment remain the same, and the initial state is given in appendix A.1

Approximating the absent CPU time to be 2000s, the hand crafted rules perform on average nearly four times faster than the basic planner, in this harder list of tasks.

This time algorithms A and B seem to match each other closely for speed, and in fact acquired similar heuristic rules. The exception was chunk /7/ (in 3(2) below) - this could only be acquired by the more powerful algorithm B, because of the limit on association chain length in algorithm A. Again the performance figures show that acquired heuristics quickly start to approximate the hand - crafted ones. For instance, after only two tasks to learn from, the time taken to solve task 3 has been cut from 1056 seconds to 367 seconds using either algorithm (remember that the latter figure is not only problem solving time but also includes time taken for chunk acquisition and refinement from the solution of task 3 !).

In task 6 a curious situation arose: A and B actually produced fewer nodes than HAND-CRAFTED. Further investigation showed that one of my handcrafted rules was faulty - I had assumed that the target concept for 'in_room(Box, Room)' was very similar in structure to 'in_room(robot, Room)' - in fact this is incorrect with the operator set used in appendix A. FM found it was less effort pushing a box into a room through two open doors (4 operators) than one closed door (5 operators), and made chunks accordingly. This can be seen by examining the rules acquired using algorithm B from list2 (shown below in 3(8)); contrast chunk /1/ with /6/: for the latter, the status of the door dosen't matter, whereas the 'open' condition is present in the former.

```
IF GOAL = in_room(robot,x(2)) THEN
/3/  IF (in_room(robot,x(3))&open(x(1)),
         connect(x(3),x(2),x(1)) )          unless <>        OR
/6/  IF ( in_room(robot,x(3)),
         connect(x(3),x(2),x(1)) )          unless <>        OR
/2/  IF ( in_room(robot,x(4))&open(x(1))&open(x(3)),
         connect(x(4),x(5),x(3))&connect(x(5),x(2),x(1))&
         ne(x(4),x(2))&ne(x(3),x(1)) )      unless </3/6/>   OR
/5/  IF (in_room(robot,x(3))&open(x(1)),
         connect(x(3),x(4),x(5))&connect(x(4),x(2),x(1))&
         ne(x(3),x(2))&ne(x(5),x(1))        unless </3/6/>   OR
/7/  IF ( in_room(robot,x(5))&open(x(3))&open(x(4))&open(x(1)),
         connect(x(5),x(6),x(3))&connect(x(6),x(7),x(4))&
         connect(x(7),x(2),x(1))&ne(x(5),x(7))&ne(x(5),x(2))
         &ne(x(3),x(4))&ne(x(3),x(1))&
         ne(x(6),x(2))&ne(x(4),x(1)) )      unless <>
     THEN CHOOSE gothrudoor(x(1),x(2))
```

```
IF GOAL = in_room(x(1),x(3)) THEN
/1/  IF ( in_room(x(1),x(4))&open(x(2)),
          connect(x(4),x(3),x(2))&type_of(x(1),box)&
          fits_thru(x(1),x(2)) )                    unless <>       OR
/4/  IF ( in_room(x(1),x(5))&open(x(4))&open(x(2)),
          connect(x(5),x(6),x(4))&connect(x(6),x(3),x(2))&
          fits_thru(x(1),x(4))&type_of(x(1),box)&
          fits_thru(x(1),x(2))&ne(x(5),x(3))
          &ne(x(4),x(2)) )                          unless </1/>    OR
     THEN CHOOSE pushthrudoor(x(1),x(2),x(3))                     ..3(8)
```

The second pair of tables were constructed after both task lists had been executed again, but this time the acquired heuristics rules from the first runs were used. Rule acquisition continued, so that more chunks could be formed if necessary, and rules could be further refined. Task listXY means task listY is run with heuristics acquired from execution of listX.

Although performance is similar to hand-crafted rules, one or two tasks still lag behind, even in LIST11 where exactly the same tasks are given again. This is because of chunk interaction - situation type (c) is occurring (see section 3.2, figure 3/9); once these corrective exceptions are made, this problem disappears.

A more important problem may occur where a task has multiple solutions of equal length. In list1/task4/B a chunk advises two paths to follow, and is subsequently called in for strengthening. There is a dilemma: should the planner search exhaustively to make sure that the non-solution path does not lead to an equal solution (each time this occurs!) as did the Lex system [Mitchell 83] or should it always apply the strengthening algorithm?. There does not seem to be any definitive answer to this, since searching for every possible solution is obviously expensive if not impossible. I decided on the following compromise for the actual implementation:

* don't look down each path - this is far too slow if the learning time is being taken as an integral part of the planning time (as in our case).

* send the advising chunk to be strengthened. If the paths are indeed equal, the algorithm may find no distinguishing features, in which case the chunk is re-instated, and not allowed to be strengthened again. If there are distinguishing features, these are of course added; this means that at worst an over-specific chunk would be developed (but no more so than would be developed using strict E.B.G.!).

Another problem with chunks, which has plagued learning systems at least as far back as STRIPS with Macrops [Fikes et al 72] is the overhead in matching cost of learning components. As the

application domains we consider become more complex, chunks will have to be created with a hierarchical structure, as will operators and the rest of the domain definition. Our chunk optimisation routines (specified in 3.2) alleviate this: - reordering predicates so that the sparse ones are matched first, and reordering chunks so that the most commonly used and simplest are tried first. However, we found that the matching cost of the hand-crafted chunks was much lower because of an incorrect, but efficient assumption we had made: Compare chunk /7/ in 3(8) above with it's counter-part in the hand-crafted set (appendix B):

```
......................
......................
/7/  IF ( in_room(robot,x(5))&open(x(3))&open(x(4))&open(x(1)),
          connect(x(5),x(6),x(3))&connect(x(6),x(7),x(4))&
          connect(x(7),x(2),x(1))&ne(x(5),x(7))&ne(x(5),x(2))
          &ne(x(3),x(4))&ne(x(3),x(1))&
          ne(x(6),x(2))&ne(x(4),x(1)) )      unless <>
     THEN CHOOSE gothrudoor(x(1),x(2))
```

Two of the three 'open' conditions in /7/ have free variables, and cause a considerable overhead in matching, which is not present in the hand crafted (over general) rule above. In my implementation the state condition is matched, then the environment (i.e. a procedural implementation of equation 3(4)) Thus the correct instantiation would generally only be found after some backtracking. I tried merging the state and environment conditions of chunks, then ordering their predicates according to sparseness: but this did not alleviate the problem, since structuring the chunk into 'changeable' and 'non-changeable' components had itself been efficient. Structuring the chunk further so that components have a slot in which to place predicates which contain only 'free' variables, seems to be the answer.

In comparison, Minton and Carbonell [Minton and Carbonell 87] treat the problem differently: their Prodigy system uses feedback on the problem solving time a rule saves vs. its matching cost; if this comparison is adverse, the system may delete a rule all together.

A final problem encountered by c-chunk creation in the robot world is that of picking the wrong discriminating feature. Obviously the probability is lessened by the goal regression procedure, and by 'backwards-directed' strengthening, but may still occur; This problem was highlighted in Winston's arch-learning program [Winston 75]: the trainer was expected to present counter examples with just one major discriminatory feature (although Winston claimed the program would keep alternatives and 'backtrack' if necessary). In his system, the counter examples presented are analogous to our incorrect paths. The the problem is: how can the learner pick out which features are co-incidental, and which discriminate.

In the case of my implementation used for the tests, this problem is dealt with effectively since chunk creation is incremental and heuristic rules are refined: erroneous heuristics simply make learning slower and may initially lead to non-minimal solutions. This is since chunks are learned and used immediately; one way of eradicating the effects of a 'bad chunk' is by introducing a learning session, where tasks are solved in learning mode only, where heuristics are acquired but not used (earlier learning programs like LEX [Mitchell et al 83] seem to only work in this manner). To exemplify this, and to show the incremental chunking mechanism is reasonably robust to this sort of 'noise' I created a new configuration:

Configuration 'Bad'

MEA is equipped with c-chunk creation under strengthening algorithm B, and rule refinement as specified in 3.2; it is supplied with the chunks obtained from list1, and the two bad chunks given below:

```
ch(ch1, gothrudoor(x(1),x(2)),
        in_room(robot,x(2)),
        in_room(x(0),x(3))&open(x(1)),
        connect(x(3),x(2),x(1))&type_of(x(0),box) )

ch(ch2,pushthrudoor(x(1),x(2),x(3)),
        in_room(x(1),x(3)),
        in_room(robot,x(4))&open(x(2)),
        connect(x(4),x(3),x(2))&type_of(x(1),box)&
        fits_thru(x(1),x(2)) )
```

List2 was then executed; since this is analogous to test list12, I will call the results list12':

RESULTS FOR TASK LIST12'

TASK &SOLN SIZE	NO-CHUNKING CPU TIME USED	NO-CHUNKING NODES EXPAND-ED	CONFIG.'BAD' CPU TIME USED	CONFIG.'BAD' NODES EXPAND-ED	CONFIG. B CPU TIME USED	CONFIG. B NODES EXPAND-ED	HAND-CRAFTED CPU TIME USED	HAND-CRAFTED NODES EXPAND-ED
1: 7	88s	25	63s	15	36s	8	40s	8
2: 11	131s	21	122s	18	79s	15	84s	15
3: 15	1056s	161	271s	44	244s	42	252s	42
4: 11	291s	39	88s	16	79s	16	82s	16
5: 7	45s	14	41s	11	38s	11	41s	11
6: 11	541s	71	146s	32	124s	32	100s	24
7: 12	772s	99	254s	47	184s	40	196s	40
8: 10	104s	19	163s	28	58s	12	59s	12
9: 14	332s	44	199s	28	141s	26	151s	26
10: 22	***	**	462s	70	425s	78	372s	64

These results compare well alongside the list12 results, and indicate that the chunk refinement technique can cope with 'noisy' or bad chunks. The residual rule set after these tasks have been executed are supplied in appendix B. The reader will see that the original chunks have been 'exceptioned out'.

In conclusion, despite the problems listed, all these results in the robot world show that c-chunk creation works extremely well in this domain, with the performance of algorithm B being generally better than A. LIST1/B and LIST12/B give most encouraging results: six out of the first ten times being better than hand-crafted, and by the second 10 tasks the times are better than the hand-crafted in 8 out of 10 cases. This is as expected: the 20 different tasks by and large increase in difficulty (with the exception of LIST1/10).

3.33 The Warehouse World Experiments

This domain proved to be more complex than the robot world since the number of goal interactions among preconditions was much higher. Also the number of target concepts (operator - goal combinations) is much higher at over twenty. This became apparent when I attempted to handcraft rules by studying the domain, and the final set of chunks (in appendix B) was far from ideal; in fact in the test runs, the handcrafted set benefited greatly from chunk acquisition. My experience with trying to produce optimal control rules seems to concur with others who have tried this exercise (e.g. [Carbonell 88]): letting FM produce the chunks was far easier!

Again two task lists, list3 and list4 were chosen and executed. This was a random choice except for the criteria as discussed in the previous robot experiments, and ensuring that the basic problem solver could actually solve the problems themselves.

The configurations chosen were:

Configuration NO-CHUNKING: as in the robot world.

Configuration B: MEA is equipped with c-chunk creation under strengthening algorithm B, and rule refinement as specified in 3.2. Learning and problem solving times are shown separately this time (in the robot world learning time was not significant, averaging at less than ten per cent of the total).

Configuration HAND-CRAFTED: as in the robot world.

Again the CPU times given are inclusive. An initial world and environment is shown in figure 3/10, and the full domain definition is in appendix A.2. Sample results are included in appendix B.

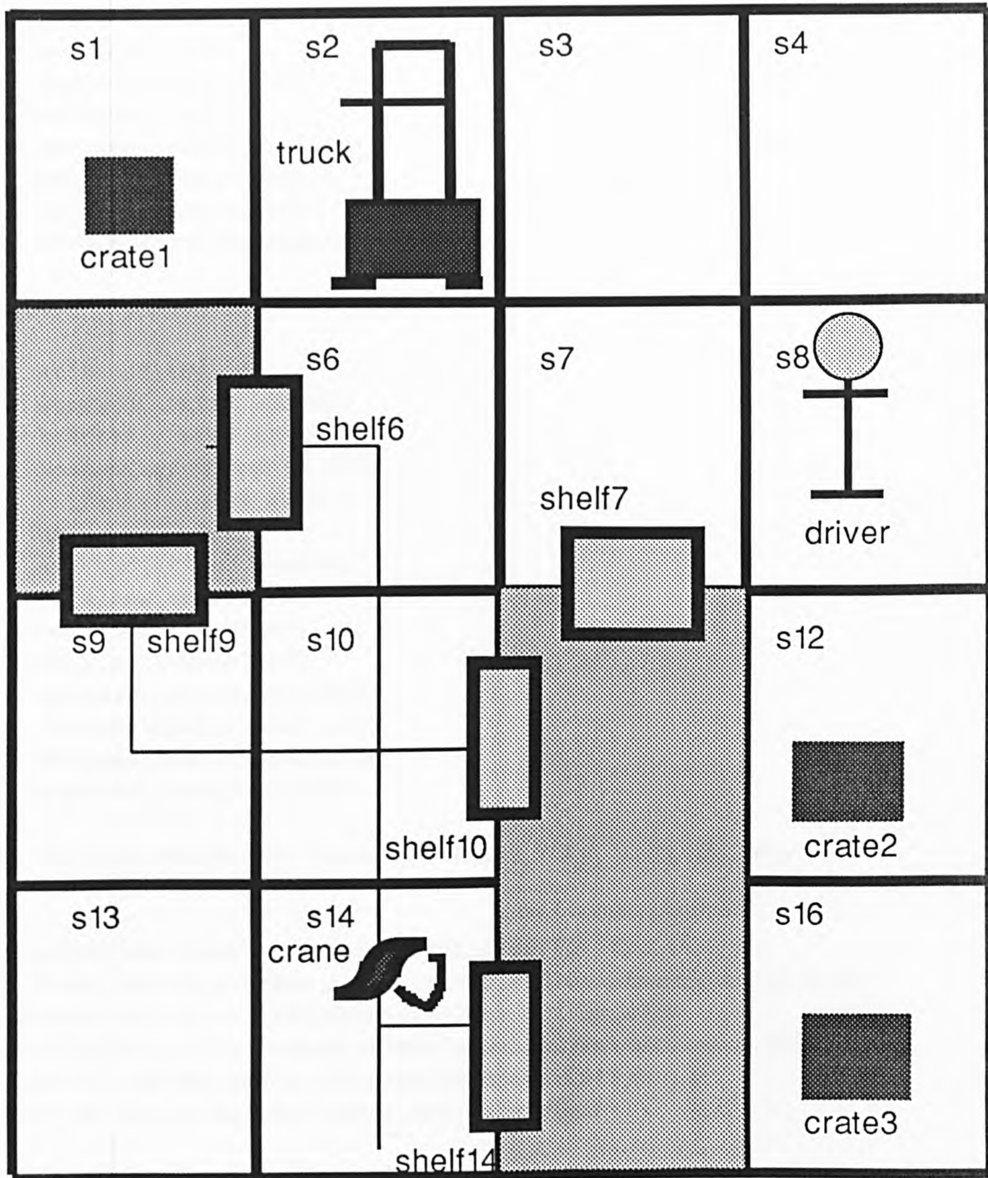

figure 3/11: warehouse world

The task lists were:

LIST3:

```
1:  in_truck.
2:  at(truck,s8).
3:  loaded(truck,crate3,s16).
4:  at(crate3,s8).
5:  loaded(truck,crate2,s7).
6:  at(crate2,s9).
7:  stacked(crate2,shelf10).
8:  stacked(crate2,shelf14).
9:  on_floor(crate1,s10).
10: stacked(crate3,shelf7).
```

LIST4:

```
1:  at(truck,s6).
2:  stacked(crate2,shelf7).
3:  stacked(crate1,shelf14).
4:  loaded(truck,crate3,s12).
5:  on_floor(crate3,s2)&
    at(crate2,s4).
6:  stacked(crate3,shelf6).
7:  on_floor(crate2,s1)&
    on_floor(crate3,s9).
8:  on_floor(crate1,s2).
9:  stacked(crate3,shelf9)&
    loaded(truck,crate1,s12).
10: stacked(crate1,shelf10)&
    stacked(crate2,shelf6).
```

The initial states for list3 and list4 were, respectively:

```
unloaded(truck)&unloaded(crane)&on_floor(driver,s9)&
on_floor(crate2,s12)&on_floor(crate3,s16)&on_floor(crate1,s1)&
at(crate1,s1)&clear(s10)&clear(s3)&at(driver,s9)&
clear(s4)&clear(s5)&clear(s7)&at(truck,s2)&at(crate2,s12)&
at(crate3,s16)&clear(s14)&clear(s6)&clear(s11)&
clear(s8)&clear(s13)&at(crane,s14)&clear(s15)
```

```
unloaded(truck)&unloaded(crane)&on_floor(driver,s12)&
on_floor(crate2,s2)&on_floor(crate3,s8)&on_floor(crate1,s1)&
at(crate1,s1)&clear(s10)&at(driver,s12)&clear(s5)&
clear(s7)&at(truck,s16)&at(crate2,s2)&at(crate3,s8)&
clear(s14)&clear(s6)&clear(s11)&clear(s4)&clear(s13)&
clear(s3)&clear(s9)&at(crane,s14)&clear(s15)
```

RESULTS FOR TASK LIST3

TASK NO &SOLN SIZE	NO-CHUNKING CPU TIME USED	NODES EXPAND-ED	CONFIG. B CPU TIME PROB.SOLV;	LEARNING	NODES EXPANDED	HAND-CRAFTED TOT. TIME USED	NODES EXPAND-ED
1: 3	70s	9	70s	9s	9	81s	9
2: 3	36s	5	35s	10s	5	36s	4
3: 5	53s	5	53s	5s	5	62s	5
4: 2	20s	2	19s	4s	2	18s	2
5: 3	116s	12	124s	7s	12	104s	10
6: 3	44s	7	42s	11s	7	50s	6
7: 2	35s	3	32s	3s	3	35s	3
8: 6	379s	32	386s	12s	32	350s	31
9: 7	701s	94	389s	24s	48	376s	48
10: 7	472s	60	181s	2s	23	164s	23

RESULTS FOR TASK LIST4

TASK NO &SOLN SIZE	NO-CHUNKING CPU TIME USED	NODES EXPAND-ED	CONFIG. B CPU TIME PROB.SOLV.	LEARNING	NODES EXPANDED	HAND-CRAFTED TOT. TIME USED	NODES EXPAND-ED
1: 7	383s	42	384s	16s	42	466s	42
2: 5	72s	10	67s	10s	9	72s	9
3: 7	168s	29	165s	33s	27	199s	25
4: 7	462s	58	106s	31s	12	147s	13
5: 7	1480s	80	1435s	34s	80	1007s	66
6: 9	764s	56	750s	24s	56	720s	53
7: 10	704s	82	631s	24s	69	669s	65
8: 7	1319s	138	279s	11s	43	313s	43
9: 8	91s	17	96s	15s	14	116s	14
10: 13	2086s	151	597s	41s	58	632s	55

RESULTS FOR TASK LIST33

TASK NO &SOLN SIZE	NO-CHUNKING		CONFIG. B			HAND-CRAFTED	
	CPU TIME USED	NODES EXPAND-ED	CPU TIME PROB.SOLV;	LEARNING	NODES EXPANDED	TOT. TIME USED	NODES EXPAND-ED
1: 3	70s	9	25s	1s	3	81s	9
2: 3	36s	5	39s	1s	4	36s	4
3: 5	53s	5	51s	2s	5	62s	5
4: 2	20s	2	15s	1s	2	18s	2
5: 3	116s	12	35s	1s	3	104s	10
6: 3	44s	7	22s	1s	3	50s	6
7: 2	35s	3	30s	1s	2	35s	3
8: 6	379s	32	94s	5s	10	350s	31
9: 7	701s	94	199s	14s	20	376s	48
10: 7	472s	60	163s	2s	23	164s	23

RESULTS FOR TASK LIST43

TASK NO &SOLN SIZE	NO-CHUNKING		CONFIG. B			HAND-CRAFTED	
	CPU TIME USED	NODES EXPAND-ED	CPU TIME PROB.SOLV;	LEARNING	NODES EXPANDED	TOT. TIME USED	NODES EXPAND-ED
1: 3	70s	9	71s	9s	9	81s	9
2: 3	36s	5	35s	8s	4	36s	4
3: 5	53s	5	51s	2s	5	62s	5
4: 2	20s	2	19s	4s	2	18s	2
5: 3	116s	12	118s	7s	13	104s	10
6: 3	44s	7	51s	7s	6	50s	6
7: 2	35s	3	25s	1s	4	35s	3
8: 6	379s	32	53s	2s	9	350s	31
9: 7	701s	94	170s	19s	20	376s	48
10: 7	472s	60	239s	2s	26	164s	23

RESULTS FOR TASK LIST44

TASK NO &SOLN SIZE	NO-CHUNKING CPU TIME USED	NO-CHUNKING NODES EXPAND-ED	CONFIG. B CPU TIME PROB.SOLV.	CONFIG. B CPU TIME LEARNING	CONFIG. B NODES EXPANDED	HAND-CRAFTED TOT. TIME USED	HAND-CRAFTED NODES EXPAND-ED
1: 7	383s	42	107s	6s	10	466s	42
2: 5	72s	10	83s	6s	10	72s	9
3: 7	168s	29	130s	26s	18	199s	25
4: 7	462s	58	81s	16s	8	147s	13
5: 7	1480s	80	452s	20s	44	1007s	66
6: 9	764s	56	134s	3s	13	720s	53
7: 10	704s	82	176s	7s	20	669s	65
8: 7	1319s	138	48s	2s	7	313s	43
9: 8	91s	17	79s	2s	9	116s	14
10: 13	2086s	151	441s	39s	43	632s	55

RESULTS FOR TASK LIST34

TASK NO &SOLN SIZE	NO-CHUNKING CPU TIME USED	NO-CHUNKING NODES EXPAND-ED	CONFIG. B CPU TIME PROB.SOLV.	CONFIG. B CPU TIME LEARNING	CONFIG. B NODES EXPANDED	HAND-CRAFTED TOT. TIME USED	HAND-CRAFTED NODES EXPAND-ED
1: 7	383s	42	509s	17s	42	466s	42
2: 5	72s	10	114s	2s	15	72s	9
3: 7	168s	29	75s	9s	13	199s	25
4: 7	462s	58	141s	32s	12	147s	13
5: 7	1480s	80	1461s	12s	76	1007s	66
6: 9	764s	56	558s	16s	47	720s	53
7: 10	704s	82	464s	12s	53	669s	65
8: 7	1319s	138	230s	4s	35	313s	43
9: 8	91s	17	84s	15s	13	116s	14
10: 13	2086s	151	450s	7s	45	632s	55

3.331 Discussion of Results

The results again show that the presence of c-chunking generally increases the time and space efficiency of problem solving, but with this domain, improvement is not as smooth nor as quick, although in some cases it can be very large (list44/B/task8 for example). The handcrafted rules were poor compared to the robot world, and relied on automatically created c-chunks to improve their performance.

List3/B took until task 9 to use any of its acquired chunks, and then the last two tasks showed a marked improvement. The reason for this is seen by considering the other runs for this list: the first seven tasks in list3 are mostly too simple to need strong heuristics. The larger problems 8, 9 and 10, however, benefit from them; in retrospect, if I were to bias list3 to show chunk acquisition in a better light, then I would have made tasks 1-7 more complex!

List4/B includes some dramatic improvements: for example after only learning from 3 problems, task 4 is solved in just over one fifth of the expanded nodes that were taken in list4/no-chunking. Task 5 unfortunately did not benefit at all, whereas task 8's time is cut down by a factor of almost 5.

The test runs which use previous experience show that problem solving time and learning time generally decrease (except as previously mentioned, where tasks are too simple to benefit) e.g. after a shakey start, list34/B outperforms list4/B, whereas list44 is dramatically better than both of these.

Similar problems cropped up in warehouse as discussed in 3.321: as in the robot world, refinement of chunks was not smooth, the major cause being the matching problem. In particular, long relational chains could be generated using algorithm B, if discriminating factors were purely spatial. For example, moving a crate from s16 to s13 involves a repetition of operator 'drive_load' over several spaces. The weakest precondition of the solution sequence would contain a long spatial chain (of relation 'next') which would cause a large matching overhead, if it were picked out by the strengthening algorithm.

To deal with this, a limit was placed on the size of relational chains which could be produced during the strengthening algorithm. This brought it into line with algorithm A, whose association chains already have a complexity limit. Although it is actually a problem of the inadequacy of representing spatial relations using propositional and not analogue means, some automatic control for combinatorics in matching is inevitable. Unfortunately, this in turn means that discriminating features of a weakest precondition which are relationally far from the final goal will not be picked up by strengthening.

3.34 Conclusions

The experiments above are evidence that creating, using and refining c-chunks increases the efficiency of a general planner in the particular domain to which it is applied: tasks which share similarities are solved in decreasing time. In a sense chunking also increases the power of the planner since more difficult tasks can be solved than previously, because the the chunks also cut down on the number of expanded nodes. Other general conclusions are that more experience leads to more refined heuristics and does not tend to 'clog up' the system, and in the robot world at least, the acquired heuristic rules converge towards the hand crafted target concepts.

As stated in 3.32 and 3.33, the task lists were chosen at random except for the criteria stated in 3.32. In particular they were not carefully selected to give good performance.

The difference in learning curve between the robot and warehouse domains can be reduced to two major differences: in the robot world there were only essentially two target concepts that needed to be learned, in the warehouse there are nearer twenty; also, in the more complex world some chunks were not made because of over-long chains of relations.

Finally I will suggest some improvements to c-chunking which should go some way towards solving the problems encountered.

To summarise the problems from 3.32 and 3.33:

1. Learning is slowed because of chunk interaction;

2. Generated chunks may include features which should not be included in the target concept.

3. multiple equal length solutions can, on occasion, cause strengthening where it is not needed.

4. the overhead in matching cost of some chunks is high.

Two strategies which go some way to solving problem 1. and 2. respectively are as follows:

1. In the present system, an operator instance only has to be matched by one chunk for it to be favoured for expansion; instead the set of all chunks that favour operators could be recorded, with a speed-up in rule refinement, since more examples (chunks that favour the correct path) and counter examples (chunks that need specialising because they favour the wrong path) of the target concept would be available.

2. During the strengthening process, a chunk in made after the

FIRST discriminating feature is found. If this process would be allowed to continue, picking out other discriminating features and creating more chunks for the same target concept, it would ensure that no important features were missed, at the expense of creating more chunks. Further experience would single out the chunks nearest the target concept.

Both these methods, however, would cause an overhead: the first in problem solving time, which is more critical, and the second in learning time.

Problem 3. may be solved by using a more powerful general planner instead of MEA, one that produces a partially ordered operator set as a solution (thus specifying a set of equal length solutions): it would give the learner better information from which to extract heuristics, as pointed out in 4.1 below.

I would speculate that problem 4. is an issue of basic domain representation: it may be attacked by the introduction of abstraction levels, or some form of automatic representation change such as that used in constructive induction (defined in [Michalski et al 83]). Cruder techniques have been advocated such as deleting heuristics which prove worthless over a period of problem solving (e.g.[Minton 85]). Deleting chunks which have not been successfully used in the last 'N' problems had occurred to me, and most certainly would have cut down some matching time (e.g. task 1 in list34/B) but appears crude: addition of ad hoc optimisation rules for these two particular domains may have made the results more attractive, but would not have benefited the overall thesis.

4. TOWARDS HEURISTIC ACQUISITION IN NON-LINEAR PLANNERS

4.1 Introduction

Experimental research into learning and planning has predominantly been carried out with linear performance systems. A non-linear planner (NLP) has a number of advantages as the performance component of a learning system. For instance, an NLP can solve more complex problems than usual goal directed linear planners i.e. those which try to find solutions to goal predicates without the power to interleave these with solutions to other goal predicates. Under certain constraints, the finding of an optimal solution in polynomial time and space using NLP can be guaranteed (as proved in [Chapman 87]). Also, the output of a linear planner may be misleading to a learning component that accepts it, because of its over commitment to ordering; in the linear planner of the chapter before, when the first solution is found, planning stops and this solution is used for chunking. It seems unreasonable to carry on and search for multiple minimal solutions, without knowing of there existence, or when to terminate. But better quality heuristic acquisition would be obtained from the analysis of the correct partial order or least committed sequence of operators, where any completion of the sequence can form a solution, such as that output from my constraint posting NLP.

This chapter is mostly speculative: using the system as described in chapter 1.23 and appendix D.5, I discuss the type of choices made in the goal achievement components of NLP, explore one method of heuristic acquisition, and develop an example from a prototype implementation (N.B. [McCluskey 88a] or appendix D.5 is background to this chapter and introduces all the relevant notation).

4.2 The Search through Partial Plan Space

Our NLP has to make several types of choices during search and partial plan generation, and some are similar in nature to those discussed in 3.1, i.e.

1 -- Which node (partial plan) PP to expand next?
2 -- Which goal predicate P in PP to achieve?

As stated previously, 1 can be addressed indirectly by creating less partial plans, and 2 is alleviated by the addition of hierarchy (indeed our original version of NLP has already been extended to an hierarchical planner called HNLP see [Fox 88]).

The choices involved in goal achievement itself are different, however, and these are the ones we will investigate. Referring to NLP's specification we can deduce the types of choice available to a goal achievement algorithm implementing it; consider an arbitrary partial plan PP = pp(Os,Ts,Ps,As,Es) generated during the search

for a solution to task (I,G,E,OS). If P is an unachieved precondition predicate of O in Os (i.e. (P,O) is in Ps), then in trying to achieve goal P at O we can choose:

--to use an existing operator instance within Os (if possible),
or
--to add a new operator to Os, from OS. -4(1)

Both of these choices may themselves involve choices. Once an operator A has been selected, is present in the partial plan, and has been partially instantiated so that it's add-list will necessarily assert P, there are choices involved in 'de-clobbering' P. For instance, if an operator instance C in Os can possibly delete P, a choice must be made to:

--constrain Ts so that C is necessarily before A
or
--constrain Ts so that O is necessarily before C
or
--constrain Es so that nothing in C's delete list unifies with P

-4(2)

Adding a new operator from OS to a partial plan is the only way that otherwise necessarily achieved preconditions can be clobbered. Each P in Ps must therefore be de-clobbered every time a new operator is added - this again may entail choices of constraint addition such as the three described in 4(2).

The discussion above implies that c-chunks may not be useful in cutting down the search space of the NLP. After experimenting with this possibility, I reached the following conclusions:

a. C-chunks cannot be generated in the same way as they are in linear goal directed search, since failed paths at the 'state space' level are not available.

b. The NLP is a constraint posting planner, meaning choices between operator instantiations are postponed, and therefore the main application of the c-chunk, to choose between instantiations, is not present in the search. Chunks can, however, choose a particular operator in favour of others, and may also be used in 'generate mode'. In the latter case, they generate promising operator instantiations and add them to the partial plan.

c. As mentioned in b., chunks in generate mode do help the search, but unfortunately miss the main source of combinatorics: this is invariably in the choice of temporal and variable constraints possible during the declobbering stage.

From 3. it follows that an approach to heuristic acquisition in this type of planner must be integrated with NLP; in effect, generated heuristics should advise on all the choices in goal achievement. The approach we shall investigate relies on applying

the state space paradigm to partial plan space: each choice considered in 4(1) and 4(2) will actually correspond to the application of a 'partial plan space operator'. These 'operators' produce new nodes by changing the components of a partial plan and are in effect the procedural implementation of part of our NLP specification in appendix D.5; to avoid confusion with planning operators we will call them 'transforms'.

In general, given a partial plan PP = pp(Os,Ts,Ps,As,Es), a transform produces a new partial plan of the form (see figure 4/1):

PP' = pp(Os+Op, Ts+T, Ps-(P,O)+Pre(Op), As+(P,O), Es+B)

where PP' is valid (refer to the defining data type invariant in A.5) and:

* Op is a operator instance, and Op.a contains a predicate unifying with P;

* T is a temporal partial order on Os+Op;

* B is a set of variable constraints;

* Pre(O) = { (P,O): P is a precondition predicate of operator O};

(..any of the above components may also be null)

* (P,O) is a member of Ps;

4.3 Heuristic Acquisition using EBG

We have effectively re-represented non-linear planning as a state space search at a higher level (i.e. in partial plan space). Since the transforms have been stated declaratively, the technique of Explanation Based Generalisation can be used, as in chapter 2, in which to substantiate our approach to heuristic acquisition. Recall that there must be four components involved in EBG (from 2(1)):

(a) the target concept: what is to be learned;

(b) operationality criteria: the form in which the learned concept description must be encoded;

(c) the domain theory: a 'deep' non-operational definition of the target concept;

(d) an example of the target concept.

In the partial plan space these will be:

(a) a matching condition defining the set of partial plans for which an operator H of form:

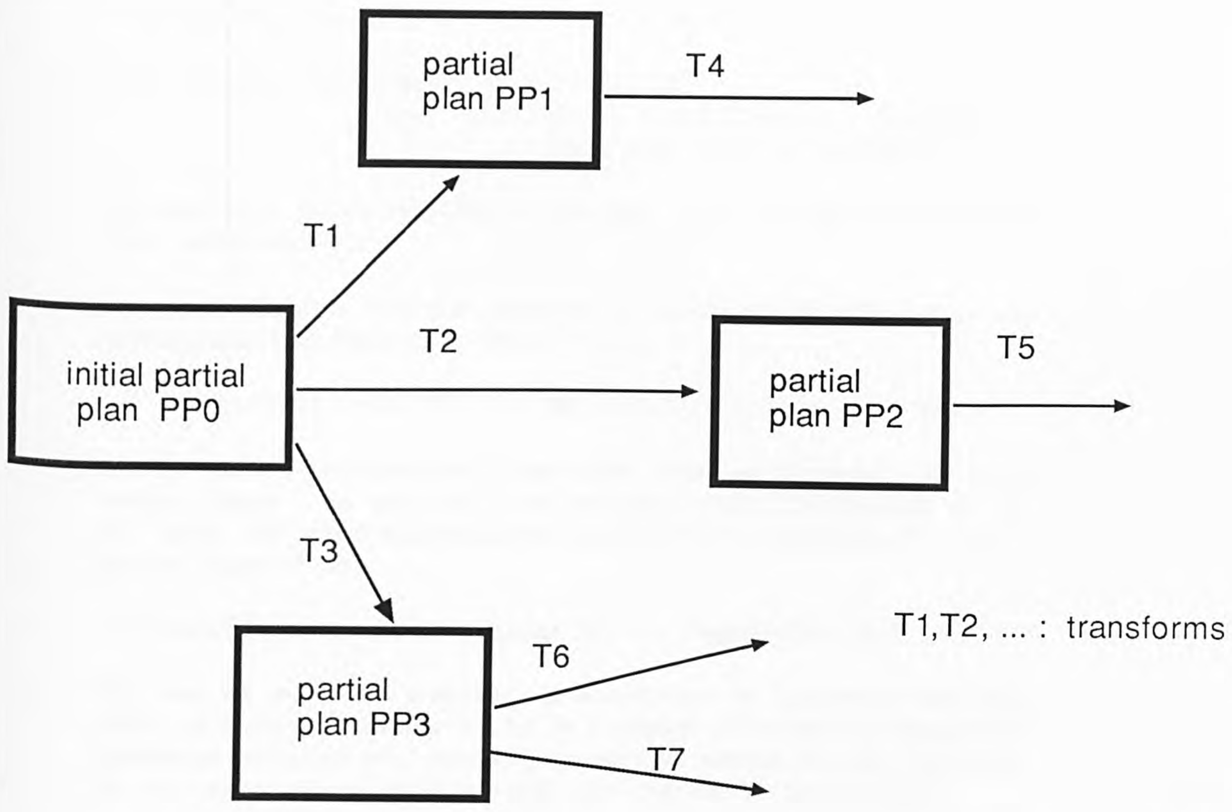

figure 4/1: Partial Plan Space

```
(+Op,+T,-(P,O)+Pre(Op),+(P,O),+B)
```

is useful, in other words an operational definition of

```
{PP : PP is a valid partial plan, PP is not a 'goal state'
      (i.e. not(PP.Ps = { }) and H(PP) is on the
            minimal solution path };
```

(b) a description which matches with (and thus defines a set of) partial plans;

(c) problem solving rules such as in [Mithchell et al 86]:

- for all PP, solved(PP) if and only if PP.Ps = { }.

```
- for all PP, solvable(PP) if and only if
                          (solved(PP) V there exists  operator
                              H such that H(PP) is solvable)
```

together with rules specific to the NLP, e.g. the specification of 'goal achievement';

(d) a partial plan PP0 and operator H1 such that H1(PP0) is on the optimal solution path from PP0:

```
PP0  --- H1 --->  PP1  --- H2 ---> .... --- Hn --->  PPn
```

Having posed the problem in the same terms as Mitchell et al's seminal paper, we conclude from the proof tree ([Mithchell et al 86] page 63) that the heuristic condition for applying H1 to a partial plan PP is:

```
matches( PP, regress(H1,regress(H2, ... regress(Hn,PPn) ...)).
```

This can be used as a heuristic precondition in 'generate and test mode' in future planning: if H1 is a member of a set of transforms generated for plan PP, then H1(PP) will be chosen as the next node in the search space if PP matches the regression expression.

There are complications, however: the language of all valid partial plans is complex and does not readily admit a convenient concept description language; also transforms are not stored or known a priori, and in fact are theoretically infinite in number. But I will now address another important issue - how to evaluate the expression 'regress(H,PP)' for any valid partial plan PP and transform H.

A transform H = (+Op,+T,-(P,O)+Pre(Op),+(P,O),+B) applied to PP, by definition, achieves goal P at operator O in H(PP). The regression of the final transform Hn from last partial plan PPn = pp(Os,Ts,Ps,As,Es) is the precondition for its application minus the addition it makes to the partial plan i.e. it is given by the actual instances of Os, Ts and Es used in the proofs of the

goal achievement specification (see section 1.3), minus the parts added by it.

Hence if Os', Ts', Es' are the actual subsets of Os, Ts, Es respectively, used in the specification proofs, then

regress((+Op,+T,-(P,O)+Pre(Op),+(P,O),+B), pp(Os,Ts,Ps,As,Es))

= (Os'-Op, Ts'-T, Ps+(P,O)-Pre(Op), As-(P,O), Es'+B) = Rn

Continuing in a similar fashion to the the regression technique discussed in 2.2, if

Hn-1 = (+Op',+T',-(P',O')+Pre(Op'),+(P',O'),+B') then

regress(Hn-1, regress(Hn , pp(Os,Ts,Ps,As,Es))

= Rn U

(Os"-Op', Ts"-T', Ps+(P',O')-Pre(Op'), As-(P',O'), Es"+B'),

where Os", Ts" and Es" are the subsets of PPn-1's relevant components used in the specification proofs.

Finally, we can make the recursive definition:

regress(H1,regress(H2, ... regress(Hn,PPn) ...)) =

regress(H2, ... regress(Hn,PPn) ...))

U (Os"-Op, Ts"-T, Ps+(P,O)-Pre(Op), As-(P,O), Es"+B) ..4(3)

where H0 = (+Op,+T,-(P,O)+Pre(Op),+(P,O),+B), and and Os", Ts" and Es" are the subsets of PP0's relevant components used in the specification proofs.

4(3) can be built for any final subsequence of the achieving transforms, Hi, .. Hn, 1 =< i =< n, and is a 'generalised partial plan' which specifies the set of partial plans which contain it. Using the same argument as that in chapter 2, the regression expression can be generalised further by generalising objects in the transforms as long as the proofs of the preconditions for each transform is not violated.

4.4 An Example Application

The implementation of NLP has one efficiency feature which limits the full reconstruction of transforms: partial plan variables are also prolog variables, and so their instantiations are lost. This could be remedied by the use of a meta-interpreter (e.g. like that used in [Krawchuk and Witten 88]) but my implementation for simplified transforms is adequate to throw light on some major problems facing this approach, without the need for this extra

complexity. In fact the meta-interpreter of [Kedar-Cabelli and McCarty 87] was tried on NLP, using the prolog implementation as a specification of the performance component. Unfortunately a combination of the complexity of the target program and the use of some 'procedural bits' proved too much for this approach. Instead the implementation of the partial plan abstract data type was extended to include a component which recorded those parts of a partial plan accessed and changed during goal achievement.

The example below is taken from the test runs. We use the block's world example, and take the specific task to be the well known 'Sussman's Anomaly'. E and OS are defined in appendix A.8: note that a simpler list format representation is used for operator schemas than presented in chapter 1, so that addition of operator instantiations to partial plans is more efficient. A routine is included in the same appendix defining the change between this representation and that of FM.

```
If  I  = on(a,table)&on(c,a)&on(b,table)&clear(c)&clear(b)
    G = on(a,b)&on(b,c)
```

then the transforms to achieve this task specification are, (written in the simplified form (+Op,+T,+(P,O)):

```
H1  = (puton(b,c),[],(on(b,c),goal))

H2  = (puton(a,b),[],(on(a,b),goal))

H3  = (none,[],(clear(c),puton(b,c)))

.......

.......

H9  =   (none,[t(newtower(c,a),puton(b,c))],(clear(c),newtower(c,a)))

H10 =   (none,[],(on(c,a),newtower(c,a)))
```

Now partial plan PP10 is (where op1, op2, op7, init, goal are simply identifiers)

```
pp( [
 (init,init,[],
       [on(a,table),on(c,a),on(b,table),clear(c),clear(b)],[]),
 (goal,goal,[(on(a,b),on(b,c)],[],[]),
 (op1,puton(b,c),[on(b,table),clear(b),clear(c)],
       [on(b,c),clear(table)],[on(b,table),clear(c)]),
 (op2,puton(a,b),[on(a,table),clear(a),clear(b)],
       [on(a,b),clear(table)],[on(a,table),clear(b)]),
 (op7,newtower(c,a),[on(c,a),clear(c)],[clear(a),on(c,table)],
       [on(c,a)]) ],

[t(op7,op1), t(op7,op2), t(op1,op2)],
```

```
[ ],

[(on(c,a),op7), (clear(c),op7), (on(a,table),op2), (clear(a),op2),
(clear(b),op2), (on(b,table),op1), (clear(b),op1),
(clear(c),op1), (on(a,b),goal), (on(b,c),goal)], [ ] ).
```

and using our regression formula:

```
regress(PP10,H10) =

(Os contains [init, puton(b,c), puton(a,b), newtower(c,a) ]),
(init contains on(c,a)) & (Ts contains [t(newtower(c,a),puton(b,c)),
 t(newtower(c,a),puton(a,b))])
```

After generalisation along the lines of the EBG theory, as applied to forward search as detailed in chapter 2, this could generate a heuristic such as:

```
for all (distint) operator instantiations O1,O2,O3,
for all predicates P,  IF

(Os contains [init, O1, O2, O3 ]) &
(init adds P) &
(Ps contains P) &
(Ts contains [t(O3,O1), t(O3,O2)]) THEN

use transform (none,[],(P, O3))
```

4.5 Discussion and Future Work

This chapter has been mainly speculative, although the regression equation described has been implemented on top of the basic NLP. A clean, novel approach to speeding up planning in domain independent constraint posting planners (as typified by NLP) is proposed by the automatic construction of search control rules in partial plan space. These rules would store not only the conditions under which certain operators are needed, but also the correct choices of temporal and variable constraints. But major problems face the development of this line of research:

(1) the regression of transforms leads in general to disjunctive formulae, since there is usually more than one way that goals may be declobbered. For instance, in the example above, sub-goal on(c,a) may be declobbered by the fact that it is not deleted by either puton(a,b) or puton(b,c), rather than using the temporal constraints. Therefore there are alternatives, and choosing one leads to an over-specific heuristic.

(2) the generalisation space for partial plans is non-trivial! Compared with early examples of generalisation spaces, such as LEX's spaces of algebraic expressions [Mitchell 83], or generalisation using Michalski's VL logic (in [Michalski et al

83]), a space of generalised partial plans would be quite complex to construct, search and manipulate.

Other forms of learning in plan space have been advocated: an interesting model of learning and problem solving is described (but not implemented!) in [Carbonell 83]; the thesis is as follows: a problem solver should initially use weak methods to solve problems, but then use past solutions to similar problems as a starting point for future problem solving. Problems are judged to be similar to past problems by a difference function, which depends on such features as goal conditions and initial problem states. When a solution to a past problem is used for a new problem, problem solving progresses through 'problem space' by incrementally adjusting the old solution until it is changed into a full solution to the new problem. Carbonell introduced a variety of problem space operators to accomplish this, including ones to insert and delete bits of the solution.
The incremental adjustment is then meant to improve using concept learning techniques, where successful adjustments are considered as positive examples, etc (as in the typical concept learning paradigm). Carbonell's work on this seems to have lead to later work on derivational analogy in PRODIGY (referred to in figure 5.1 below).

Now NLP can be used in three modes:

1-simple best first search through partial plan space;

2-search as in 1-, but using chunks or EBG-generated heuristics (as discussed in 4.3) to cut down search branching;

3-search as in 1-, but after chunks are created from operator solution sequences, they are used to suggest operator inclusion in later planning, as mentioned in 4.2 conclusion b.

Mode 3- is similar to Carbonell's line of research: rather than use chunks for 'generate then select' mode, examination of the new task (I,G,E,OS) can instantiate the left hand side of chunks (refer to the chunk form given in 3(5)) so that their right hand side can be used to compile an advanced partial plan from which to start problem solving, consisting of an initial operator set Os'.

An experimental implementation of this, however, showed up a serious problem: given an arbitrary goal P to solve, using the goal achievement algorithm, there is still a choice between using an operator from the partial solution to solve P or adding another operator - it is not necessarily true that an operator from the 'advanced partial solution' should solve P. This leads to more choices than in the basic search: choices in 4(1) and 4(2) do not go away, but increase because of the extra operators in the partial plan. We found that using an old plan which is 'close' to a new problem (as measured by some a priori rules) is generally no more efficient than heuristically improving search or even search from

first principles. The general conclusion is therefore to favour heuristic acquisition techniques that cut down search in the NLPs, and some of my future work will run along these lines.

5. CONCLUSIONS

5.1 General Conclusions

The work from which this thesis is compiled spanned a wide investigation of experience-based performance improvement in standard, general-problem-solving paradigms. It entailed designing and implementing FM, a large learning and planning system, which was used as a testbed for performance improvement techniques, and resulted in the construction of five types of heuristic acquisition components, respectively creating:

** the closed macro (chapter 2 and appendices D.1 and D.3)

** the basic chunk (chapter 2 and appendix D.2)

** the b-chunk (appendices D.2 and D.4)

** the c-chunk (chapter 3)

** the NLP heuristics (chapter 4)

The specific aim of this thesis was to investigate the hypothesis that a general planner could significantly improve it's efficiency through successful experience when presented with a domain specification, by acquiring domain dependent heuristics. Chapter 3 of this thesis describes a type of heuristic acquisition that supports our hypothesis, at least for a certain class of planners and application domains; consequently in the write-up emphasis has been placed on the automatic creation of heuristic control rules, formed with c-chunks, for goal directed linear systems. Without doubt this has been the more successful line of the research. Of the two main technique types, chunk creation and macro creation, the former proved most successful for a simple reason: chunks cut down search - that is they attack the central cause of a general problem solver's inefficiency. On the other hand closed macros are more generally applicable - they can be used in all three types of planner without change, whereas a different version of the chunk is needed for each type of search. On the whole, however, tests with FM involving macros, or using chunks to create initial partial solutions, bring with them their own combinatorial problems, and consequently make their general application difficult.

Next I will list what I believe to be the most important achievements of FM's c-chunks, which are supported by the experimental results in both the test domains of section 3.3. Problems encountered with this method were discussed at the end of that section, along with possible ways of overcoming them, so I will concentrate here on the positive results:

5.11: they generally show monotonicity in efficiency improvement,

and avoid falling into a similar trap that apparently happened to Macrops acquisition of [Fikes et al 72]: the system would become bogged down with combinatorics as more Macrops were formed, and spend more time in matching heuristic's preconditions than it would have done in problem solving from first principles (to paraphrase [Porter and Kibler 84, p.278]'s argument, following [Fikes et al 72]). Although some matching problems were inevitably encountered in chunking, this had little to do with the amount of training examples, but the inadequate propositional representation of a spatial domain (cf. section 3.33) ; indeed over the course of many training examples, fewer and fewer chunks are formed as experience builds up the heuristic rules and less choice points (which initiate chunking) are recorded during search.

5.12: the learned heuristics show a high degree of 'across-task transfer'. The tests' speedy marked improvement in problem solving efficiency is evidence of this, and is a result of the generality bias inherent in the strengthening algorithms. Recently I have encountered supporting research for generality bias away from strict EBL in the 'Progressive Refinement' learning techniques of [Van der Valde 88] (also see 5.33): for certain domains the author advocates forfeiting the correctness of EBL for over-generalisation and then forced specialisation (i.e. rule repair!).

5.13: the accumulation of chunks into heuristic rules is an incremental method, and is relatively noise-immune. This was tested in section 3.32 by the addition of some erroneous chunks before a batch of test problems were executed, and performance still approximated to the ordinary learning curve.

5.14: rule acquisition is relatively immune to problem ordering: experiments in both the domains of 3.3 show that useful rules are acquired when task lists are tried in either order; ideally, of course, simple problems should be given first.

While it is probably the case that a general problem solver which acquired heuristics for a particular domain could hardly reach the efficiency level of a special purpose domain specific knowledge-based system, much time and effort would be saved in handcrafting the rules; in fact the handcrafted rules designed for two sample domains were extremely laborious to construct. In the case of the warehouse domain they turned out to be erroneous and incomplete, and had to be de-bugged through testing (in a similar fashion to application domain construction).

In a sense, work on control rule acquisition is complementary to work on domain level, inductive rule acquisition, a good example of which is given in [Michalski and Chilausky 80]. There is also an analogous situation in the area of program transformation which is concerned with the derivation of an efficient program from a declarative program specification: handcrafting a program will produce a more efficient implementation than automatic transformation from the specification, although the latter is more desirable.

I have demonstrated through the experiments in 3.3 that the C-chunk method has been successful for a particular goal directed implementation, and the particular framework of '(I,G,E,OS)', but claim that it can be generalised to general problem solving systems bearing the following characteristics:

5.15: the overall search strategy must be goal directed, in which operator instantiations that achieve goals are used to extend search through a space of goal nodes in a best first manner;

5.16: the operator schemas themselves must be declaratively specified, and should be reasonably consistent models of some actions;

5.17: effective planning should be possible on simple problems using weak heuristics only.

5.15 ensures that the c-chunk strengthening method can be adopted, i.e. failed operator instantiations can be recorded and used for discrimination purposes, while 5.16 allows weakest preconditions to be constructed easily. Finally, Condition 5.17 is for 'bootstrapping' heuristic acquisition, in that the system must be able to solve simple problems from which to learn.

Part of the second condition has, however, been weakened for FM by the work of [Porteous 87]: during plan execution, if an operator could not be applied because of a mismatch between FM's beliefs and the 'real' world (which was modelled by some separate data structure), then replanning would take place (see reference for details). Some complementary work has been carried out on operator precondition repair in [Carbonell and Gil 87] and is shown as the 'Experimentation' module in figure 5/1; in standard FM, however, the task framework in assumed to be complete and consistent.

As well as failing point 5.15, both NLP and FOR flounder on 5.17. NLP was most disappointing: the combinatorics of temporal combinations and choices in constraint posting made it an order of magnitude slower than MEA.

5.2 Comparisons

5.21 Prodigy

As far as I can tell from the literature, there are few research groups which have attempted to create a domain independent learning and planning system that strengthens weak heuristics with acquired control knowledge through experience.

One such general planning system that learns by experience is the Prodidgy program referred to previously, created at CMU by various authors including Minton, Carbonell, Etzioni, Knoblock and Kuokka discussed in [Minton and Carbonell 87], and in [Minton el al 87]. The work on Prodigy was carried out in parallel with my own, the

major difference being in scale!

The top level architecture of the whole system is shown in figure 5/1 and is taken from [Carbonell 88]. Various types of knowledge acquisition components have been integrated into the one system: including a learning-by-experimentation module, which refines the system's domain information which had been incorrect or incomplete [Carbonell And Gil 87]; and it also contains a derivational analogy component, following Carbonell's earlier work on analogy [Carbonell 83]. It is obviously a large implementation, at the heart of which is a Strips-type general planner, not dissimilar to MEA.

The 'search control rule' acquisition sub-system is analogous to FM's chunk creation processes, and consists of three components: Explanation-based Learning to acquire the initial control rule, Compression to optimise it and Evaluation to monitor its usefulness. The main advantages of this sub-system are that EBL's 'target concepts' like 'operator-succeeds', 'operator fails' can be given 'declaratively' to the system via a set of rules, and that different modes of rule use can be set up - e.g. operator preference or rejection.

Learning from success is weak but is substantially improved by learning from failure - in fact the two together seem similar to FM's c-chunk creation (c-chunks use operator failure implicitly in the strengthening algorithms). As referred to in section 3.2, [Minton and Carbonell 87]'s example developed from a blocks world application produces a very specific chunk. It is reproduced below: it's informal meaning is 'to hold a block X which is on a block Y, unstack(X,Y) is the correct operator to add to the goal directed search if currently a block W is on X, the arm is empty and W is clear. Minton and Carbonell denote this rule as:

```
OP-SUCCEEDS(OP, G, NODE) if
MATCHES (OP, UNSTACK(X,Y)) &
MATCHES (G , HOLDING(X) )  &
KNOWN (NODE, ON(X,Y)&ON(W,X)&CLEAR(W)&ARMEMPTY )
```

In FM notation, this is equivalent to chunk:

```
ch( ch1, UNSTACK(X,Y), HOLDING(X),
  ON(X,Y)&ON(W,Z)&CLEAR(W)&ARMEMPTY, nil)
```

From the same solution sequence, FM would create a C-chunk thus:

```
ch( ch1, UNSTACK(X,Y), HOLDING(X),
  ON(X,Y)&ARMEMPTY, TYPE(X,block)&TYPE(Y,block)&NE(X,Y))
```

This rule is more general (containing only 2 conditions instead of 4) and nearer part of an optimal control rule, which would not

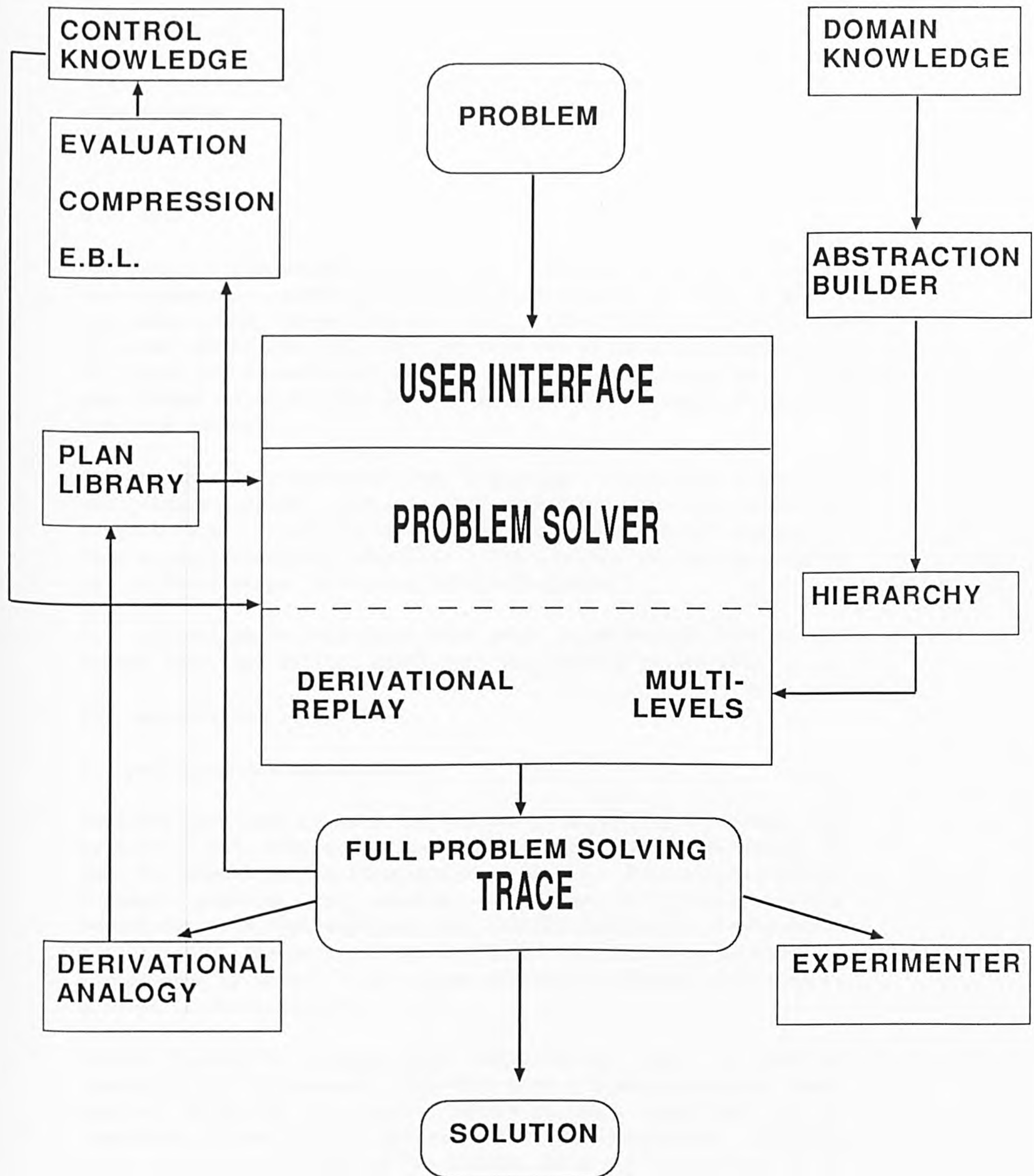

figure 5/1: the prodigy system

include the irrelevant fact 'ARMEMPTY'.

Unlike FM, Prodigy creates 'correct' rules relying on strict EBL and does not incrementally refine or repair them by any generalisation/specialisation techniques. As shown in the figure, it does optimise rules, in fact in a similar way to FM. Prodigy also monitors the amount of CPU time they save, deleting them if they prove not to be effective (this is a little puzzling since effectiveness could only be judged if the problem solver searches exhaustively trying all paths in an ordinary problem solving session).

5.22 Soar

The Soar problem solving program has similarities to FM in that it uses experience learning to improve performance. In fact my use of the term 'chunk' arose from this work. Like Prodigy, this system was also created at CMU, arising from the doctoral dissertations of Laird and Rosenbloom, and was developed by a large team (e.g see [Steier et al 87] for latest developments, a paper which has has nine authors!).

Soar has one learning mechanism: 'chunking'. Chunks are created by analysing successful solutions, but rather than just for heuristic control rules, they are created for 'decisions' in all aspects of Soar's problem solving behaviour. The creation process is carried out in three steps (following [Laird et al 86]):

(1) collect conditions which were used in processing before the solved goal, and actions which were the 'result of the goal'.

(2) variable-ise identifiers;

(3) perform chunk optimisation.

Soar has been used in many applications, including the usual toy problems, but also non-trivial applications such as speeding up the 'R1' expert system [Rosenbloom et al 85]. When applied to the 8-puzzle problem, for example, it seems to produce similar behaviour to FM when equipped with the FOR search and closed macro acquisition: 'macro1' created by FM and reproduced in section 2.7 corresponds to Soar's 'with-column symmetry transfer' explained on p.39 of [Laird et al 86].

Unlike Prodigy's control rule acquisition, Soar is not so theoretically transparent, but does have a clear advantage: Soar creates decisions for every choice it must make and it is therefore claimed to be a universal learning mechanism, chunking as it does on all levels of the system. From the literature, it is difficult to extract exactly what the specification language for Soar applications is, and in what form the chunks appear. To quote [Laird et al 86], p.31: "When the problem solver knows what its doing, everything works fine, ..." !.

5.23 Progressive Refinement

Another incremental, evolutionary approach to learning in problem solving is the Progressive Refinement of [Van der Velde 88]. Here the acquisition of 'heuristic associations' is described, the main application being second generation diagnostic expert systems, rather than planning domains.

He divides previous similar work into learning control knowledge and learning shortcuts, putting his work firmly into the latter category. Although only FM's closed macros fall into his 'short-cut' category, there are similarities with my research. The system learns from problem and solution pairs, relying on background knowledge of the domain, and makes deliberate over-generalisations of learned heuristics (especially for domains which he classes as non-critical and non-diverse), then forces specialisation in the face of incorrectness. Similarly, my C-chunk algorithm often produces over-general chunks, which are simpler, leaner and practically correct, but of course may be subsequently specialised in the rule repair module.

5.3 Future Directions

The primary future objective is to evolve the control rule acquisition mechanism specified in chapter 4 into an efficient general representation that will increase the efficiency of NLP while preserving its advantages of non-linearity. The main snag in using strict E.B.G. in partial plan space is that the acquired heuristics are too specific and detailed to allow efficient matching when they are in use.

I would also like to use FM with MEA in a large embedded domain, perhaps in a real robot manipulation task area, but before this can be accomplished, the c-chunking method needs to be perfected, along the lines discussed in the conclusions of section 3.3. In the limit c-chunking would most certainly require an automatic representation change facility for the domain definition, when for example, chains of relations in target concepts become unwieldly. Indeed I would go as far as to say that the success of FM with c-chunks in those domains to which it has been applied is due mainly to the fact that target concept approximations can be formed relatively easily in the predicate language defined by the user. This is reminiscent of Lenat's conclusions in his re-analysis of his AM discovery system [Lenat and Brown 83]: he admitted that a major reason why it had worked well was that the Lisp representations were 'close' to the mathematical world in which AM was exploring!

Since FM's chunk is not a 'universal learning' mechanism, we also need heuristic acquisition for other aspects of the system that involve decisions e.g. decisions about which linear ordering to

choose to solve a conjunction of goal predicates.

Finally I will elaborate on future work needed to improve the performance components. The main task when using FM is in developing a consistent domain specification. This is a non-trivial task, and in future could be alleviated by work in two directions:

* automatic tools for specification development and checking;

* automatic domain knowledge acquisition components, where FM is embedded in an outside environment and receives feedback.

The first direction would have helped the construction of the two test domains considerably, whereas 'debugging' the operator schemas, for instance, had to be done by trial and error. Specifically, procedures that perform the following functions are needed:

* to check operators leave state descriptions in a consistent form with respect to domain facts and rules, after their application; for example, consider the simple structuring rule in the warehouse domain:

```
"for all Objects, Positions
          on_floor(Object,Position) -> at(Object,Position)"
```

Any operators that change an Objects position, and fail to change both the facts in this rule, would be spotted as inconsistent by the check procedures, using this rule as an axiom.

* to check domain facts I.f and E.f and the domain rules I.r and E.r are consistent with each other.

For MEA and NLP to cope with more complex domains, the addition of goal hierarchy and operator abstraction levels is necessary. In a sense this is simply to add another (powerful) 'weak heuristic', and should complement control rule creation. As mentioned in chapter 4, my non-linear planner has already been expanded to the hierarchical planner 'HNLP' described in [Fox 88], and work is in progress to add chunking components to this. The Prodigy system discussed above in 5.2, and shown in figure 5/1, has likewise been extended to allow hierarchically structured domains, but is still tied to a linear Strips-type planner [Carbonell 88]. Similarly, other weak heuristics, such as the addition of rules to detect inconsistent goal conjunctions, complement my approach: the true 'Heuristic-learning, Problem Solving Shell' will be formed by combining weak heuristics for learning with traditional goal oriented and hierarchical weak methods. Extending the expressive power of the FM framework is another possible future direction, and integrating planners that break out of the 'Strips assumptions', with learning components, yet another.

APPENDIX A.1: THE ROBOT WORLD

```
frame(
        name: robot_world01,
        type: context,
        /* E.f :-- */
        always:
        type_of(room1,room)&type_of(room2,room)
        &type_of(room3,room)&type_of(room4,room)
        &type_of(room5,room)&type_of(room6,room)
        &type_of(room7,room)&type_of(door23,door)
        &type_of(door24,door)&type_of(door25,door)
        &type_of(door35,door)&type_of(door45,door)
        &type_of(door56,door)&type_of(door12,door)
        &type_of(door47,door)&type_of(door57,door)
        &type_of(door67,door)&type_of(box2,box)
        &type_of(big_box,box)&type_of(box1,box)
        &connect(room2,room3,door23)&connect(room3,room2,door23)
        &connect(room2,room4,door24)&connect(room4,room2,door24)
        &connect(room2,room5,door25)&connect(room5,room2,door25)
        &connect(room6,room5,door56)&connect(room5,room6,door56)
        &connect(room5,room3,door35)&connect(room3,room5,door35)
        &connect(room5,room4,door45)&connect(room4,room5,door45)
        &connect(room4,room7,door47)&connect(room7,room4,door47)
        &connect(room5,room7,door57)&connect(room7,room5,door57)
        &connect(room6,room7,door67)&connect(room7,room6,door67)
        &connect(room1,room2,door12)&connect(room2,room1,door12)
        &fits_thru(box1,door24)&fits_thru(box1,door23)
        &fits_thru(box1,door25)&fits_thru(box1,door56)
        &fits_thru(box1,door45)&fits_thru(box1,door35)
        &fits_thru(box1,door47)&fits_thru(box1,door57)
        &fits_thru(box1,door12)&fits_thru(box1,door67)
        &fits_thru(box2,door24)&fits_thru(box2,door23)
        &fits_thru(box2,door25)&fits_thru(box2,door56)
        &fits_thru(box2,door45)&fits_thru(box2,door35)
        &fits_thru(box2,door47)&fits_thru(box2,door57)
        &fits_thru(box2,door12)&fits_thru(box2,door67)
        &fits_thru(big_box,door24)
        &fits_thru(big_box,door25)&fits_thru(big_box,door56)
        &fits_thru(big_box,door35)
        &fits_thru(big_box,door47)&fits_thru(big_box,door57)
        &fits_thru(big_box,door12)&fits_thru(big_box,door67),
        /* S.r :-- */
        axioms: [at_door(O,D,R),in_room(O,R),
                 next_to(O1,O2)&in_room(O1,R),in_room(O2,R),
                 next_to(O2,O1)&in_room(O1,R),in_room(O2,R) ] ).
        /* E.r :-- */
env_axioms([    [connect(X,Y,_), ne(X,Y)],
                [connect(_,Y5,Z5), ne(Y5,Z5)],
                [connect(X4,_,Z4), ne(X4,Z4)],
                [connect(X1,Y1,Z1), connect(Y1,X1,Z1)],
                [connect(_,Y2,_), type_of(Y2,room)],
                [connect(_,_,Z3), type_of(Z3,door)]]
        ).
```

```
/* initial world for list1 */
init_world(
    in_room(box1,room4)&in_room(box2,room4)&
    in_room(big_box,room1)&in_room(robot,room6)&
    closed(door67)&open(door47)&open(door57)&closed(door45)&
    closed(door56)&open(door35)&closed(door12)&
    open(door23)&open(door25)&open(door24)
    ).

/* initial world for list2 */
init_world(
    in_room(box1,room2)&in_room(box2,room2)&in_room(big_box,room4)&
    in_room(robot,room6)&open(door67)&open(door47)&
    open(door57)&open(door45)&open(door56)&
    open(door35)&open(door12)&
    open(door23)&open(door25)&open(door24)
    ).


/* LIST OF OPERATORS (OS) */

frame(
        name: gothrudoor(D,R),
        type: operator,
        filter: nil,
        check:  connect(R1,R,D),
        precon: at_door(robot,D,R1)&
                open(D) ,
        padd:   in_room(robot,R),
        add:    next_to(robot,D),
        delete: at_door(robot,_,_)
                &next_to(robot,_)
                &in_room(robot,_)
        ).



frame(
        name: gotodoor(D,R),
        type: operator,
        filter: nil,
        check:  connect(R,_,D),
        precon: in_room(robot,R),
        padd:    at_door(robot,D,R),
        add:    nil,
        delete: at_door(robot,_,_)
                &next_to(robot,_)
        ).
```

```
frame(
        name: pushthrudoor(Ob,D,To_room),
        type: operator,
        filter: nil,
        check:  type_of(Ob,box)&
                fits_thru(Ob,D)&
                connect(R,To_room,D),
        precon: at_door(Ob,D,R)&
                in_room(Ob,R)&
                next_to(robot,Ob)&
                open(D),
        padd:   in_room(Ob,To_room),
        add:    in_room(robot,To_room)&
                next_to(robot,Ob),
        delete: at_door(robot,_,_)
                &at_door(Ob,_,_)
                &next_to(robot,_)
                &next_to(Ob,_)
                &next_to(_,Ob)
                &in_room(robot,_)
                &in_room(Ob,_)
        ).


frame(
        name: pushtodoor(Ob,D,R),
        type: operator,
        filter: nil,
        check:  type_of(Ob,box)&
                connect(R,_,D),
        precon:
                in_room(Ob,R)&
                next_to(robot,Ob),
        padd:   at_door(Ob,D,R),
        add:    next_to(robot,Ob),
        delete: at_door(Ob,_,_)
                &next_to(robot,_)
                &next_to(Ob,_)
                &next_to(_,Ob)
        ).

frame(
        name: open(D),
        type: operator,
        filter: nil,
        check:  type_of(D,door)&connect(R,_,D),
        precon: closed(D)&
                at_door(robot,D,R),
        padd:   open(D),
        add:    nil,
        delete: closed(D)
                &next_to(robot,_)
        ).
```

```
frame(
        name: pushtobox(Ob1,Ob2),
        type: operator,
        filter: nil,
        check:  type_of(Ob1,box)&
                type_of(Ob2,box)&
                ne(Ob1,Ob2),
        precon:
                next_to(robot,Ob1)&
                in_room(Ob2,R)&
                in_room(Ob1,R),
        padd:   next_to(Ob1,Ob2),
        add:    next_to(robot,Ob1),
        delete: at_door(robot,_,_)
                &at_door(Ob1,_,_)
                &next_to(robot,_)
                &next_to(Ob1,_)
                &next_to(_,Ob1)
        ).


frame(
        name: goto(X),
        type: operator,
        filter: nil,
        check:  type_of(X,box),
        precon: in_room(X,R)&
                in_room(robot,R),
        padd:   next_to(robot,X),
        add:    nil,
        delete: at_door(robot,_,_)
                &next_to(robot,_)
        ).
frame(
        name: close(D),
        type: operator,
        filter: nil,
        check:  type_of(D,door)&connect(R,_,D),
        precon: open(D)&
                at_door(robot,D,R),
        padd:   closed(D),
        add:    nil,
        delete: open(D)
                &next_to(robot,_)
        ).
```

APPENDIX A.2 THE WAREHOUSE WORLD

```
/*  ************ warehouse environment ***************** */

frame(
        name: warehouse,
        type: context,
        /* E.f:--   */
        always:
                type_of(crate1,crate)
                &type_of(crate2,crate)
                &type_of(crate3,crate)
                &type_of(s1,space)
                &type_of(s2,space)&type_of(s3,space)
                &type_of(s4,space)&type_of(shelf6,shelf)
                &type_of(s6,space)&type_of(s7,space)
                &type_of(shelf7,shelf)
                &type_of(s8,space)&type_of(s9,space)
                &type_of(shelf9,shelf)
                &type_of(s10,space)&type_of(shelf10,shelf)
                &type_of(s12,space)&type_of(s13,space)
                &type_of(s14,space)&type_of(shelf14,shelf)
                &type_of(s16,space)
                &pickup_point(s6)&
                &pickup_point(s9)
                &pickup_point(s10)
                &pickup_point(s14)

                &connect(s6,shelf6)&connect(shelf6,s6)
                &connect(s10,s6)&connect(s6,s10)
                &connect(s10,s9)&connect(s9,s10)
                &connect(s9,shelf9)&connect(shelf9,s9)
                &connect(s14,s10)&connect(s10,s14)
                &connect(s10,shelf10)&connect(shelf10,s10)
                &connect(s14,shelf14)&connect(shelf14,s14)

                &next(s1,s2)&next(s2,s1)
                &next(s2,s3)&next(s3,s2)
                &next(s4,s3)&next(s3,s4)
                &next(s6,s7)&next(s7,s6)
                &next(s8,s7)&next(s7,s8)
                &next(s9,s10)&next(s10,s9)
                &next(s14,s13)&next(s13,s14)
                &next(s13,s9)&next(s9,s13)
                &next(s6,s2)&next(s2,s6)
                &next(s6,s10)&next(s10,s6)
                &next(s14,s10)&next(s10,s14)
                &next(s3,s7)&next(s7,s3)
                &next(s8,s4)&next(s4,s8)
                &next(s8,s12)&next(s12,s8)
                &next(s16,s12)&next(s12,s16)
                &between(s7,s6,shelf6)&between(s3,s7,shelf7)
                &between(s13,s9,shelf9)&between(s13,s14,shelf14)
                &between(s9,s10,shelf10),
```

```
        /* S.r :-- */
        axioms: [loaded(truck,B,Y),at(B,Y),
                 on_floor(X1,Y1),at(X1,Y1),
                 loaded(crane,Y2,Z2),at(crane,Z2),
                 loaded(crane,Y6,Z6),above_floor(Y6,Z6),
                 stacked(X3,Y3),at(X3,Y3),
                 in_truck&at(truck,Y5),at(driver,Y5) ]

        ).

        /* E.r :-- */
env_axioms([    [next(X,Y), ne(X,Y)],
                [next(X3,Y3), next(Y3,X3)],
                [between(X1,X2,_),next(X1,X2)],
                [connect(X4,X5), connect(X5,X4) ]  ]

        ).


/*  an initial state for the warehouse world */

init_world(
                unloaded(truck)&
                unloaded(crane)&
                on_floor(driver,s13)&
                on_floor(crate2,s12)&
                on_floor(crate3,s16)&
                on_floor(crate1,s1)&
                at(crate1,s1)&
                clear(s10)&
                clear(s3)&
                at(driver,s13)&
                clear(s4)&
                clear(s5)&
                clear(s7)&
                at(truck,s2)&
                at(crate2,s12)&
                at(crate3,s16)&
                clear(s14)&
                clear(s6)&
                clear(s11)&
                clear(s8)&
                clear(s9)&
                at(crane,s14)&
                clear(s15)
).
```

```
/* LIST OF OPERATORS (OS) */


/*1*/frame(
        name:   load(B,X,Y),
        type:   operator,
        filter: at(B,X),
        check:  type_of(B,crate)&
                next(X,Y)&
                type_of(X,space)&
                type_of(Y,space),
        precon:
                in_truck&
                on_floor(B,X)&
                unloaded(truck)&
                at(truck,Y),
        padd:   at(B,Y)&loaded(truck,B,Y)&clear(X),
        add:    nil,
        delete: on_floor(B,X)&
                at(B,X)&
                unloaded(truck)
        ).


/*2*/frame(
        name:   unload(B,Y,X),
        type:   operator,
        filter: nil,
        check:  type_of(B,crate)&
                next(X,Y)&
                type_of(X,space)&
                type_of(Y,space),
        precon:
                clear(X)&
                in_truck&
                loaded(truck,B,Y),
        padd:   unloaded(truck)&on_floor(B,X)&at(B,X),
        add:    nil,
        delete: loaded(truck,B,Y)&
                clear(X)&
                on_floor(B,Y)&
                at(B,Y)
        ).
```

```
/*3*/frame(
        name:   get_in(X,Y),
        type:   operator,
        filter: at(truck,Y),
        check:
                type_of(X,space)&
                type_of(Y,space)&
                next(X,Y),
        precon:
                at(driver,X)&
                on_floor(driver,X)&
                at(truck,Y),
        padd:   in_truck,
        add:    clear(X)&at(driver,Y),
        delete: on_floor(driver,X)&
                at(driver,X)
        ).
```

```
/*4*/frame(
        name:   get_out(Y,X),
        type:   operator,
        filter: nil,
        check:
                type_of(X,space)&
                type_of(Y,space)&
                next(X,Y),
        precon:
                in_truck&
                clear(X)&
                at(truck,Y),
        padd:   on_floor(driver,X),
        add:    clear(X)&at(driver,X),
        delete: in_truck&
                clear(X)&
                at(driver,Y)
        ).
```

```
/*5*/frame(
       name:   drive_load(B,X,Y),
       type:   operator,
       filter: nil,
       check:
               next(X,Y)&
               type_of(X,space)&
               type_of(Y,space)&
               type_of(B,crate),
       precon:
               in_truck&
               loaded(truck,B,X)&
               clear(Y),
       padd:   loaded(truck,B,Y)&at(B,Y),
       add:    at(driver,Y)&clear(X)&at(truck,Y),
       delete:
               at(driver,X)&
               loaded(truck,B,X)&
               at(B,X)&
               at(truck,X)&
               clear(Y)
       ).
```

```
/*6*/frame(
       name:   drive(X,Y),
       type:   operator,
       filter: nil,
       check:
               type_of(X,space)&
               type_of(Y,space)&
               next(X,Y),
       precon:
               in_truck&
               unloaded(truck)&
               at(truck,X)&
               clear(Y),
       padd:   at(truck,Y)&clear(X),
       add:    at(driver,Y),
       delete:
               at(driver,X)&
               at(truck,X)&
               clear(Y)
       ).
```

```
/*7*/frame(
        name: walk(X,Y),
        type: operator,
        filter: nil,
        check:  next(X,Y)&
                type_of(X,space)&
                type_of(Y,space),
        precon: at(driver,X)
                &on_floor(driver,X)&
                clear(Y),
        padd:   at(driver,Y),
        add:    on_floor(driver,Y)&clear(X),
        delete:
                at(driver,X)&on_floor(driver,X)&
                clear(Y)

        ).

/*8*/frame(
        name:   lift_up(B,Y),
        type:   operator,
        filter: nil,
        check:
                pickup_point(Y)&
                type_of(B,crate),
        precon:
                unloaded(crane)&
                at(crane,Y)&
                on_floor(B,Y),
        padd:   above_floor(B,Y)&loaded(crane,B,Y),
        add:    nil,
        delete: unloaded(crane)&
                on_floor(B,Y)
        ).


/*9*/frame(
        name:   lift_down(B),
        type:   operator,
        filter: above_floor(B,Y)&at(B,Y),
        check:
                type_of(B,crate),
        precon:
                above_floor(B,Y)&
                loaded(crane,B,Y),
        padd:   on_floor(B,Y)&unloaded(crane),
        add:    nil,
        delete: loaded(crane,B,Y)&
                above_floor(B,Y)
        ).
```

```
/*10*/frame(
        name:   move_crane(X,Y),
        type:   operator,
        filter: nil,
        check:
                pickup_point(X)&
                pickup_point(Y)&
                connect(X,Y),
        precon:
                at(crane,X)&
                unloaded(crane),
        padd:   at(crane,Y),
        add:    nil,
        delete: at(crane,X)
        ).


/*11*/frame(
        name:   crane_stack(B,X,S),
        type:   operator,
        filter: nil,
        check:
                pickup_point(X)&
                connect(X,S)&
                type_of(S,shelf)&
                type_of(B,crate),
        precon:
                loaded(crane,B,X),
        padd:   stacked(B,S)&unloaded(crane),
        add:    nil,
        delete: loaded(crane,B,X)
        ).


/*12*/frame(
        name:   crane_unstack(B,Y,S),
        type:   operator,
        filter: stacked(B,S)&above_floor(B,Y),
        check:
                connect(S,Y)&
                type_of(S,shelf)&
                type_of(B,crate),
        precon:
                unloaded(crane)&
                stacked(B,S)&
                at(crane,Y),
        padd:   loaded(crane,B,Y),
        add:    nil,
        delete: unloaded(crane)&
                stacked(B,S)
        ).
```

```
/*13*/frame(
        name: truck_stack(B,X,Y,Z),
        type: operator,
        filter: nil,
        check:
                between(X,Y,Z)&
                type_of(X,space)&
                type_of(Y,space)&
                type_of(Z,shelf)&
                type_of(B,crate),
        precon:
                at(truck,X)&
                unloaded(truck)&
                in_truck&
                on_floor(B,Y),
        padd:   stacked(B,Z),
        add:    above_floor(B,Y),
        delete:
                on_floor(B,Y)
        ).


/*14*/frame(
        name: truck_unstack(B,X,Y,Z),
        type: operator,
        filter: stacked(B,Z),
        check:
                between(X,Y,Z)&
                type_of(X,space)&
                type_of(Y,space)&
                type_of(Z,shelf)&
                type_of(B,crate),
        precon:
                at(truck,X)&
                unloaded(truck)&
                in_truck&
                stacked(B,Z),
        padd:   on_floor(B,Y),
        add:    nil,
        delete:
                stacked(B,Z)&
                above_floor(B,Y)
        ).
```

APPENDIX A.3 THE 8-PUZZLE

```
frame(
        name: eight_puzzle,
        type: context,

        always:
                next(p1,p2)
                &next(p2,p1)
                &next(p1,p4)
                &next(p4,p1)
                &next(p5,p2)
                &next(p2,p5)
                &next(p3,p2)
                &next(p2,p3)
                &next(p3,p6)
                &next(p6,p3)
                &next(p6,p5)
                &next(p5,p6)
                &next(p6,p9)
                &next(p9,p6)
                &next(p9,p8)
                &next(p8,p9)
                &next(p8,p5)
                &next(p5,p8)
                &next(p7,p8)
                &next(p8,p7)
                &next(p4,p7)
                &next(p7,p4)
                &next(p4,p5)
                &next(p5,p4)
                &type_of(tile1,tile)
                &type_of(tile2,tile)
                &type_of(tile3,tile)
                &type_of(tile4,tile)
                &type_of(tile6,tile)
                &type_of(tile7,tile)
                &type_of(tile8,tile)
                &type_of(tile5,tile)
                &type_of(p1,p)
                &type_of(p2,p)
                &type_of(p3,p)
                &type_of(p4,p)
                &type_of(p5,p)
                &type_of(p6,p)
                &type_of(p7,p)
                &type_of(p8,p)
                &type_of(p9,p)
                ,
      axioms : []
      ).
```

```
/* this info is needed for efficient forward search */

inverse( move(X,Y,Z), move(X,Z,Y) ).


init_world(
            at(tile3,p3)&
            at(tile2,p2)&
            at(tile6,p1)&
            at(tile1,p4)&
            at(tile7,p8)&
            at(tile5,p9)&
            at(tile4,p6)&
            at(blank,p5)&
            at(tile8,p7)
            ).

/* OPERATOR (OS) */

frame(
        name: move(T,S,D),
        type: operator,
        filter: nil,
        check:  next(S,D)&ne(T,blank),
        precon: at(T,S)&at(blank,D),
        padd:   at(T,D)&at(blank,S),
        add:    nil,
        delete: at(T,S)&at(blank,D)
        ).
```

APPENDIX A.4 A BLOCKS WORLD

```
frame(
        name: blocks1,
        type: context,
        always:
                type_of(a,box)
                &type_of(b,box)
                &type_of(c,box)
                &type_of(d,box)
                &type_of(e,box)
        axioms: []).


init_world(
        on(a,table)& on(d,table)&
        on(e,table)& on(c,a)&
        on(b,table)&clear(c)&
        clear(e)&clear(d)&
        clear(b)
            ).

/* LIST OF OPERATORS (OS) */

frame(
        name:   puton(Ob1,Ob2),
        type:   operator,
        macrop: [],
        check:  ne(Ob1,Ob3)&ne(Ob2,Ob3)&
                ne(Ob1,Ob2)&type_of(Ob1,box)&
                type_of(Ob2,box),
        precon: on(Ob1,Ob3)&clear(Ob1)&
                clear(Ob2),
        padd:   on(Ob1,Ob2),
        add:    clear(Ob3),
        delete: on(Ob1,Ob3)&clear(Ob2)
        ).


frame(
        name:   newtower(Ob1,Ob2),
        type:   operator,
        macrop: [],
        check:  type_of(Ob2,box)&type_of(Ob1,box)&
                ne(Ob1,Ob2),
        precon: on(Ob1,Ob2)&clear(Ob1),
        padd:   clear(Ob2),
        add:    on(Ob1,table),
        delete: on(Ob1,Ob2)
        ).
```

APPENDIX A.5 THE TOWER OF HANOI PUZZLE

```
frame(
        name: toh1,
        type: context,

        always:
                type_of(p1,pole)
                &type_of(p2,pole)
                &type_of(p3,pole)
                &type_of(d1,disc)
                &type_of(d2,disc)
                &type_of(d3,disc)
                &type_of(d4,disc)
                &type_of(base1,base)
                &type_of(base2,base)
                &type_of(base3,base)
                &smaller(d1,base1)
                &smaller(d1,base2)
                &smaller(d1,base3)
                &smaller(d2,base1)
                &smaller(d2,base2)
                &smaller(d2,base3)
                &smaller(d3,base1)
                &smaller(d3,base2)
                &smaller(d3,base3)
                &smaller(d4,base1)
                &smaller(d4,base2)
                &smaller(d4,base3)
                &smaller(d1,d2)
                &smaller(d1,d4)
                &smaller(d2,d4)
                &smaller(d3,d4)
                &smaller(d2,d3)
                &smaller(d1,d3),

        axioms: [ ontop(X,Y)&on(Y,P),on(X,P),
                  ontop(X,Y)&on(X,P),on(Y,P) ]

        ).

inverse( move1(D,P1,P2),move1(D,P2,P1) ). /*  move2 inverse not specified*/

init_world(     top(d1)& top(base2)&
                on(base1,p1)&top(base3)&
                on(base2,p2)&on(base3,p3)&
                on(d1,p1)&on(d2,p1)&
                on(d3,p1)&on(d4,p1)&
                ontop(d1,d2)&
                ontop(d2,d3)&
                ontop(d3,d4)&
                ontop(d4,base1) ).
```

```
/* OPERATOR (OS) */

 frame(
        name:   move1(D,Ob1,Ob2,P1,P2),
        type:   operator,
        filter: [],
        check:  type_of(P1,pole)&
                type_of(P2,pole)&
                type_of(D,disc)&
                smaller(D,Ob1)&
                ne(P1,P2),
        precon:
                top(D)&
                top(Ob1)&
                on(D,P1)&
                ontop(D,Ob2)&
                on(Ob1,P2),
        padd:   on(D,P2),
        add:    top(Ob2)&
                ontop(D,Ob1),
        delete: on(D,P1)&
                ontop(D,Ob2)&
                top(Ob1)
        ).
```

APPENDIX A.6 MACBETH WORLD

```
frame(
        name: story_of_macbeth,
        type: context,

        always:
                type_of(macduff,person)
                &type_of(macbeth_lady,person)
                &type_of(macbeth,person)
                &type_of(duncan,person)
                &type_of(a_dagger,weapon)
                &type_of(a_dagger,object)
                &is_strong(macbeth)
                &is_evil(macbeth_lady)
                &is_strong(duncan)
                &is_strong(macduff)
                &knows(macbeth,macduff)
                &knows(macbeth,duncan)
                &knows(macbeth_lady,macduff)
                &knows(macduff,duncan)
                &knows(macbeth_lady,duncan)
                &married(macbeth_lady,macbeth)
                &can_influence(macbeth_lady,macbeth)
                                                 ,
        axioms: [ ]
        ).

init_world(
            has(macbeth_lady,a_dagger)&
            has_motive(macbeth_lady)&
            nearby(macbeth,duncan)& nearby(macduff,duncan)&
            alive(macbeth)&alive(macduff)&
            alive(duncan)& alive(macbeth_lady) ).

/* LIST OF OPERATORS (OS) */

frame(
        name: kill(Killer,Inst,Killed),
        type: operator,
        filter: [],
        check:  type_of(Killer,person)&
                type_of(Killed,person)&
                is_strong(Killer)&
                type_of(Inst,weapon)&
                ne(Killer,Killed),
        precon: has(Killer,Inst)&
                next_to(Killer,Killed)&
                has_motive(Killer),
        padd:   dead(Killed),
        add:    murderer(Killer),
        delete: alive(Killed)  ).
```

```
frame(
        name: give_motive(Accomplice,Killer),
        type: operator,
        filter: [],
        check:  type_of(Killer,person)&
                type_of(Accomplice,person)&
                type_of(Weapon,weapon)&
                is_evil(Accomplice)&
                can_influence(Accomplice,Killer),
        precon: has_motive(Accomplice),
        padd:   has_motive(Killer),
        add:    needs(Killer,Weapon),
        delete: nil
        ).


frame(
        name: meet(P1,P2),
        type: operator,
        filter: [],
        check:  type_of(P1,person)&
                type_of(P2,person),
        precon:
                nearby(P1,P2),
        padd:   next_to(P1,P2),
        add:    nil,
        delete: nil
        ).


frame(
        name: give(Giver,Obj,Given),
        type: operator,
        filter: [],
        check:  type_of(Giver,person)&
                type_of(Giver,person)&
                type_of(Obj,weapon),
        precon: has(Giver,Obj)&needs(Given,Obj),
        padd:   has(Given,Obj),
        add:    nil,
        delete: has(Giver,Obj)
        ).
```

APPENDIX A.7: BOX WORLD

```
/* This operators are equivalent to Minton's IJCAI 87 (p229) ones*/


frame(
        name: blocks_world1,
        type: context,
        always:
                type_of(box2,box)
                &type_of(box5,box)
                &type_of(box4,box)
                &type_of(box1,box)
                &type_of(box3,box),
        axioms: []
        ).

inverse(  pickofffloor(X),putonfloor(X) ).
inverse(  putonfloor(X),pickofffloor(X) ).
inverse(  pickoffbox(X,Y),putonbox(X,Y) ).
inverse(  putonbox(X,Y),pickoffbox(X,Y) ).

env_axioms([]).

/* init world for blocks */

init_world(
handempty&
onfloor(box3)&
onfloor(box4)&
onfloor(box5)&
ontop(box1,box2)&
ontop(box2,box3)&
clear(box4)&
clear(box5)&
clear(box1)
          ).

frame(
       name:   putonbox(Ob1,Ob2),
       type:   operator,
       macrop: [],
       check:
               type_of(Ob1,box)&
               type_of(Ob2,box)&ne(Ob1,Ob2),
       precon: clear(Ob2)&
               holding(Ob1),
       padd:   ontop(Ob1,Ob2),
       add:    handempty&clear(Ob1),
       delete: clear(Ob2)&
               holding(Ob1)
       ).
```

```
frame(
        name:   pickoffbox(Ob1,Ob2),
        type:   operator,
        macrop: [],
        check:
                type_of(Ob1,box)&
                type_of(Ob2,box)&ne(Ob1,Ob2),
        precon: clear(Ob1)&
                handempty&
                ontop(Ob1,Ob2),
        padd:   clear(Ob2)&holding(Ob1),
        add:    nil,
        delete: clear(Ob1)&
                handempty&
                ontop(Ob1,Ob2)
        ).

frame(
        name:   pickofffloor(Ob1),
        type:   operator,
        macrop: [],
        check:  type_of(Ob1,box),
        precon: clear(Ob1)&
                handempty&
                onfloor(Ob1),
        padd:   holding(Ob1),
        add:    nil,
        delete: clear(Ob1)&
                handempty&
                onfloor(Ob1)
        ).

frame(
        name:   putonfloor(Ob1),
        type:   operator,
        macrop: [],
        check:  type_of(Ob1,box),
        precon: holding(Ob1),
        padd:   handempty,
        add:    clear(Ob1)&onfloor(Ob1),
        delete: holding(Ob1)
        ).
```

APPENDIX A.8 NLP BLOCKS WORLD
(This is the representation used for NLP with EBL,
and below it is given the procedure that changes from FM
representation to this one.)

```
env(  [        type_of(a,box)
                 ,type_of(b,box)
                 ,type_of(c,box)
                 ,type_of(d,box)
                 ,type_of(e,box)]     ).


/* init world for blocks */

init_world(
       [ on(a,table),  on(c,a),
        on(b,table),   clear(c),    clear(b)]     ).


operator(puton(_125545,_125560),[ne(_125545,_125641),
ne(_125560,_125641),ne(_125545,_125560),
type_of(_125545,box),type_of(_125560,box)],
[on(_125545,_125641),clear(_125545),clear(_125560)],
[on(_125545,_125560),clear(_125641)],
[on(_125545,_125641),clear(_125560)]).


operator(newtower(_126467,_126482),
     [type_of(_126482,box),type_of(_126467,box),
      ne(_126467,_126482)],
     [on(_126467,_126482),clear(_126467)],
     [clear(_126482),on(_126467,table)],
     [on(_126467,_126482)]).
```

```
/* procedure for changing rep's */


:- op(700,xfx,':').
:- op(100,xfy,'&').

change(Infile, Outfile) :-
        see(Infile),
        read(
                frame(  name: N,
                        type: operator,
                        macrop: _,
                        check: EnvA,
                        precon:PreA,
                        padd: AA,
                        add: AddA,
                        delete: DelA) ),
        andtolist(EnvA,Env),
        andtolist(PreA,Pre),
        andtolist(AA,A),
        andtolist(AddA,Add),
        andtolist(DelA,Del),
        append(A,Add,Added),
        tell(Outfile),
        write( operator( N,
                Env,
                Pre,
                Added,
                Del)),
        write('.'),nl,change(Infile,Outfile).
change(_, Outfile) :- tell(Outfile),told.

/*****************************************************************/

andtolist(nil,[]) :- !.
andtolist(X&Y,[X|Z]) :-!, andtolist(Y,Z).
andtolist(X,[X]) :- !.

append([],L,L):-!.
append([H|T],L,[H|Z]) :- append(T,L,Z),!.
```

```
/* APPENDIX B.1 */
/* Results of list 1 with chunking, algorithm B */

no. of expanded nodes: 20
taskfile: list12 taskno: task1 of length 5
goal: in_room(robot,room2) by: [gotodoor(door56,room6),open(door56),
gothrudoor(door56,room5),gotodoor(door25,room5),gothrudoor(door25,room2)]

CPUsed=65 secs

no. of expanded nodes: 5
taskfile: list12 taskno: task2 of length 5
goal: in_room(box1,room2) by: [gotodoor(door24,room2),
gothrudoor(door24,room4),goto(box1),pushtodoor(box1,door24,room4),
pushthrudoor(box1,door24,room2)]

CPUsed=28 secs

no. of expanded nodes: 9
taskfile: list12 taskno: task3 of length 7
goal: in_room(box2,room3) by: [gotodoor(door24,room2),
gothrudoor(door24,room4),goto(box2),pushtodoor(box2,door24,room4),
pushthrudoor(box2,door24,room2),pushtodoor(box2,door23,room2),
pushthrudoor(box2,door23,room3)]

CPUsed=63 secs

no. of expanded nodes: 28
taskfile: list12 taskno: task4 of length 12
goal: in_room(big_box,room3) by: [gotodoor(door23,room3),
gothrudoor(door23,room2),gotodoor(door12,room2),open(door12),
gothrudoor(door12,room1),goto(big_box),pushtodoor(big_box,door12,room1),
pushthrudoor(big_box,door12,room2),pushtodoor(big_box,door25,room2),
pushthrudoor(big_box,door25,room5),pushtodoor(big_box,door35,room5),
pushthrudoor(big_box,door35,room3)]

CPUsed=201 secs

no. of expanded nodes: 15
taskfile: list12 taskno: task5 of length 9
goal: in_room(box1,room6)&closed(door56) by: [gotodoor(door23,room3),
gothrudoor(door23,room2),goto(box1),pushtodoor(box1,door25,room2),
pushthrudoor(box1,door25,room5),pushtodoor(box1,door56,room5),
pushthrudoor(box1,door56,room6),gotodoor(door56,room6),close(door56)]

CPUsed=75 secs

no. of expanded nodes: 4
taskfile: list12 taskno: task6 of length 4
goal: in_room(robot,room3) by: [open(door56),gothrudoor(door56,room5),
gotodoor(door35,room5),gothrudoor(door35,room3)]
```

```
CPUsed=17 secs

no. of expanded nodes: 9
taskfile: list12 taskno: task7 of length 9
goal: in_room(box1,room3) by: [gotodoor(door35,room3),
gothrudoor(door35,room5),gotodoor(door56,room5),
gothrudoor(door56,room6),goto(box1),pushtodoor(box1,door56,room6),
pushthrudoor(box1,door56,room5),pushtodoor(box1,door35,room5),
pushthrudoor(box1,door35,room3)]

CPUsed=50 secs

no. of expanded nodes: 5
taskfile: list12 taskno: task8 of length 5
goal: in_room(box2,room7) by: [goto(box2),pushtodoor(box2,door35,room3),
pushthrudoor(box2,door35,room5),pushtodoor(box2,door57,room5),
pushthrudoor(box2,door57,room7)]

CPUsed=30 secs

no. of expanded nodes: 6
taskfile: list12 taskno: task9 of length 5
goal: next_to(box2,big_box) by: [pushtodoor(box2,door57,room7),
pushthrudoor(box2,door57,room5),pushtodoor(box2,door35,room5),
pushthrudoor(box2,door35,room3),pushtobox(box2,big_box)]

CPUsed=33 secs

no. of expanded nodes: 68
taskfile: list12 taskno: task10 of length 23
goal: in_room(box1,room6)&in_room(box2,room6)&in_room(big_box,room6) by:
[goto(box1),pushtodoor(box1,door35,room3),
pushthrudoor(box1,door35,room5),pushtodoor(box1,door56,room5),
pushthrudoor(box1,door56,room6),gotodoor(door56,room6),
gothrudoor(door56,room5),gotodoor(door35,room5),
gothrudoor(door35,room3),goto(box2),pushtodoor(box2,door35,room3),
pushthrudoor(box2,door35,room5),pushtodoor(box2,door56,room5),
pushthrudoor(box2,door56,room6),gotodoor(door56,room6),
gothrudoor(door56,room5),gotodoor(door35,room5),
gothrudoor(door35,room3),goto(big_box),pushtodoor(big_box,door35,room3),
pushthrudoor(big_box,door35,room5),pushtodoor(big_box,door56,room5),
pushthrudoor(big_box,door56,room6)]

CPUsed=415 secs



/* APPENDIX B.1 */
/* rule set after execution os list1, addition of
   2 bad heuristics and execution of list2 */


ch10gothrudoor(x(1),x(2))in_room(robot,x(2))
```

```
in_room(robot,x(3))&closed(x(4))&open(x(1))
connect(x(3),x(5),x(4))&connect(x(5),x(2),x(1))&ne(x(3),x(2))&ne(x(4),x(1))

ch20gothrudoor(x(1),x(2))in_room(robot,x(2))
in_room(robot,x(3))
connect(x(3),x(2),x(1))

ch102gothrudoor(x(1),x(2))in_room(robot,x(2))
in_room(robot,x(3))&open(x(4))&open(x(1))
connect(x(3),x(5),x(4))&connect(x(5),x(2),x(1))&ne(x(3),x(2))&ne(x(4),x(1))

ch_bad2gothrudoor(x(1),x(2))in_room(robot,x(2))
in_room(x(3),x(4))&open(x(1))
connect(x(4),x(2),x(1))&type_of(x(3),box)

ch30pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(x(1),x(4))&open(x(2))
connect(x(4),x(3),x(2))&type_of(x(1),box)&fits_thru(x(1),x(2))

ch50pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(x(1),x(4))&closed(x(5))&open(x(2))
connect(x(4),x(6),x(5))&connect(x(6),x(3),x(2))&fits_thru(x(1),x(5))&
type_of(x(1),box)&fits_thru(x(1),x(2))&ne(x(4),x(3))&ne(x(5),x(2))

ch60pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(x(1),x(4))
connect(x(4),x(3),x(2))&type_of(x(1),box)&fits_thru(x(1),x(2))

ch80pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(x(1),x(4))&in_room(robot,x(3))&open(x(5))&open(x(2))
connect(x(4),x(6),x(5))&connect(x(6),x(3),x(2))&
fits_thru(x(1),x(5))&type_of(x(1),box)&fits_thru(x(1),x(2))&
ne(x(4),x(3))&ne(x(5),x(2))

ch_bad1pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(robot,x(4))&open(x(2))
connect(x(4),x(3),x(2))&type_of(x(1),box)&fits_thru(x(1),x(2))

ch1pushthrudoor(x(1),x(2),x(3))in_room(x(1),x(3))
in_room(x(1),x(4))&open(x(5))&open(x(2))
connect(x(4),x(6),x(5))&connect(x(6),x(3),x(2))&fits_thru(x(1),x(5))&
type_of(x(1),box)&fits_thru(x(1),x(2))&ne(x(4),x(3))&ne(x(5),x(2))
```

```
ch50 4 0 discrim
ch_bad2 1 0 discrim
ch102 10 10 strengthened
ch80 7 30 strengthened
ch_bad1 1 10 discrim
ch10 1 30 discrim
ch60 4 20 discrim
ch1 2 10 multiples
ch30 2 180 discrim
```

```
ch20 1 190 discrim
ch101 9 30 discrim


ch10 has exception ch20
ch70 has exception ch30
ch50 has exception ch30
ch101 has exception ch30
ch90 has exception ch20
ch_bad2 has exception ch102
ch_bad2 has exception ch20
ch102 has exception ch20
ch_bad1 has exception ch30
ch1 has exception ch30
ch1 has exception ch60
ch60 has exception ch101


/* APPENDIX B.2 */
/* results for list44 */

no. of expanded nodes: 10
taskfile: list taskno: task1 of length 7
goal: at(truck,s6) by: [get_in(s12,s16),drive(s16,s12),
load(crate3,s8,s12),unload(crate3,s12,s16),drive(s12,s8),
drive(s8,s7),drive(s7,s6)]
CPUsed=107 secs
CPUsed=6 secs

no. of expanded nodes: 10
taskfile: list taskno: task2 of length 5
goal: stacked(crate2,shelf7) by: [drive(s6,s7),drive(s7,s3),
load(crate2,s2,s3),unload(crate2,s3,s7),truck_stack(crate2,s3,s7,shelf7)]
CPUsed=83 secs
CPUsed=6 secs

no. of expanded nodes: 18
taskfile: list taskno: task3 of length 7
goal: stacked(crate1,shelf14) by: [drive(s3,s2),load(crate1,s1,s2),
drive_load(crate1,s2,s6),drive_load(crate1,s6,s10),
unload(crate1,s10,s14),lift_up(crate1,s14),
crane_stack(crate1,s14,shelf14)]
CPUsed=130 secs
CPUsed=26 secs

no. of expanded nodes: 8
taskfile: list taskno: task4 of length 7
goal: loaded(truck,crate3,s12) by: [drive(s10,s6),drive(s6,s2),
drive(s2,s3),drive(s3,s4),drive(s4,s8),drive(s8,s12),
load(crate3,s16,s12)]
CPUsed=81 secs
CPUsed=16 secs

no. of expanded nodes: 44
```

```
taskfile: list taskno: task5 of length 7
goal: on_floor(crate3,s2)&at(crate2,s4) by: [drive_load(crate3,s12,s8),
drive_load(crate3,s8,s4),drive_load(crate3,s4,s3),
unload(crate3,s3,s2),truck_unstack(crate2,s3,s7,shelf7),
load(crate2,s7,s3),drive_load(crate2,s3,s4)]
CPUsed=452 secs
CPUsed=20 secs

no. of expanded nodes: 13
taskfile: list taskno: task6 of length 9
goal: stacked(crate3,shelf6) by: [move_crane(s14,s10),
move_crane(s10,s6),unload(crate2,s4,s8),drive(s4,s3),
load(crate3,s2,s3),drive_load(crate3,s3,s7),unload(crate3,s7,s6),
lift_up(crate3,s6),crane_stack(crate3,s6,shelf6)]
CPUsed=134 secs
CPUsed=3 secs

no. of expanded nodes: 20
taskfile: list taskno: task7 of length 10
goal: on_floor(crate2,s1)&on_floor(crate3,s9) by: [load(crate2,s8,s7),
drive_load(crate2,s7,s3),drive_load(crate2,s3,s2),
unload(crate2,s2,s1),crane_unstack(crate3,s6,shelf6),
lift_down(crate3),load(crate3,s6,s2),drive_load(crate3,s2,s6),
drive_load(crate3,s6,s10),unload(crate3,s10,s9)]
CPUsed=176 secs
CPUsed=7 secs

no. of expanded nodes: 7
taskfile: list taskno: task8 of length 7
goal: on_floor(crate1,s2) by: [move_crane(s6,s10),move_crane(s10,s14),
crane_unstack(crate1,s14,shelf14),lift_down(crate1),
load(crate1,s14,s10),drive_load(crate1,s10,s6),unload(crate1,s6,s2)]
CPUsed=48 secs
CPUsed=2 secs

no. of expanded nodes: 9
taskfile: list taskno: task9 of length 8
goal: stacked(crate3,shelf9)&loaded(truck,crate1,s12) by:
 [move_crane(s14,s10),move_crane(s10,s9),lift_up(crate3,s9),
crane_stack(crate3,s9,shelf9),load(crate1,s2,s6),
drive_load(crate1,s6,s7),drive_load(crate1,s7,s8),
drive_load(crate1,s8,s12)]
CPUsed=79 secs
CPUsed=2 secs

no. of expanded nodes: 43
taskfile: list taskno: task10 of length 13
goal: stacked(crate1,shelf10)&stacked(crate2,shelf6) by:
[move_crane(s9,s10),drive_load(crate1,s12,s8),drive_load(crate1,s8,s7),
drive_load(crate1,s7,s6),unload(crate1,s6,s10),lift_up(crate1,s10),
crane_stack(crate1,s10,shelf10),move_crane(s10,s6),
drive(s6,s2),load(crate2,s1,s2),unload(crate2,s2,s6),
lift_up(crate2,s6),crane_stack(crate2,s6,shelf6)]
CPUsed=441 secs
```

```
CPUsed=39 secs

/* acquired chunks after execution of list4 */


chx3drive(x(1),x(2))at(truck,x(2))
at(driver,x(1))&unloaded(truck)
type_of(x(1),space)&type_of(x(2),space)&next(x(1),x(2))

chx23drive(x(1),x(2))at(truck,x(2))
at(truck,x(1))&in_truck&clear(x(2))
type_of(x(1),space)&type_of(x(2),space)&next(x(1),x(2))

chx2drive(x(1),x(2))at(truck,x(2))
at(driver,x(3))&clear(x(2))&unloaded(truck)
type_of(x(3),space)&next(x(3),x(1))&type_of(x(2),space)&
next(x(1),x(2))&ne(x(3),x(2))

chx1drive(x(1),x(2))at(truck,x(2))
at(driver,x(3))&clear(x(2))&clear(x(1))&unloaded(truck)
type_of(x(3),space)&next(x(3),x(4))&next(x(4),x(1))&
type_of(x(1),space)&type_of(x(2),space)&next(x(1),x(2))&
ne(x(3),x(1))&ne(x(3),x(2))&ne(x(4),x(2))

chx11drive(x(1),x(2))at(truck,x(2))
in_truck&clear(x(2))&clear(x(1))&unloaded(truck)
next(x(3),x(1))&type_of(x(1),space)&type_of(x(2),space)&
next(x(1),x(2))&ne(x(3),x(2))

chx12drive(x(1),x(2))at(truck,x(2))
in_truck&clear(x(3))&clear(x(1))&clear(x(2))&unloaded(truck)
next(x(4),x(3))&type_of(x(3),space)&next(x(3),x(1))&
type_of(x(1),space)&type_of(x(2),space)&next(x(1),x(2))&
ne(x(4),x(1))&ne(x(4),x(2))&ne(x(3),x(2))

chy3drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
at(truck,x(2))&in_truck&unloaded(truck)
next(x(2),x(3))&type_of(x(2),space)&ne(x(3),x(2))

chx17drive_load(x(1),x(2),x(3))at(x(1),x(3))
at(truck,x(2))&in_truck&clear(x(3))&unloaded(truck)
next(x(4),x(2))&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)&ne(x(4),x(3))

chx15drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
loaded(truck,x(1),x(4))&in_truck&clear(x(3))&clear(x(2))&clear(x(5))
next(x(4),x(5))&type_of(x(4),space)&next(x(5),x(2))&type_of(x(5),space)&
next(x(2),x(3))&type_of(x(2),space)&type_of(x(3),space)&
type_of(x(1),crate)&ne(x(4),x(2))&ne(x(4),x(3))&ne(x(5),x(3))

chx16drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
loaded(truck,x(1),x(4))&in_truck&clear(x(3))&clear(x(2))
next(x(4),x(2))&type_of(x(4),space)&next(x(2),x(3))&
type_of(x(2),space)&type_of(x(3),space)&
```

```
type_of(x(1),crate)&ne(x(4),x(3))

chx21drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
at(truck,x(4))&on_floor(x(1),x(5))&in_truck&clear(x(2))&clear(x(3))
type_of(x(4),space)&next(x(4),x(2))&next(x(5),x(2))&
type_of(x(5),space)&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)&type_of(x(1),crate)&ne(x(4),x(5))&
ne(x(4),x(3))&ne(x(5),x(3))

chx9drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
on_floor(x(1),x(4))&in_truck&clear(x(3))&unloaded(truck)
next(x(4),x(2))&type_of(x(4),space)&next(x(2),x(3))&
type_of(x(3),space)&type_of(x(1),crate)&ne(x(4),x(3))

chy2drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
at(truck,x(4))&in_truck&clear(x(3))&unloaded(truck)
next(x(4),x(2))&type_of(x(4),space)&next(x(2),x(3))&
type_of(x(3),space)&ne(x(4),x(3))

chy8drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
in_truck&clear(x(3))&clear(x(2))
next(x(4),x(2))&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)&ne(x(4),x(3))

chy9drive_load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
on_floor(x(1),x(4))&in_truck&clear(x(2))&clear(x(3))&unloaded(truck)
next(x(5),x(2))&next(x(4),x(6))&type_of(x(4),space)&
next(x(6),x(2))&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)&type_of(x(1),crate)&ne(x(4),x(2))&
ne(x(5),x(4))&ne(x(5),x(6))&ne(x(5),x(3))&ne(x(2),x(4))&
ne(x(2),x(6))&ne(x(4),x(3))&ne(x(6),x(3))

chy11unload(x(1),x(2),x(3))on_floor(x(1),x(3))
on_floor(x(1),x(4))&in_truck&unloaded(truck)
next(x(4),x(2))&type_of(x(4),space)&type_of(x(1),crate)&ne(x(2),x(4))

chx5unload(x(1),x(2),x(3))on_floor(x(1),x(3))
on_floor(x(1),x(4))&in_truck&clear(x(2))&clear(x(3))&unloaded(truck)
next(x(4),x(2))&type_of(x(4),space)&type_of(x(1),crate)&
next(x(3),x(2))&type_of(x(3),space)&type_of(x(2),space)&ne(x(4),x(3))

chx20unload(x(1),x(2),x(3))on_floor(x(1),x(3))
on_floor(x(1),x(4))&in_truck&clear(x(3))&clear(x(2))
next(x(4),x(5))&type_of(x(4),space)&next(x(5),x(2))&
type_of(x(1),crate)&next(x(3),x(2))&type_of(x(3),space)&
type_of(x(2),space)&ne(x(4),x(3))&ne(x(4),x(2))&ne(x(5),x(3))

chy1unload(x(1),x(2),x(3))on_floor(x(1),x(3))
at(truck,x(4))&in_truck&clear(x(2))&clear(x(3))&unloaded(truck)
next(x(4),x(5))&type_of(x(4),space)&next(x(5),x(2))&
next(x(3),x(2))&type_of(x(3),space)&type_of(x(2),space)&
ne(x(4),x(3))&ne(x(4),x(2))&ne(x(5),x(3))

chy5unload(x(1),x(2),x(3))on_floor(x(1),x(3))
```

```
at(truck,x(4))&in_truck&clear(x(3))&clear(x(2))&unloaded(truck)
next(x(5),x(4))&next(x(4),x(2))&type_of(x(4),space)&
next(x(3),x(2))&type_of(x(3),space)&type_of(x(2),space)&
ne(x(5),x(3))&ne(x(5),x(2))&ne(x(4),x(3))

chx22load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
on_floor(x(1),x(2))&in_truck
type_of(x(1),crate)&next(x(2),x(3))&
type_of(x(2),space)&type_of(x(3),space)

chx18load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
at(truck,x(3))&in_truck&unloaded(truck)
type_of(x(1),crate)&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)

chx4load(x(1),x(2),x(3))clear(x(2))
at(driver,x(3))&on_floor(x(1),x(2))&unloaded(truck)
type_of(x(1),crate)&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)

chx10load(x(1),x(2),x(3))loaded(truck,x(1),x(3))
on_floor(x(1),x(2))&in_truck&unloaded(truck)
type_of(x(1),crate)&next(x(2),x(3))&type_of(x(2),space)&
type_of(x(3),space)

chx6crane_stack(x(1),x(2),x(3))stacked(x(1),x(3))
at(crane,x(2))
connect(x(2),x(3))&type_of(x(3),shelf)&
type_of(x(1),crate)&ne(x(2),x(3))&pickup_point(x(2))

chx19crane_stack(x(1),x(2),x(3))stacked(x(1),x(3))
nil
connect(x(2),x(3))&type_of(x(3),shelf)&
type_of(x(1),crate)&ne(x(2),x(3))&pickup_point(x(2))

chy4lift_down(x(1))on_floor(x(1),x(2))
above_floor(x(1),x(2))&at(crane,x(2))
type_of(x(1),crate)

chy6lift_down(x(1))on_floor(x(1),x(2))
above_floor(x(1),x(2))
type_of(x(1),crate)

chy7crane_unstack(x(1),x(2),x(3))loaded(crane,x(1),x(2))
stacked(x(1),x(3))&unloaded(crane)
connect(x(3),x(2))&type_of(x(3),shelf)&
type_of(x(1),crate)&ne(x(2),x(3))

chx3 1 0 discrim
chx4 1 0 discrim
chx5 2 0 discrim
chx6 3 0 discrim
chx10 3 10 discrim
chx11 4 0 discrim
```

```
chx12 4 0 discrim
chx1 1 10 discrim
chx2 1 20 multiples
chx17 5 0 discrim
chx20 6 0 discrim
chx21 6 0 discrim
chx23 6 0 discrim
chy1 1 0 discrim
chy2 1 0 discrim
chy4 1 0 discrim
chy5 2 0 discrim
chy3 1 10 discrim
chx18 5 20 discrim
chy6 2 0 discrim
chy7 2 0 discrim
chy8 3 0 discrim
chy9 3 0 strengthened
chx9 3 20 multiples
chx15 5 10 multiples
chx16 5 10 multiples
chx19 6 30 discrim
chy11 4 0 discrim
chx22 6 10 discrim


chy8 has exception chx15
chy8 has exception chx16
chy8 has exception chx22
chy3 has exception chx22

/* APPENDIC B.3 */
/* handcrafted rules for robot world */

ch(ch10,gothrudoor(_406551,_406552),in_room(robot,_406552),
in_room(robot,XX),connect(XX,U,W)&connect(U,_406552,_406551)&
ne(XX,_406552)&ne(W,_406551)).

ch(ch20,gothrudoor(_406551,_406552),in_room(robot,_406552),
in_room(robot,_406558),connect(_406558,_406552,_406551)).

ch(ch30,pushthrudoor(_406551,_406552,Z),
in_room(_406551,Z),in_room(_406551,U),
connect(U,Z,_406552)&type_of(_406551,box)&fits_thru(_406551,_406552)).

ch(ch50,pushthrudoor(_406551,_406552,Z),
in_room(_406551,Z),in_room(_406551,U),
connect(U,W,V)&connect(W,Z,_406552)&
fits_thru(_406551,V)&type_of(_406551,box)&
fits_thru(_406551,_406552)&ne(U,Z)&ne(V,_406552)).

ch(ch100,pushthrudoor(X,Y,Z),
in_room(X,Z),in_room(X,U),
connect(U,W,V)&connect(W,Z1,Y1)&connect(Z1,Z,Y)&
fits_thru(X,Y1)&fits_thru(X,V)&type_of(X,box)&
```

```
fits_thru(X,Y)&ne(U,Z)&ne(V,Y)&ne(Y,Y1)&ne(Y1,V)).

ch(ch200,gothrudoor(YD,YY),in_room(robot,YY),
in_room(robot,XX),connect(XX,U,W)&connect(U,Y,X)&
connect(Y,YY,YD)&ne(YD,X)&ne(YD,W)&ne(XX,Y)&ne(W,X)).


ch_record(ch10,1,40,multiples).
ch_record(ch100,1,40,multiples).
ch_record(ch200,1,80,multiples).
ch_record(ch20,1,80,multiples).
ch_record(ch50,4,70,multiples).
ch_record(ch30,2,80,multiples).

ch_ex(ch10,ch20).
ch_ex(ch50,ch30).

ch_ex(ch200,ch20).
ch_ex(ch100,ch30).
ch_ex(ch200,ch10).
ch_ex(ch100,ch50).


/* attemp at handcrafted rules for 'warehouse' world */

/* load **********/

ch(ch100,load(X1,X2,X3),loaded(truck,X1,X3),
at(truck,X3),
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).

ch(ch101,load(X1,X2,X3),loaded(truck,X1,X3),
at(truck,X4)&clear(X3),
next(X4,X3)&
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).

ch(ch102,load(X1,X2,X3),loaded(truck,X1,X3),
at(truck,X4)&clear(X3)&clear(X5),
next(X4,X5)&next(X5,X3)&
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space)&
type_of(X4,space)&type_of(X5,space)).

/* unload (1) **********/

ch(ch105,unload(X1,X2,X3),on_floor(X1,X3),
at(truck,X2)&loaded(truck,X1,X2),
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).

ch(ch106,unload(X1,X2,X3),on_floor(X1,X3),
at(truck,X4)&clear(X2)&loaded(truck,X1,X4),
next(X4,X2)&
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).
```

```
/* unload (2) *********/
ch(ch108,unload(X1,X2,X3),at(X1,X3),
at(truck,X2)&loaded(truck,X1,X2),
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).

ch(ch109,unload(X1,X2,X3),at(X1,X3),
at(truck,X4)&clear(X2)&loaded(truck,X1,X4),
next(X4,X2)&
type_of(X1,crate)&next(X2,X3)&type_of(X2,space)&type_of(X3,space) ).

/* drive **********/
ch(ch200,drive(X1,X2),at(truck,X2),
at(truck,X1)&clear(X2),
type_of(X1,space)&type_of(X2,space)&next(X1,X2) ).

ch(ch201,drive(X1,X2),at(truck,X2),
at(truck,X4)&clear(X1),
next(X4,X1)&
type_of(X1,space)&next(X1,X2)&type_of(X2,space)&type_of(X4,space)).

ch(ch202,drive(X1,X2),at(truck,X2),
at(truck,X5)&clear(X1)&clear(X3),
next(X5,X3)&next(X1,X3)&
type_of(X1,space)&next(X1,X2)&type_of(X2,space)&type_of(X3,space)&
ne(X5,X1)&ne(X2,X3)&ne(X5,X2)).



/* drive_load */
ch(ch302,drive_load(X1,S2,S3),loaded(truck,X1,S3),
in_truck&clear(S3)&loaded(truck,X1,S),
next(S,S2)&type_of(S,space)&next(S2,S3)&
type_of(S3,space)&type_of(X1,crate)&ne(S,S3)).

ch(ch305,drive_load(X1,S2,S3),at(X1,S3),
in_truck&clear(S3)&loaded(truck,X1,S),
next(S,S2)&type_of(S,space)&next(S2,S3)&
type_of(S3,space)&type_of(X1,crate)&ne(S,S3)).

ch(ch301,drive_load(X1,S2,S3),loaded(truck,X1,S3),
loaded(truck,X1,S)&in_truck&clear(S8)&
clear(S3)&clear(S2)&unloaded(truck),
next(S,S8)&type_of(S,space)&next(S8,S2)&
next(S2,S3)&type_of(S2,space)&type_of(S3,space)&
type_of(X1,crate)&ne(S,S2)&ne(S,S3)&ne(S8,S3)).

ch(ch306,drive_load(X1,S2,S3),at(X1,S3),
loaded(truck,X1,S)&in_truck&clear(S8)&clear(S3)&
clear(S2)&unloaded(truck),
next(S,S8)&type_of(S,space)&next(S8,S2)&next(S2,S3)&
type_of(S2,space)&type_of(S3,space)&type_of(X1,crate)&
ne(S,S2)&ne(S,S3)&ne(S8,S3)).

ch_ex(ch201,ch200).
```

```
ch_ex(ch202,ch201).
ch_ex(ch202,ch200).

ch_ex(ch102,ch101).
ch_ex(ch102,ch100).
ch_ex(ch101,ch100).

ch_ex(ch106,ch105).
ch_ex(ch109,ch108).

ch_record(ch200,1,0,multiples).
ch_record(ch201,1,0,multiples).
ch_record(ch202,1,0,multiples).
ch_record(ch100,1,0,multiples).
ch_record(ch101,1,0,multiples).
ch_record(ch102,1,0,multiples).
ch_record(ch105,1,0,multiples).
ch_record(ch106,1,0,multiples).
ch_record(ch108,1,0,multiples).
ch_record(ch109,1,0,multiples).
ch_record(ch301,1,0,multiples).
ch_record(ch302,1,0,multiples).
ch_record(ch305,1,0,multiples).
ch_record(ch306,1,0,multiples).
```

APPENDIX C: PROGRAM LISTINGS

The Separate NLP_EBL implementation

MODULE NAME	USED IN..	FUNCTION
nlpd	NLP with E.B.L.	driver for NLP with E.B.L.
nlp0	NLP with E.B.L.	top level strategy through partial plan space
nlp1	NLP with E.B.L.	goal achievement within a partial plan
nlp2	NLP with E.B.L.	implementation of partial plan abstract data type
nlp3	NLP with E.B.L.	utilities
nlpl	NLP with E.B.L.	heuristic aquisition procedures

The FM Implementation

bootc	MEA, FOR, NLP	boot file for the FM system
fenv	MEA, FOR	enviromental variables
fmc	MEA	the MEA strategy
fused	MEA	c-chunk use (called by fmc)
fcomp	MEA	c-chunk optimisation
driver_mec	MEA	provides a user interface
driver_help	MEA, FOR	provides a user help facility
fchmea	MEA	c-chunk creation

faugmac	MEA	c-chunk strengthening
futile	MEA, FOR	utilities
fmacgr	MEA, FOR	macro creation
fuse	FOR	basic chunk use
fexh	FOR	state space search
driver_f	FOR	user interface

(other files which implement NLP within the FM framework are not included for brevity, and because they are similar to NLP_EBL given above.)

```
/************* nlpd nlpd nlpd nlpd nlpd nlpd ****************/
/* driver for NLP with E.B.L. ****************************************/


b :-
        init_world(I),nl,init_count,
        write('This is the non-linear planner..'),nl,
        write('with E.B.L...'),nl,
        write('My current world is '),nl,write(I),nl,
        nl,write('Enter list of goal(s)>'),
        read(G),
        trans(G,goal,Gpp),              /* put goal into internal rep'n*/
                                        /* of Ps in partial plan       */
        assert(ppWP([],[],[],[],[])),
/* now CALL MAIN PREDICATE in TOP LEVEL with initial partial plan*/
        nlp([pp(r([],[],[],[]),
        [],[opi(init,init,[],I,[]),opi(goal,goal,[],[],[])],
                [],Gpp,[],env([],[])) ]).
b :-
        nl,write('Enter LIST of goal(s)>'),b.


/* this is called after nlp has succeeded */
write_out(pp(_,H,Os,Tcons,_,_,_)) :-
        retract(init_world(I)),
        setof(Op1,linearops(Os,Tcons,   Op1), [O|Rest]),
        applyopseq(O,I,  S),
        write('By sequence of operators '),nl,nl,write(O),nl,nl,
        write('goal is satisfied, new state is'),
        nl,nl,write(S),nl,
        assert(init_world(S)),
        write('other sequences '),nl,wlist(Rest),nl,
        learn(H,Os).


/*****************nlp0 nlp0 nlp0 nlp0 -- TOP LEVEL************/
/* for NLP with E.B.L.*****************************************/

nlp(PPs) :-
     empty(PPs),
     write('failure - task is impossible'),
     !.

nlp(PPs) :-
     member(PP,PPs),
     all_goals_achieved(PP),
     write_out(PP).

nlp(PPs) :-
     remove_partial_plan(PPs,   PP,PPs0),
     get_unachieved_predicate(PP,   P,O),
     achieve_all(P,O,PP,   PPs1),
     append(PPs0,PPs1,   PPs2),
     nlp(PPs2).
```

```
nlp(PPs) :-
     remove_partial_plan(PPs,   PP,PPs0),
     get_unachieved_predicate(PP,   P,O),
     nl,write('goal failed***see freda1'),
     tell(freda1),write(P),write(O),nl,
     writepp(PP),nl,nl,tell(user),
     nlp(PPs0).

/************local utilities************************/

empty([]).

all_goals_achieved(PP) :-
        get_Ps(PP,   []),
        !.

/*remove_partial_plan([PP|PPs0],   PPmin,PPs0).*/
/* simple heuristic: find plan that minimises |Os|+|Ps| */
remove_partial_plan([PP|PPs],   PPmin,PPs0) :-
        get_Ps(PP,   Ps),
        get_Os(PP,   Os),
        length(Ps,   PL),
        length(Os,   OL),
        Score is OL+PL,
        for_all_els(PPs,min_pp,plan(PP,Score),     plan(PPmin,SS)),
        write(SS),
        removeL(PPmin,[PP|PPs],   PPs0).
min_pp(PP,plan(PP1,Score1),   plan(PP1,Score1)) :-
        get_Ps(PP,   Ps),
        get_Os(PP,   Os),
        length(Ps,   PL),
        length(Os,   OL),
        Score2 is OL+PL,
        Score2 >= Score1,!.
min_pp(PP,plan(_,Score1),   plan(PP,Score2)) :-
        get_Ps(PP,   Ps),
        get_Os(PP,   Os),
        length(Ps,   PL),
        length(Os,   OL),
        Score2 is OL+PL,
        Score2 < Score1,!.


get_unachieved_predicate(PP,   P,O) :-
        get_Ps(PP,   Ps),
        most_inst(Ps,g(P,O)),
        !.

/* achieve asserts new plans as prolog clauses since
   this makes the variables therein, independent */

achieve_all(P,O,PP,   PPs) :-
        achieve_all(P,O,PP),
        setof(X,retract(newplan(X)),PPs).
```

```
/*achieve_all(P,O,PP) :-
        init_pp_trans,
        achieve1(init,P,O,PP,   PP0),
        init_pp_trans,
        init_rec(PP0,PP1),
        assert(newplan(PP1)),
        fail.  */
achieve_all(P,O,PP) :-
        init_pp_trans,
        achieve1(_,P,O,PP,   PP0),
        init_pp_trans,
        init_rec(PP0,PP1),
        assert(newplan(PP1)),
        fail.
achieve_all(P,O,PP) :-
        init_pp_trans,
        achieve2(P,O,PP,   PP0),
        init_pp_trans,
        init_rec(PP0,PP1),
        assert(newplan(PP1)),
        fail.
achieve_all(_,_,_).

/*****************TOP LEVEL END********************************/
/*****************nlp1 nlp1 nlp1 nlp1 nlp1 *******************/
/*****************GOAL ACHIEVEMENT BY EXISTING OP**************/

/* The specification of this program is written as a series of
   post conditions on the R.H.S. of the code */
/* apart from simple retrieve fns this relies on the correct
   implementation of six predicate:
        unify(P,Q,Ts);
        unify(P,Q,PP,   PP0);
        before(P,Q,Ts);
        before(P,Q,PP,   PP0);
        constrain(P,Q,PP,   PP0);
        insert_op(P,PP,   A,PP0)
*/

achieve1(A,P,O,PP,   PP6) :-

        get_el_Os(PP,   A),           /* there exists A in Os: */
        achieve(P,O,A,PP,   PP3),     /* achieve(P,O,A)       */

        add_el_As(P,O,PP3,   PP4),
        del_el_Ps(P,O,PP4,   PP5),
                                      /* for monitoring only.. */


        sift_out(PP5,PP51),
        get_history(PP,   H1),
        increment_count(N1),
        append(H1,[N1],H),
        rep_history(PP51,H,   PP6),
```

```
        retract(pp_trans(X1,X2,X3,X4,X5)),
        assert(ppA(H,A,X1,X2,X3,X4,X5)),
        store_rec(PP6),

        tell(freda),
        nl,write(ppA(H,A,X1,X2,X3,X4,X5)),nl,
        nl,write(P),write(O),
        writepp(PP6),tell(user),
        write(.),ttyflush.

achieve(P,O,A,PP,   PP5) :-
        get_el_add(A,PP,   Q),          /* there exists Q in A.a:*/
        unify(P,Q,PP,   PP1),           /* nec_unify(P,Q,Es) &   */
        before(A,O,PP1,   PP2),         /* before(A,O,Ts) &      */
        bef_rec(A,O,PP2,   PP3),
        con_rec(A,Q,PP3,   PP4),
        get_Os(PP4,   Os),              /* for all C in Os:      */
        for_all_els(Os,
                declobber(P,A,O),
                PP4,   PP5).            /* declobber(P,A,O,C)    */

/* This part makes sure C is not a clobberer */

declobber(_,O,_,O,PP, PP) :-            /* C = O V               */
        !.
declobber(_,_,A,A,PP, PP) :-            /* C = A V               */
        !.
declobber(_,_,O,C,PP, PP3) :-
        get_Ts(PP,   Ts),
        before(O,C,Ts),                 /* before(O,C,Ts) V      */
        bef_rec(O,C,PP,   PP3),
        !.
declobber(_,A,_,C,PP, PP3) :-
        get_Ts(PP,   Ts),
        before(C,A,Ts),                 /* before(C,A,Ts) V      */
        bef_rec(C,A,PP,   PP3),
        !.
declobber(P,_,_,C,PP, PP3) :-
        get_Es(PP,   Es),
        not(   get_el_del(C,PP,   Q),   /*not(there exists Q in  */
               unify(P,Q,Es)  ),        /*C.d: pos_unify(Q,P,Es))*/
        uni_rec(P,C,PP,   PP3),
        !.

/* If this point is reached then C is a clobberer;
   this part CHANGES the partial plan to avoid this */

/* Don't need to bother recording additions because these are recorded
   for the H operator anyway - although they must still be taken
   from collection of required proof objects */

declobber(_,_,O,C,PP, PP0) :-
        before(O,C,PP,   PP0).          /* make before(O,C,Ts) V */
```

```
declobber(_,A,_,C,PP, PP0) :-
        before(C,A,PP,   PP0).          /* make before(C,A,Ts) V */

/* C.d contains at least one predicate which clobbers P .. */

declobber(P,_,_,C,PP, PP0) :-
        get_del(C,PP,  Cd),             /* for all Q in C.d: */
        for_all_els(Cd,
                    constrain(P),
                    PP,   PP0).         /* make not(unify(P,Q,Es)) */

/*****************GOAL ACHIEVEMENT END************************/

/*****************GOAL ACHIEVEMENT BY NEW OP**************/
achieve2(P,O,PP,   PP8) :-

        insert_op(P,PP,   A,PP1),      /* there exists new A in Os:*/
        achieve(P,O,A,PP1,   PP4),     /* achieve(P,O,A)        */
        get_As(PP4,  As),              /* for all (P,O) in As:  */
        for_all_els(As,
                    declobber_As(A),
                    PP4,   PP5),       /* declobber_As(A,(P,O))*/

        add_el_As(P,O,PP5,   PP6),
        del_el_Ps(P,O,PP6,   PP7),
                                       /* for monitoring only.. */
        sift_out(PP7,PP71),
        get_history(PP,   H1),
        increment_count(N1),
        append(H1,[N1],H),
        rep_history(PP71,H,   PP8),

        add_Os_trans(A,PP8),
        retract(pp_trans(X1,X2,X3,X4,X5)),
        assert(ppA(H,A,X1,X2,X3,X4,X5)),
        store_rec(PP8),

        tell(freda),
        nl,write(ppA(H,A,X1,X2,X3,X4,X5)),nl,
        nl,write(P),write(O),
        writepp(PP8),tell(user),
        write(.),ttyflush.


declobber_As(A,g(_,A),PP, PP) :       /* A = O  V                */
        !.
declobber_As(A,g(_,O),PP, PP3) :-
        get_Ts(PP,   Ts),              /* before(O,A,Ts) V        */
        before(O,A,Ts),
        bef_rec(O,A,PP,   PP3),
        !.
declobber_As(A,g(P,O),PP, PP3) :-      /* for all Q in Ad:        */
        get_del(A,PP,   Ad),
```

```
        for_all_els(Ad,
                    declobber_pred(A,O,P),
                    PP,   PP3).        /* declobber_pred(A,O,P,Q)*/



declobber_pred(_,_,P,Q,PP,   PP):-
        get_Es(PP,   Es),
        not(unify(Q,P,Es)),           /* not(pos_unify(Q,P,Es)) V */
        !.
declobber_pred(A,O,P,Q,PP,   PP3):-
        get_el_Os(PP,   W),           /* there exists W in Os: */
        get_Ts(PP,   Ts),
        before(A,W,Ts),               /* before(A,W,Ts) &      */
        before(W,O,Ts),               /* before(W,O,Ts) &      */
        bef_rec(A,W,PP,   PP1),
        bef_rec(W,O,PP1,   PP3),
        get_el_add(W,PP,   R),        /* there exists R in W.a:*/
        not(not(
          unify(P,Q,PP,_),          /* nec_unify(P,Q,Es) &   */
          P == R   )),       /*       */
        !.

/* Otherwise we'll have to add constraints, now we know   */
/* A is a Clobber for some P in O, possibly before O.     */


declobber_pred(A,O,_,_,PP, PPO) :-
        before(O,A,PP, PPO).          /* if nec, make before(O,A,Ts) */

declobber_pred(_,_,P,Q,PP,   PPO):-
        constrain(Q,P,PP,PPO).       /* put constraint on P or Q   */

declobber_pred(A,O,P,Q,PP,   PPO):-
        get_el_Os(PP,   W),           /* there exists W in Os:  */
        before(A,W,PP,  PP1),         /* make before(A,W,Ts) &  */
        before(W,O,PP1,  PPO),        /* make before(W,O,Ts) &  */
        get_el_add(W,PPO,   R),       /* there exists R in W.a:*/
        not(not(
          unify(P,Q,PPO,_),         /* nec_unify(P,Q,Es) &    */
          P == R   )).       /*       */

/* One other possibility is where W is got by adding a new operator
   altogether! we leave this out! */

/******************GOAL ACHIEVEMENT END*************************/


/*************nlp2 nlp2 nlp2 nlp2 nlp2**************************/
/*      Partial Plan ADT                **************************/
/*
<plan rep> = pp(R,H,Os,Ts,Ps,As,Es).

R  temporarily stores the change info for learner.
```

```
H = history

<Os rep> =   list of  opi(<id>,<op_name>,<preL>,<addL>,<delL>)
<Ts rep> =   list of t(<op_identifier>,<op_identifier>)
<Ps rep> =   list of g(<predicate>,<op_identifier>)
<As rep> =   list of g(<predicate>,<op_identifier>)
<Es rep> =  env( ne's -binding restrictions, other restrictions )

NOTE: driver must also be changed!!!!!!!

*/

/*********************************************************************/


get_history(pp(_,H,_,_,_,_,_),   H).
rep_history(pp(R,_,A,B,C,D,E),H,   pp(R,H,A,B,C,D,E)).
increment_count(N1) :-
        retract(count(N)), N1 is N+1, assert(count(N1)),!.
init_count :- retract(count(_)), assert(count(0)),!.
init_count :- assert(count(0)).

/* plan component access */

get_Os(pp(_,_,Os,_,_,_,_),   OsI) :-
        lop(Os,OsI).            /* just get operator identifiers */
lop([opi(Id,_,_,_,_)|R],[Id|A]) :-
        lop(R,A),
        !.
lop([],[]).

get_O(Id,pp(_,_,Os,_,_,_,_),A) :-
        member(opi(Id,A,_,_,_), Os),!.

get_Ts(pp(_,_,_,Ts,_,_,_),   Ts).
get_Ps(pp(_,_,_,_,Ps,_,_),   Ps).
get_As(pp(_,_,_,_,_,As,_),   As).
get_Es(pp(_,_,_,_,_,_,Es),   Es).


/* plan component member access */

get_el_Os(pp(_,_,Os,_,_,_,_),   O) :-
        lop(Os,OsI),
        member(O,OsI).

get_el_Ps(pp(_,_,_,_,Ps,_,_),   g(P,O)) :-
        member(g(P,O),Ps).

get_el_As(pp(_,_,_,_,_,As,_),   g(P,O)) :-
        member(g(P,O),As).

/* operator component access */
```

```
get_del(Id,pp(_,_,Os,_,_,_,_),   Del) :-
        member(opi(Id,_,_,_,Del),Os),
        !.

/* operator component member access */

get_first_el_add(Id,pp(_,_,Os,_,_,_,_),   A) :-
        member(opi(Id,_,_,Add,_),Os),!,
        member(A,Add),!.
get_el_add(Id,pp(_,_,Os,_,_,_,_),   A) :-
        member(opi(Id,_,_,Add,_),Os),!,
        member(A,Add).
get_el_del(Id,pp(_,_,Os,_,_,_,_),   D) :-
        member(opi(Id,_,_,_,Del),Os),!,
        member(D,Del).


/* plan component update */

rep_Ts(pp(R,H,Os,_,Ps,As,Es),New,   pp(R,H,Os,New,Ps,As,Es)).
rep_Es(pp(R,H,Os,Ts,Ps,As,_),New,   pp(R,H,Os,Ts,Ps,As,New)).


add_Ts(pp(R,H,Os,Ts,Ps,As,Es),NewT,   pp(R,H,Os,[NewT|Ts],Ps,As,Es)) :-
        add_Ts_trans(NewT,Os).

add_op(O,pp(R,H,C1,C2,C3,C4,C5),   pp(R,H,C,C2,C3,C4,C5)) :-
        append(C1,[O],C).


add_Ps(Gs,Id,pp(R,H,Os,C2,Ps,C4,C5),   pp(R,H,Os,C2,NewPs,C4,C5)) :-
        trans(Gs,Id,Gsid),
        append(Gsid,Ps,NewPs),
        add_Ps_trans(Gsid,Os),!.

add_el_As(P,O,pp(R,H,Os,Ts,Ps,As,Es),   pp(R,H,Os,Ts,Ps,[g(P,O)|As],Es) ) :-
        add_As_trans(g(P,O),Os),!.
/* updates Es with input from a new operator */
add_Es(E,pp(R,H,C1,C2,C3,C4,env(E1,E2)),
                                pp(R,H,C1,C2,C3,C4,env(NewE1,NewE2))) :-
        sortne(E,Ene,Erest),
        append(Ene,E1,NewE1),
        append(Erest,E2,NewE2),
        !.
sortne([],[],[]).
sortne([ne(A,B)|ER],[ne(A,B)|Ene],Erest) :-
        sortne(ER,Ene,Erest).
sortne([E|ER],Ene,[E|Erest]) :-
        sortne(ER,Ene,Erest).


del_el_Ps(P,O,pp(R,H,Os,Ts,Ps,As,Es),   pp(R,H,Os,Ts,NewPs,As,Es)) :-
        removeL_equiv(g(P,O),Ps,NewPs),
        !.
```

```
writepp(pp(R,H,Os,Ts,Ps,As,env(E1,E2))) :-
        file_dump(on),nl,nl,            /* flag for storing trace */
        write(H),nl,
        write(R),nl,nl,
        wlist(Os),nl,
        wlist(Ts),nl,
        wlist(Ps),nl,
        wlist(As),nl,
        wlist(E1),nl,
        wlist(E2),nl,
        write('*****************************************************'),nl,
        !.
writepp(_).

/******Record for Learning ***************************************/



/******IMPLEMENTATION OF THE SIX AUX. FNS IN NLP1*****************/

/************* before(X,Y,PP,PP0): make X nec. before Y in PP0 */

before(X,X,_,   _) :-
        !,fail.                 /* can't put X before itself!*/
before(X,Y,PP,    PP) :-
        get_Ts(PP,   Ts),
        before(X,Y,Ts),
        !.
before(X,Y,PP,    PP0) :-
        get_Ts(PP,   Ts),
        not( before(Y,X,Ts)),
        add_Ts(PP,t(X,Y),   PP0),
        !.

/*************  before(X,Y,Ts): X is nec. before Y in Ts */
/* before(X,Y,Ts)'s  specification:
        X < Y   <=>  X=init V Y=goal V
                    { t(X,Y) in Ts V
                      there exists Z in Os:
                     t(X,Z) in Ts & Z < Y }
*/

before(X,Y,Ts) :-
        not(X=Y),
        not(Y=init),
        not(X=goal),
        befor(X,Y,Ts).

befor(init,_,_) :- !.
befor(_,goal,_) :- !.
befor(X,Y,Ts) :-
        member(t(X,Y),Ts),      /***uses Ts's REP ***/
        !.
```

```
befor(X,Y,Ts) :-
        member(t(X,Z),Ts),
        befor(Z,Y,Ts),
        !.


/******************** unify(P,Q,PP,PP0) */
unify(P,Q,PP,   PP) :-
        P = Q,
        get_Es(PP,   Es),
        not(not(consis(Es))).

/******************  unify(P,Q,Es) */
unify(P,Q,Es) :-
        not(not(
                P = Q,
                consis(Es)   )).

consis(env(Ene,Econs)) :-
        env(Env),
        follows(Econs,Env),
        numbervars(Ene,1,_),
        consist(Ene),
        !.
follows([],_).
follows([Y|R],E) :-
        member(Y,E),
        follows(R,E).
consist([]).
consist([ne(X,X)|_]) :-
                !,fail.
consist([_|Y]) :-
                consist(Y).
/*****************************************************************/

insert_op(G,PP,   OP,PP4) :-
                operator(N,
                        E,
                        P,
                        A,
                        D),
        not(not( A = [G|_] )),      /* verefy G is first in padd */
        gensym(op,OP),
        add_op(opi(OP,N,P,A,D),PP,   PP1),
        add_Ps(P,OP,PP1,   PP2),
        add_Es(E,PP2,   PP4).

/*****************************************************************/


constrain(P,Q,PP,   PP) :-
        get_Es(PP,   Es),
        not(unify(Q,P,Es)),
        !.
```

```
constrain(P,Q,PP,   PP0) :-
        get_Es(PP,   Es),
        put_constraint(Q,P,Es,NewEs),   /* put constraint on P or Q */
        rep_Es(PP,NewEs,   PP0).

put_constraint(Q,P,Es,NewEs) :-
        put_c(P,Q,Es,NewEs),
        not(unify(P,Q,NewEs)),
        not(not(consis(NewEs))).

put_c(U,P,env(X,Y),env([ne(A,B)|X],Y) ) :-
        U =.. [_|TU],
        P =.. [_|TP],
        p_c(TU,TP,ne(A,B)).

p_c([T1|_],[T2|_],ne(T1,T2)) :-
        var(T1).
p_c([T1|_],[T2|_],ne(T1,T2)) :-
        var(T2).
p_c([_|R1],[_|R2],O) :-
        p_c(R1,R2,O).
p_c([],[],none).

/***********************************************************/
/* Constructs a sufficient condition for achieve satisfaction */
/* present rep'n = r(P in A,Os',Ts',Es') */

init_rec(pp(_,H,V,W,X,Y,Z), pp(r([],[],[],[]),H,V,W,X,Y,Z)).

store_rec(pp(r(A,_,C,D),H,Os,_,_,_,_)) :-
        lop(Os,OsI),
        removeL(goal,OsI,Os1),
        removeL(init,Os1,Os2),
        assert(ppR(H,A,Os2,C,D)),!.

con_rec(A,Q, pp(r(Con,O,T,E),H,Os,W,X,Y,Z),
               pp(r([c(A,Q)|Con],O,T,E),H,Os,W,X,Y,Z) ) :- !.

/* Os?? - fill when storing?? */

bef_rec(A,C,pp(r(Con,O,T,E),H,Os,W,X,Y,Z),
                      pp(r(Con,O,[t(A,C)|T],E),H,Os,W,X,Y,Z)) :-
        not(A=init),
        not(C=goal),
        !.
bef_rec(_,_,PP,  PP).

/*      member(opi(A,Op1,_,_,_), Os),
        member(opi(C,Op2,_,_,_), Os),!. */

uni_rec(P,C,pp(r(Con,O,T,E),H,Os,W,X,Y,Z),
                      pp(r(Con,O,T,[n(C,P)|E]),H,Os,W,X,Y,Z)) :- !.
```

```
/******************end of adt ****************************************/
/* This section builds up the declarative form of the PP transform **/
/* form is   pp_trans(+Os,+set of Ts,+As(=-Ps),+set of Ps,+set of Es) */

add_Os_trans(A,PP) :-
        retract(pp_trans(_,B,D,E,F)),
        assert(pp_trans(A,B,D,E,F)),!.

add_Ts_trans(t(_,goal),_) :- !.
add_Ts_trans(t(init,_),_) :- !.
add_Ts_trans(t(Op1,Op2),_) :-
        /*member(opi(Op1,N1,_,_,_), Os),
        member(opi(Op2,N2,_,_,_), Os),*/
        retract(pp_trans(A,B,D,E,F)),
        assert(pp_trans(A,[t(Op1,Op2)|B],D,E,F)),!.
add_Ps_trans(Ps,_) :-
        retract(pp_trans(A,B,_,C,E)),
        assert(pp_trans(A,B,Ps,C,E)),!.
add_As_trans(g(P,Op),_) :-
        /*member(opi(Op,N,_,_,_), Os),*/
        retract(pp_trans(A,B,D,_,E)),
        assert(pp_trans(A,B,D,g(P,Op),E)),!.

init_pp_trans :-
        retract(pp_trans(_,_,_,_,_)),
        assert(pp_trans(none,[],[],none,[])),!.
init_pp_trans :-
        assert(pp_trans(none,[],[],none,[])),!.

/*************************************************/
sift_out(PP,  PP1) :-
        get_Es(PP,   env(E1,E2)),
        remove_ground_preds(E1,E3),
        remove_ground_preds(E2,E4),
        sift(E3,NE1),
        sift(E4,NE2),
        rep_Es(PP,env(NE1,NE2),   PP1),
        !.

remove_ground_preds([X|Y],Z) :-
        is_ground(X),
        remove_ground_preds(Y,Z).
remove_ground_preds([X|Y],[X|Z]) :-
        remove_ground_preds(Y,Z).
remove_ground_preds([],[]).
is_ground(X) :-
        X =.. [_|Y],
        is_groundL(Y).
is_groundL([X|T]) :-
        !,
        not(var(X)),
        is_groundL(T).
is_groundL([]).
```

```
increment_count(N1) :-
        retract(count(N)), N1 is N+1, assert(count(N1)),!.
/* general predicate for removing equiv. dupes from a list
   which contains perhaps uninst. vars */
sift([],[]) :- !.
sift([X],[X]) :- !.
sift([H|T],   [H|O]) :-
        not_mem(H,T),
        sift(T,O),
        !.
sift([_|T],   O) :-
        sift(T,O),
        !.
not_mem(E,[X|_]) :-
        E == X,
        !,fail.
not_mem(E,[_|Y]) :-
        not_mem(E,Y).
not_mem(_,[]).

/*************nlp3 nlp3 nlp3 nlp3 nlp3*************************/
/***********************************************************/
/*              local utilities                ***************/


/* implements "for all els in list do OP(args, el., I, O)" */
for_all_els([X|Rest],Op,I,O) :-
        Op =.. OL,
        append(OL,[X,I,I1],OL1),
        Pred =.. OL1,
        call(Pred),
        for_all_els(Rest,Op,I1,O).
for_all_els([],_,I,I).


/* list of goals -> g(el,Id) for all el in list
        - used in plan adt and driver*/
trans([],_,[]).
trans([L|LR],Id,[g(L,Id)|LRN]) :- trans(LR,Id,LRN).
detrans([],[]).
detrans([g(L,_)|LRN],[L|LR]) :- detrans(LRN,LR).


/* terrible impl. of not */
not(X) :- call(X),!,fail.
not(_).
not(X,Y) :- call(X),call(Y),!,fail.
not(_,_).
not(X,Y,Z) :- call(X),call(Y),call(Z),!,fail.
not(_,_,_).


/*****************converts a partial order into a linear sequence********/
```

```
/* patch *************************/
linearops(Lops,[],   [N]) :-
        member(opi(_,N,_,_,_),Lops),
        not(N = init), not(N= goal), !.

linearops(Lops,Tcons,   Linops) :-
        line(Tcons,   LTcons),
        id_to_name(LTcons,Lops,   Linops).

line(T,  TO) :-
        get_identifsT(T,IdL),
        list_to_set(IdL,   IdS),
        sortI1(IdS,T,   TO).

get_identifsT([],[]).
get_identifsT([t(OP1,OP)|GR],[OP1,OP|Y]) :-
        get_identifsT(GR,Y).

/* Now make the p.o. into a lattice by adding init & goal */

sortI1(Ids,T, Ord):-
        add_limit(Ids,T,   NewT),
        sortI([init,goal |Ids] ,NewT,[],  Ord1),
        removelast(Ord1,[_|Ord]).

add_limit([I|L],T, [t(I,goal),t(init,I) | NewT]) :-
        add_limit(L,T,  NewT).
add_limit([],T,  T).

/* now sort the lattice */

sortI([X],_,_,   [X]).
sortI(IdS,T,Used,   [L|R]) :-
        lowest(IdS,T,Used,   L),
        removeL(L,IdS,   IS),
        sortI(IS,T,[L|Used],   R).

lowest(IdS,T,Used,   L) :-
        member(L,IdS),
        member(t(L,_),T),
        not(    member(t(X,L),T), not( member(X,Used) ) ).
lowest([Id|_],_,   Id).

id_to_name([],_,   []).
id_to_name([Id|R],Lops,   [Name|NR]):-
        member(opi(Id,Name,_,_,_),Lops),
        id_to_name(R,Lops,   NR).


/**********************************************************/
/* this part allows dumps to be turned off or on */

file_dump(on).
fdump :- retract(file_dump(_)),
```

```
        assert(file_dump(on)).
no_fdump :- retract(file_dump(_)),
        assert(file_dump(off)).
close_files :- tell(freda),told.

/********* applies a list of ops to create a FINAL state */
applyopseq([Op|T1],S,S2) :-
        applyop(S,Op,S1),
        applyopseq(T1,S1,S2).
applyopseq([],S,S) :- !.

applyop(State,Op,New) :- operator(Op,
                                   _,
                                   _,
                                  Add,
                                  Del
                                   ),
                        remove_list(Del,State,New1),
                        add_list(Add,New1,New).


/**********list processing functions************************/

remove_list([D|Del],State,New) :-
        remove_ALL(D,State,New2),
        remove_list(Del,New2,New),!.
remove_list([_|Del],State,New) :-
        remove_list(Del,State,New),!.
remove_list([],New,New) :- !.

add_list(Add,New1,New) :-
        append(Add,New1,New2),
        list_to_set(New2,   New),!.

reverse(X,Y) :- reverse_x(X,[],Y),!.
reverse_x([],C,C).
reverse_x([H|T],C,R) :- reverse_x(T,[H|C],R).

last([E|[]],E) :- !.
last([_|T],E) :- last(T,E).

last2([E,_],E) :- !.
last2([_|T],E) :- last2(T,E).

append([],L,L).
append([H|T],L,[H|Z]) :- append(T,L,Z).

/* removes the first unif'y occurence of E1 in list */

removeL(E1,[E1|T],T) :- !.
removeL(E1,[X|T1],[X|T2]) :- removeL(E1,T1,T2),!.

/* removes the first equiv. occurence of E1 in list */
removeL_equiv(E1,[E2|T],T) :- E1 == E2, !.
```

```
removeL_equiv(E1,[X|T1],[X|T2]) :- removeL_equiv(E1,T1,T2),!.

/* removes all unif'y occurences of E1 in list */
remove_ALL(_,[],[]).
remove_ALL(E1,[E1|T],T1) :- remove_ALL(E1,T,T1),!.
remove_ALL(E1,[X|T1],[X|T2]) :- remove_ALL(E1,T1,T2),!.

remove_ALL_L([],L,L) :- !.
remove_ALL_L([H|T],L,L1) :-
        remove_ALL(H,L,L2),
        remove_ALL_L(T,L2,L1),!.

removelast([_],[]) :- !.
removelast([X|X1],[X|Y1]) :- removelast(X1,Y1).

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

/*pretty print lists*/
writeL([]).
writeL([X|Y]) :- write(X),writeL(Y).

wlist([X|Y]) :- write(X),nl,wlist(Y).
wlist([]).

/* This procedure changes a list to a set */
list_to_set([],[]) :- !.
list_to_set([L|T],S) :- member(L,T),!,list_to_set(T,S).
list_to_set([L|T],[L|T1]) :- list_to_set(T,T1).


/**************** generate symbol predicate*************** */

gensym(Root,Atom) :-
                        getnum(Root,Num),
                        name(Root,Name1),
                        name(Num,Name2),
                        append(Name1,Name2,Name),
                        name(Atom,Name),
                        !.
getnum(Root,Num) :-
                        retract(current_num(Root,Num1)),!,
                        Num is Num1+1,
                        asserta(current_num(Root,Num)).
getnum(Root,1) :- asserta(current_num(Root,1)).


/* used in weak heuristic: */
/* finds the most inst'd el' of a list by looking at simply the
   second level terms i.e. g(i,o), f(I,o), d(F,D,f) would give g(i,o) */
 /*PRE of most_inst(X,Y)= X is list of >0 arity fns */
 /*POST: Y = function which maximises {no.of const*(1+1/arity)} */

most_inst([G1|G],B) :-
```

```
        value(G1,N),
        m_inst(G,G1,N,B,_),
        !.

m_inst([G1|T],_,N,   G2,N2) :-
        value(G1,N1),
        N1 >  N,
        m_inst(T,G1,N1,   G2,N2).
m_inst([_|T],G,N,   G2,N2) :-
        m_inst(T,G,N,   G2,N2).
m_inst([],G,N,   G,N) :-
        !.

value(g(G,_),1) :-
        atom(G),
        !.
value(g(G,_),N) :-      /* specifically when terms are of form g(X,_) */
        G =.. [_|L],    /* and we are interested in X */
        value1(L,0,N1),
        length(L,Len),
        N is N1+ N1/Len,
        !.
value(G,1) :-
        atom(G),
        !.
value(G,N) :-
        G =.. [_|L],
        value1(L,0,N1),
        length(L,Len),
        N is N1+ N1/Len,
        !.
value1([H|T],N1,N) :-
        nonvar(H),
        N2 is N1+1,
        value1(T,N2,N).
value1([_|T],N1,N) :-
        value1(T,N1,N).
value1([],N,N) :-
        !.

/**** nlpl nlpl nlpl nlpl nlpl nlpl nlpl nlpl nlpl *********/

learn(H,Os) :-
        assert(wp(0,[],[],[],[],[])),
        learn1(H,0,Os).
learn1([],I,_) :- retr,tell(freda),nl,write(I),write(' transforms'),
                nl,told,!.
retr :- retract(ppA(_,_,_,_,_,_)),fail.
retr :- retract(ppR(_,_,_,_,_)),fail.
retr :- retract(wp(_,_,_,_,_,_)),fail.
retr.

/* start with last transform first .. */
learn1(H,I,Os) :-
```

```
        ppA(H,A,OsA,TsA,PsA,AsA,EsA),
        ppR(H,Con,OsS,TsS,EsS),
        writeppT(ppT(A,OsA,TsA,PsA,AsA,EsA)),
        writeppR(ppR(Con,OsS,TsS,EsS)),
        print_others(H),
        wp(I,WPcons,WpOs,WpTs,WpPs,WpEs),

        append(WPcons,Con,WPcons0),
        append(WpOs,OsS,WpOs1),
        append(WpTs,TsS,WpTs1),
        append(WpPs,[AsA],WpPs1),        /* add achieved P to Ps*/
        /*append(WpAs,AsS,WpAs1),*/
        append(WpEs,EsS,WpEs1),

        remove_ALL_L([OsA],WpOs1,WpOs2),
        remove_ALL_L(TsA,WpTs1,WpTs21),
/* remove any fact in wpT which refers to OsA - the op added*/
        remove_op(OsA,WpTs21,WpTs2),
        remove_ALL_L(PsA,WpPs1,WpPs2),
        remove_ALL_L(EsA,WpEs1,WpEs21),
        remove_opN(OsA,WpEs21,WpEs2),
        remove_opC(OsA,WPcons0,WPcons1),

        list_to_set(WpOs2,WpOs3),
        list_to_set(WpTs2,WpTs3),
        list_to_set(WpPs2,WpPs3),
        list_to_set(WpEs2,WpEs3),

        I1 is I+1,
/* changes names of ops to ids **/
        rem_ident1(WPcons1,WPcons2,Os),
        rem_ident2(WpOs3,WpOs4,Os),
        rem_ident3(WpTs3,WpTs4,Os),
        rem_ident4(WpPs3,WpPs4,Os),
        assert(wp(I1,WPcons1,WpOs3,WpTs3,WpPs3,WpEs3)),
        writeWP(wp(I1,WPcons2,WpOs4,WpTs4,WpPs4,WpEs3)),
        removelast(H,H1),
        learn1(H1,I1,Os),!.

print_others(H) :-
        last2(H,HL),
        ppA(H1,X1,X2,X3,X4,X5,X6),
        ppR(H1,X7,X8,X9,X10),
        last2(H1,HL),
        not(H1 = H),
        tell(freda),nl,nl,write('others...************************'),
        writeppT(ppT(X1,X2,X3,X4,X5,X6)),
        writeppR(ppR(X7,X8,X9,X10)),
        tell(freda),nl,write('**********************************'),
        tell(user).
print_others(_).


writeppT(ppT(A,OsA,TsA,PsA,AsA,EsA)) :-
```

```
        tell(freda),nl,nl,nl,
        write(ppT(A,OsA,TsA,PsA,AsA,EsA)),
        tell(user).
writeppR(ppR(Con,OsS,TsS,EsS)) :-
        tell(freda),nl,nl,nl,
        write(ppR(Con,OsS,TsS,EsS)),
        tell(user).
writeWP(wp(I,WpCons,WpOs,WpTs,WpPs,WpEs)) :-
        tell(freda),nl,nl,nl,
        write(I),write(WpCons),nl,
        write(WpOs),nl,
        write(WpTs),nl,
        write(WpPs),nl,
        write(WpEs),nl,
        nl,tell(user).

remove_op(A,[t(I,I1)|R],[t(I,I1)|NR]) :-
        not(I = A),
        not(I1 = A),
        remove_op(A,R,NR),
        !.
remove_op(A,[_|R],NR) :-
        remove_op(A,R,NR),
        !.
remove_op(_,[],[]).

remove_opN(init,X,X) :- !.
remove_opN(none,X,X) :- !.
remove_opN(A,[n(A,_)|R],NR) :-
        remove_opN(A,R,NR),
        !.
remove_opN(A,[X|R],[X|NR]) :-
        remove_opN(A,R,NR),
        !.
remove_opN(_,[],[]).

remove_opC(init,X,X) :- !.
remove_opC(none,X,X) :- !.
remove_opC(A,[c(A,_)|R],NR) :-
        remove_opC(A,R,NR),
        !.
remove_opC(A,[X|R],[X|NR]) :-
        remove_opC(A,R,NR),
        !.
remove_opC(_,[],[]).


rem_ident1([c(I,P)|R],[c(Name,P)|NR],Os) :-
        member(opi(I,Name,_,_,_),Os),
        rem_ident1(R,NR,Os),
        !.
rem_ident1([],[],_).

rem_ident2([I|R],[Name|NR],Os) :-
```

```
        member(opi(I,Name,_,_,_),Os),
        rem_ident2(R,NR,Os),
        !.
rem_ident2([],[],_).

rem_ident3([t(I,I1)|R],[t(Name,Name1)|NR],Os) :-
        member(opi(I,Name,_,_,_),Os),
        member(opi(I1,Name1,_,_,_),Os),
        rem_ident3(R,NR,Os),
        !.
rem_ident3([],[],_).

rem_ident4([g(P,I)|R],[g(P,Name)|NR],Os) :-
        member(opi(I,Name,_,_,_),Os),
        rem_ident4(R,NR,Os),
        !.
rem_ident4([],[],_).

/************ FM FM FM FM FM FM FM FM FM FM FM FM FM FM ******/
/************ THIS IS THE BOOT FILE FOR FM WITH C-CHUNKS******/

boot(mea,O,E,I) :-
        consult([
        '../lp/fenv',
        '../lp/fmc',
        '../lp/fcm',
        '../lp/fused_bk',
        '../lp/fcomp_bk',
        '../lp/driver_mec',
        '../lp/driver_help',
        '../lp/fchmea_bk',
        '../lp/faugmac',
        '../lp/futile',
        '../lp/fmacgr',
        O,
        E,
        I
                ]).
boot(forward,O,E,I) :-
        consult([
        '../lp/fenv',
        '../lp/fcm',
        '../lp/fuse',
        '../lp/fexh',
        '../lp/driver_f',
        '../lp/driver_help',
        '../lp/fchunk',
        '../lp/faugmac',
        '../lp/futile',
        '../lp/fmacgr',
        O,
        E,
        I
                ]).
```

```
boot(nlp,O,E,I) :-
        consult([
        '../nlp/twd',
        '../nlp/futile_nlp',
        '../nlp/driver_nlp',
        '../nlp/tweak',
        '../nlp/fmacgrd',
        '../nlp/new_fuse',
        O,
        E,
        I,
        '../lp/fcm',
        '../lp/driver_help',
        '../lp/fchunkmea',
        '../lp/faugmac',
        '../lp/fm'
        ]).

:-      consult(options),
        strategy(S),
        operator_file(O),
        environment_file(E),
        init_file(I),
        boot(S,O,E,I),
        frame(name:Environment,type: context,_,_),
        assert(environment(Environment)).
/*

************** fenv fenv fenv fenv fenv fenv fenv *****************

 includes 'agenda' control plus problem step addition */

:- op( 700, xfx, ':').

:- op( 100, xfy, '&').

/* ENVIRONMENTAL PARAMETERS are in 'options' file */

clock(0).               /* increment after each task */
activation(0).          /* increment after each process finishes */
                        /* must be reset by driver */
numberofnodes(0).        /* increment after each node expansion */

/* INITIAL AGENDA (PROCESS AND PASSIVE QUEUES)   */
processq([]).
passiveq([]).
q(_).           /* for flagging active or passive q's */
count(0).       /* used in main loop */
order_store([]).        /* root for fchunkmea's storing of chunk seqs*/


/* MAIN LOOP FOR EVERYTHING */
```

```
go2(_):-processq(X),
        highest(X,d(P,N)),

        length(X,NN),write(' '),write(NN),
        retract(count(C)),C1 is C+1,assert(count(C1)),
        ((C1 > 8,nl,retract(count(_)),assert(count(0))) ; true ),

        removeL(d(P,N),X,X1),
        increment(X1,X2,1),
        retract(processq(_)),
        assertz(processq(X2)),
        start(P),!,fail.
go :- procnumber(X),go2(X). /* go fails when no tasks left */
go :-
        processq([_|_]),         /* something left in proceeq */
        tell(user),
        write('*** maximum no. of activations reached ***'),
        retract(clock(N)),N1 is N+1,assert(clock(N1)),!.
go.

procnumber(X) :-
        increment_activation(X).
procnumber(X) :-
        max_activations(M),
        activation(A),
        A < M,
        procnumber(X).

increment_activation(X) :-
        retract(activation(X1)),
        X is X1+1,
        assert(activation(X)),
        !.

highest([d(P,N)],d(P,N)).
highest([d(P,N)|T],Z) :- highest(T,d(Q,M)),
                        ( (M<N,!,Z=d(P,N)) ;
                          Z=d(Q,M)        ).

increment([d(P,N)],[d(P,N1)],I) :- N1 is N+I.
increment([d(P,N)|T],[d(P,N1)|L],I) :- N1 is N+I,
                                       increment(T,L,I).
increment([],[],_).



start(X) :- call(X),!.
start(_).

    /* note: process assumed not to already exist   */

addprocess(P,N) :-      retract(processq(X)),
                        assertz(processq([d(P,N)|X])).
```

```
addpassive(P,N) :-      retract(passiveq(X)),
                        assertz(passiveq([d(P,N)|X])).

/*****************************************************/



/**********fm fm fm fm fm fm fm fm*****************/
/* definiton of fn to use M.E.A. on a partial soln */
/* which is applied only once to each task         */

mea_step(X) :-
        frame(  name: X,
                type: problem,
                ancest:Ancest,
                context: Context,
                init_world: I,
                goal: G,
                trace: [],
                solution: S),
        del(I,G,G1),      /* only try for unachieved goals */
        /* new active tasks are initiated on only the 1st*/
        /* unachieved goal predicate's op's preconditions*/
        retract(q(_)),assert(q(active)),

/* BOOK KEEPING */
        pushtrace(PT),
        tell(PT),
        writeL([nl,'node name: ',X,' ','goal: ',G,' ancestors:-',nl]),
        wlist(Ancest),nl,
        tell(user),!,
        retract(numberofnodes(Nn)),Nn1 is Nn+1,
        assert(numberofnodes(Nn1)),

        expandmea(X,p(I,[],[]),Lnewps,G1,Context,G1,Ancest),
        success(X,G,Context,Ancest,Lnewps,S),
        ( (not( var(S) ),mac(X)) ; true),/*not varS=success*/
        retract(frame(name:X,type:problem,_,_,_,_,_,_)),
        assertz(frame(name: X,
                type: problem,
                ancest:Ancest,
                context: Context,
                init_world: I,
                goal: G,
                trace: Lnewps,
                solution: S)),!.


/* M.E.A. sort of expansion */

expandmea(Pn,X,Ln,G&G1,Context,Goal,Ancest) :- !,
                expandmea1(Pn,X,Ln1,G,Context,Goal,Ancest),
                expandmea(Pn,X,Ln2,G1,Context,Goal,Ancest),
                append(Ln1,Ln2,Ln).
```

```
expandmea(Pn,X,Ln,G,Context,Goal,Ancest) :-
                expandmea1(Pn,X,Ln,G,Context,Goal,Ancest).

/* Find a set of primitive ops OL whose primary add literal -> this goal
   literal G. Any ops in OL which are applyable are applied. If none then
   then add the OL's to the m.e.a. agenda */

expandmea1(Pn,p(Cstate,_,_),Lnewps,G,Context,Goal,Ancest) :-

                setof(O,nonemp(O,G,Cstate,Context,Goal),OL),
                split(Cstate,OL,Lops,Otherops,Context),
                expmea(Pn,_,Cstate,Lnewps,Lops,Otherops,G,Context,
                                                     Goal,Ancest).
expandmea1(_,_,_,G,_,_,_) :- zzz,write('*op.f.*'),write(G).
zzz.

expmea(Pn,_,Cstate,[],[],Otherops,G,Context,Goal,Ancest) :-
                use_heuristics(Pn,Cstate,G,Context,Otherops,
                                                Otherops1,Ancest),
                addps(Pn,Otherops1,Cstate,Status,Context,G,Goal),
                ( (not(var(Status)),retract(q(_)),assert(q(passive)))
                                                          ;true).
expmea(_,N,Cstate,Lnewps,Lops,_,G,_,_,[a(Pn,_,_)|_]) :-
                 applyops(Cstate,Lops,Lstates),
                 genps(Pn,N,Lops,Lstates,Lnewps,G).
expmea(_,N,Cstate,Lnewps,Lops,_,G,_,_,[]) :-
                 applyops(Cstate,Lops,Lstates),
                 genps(start,N,Lops,Lstates,Lnewps,G).



/* finds an operator which can add a subgoal */

nonemp(O,G,Cstate,Context,_) :-
                frame(  name: O,
                        type: operator,
                        filter: FA,
                        check: Ch,
                        precon:_,
                        padd: A,
                        _,_),
                hold(G,A),      /*  G contained in primary add lit?*/
                hold(FA,Cstate),  /* FILTER */
                frame(name:Context,_,always:Always,_),
                hold(Ch,Always).  /* check this instantiation is poss*/

nonemp(O,G,Cstate,Context,Goal) :-
                usemacros(on),
                eqfirst(G,Goal),
                frame(  name: O,
                        type: operator,
                        macrop: [_|_],
                        check: Ch,
                        precon:P,
```

```
                        padd: A,
                        _,_),
                del(Goal,A,nil),
                frame(name:Context,_,always:Always,_),
                hold(Ch,Always),
                mea_macros(Argnum),
                ( (O =.. SL,length(SL,SL1),SL1 =< Argnum) ;
                hold(P,Cstate) ).
eqfirst(G,G&_).
eqfirst(G,G).
                                        /* to allow macros to be mea'd */
split(_,[],[],[],_).                    /* remove hold(P.. above, and */
                                        /* comment out opposite 'split' */
split(S,[Soh|Sot],[Soh|T],X,Context):-
                frame(  name: Soh,
                        type: operator,
                        macrop: [_|_],
                        _,_,_,_,_),
                Soh =.. SL,length(SL,SL1),
                mea_macros(Argnum),
                SL1 > Argnum,
                split(S,Sot,T,X,Context).
split(S,[Soh|Sot],[Soh|T],X,Context) :-
                frame(  name: Soh,
                        type: operator,
                        filter: _,
                        check: Ch,
                        precon: P,
                        _,_,_),
                frame(name:Context,_,always:Always,_),
                hold(P,S),
                hold(Ch,Always),
                split(S,Sot,T,X,Context).
split(S,[Soh|Sot],X,[Soh|T],Context):-
                split(S,Sot,X,T,Context).


/******************** add desirable op precons as new tasks ***********/
/*******************************************************************/

addps(_,[],_,_,_,_,_).
addps(Pn,[O|T],S,St,Context,G,Goal) :-
                frame(  name: O,
                        type: operator,_,
                        check: C,
                        precon: P,_,_,_),
                frame(name:Context,_,always:Always,_),
                hold(C,Always),         /* instantiation must be valid */
                frame(  name: O,
                        type: operator,_,
                        check: C1,
                        precon: P1,_,_,_),
                rev(Always,Always1),
                hold(C1,Always1),
```

```
                findgoals(Gp1,Gp2,S,P,P1,Pn,G,Goal), /* finds inst. of
                                                       P for O */
                ( (Gp1=nil,Gp2=nil) ; St=added ),
                addpsO(Gp1,P,Pn,O,S,Context,G,Goal),
                addpsO(Gp2,P1,Pn,O,S,Context,G,Goal),
                addps(Pn,T,S,St,Context,G,Goal).
addpsO(nil,_,_,_,_,_,_,_) :- !.
addpsO(_,P,Pn,O,S,Context,G,_Goal) :-
                gensym(aux,A),
                frame(name:Pn,type:problem,ancest:Ancest,_,_,_,_,_),
                asserta(
                        frame(name:A,
                        type:problem,
                        ancest:[a(Pn,O,G)|Ancest],
                        context:Context,
                        init_world:S,
                        goal:P,
                        trace:[],
                        solution:_)
                        ),

                /*record parallel goals to be solved, for incon. checks
                shelve -- must regresss II goals, not store them.
                acculm_goal(Pn,AG),
                del(G,_Goal,Goal1),
                for now just register top level goals (Pn = task..)
                  (name(Pn,[116,97,115,107|_]),
                   ad(AG,Goal1,NewAG) ; NewAG=AG ) ,
                assert(acculm_goal(A,NewAG)),  */

                /*WEAK HEURISTIC  --     favour smaller goalists*/
                del(S,P,PP),andtolist(PP,PPL),length(PPL, Le),
                Interest is 400 - 2*Le,  /*- 2*Le,*/
                q(Status),
                ( (Status=active,addprocess(mea_step(A),Interest)) ;
                  (Status=passive,addpassive(mea_step(A),Interest)) ),
        pushtrace(PT),
        tell(PT),
        writeL([nl,'child name: ',A,' ','priority: ',Interest,'
        status:-',Status,nl]),  tell(user),!.

/* This should really be finding every distinct instantiation of
   Precons that maximally intersects S,& setting Gp =Precons' compliment
   -instead it finds a max of 2 inst'ns by also reversing S.

   To stop the plthora of nodes two weak heuristics are employed:
   1. If an identical predicate has been already mea'd further up
      the tree, then reject the whole  cjn (say Gp.. = nil)

***don't bother--- 2. If the acculmulated parallel goals are in
    consistent with a Gp, then likewise set the Gp to nil */

findgoals(Gp1,Gp2,S,P,P1,Pn,G,_Goal) :-
                del(S,P,Gp1t),
```

```
            rev(S,S1),
            del(S1,P1,Gp2t), /* Gp's are now ground wffs */
            ( (equivc(Gp1t,Gp2t),Gp2=nil) ; true ),
            /* Gp1t=Gp2t --> call one nil so only achieve once */

            ( (circular(Gp1t,Pn,G),Gp1=nil) ; true ),
            /* Gp1t circular --> set Gp1 = nil */

            ( (var(Gp2),circular(Gp2t,Pn,G),Gp2=nil) ; true ),
            /* Gp2 not nil& Gp2t circular --> set Gps = nil */

        /* ( (var(Gp1),incon_goal(Gp1t,Pn,G,_Goal),Gp1=nil) ; true ),
            ( (var(Gp2),incon_goal(Gp2t,Pn,G,_Goal),Gp2=nil) ; true ),
        */

            ( (var(Gp1),Gp1=Gp1t) ; true ),
            ( (var(Gp2),Gp2=Gp2t) ; true ).

/*  shelving............
incon_goal(Gp,Pn,G,Goal) :-
            inconsistent_check(on),
            acculm_goal(Pn,AG),
            !,
            not(AG=nil),
            del(G,Goal,Goal1),
              (name(Pn,[116,97,115,107|_]),
               ad(AG,Goal1,NewAG) ; NewAG=AG ) ,
            inconsistent(Gp,NewAG).
*/
circular(Gp,X,G) :-     /* succeeds if goal is in ancestory */
            frame(name:X,
            type:problem,
            ancest:L,_,_,_,_,_),
            (intersect(G,Gp,_) ; interL(Gp,L) ).
interL(Gp,[a(_,_,G)|T]) :- intersect(Gp,G,_) ; interL(Gp,T).


/*****************************************************************/

/* remove all problem frames whose ancestory contains X's Dad
 as long as they were after the same goal G */

killps(X) :-    frame(  name:X,
                        type: problem,
                        ancest: [a(Dad,_,G)|_],_,_,_,_,_),
                killall(Dad,G,X).
killps(X) :-    frame(  name:X,
                        type: problem,
                        ancest: [],_,_,_,_,_),
                killall(_,_,task1).
killall(Dad,G,X) :-
                frame(  name:Y,
                        type: problem,
                        ancest: L,_,_,_,_,_),
```

```
                member(a(Dad,Z,G),L),
                not( Z = nil), /* deals with case when goal is trying to
                                be acieved from an enhanced state */
                not( X = Y ),
                retract(frame(  name:Y,
                        type: problem,
                        ancest: L,_,_,_,_,_)),
                killps1(Y).
killall(_,_,_).
killps1(Y) :-
                processq(Q),
                passiveq(QQ),
                (
                (removeL(d(mea_step(Y),_),Q,Q1),
                retract(processq(Q)),
                assert(processq(Q1))) ;
                (removeL(d(mea_step(Y),_),QQ,QQ1),
                retract(passiveq(QQ)),
                assert(passiveq(QQ1)))
                ),
                !,fail.

/*****************************************************/
/*  applies specific list of ops to every p.sol. in X */

apply_step(X,L,LN) :-          /* LN is list of task names */
                frame(  name: X,
                        type: problem,
                        ancest:Ancest,
                        context: C,
                        init_world: I,
                        goal: G,
                        trace: P,
                        solution: S),
                apply_all(I,[p(I,[],[])|P],L,LN,Lnewps,C),
                /*(name(X,[97,117,120|_]) ;*/ macros(Lnewps,C,I,G,X) ,
                append(P,Lnewps,P1),
                success(X,G,C,Ancest,Lnewps,S),
                retract(frame(name:X,type:problem,_,_,_,_,_,_)),
                assertz(frame(name: X,
                        type: problem,
                        ancest:Ancest,
                        context: C,
                        init_world: I,
                        goal: G,
                        trace: P1,
                        solution: S)).

apply_all(I,[P|T],L,LN,Lnewps,Context) :-
                apply_1(I,P,L,LN,Ln,Context),
                apply_all(I,T,L,LN,Ln1,Context),
                append(Ln,Ln1,Lnewps).
apply_all(_,[],_,_,[],_).
```

```
apply_1(_,p(St,Op,N),L,LN,[p(Ns1,Opn,N2)],Context) :-
                applyopseq(L,St,Ns,Context),
                last(Ns,Ns1),
                append(Op,L,Opn),
                append(N,LN,N2).

        /* failure of seq. may mean wrong ordering so try other way */

apply_1(I,p(_,Op,N),L,LN,[p(Ns,Opn,N2)],Context) :-
                applyopseq(L,I,NsL1,Context),
                last(NsL1,Ns1),
                applyopseq(Op,Ns1,NsL2,Context),
                last(NsL2,Ns),
                append(L,Op,Opn),
                append(LN,N,N2).

apply_1(_,_,_,_,[],_).



/*****************************************************/
/*  These procedures are shared by both controls       */

success(_,_,_,_,[],_).
success(X,G,_,[],Lnewps,S) :-
                        follows(G,Lnewps,S),
                        killps(X), /* this may not remove  'top level'    */
                                   /* processes so empty processq as well */
                        retract(processq(_)),
                        assert(processq([])),
                        addprocess(status(X),900),
                        addprocess(critic(X),800),
                        write(X),write(' succeeds'),
                        clock(M),macrofade(M).
success(X,G,_,[a(An,nil,_)|_],Lnewps,p(St,Op,N)) :-
                        follows(G,Lnewps,p(St,Op,N)),
                        killps(X),
                        write(X),write(' succeeds'),
                        unlockmea(An),
                        addprocess(apply_step(An,Op,N),800).
success(X,G,_,[a(An,Opr,GG)|_],Lnewps,p(St,Op,N)) :-
                        follows(G,Lnewps,p(St,Op,N)),
                        killps(X),
                        write(X),write(' succeeds'),
                        unlockmea(An),
                        append(Op,[Opr],OL),
                        append(N,[t(An,GG)],NL),
                        addprocess(apply_step(An,OL,NL),800).
success(X,G,C,Ancest,Lnewps,_) :-
              addnewps(X,G,C,Ancest,Lnewps). /* no success,try expanding*/
                                             /* partial solns             */
/* makes a sister process to a successful one active */

unlockmea(An) :-
```

```
                passiveq(Pq),
                member(d(mea_step(X),_),Pq),
                frame(name:X,_,ancest:[a(An,_,G)|_],_,_,_,_,_),
                setof(Z,group(An,G,Z),L),
                swapL(L).
group(An,G,Z) :-
                passiveq(Pq),
                member(d(mea_step(Z),_),Pq),
                frame(name:Z,_,ancest:[a(An,_,G)|_],_,_,_,_,_).
swapL([]).
swapL([H|T]) :-
                retract(passiveq(Pq)),
                removeL(d(mea_step(H),_),Pq,Pq1),
                addprocess(mea_step(H),500),
                assert(passiveq(Pq1)),
                swapL(T).
unlockmea(_).


/* create a new set of tasks that try to reach G from enhanced state */
addnewps(X,G,C,Ancest,[p(S,_,_)|T]) :-
                gensym(aux,A),
                del_cut(S,G,G1),
                not(member(a(_,_,G1),Ancest)),/*no circ goals*/
                /*  acculm_goal(X,AG),
                    asserta(acculm_goal(A,AG)),  -see incon. goal work */
                asserta(
                        frame(name:A,
                        type:problem,
                        ancest:[a(X,nil,G1)|Ancest],
                        context:C,
                        init_world:S,
                        goal:G,
                        trace:[],
                        solution:_)
                        ),
                addprocess(mea_step(A),500),
                addnewps(X,G,C,Ancest,T).
addnewps(X,G,C,Ancest,[_|T]) :-
                addnewps(X,G,C,Ancest,T).
addnewps(_,_,_,_,[]).

macrofade(N) :- macrofade_is(on),
                frame(  name: O,
                        type: operator,
                        macrop: [M|_],
                        _,_,_,_,_),
                N > M-1,
                nl,write(O),write(' gone'),
                retract(frame(  name: O,
                        type: operator,
                        macrop: [M|_],
                        _,_,_,_,_)),
                fail.
```

```
macrofade(_) :- macrofade_is(off).
macrofade(_) :- nl.

follows(G,[p(X,Y,N)|T],S) :-
                                (hold(G,X),
                                S =  p(X,Y,N)) ;
                                follows(G,T,S).

/* put the new nodes into p(,,) format       */

genps(_,_,[],[],[],_).
genps(Y,_,[Op|Opt],[Sr|St],[p(Sr,[Op],[t(Y,G)])|T],G) :-
                        genps(Y,_,Opt,St,T,G).


/*  ordering predicate not used at the moment */
/*  orderg(G&H,G1&G3) :- !,
                maxi(G&H,G1),
                del(G1,G&H,G2),
                orderg(G2,G3).
orderg(G,G).
maxi(G&H,Max) :- !,
                ord(G,N),
                maxi(H,H1),
                ord(H1,M),
                ( (N>M,!, Max = G) ; Max = H1 ).
maxi(G,G). */


/* fcmfcmfcmfcmfcmfcmfcmfcmfcmfcmfcmfcmfcm */

/* this control will have chunks leading the search for a soln */

cea(Goal,I,E) :-
        frame(name:E,_,always:C,_),
        xxx(OL, Goal, I, C),
        /*get_opp(OL,_,I,C,Goal) ,*/
        tell(fred10),write(OL),nl,nl,tell(user).
xxx(OL, Goal, I, C) :-
        setof(O,get_op(O,Goal,I,C),OL).
cea(_,_,_).
get_op(O,Goal,I,E) :-
                ch(_,O,G,P,C),
                del(Goal,G,nil),
                hold(P,I),
                hold(C,E).
/*
get_opp([],[],_,_,_).
get_opp([O|R],[Res|R1],I,E,Goal) :-
                ch(_,O,G,P,C),
                del(Goal,G,nil),
                hold(C,E),
                del(I,P,Res),
                get_opp(R,R1,I,E,Goal).
```

```
*/

/* for a particular problem, store all applicable rules */

make_heurs :-
        init_world(I),
        environment(E),
        frame(name:E,_,_:E1,_),
        delete_old_heuristics,
        tell(fred20),write(I),nl,nl,write(E),nl,
        make_new(I,E1).
delete_old_heuristics :-
        retract(pr(_,_,_,_)),
        fail.
delete_old_heuristics.
make_new(I,E) :-
        ch(Nm,O,G,I1,E1),
        hold(I1,I),
        not(not(hold(E1,E))),
        ( pr(_,G,O,E1) ;
          assert(pr(Nm,G,O,E1)),tell(fred20),nl,write(pr(Nm,G,O,E1)),
          nl,write('from chunk....'),nl,write(ch(Nm,O,G,I1,E1)),nl,nl   ),
        fail.   /* might have to put in a recursive call here instead*/
make_new(_,_) :- tell(fred20),told,!.




/******** fuse fuse fuse fuse fuse fuse fuse   ***************/
/* Version for new-b-chunks post 28/7/88 -no ancestory*/
/* 'A' is superflouous *******/
/* tries to cut down alternative mea operator inst's using chunks;
   cutting down is irrevocabe at the moment although the
   chks are just heuristics. Also, choose the one(s) that
   satisfies most chks in the event of a tie */
use_heuristics(Pn,Cstate,Goal,C,Oin,Oout,_) :-
                                              /* treat macros separately */
        usechunks(on),
        separate(Oin,Mac,Prim),
        chks(Pn,Cstate,Goal,C,Mac,Out1),
        chks(Pn,Cstate,Goal,C,Prim,Out2),
        append(Out1,Out2,Oout).
use_heuristics(Pn,Cstate,Goal,C,Oin,Oin,_) :-
        usechunks(off),
        separate(Oin,Mac,Prim),
        chks(Pn,Cstate,Goal,C,Mac,_),
        chks(Pn,Cstate,Goal,C,Prim,_).

chks(_,_,_,_,Oin,Oin) :- usechunks(off),chunking_is(off).
chks(_,_,_,_,Oin,Oin) :- length(Oin,N),N =< 1.
chks(Pn,_,G,_,Oin,Oin) :-
        not( ch(_,_,_,_,_) ),
        tell(fred5),
```

```
        nl,write(chunk_used(Pn,none,G,Oin,[])),
        nl,tell(user),
        assert(chunk_used(Pn,none,G,Oin,[])).
        /* no chunks abroad */

chks(Pn,Cstate,Goal,C,Oin,Oout) :-
        uc1(Cstate,Goal,C,Oin,Scores,Used_chs),
        max(Scores,M),              /* finds max no. of a list */
        chks2(Pn,Goal,Scores,Oin,Oout,M,Used_chs).

chks2(Pn,G,_,Oin,Oin,0,_) :-
        tell(fred5),
        nl,write(chunk_used(Pn,none,G,Oin,[])),
        nl,tell(user),
        assert(chunk_used(Pn,none,G,Oin,[])),!.
chks2(Pn,G,Scores,Oin,Oout,M,Used_chs2) :-
        collect(Oin,Scores,M,Oout2),
        except_ch(Used_chs2,Oout2,Used_chs2,Oout,Used_chs),
        length(Oin,L1),length(Oout,L2),
        L3 is L1-L2,writeL(['uc',L3,'-',L1,L2,' ']),
        tell(fred5),wlist(Oin),nl,wlist(Oout),tell(user),
        assert(chunk_used(Pn,some,G,Oin,Used_chs)),
        !.

uc1(_,_,_,[],[],[]).
uc1(Cstate,Goal,C,[O|OT],[1|NT],[c(O,Chunk)|Used_chs]) :-
        frame(name:C,_,always:Always,_),
        uc2(Cstate,Goal,C,O,Chunk,Always),
        tell(fred5),nl,write(Chunk),nl,
        write('used on '),write(O),nl,tell(user),
        uc1(Cstate,Goal,C,OT,NT,Used_chs),
        !.
uc1(Cstate,Goal,C,[_|OT],[0|NT],Used_chs) :-
        uc1(Cstate,Goal,C,OT,NT,Used_chs),
        !.
uc2(Cstate,Goal,_,O,ch(Name,O,Goal,S,Ch),Always) :-
        ch(Name,O,Goal,S,Ch),
        hold(S,Cstate),
        hold(Ch,Always),
        !.

/* gets rid of chs that are excepted by others in Used_chs */
except_ch(Used_chs,Oout,Used_chs,Oout,Used_chs) :-
        length(Used_chs,1),
        !.
except_ch(Used_chs,[O|OR],[c(O,ch(N,_,_,_,_))|L],OR1,L1) :-
        ch_ex(N,Ex_ch),
        member(c(_,ch(Ex_ch,_,_,_,_)),Used_chs),
        except_ch(Used_chs,OR,L,OR1,L1).
except_ch(Used_chs,[O|OR],[C|L],[O|OR1],[C|L1]) :-
        except_ch(Used_chs,OR,L,OR1,L1).
except_ch(_,[],[],[],[]) :-!.
```

```
collect([],[],_,[]).
collect([Oin|OL],[Max|NL],Max,[Oin|OL1]) :-   /*save op if its score=Max*/
        collect(OL,NL,Max,OL1).
collect([_|OL],[_|NL],Max,OL1) :-              /*otherwise discard */
        collect(OL,NL,Max,OL1).

separate([],[],[]).
separate([O1|OL],[O1|M],P) :-
        frame(name:O1,_,macrop:[_|_],_,_,_,_,_),
        separate(OL,M,P).
separate([O1|OL],M,[O1|P]) :-
        separate(OL,M,P).


/***   SUBSUMES AND **fcomp*fcomp*fcomp*fcomp****************/
/***   COMPRESSES CHUNKS ****************/

compress(ch(N,O,G,Ps,Pe),Mode) :-
        compress_chunks(off),
        (ch(_,O,G,Ps,Pe) ; subsume(ch(N,O,G,Ps,Pe),Mode)),!.
compress(ch(N,O,G,Ps1,Pe),Mode) :-
        env_axioms(L),
        use_env(Pe,Pe,Pe2,L),
        /* put relns at front ..*/
        pred_ord2(Pe2,Pe4),
        pred_ord3(Pe4,Pe3),
        pred_ord2(Ps1,Ps4),
        pred_ord3(Ps4,Ps),
        remove(nil,Ps,Ps5),
        remove(nil,Pe3,Pe1),
        (ch(_,O,G,Ps5,Pe1) ; subsume(ch(N,O,G,Ps5,Pe1),Mode)),!.

pred_ord3(Pe,X&Pe1) :-
        remove_bt(X,Pe,Pe3),
        X =.. Y,
        length(Y,L),
        L > 3,
        pred_ord3(Pe3,Pe1),!.
pred_ord3(Pe,Pe) :- !.
pred_ord2(Pe,X&Pe1) :-
        remove_bt(X,Pe,Pe3),
        X =.. Y,
        length(Y,L),
        L > 2,
        pred_ord2(Pe3,Pe1),!.
pred_ord2(Pe,Pe) :- !.

get_chunks([ch(A,S,D,F,G)|C]) :-
        retract(ch(A,S,D,F,G)),
        get_chunks(C),!.
get_chunks([]) :- !.

use_env(P&Pe,Pe1,PeRes,L) :-
        not( not( env_follows(P,Pe1,L) ) ),
```

```
        rem_equiv(P,Pe1,Pe2),
        use_env(Pe,Pe2,PeRes,L),
        !.
use_env(P&Pe,Pe1,P&PeRes,L) :-
        use_env(Pe,Pe1,PeRes,L), !.
use_env(P,Pe,nil,L) :-
        not( not( env_follows(P,Pe,L) ) ), !.
use_env(P,_,P,_) :- !.

env_follows(P,Pstart,L) :-
        numvars(Pstart,1,_),
        env_follows1(P,Pstart,L).
env_follows1(_,_,[]) :- !,fail.
env_follows1(P,Pstart,[[A,P]|_]) :-
        hold(A,Pstart),!.
env_follows1(P,Pstart,[_|Rest]) :-
        env_follows1(P,Pstart,Rest).

/**************************************/
/*will have to sort out 'Mode' before using again..*/
subsume(ch(N,A,B,C,D),M) :-
        subsume_chunks(off),
        chunk_assert(ch(N,A,B,C,D),M),!.
subsume(ch(N,A,B,C,D),M) :-
        not(ch(_,_,_,_,_)),     /* no chunks made */
        chunk_assert(ch(N,A,B,C,D),M),!.

subsume(ch(N,A,B,C,D),_) :-
        get_chunks(Cs),
        subsume1([ch(N,A,B,C,D)],Cs),
        subsume_old(ch(N,A,B,C,D),Cs),
        !.

subsume_old(ch(N,A,B,C,D),Cs) :-
        ch(N,_,_,_,_),
        subsume1(Cs,[ch(N,A,B,C,D)]).
subsume_old(_,Cs) :-
        put_chunks(Cs).


put_chunks([ch(A,S,D,F,G)|C]) :-
        asserta(ch(A,S,D,F,G)),
        put_chunks(C),!.
put_chunks([]) :- !.
get_chunks([ch(A,S,D,F,G)|C]) :-
        retract(ch(A,S,D,F,G)),
        get_chunks(C),!.
get_chunks([]) :- !.

subsume1([C1|C2],C) :-
        not( subsumed(C1,C)),
        chunk_assert(C1),
        subsume1(C2,C).
subsume1([_|C2],C) :-
```

```
        subsume1(C2,C).
subsume1([],_) :- !.

subsumed(ch(N,O,G,Ps,Pe),C) :-
        numvars(ch(N,O,G,Ps,Pe),1,_),
        exp_env(Pe,Pe1),
        adcut(Ps,Pe1,PP),
        subsumed1(N,O,G,PP,C),
        !.

subsumed1(Name,O,G,C1,[ch(N,O,G,Ps,Pe)|_]) :-
        ch_record(Name,_,S1),
        ch_record(N,_,S2),
        S2 > S1,
        adcut(Ps,Pe,PP),
        del(C1,PP,nil), /* PP is more gen than C1 */
        write(N),
        write(' has subsumed '),write(Name),nl,
        tell(fred21),nl,write(N),
        write(' has subsumed '),write(Name),nl,tell(user),!.
subsumed1(Name,O,G,C1,[_|CL]) :-
        subsumed1(Name,O,G,C1,CL).




chunk_assert(_,non_discrim) :-
        retract(weak_ch(N,A,B,C,D)),
        retract(ch_record(N,N1,N2,_)),
        tell(user),write(N),write(' failed to strengthen'),nl,
        /* chunk has been sent for strengthening..*/
        /* ..its done no good */
        assert(ch(N,A,B,C,D)),
        assert(ch_record(N,N1,N2,multiples)),
        !.
chunk_assert(_,multiples) :-
        retract(weak_ch(N,A,B,C,D)),
        retract(ch_record(N,N1,N2,_)),
        tell(user),write(N),write(' failed to strengthen'),nl,
        /* chunk has been sent for strengthening..*/
        /* ..its done no good */
        assert(ch(N,A,B,C,D)),
        assert(ch_record(N,N1,N2,multiples)),
        !.
chunk_assert(ch(N,A,B,C,D),discrim) :-
        retract(weak_ch(_,_,_,_,_)),
        /* chunk has been sent for strengthening..*/
        retr_ex(N),
        retract(ch_record(N,_,_,_)),
        current_num(task,Num),
        gensym(ch,New),
        assert(ch_record(New,Num,0,strengthened)),
        write(N),write(' strengthened to '),write(New),nl,
        assert(ch(New,A,B,C,D)).
```

```
retr_ex(N) :- retract(ch_ex(N,_)),fail.
retr_ex(N) :- retract(ch_ex(_,N)),fail.
retr_ex(_).

        /*usual case .. */
chunk_assert(ch(N,A,B,C,D),M) :-
        max_chain_size(N1),
        not(big_chains(D,N1)),
        assert(ch(N,A,B,C,D)),
        write(N),write(' made'),nl,
        current_num(task,Num),
        assert(ch_record(N,Num,0,M)),
        !.



chunk_assert(ch(N,_,_,_,_),_) :-
        nl,write(N),write(' chains too big'),nl.

big_chains(D,N) :-
        andtolist(D,DL),
        rem_all(ne(_,_),DL,DL1),
        rem_all(type_of(_,_),DL1,DL2),
        big_c(DL2,N).
big_c(D,N) :-
        member(DM,D),
        count_inst(DM,D,C),
        C > N.

rem_all(X,DL,DL1) :-
        removeL(X,DL,DL2),
        rem_all(X,DL2,DL1),!.
rem_all(_,DL,DL).

count_inst(DM,[DM|DR],C) :-
        count_inst(DM,DR,C1),
        C is C1+1,!.
count_inst(DM,[_|DR],C) :-
        count_inst(DM,DR,C),!.
count_inst(_,_,0).




/************* drive driver driver driver driver driver *****************/

/* This drives the simulation of a robot environment */
/* version with new chunks */

b :-
        environment(C),
        init_world(I),nl,
        write('My environment is called '),write(C),nl,
```

```
        write('My current world is '),nl,write(I),nl,
        task(C,I).
bfile :- environment(C),init_world(I),nl,
        write('My environment is called '),write(C),nl,
        write('My current world is '),nl,write(I),nl,
        retract(numberofnodes(_)),
        assert(numberofnodes(0)),
        task_file(FI),see(FI),read(G),task1(C,I,G),bfile.
bfile.

task(C,I) :-
                retract(numberofnodes(_)),
                assert(numberofnodes(0)),
                nl,write('Enter task or "h" for help>'),
                read(G),
                help(G,C,I). /* see driver_help */


task1(_,_,end_of_file) :-
                task_file(FI),see(FI),seen,!,fail.
task1(C,I,G) :-
                gensym(task,T),
                assert(
                        frame(  name: T,
                                type: problem,
                                ancest: [],
                                context: C,
                                init_world: I,
                                goal: G,
                                trace:[],
                                solution: _)
                        ),
                retract(processq(_)),
                assert(processq([d(mea_step(T),1000)])),
assert(acculm_goal(T,nil)),
                !,go,
abolish(acculm_goal,2),
                numberofnodes(Nn),
                retract(activation(_)),
                assert(activation(0)),
                writeL([nl,'no. of expanded nodes: ',Nn,nl]),

                tell(fred100),
                writeL([nl,'no. of expanded nodes: ',Nn,nl]),
                frame(  name: T,
                        type: problem,
                        ancest: _,
                        context: C,
                        init_world: _,
                        goal: G,
                        trace:_,
                        solution:p(S,Om,Tr)),
                length(Om,OmL),
                task_file(FI),
```

```
                writeL(['tfile: ',FI,' no: ',T,' of length ',OmL]),nl,
                writeL(['goal: ',G,' by: ',Om,' trace ',Tr]),
                tell(fred5),writeL([T,' over ************']),tell(user),

                not(var(S)),
                write('task finished'),nl,
                (print_stats ; true ),  /* true for C-prolog */
                new_chunk(I,Om,Tr,G,C),
                replacemacs(Om,Omm,C),
                write('By sequence of operators '),nl,nl,write(Omm),nl,
                write('goal '),write(G),write(' is satisfied, state is'),
                nl,nl,write(S),nl,
                retract(init_world(_)),
                assert(init_world(S)),
                (print_stats ; true ).  /* true for C-prolog */

print_stats :-
        statistics(runtime,[_,CP]), CPused is fix(CP/100),
        tell(fred100),writeL(['CPUsed=',CPused,' secs ']),nl,tell(user),
        writeL(['CPUsed=',CPused,' secs ']),nl.
/*
        statistics(runtime,[_,CP]), CPused is fix(CP/100),
        statistics(clause_store,[X1,X2]), C is fix(100*(X2/X1)),
        statistics(global_stack,[X3,X4]), G is fix(100*(X4/X3)),
        statistics(local_stack,[X5,X6]), L is fix(100*(X6/X5)),
        statistics(trail,[X7,X8]), T is fix(100*(X8/X7)),
        writeL(['CPUsed=',CPused,' secs, Space left: ']),
        tell(fred100),writeL(['CPUsed=',CPused,' secs ']),nl,tell(user),
        writeL(['C=',C,'% G=',G,'% L=',L,'% T=',T,'%',nl]).
*/

task1(_,_,_) :- see(user),nl,nl,write('**TASK FAILURE**'),nl,nl,
                storefs(fred4,problem,8),see(user),
                write('see fred4 for problem dump').


/*************  driver_help driver_help driver_help *****************/

help(h,C,_) :-
        gather(Pads,Ens),
        write_info(C,Pads,Ens),
        !,
        b.
help(end_of_file,C,I) :-
        task1(C,I,end_of_file).

help(G,_,I) :-
        hold(G,I),
        nl,write(G),write('is already true!'),nl,nl,
        !,
        b.

help(G,C,_) :-
        not(poss_achieve(G,C)),
```

```
        gather(Pads,Ens),
        writeL([nl,'"',G,'"',' is not a valid task!',nl,nl]),
        write_info(C,Pads,Ens),
        !,
        b.

help(Gexp,C,I) :-
        /* exp_state(G,Gexp),*/
        nl,write(Gexp),nl,
        !,task1(C,I,Gexp).  /* TASK1 IS IN ALL DRIVERS */

write_info(C,Pads,Ens) :-
        write('Tasks may be any conjunct of .. '),nl,nl,
        andtolist(Pads,L),
        wlist(L),nl,
        write('with environmental restrictions including.. '),nl,nl,
        andtolist(Ens,EnsL),
        select_write(EnsL),
        frame(name:C,_,always:Always,_),
        writeL([' etc .. etc .. and environment ',Always,nl]).


select_write([type_of(A,B)|T]) :-
        writeL(['type of ',A,' is ',B,';  ']),
        select_write(T).
select_write([_|T]) :-
        select_write(T).
select_write([]) :- nl.




poss_achieve(nil,_) :- !.
poss_achieve(G,C) :-
        frame(_,_,_,check:W,_,padd:X,_,_),
        frame(_,_,always:Always,_),
        del(X,G,SmallerG),
        not(SmallerG = G),
        hold(W,Always),
        poss_achieve(SmallerG,C).
poss_achieve(nil,_).

gather(Pads,Ens) :-
        gathered(Pads,Ens),
        !.
gather(Pads,Ens) :-
        setof(X,collect(X),PL),
        numvars(PL,1,_),
        gather1(PL,Pads,Ens),
        assert(gathered(Pads,Ens)),
        !.
collect(f(W,X)) :-
        frame(_,_,_,check:W,_,padd:X,_,_).
```

```
gather1([f(E,P)|T],PA0,PA1) :-
        gather1(T,PA2,PA3),
        ad(P,PA2,PA0),
        ad(E,PA3,PA1).
gather1([],nil,nil).


/******************************************************************/

 /******** fchmea fchmea fchmea fchmea fchmea ***********/
/* This contains the new (***post 8/88**) problem solution chunker for
backward search methods.*/


new_chunk(I,OL,TL,G,C) :-
        chunking_is(on),
        length(OL,Le),Le > 1,
        tell(user),nl,write('..chunking..'),nl,
        new_chunk00(I,OL,TL,G,C),
        tell(user),nl,write('chunking finished'),nl,
        remove_ch_used,!.
new_chunk(_,_,_,_,_).
remove_ch_used :- retract(chunk_used(_,_,_,_,_)),fail.
remove_ch_used.

new_chunk00(I,OL,TL,G&GR,C) :-
        andtolist(G&GR,GL),
        member(t(_,G1),TL),
        member(G1,GL),
        removeL(G1,GL,GL1),
        listtoand(GL1,GA),
        split_up_to(TL,t(_,G1),TL1,TL2),
        length(TL1,LN),
        split_to_no(OL,LN,OL1,OL2),
        new_chunk0(I,OL1,TL1,C,IN),
        new_chunk00(IN,OL2,TL2,GA,C).
new_chunk00(_,[],[],_,_) :- !.
new_chunk00(I,OL,TL,G,C) :-
        not(G = _&_),
        new_chunk0(I,OL,TL,C,_).
new_chunk00(_,_,_,_,_) :- write('**exception in new chunk00**'),!.



new_chunk0(I,OL,TL,C,LS) :-
                applyopseq1(OL,I,SL),
                removelast1(SL,SL1,LS),
                reverse(OL,REV_O),
                reverse(TL,REV_TL),
                reverse([I|SL1],REV_S),
                get_Len(REV_O,REV_S,C,REV_E,REV_P),
        new_chunk1(C,I,REV_O,REV_TL,REV_S,REV_E,REV_P).
```

```
new_chunk1(_,_,[_],_,_,_,_) :- !.
new_chunk1(C,IS,[O|OL],[T|TL],[S|SL],[E|EL],[P|PL]) :-
        new_chunk2(C,IS,[O|OL],T,[S|SL],[E|EL],[P|PL]),
        !,
        new_chunk1(C,IS,OL,TL,SL,EL,PL).
new_chunk1(C,IS,[_|OL],[_|TL],[_|SL],[_|EL],[_|PL]) :-
        new_chunk1(C,IS,OL,TL,SL,EL,PL).




new_chunk2(C,IS,[O|OL],t(T,G),[S|SL],[Check|EL],[PP1|PL]) :-
                                /* some chs used */
        chunk_used(T,some,G,_,CL),
        member(c(O,_),CL),
        removeL(c(O,ch(N,_,_,_,_)),CL,CL1),
        !,
        retract(ch_record(N,TC,Score,M)),
        Score1 is Score + 10,
        assert(ch_record(N,TC,Score1,M)),
        put_excep(N,CL1),
        strengthen(CL1,N,C,IS,[O|OL],t(T,G),[S|SL],[Check|EL],[PP1|PL]).

new_chunk2(C,IS,[O|OL],t(T,G),[S|SL],[Check|EL],[PP1|PL]) :-
        /* this part finds P = {Ps in WP1:PP&E(~PP1 in preds) => Ps } */
        chunk_used(T,none,G,Ochoices,_),
        length(Ochoices,LOch),LOch > 1,
        member(O,Ochoices),
        getprec(G,[O|OL],[S|SL],C,WP1is,_,ConstsL),
        list_to_set(ConstsL,ConstL),
        (intersection(PP1,WP1is,P) ; P = nil ),
        generalise(O,ConstL,Og),
        generalise(G,ConstL,Gg),
        generalise(P,ConstL,Pg),
        generalise(Check,ConstL,Checkg),!,
        andtolist(Check,ChL),         /* this part ensures min. genn */
        generaliseL(ConstL,ChL,_,Cv),/* by adding not eq. literals to */
        list_to_set(Cv,Cv1),          /* the check literals        */
        add_ne(Cv1,Ne1),
        slim(Ne1,Ne2), /* del. some ne's */
        remove(nil,Ne2,Ne),
        ad(Checkg,Ne,Ch1),
        gensym(ch,CH),
        tell(fred2),writeL([ch(CH,Og,Gg,Pg,Ch1),'. ']),told,
        tell(fred21),
        writeL([ch(CH,Og,Gg,Pg,Ch1),'. ']),nl,nl,tell(user),
        see(fred2),
        read(ch(CH,O1,G1,P1,Check1)),
        seen,
        assert(poss_ch(CH,O1,G1,P1,Check1)),
    (discrim(Ochoices,[Check|EL],nil,ConstL,IS,WP1is,G,C,O,Check,PP1,P);
     ndiscrim(Ochoices,EL,PL,Check,PP1,ConstL,IS,WP1is,G,C,O)),
        !.
new_chunk2(C,IS,[O|OL],t(T,G),[S|SL],[Check|EL],[PP1|PL]) :-
```

```
        /* this part finds P = {Ps in WP1:PP&E(~PP1 in preds) => Ps } */
        use_chunks(off), /* in case we have bad chunk */
        chunk_used(T,some,G,Ochoices,BadC),
        length(Ochoices,LOch),LOch > 1,
        member(O,Ochoices),
        getprec(G,[O|OL],[S|SL],C,WP1is,_,ConstsL),
        list_to_set(ConstsL,ConstL),
        (intersection(PP1,WP1is,P) ; P = nil ),
        generalise(O,ConstL,Og),
        generalise(G,ConstL,Gg),
        generalise(P,ConstL,Pg),
        generalise(Check,ConstL,Checkg),!,
        andtolist(Check,ChL),         /* this part ensures min. genn */
        generaliseL(ConstL,ChL,_,Cv),/* by adding not eq. literals to */
        list_to_set(Cv,Cv1),          /* the check literals        */
        add_ne(Cv1,Ne1),
        slim(Ne1,Ne2), /* del. some ne's */
        remove(nil,Ne2,Ne),
        ad(Checkg,Ne,Ch1),
        gensym(ch,CH),
        tell(fred2),writeL([ch(CH,Og,Gg,Pg,Ch1),'. ']),told,
        tell(fred21),
        writeL([ch(CH,Og,Gg,Pg,Ch1),'. ']),nl,nl,tell(user),
        see(fred2),
        read(ch(CH,O1,G1,P1,Check1)),
        seen,
        assert(poss_ch(CH,O1,G1,P1,Check1)),
        (discrim(Ochoices,[Check|EL],nil,ConstL,IS,WP1is,G,C,O,Check,PP1,P);
        ndiscrim(Ochoices,EL,PL,Check,PP1,ConstL,IS,WP1is,G,C,O)),
        tell(fred100),nl,write('**BAD CH '),
        write(BadC),nl,write(CH),nl,tell(user),
        put_excep(CH,BadC),
        !.

strengthen([],_,_,_,_,_,_,_,_).
strengthen(CL1,N,_,_,_,_,_,_,_) :- /* only one choice-chunk--*/
        not(member(c(_,ch(N,_,_,_,_)),CL1)).
strengthen(CL1,N,C,IS,[O|OL],t(_,G),[S|SL],[Check|EL],[PP1|PL]) :-
        strengthen_chunks(on),
        not(ch_record(N,_,_,strengthened)),
        not(ch_record(N,_,_,multiples)),
        ch(N,O1,G1,P1,Check1),
        O = O1,
        frame(name:C,_,always:EN,_), /* vars not already done by  */
        hold(P1,IS),
        hold(Check1,EN),
        get_chN(CL1,N,Och1),
        getprec(G,[O|OL],[S|SL],C,WP1is,_,ConstsL),
        list_to_set(ConstsL,ConstL),
        (intersection(PP1,WP1is,P) ; P = nil ),
        retract(ch(N,O1,G1,P1,Check1)),
        assert(poss_ch(N,O1,G1,P1,Check1)),
        assert(weak_ch(N,O1,G1,P1,Check1)),
    (discrim([O|Och1],[Check|EL],nil,ConstL,IS,WP1is,G,C,O,Check,PP1,P);
```

```
      ndiscrim([O|Och1],EL,PL,Check1,P1,ConstL,IS,WP1is,G,C,O)),
        !.
strengthen(_,_,_,_,_,_,_,_,_).

put_excep(_,[]).
put_excep(N,[c(_,ch(N1,_,_,_,_))|CL]) :- /* alredy exception */
        ch_ex(N1,N),
        put_excep(N,CL).
put_excep(N,[c(_,ch(N1,_,_,_,_))|CL]) :- /* retract if circular */
        retract(ch_ex(N,N1)),
        put_excep(N,CL).
put_excep(N,[c(_,ch(N,_,_,_,_))|CL]) :- /* will strengthen ..*/
        put_excep(N,CL).
put_excep(N,[c(_,ch(N1,_,_,_,_))|CL]) :-
        assert(ch_ex(N1,N)),write(ch_ex(N1,N)),nl,
        put_excep(N,CL).

get_chN([c(O,ch(N,_,_,_,_))|R],N,[O|Och1]) :-
        get_chN(R,N,Och1),!.
get_chN([_|R],N,Och1) :-
        get_chN(R,N,Och1),!.
get_chN([],_,[]).


get_Len([O1|T],[S|T1],C,[Ch1|Ch2],[Pr1|PrR]) :-

        frame(name:O1,_,_,check:Ch1,precon:Pr,_,_,_),
        hold(Pr,S),                    /* These 3 lines instantiate */
        frame(name:C,_,always:A,_), /* vars not already done by  */
        hold(Ch1,A),                   /* op parameter instants     */
        exp_state(Pr,Pr1),

        get_Len(T,T1,C,Ch2,PrR).

get_Len([],_,_,[],[]) :- !.


/* nfaugmac ****nfaugmac ****nfaugmac ****  */

/* augments a macro precon. */

/**************************************************************/
/* This part find whether the formed chunk needs strengthening */
/* NOTE: post 8/88: -- WPi/WPiI are just Oi.pre/Oi.preI */


ndiscrim(_,_,_,_,_,_,_,_,_,_,_) :-
        not(poss_ch(_,_,_,_,_)),!.
/* acculm version +endbit (in case of 2=paths)*/
ndiscrim(OSL,[E1],[P1],EA,PA,ConstL,IS,WP1,G,C,O) :-
        adcut(E1,EA,EAA),
        adcut(P1,PA,PAA),
        discrim1(EAA,PAA,ConstL,IS,WP1,G,C,O,OSL),
        discrim3(EAA,PAA,ConstL,IS,WP1,G,C,O,OSL).
```

```
ndiscrim(OSL,[E1|ER],[P1|PR],EA,PA,ConstL,IS,WP1,G,C,O) :-
        adcut(E1,EA,EAA),
        adcut(P1,PA,PAA),
        discrim1(EAA,PAA,ConstL,IS,WP1,G,C,O,OSL),
        ndiscrim(OSL,ER,PR,EAA,PAA,ConstL,IS,WP1,G,C,O).

discrim1(EAA,PAA,ConstL,IS,WP1,G,C,O,OSL) :-
        one_uc1(IS,G,C,OSL,ScoreL),
        sum(ScoreL,N),
        discrim2(N,EAA,PAA,ConstL,WP1,G,C,O).

discrim2(1,_,_,_,_,_,_,_) :-
        retract(poss_ch(Nm,A1,A2,A3,A4)),
        compress(ch(Nm,A1,A2,A3,A4),discrim),!. /* at last chunk is made..*/
discrim2(_,EAA,PAA,ConstL,WP1,G,C,O) :-
        new_aug_chunk(EAA,PAA,ConstL,WP1,G,C,O).


discrim3(_,_,_,_,_,_,_,_,_) :-
        not(poss_ch(_,_,_,_,_)),!.
discrim3(_,_,_,_,_,_,_,_,_) :- /* get here - non-disc chunk */
        retract(poss_ch(Nm,A1,A2,A3,A4)),
        compress(ch(Nm,A1,A2,A3,A4),multiples),!.


one_uc1(_,_,_,[],[]) :- !.
one_uc1(Cstate,Goal,C,[O|OT],[N|NT]) :-
        frame(name:C,_,always:Always,_),
        (       (one_uc2(Cstate,Goal,O,Always), N=1)
                ; N=0       ),
        one_uc1(Cstate,Goal,C,OT,NT),!.

one_uc2(Cstate,Goal,O,Always) :-
        poss_ch(_,O,Goal,S,Ch), /*MAY CHANGE TO del(G,Goal..*/
        hold(S,Cstate),
        hold(Ch,Always).

/*************************************************************/
/* Note: WPiI is simO in IJCAI paper or C(i) in ML one */

new_aug_chunk(EAA,PAA,ConstL,WP1,G,_,O) :-


        (intersection(PAA,WP1,P) ; P = nil ),!,
        get_terms(P,TL1), /* TL1 = list of terms in P */
        list_to_set(TL1,TL),  /* get rid of dupes */
        get_preds_with_terms(TL,EAA,ER),

        generalise(O,ConstL,Og),
        generalise(G,ConstL,Gg),
        generalise(P,ConstL,Pg),
        generalise(ER,ConstL,Checkg),!,

        andtolist(ER,ChL),      /* this part ensures minimal gen'n */
```

```
        generaliseL(ConstL,ChL,_,Cv),/* by adding 'ne' literals to */
        list_to_set(Cv,Cv1),    /* the 'check' literals            */
        add_ne(Cv1,Ne1),
        slim(Ne1,Ne2), /*DIRTY way of del'ing some ne's */
        remove(nil,Ne2,Ne),
        ad(Checkg,Ne,Ch1),
        retract(poss_ch(Nm,_,_,_,_)),

        tell(fred2),writeL([ch(Nm,Og,Gg,Pg,Ch1),'. ']),told,
        see(fred2),read(ch(Nm,Og1,Gg1,Pg1,Checkg1)),seen,
        tell(fred21),write('enhanced chunk:-'),nl,
        writeL([ch(Nm,Og,Gg,Pg,Ch1),'. ']),nl,nl,tell(user),
        assert(poss_ch(Nm,Og1,Gg1,Pg1,Checkg1)),!.


get_terms(X&Y,T) :-
        X =.. [_|TL],
        get_terms(Y,T1),
        append(TL,T1,T).
get_terms(X,T) :-
        X =.. [_|T].

get_preds_with_terms(TL,E&A,E&R) :-
        get_terms(E,ET),
        member(M,ET),
        member(M,TL),
        get_preds_with_terms(TL,A,R),!.
get_preds_with_terms(TL,_&A,R) :-
        get_preds_with_terms(TL,A,R).
get_preds_with_terms(TL,E,E) :-
        get_terms(E,ET),
        member(M,ET),
        member(M,TL),!.
get_preds_with_terms(_,_,nil).


/* *************** futile futile futile futile ************** */

/* utility fns-used by more than one part +++++++++ */
not(X) :- X,!,fail.
not(_).
/* write out list of terms */
writeL([nl|Y]) :- nl,writeL(Y).
writeL([X|Y]) :- write(X),writeL(Y).
writeL([]) :- !.

/* call list of goals */
call_list([C1|T]) :- call(C1),call_list(T).
call_list([]).

append([],L,L).
append([H|T],L,[H|Z]) :- append(T,L,Z).

sum([],0) :- !.
```

```
sum([H|T],N) :- sum(T,N1),N is N1 + H.

intersect(S1,S2,X) :- elem(X,S1),elem(X,S2).
intersection(X,Y,I) :- setof(I1,intersect(X,Y,I1),L),listtoand(L,I).

i([],_,[]) :- !.          /* intersection for lists */
i([E|T],Y,[E|T1]) :-      /* NOTE: error.. e.g. i([x,y],[x],[]) succeeds!!*/
                    member(E,Y),
                    !,
                    i(T,Y,T1).
i([_|T],Y,Z) :- i(T,Y,Z).

i_equiv([],_) :- fail,!.     /* succeeds if there's a common equiv mem*/
i_equiv([E|_],Y) :-
                    eq_member(E,Y),
                    !.
i_equiv([_|T],Y) :- i_equiv(T,Y).

eq_member(E,[Y|_]) :-
         E == Y,!.
eq_member(E,[_|T]) :-
        eq_member(E,T),!.


listtoand([],nil)./* could just have one of these but ordering matters */
listtoand([X],X) :- !.
listtoand([X|Y],X&T) :- !,listtoand(Y,T).
andtolist(nil,[]).
andtolist(X&Y,[X|Z]) :-!, andtolist(Y,Z).
andtolist(X,[X]) :- !.

/* removes 1st occurence of a list el */
/* fails if no occurence */

removeL(El,[El|T],T) :- !.
removeL(E1,[X|T1],[X|T2]) :- removeL(E1,T1,T2).

removelast([_],[]) :- !.
removelast([X|X1],[X|Y1]) :- removelast(X1,Y1).
removelast1([L],[],L) :- !.
removelast1([X|X1],[X|Y1],L) :- removelast1(X1,Y1,L).

/* list utilities specific to 'fchmea' */
split_up_to([t(_,G)|TLR],t(_,G),[t(_,G)],TLR) :- !.
split_up_to([X|TLR],t(_,G),[X|TL1],TL2) :-
        split_up_to(TLR,t(_,G),TL1,TL2).

split_to_no(OL,0,[],OL) :- !.
split_to_no([O|OL],N,[O|OL1],OL2) :-
        N1 is N-1,
        split_to_no(OL,N1,OL1,OL2).

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).
```

```
/* finds max in a List of nos, fails if empty list */
max([N],N) :- !.
max([N|NL],M) :- max(NL,M1),( (N>M1,N=M) ; (M1=M) ).

/* 'i' has already been written */
interL([E|X],Y,[E|Z]) :- member(E,Y),interL(X,Y,Z).
interL([],_,[]).

elem(X,Y&_):- elem(X,Y).
elem(X,_&C):-!,elem(X,C).
elem(X,X).

/* fails if P is not == to a P1 in arg2 */
rem_equiv(P,P1&Pr,Pr) :-
        P == P1,
        !.
rem_equiv(P,P1&Pr,P1&Pr1) :-
        rem_equiv(P,Pr,Pr1).

rev(X,Y) :- rev_x(X,nil,Y1),remove(nil,Y1,Y),!.
rev_x(X&Y,C,R) :- rev_x(Y,X&C,R).
rev_x(X,C,X&C).


reverse(X,Y) :- reverse_x(X,[],Y),!.
reverse_x([],C,C).
reverse_x([H|T],C,R) :- reverse_x(T,[H|C],R).


/* checks equiv of 2 ground &-exps */

equivc(X,Y) :- del(X,Y,nil),!,del(X,Y,nil),!.

/* checks to see whether 2 &exps are inconsistent */

inconsistent(G&_,G1) :-
        inconsist(G,GP), hold(GP,G1),!.
inconsistent(_&Gg,G1) :-
        inconsistent(Gg,G1).
inconsistent(G,G1) :-
        inconsist(G,GP), hold(GP,G1),!.

vars_in(X&Y) :- ((X =.. Z,vars_inl(Z)) ; vars_in(Y)),!.
vars_in(X) :- X =.. Z,vars_inl(Z).
vars_inl([H|T]) :- ( var(H) ; vars_inl(T) ).

/* pre: ithlist(i,j,x,y) input i=pos. of first in list, y is el. of x*/
ithlist(I,I,[X|_],X) :- !.
ithlist(I,J,[_|L],X) :- I1 is I+1,ithlist(I1,J,L,X).
/* post: ithlist(i,j,x,y) ouput j = pos(y in x) + i -1 */


/* This procedure changes a list to a set */
```

```
list_to_set([],[]) :- !.
list_to_set([L|T],S) :- member(L,T),!,list_to_set(T,S).
list_to_set([L|T],[L|T1]) :- list_to_set(T,T1).

/* numbervars */

numvars(A,B,C) :- numvars1(A,B,C),!.
numvars1(x(N),N,N1) :-
        N1 is N+1.
numvars1(Term,N1,N2) :-
        nonvar(Term),
        functor(Term,_,N),
        numvars1(0,N,Term,N1,N2).
numvars1(N,N,_,N1,N1).
numvars1(I,N,Term,N1,N3) :-
        I < N,
        I1 is I+1,
        arg(I1,Term,Arg),
        numvars1(Arg,N1,N2),
        numvars1(I1,N,Term,N2,N3).

/* generate symbol predicate */

gensym(Root,Atom) :-
                        getnum(Root,Num),
                        name(Root,Name1),
                        name(Num,Name2),
                        append(Name1,Name2,Name),
                        name(Atom,Name).

getnum(Root,Num) :-
                        retract(current_num(Root,Num1)),!,
                        Num is Num1+1,
                        asserta(current_num(Root,Num)).

getnum(Root,1) :- asserta(current_num(Root,1)).


/********* finds all applicable operators in state S */

findops(Cstate,Lops,Context) :- setof(O,f(O,Cstate,Context),Lops).

f(O,S,Context) :- usemacros(off),
                  frame(  name: O,
                          type: operator,
                          filter:_,
                          check: Ch,
                          precon: P,
                          padd: _,
                          _,_),
          hold(P,S),
          frame(name:Context,_,always:Always,_),
```

```
          hold(Ch,Always).
f(O,S,Context) :- usemacros(on),
                frame(  name: O,
                        type: operator,
                        _,
                        check: Ch,
                        precon: P,
                        padd: _,
                        _,_),
          hold(P,S),
          frame(name:Context,_,always:Always,_),
          hold(Ch,Always).


/* finds instantiations of <arg1>& S s.t. <arg1> follows from S -
   or if it is always true in that context. On bactracking this
   will try for an alternate intantiation of course, but note that
   the most general S will not nec. appear first !!              */

hold(nil&X,Y) :- hold(X,Y).   /* nil IS REALLY TRUE  */
hold(nil,_).

hold(ne(U,V)&Y,S) :-!, not(U == V),hold(Y,S).
hold(ne(U,V),_) :-!, not(U == V).
hold(X&Y,S) :- elem(X,S),hold(Y,S).
hold(X,S) :- elem(X,S).

/********* applies a list of ops to create a seq of states */
/* Note seq. will fail if unapplicable or one of addlists is superfluous*/

applyopseq([Op|T1],S,[S1|T2],Context) :-   !,f(Op,S,Context),
                                     applyop(S,Op,S1),!,
                                     applyopseq(T1,S1,T2,Context).
applyopseq([],_,[],_) :- !.

/********* applies a list of ops to create a seq of states */
/* as above but no precondition checking*/

applyopseq1([Op|T1],S,[S1|T2]) :-
                                     applyop(S,Op,S1),
                                     applyopseq1(T1,S1,T2).
applyopseq1([],_,[]) :- !.

/********* applies each op to create a list of new states */

applyops(State,[Op|Ol],[Ns|T]) :- applyop(State,Op,Ns),
                                  applyops(State,Ol,T).
applyops(_,[],[]).

/*    MACROS ARE OUT FOR NOW
applyop(State,Op,Ns) :- frame(  name: Op,
                                type: operator,
                                macrop: [_|L],
                                _,_,_,_,_),
```

```
                    applyopseq1(L,State,SS),
                    last(SS,Ns).
*/

applyop(State,Op,Ns) :- frame(  name: Op,
                                type: operator,
                                filter: _,
                                check: _,
                                precon: _,
                                padd: Padd,
                                add: Add,
                                delete: Del),
                        del(Del,State,Ns1),
                        ad(Add,Padd,Tadd),
                        ad(Tadd,Ns1,Ns).

/* del(X,Y,Z) Z is Y-Z, where X and Y are SETS &ed */
/* seems to work ok for an instantiated S on backtracking */
/* last results show that later alternatives generated are rubbish */

del(X,Y,Z) :- del3(X,Y,Z1),remove(nil,Z1,Z).

del3(X&D,S,Ns) :- !,del2(X,S,Ns1),del3(D,Ns1,Ns).
del3(X,S,Ns) :- del2(X,S,Ns).

del2(X,S,Z) :- elem(X,S),remove(X,S,Z).
del2(X,S,S) :- not(elem(X,S)).

del_cut(A,B,C) :- del(A,B,C),!.

remove(nil,X&nil,X) :- !. /*copes with removing nil  */
remove(X,X&S,S) :- !.
remove(X,Y&S,Y&Z) :- !,remove(X,S,Z).
remove(X,X,nil) :- !.
remove(nil,X,X).     /* this is so rem. nil will always succeed */

remove_bt(X,X&S,S).
remove_bt(X,Y&S,Y&Z) :- remove_bt(X,S,Z).
remove_bt(_,_&_,_) :-!,fail.
remove_bt(X,X,nil) :- !.
/************/

ad(X&A,S,Ns) :- elem(X,S),ad(A,S,Ns).
ad(X&A,S,X&Ns) :- ad(A,S,Ns).
ad(X,S,S) :- elem(X,S).
ad(nil,S,S).
ad(S,nil,S).
ad(X,S,X&S).

adcut(Ps,Pe,PP) :-
        ad(Ps,Pe,PP),!.


last([E|[]],E) :- !.
```

```
last([_|T],E) :- last(T,E).

/* adds extra predicates to a state which are implied implicitly */

exp_state(I,O) :- frame(_,_,_,axioms: L),exps1(I,O,L).

exps1(O,O,[]) :- !.
exps1(I,O,[Pre,Post|R]) :-
                del(I,Pre,nil),
                ad(Post,I,O1),
                exps1(O1,O,R).
exps1(I,O,[_,_|R]) :-
                exps1(I,O,R).

/* adds extra preds to environment facts which are implied implicitly */

exp_env(Pe,Pe2) :-
        env_axioms(L),
        expe1(Pe,Pe1,L),
        adcut(Pe,Pe1,Pe2),
        !.

expe1(_,nil,[]) :- !.
expe1(Pe,B&Pe1,[[A,B]|L]) :-
        hold(A,Pe),
        expe1(Pe,Pe1,L).
expe1(Pe,Pe1,[_|L]) :-
        expe1(Pe,Pe1,L).


/* collects all the constants in a cjn of preds into a list */

get_consts(P&PS,CL) :- get_con(P,CL1),get_consts(PS,CL2),
                              append(CL1,CL2,CL),!.
get_consts(P,CL) :- get_con(P,CL),!.

get_con(P,CL) :- P =.. [_|T], get_con1(T,CL).
get_con1([],[]).
get_con1([C|T],[C|CL]) :- not(var(C)),get_con1(T,CL).
get_con1([_|T],CL) :- get_con1(T,CL).



/*-------------------------------------------*/
/* chunk utilities */

dchunks :- retract(ch(_,_,_,_,_)),fail.
dchunks.

lchunks(F) :- tell(F),lchs1([]),lrec,lexch,told.
lchs1(X) :- ch(_,A,_,_,_),not(member(A,X)),lchs(A),lchs1([A|X]).
lchs1(_).
lchs(A) :- ch(Z,A,B,C,D),numvars(ch(Z,A,B,C,D),1,_),write(Z),
                write(A),write(B),nl,write(C),nl,write(D),nl,nl,fail.
```

```
lchs(_).

lrec :- nl,nl,ch_record(A,B,C,D),writeL([A,' ',B,' ',C,' ',D]),nl,fail.
lrec.

lexch :- nl,nl,ch_ex(A,B),writeL([A,' has exception ',B]),nl,fail.
lexch.

/*save chs for use in other task*/
st_chunks(F) :- tell(F),schs,srec,sexch,nl,told,
                tell(init_adv),init_world(I),write(init_world(I)),
                write('. '),nl,told.
schs :- ch(Z,A,B,C,D),
        write(ch(Z,A,B,C,D)),
        write('. '),nl,fail.
schs.
srec :- ch_record(A,B,C,D),write(ch_record(A,B,C,D)),
        write('. '),nl,fail.
srec.
sexch :-ch_ex(A,B),write(ch_ex(A,B)),
        write('. '),nl,fail.
sexch.



/*progress of a task

progrss(File) :- processq(X), writeout(File,X).
writeout(File,[H|T]) :-

*/


/* status of a problem */

status(P) :-    frame(name: P,_,_,_,_,_,trace: T,solution: S),
                tell(fred),
                pw(T),
                var(S),
                addprocess(status(P),75),
                tell(user).
status(P) :-    frame(name: P,_,_,_,_,_,trace: _,solution: p(X,Y,Z)),
                tell(fred),
                nl,nl,write('solution**************'),nl,
                    write('   state:'),nl,write(X),nl,
                    write('   ops:'),nl,wlist(Y),nl,
                    write('   cost: '),write(Z),nl,nl,nl,
                tell(user).

pw([p(X,Y,Z)|T]) :- write('partial trace:-'),nl,
                    write('   state:'),nl,write(X),nl,
                    write('   ops:'),nl,wlist(Y),nl,
                    write('   cost: '),write(Z),nl,nl,nl,
                    pw(T).
```

```
pw([]).

wlist([X|Y]) :- write(X),nl,wlist(Y).
wlist([]).

/*-------------------------------------------*/

/* pretty list frames to file F  ONLY for succ/fail lists */

writeframes(F) :- tell(fred1),listing(frame),told,
                  see(fred1),tell(F),read(Fr),
                  wfs(Fr),told,seen.

wfs(end_of_file).
wfs(Frame) :- pp(Frame),read(Next),wfs(Next).

pp(F) :- F =.. [frame|T],write('frame('),nl,nl,
         pp1(T),write('   ).'),nl,nl.

pp1([N,type:operator,_,_,_,_,_,_,succ:S,fail:F]) :-
                write(N),nl,nl,
                write('succesful contexts:'),nl,nl,
                wlrs(S),
                write('failed contexts:'),nl,nl,
                wlrs(F).
pp1(T) :- wlist(T).

wlrs([]).
wlrs([r(G,S,O,P)|L]) :- write(P),tab(5),write(G),tab(5),
                        write(O),nl,nl,wlss(S),nl,nl,wlrs(L).
wlss(X&Y&Z) :- write(X),write('&'),write(Y),nl,wlss(Z).
wlss(X) :- write(X),nl,nl.

/* prints to File all frames of type T with arity A */

storefs(File,T,A) :-
                        tell(File),
                        functor(X,frame,A),
                        call(X),
                        X =.. [_,_,type:T|_],
                        X =.. Y,
                        wlist(Y),nl,fail.
storefs(File,_,_) :-
                        tell(File),told.


/*++++++++++++++++++++++++++++++++++++++++++++++ */


/* ************* fmacgr fmacgr fmacgr fmacgr fmacgr **************** */


/* This file contains a procedure to make solutions
```

```
   into MACRO OPERATORS      F M A C G R
   --- Uses fred4 as a scratch file
   --- This version uses 'goal regression'  */

mac(X) :- frame(name: X,
                type: problem,
                ancest:_,
                context: C,
                init_world: I,
                goal: GG,
                _,
                solution: S),
                mac1(S,C,I,GG,X).

macros(R,C,I,G,X) :-
        producemacros(on),
        macros1(R,C,I,G,X).
macros(_,_,_,_,_).


macros1([p(S,O,_)|R],C,I,G,X) :-
        del(S,G,G),writeL(['*mac*-no goals solved by op ',O,nl,S,nl,G]),
        macros1(R,C,I,G,X).
macros1([p(_,O,_)|R],C,I,G,X) :-
        length(O,1),
        macros1(R,C,I,G,X).
macros1([p(S,O,_)|R],C,I,G,X) :-
        intersection(S,G,GG),
        gensym(X,XX),
        mac1(p(S,O,_),C,I,GG,XX),
        macros1(R,C,I,G,X).
macros1(_,_,_,_,_).

mac1(p(S,OLm,_),C,I,GG,X) :-

        producemacros(on),
        nonvar(S),
        length(OLm,Len),
        replacemacs(OLm,OL,C),    /* primitives->macs,update mac'score*/
        !,Len > 1,      /* original opseq must be > 1        */
        del(I,GG,G),
        applyopseq1(OL,I,SS1),         /* no precon check */
        removelast(SS1,SS),

        reverse([I|SS],SSR),
        reverse(OL,OLR),
        /*       v  v   v  v    ^    ^    ^             Ch = WPe1       */
        getprec(G,OLR,SSR,C, WPs1,Ch,ConstLD), /* old method below */
        list_to_set(ConstLD,ConstL),

        /*getprecon(OL,Pre,Ch,[I|SS],C),Pre= O1.pre+(Oi.pre-O(i-1).add) */

        applyopseq1(OL,WPs1,FL),      /*Ch are acculmutated perman.cons.*/
        last(FL,Finalstate),
```

```
        del(WPs1,Finalstate,Add),      /* Add = Finalstate - Pre */
        del(G,Add,Sadd),

        generalise(WPs1,ConstL,P1),
        generalise(Ch,ConstL,Ch11),
        generalise(G,ConstL,A2),
        generalise(Sadd,ConstL,A1),    /* consts which are not in ConstL */
                                       /* are turned into variables  */
        generaliseL(ConstL,OL,OL1,V4),  /* generalize op seq. instances
                                            & collect vars in V4*/
        list_to_set(V4,V5),     /* assume NO CONSTS IN UNINST. OPS NAMES */
        V =.. [X|V5],

        andtolist(Ch,ChL),            /* this part ensures minimal gen'n */
        generaliseL(ConstL,ChL,_,Cv),/* by adding not equal literals to */
        list_to_set(Cv,Cv1),          /* the 'check' literals           */
        add_ne(Cv1,Ne1),
        slim(Ne1,Ne2),
        remove(nil,Ne2,Ne),
        ad(Ch11,Ne,Ch1),
        possible_macro(X,V,OL1,Ch1,P1,A2,A1).

possible_macro(_,_,OL1,_,_,_,_) :-
        frame(name: _,
                        type: operator,
                        macrop:[_|OL1],
                        check: _,
                        precon: _,
                        padd: _,
                        add: _,
                        delete: _
                        ).

possible_macro(X,V,OL1,Ch1,P1,A2,A1) :-
        clock(Time),             /* attach a value to a macro */
        Ti is Time + 2,          /* currently 2 more than task no. */
        tell(X),
        write(  frame(name: V,
                        type: operator,
                        macrop:[Ti|OL1],
                        check: Ch1,
                        precon: P1,
                        padd: A2,
                        add: A1,
                        delete: nil)),
                write('.'),nl,told,   /* turns upper case into vars!!! */
        see(X),read(TERM),seen,assert(TERM),
        tell(X),     /* ******* report new macro ******* */
        write(V),nl,nl,
        write('check: '),write(Ch1),nl,
        write('precon: '),write(P1),nl,
        write('padd: '),write(A2),nl,
        write('add: '),write(A1),nl,
        write('macrop: '),write(OL1),nl,
```

```
        told.


add_ne([_],nil).
add_ne([X|Y],Z) :- add_ne1(X,Y,Z1),add_ne(Y,Z2),ad(Z1,Z2,Z).

add_ne1(_,[],nil).
add_ne1(X,[Y],ne(X,Y)).
add_ne1(X,[Z|Y],ne(X,Z)&Z1) :-
                add_ne1(X,Y,Z1).

slim(ne(U,V)&Y,ne(U,V)&Z) :-
                name(U,[U1|_]),name(V,[U1|_]),slim(Y,Z).
slim(ne(_,_)&Y,Z) :- slim(Y,Z).
slim(ne(U,V),ne(U,V)) :-
                name(U,[U1|_]),name(V,[U1|_]).
slim(_,nil).
                        /*   ^   ^   ^    */
getprec(Abs,[O1|T],[S|T1],C,Abs4,Ch,ConstL) :-

                functor(O1,Fun,NN),/* ops must be unique */
                functor(Ou,Fun,NN),/* must look at uninst'ed operator */
                frame(name: Ou,_,_,_:EV,precon:PV,_,_,_),
                get_consts(PV,CS1),get_consts(EV,CS2),
                append(CS1,CS2,CS),

                frame(name:O1,_,_,check:Ch1,precon:Pr,padd:PA,add:AD,_),
                hold(Pr,S),                  /*These 3 lines instantiate */
                frame(name:C,_,always:A,_), /*vars not already done by  */
                hold(Ch1,A),                 /*op parameter instants     */

                del(PA,Abs,Abs1),
                del(AD,Abs1,Abs2),
                ad(Pr,Abs2,Abs3),
                getprec(Abs3,T,T1,C,Abs4,Ch2,CSC),
                append(CSC,CS,ConstL),
                ad(Ch1,Ch2,Ch).

getprec(Abs,[],_,_,Abs,nil,[]) :- !.

/*              getprecon([O1|T],P,Ch,[S|T1],C) :-
                frame(name:O1,_,_,check:Ch1,precon:Pr,_,_,_),
                hold(Pr,S),
                frame(name:C,_,always:A,_),
                hold(Ch1,A),
                getpre([O1|T],P1,Ch2,T1,nil,C),
                ad(Ch1,Ch2,Ch),
                ad(Pr,P1,P).
                getpre([O1,O2],P1,Ch,[S],Astore,C) :-
                frame(name:O1,_,_,_,_,padd:PA,add:A,_),
                frame(name:O2,_,_,check:Ch,precon:Pr,_,_,_),
                hold(Pr,S),
                frame(name:C,_,always:Always,_),
                hold(Ch,Always),
```

```
                ad(PA,A,AA),
                ad(AA,Astore,A1),
                del(A1,Pr,P1).
                getpre([O1|[O2|T]],P2,Ch,[S|T1],Astore,C) :-
                frame(name:O1,_,_,_,_,padd:PA,add:A,_),
                frame(name:O2,_,_,check:Ch2,precon:Pr,_,_,_),
                hold(Pr,S),
                frame(name:C,_,always:Always,_),
                hold(Ch2,Always),
                ad(PA,A,AA),
                ad(AA,Astore,A1),
                del(A1,Pr,P1),
                getpre([O2|T],P,Ch3,T1,A1,C),
                ad(Ch2,Ch3,Ch),
                ad(P1,P,P2).                  */

/* This part changes constants to variables    NB: only 1-depth         */
/* by turning the 1st letter to a capital.                              */
/* -the first parameter must be a &-exp           constants considered */

generalise(nil,_,nil).
generalise(X&Y,ConstL,X1&Y1) :-
                X =.. [H|T],
                genlist(ConstL,T,T1,_),
                X1 =.. [H|T1],!,
                generalise(Y,ConstL,Y1).
generalise(X,ConstL,X1) :-
                X =.. [H|T],
                genlist(ConstL,T,T1,_),
                X1 =.. [H|T1].

generaliseL(ConstL,[X|Y],[X1|Y1],L2) :-
                X =.. [H|T],
                genlist(ConstL,T,T1,L1),
                X1 =.. [H|T1],!,
                generaliseL(ConstL,Y,Y1,L3),
                append(L1,L3,L2).
generaliseL(_,[],[],[]).

genlist(_,[],[],[]).
genlist(ConstL,[H|T],[H1|T1],[H1|T2]) :-
                not(var(H)),
                not(member(H,ConstL)), /*CHECK H IS NOT A SPECIAL CONST */
                name(H,[I|J]),
                I1 is I - 32,
                name(H1,[I1|J]),
                genlist(ConstL,T,T1,T2).
genlist(ConstL,[H|T],[H|T1],L) :-
                genlist(ConstL,T,T1,L).

/* take a list of instantiated ops and replace any macros with their
   primitives */

replacemacs([],[],_).
```

```
replacemacs([O|OL],T1,C) :-
                frame(name:O,_,macrop:[_|T],check:Ch,_,_,_,_),
                increment(O),
                frame(name:C,_,always:Always,_),  /* instantiate params*/
                hold(Ch,Always),
                replacemacs(OL,T2,C),
                append(T,T2,T1).
replacemacs([O|T],[O|T1],C) :-
                        replacemacs(T,T1,C).

increment(O) :-
                functor(O,Fun,NN),
                functor(Ou,Fun,NN),
                retract(frame(name: Ou,
                        type: operator,
                        macrop:[N|T],
                        check: Ch,
                        precon: P,
                        padd: PA,
                        add: A,
                        delete: D)),
                N1 is N+1,
                asserta( frame(name: Ou,
                        type: operator,
                        macrop:[N1|T],
                        check: Ch,
                        precon: P,
                        padd: PA,
                        add: A,
                        delete: D)).

/********** fexh fexh fexh  fexh     ****************/

/* ---breadth first forward search with a check for duplicate states
      (sift predicate) */
/* NOTE: THIS IS DIFFERENT FROM LP'S- THE SEMANTICS OF INVERSE OPERATORS
   HAVE BEEN PUT INTO ENVIRONMENT*/
/* definiton of fn to expand a partial soln exhaustively */

exhaustive_step(X) :-
                frame(  name: X,
                        type: problem,
                        ancest:Ancest,
                        context: C,
                        init_world: I,
                        goal: G,
                        trace: P0,
                        solution: S),
                ( (P0=[],P=[p(I,[],0)]) ; P = P0 ),

/* print no. of open nodes */
                length(P0,Le),write('#'),write(Le),write('#'),
                lowcost(P,Plc),
```

```
/* add one to expanded node count */
                retract(numberofnodes(Nod)),
                Nod1 is Nod +1,
                assert(numberofnodes(Nod1)),

                expand(Plc,Lnewps,G,C),
                removeL(Plc,P,P1),
                ( (Lnewps=[],P2=P1) ; append(Lnewps,P1,P2) ),
                ex_success(X,G,Lnewps,S),
                retract(frame(name:X,type:problem,_,_,_,_,_,_)),
                tell(fred3),wlist(P2),told,
                assertz(frame(name: X,
                        type: problem,
                        ancest:Ancest,
                        context: C,
                        init_world: I,
                        goal: G,
                        trace: P2,
                        solution: S)).


expand( p(Cstate,Y,N),Lnewps,G,C) :-
                        length(Y,Le),writeL(['OL',Le]),
                        findops(Cstate,Lops,C),
                        fastsift(Y,Lops,Lops1),
                        use_heuristics(Cstate,G,C,Lops1,Lops2,forward),
                        applyops(Cstate,Lops2,Lstates),!,
                        /*sift(Lops,Lstates,Lops1,Lstates1),*/
                        genps(Y,N,Lops2,Lstates,Lnewps,G).


genps(_,_,[],[],[],_).
genps(Y,N,[Op|Opt],[Sr|St],[p(Sr,Or,N1)|T],G) :-
                        append(Y,[Op],Or),
                        pathcost(Sr,Or,N,N1,G),
                        genps(Y,N,Opt,St,T,G).

/* take into account number of goals solved */
pathcost(S,Or,_,N1,G) :- length(Or,Len),del(S,G,Left),andtolist(Left,L),
                        length(L,NN),N1 is 2*NN +Len.



follows(G,[p(X,Y,N)|T],S) :-
                                (hold(G,X),
                                S =  p(X,Y,N)) ;
                                follows(G,T,S).
 /*        asslist(Lstates). -only needed if all states are being saved
asslist([]).
asslist([X|Y]) :- gensym(cs,CS),write(CS),assert(state(X)),asslist(Y).*/

ex_success(X,G,Lnewps,S) :-  follows(G,Lnewps,S),
                          wipestates,
                          addprocess(critic(X),800),
```

```
                              addprocess(chunk(X),800),
                              /*addprocess(status(X),900),*/
                              addprocess(mac(X),900).
wipestates :- retract(state(_)),fail.
wipestates.


ex_success(X,_,_,_) :-
                           addprocess(exhaustive_step(X),200).
                            /*  regenerates itself? */
fastsift([],X,X).
fastsift(_,[],[]).
fastsift([Y1|_],[O|OL],OL1) :-
                inverse(O,Y1),
                fastsift([Y1|_],OL,OL1).
fastsift([Y1|_],[O|OL],[O|OL1]) :-
                fastsift([Y1|_],OL,OL1).

/* gets rid of states that have been seen before -- */

sift([],[],[],[]).
sift([_|R1],[S|R2],OL,SL) :-
                        state(ST),
                        del(S,ST,nil),
                        sift(R1,R2,OL,SL).
sift([O|R1],[S|R2],[O|OL],[S|SL]) :-
                        sift(R1,R2,OL,SL).

/* gets rid of an operator that is just the inverse of the last one-*/

fastsift(_,[],[],[],[]).
fastsift(CS,[_|R1],[S|R2],OL,SL) :-
                        hold(S,CS),
                        fastsift(CS,R1,R2,OL,SL).
fastsift(CS,[O|R1],[S|R2],[O|OL],[S|SL]) :-
                        fastsift(CS,R1,R2,OL,SL).

/* find lowest cost of list of p(<states>,<ops>,<n>) */

lowcost([p(X,Y,N)],p(X,Y,N)) :- !.
lowcost([p(X,Y,N)|T],Z) :- lowcost(T,p(X1,Y1,N1)),
                          ( (N<N1 ,!, Z=p(X,Y,N)) ;
                            Z=p(X1,Y1,N1)        ).


/************* drive driverf driverf driverf driverf driver *********/

/* FORWARD SEARCH */


b :- environment(C),init_world(I),nl,
        write('My environment is called '),write(C),nl,
        write('My current world is '),nl,write(I),nl,
        task(C,I).
```

```
bfile :- environment(C),init_world(I),nl,
        write('My environment is called '),write(C),nl,
        write('My current world is '),nl,write(I),nl,
        see('list.tsk'),task(C,I),bfile.
bfile.

task(C,I) :-
                retract(numberofnodes(_)),
                assert(numberofnodes(0)),
                nl,write('Enter task or "h" for help>'),
                read(G),
                help(G,C,I). /* see driver_help */


task1(_,_,end_of_file) :-
                see('list.tsk'),seen,!,fail.
task1(C,W,G) :-
                gensym(task,T),
                assert(
                        frame(  name: T,
                                type: problem,
                                ancest: [],
                                context: C,
                                init_world: W,
                                goal: G,
                                trace:[],
                                solution: _)
                        ),
                retract(processq(_)),
                assert(processq([d(exhaustive_step(T),1000)])),
                !,go,
                numberofnodes(Nn),
                retract(activation(_)),
                assert(activation(0)),
                writeL([nl,'no. of expanded nodes: ',Nn,nl]),
                frame(  name: T,
                        type: problem,
                        ancest: _,
                        context: C,
                        init_world: _,
                        goal: G,
                        trace:_,
                        solution:p(S,Om,_)),
                not(var(S)),
                replacemacs(Om,O,C),
                write('By sequence of operators '),nl,nl,write(O),nl,nl,
                write('goal '),write(G),write('is satisfied, state is'),
                nl,nl,write(S),nl,
                retract(init_world(_)),
                assert(init_world(S)),
                (print_stats ; true ).  /* true for C-prolog */

print_stats :-
        statistics(runtime,[_,CP]), CPused is fix(CP/100),
```

```
        statistics(clause_store,[X1,X2]), C is fix(100*(X2/X1)),
        statistics(global_stack,[X3,X4]), G is fix(100*(X4/X3)),
        statistics(local_stack,[X5,X6]), L is fix(100*(X6/X5)),
        statistics(trail,[X7,X8]), T is fix(100*(X8/X7)),
        writeL(['CPUsed=',CPused,' secs, Space left: ']),
        writeL(['C=',C,'% G=',G,'% L=',L,'% T=',T,'%',nl]).


task1(_,_,_) :- nl,nl,write('**TASK FAILURE**'),nl,nl,
                storefs(fred4,problem,8),
                write('see fred4 for problem dump').



/******** fchunk fchunk fchunk fchunk fchunk fchunk ***********/
/* uses numerous routines from FMACGR */
/* This contains the problem solution chunker for
forward search methods- the chunk can then contain
the subset of the state (-in which it was applied
successfully) which was nec. for the success of the
rest of the op sequence.
*/

chunk(X) :-     chunking_is(off).

chunk(X) :-     chunking_is(on),
                frame(  name: X,X1,X2,context:Context,
                        init_world: I,
                        goal: G,
                        trace: T,
                        solution: p(S,OL,C) ),
                removelast(OL,OL1),
                applyopseq1(OL1,I,SL),
                rechunks(I,G,OL,[I|SL],Context).



/* records chunks */

rechunks(I,G,[O|T1],[S|T2],C) :-
        reverse([O|T1],OLR),
        reverse([S|T2],SLR),
        getprec(G,OLR,SLR,C,P,Check,ConstLD),
        list_to_set(ConstLD,ConstL),
        /* only for mea  --- intersection(PP,I,P),  */
        generalise(O,ConstL,Og), /* also note inters.(x,y,nil) fails!*/
        generalise(G,ConstL,Gg),
        generalise(P,ConstL,Pg),
        generalise(Check,ConstL,Checkg),

        andtolist(Check,ChL),   /* this part ensures minimal gen'n */
        generaliseL(ConstL,ChL,_,Cv),  /*by adding not eq.literals to */
        list_to_set(Cv,Cv1),    /* the 'check' literals            */
```

```
        add_ne(Cv1,Ne1),
        slim(Ne1,Ne2),  /* dirty way (by 1st letter) of del'ing some ne's */
        remove(nil,Ne2,Ne),
        ad(Checkg,Ne,Ch1),
        gensym(ch,Nm),
        tell(fred),writeL([ch(Nm,Og,Gg,Pg,Ch1),'. ']),told,
        see(fred),read(ch(Nm,O1,G1,P1,Check1)),seen,
        ( ch(_,O1,G1,P1,Check1) ; assert(ch(Nm,O1,G1,P1,Check1)) ),!,
        rechunks(I,G,T1,T2,C).
rechunks(I,_,[],[],C).

/****************************END*********************/
```

References

[Boswell 86]
Boswell, R., "Analytic goal regression: Problems, solutions and enhancements" Proceedings of the Seventh European Conference on Artificial Intelligence, Brighton, 1986.

[Carbonell 83]
Carbonell, J.G., "Learning by analogy", in Michalski, R.S., Carbonell, J.G., Mitchell T.M., "Machine learning, an artificial intelligence approach" Morgan Kaufmann, 1983.

[Carbonell and Gil 87]
Carbonell, J.G., Gil, Y., "Learning by experimentation", Proceedings of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[Carbonell 88]
Carbonell, J.G., "Prodigy: An integrated architecture for planning and learning" Invited talk at the Third Symposium on Methodologies for Intelligent Systems, Turin, Italy, October 1988.

[Chapman 87]
Chapman, D., "Planning for conjunctive goals", Artificial Intelligence 29, July 1987.

[Dawson and Siklossy 77]
Dawson C., and Siklossy, L., "The role of preprocessing in problem solving systems" Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 465-471, 1977.

[Dejong and Mooney 86]
Dejong, G., and Mooney, R., "Explanation-based learning: An alternative view", Machine Learning Vol. 1, No. 2, 1986.

[Fikes et al 72]
Fikes, R., Hart, P. and Nilsson, N. "Learning and executing generalised robot plans", Artificial Intelligence 3, 251-288, 1972.

[Fox 88]
Fox, M., "Commonsense reasoning in a hierarchical non-linear planner", Division of Computer Science, The Hatfield Polytechnic, 1988.

[Gries 83]
Gries, G., "The Science of Programming", Springer-Verlag, 1983.

[Hirsh 87]

Hirsh, H., "Explanation-based generalisation in a logic programming environment" Proceedings of the 10th International Joint Conference Artificial Intelligence, Morgan Kaufman, August 1987.

[Iba 85]
Iba, G., "Learning by discovering macros in puzzle solving" Proceedings of the Nineth International Joint Conference on Artificial Intelligence, Los Angeles, 1985.

[Kedar-Cabelli and McCarty 87]
Kedar-Cabelli, S.T. and McCarty, L.T., "Explanation-based generalisation as resolution theorem proving", Proceedings of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[Kodratoff 84]
Kodratoff, I., "Careful generalisation for concept learning", Proceedings of the Sixth European Conference on Artificial Intelligence, 229-238, 1984.

[Korf 85]
Korf, R. E., "Macro operators: A weak method for learning", Artificial Intelligence 26, 35-77, 1985.

[Krawchuk and Witten 88]
Krawchuk, B.J. and Witten I.H., "Problem solvers that learn", Proceedings of the Third European Working Session on Learning, Pitman, October 1988.

[Lenat and Brown 83]
Lenat, D.B. and Brown, J.S., "Why AM and Eurisko appear to work" Proceedings of A.A.A.I. 83, 236-240, 1983.

[Laird et al 84]
Laird, J., Rosenbloom, P. and Newell, A., "Toward chunking as a general learning mechanism", Proceedings of A.A.A.I. 84, 188-192, 1984.

[Laird et al 86]
Laird J.E., Rosenbloom P.S., Newell A., "Chunking in Soar: the anatomy of a general learning mechanism" Machine Learning 1, Kluwer Academic, 1986.

[McCluskey 86]
McCluskey, T.L., "Revised Research Plan", unpublished, revised Ph.D. proposal, March 1986.

[McCluskey 87a]
McCluskey, T.L., "Machine learning of heuristics in general problem solvers" Invited talk to the 10th Visegrad Winter School on Intelligent Systems, Veszprem, Hungary, January 1987.

[McCluskey 87b]

McCluskey, T.L., "The anatomy of a weak learning method for use in goal directed search", Proceedings of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[McCluskey 87c]
McCluskey, T.L., "Combining weak learning heuristics in general problem solvers", Proceedings of the 10th International Joint Conference on Artificial Intelligence, Morgan Kaufman, August 1987.

[McCluskey 87d]
McCluskey, T.L., "A weak learning method for use in goal directed search", poster given to the Workshop on Machine Learning, University of California, June 1987, and report TCU/CS/87/15.

[McCluskey 88a]
McCluskey, T.L., "Deriving a correct logic program from the formal specification of a non-linear planner" Proceedings of the the Third Symposium on Methodologies for Intelligent Systems, North Holland, October 1988.

[McCluskey 88b]
McCluskey, T.L., "The FM user guide" Report TCU/CS/88/30, October 1988.

[Michalski and Chilausky 80]
Michalski, R.S., Chilausky, R.L., "Knowledge acquisition by encoding expert rules versus computer induction from examples: a case study involving soybean pathology", International Journal for Man-Machine Studies, 1980.

[Michalski et al 83]
Michalski, R.S., Carbonell J.G., Mitchell T.M., (eds) "Machine learning, an artificial intelligence approach" Morgan Kaufmann, 1983.

[Michalski 85]
Michalski, R.S., "Automated Knowledge Refinement", Artificial Intelligence Laboratory, University of Illinois at Champaign-Urbana, 1985.

[Minton 85]
Minton, S., "Selectively generalising plans for problem solving", Proceedings of the Nineth International Joint Conference on Artificial Intelligence, Los Angeles, 1985.

[Minton and Carbonell 87]
Minton, S., Carbonell, J., "Strategies for learning search control rules: an explanation-based approach", Proceedings of the 10th International Joint Conference on Artificial Intelligence, Morgan Kaufman, August 1987.

[Minton et al 87]
Minton, S., Carbonell, J.G., Etzioni, O., Knoblock, C.A., Kuokka,

D.R., "Acquiring effective search control rules: Explanation-based learning in the PRODIGY system", Proceedings of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[Mitchell 83]
Mitchell, T.M., "Learning and Problem Solving", Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1983.

[Mitchell et al 83]
Mitchell, T.M., Utgoff, P., Banerji, R.B., "Learning by experimentation: Acquiring and refining problem solving heuristics" in Michalski R.S., Carbonell J.G., Mitchell T.M., "Machine learning, an artificial intelligence approach" Morgan Kaufmann, 1983.

[Mitchell et al 86]
Mitchell, T.M., Keller, R.M., and Kedar-Cabelli S.T., "Explanation-based generalisation: a unifying view", Machine Learning 1, Kluwer Academic, 1986.

[Porteous 87]
Porteous, J., "A graphical interface to a general problem solver", M.Sc. dissertation, Computer Science Department, University College, London, September 1987.

[Porter and Kibler 84]
Porter, B. and Kibler, D., "Learning Operator Transformations", Proceedings of A.A.A.I. 84, 278-282, 1984.

[Porter and Kibler 86]
Porter, B. and Kibler, D., "A Comparison of Analytic and Experimental Goal Regression for Machine Learning", Proceedings of the Nineth International Joint Conference on Artificial Intelligence, 555-559, 1985.

[Sacerdoti 74]
Sacerdoti, E.D., "Planning in a hierarchy of abstraction spaces", Artificial Intelligence 5, 115-135, 1974.

[Sacerdoti 75]
Sacerdoti, E.D., "The non-linear nature of plans" , Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 1975.

[Steier et al 87]
Steier, D.M., Laird, J.E., Newell, A., Rosenbloom, P.S., Flynn, R.A., Golding, A., Polk, T.A., Shivers, O.G., Unruh, A., Yost G.R., "Varieties of learning in Soar: 1987", Proceedings of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[Van der Velde 86]

Van der Velde, W., "Explainable knowledge production", Proceedings of the Seventh European Conference on Artificial Intelligence, Brighton, 1986.

[Van der Velde 88]
Van der Velde, W., "Learning through progressive refinement", Proceedings of the Third European Working Session on Learning, Pitman, October 1988.

[Vere 77]
Vere, S.A., "Induction of Relational Productions in the presence of background information", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 349-355, 1977.

[Wilkins 84]
Wilkins, D.E., "Domain independent planning: Representation and plan generation", Artificial Intelligence 22, 269-301, 1984

[Winston 75]
Winston, P.H., "Learning structural descriptions from examples", in "The psychology of computer vision", P.H. Winston (Ed.), McGraw-Hill, 1975.

APPENDIX D.1

[McCluskey 87a]

"Machine Learning of Heuristics in General Problem Solvers"
Invited paper for the 11th Visegrad Winter School on Foundations of Intelligent Systems, Veszprem, Hungary, January 1987.

"MACHINE LEARNING OF HEURISTICS IN GENERAL PROBLEM SOLVERS"

by T.L.McCLUSKEY of the
DEPARTMENT OF COMPUTER SCIENCE,
CITY UNIVERSITY, LONDON E.C.1.;ENGLAND

ABSTRACT

This paper explores methods of heuristic learning within general problem solvers with the following ('STRIPS-like') constraints: operators are represented by structured precondition, delete and add components; states and environments are represented in first order logic; goals and initial states may vary from problem to problem.

In contrast research has focussed on heuristic learning in domain specific problem solvers which typically carry out a forward state space search towards a fixed goal. Their application domains have generally been puzzles or mathematical rewrite problems. This work has shown techniques arising from concept generalisation and goal regression to be useful in the creation of heuristics.

General problem solvers are notoriously slow and prone to combinatorial explosion since to maintain their generallity they are supplied only with weak (syntactic) heuristics. In an attempt to overcome this efficiency vs. generallity trade-off, some researchers have proposed that a general problem solver could be equipped with 'weak' learning components whose task it is to create strong domain-specific heuristics either by experience or discovery.

We have explored this thesis using a problem solver shell called 'FM' which has been supplied with various learning components. It has been applied to several domains, creating heuristics by experience and subsequently improving its efficiency. This paper will outline the system's architecture and show examples of the heuristics learnt.

Details of two important and complementary learning heuristics which are used to improve the performance of FM's goal reduction search strategy will be discussed: (a) the creation of heuristics to prune operator choices; and (b) the compilation of operator sequences into macros and their selective use.

1. INTRODUCTION

The research of which this paper forms a part seeks to discover methods of learning that can be integrated into state space general problem solvers. Specifically we are interested in weak learning heuristics (discussed in [Korf 85] and [Langley 85]) which can form part of a problem solver shell and subsequently help it acquire stronger problem solving heuristics when applied to any specific domain. We wish to concentrate on heuristic learning that is not restricted to one search strategy, that works in domains that demand varied goals and can transfer learning to new domains.

This constitutes a different approach to much recent work on heuristic learning in problem solvers (e.g. [Mitchell et al 83], [Korf 85], [Langley 85]). Their systems typically improve in domains with a fixed goal, employ a more specialised representation scheme, and a forward search strategy.

A successful weak learning heuristic incorporated into a general problem solver of this kind is the creation of macro-operators (e.g. the Macrop of [Fikes et al 72] or the Bigop of [Dawson & Siklossy 77]). Recently, research interest in this old idea of abstracting, generalising and storing past successful operator sequences has been revived (e.g. [Minton 85] or in a 'dispersed' form [Laird et al 86]). In this way macro-operators are formed from experience, although other approaches involving puzzle problems use discovery methods ([Iba 85],[Korf 85]), or preprocessing ([Dawson & Siklossy 77]) .

We maintain in accordance with [Korf 85] that acquiring useful macro-operators is but one of several weak learning heuristics which must be available to what we term a 'heuristic-learning problem solver shell' for it to improve its performance within a particular domain: it must build up a large well organised, operational body of knowledge to gain problem solving efficiency.

This paper proposes a new weak learning heuristic for goal reduction state space problem solvers which creates what we call 'b-chunks'. These are dispersed search control heuristics gained through experience, bearing a similarity to the chunks of [Laird et al 86] but appearing within the problem solver shell as distinct and complementary components to macros.

We have built a modular problem solver shell called FM which allows states, goals and environments to be expressed as conjunctive expressions and production rules, and operators expressed in terms of structured add, delete and precondition predicates. Domains are of course interchangeable as are control stategies (state space or goal reduction, along with their associated weak problem solving heuristics) and weak learning heuristics such as macros and chunks.

2. RELATED WORK

2.1 Macrops

Evidently Samuel's famous draughts program [Samuel 59] first implemented the general technique of learning by re-using previously stored solutions, but two important sophistications over simply storing a problem-solution pair were introduced by the method of macro operator or 'Macrop' creation in [Fikes et al 72]:

-abstracting out any details in the initial problem state that do not contribute to the finding of a solution,

-the selective generalisation of certain constants occurring in the problem's solution.

Both these techniques will increase the generality of the solution procedure, but the latter is not straightforward. It depends on the constant appearing as an arbitrary member of some particular type in the solution's operator sequence and so can be generalised to a variable with that type restriction. Generalisation is justified since no operator in the solution sequence referred to the constant specifically but only to its type.

The situation where distinct constants of the same type are over-generalised to independant variables crops up in [Fikes et al 72] where two ad hoc techniques for avoiding inefficiency and inconsistency caused by this are suggested. [Laird et al 86] and [Kodratoff 85] for example, argue for this distinction to be kept, adding a binding restriction that prohibits variable instantiations to be equal.

Another feature of the Macrops was their open representation within STRIPS as 'triangle tables' which allowed subsequences of operators to be later re-used. This method of storing 'open macros' for the purpose of re-using arbitrary subsequences has been criticised in [Minton 85] and [Carbonell 83] on the grounds of inefficiency, although in the original paper the representation was also used for its benefits in monitoring plan execution.

2.2 Closed Macros

Recently research into the formation of macro operators in state space problem solving has been re-vitalised. In the Morris program of [Minton 85] useful parts of a solution sequence are compiled and generalised into a macro which has the same structure as a primitive operator and can only be used as one. We call this a 'closed macro' in contrast with STRIPS's Macrops. Morris defines a useful operator in the following way: it keeps a record of past solutions and makes macros out of any two operator subsequences that unify after generalisation. The macro set is pruned by deleting the least successfully used macros.

Significant improvement in problem solving within simulated robot worlds is achieved but there are drawbacks in using closed macros as the sole learning component:

-search trees shorten but grow bushy since distinct instantiations of goal achieving macros proliferate. A shift in problem representation is effectively made but within which a weak heuristic evaluation function is generally inadequate.

-solutions which comprise of closed macros are prone to produce non optimal paths (even after, checks for redundant primitive operator sequences have been made).

2.3 Chunks

The chunks created by Soar [Laird et al 86] improve the system's subsequent problem solving behaviour by providing search control knowledge. They are formed using similar ideas of abstraction and generalisation involved in macro creation, but the learnt components are stored in a dispersed manner. We outline part of their technique below.

During problem solving, when a non-trivial goal is solved, a production rule (chunk) is created for each correct operator choice in the problem solving trace (i.e. along the solution path). Its condition part is built from 'aspects of the situation that existed prior to a goal and which were examined during the processing of a goal' [ibid p.23], and the action is to recommend the application of the operator. Constants occurring in the production are then carefully generalised. When in subsequent problem solving a choice between operator applications has to be made, a chunk can recommend its associated operator if the chunk's precondition matches the current situation.

This form of dispersed, compiled knowledge has certain advantages over the macro because it helps search with existing operators rather than (in the extreme case) thickening search with extra operators.

2.4 More specialised work on strategy acquisition

Many researchers have concentrated on problem solvers that attempt to solve fixed goals, employ a forward directed search strategy and a specific representation scheme. (see [Langley 85] for recent advances). Applications are typified by the symbolic integration system LEX which modifies its search heuristics by analysing the trace of a successful solution. It may generalise the concept of a useful heuristic for an operator's application if that operator appeared in the solution sequence; or specialise the concept so as not to admit the re-use (within a similar context) of an operator which was found to lead away from the solution sequence. (see [Mitchell 83] for details]).

The success of systems such as these relies partly on a special concept description language (c.d.l) being supplied within which types of operator bindings are partially ordered. They improve by experience, narrowing the interval in this ordering (for each operator) and therefore becoming more selective about the situations when operators are applied (see [Bundy 82] for a comparison of early work).

Our experience has shown that the transfer of techniques developed in the work referred to above, to our 'STRIPS' like formulation, is a difficult task for the following reasons:

a. Assume that a goal has been achieved by the application of a sequence of operators. The occurrence of an operator in this solution sequence, in general, depends on the goal condition, the operator binding and the state description in which it was applied. A c.d.l. in which useful generalisations can take place would therefore need to involve all these components.

b. In a backward search space starting from a goal condition the nodes are not complete state descriptions but 'goal kernels' (as explained in [Dawson & Siklossy 77]). The heuristics learnt, therefore, must be tailored to act in the latter problem space.

More recently, some systems of this type (see [Porter and Kibler 84], [Porter and Kibler 85] or [Mitchell 83]) have resorted to goal regression techniques to create chunks of knowledge encapsulating operator application heuristics, through the generalisation of one example solution. The system's set of operators must be in (or be transformed into) a form where their semantics are declaritively available, so that it can reason backwards from the goal, and hence build an 'justification' for the generalisation (an instance of 'Explanation Based Learning' described in [Mitchell et al 86]). This process is similar to the build-up of a macro precondition explained below in section 4.1.

3. THE ARCHITECTURE OF FM

The design decisions reflected in FM's implementation were made to allow the addition of various learning components as well as providing an effective problem solver shell. At present the control can perform a state space or goal reduction search, the latter explained below.

3.1 GOAL NODE SEARCH IN FM

The backward search of FM proceeds in a goal reduction manner, starting with the initial goal, through a space of goal nodes (similar to the one in [Dawson & Siklossy 77]). Each goal node can be modelled as a 6-tuple:

(Identifier, Goal, Initial State, Ancestors, Purpose, Trace),

The Trace records attempts to solve the Goal, whereas the Purpose records why the goal node was created (typically to solve the unsatisfied preconditions of an operator). Goals, expressed as conjunctions of predicates, are initially assumed to be decomposable: when a goal node is activated, operator instantiations which add goal predicates have their unsatisfied preconditions form another goal node, unless they are already satisfied in which case those operators are applied to the initial state and the result recorded in the trace.

When the trace of a goal node eventually contains a state satisfying its goal (via an operator sequence Os), we say that the goal node is solved, and all nodes which are ancestors of it are removed from the search. If it was activated to solve an operator O's preconditions, then the sequence Os + O is applied to the goal node's parent's initial state and the result recorded in the parent's trace.

A goal node's initial state may be the state inherited from a parent node, or may be an advanced state partially satisfying the parent's goal. The latter is the case when goals cannot be solved by simple decompositon: FM examines the trace and forms new goal nodes whose goal predicates are inherited but whose initial states are selected from intermediate states taken from the parent's trace.

The kind of representation of goal nodes described above aids both the formation and use of strong heuristics. The trace is available for analysis and criticism after the solution of each goal node, allowing 'within-trial transfer of learning' to take place (as in [Laird et al 84]). In our implementation of FM we have experimented with the formation of closed macros, b-chunks and also subgoal ordering heuristics at this stage, but we shall limit our discussion to the first two.

4. MACROS IN FM

4.1 Construction

Macros are created and stored when goal nodes are solved, and then they are immediately available for problem solving. They are of the closed variety, compiled from the successful operator sequence into a primitive operator format. The major part of this compilation process is in building up a precondition M.p (a set of predicates) of a macro M. This can be accomplished in a forward fashion by collecting up those preconditions of primitive operators not supplied by an earlier operator's add-list. Consider a sequence of n fully instantiated operators between n+1 states, where 'U' and '-' mean set union and subtraction, and O[i].p, O[i].a stand for the precondition predicates and add predicates of operator i respectively. Then:

$$M.p = O[1].p \cup (O[2].p - O[1].a) \cup (O[3].p - (O[1].a \cup O[2].a)) \ldots \cup (O[n].p - (O[1].a \cup \ldots \cup O[n-1].a))$$

Another, more efficient method for forming M.p (implemented in FM) is to use a goal regression procedure. If G is the set of goal predicates for the solution sequence then:

$$M.p = Pn \text{ where } P0 = G$$
$$Pi = (Pi-1 - O[n+1-i].a) \cup O[n+1-i].p, \quad i = 1 \text{ to } n$$

In this context, the methods give equivalent results.

A compiled macro is then carefully generalised as outlined in 2.1. Identical constants are generalised to the same variable throughout the macro, but equality binding restrictions are added where variables of the same type are generalised from distinct constants.

4.2 Use

A backward directed search works well if, for each goal predicate G, the number of operator instantiations that achieve G is low (or can be kept low by heuristic pruning). Macros which break this rule are therefore exluded from the search ([Minton 85] p.598 complains of the same problem) although they may, of course, be used at any goal node whose initial state satisfies their preconditions.

A record of the frequency of use is kept for each macro and the least used are deleted periodically. This method is similar to that of Minton's for 'S-Macros', but while still keeping the most useful macros, it does not involve storing every problem's solution.

5. THE CHUNKS OF FM

B-Chunks are learnt in FM to provide heuristics for the search strategy outlined in section 3. They differ from the chunks of SOAR described in [Laird et al 86] in both construction and use: in FM's goal directed search they are created to be distinct and complementary heuristics to that of macros. The absence of such a learning component in STRIPS with Macrops is criticised in [Porter and Kibler 84]. Minton's Morris system [Minton 85] apparently uses a 'built-in' heuristic evaluation function.

Chunks can be also created when FM executes a state space search. In this case, although it is difficult to compare their structure with SOAR's since this would require a precise definition of the latter, their use and performance seem similar.

5.1 Construction

Consider O[i] (1≤i≤n) taken from an operator sequence O[1],O[2], ...,O[n] which achieves a goal node (with goal predicate(s) G) from a initial state I. We build a b-chunk (O[i]',G',P') for each O[i] using similar methods to those outlined in 4.1:

P = I intersected with the macro precondition of the sequence O[i],O[i+1], ...,O[n].

(O[i]',G',P') = the generalisation of (O[i],G,P) where identical constants are changed to the same variable names etc.

A chunk to guide state space search is constructed in the same manner except that P is simply the operator sequence's macro precondition.

O[i]'s b-chunk's third component may be roughly described as those predicates which were present in the goal node's initial state and that were also involved in the achievement of G after O[i-1]. This includes environment information (which is assumed to be a part of every state) that has been used in the satisfaction of the operators's preconditions.

5.2 Augmented B-chunks

In some situations the b-chunk, as presented in 5.1, may not be discriminatory if used on an identical problem to which it was formed. This tends to happen toward the end of an operator sequence when important similarities with the initial state are lost. We are at present experimenting with a procedure that augments the chunks by adding to P extra initial state predicates and 'association chains' from the environment (see [Vere 77]) that link them to predicates of the macro precondition.

5.3 Use

B-chunks are consulted when the problem solver finds multiple instantiations of the same operator to achieve a goal predicate Gp, but none of the preconditions are completely satisfied (a syntactic weak heuristic, such as favouring the instantiations with the lowest number of preconditions, is generally inadequate). A b-chunk (O1,G1,P) favours an operator instantiation O applied to a goal node if P logically follows from I (the initial state) under the variable bindings obtained by the successful matching of O1 to O and G1 to Gp or one of Gp's ancestors. The instantiation(s) favoured by the most chunks is then chosen to form a new goal node.

In conclusion, b-chunks can record the important similarities among the environment, initial state and goal obtained from successful problem solving, in a form usable for future goal directed search.

6. EXAMPLES

We present two examples from different domains. Note capital letters stand for variables in the chunks and macros shown.

6.1 The 8-puzzle

This example shows the creation and use of chunks in state space search and the similarity to Soar's chunks for this type of stategy. (see [Laird 86] which also gives details of the 8-puzzle problem). Basically the board has 9 numbered positions (p1,p2,...,p9) on which there are 8 numbered tiles (t1,t2,...,t8) and a 'blank'. The idea is to find a sequence of moves (i.e. swapping a tile with the blank horizontally or vertically) linking a pair of states, (S,G).

```
----------------
| p1 | p2 | p3 |
----------------
| p4 | p5 | p6 |
----------------
| p7 | p8 | p9 |
----------------
```

figure 1. The 8-puzzle board.

A state can be specified by the formula: at(tx,p1)&at(ty,p2)&...., the topology by next(p1,p2)&next(p1,p4)&.. and an operator move(tilex,py,pz) means swap tilex (which is on py) with the blank which must be on an adjacent postion pz.

The goal is 'at(t1,p1)&at(t2,p2)&..&at(blank,p5)&..' and to solve this the system must solve the decomposable goals

```
G1=     at(blank,p5),
G2=     at(blank,p5)&at(t1,p1),
G3=     at(blank,p5)&at(t1,p1)&at(t2,p2),
G4=    ... etc (see [Korf 85] for details)
```

To solve these goals from arbitary positions FM can create a series of chunks through its search experience. Consider the problem (I,G2) where I = at(blank,p5)&at(t1,p2)&..(anything). Using only weak heuristics FM finds the correct operator sequence, and creates chunks such as (ignoring some details):-

```
 ( move(T1,X1,X2):
     at(T4,X4):
     at(T1,X1)&at(blank,X2)&at(T2,X4)&at(T4,X3)&
     next(X2,X3)&next(X3,X4)&next(X4,X1)&next(X1,X2).).
```

If the system is subsequently given the problem (I',G2) where I = at(blank,p5)&at(t1,p4)& ...(anything) then no choices in the search will appear since chunks including the one above advise the correct operator bindings through each step in the state space search. This sort of improvement is termed 'symmetrical transfer' in [Laird 86].

6.2 Robot World

This example gives an idea of how domain specific heuristics can be built up in a simulated robot world similar to that found in [Fikes et al 72]. The system is asked to simulate a robot performing various box-moving tasks in the world described in figure 2.

```
---------------------------------------
|room1|room2              |room3      |
|     |                   \           |
---\ --                               |   figure 2:a robot world
|                         |           |
---\ ---------\ ------------\ -------
|room4    |room5                |room6
|         \                     \     |
|                                     |
|         |                     |     |
---\ ---------------\ ----------\ --
|room7                                |
|                                     |
---------------------------------------
```

FM learns various useful macros by experience such as (ignoring some details):-

```
mac-1(Roomx,Box,Door,Roomy) is
     preconditions: Roomx connects Roomy by Door &
                    Box is in Roomx on the floor & the Door is
                    open & the robot is in Roomy;
     action:        Puts Box in Roomy next to the Robot
     primitives: gotodoor(Door,Roomy),
                 gothrudoor(Door,Roomx),
                 goto(Box),
                 pushtodoor(Box,Door,Roomx),
                 pushthrudoor(Box,Door,Roomy).
```

B-chunks are learnt by experience as described above. They advise on the best instantiations of these macros during search e.g this chunk reates to the macro above (again some details are left out):

```
    ( mac-1(Room1,Box1,Door1,Room2);
        Box is in Room3;
        Box1 is in Room1 on the foor & Door1 is open & Room1
        connects Room2 by Door1 & Room2 connects Room3 ).
```

This advises instantiations for the macro that would move the box to to a room next to the one of its goal position, and advises against any instantiations that do not conform to this.

6. FUTURE WORK

Experiments have shown that impressive results are achieved when chunks are created for macros, because the number of possible instantiations of macros in the backward search tends to be much higher than primitives, and so the need for heuristic pruning is greater. To assess the full power of the b-chunk we must investigate:

-methods of chunk creation for macros that avoids any combinatorial explosion in the learning phase.

-concept learning to generalise existing chunks where appropriate (e.g. the example chunk is too specific in that it insists on open doors).

7. CONCLUSION

We have presented a new weak learning heuristic for state space, general problem solvers for use in goal directed reasoning. Given a particular domain, this weak method can create strong heuristics, in the form of b-chunks, through the experience of successful problem solving. These chunks record for each operator and generalised goal pair, the adviseable instantiations for operator variables, by storing important environment and initial state information.

8. REFERENCES

1. Bundy, A. and Silver, B., "A Critical Survey of Rule Learning Programs", Proceedings of the European Conference on Artificial Intelligence, 1982.

2. Carbonell, J.G. "Learning by Analogy", in "Machine Learning", Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

3. Dawson, C. and Siklossy, L. "The Role of Preprocessing in Problem Solving Systems" Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 465-471, 1977.

4. Fikes, R., Hart, P. and Nilsson, N. "Learning and Executing Generalised Robot Plans". Artificial Intelligence 3, 251-288, 1972.

5. Iba, G. "Learning by Discovering Macros in Puzzle Solving" Proceedings of the Nineth International Joint Conference on Artificial Intelligence, 640-642, 1985.

6. Kodratoff, I., "Careful Generalisation for Concept Learning", Proceedings of the European Conference on Artificial Intelligence, 229-238, 1984.

7. Korf, R. E., "Macro Operators: A Weak Method for Learning", Artificial Intelligence 26, 35-77, 1985.

8. Laird, J., Rosenbloom, P. and Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism" Machine Learning 1, 11-46, 1986.

9. Langley, P. "Learning to Search: From Weak Methods to Domain-Specific Heuristics", Cognitive Science 9, 217-260, 1985.

10. Minton, S. "Selectively Generalising Plans for Problem Solving", Proceedings of the Nineth International Joint Conference on Artificial Intelligence, 596-599, 1985.

11. Mitchell, T.M., Utgoff, P.E. and Banerji, R.B. "Learning by Experimentation: Acquiring and Refining Problem Solving Heuristics", in "Machine Learning", Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. (eds.), Tioga Publishing, 1983.

12. Porter, B. and Kibler, D. "Learning Operator Transformations", A.A.A.I. Proceedings, 278-282, 1984.

13.Porter, B. and Kibler, D. "A Comparison of Analytic and Experimental Goal Regression for Machine Learning", Proceedings of the Nineth International Joint Conference on Artificial Intelligence, 555-559, 1985.

14. Samuel, A.L., "Some Studies in Machine Learning Using the Game of Checkers", IBM Journal of Research and Development, 211-229, 1959.

APPENDIX D.2

[McCluskey 87b]

"The Anatomy of a Weak Learning Method for use in Goal Directed Search",
in Proceedings of the Fourth International Workshop on Machine Learning, Morgan Kaufman, June 1987.

The Anatomy of a Weak Learning Method for use in Goal Directed Search

T.L.McCLUSKEY
Department of Computer Science, The City University, London, England, EC1 OHB.

Abstract

We present an overview of a problem solving system that has been built to explore the potential of experience-based, weak methods for learning. One such method, that of creating heuristics from experience, is described in depth with the help of examples. This method creates heuristics to improve goal directed search by analysing why an operator appeared in a successful sequence. Heuristics are formed by the model-based generalisation of similarities between the operator's weakest precondition and a problem's specification. An algorithm is presented which uses background knowledge of the application domain to strengthen these similarities.

1. Introduction

The objective of this research is to explore the learning of heuristics by experience, and the hypothesis that a general problem solver shell can create useful domain dependent heuristics by such experience. To this end we have designed and implemented such a shell called FM. When applied to an application domain, it uses weak methods to acquire strong problem solving heuristics through experience, resulting in improved performance. FM can form heuristics to bring about improvements in search, representation change and goal ordering. In section 2 we present an overview of its architecture; in section 3 we describe at length, with the help of examples, the anatomy of one such weak method for reducing search within a goal directed control strategy.

Similar lines of research have been followed in the last few years which have inspired this work (Korf, 85; Laird, Rosenbloom & Newell, 86; Langley, 85; Carbonell, 83). They involve combining learning components with typical A.I. paradigms of Problem Solving, which are consequently used as the performance element for the learning techniques. A parallel effort is the development of the so called second generation expert system; here rules for shallowing reasoning can be created and modified incrementally from a deep application model when, during problem solving, the present rule set fails (e.g. Van de Velde 86). This work is aimed at improving knowledge intensive systems, and oriented towards the creation of rules within a knowledge base, rather than the improvement of search and representation in search intensive systems.

Our problem solver shell has been equipped with weak learning methods and designed and implemented with the following characteristics: it accepts application domains in the form of an environment (facts and rules) and a set of structured, transparent operators (in the sense of Porter & Kibler 84). Problems are posed as initial state and goal condition pairs, and either a state space or problem space search strategy can be selected. A dual format for learnt

heuristics is used: they are formed explicitly for search guidance but also appear implicitly through representation changes in the application domain's operators.

2. An Overview of FM

The main modules of FM are shown in figure 1, arrows showing data flow in and out of the two main processes; each component is present in modular form and therefore easily changed. A brief description of components is given below with the help of a simple robot world example shown in figure 2. FM has been tested with typical micro-worlds (e.g the Eight Puzzle, Tower of Hanoi, various robot worlds and blocks worlds). It exhibits learning during and after problem solving sessions (the former type has helped problem solving within that session); it forms heuristics that apply to problems with different initial and goal states, and also to both scaled up and reconfigured environments. Methods for learning three types of heuristic are being investigated: for search guidance, for representation change in operators and for goal decomposition.

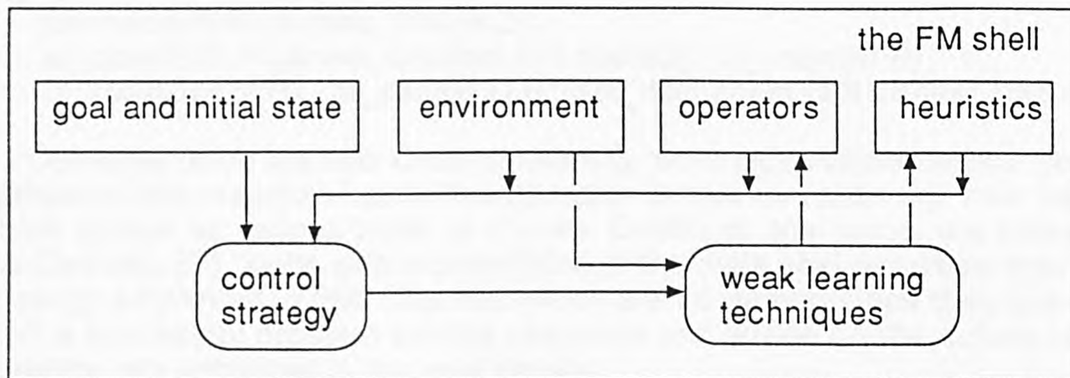

Figure 1: The FM achitecture

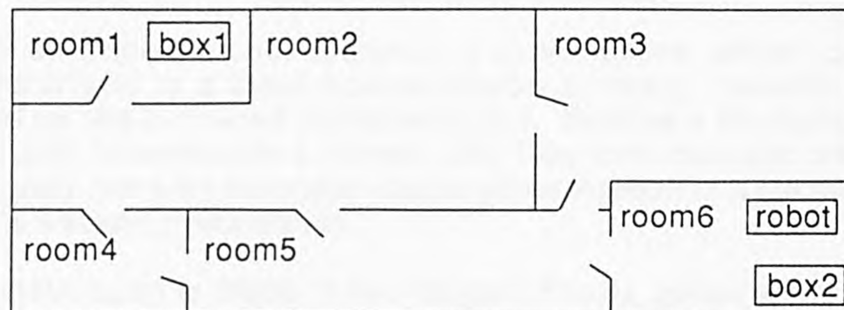

Figure 2: A Robot World

A goal and initial state are stated as conjunctions of literals in first order logic; e.g. goal = in_room(box1,room6)&closed(door56),
initial state = in_room(box1,room1)&in_room(robot,room6)&...open(door56)&...

The environment is a collection of facts and rules which act as background knowledge and include the typing taxonomy of objects;
e.g. environment = type_of(box1,box)&...type_of(room1,room)&...
connects(room5,room6,door56)&fits_thru(box1,door56)&...
(in_room(X,R)&next_to(X,Y) -> in_room(Y,R))&...
(at_door(X,D,R) -> in_room(X,R))&...

An operator O has five main parts, (O.e, O.p, O.a, O.s, O.d), representing environmental preconditions, main preconditions, add-list, side effects and delete-list, respectively. They are all specified as conjunctions of literals containing variables whose scope ranges throughout the operator, and are assigned a STRIPS-like semantics; (note: capital letters are used within predicates to represent variables in this paper)
e.g. pushthrudoor(B,D,R) =
(connects(R,R1,D)&fits_thru(B,D),
at_door(B,D,R1)&next_to(robot,B)&open(D), in_room(B,R),
in_room(robot,R), at_door(B,D,R1)&at_door(robot,D,R1)&next_to(B,X)& ...)

Currently there are two kinds of learning techniques implemented: compiling operators into macros in primitive operator format and creating new heuristics which appear as various types of chunks. Details of the former are presented in (McCluskey, 87), along with a description of the main goal reduction-type control strategy employed in FM. Chunks, which are so named since they are created from a successful problem solving sequence and advise on the future use of an operator, are explained in the next section.

3. Chunks in FM

3.1 State Space Chunks

We will describe the construction of a *simple* chunk, which can be used by FM as a heuristic in a state space search strategy, because it provides background for the b-chunks explained in 3.3. Bearing a similarity to the chunk of SOAR (Laird, Rosenbloom & Newell, 86), they form heuristic preconditions for (operator, goal) pairs by the model-based generalisation of a successful operator sequence's weakest precondition.

The construction is made in two stages: Firstly, given a sequence of fully instantiated operators {O(i): 1=< i =<n} achieving goal predicates G, form the quad Q = (O(i), G, WPs(i), WPe(i)) for each i. Here WPs(i), WPe(i) are the state and environment components of the weakest precondition WP(i) of the operator sequence {O(j) : i =<j =< n}, respectively, defined here by:

WPs(i) = P(n+1-i) where P(0) = G,
P(j) = (P(j-1) -- (O(n+1-j).a & O(n+1-j).s)) & O(n+1-j).p;

and WPe(i) = O(i).e&O(i+1).e& .. &O(n).e.

The symbols '--' and '&' mean set difference and set union, respectively. Since we are dealing with conjunctions of ground literals these operations are well defined.

Secondly, Q is generalised into a chunk in the following way: Every constant symbol that occurs in {O(j) : i=<j=<n} as a result of substitution for an operator variable, in the original proof of the operator preconditions, is generalised to a variable, with identical constants turning to identical variables, throughout Q. If two constants s, t of the same type occur, then they may be generalised to variables S, T, but an extra predicate 'not_equal(S,T)' is added to WPe(i). This has the special meaning that the variables may not be later instantiated to the same constant and, together with the other collected constraints in WPe(i), ensures a minimum generalisation.

3.2 Use of State Space Chunks

FM uses its constructed chunks described above in a forward state space search. Consider S, a conjunction of ground literals representing the 'current state', in a search for goal state or condition G, with respect to an environment E. Assume that the set of operator(s') instantiations that may be applied to S is Os; then any O in Os that maximises the size of the following set of chunks is heuristically favoured to form a successor to S:

{(O',G',WPs,WPe): (O,O')t unifies for some substitution set t such that G contains (G')t, S contains (WPs)t and E contains (WPe)t }

In other words the operator(s') instatiantions that 'match' the most chunks are chosen to continue the search.

Chunks created for application domains that warrant a state space search can significantly reduce search times. We have applied FM to such domains as the Eight Puzzle and achieved results supporting the claims of (Laird, Rosenbloom & Newell, 86). Below is a sample chunk created during problem solving in the Eight Puzzle world which would correspond to their 'symmetrical transfer' of learning, involving a cyclic movement of three tiles:

```
ch1 = (move(T1,X1,X2),
      at(T4,X4),
      at(T1,X1)&at(blank,X2)&at(T2,X4)&at(T4,X3),
      next(X2,X3)&next(X3,X4)&next(X4,X1)&next(X1,X2)&
      type_of(T1,tile)& ...type_of(X1,position)&...) .
```

Only one simple operator is needed with the relational representation we are using (i.e. move a tile T from position P1 to P2):

```
move(T,P1,P2) =
    (next(P1,P2)&type_of(T,tile)& ...,  at(T,P1)&at(blank,P2),
     at(T,P1)&at(blank,P2),  _,  at(T,P1)&at(blank,P2))
```

3.3 The B-chunk

FM is equipped with a goal reduction strategy which can search through a space of nodes representing partial solutions to a problem (McCluskey, 87). Nodes may be generated by adding an operator that achieves desirable goal(s) to a parent's partial solution. The goal of the new node is to solve the operator's preconditions (similar to the goal node search of (Dawson & Siklossy, 77)). Nodes may also be generated to change operator ordering in the face of conflicting goals, or to add a whole sequence of operators which solve one particular subgoal. The chunks described above do not, of course, apply here since they rely on the idea of matching a current state. To improve the goal reduction search FM analyses a successful operator sequence that solves some goal node (modelled simply as (State, Goal, Environment) below), and creates b-chunks.

The general idea is as follows: Consider a fully instantiated operator sequence {O(i): 1=<i=<n} that solves some goal node (S,G,E). For each O(i), the features occurring in (S,G,E) which seem to be the reasons why O(i) occurred in the sequence, are stored in the form of a b-chunk. Another problem (or node) (S',G',E'), which matches this chunk (see 3.4 for the precise definition of matching), would then have that particular operator proposed (along with the constraints embedded in the chunk) to be part of the solution sequence. In the present version of FM, however, the chunk is used more as a operator application discriminator than a proposer i.e. chunks cut down the branching factor of goal directed search.

We will now introduce an example operator sequence that will be used to clarify the b-chunk idea. Using the figure 2 as a diagramatic description of the initial state, with G= in_room(box1,room3), let
{O(1), O(2), ... O(11)} = {gotodoor(door56,room6), gothru(door56,room5), ...
pushthru(door23,box1,room3) }.

Using similar techniques to section 3.1 above, we form the core, C(i), of a b-chunk's heuristic precondition on states as the conjunction of the set:

{P in WPs(1) : WPs(i)&E => P}

e.g. consider O(9) = pushthru(box1,door12,room2) above. Then with
WPs(9) = at_door(box1,door12,room1)&open(door12)& ...,
E = (at_door(X,Y,Z) -> in_room(X,Z)))& ...,
WPs(1) = in_room(box1,room1)&open(door12)& ...,
we have C(i) = in_room(box1,room1)&open(door12)&open(door23).

A b-chunk is now formed by generalising (O(i),G,C(i),WPe(i)) as in 3.1. The chunk for the above example would be:

```
ch2 = (pushthru(B1,D1,R2),
       in_room(B1,R3),
       in_room(B1,R1)&open(D1)&open(D2),
       connect(R1,R2,D1)&connect(R2,R3,D2)&fits_thru(B1,D1)& ... ).
```

This chunk would discriminate against other possible operator instantiations in the goal directed search since it incorporates similarities to the initial state that only pushing the box from room1 would satisfy. Chunks formed from operators that occur towards the end of a sequence usually turn out not to be discriminatory. In this case, FM 'strengthens' them by the following iterative process:

W1(0) := C(i); W2(0) := WPe(i); j := 1;
repeat
(X, Y) := {x, y : x is a predicate in {WPs(i) -- C(i)} and is related to some predicate in WPs(1)} by an association chain y of length j in E};
W1(j) := W1(j-1)&X; W2(j) := W2(j-1)&Y; j := j+1
until either the generalised chunk (O,G,W1(j),W2(j)) is discriminatory, or some complexity bound is reached.

The technique of using association chains is explained in (Vere, 77). The stengthening process adds important features that connect WPs(i) and WPs(1) to the last two components of the chunk. Consider creating a chunk for pushthru(box1,door23,room3) occurring as O(11) in our example. Then C(i) = open(door23), which would not discriminate against pushthru(box1, door35,room3). After the action of this process (for j=1 only) the chunk is now more useful:

ch3 = (pushthru(B1,D1,R1),
in_room(B1,R1),
in_room(B1,R3)&open(D1)&open(D2),
connects(R1,R2,D1)&connects(R2,R3,D2)&fits_thru(B1,D1)& ...)

3.4 The Use of B-chunks

Consider an operator instantiation O which has been proposed as part of a solution sequence to a node (S,G,E). B-chunks that match the operator and the node's three components in the following way, favour that instantiation (similar to 3.2): for b-chunk (O',G',WPs,WPe) we require (O,O')t to unify for some substitution set t such that G, or one of the ancestors of (S,G,E)'s goals, contains (G')t, S contains (WPs)t and E contains (WPe)t . Consider our simple example again, after the eleven operator sequence to transport the box into room3 has been 'executed'. During the solving of a new task 'in_room(box2,room3)', the chunk 'ch3' will advise in favour of the operator instantiation pushthru(box2,door35,room5), since it conforms to the chunk's constraints within this task specification.

4. Conclusions and Future Work

We have given an overview of a general problem solver shell employed in exploring weak methods for learning, and described a novel method, that of creating b-chunks, for cutting down search in goal directed systems. B-chunks integrate well with another method used in FM, that of *closed* macro creation (McCluskey, 87), since they advise on the use of both primitive and macro

operators. This is important since although the latter reduce the depth of search, the branching factor increases, and so the need for an improving heuristic is more acute. B-chunks can be adapted to advise on goal ordering and we hope to to extend their application to non-linear planning processes. We also envisage optimising the chunks created using techniques fron Incremental Concept Induction and further feedback monitoring.

We have sidestepped many problems e.g. the system relies on successful operator sequences before learning can begin, and makes the assumption that sequences are optimal. It has been useful testing FM on micro-worlds, since experimenting with different learning methods and representation shifts is easier, but we are at present embedding it in a 'real world' robot inspection problem. Regarding the implementation, we have found that the complexity of the strengthening algorithm given in 3.4 constrains its use to small values of 'j', and its performance depends heavily on the initial domain representation. Finally, the system, as described, has been implemented and runs in C-Prolog.

References

Carbonell, J.G. (1983). *Learning by Analogy*, in R.S. Michalski,J.G. Carbonell, Mitchell, T.M. (eds.). *Machine learning, An artificial intelligence approch.* Los Altos, CA: Morgan Kaufmann.

Dawson, C. and Siklossy, L. (1977). *The Role of Preprocessing in Problem Solving Systems*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, (pp 465-471), Cambridge, Mass: Morgan Kaufmann.

Korf, R. E., (1985). *Macro operators: a weak method for learning*, Artificial Intelligence 26, (pp 35-77).

Laird, J., Rosenbloom, P. and Newell, A. (1986). *Chunking in Soar: the anatomy of a general learning mechanism*, Machine Learning 1, (pp 11-46).

Langley, P. (1985). *Learning to search: from weak methods to domain- specific heuristics* , Cognitive Science 9, (pp 217-260).

McCluskey, T.L. (1987). *Machine learning of heuristics in general problem solvers*, Report TCU/C5/1987/5, Computer Science Dept., City University, London.

Porter, B. and Kibler, D. (1984). *Learning operator transformations*, Proceedings of the National Conference on Artificial Intelligence, (pp. 278-282). Austin,TX: Morgan Kaufmann.

Vere, S.A. (1977). *Induction of relational productions in the presence of background information.* In Proceedings of the Fifth International Conference on Artificial Intelligence, (pp. 349- 355). Cambridge, MA: Morgan Kaufmann.

APPENDIX D.3

[McCluskey 87c]

"Combing Weak Learning Heurististics in General Problem Solvers"
in Proceedings of the Tenth International Joint Conference on Artificial Intelligence", Morgan Kaufman, August 1987.

COMBINING WEAK LEARNING HEURISTICS IN GENERAL PROBLEM SOLVERS

T.L. McCLUSKEY

ABSTRACT

This paper is concerned with state space problem solvers that achieve generality by learning strong heuristics through experience in a particular domain. We specifically consider two ways of learning by analysing past solutions that can improve future problem solving: creating macros and the chunks. A method of learning search heuristics is specified which is related to 'chunking' but which complements the use of macros within a goal directed system. An example of the creation and combined use of macros and chunks, taken from an implemented system, is described.

I INTRODUCTION

Integrating ideas and techniques devoloped in Machine Learning, with those of Problem Solving, has attracted substantial recent research effort (e.g. [Laird et al 86], [Korf 85], [Langley 85], [Mitchell et al 83]). An important aspect is the revival of the 'general' problem solver. Its demise was due in part to the failure of its weak heuristics to tackle problems of complexity in some given application domain; now it returns equipped with not just weak problem solving heuristics but with weak heuristics for learning strong, i.e.domain dependent, heuristics. The latter may take the form of useful shifts in the problem space representation (a simple example is the learning of macro operators) or improving search through a particular space by the acquisition of search control heuristics. Thus, while its generallity is maintained, learning may improve the problem solver's efficiency during the application to a particular domain. This is the approach we have taken in the construstion of a 'heuristic learning problem solver shell' called FM; it can acquire strong heuristics from problem solving experience when it is applied to specific domains. A complementary approach is to acquire or discover them during a preprocessing stage as in [Iba 85], [Korf 85]) and [Dawson & Siklossy 77].

FM's application domains can have variable initial and goal states. Applications are interchangeable by specifying domain environments, states and goals as expressions in first order logic, and operators in terms of structured add, delete and precondition predicates. Control stategies may be interchanged (e.g. forward best-first or goal reduction) as can weak learning methods such as macro and chunk creation.

This constitutes a more general approach to recent work on heuristic learning in problem solvers (e.g. [Mitchell et al 83], [Korf 85]), where systems typically improve in domains with a fixed goal, employ a more specialised representation scheme, and a forward state space search strategy. This paper will outline FM's goal directed search and describe how macros and chunks are created and used as complementary heuristics during that search.

II GOAL NODE SEARCH IN FM

The backward search of FM proceeds in a goal reduction manner, starting with the initial goal, through a space of goal nodes (similar to those in [Dawson & Siklossy 77]). Each goal node can be modelled as a 6-tuple:

(identifier, goal, initial state, ancestors, purpose, trace).

The trace records attempts to solve the goal, whereas the purpose records why the goal node was created (typically to solve the unsatisfied preconditions of an operator). Goals, expressed as conjunctions of predicates, are initially assumed to be decomposable: when a goal node is activated, operator instantiations which add goal predicates have their unsatisfied preconditions form another goal node, unless they are already satisfied in which case those operators are applied to the initial state and the result recorded in the trace.

When the trace of a goal node eventually contains a state satisfying its goal (via an operator sequence Os), we say that the goal node is solved, and all nodes which are ancestors of it are removed from the search. If it was activated to solve an operator O's preconditions, then the sequence Os + O is applied to the goal node's parent's initial state and the result recorded in the parent's trace.

A goal node's initial state may be the state inherited from a parent node, or may be an advanced state partially satisfying the parent's goal. The latter is the case when goals cannot be solved by simple decompositon; FM examines the trace and forms new goal nodes whose goal predicates are inherited but whose initial states are selected from intermediate states taken from the parent's trace.

The kind of representation of goal nodes outined above aids both the formation and use of strong heuristics. The trace is available for analysis and criticism after the solution of each goal node, allowing 'within-trial transfer of learning' (see [Laird et al 84]) to take place. In our implementation of FM we have experimented with the formation of closed macros, 'b-chunks' and also subgoal ordering heuristics at this stage, but we shall limit our discussion to the first two.

III CLOSED MACRO CREATION

We consider a closed macro operator to be an operator sequence that has been compiled and generalised into a form similar to that of a primitive operator (in contrast to the 'open' macrops of [Fikes et al 72]). This sequence forms part of a past solution, in the case of learning by experience, which includes fully instantiated operators and intermediate states. Here the compilation involves finding the sequence's weakest precondition through the intermediate states and using it as the macro's precondition. Within this certain constants can then be selectively generalised using a technique similar to the Explanation-Based Learning of [Mitchell et al 86].

Systems that learn closed macros ([Minton 85], [Iba 85]) seem to demonstrate significant improvement in problem solving within robot and puzzle worlds but there are pitfalls in using this technique as the sole learning component:

-search trees do shorten but unfortunately grow bushy since distinct instantiations of macros proliferate. (This is reminisent of the effect of paramodulation, a 'macro inference rule' in Theorem Proving, which combines resolution with the axioms of equality, but when used in search changes long thin trees to short bushy ones!).

-solutions which comprise of closed macros are prone to produce non-optimal paths even after checks for redundant primitive operator sequences have been made.

We claim that such problems may be overcome by the learning of strong heuristics such as chunks to complement the use of macros.

Macros are created and stored in FM when goal nodes are solved, and then are immediately available for use in problem solving. Each are compiled from a successful operator sequence into a primitive operator format. The major part of this compilation process is in building up the precondition M.p (a conjunction of predicates) of a macro M. This is accomplished by a procedure modelled on goal regression equations:

$$M.p = P_n \text{ where } P_0 = G \text{ and}$$
$$P_i = (P_{i-1} -- O[n+1-i].a) \cup O[n+1-i].p, \; i = 1 \text{ to } n$$

where 'U' and '--' mean set union and difference, O[i].p, O[i].a stand for the precondition and add predicates of operator i respectively, and G the goal predicates for the solution sequence.

Constants that appeared as arbitrary members of some particular type in the solution's operator sequence are carefully generalised to a variable with that type restriction (following [Kodratoff 84]). Generalisation is justified since no operator in the solution sequence referred to the constant specifically but only to its type. Identical constants are generalised to the same variable throughout the macro, but equality binding restrictions are added where variables of the same type are generalised from distinct constants, so that they may not be instantiated to the same constant when in use. Macros are then incorporated into future problem solving as primitive operators, although some may later be deleted if rarely used.

IV B-CHUNK CREATION

The chunks created by FM improve the system's subsequent problem solving behaviour by providing search control knowledge. They are formed during the goal directed search and advise on the search through partial solutions. The absence of such a learning component in STRIPS with Macrops is pointed out in [Porter and Kibler 84] and Minton's Morris system [Minton 85] apparently combines only weak search heuristics with the use of macros.

Consider O[i] $(1<i<n)$ taken from an operator sequence O[1],O[2], ...,O[n] which achieves a goal node (with goal predicate(s) G) from a initial state I within a domain environment E (E is a set of facts and rules constituting background knowledge for a particular application). A b-chunk (O[i]'; G'; P') is built for each O[i] to the following specification: consider a function 'sim':

$$\text{sim} : CP \times CP \times CP \times Nat0 \dashrightarrow CP$$

where CP is the space of conjunctions (or sets) of predicates, and

sim(X,Y,E,0) = {P in Y: P logically follows from X&E}

sim(X,Y,E,N) = sim(X,Y,E,N-1) union
{y el. of Y, e subset of E : y is related to an x in X
by an association chain e of length N}

Then P= sim(M(i),M(1),E,K) where M(j)= the macro precondition (see section III) of sequence O[j],O[j+1], ...,O[n]; $K \geq 0$, and finally

(O[i]'; G'; P') = the careful generalisation of (O[i]; G; P).

When K = 0 then O[i]'s chunk's third component may be roughly described as those predicates which were present in the goal node's initial state and that were also involved in the achievement of G after O[i-1]. This includes environment information (which is assumed to be a part of every state) that has been used in the satisfaction of the operator's preconditions. FM initially forms P with K=0 and then checks to see if the resulting chunk would be discriminatory if used to solve the same goal node again. If it is not the case then K is incremented and P is augmented with predicates using an 'association chain' technique similar to that described in [Vere 77].

B-chunks are then used during subsequent search when FM finds multiple operators (or operator instantiations) are available to achieve a goal predicate Gp, but none of their preconditions are completely satisfied. A b-chunk (O1; G1; P) will favour an operator instantiation O applied to a goal node if P logically follows from I&E under the variable bindings obtained by the successful matching of O1 to O, and G1 to either Gp or one of Gp's ancestors. The instantiation(s) favoured by the most chunks is then chosen to form a new goal node.

V COMBINED USE OF LEARNT HEURISTICS

To clarify the combined use of closed macros and b-chunks we use a simple example. We applied FM to a robot world using a similar operator set to [Fikes et al 72]. After box moving tasks it forms macros such as:

```
( name: macro21(Rm1, Dr1,Rm2,Box,Dr2,Rm3),
  preconditions: in_room(Box,Rm1)&next_to(robot,Box)
  &connect(Rm1,Rm2,Dr1)&connect(Rm2,Rm3,Dr2) ...,
  add: in_room(Box,Rm3),
  side_effects: in_room(robot,Rm3), ...   ).
```

Macro21 is equivalent to the primitive sequence:

```
{pushto(Box,Dr1,Rm1), pushthru(Box,Dr1,Rm2),
 pushto(Box,Dr2,Rm2), pushthru(Box,Dr2,Rm3) }.
```

In solving the goal 'in_room(boxA, room4)' from the situation in figure 1, macro21 constitutes the part of the solution shown by an arrow. One b-chunk (where K=1 in section IV) created to advise on its use is (note: we leave out some details; capital letters denote variables):

```
( macro21(Rm1, Dr1,Rm2,Box,Dr2,Rm3) ;
  in_room(Box,Rm3) ;
  in_room(Box,Rm4)&connect(Rm4,Rm1,Dr3)&
  connect(Rm1,Rm2,Dr1)&connect(Rm2,Rm3,Dr2)& ...)
```

In a future problem, this chunk will support the inclusion of instantiations of macro21 in partial solutions which conform to its constraints. For instance, consider task in_room(boxB,room6). It can be seen by the description of chunk use in section IV that instance macro21(room4,door47,room7,boxB,door67,room6) is favoured by the chunk shown above to form the first part of a solution, resulting in a filtering out of any other undersirable instantiations. Note that this chunk suggests the initial position of the robot is irrellevant.

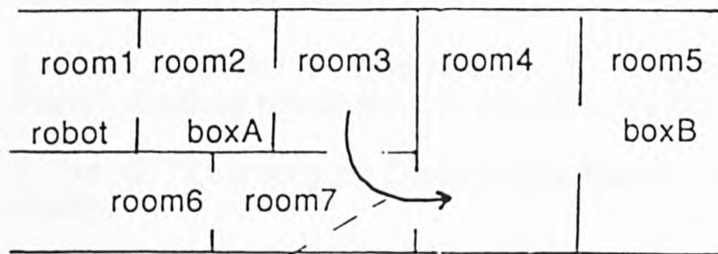

figure 1. (Note: doorXY connects roomX and roomY)

VI CONCLUSIONS

We have described a goal directed search which allows the use of weak methods for learning. Given a particular domain, these weak methods create strong heuristics, in the form of macros and b-chunks, through the experience of successful problem solving. The chunks record for each operator and generalised goal pair, the adviseable instantiations for operator variables. They do this by storing important similarities among the environment, initial state and goal in a form usable for future goal directed search. The number of possible instantiations of macros in the backward search tends to be much higher than primitives, and so the need for this heuristic pruning is greater.

We have used FM in several applications in which it builds up strong domain dependent heuristics by experience. Of particular note is the b-chunks' high degree of accross-task transfer of learning. This is because they record quite general similarities between the components of a problem space such that when these similarities are encountered again the choice of (macro) operator instantiation can be determined.

REFERENCES

1. Dawson, C. and Siklossy, L. "The Role of Preprocessing in Problem Solving Systems". In Proc. IJCAI-77, (1977).

2. Fikes, R., Hart, P. and Nilsson, N. "Learning and Executing Generalised Robot Plans", Artificial Intelligence 3, 251-288, (1972).

3. Iba, G. "Learning by Discovering Macros in Puzzle Solving". In Proc. IJCAI-85, (1985).

4. Kodratoff, I., "Careful Generalisation for Concept Learning", In Proc. ECAI-84, (1984).

5. Korf, R. E., "Macro Operators: A Weak Method for Learning", Artificial Intelligence 26, 35-77, (1985).

6. Laird, J., Rosenbloom, P. and Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism" Machine Learning 1, 11-46, (1986).

7. Laird, J., Rosenbloom, P. and Newell, A. "Toward Chunking as a General Learning Mechanism", In Proc. AAAI-84, (1984).

8. Langley, P. "Learning to Search: From Weak Methods to Domain-Specific Heuristics", Cognitive Science 9, 217-260, (1985).

9. Minton, S. "Selectively Generalising Plans for Problem Solving", In Proc. IJCAI-85 (1985).

10. Mitchell, T.M., Utgoff, P.E. and Banerji, R.B. "Learning by Experimentation: Acquiring and Refining Problem Solving Heuristics", in "Machine Learning", Michalski, R.S., Carbonell, J.G. and Mitchell, T.M. (eds.), Tioga Publishing, (1983).

11. Mitchell, T.M. "Explanation-Based Genralisation: A Unifying View" Machine Learning 1, 47-80, (1986).

12. Porter, B. and Kibler, D. "Learning Operator Transformations", In Proc. AAAI, (1984).

13. Vere, S.A., "Induction of Relational Productions in the Presence of Background Information", In Proc. IJCAI-77 (1977).

APPENDIX D.4

[McCluskey 87d]

"A Weak Learning Method for use in Goal Directed Search"
poster given to the Fouth International Workshop on Machine Learning, University of California at Irvine, June 1987, and Computer Science Technical Report TCU/CS/!987/15

The Anatomy of a Weak Learning Method for Use in Goal Directed Search

by Lee McCluskey,

Dept. of Computer Science,
The City University,
London,
England.

research area:

Learning from problem solving experience; related to research in [1],[2],[3],[4].

poster topic:

Outline description of a general problem solving shell called **FM** which has been built to experiment with various learning methods.

Details of one such learning method (the b-chunk) for use in task domains requiring goal directed problem solving strategies.

theses being investigated:

(1) A general problem solver can significantly improve its performance through experience in a particular domain, using weak methods for creating problem solving heuristics in areas such as:

- search control
- representation change
- goal ordering

(2) Useful operator heuristics can be learnt from the model based generalisation of similarities between a solved problem's specification and the reasons why the operator appeared in the solving sequence.

FM's inputs:

- task ... (initial state, goal condition) = (S,G)

- environment ... (facts,rules) = (E.f,E.r)

- operators ... (environmental conditions, preconditions, adds, side-effects, deletes) = (O.e, O.p, O.a, O.s, O.d)

- search stategy ... currently either means-end goal directed, or best first forward.

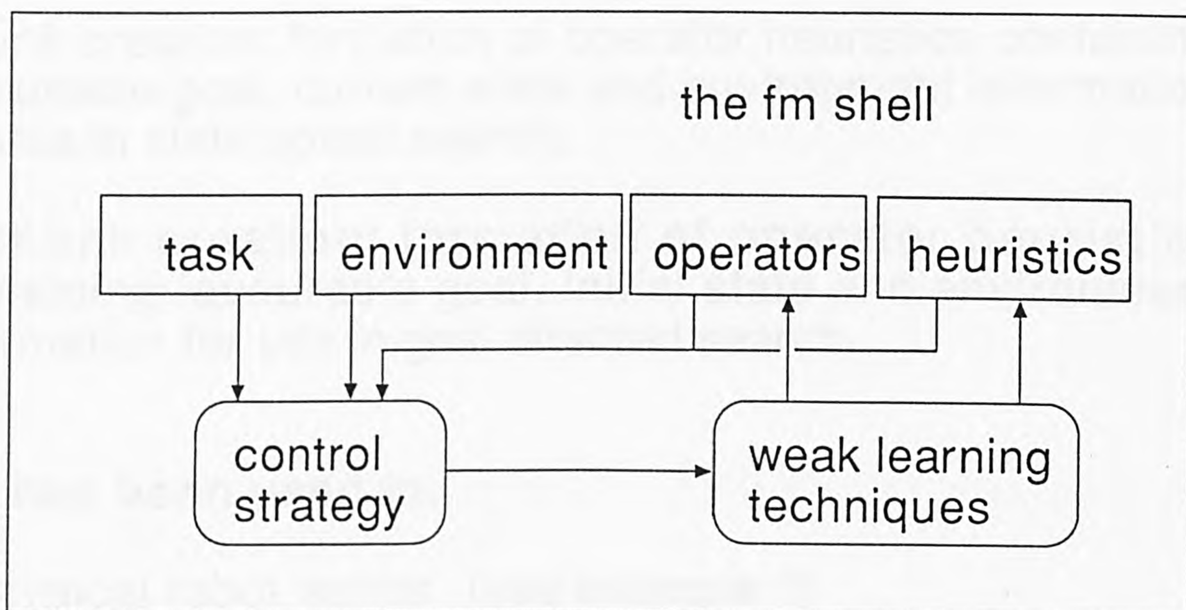

figure 1: **flow diagram of the FM shell**

FM's representations:

Components of operators, states, environment facts and goals are predicate conjunctions.

Search strategies can be interchanged but must be based around search through "goal nodes", which roughly represent partial solutions.

FM's current weak learning techniques:

Macro creation: formation of powerful operators from subsequences of more primitive ones, built in the same format.

Chunk creation: formation of operator heuristics containing favourable goal, current state and environment information for use in state space search.

B-chunk creation: formation of operator heuristics containing favourable goal, initial state and environment information for use in goal directed search.

FM has been used in:

typical robot worlds (see example 1)
blocks worlds
'tower of hanoi' puzzles
'eight' puzzles
story model (see example 2)

The B-chunk

The **form** of a chunk is (O, G, W1, W2), where

- O is the parameterized operator name
- G is the goal condition
- W1 is an initial state condition
- W2 is an environment condition

and the quad has been carefully generalised.

FM **uses** them to help find plans. Roughly, a b-chunk proposes the inclusion of an operator O with the variable bindings that occured from the successful matching of W1, G, W2 to the current task definition and environment S, G, E, respectively.

A b-chunk is **built** for operator O(i) out of a sequence O(1), ..., O(n), starting with

(O(i), G, W1(0), WPe(i))

as the the root (see figure2), and then applying strengthening and generalising processes to it.

Building B-chunks (1)

Consider a successful operator sequence
O(1), ..., O(n) for task (S,G).
Let WPs(i) = WP([Oi,...,On], G):

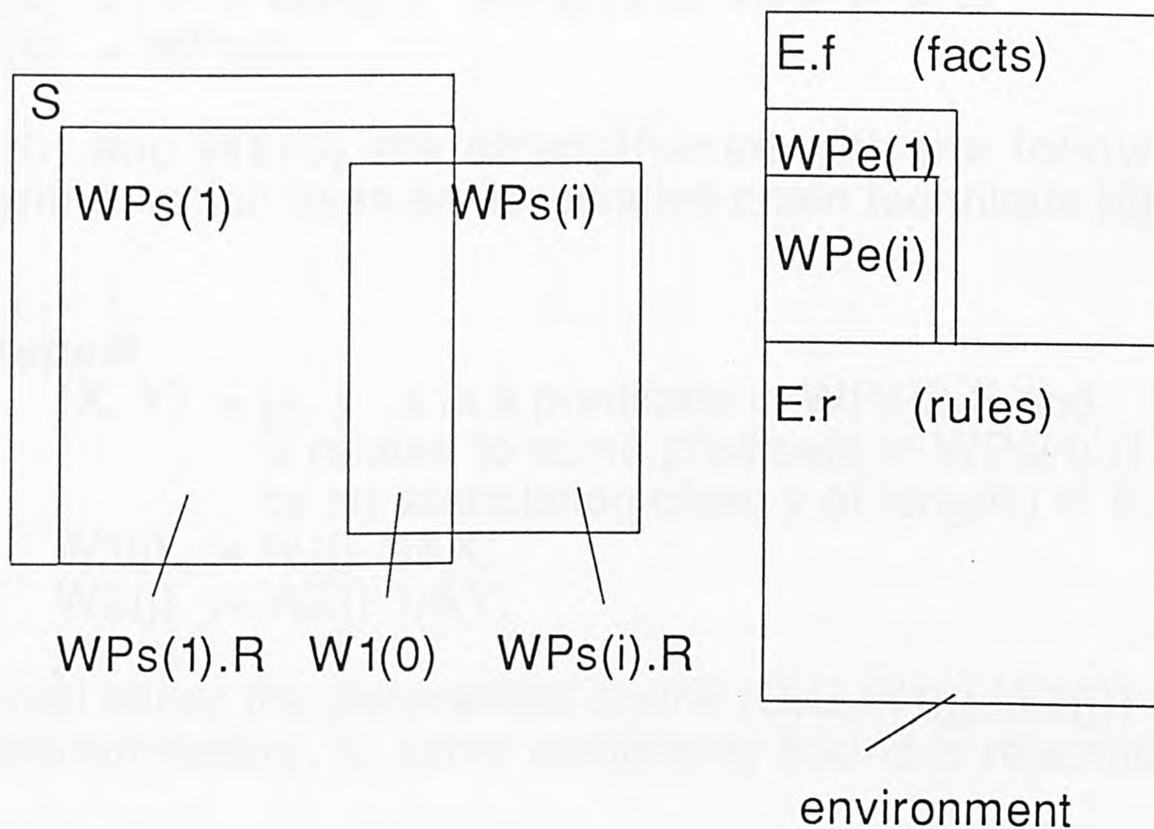

figure 2: **collecting weakest preconditions**

Using 'transparent' operators, we have:

WPs(i) = P(n+1-i) where
P(0) = G,
P(j) = (P(j-1) -- (O(n+1-j).a&O(n+1-j).s)) & O(n+1-j).p

WPe(i) = O(i).e&...&O(n).e

Building B-chunks (2)

To build a b-chunk for operator O(i), 1<i<=n, we start with the quad of ground predicates:

(O(i), G, W1(0), W2(0))   where
W1(0) = { P in WPs(1) : WPs(i)&E.r => P }   and
W2(0) = WPe(i)

W1(0) and W2(0) are strengthened with the following algorithm which uses an association chain technique [5]:

```
 j := 1;
repeat
   (X, Y) := {x, y : x is a predicate in WPs(i).R and
              is related to some predicate in WPs(1).R
              by an association chain y of length j in E.f};
   W1(j)  := W1(j-1)&X;
   W2(j)  := W2(j-1)&Y;
   j := j+1
until either the generalised chunk (O,G,W1(j),W2(j)) is
discriminatory, or some complexity bound is reached.
```

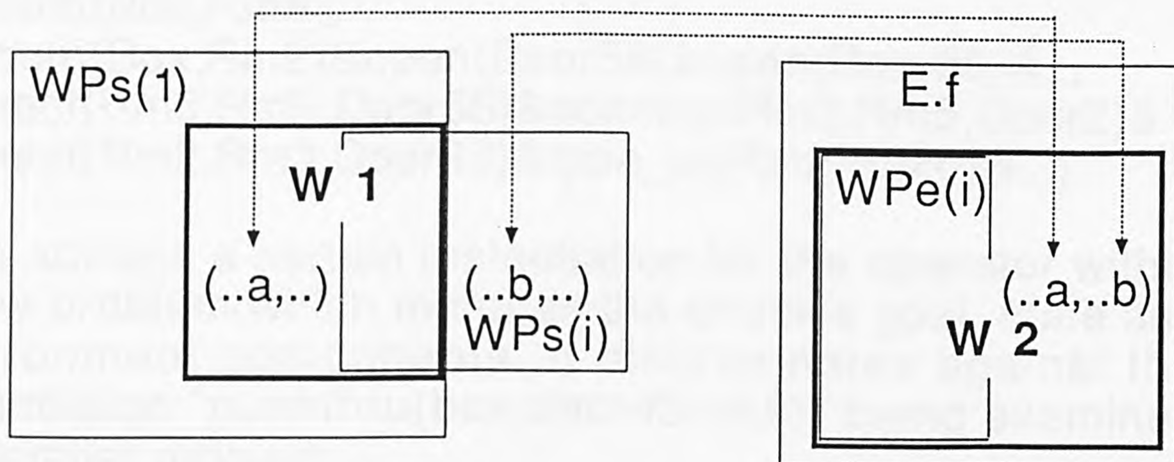

figure 3: **chunk build-up through associations**

Example 1: robot world

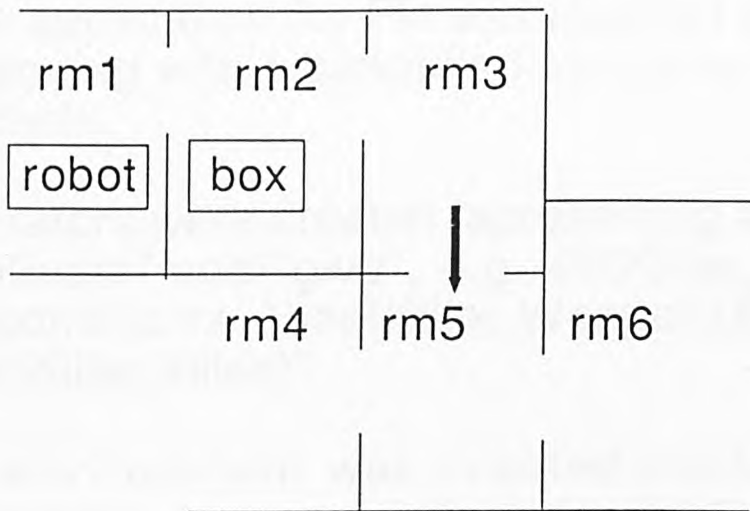

figure 4: **part of a robot world**

Assume the usual STRIPS-like model, with the initial state as in figure 4. FM solves goal "in_room(box,room6)" using a nine operator sequence, during which b-chunks are formed, e.g. for operator "pushthru", marked by an arrow:

(pushthru(Box,Door35,Rm5),
in_room(Box,Rm6),
in_room(Box,Rm2)&open(Door56)&open(Door35)&..,
connect(Rm5,Rm6,Door56)&connect(Rm3,Rm5,Door2)&
connect(Rm2.Rm3.Door23)&type_of(Rm2,room)&...)

This advises a certain instantiation for the operator within a new problem which matches the chunk's goal, state and environment components. It discriminates against the instantiation "pushthru(box,door45,rm5)" being examined in a similar problem.

Example 2: story model

This application for FM was inspired by an example in [6]: reasoning with a simplified storyline from Shakespeare's Macbeth.

Operators were created representing actions such as "kill", "motivate" and "give", e.g. kill(Killer,Weapon,Killed) has preconditions "has(Killer,Weapon)&has(Killer,Motive)& near(Killer,Killed)".

An environment was created containing the obvious taxonomic, property and relational information.
FM was given, and solved, the following task,

(wants(lady_macbeth,duncan,dead)&has(lady_macbeth, a _dagger)&has(macduff,sword)&alive(duncan)&....,
murdered(duncan))

A typical b-chunk formed after solving this goal was:

(kill(Murderer,Weapon,Murdered),
killed(Murdered),
wants(Accomplice,Murdered,dead)&has(Accomplice, Weapon),
married(Murderer,Accomplice)&can_influence(Accomplice ,Murderer)&is_evil(Murderer)¬_equal(Murderer, Murdered)&,....)

This chunk could have the more general interpretation "Someone can be killed by an evil person if currently the person's wife wants the someone dead, has a weapon and can infuence her husband", rather than being only a heuristic for future search.

Results and Conclusions

The development of FM is still at an early stage (1 year old!) but results are encouraging with respect to the generality of the weak learning methods.

The b-chunks' strength lies in their ability to embody relationships between operator sequences preconditions and initial states, using an environment as background knowledge. In all the applications mentioned, chunks have increased FM's performance in both similar and more complex tasks to those from which they were learned.

One of the biggest problems is in the proliferation and complexity of the chunks created; we envisage this can be alleviated by inductive concept learning.

References

[1] Korf, R. E., "Macro Operators: A Weak Method for Learning", Artificial Intelligence 26, (1985).
[2] Laird, J., Rosenbloom, P. and Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism" Machine Learning 1, (1986).
[3] Langley, P. "Learning to Search: From Weak Methods to Domain-Specific Heuristics", Cognitive Science 9, (1985).
[4] Mitchell, T.M. "Explanation-Based Genralisation: A Unifying View" Machine Learning 1, (1986).
[5] Vere, S.A., "Induction of Relational Productions in the Presence of Background Information", In Proc. IJCAI-77 (1977).
[6] Winston, P.H. "Learning New Principles from Precedents and Exercises", Artificial Intelligence 19,(1982)

APPENDIX D.5

[McCluskey 88a]

"Deriving a Correct Logic Program from the Formal Specification of a Non-Linear Planner",
in Methodologies for Intelligent Systems, 3, North-Holland, October 88.

DERIVING A CORRECT LOGIC PROGRAM FROM THE FORMAL SPECIFICATION OF A NON-LINEAR PLANNER

T.L. McCluskey
Computer Science Department, The City University,
Northampton Square, London, England.

ABSTRACT

The specification, design and implementation of Intelligent Systems has much to gain from the field of Formal Methods in Software Development. Such rigorous methodologies are well suited to the symbolic processing typically found in artificial intelligence applications. This paper describes the development of an application independent non-linear planner from a formal specification, relying on logical transformation to arrive at an executable logic program. We propose that this example acts as a paradigm for the development of similar problem solving systems.

1. INTRODUCTION

A Non-Linear Planner (NLP) is a problem solver which searches for a solution to a problem by generating a space of partial plans, where each partial plan is a collection of partially ordered and partially instantiated operators. The NLP technique was popularised by Sacerdoti's NOAH system in [9], and later work (e.g. [1]) has reinforced the method. We refer the reader to these references for background information.

Logic Programming is recognised as a powerful implementational paradigm for symbolic applications. It's logical base and non-deterministic flavour make it closely related to specification languages. One such feature, used in this paper, is to apply logical transformation to derive a logic program from a specification. Simple examples of its advantages in program correctness are often quoted in the literature (e.g. [4]) ; we are supplementing this with an interesting and non-trivial example: that of rigorously implementing an NLP.

We consider an NLP domain independent if a user can supply the declarative definition of a domain (for example by supplying a set of 'strips-type' operators as in [3]). It is conjunctive if it expects its goal and initial state to be specified as a conjunction of predicates. The primary operation of a conjunctive, domain independent NLP is called 'goal achievement': that is making an outstanding goal in a partial plan true at some point in its operator partial ordering. Recently, a modal logic specification of goal achievement, called the 'Modal Truth Criterion' was postulated in by Chapman in [1]. Our work draws from Chapman's paper the essence of his Modal Truth Criterion, but while his concern is chiefly the properties of the specification (e.g. completeness), we concentrate on the rigorous design and implementation aspects.

The content of this paper is as follows: first we introduce a notational framework for problem solving, then constructively describe the central data type in an NLP: the partial plan. After outlining a top level algorithm for the NLP, we define a data type invariant for all valid partial plans, then use this to concisely specify the central operations in the planner, those of goal achievement. Next we construct a top level executable logic program for goal achievement using the underlying

semantics of logic programming to preserve the meaning of the specification.

2. PROBLEM SOLVING NOTATION

The input to an NLP is the language of 'task specifications'. We define a task specification to be the quad (I,G,E,OS), where I is an initial state, G is a goal condition, E is an environment, and OS is an operator set. We will assume that I and G are conjunctions of predicates; E is a conjunction of ground predicates, representing the unchanging information in an application, and OS is a set of operator schemas, representing actions, and conforming to the 'strips assumptions' [3]. A common example application for this type of framework is that of a simulated robot world. Here OS would contain an operator schema for each type of action the robot may perform, and E would contain background knowledge such as room connections and object typing.

An operator schema O can be represented as a logical term of the form (O.n,O.e,O.p,O.a,O.d) where O.n is the schema's name, and the other components are sets of predicates corresponding to environment and state preconditions, add and delete sets, respectively. A solution to a task specification (I,G,E,OS) is then a partially ordered set of instantiated operators taken from OS, such that when applied to I in a sequence conforming to the partial order, they will produce a state containing G. For more details of this problem solving framework, and its use in as a performance element in a heuristic acquisition system, see [6].

3. THE STRUCTURE OF A PARTIAL PLAN

An NLP searches through a space of partial plans for a solution to some task (I,G,E,OS). We define the structure of a partial plan as a logical term of the form pp(Os,Ts,Ps,As,Es) where:

(i) Os is a set of partially instantiated operators, where each has a unique identifier, and is an instance of a schema in OS. Two special operators called 'init' and 'goal' are the only exceptions: init contains an add set, representing the task's initial state I, and goal contains a set of precondition predicates, representing the main goal G.

(ii) Ts specifies a consistent partial order on the set Os, representing temporal precedence. For two operator instances O1 and O2 from Os, a temporal relation can be denoted 't(O1,O2)'. We can then define predicate before(X,Y,Ts), meaning that X is necessarily before Y in the partial order specified by Ts:

for all operators X,Y and sets of binary relations on them Ts (of the form t(X,Y)):
before(X,Y,Ts) <- { member(t(X,Y),Ts) V
there exists operator Z:
(member(t(X,Z),Ts) & before(Z,Y,Ts)) }.

The operators init and goal are by definition the lower and upper bounds of the temporal partial order Ts, respectively:

for all O in Os-{init,goal}: (before(O,goal,Ts) & before(init,O,Ts))

(iii) Ps is a set of pairs (P,O) where each P is an *unachieved* precondition of some operator O from Os. If the goal condition G is a conjunction g1&g2& ... gn,

then initially Ps is set to {(g1,goal), (g2,goal), ... (gn,goal)}. Whenever an operator instance is added to a partial plan, its preconditions O.p are added to Ps, in this form.

(iv) As is a set of pairs (P,O) where each P is an *achieved* precondition of some operator O from Os. The definition of precondition achievement is given in section 5 below.

(v) Es is a set of predicates which constrain operator variables. For example, as operator instances are added to Os, their environmental preconditions, along with any necessary variable constraints, are added to Es. These constraints must be consistent with the task's environment E (note that operator variables have scope throughout the partial plan, since we model it as a logical term). We can then define the following second order *predicate* 'unify':

for all predicates P and Q:
unify(P,Q,Es) <- P and Q can possibly unify under the constraints imposed by Es and E.

4. A SIMPLE TOP LEVEL ALGORITHM FOR THE NON-LINEAR PLANNER

For any task (I,g1&g2&..&gn,E,OS), search starts with an initial partial plan called PPi, its Os consisting of the two special operators init and goal, and Ps initialised to {(g1,goal), (g2,goal), ... (gn,goal)}. Stripped of any particular search heuristics, a basic top level non-determinnistic algorithm is as follows:

```
procedure nlp(PPi, PPg);
  Store := { PPi };
  LOOP
    1 Remove a partial plan pp(Os,Ts,Ps,As,Es) from Store;
    2 Choose a (P,O) from Ps;
    3 Achieve P at O by adding a new operator instance to Os
      and/or further constraining Ts and/or Es;
    4 Add all new partial plans generated by step 3 to Store;
  UNTIL there exists partial plan PPg in Store whose Ps is empty.
end nlp.                                                        -4.1
```

More sophisticated search methods may be used instead of this algorithm, but we are not concerned with this level. Step 3 is the major concern of the rest of the paper: it contains the 'goal achievement operations', and is at the heart of any NLP.

5. THE PARTIAL PLAN INVARIANT

This section rigorously defines the partial plan data type, and therefore delimits the search space of the algorithm in section 4. Section 3 construted a model for the partial plan using 'abstract mathematical objects' in the usual VDM-type approach (see [5] for an introduction to this approach to program design). Because our target implementation will be in a logic programming language, specifically Prolog, we need hardly refine the data type as it is already implementable; the only exception to this is the use of lists to model sets, which is an adequate refinement.

We now define a data type invariant for the partial plan, which is true for any

initial partial plan, and will be preserved by the main partial plan operations. Roughly, a valid partial plan is one in which Os contains the two special operator instances, denoting the initial state and goal conditions, and all other elements of Os are partially bound operator instances from OS; Ts must specify a consistent partial order on Os, Ps and As must form the disjoint union of the operators' preconditions, every precondition predicate in Ps must be achieved and Es must be consistent with the task environment E. Hence we define the invariant condition as follows:

INV(pp(Os,Ts,Ps,As,Es)) = (a)&(b)&(c)&(d)&(e)&(f)&(g) where

(a) = for all O in Os-{init,goal}: (there exists O' in OS and variable bindings set s: O = (O')s)

(b) = for all O1,O2 in Os: not(before(O1,O2,Ts)) V not(before(O2,O1,Ts))

(c) = for all O in Os-{init,goal}: before(init,O,Ts) & before(O,goal,Ts)

(d) = for all P in O.p: member((P,O),Ps) V member((P,O),As)

(e) = Ps intersect As = { }

(f) = for all (P,O) in As: there exists A in Os: achieved(P,O,A))

(g) = there exists some binding set t: (E-> (Es)t) V Es = { } -5.1

Finally, we need to define the condition in 5.1(f), that of 'achieved(P,O,A)'. This is derived from the semantics of the strips-type operator schemas. Our definition is a simpler but slightly weaker version of Chapman's Modal Truth Criterion [Chapman 87]. Informally, P is achieved at O by some operator A in Os if A is necessarily before O, A.a contains P, and there is no operator C possibly between O and A such that C.d possibly contains P (if such a C does exist it is called a 'clobberer').

More formally, given a partial plan pp(Os,Ts,Ps,As,Es), a task specification (I,G,E,OS), and recalling the form of an operator O i.e. (O.n,O.e,O.p,O.a,O.d); then if (P,O) is in Ps, A in Os:

achieved(P,O,A) =
(there exists Q in A.a: P = Q) &
before(A,O,Ts) &
(for all C in Os: declobber(P,O,A,C))

where declobber(P,O,A,C) =
(C = O) V (C = A) V before(O,C,Ts) V before(C,A,Ts) V
(for all Q in C.d: not(unify(Q,P,Es)) -5.2

6. A CONSTRUCTIVE SPECIFICATION OF THE PARTIAL PLAN OPERATIONS

The invariant defined above now allows us to precisely specify the three main constructor operations within partial plan space, those that generate partial plans. In fact 6.2 and 6.3 below specify the generation of valid partial plans which contain one more achieved precondition than the input partial plan, whilest preserving the achievement of the rest; thus they makes progress towards a solution partial plan (one in which Ps={ }) in a goal directed fashion.

To produce an initial partial plan from a task specification:

INIT: { tasks } ---> { partial plans }
explicitly defined by INIT(I,g1&g2&..gn,E,OS) =
pp({ init, goal },{ },{ (g1,goal), .. (gn,goal) },{ },{ }). -6.1

The reader is left to check that INV(INIT(I,g1&g2&..gn,E,OS)) is trivially satisfied.

To produce partial plans that achieve a goal by constraining existing operators:

ACHIEVE1: {predicates} x {operator instances} x {partial plans} --->
sets of {partial plans}
ACHIEVE1(P,O,PP) = PP'
where PP = pp(Os,Ts,Ps,As,Es)), PP' = pp(Os',Ts',Ps',As',Es') and
pre- ACHIEVE1: member((P,O),Ps)
post-ACHIEVE1: there exists binding set t such that:
Os' = [Os]t &
Ts' contains [Ts]t &
Es' contains [Es]t &
As' = [As + (P,O)]t &
Ps' = [Ps - (P,O)]t &
there exists A in Os: achieved([P]t,[O]t,[A]t) -6.2

By our remark above it is clear that 6.2 makes progress towards termination; we must now investigate the validity of INV(PP'), given INV(PP) and the pre- and post conditions. 5.1(a) and (e) are trivially satisfied; 5.1(f) is satisfied since (from 5.2) for any (P,O) in As, addition of more legal variable or temporal bindings to a partial plan can never make the condition achieved(P,O,A) false. This is because the two main predicate before(X,Y,Ts) and not(unify(P,Q,Es)) are of 'necessary' modality (as defined in section 3). Finally we note that the conditions 5.1(b),(c) and (g) on Ts' and Es' are not necessarily true by our specification, and so should be considered as extra post-conditions on 6.2 (in fact they amount to integrity constraints).

To produce partial plans that achieve a goal by adding a new operator:

ACHIEVE2: {predicates} x {operator instances} x {partial plans} --->
sets of {partial plans}
ACHIEVE2(P,O,PP) = PP'
where PP = pp(Os,Ts,Ps,As,Es)), PP' = pp(Os',Ts',Ps',As',Es') and
pre- ACHIEVE2: member((P,O),Ps)
post-ACHIEVE2: there exists binding set t, and A in OS such that:
Os' = [Os + A]t &
Ts' contains [Ts]t &
Es' contains [Es]t &
As' = [As + (P,O)]t &
Ps' = [Ps - (P,O) + {(P,A) : P is in A.p}]t &
achieved([P]t,[O]t,[A]t) &
for all (P',O') in [As]t: declobber_As([A]t,(P',O')) -6.3

Similar arguments about the validity of INV(PP') hold as for 6.2, except that the addition of a new operator from OS may invalidate 5.1(f).The condition declobber_As(A,(P,O)) is there to check this: that each achieved P at O is still achieved after A is added to Os. This stipulates that O is necessarily before A, or if not, for any predicate in A.d that may unify with P, some achieving operator W

exists which is in between A and O and which adds P. This can be formalised:

```
declobber_As(A,(P,O)) <-
                before(O,A,Ts) V (A=O) V
                for all Q in A.d:
                        not(unify(P,Q,Es)) V
                        there exists W in Os:
                                before(A,W,Ts) &
                                before(W,O,Ts) &
                                there exists R in W.a and binding set t:
                                        (P = Q)t -> (R = P)t                -6.4
```

The expression '(P = Q)t -> (R = P)t' means that if P can be made to unify with Q under bindings t, then R unifies with P under this same substitution.

7. A DESIGN FOR ACHIEVE1

To translate the specifications to a sequential top level procedural design, essentially we let each predicate act as the post-condition to individual procedures, taking care that no previous post-condition is undone by the action of a following procedure; the declarative semantics of a logic program will thus preserve the logic of the specification. These procedures use input and output partial plan variables (PP, PP1 etc in 7.1) which allow conditions that do not hold in the input partial plan to be constrained to hold in the output partial plan.

This is similar to changing a context free grammar specification into a language parser (see chapter 9 in [2] for example). The correctness of the following design then depends on the correctness of the individual procedures: i.e. each line in the specification acts as a post-condition for its corresponding procedure. Also each procedure acting on components Ts and Es must be subject to the integrity constraints (see remark below 6.2). As a convention, we treat output variables in procedures as the right hand parameters; the top level design for 6.2 is then:

```
achieve1(P,O,PP,   PP3) :-
        get_el_Os(PP,   A),                /* there exists A in Os: */
        achieved(P,O,A,PP,   PP1),         /* achieved(P,O,A) &
                                                Os' = Os &
                                                Ts' contains Ts &
                                                Es' contains Es & */
        add_el_As(P,O,PP1,   PP2),         /* As' = As + (P,O) &   */
        del_el_Ps(P,O,PP2,   PP3).         /* Ps' = Ps - (P,O)     */     -7.1
```

Note that the primes on output values now refer to those values in the output of their corresponding procedure, and are not necessarily the same as in 6.2. Also, we leave out the extra detail of the substitutions, since components of the output partial plan are subject to a unique set of bindings, as they are being modelled by elements of a logical term.

To transform expressions of the form 'there exists X in Set ..' we employ a simple retrieve operation on the Set and allow the logic program bactracking mechanism to find the correct instance. The retrieves that we use have obvious meaning, for example:

```
get_el_Os(PP,   A)        : get A, an element of Os from PP
get_el_add(A,PP,   Q)     : get Q, an element of A.a from Os in PP
```

For an expression of the form 'for all X in S: Condition' we define a higher order predicate 'for_all_els(Set, Condition)' to mean that Condition is true for each element in the Set. Then 'for_all_els' can be operationalised by the addition of input and output parameters as introduced above. The Condition will then be supplied with three extra parameters: a set member, and an input and output partial plan. Following 5.2, the design for 'achieved' is then (we shall leave out the primes on post-conditions where they are not necessary):

```
achieved(P,O,A,PP,  PP3) :-
        get_el_add(A,PP,  Q),                /* there exists Q in A.a: */
        unify(P,Q,PP,  PP1),                 /* P = Q &                */
        before(A,O,PP1,  PP2),               /* before(A,O,Ts) &       */
        get_Os(PP2,  Os),                    /* for all C in Os:       */
        for_all_els(Os, declobber(P,A,O),
                 PP2,  PP3).                 /* declobber(P,A,O,C)   */   -7.2
```

In operationalising the specifications, we have chosen a least committment method: this means that the declobber(P,A,O,C) predicate should be in two sections, one to check if any of its conditions are satisfied (in which case the predicate is satisfied without need to constrain the partial plan - hence the use of the cut '!'), and the other to constrain the partial plan to satisfy the predicate if necessary. Any of these legal constraints are allowed and can be obtained through backtracking:

```
declobber(_,_,O,O,PP, PP) :- !.              /* C = O V                */
declobber(_,A,_,A,PP, PP) :- !.              /* C = A V                */
declobber(_,_,O,C,PP, PP) :-
        get_Ts(PP,  Ts),
        before(O,C,Ts),!.                    /* before(O,C,Ts) V       */
declobber(_,A,_,C,PP, PP) :-
        get_Ts(PP,  Ts),
        before(C,A,Ts),!.                    /* before(C,A,Ts) V       */
declobber(P,_,_,C,PP, PP) :-
        get_Es(PP,  Es),
        not( get_el_del(C,PP,  Q),           /* not(there exists Q in */
        unify(P,Q,Es) ),!.                   /* C.d: unify(Q,P,Es))   */
declobber(_,O,_,C,PP, PP0) :-
        before(O,C,PP,  PP0).                /* make before(O,C,Ts) V */
declobber(_,_,A,C,PP, PP0) :-
        before(C,A,PP,  PP0).                /* make before(C,A,Ts) V */
declobber(P,_,_,C,PP, PP0) :-
        get_del(C,PP,  Cd),                  /* for all Q in C.d:     */
        for_all_els(Cd,
          constrain(P),
          PP,  PP0).                         /* make not(unify(P,Q,Es))*/
```

'constrain' adds any possible variable constraints (conforming to enviromnment E), when necessary, to block the unification of P and a predicate from C's delete set. Its specification is:

constrain: {predicates} x {predicates} x {partial plans} ---> {partial plans},
post- constrain(P,Q,pp(Os,Ts,As,Ps,Es), pp(Os',Ts',As',Ps',Es')) =
 not(unify(P,Q,Es')),

and the data type invariant also demands that Es' satisfy condition 5.1(g).

8. A DESIGN FOR ACHIEVE2

The transformation of 6.3 and 6.4 to a top level procedure proceeds as in section 7; conditions are changed to procedures with input and output partial plan variables, and the corresponding parts of the specification act as post-conditions for the output partial plan. The top level procedural design for ACHIEVE2 is then:

```
achieve2(P,O,PP,   PP5) :-
        insert_op(PP,   A,PP1),               /* there exists A in OS:
                                                 Os' = Os + A  &
                                                 Ts' contains Ts  &
                                                 Es' contains Es  &
                                                 Ps" = Ps +{(P,A):P is in A.p} */
        achieved(P,O,A,PP1,   PP2),           /* achieved(P,O,A) &      */
        get_As(PP2,  As),
        for_all_els(As,                       /* for all (P',O') in As:      */
                declobber_As(A),
                PP2,   PP3),                  /* declobber_As(A,(P',O')) */
        add_el_As(P,O,PP3,   PP4),            /* As' = As + (P,O)        */
        del_el_Ps(P,O,PP4,   PP5).            /* Ps' = Ps" - (P,O)       */  -8.1
```

We leave the reader to refine specification 6.4 in the same manner. Assuming that the user defined operators composing OS (from task specification (I,G,E,OS)) are available as facts of the form operator(N,E,P,A,D), as described in section 2, the refinement of insert_op is:

```
insert_op(PP,   A,PP3) :-
        operator(An,Ae,Ap,Aa,Ad),                 /* there exists A in OS:  */
        gensym(op,A),                             /* utility for generating names */
        add_op(op(A,An,Ap,Aa,Ad),PP,  PP1),       /* Os' = Os + A  &        */
        add_Ps(Ap,A,PP1,   PP2),                  /* Ps' = Ps + {(P,A): P in A.p} &*/
        add_Es(Ae,PP2,   PP3).                    /* Es' contains Es  &
                                                     Ts' contains Ts       */  -8.2
```

The final two post-conditions are satisfied since the partial plan 'add' operations increment the input partial plan (in fact Ts' = Ts). That there is a consistent binding over the whole of the output partial plan (not stated explicitly here but in specification 6.3) is due to the use of a logical term as plan representation. The efficiency of 8.2 is greatly increased if a check is made that A's add set contains a predicate that can possibly unify with P from 8.2.

We must check the invariance of 5.1: 'add_Ps' does not violate 5.1(d) or (e) since the precondition pairs are unique, and they are taken from the already added operator OP. 'add_Es' adds uninstantiated environmental preconditions to Es, so 5.1(g) will hold, as obviously there must exist some bindings that make an operator's preconditions consistent with E.

5.1(f) could be falsified, since the added operator may well clobber some members of As. This part of the invariant is re- achieved, however, by procedure 'declobber_As', as explained above.

9. THE PARTIAL PLAN ABSTRACT DATA TYPE

We have two tasks left: to implement the partial plan ADT, and the partial order and unification procedures used in sections 7. and 8., according to their proper specifications. Our choice of constuction for the partial plan (a five slot logical

term, each slot containing a set) is immediately representable in Prolog; all that is required, as stated in section 5, is to represent the sets by lists. The add and retrieve operations may simply be implemented by facts, e.g. get_Ts(PP, Ts) is implemented as:

```
get_Ts(pp(_,Ts,_,_,_),   Ts).
```

Conforming to section 3(i), operators in Os should be augmented with an identifier when added to the partial plan (this is accomplished by the 'gensym' utility in 8.2); this can be used to represent that operator instance in the representation of partial order Ts.

Implementation of partial order procedures

The specification of before(X,Y,Ts) is outlined in section 3. It is a predicate and does not change the 'current' partial plan, hence will not upset the data type invariant; in its complete form it is:

```
before(X,Y,Ts) <-
        not(X = Y) & not(Y = init) & not(X = goal) &
            [  { member(t(X,Y),Ts)  V
                 there exists operator Z:
                 (member(t(X,Z),Ts) & before(Z,Y,Ts))} ].
```

Using logical transformation, and an auxillary predicate before'(X,Y,Ts), we change this to Horn clauses thus:

```
before(X,Y,Ts) <-   not(X = Y) & not(Y = init) & not(X = goal) & before'(X,Y,Ts).
before'(X,Y,Ts) <-  member(t(X,Y),Ts).
before'(X,Y,Ts) <-  member(t(X,Z),Ts) & before(Z,Y,Ts).
```

and syntactic changes will render this executable. The four slot version of 'before' uses this predicate, and has a similar implementation.

Implementation of unification procedures

Both the unification predicate and procedure use Prolog's unification mechanism. Since this will at most further constrain variables, any unifications previously made cannot be undone, and so this will not violate any achieved conditions. Part (g) of 5.1, however, has to be preserved or else the unification must fail. When checking for unification or consistency we use a Prolog trick so as not to needlessly bind any variables in the partial plan: the predicate is called twice by the 'not' operator, preserving the logic, but losing the bindings in the predicate's satisfaction. We therefore state their implementations as follows:-

```
unify(P,Q,Es) :-   not(not( P = Q,  consistent(Es)   )).
unify(P,Q,PP,   PP) :-  P = Q,  get_Es(PP,   Es),  not(not( consistent(Es))).
```

and leave the implementation of predicate consistent(Es) to the reader, where

```
consistent(Es)  <-   (there exists bindings t : E -> (Es)t),
```

since this will depend on the implementational details of the environment E. For example, consider a unification which wrongly binds a typed variable: E may contain the correct typing of objects in the planner's application domain, say E

contains 'type_of(doorA,door)' and Es contains 'type_of(X,box)'. If X is a variable in P or Q which becomes bound to 'doorA' through their unification, then consistent(Es) and subsequently unify(P,Q,Es) will be forced to fail, since 'type_of(doorA,box)' is not satisfied by E.

10. REMARKS AND CONCLUSIONS

In effect we have shown, by example, a rigorous method for the development of computational models in planning systems, and demonstrated the use of logical transformation to construct an executable logic program. The choice of an implementable high level data stucture - the logical term, and the closeness of Prolog to the original logic specification, remove much effort in verifying the top level designs and the final implementation. The example application was also less troublesome because of the availability of a 'tight' specification. A full implementation can be found in [8].

This methodology gives the usual advantages with respect to implementation correctness. Extensibility is also improved: shifts in the specification and extensions to the partial plan data type can be easily made and verified.

The specification can be used to show clearly where choices occur in partial plan generation, and so where heuristics are needed; it may also be used to generate explanations as to *why* a solution was found. We are exploring both these avenues using the implemented NLP as the performance component within a machine learning system [7].

REFERENCES

[1] Chapman, D. "Planning for conjunctive goals", Artificial Intelligence 29, 333-377, July 1987.

[2] Clocksin, W. and Mellish C., "Programming in Prolog", Springer Verlag, 1984.

[3] Fikes, R. Hart, P. and Nilsson, N. "Learning and Executing Generalised Robot Plans", Artificial Intelligence 3, 251-288, 1972.

[4] Hogger, C. "Introduction to logic programming", Acedemic Press, 1984.

[5] Jones, C. "Systematic software development using VDM", Prentice Hall, 1986.

[6] McCluskey, T.L. "The anatomy of a weak learning method for use in goal directed search", Proceeding of Fourth International Workshop on Machine Learning, Morgan Kaufmann, June 1987.

[7] McCluskey, T.L. "Learning and Non-linear Planning" Computer Science Technical Report TCU/CS/88/7, City University, London 1988.

[9] McCluskey, T.L. "Deriving a correct logic program from the formal specification of a non-linear planner" Computer Science Technical Report TCU/CS/88/1, City University, London 1988.

[9] Sacerdoti, E.D. "The non-linear nature of plans" , Proceedings of the 4th I.J.C.A.I., 1975.

APPENDIX D.6

[McCluskey 88b]

" The FM Problem Solving Environment User Guide",
Computer Science Technical Report, TCU/CS/1988/30

THE FM PROBLEM SOLVING ENVIRONMENT

USER GUIDE

version 1.2

1. Introduction

2. The Problem Specification Language

3. FM's Implementation

4. Running FM

1. Introduction

The FM environment provides problem solving and learning mechanisms which can be harnessed to a user defined application. This tutorial document will explain how to use the problem solving capabilities, leaving the learning components to a later release. It shows the user:

- how to specify an application in FM;
- how to run FM applications.

The reader is assumed to have some familiarity with A.I. and Problem Solving. For a relevant introduction see for example: 'Introduction to Artificial Intelligence', E.Rich, McGraw-Hill, 1984. FM can run in 'NIP' or C-prolog (see section 3).

2. The Problem Specification Language.

FM's approach is related to the 'STRIPS' family of problem solvers. A particular task is defined by two components:

- an initial state
- a goal condition

The specification of any application must be written as two separate components. These are:

- an operator set
- an environment

We next describe these four components individually.

2.1 The Initial State

We define a state description as a conjunction of ground predicates. It models some notionally unique world in an application.

Its syntax is:

```
init_world(<conjunction of predicates>).
```

e.g. in a box world application this could be:

```
init_world(
onfloor(box4)&handempty&
onfloor(box3)&onfloor(box2)&
onfloor(box1)&ontop(box5,box1)&
clear(box2)&clear(box3)&
clear(box4)&clear(box5)
).
```

An initial state is simply a state description from which problem solving starts.

2.2 The Goal Condition

A goal condition is a conjunction of predicates which the problem solv tries to achieve. It does so by searching for a list of ground operato such that when they are applied sequentially to the initial state, they will produce a state which contains the goal condition.
A necessary condition for a goal condition to be achieved is that each of its components predicates are present in the add list of some operator.

2.3 The Operator Set

This specifies the actions that can be made by the 'agent' being modelled in the application. General constraints on the operators include the 'STRIPS' assumptions, i.e. each operator models an instantaneous action and includes every effect of it; furthermore it is assumed that the only changes occurring in the application are made by by the operators.
To alleviate the computational complexity of the problem solving algorithms, each component of the operator is restricted to predicate conjunctions rather than the full first order logic of the original STRIPS formulation.

Each operator has the following free format syntax:

```
 frame(
        name:   <name>(<parameter-list>),
        type:   operator,
        filter: <conjunction of predicates>,
        check:  <conjunction of predicates>,
        precon: <conjunction of predicates>,
        padd:   <conjunction of predicates>,
        add:    <conjunction of predicates>,
        delete: <conjunction of predicates>,
        ).
where:
 <conjunction of predicates> =
              nil |
              <predicate> |
              <predicate>&<conjunction of predicates>
```

Parameters must start with a capital letter and obey Prolog's syntax for variables. Predicate symbols must be likewise written according to Prolog's syntax. These predicates are all user defined except for two, 'ne(X,Y)', which is interpreted by FM as meaning 'X is not equals to Y', and 'type_of(X,Y)', which is defined in the environment (see below).

An example operator from a 'box world' application is:

```
 frame(
        name:   putonbox(Ob1,Ob2),
        type:   operator,
        filter: nil,
        check:  type_of(Ob1,box)&
                type_of(Ob2,box)&ne(Ob1,Ob2),
        precon: clear(Ob2)&
                holding(Ob1),
        padd:   ontop(Ob1,Ob2),
        add:    handempty&clear(Ob1),
        delete: clear(Ob2)&
                holding(Ob1),
        ).
```

This models the putting down of an object by some imagininary robot arm onto another box. (see directory 'boxes' ,file 'fops', in the FM implementation).

2.31 Application of FM Operators.

STRIPS-type operators have clear semantics; this allows them to be easily manipulated by learning components, and to by-pass the famous 'Frame Problem'. There are basically three components to the operator:

--1. Preconditions: This contains the predicates that must be true before application of the operator is allowed.

In FM these are factored out into:

'check' predicates: these are the unchanging facts/constraints that must be satisfied by the environment;
'filter' predicates: these are a subset of the precon predicates below. In a goal directed search, the operator will only be added to a current partial plan if the filter predicates are true in the current initial state.
'precon' predicates: these are the facts that must be satisfied by the 'current' state description.

--11. Delete-list: This contains the predicates that, when the operator is applied to a state, will be removed from the state.

-111. Add-list: This contains the predicates that, when an operator is applied to a state, and the delete list has been removed, are then added to make up the new state.

In FM these are factored out into:

'padd' predicates: These are 'primary' add predicates, that is the most important ones that the operator adds.

'add' predicates: These are the predicates that the user considers as side affects, which are not the chief purpose of the operator.

N.B. The preconditions of an operator may be satisfied by a state under more than one instantiation.

Example:

We can apply our example operator to the state:

```
holding(box4)&onfloor(box3)&onfloor(box2)&
onfloor(box1)&ontop(box5,box1)&
clear(box2)&clear(box3)&clear(box5)
```

in three different ways!

If we chose 'putonbox(box4,box5)' then the resulting state would be:

```
onfloor(box3)&onfloor(box2)&handempty&
onfloor(box1)&ontop(box5,box1)&ontop(box4,box5)&
clear(box2)&clear(box3)&clear(box4)
```

2.4 The Environment

The environment is a collection of unchanging facts or constraints about the application. Usually this may just contain variable typing information.
Its syntax is that of a Prolog term of the form:

```
frame(  name: <name>,
        type: context,
        always: <conjunction of predicates>,
        axioms: <list of rules>  ).
```

An example from the boxes world is:

```
frame(  name: blocks_world1,
        type: context,
        always:
                type_of(box2,box)&type_of(box5,box)
                &type_of(box4,box)&type_of(box1,box)
                &type_of(box3,box),
        axioms: []).
```

The axioms slot can be filled by a list of rules modelling the application. It is chiefly used by learning components, and so discussion will be detained until the next version of this document. The reader is advised to consult some of the sample applications in fm_user for more complex environments.

3. FM's Implementation

3.1 Getting started

FM can be found in a unix directory structure called 'fm_user'. It is available for copying to a users directory from 'csgould' pathname 'lee/fm_user'. fm_user is a collection of directories; two are called 'lp' and 'nlp' and contain Prolog source files. The other directories are sample applications.

Once you have copied the FM structure, you may create your own application by the following procedure:

a. Create a new directory for your application inside fm_user, and change directory to it.

b. Create your operator set, environment and initial state in three

sepate files, e.g.
operator set is in 'ops';
environment is in file 'env';
initial state is in 'init';
Don't forget to follow the syntax for Prolog terms,
for instance, make sure each operator in your operator set
is terminated by a full stop.

c. Copy two files 'options' and 'boot' from a sample application to your directory - e.g. using directory 'boxes' the commands would be:

```
cp ../boxes/options .
cp ../boxes/boot .
```

At this point your should have five files in your new directory.
Finally, you must adjust the options file to suit your application.

3.2 The Options File

This is a file of Prolog terms providing control over current choices available to the user. FM is primarily a experimental system, and the user is warned not to change any options apart from the first four, without seeking further help. The first four terms in the file are:

```
strategy(<search type>).
operator_file(<file_name>).
environment_file(<file_name>).
init_file(<file_name>).
```

<search type> may be either
'nlp' .. to choose the non-linear problem solver;
'mea' .. to choose the goal directed linear problem solver;
'forward' .. to choose the breadth first problem solver.

For example, for the file names in 3.1, you should change the first four lines of the options file to:

```
strategy(mea).
operator_file(ops).
environment_file(env).
init_file(init).
```

This chooses the mea problem solver, which at the moment is by far the most efficient!

4. Running FM

To run the FM system, you must first change directory to that of your desired application.

If using the NIP system, invoke prolog by typing 'nip', then type
        ['../lp/bootnip'].

This will load in the prolog files, and will take about two minutes.
If you are using C-prolog, type the 'prolog' command to invoke prolog then type:
        [boot].

To start your application, simply type 'b.' followed by a return; you will be asked to enter a task (a conjunction of predicates followed by a full stop). After FM has solved your task (be patient!), you may enter another by typing 'b.' again.
Note that the next task will start from the advanced state containing your last goal condition.

Here is an example trace, using mea search, and the boxes example.

```
Script started on Thu Feb 25 16:33:09 1988
cssun5% nip

Edinburgh Prolog, version 1.5      (1st June 1987)
AI Applications Institute, University of Edinburgh

| ?- [boot].

options consulted:   2008 bytes      0.50 seconds

(*** junk .......... ***)

../lp/boot consulted:   88588 bytes      137.75 seconds

boot consulted:   88608 bytes      137.77 seconds

yes
| ?- b.

This is the non-linear planner..
My environment is called boxes_1
My current world is
onfloor(box4)&handempty&onfloor(box3)&onfloor(box2)&
onfloor(box1)&ontop(box5,box1)&clear(box2)&clear(box3)
&clear(box4)&clear(box5)

Enter task or "h" for help>ontop(box2,box5).

ontop(box2,box5)
 1****expanding init size 0
```

```
...
planning completed
By sequence of operators

[pickofffloor(box2),putonbox(box2,box5)]

goal ontop(box2,box5) is satisfied, new state is

handempty&clear(box2)&ontop(box2,box5)&onfloor(box4)&
onfloor(box3)&onfloor(box1)&ontop(box5,box1)&clear(box3)
&clear(box4)
10 plans generated

yes
| ?- b.

This is the non-linear planner..
My environment is called boxes_1
My current world is
handempty&clear(box2)&ontop(box2,box5)&onfloor(box4)&
onfloor(box3)&onfloor(box1)&ontop(box5,box1)&clear(box3)
&clear(box4)

Enter task or "h" for help>holding(box5)&ontop(box2,box3).

holding(box5)&ontop(box2,box3)
..
planning completed
By sequence of operators

[pickoffbox(box2,box5),putonbox(box2,box3),pickoffbox(box5,box1)]

goal holding(box5)&ontop(box2,box3) is satisfied, new state is

clear(box1)&holding(box5)&clear(box2)&ontop(box2,box3)
&onfloor(box4)&onfloor(box3)&onfloor(box1)&clear(box4)
19 plans generated

yes
| ?-
Prolog terminated
cssun5%
script done on Thu Feb 25 16:40:37 1988
```