WARPLAN:
A SYSTEM FOR GENERATING PLANS.

by

David H. D. Warren

Memo No 76

DEPARTMENT OF
COMPUTATIONAL LOGIC

UNIVERSITY OF EDINBURGH

SCHOOL OF
ARTIFICIAL INTELLIGENCE

WARPLAN:

A SYSTEM FOR GENERATING PLANS.

by

David H. D. Warren

Memo No 76

June 1974.

CONTENTS

	Page
ABSTRACT	1
PREFACE	2
1. INTRODUCTION	3
2. THE 3 BLOCKS PROBLEM	4
3. SPECIFYING A PROBLEM	6
4. A 5 BLOCKS PROBLEM AND ITS SOLUTION	8
5. IMPLEMENTATION OF THE SYSTEM	10
6. DEFICIENCIES OF THE SYSTEM	13
7. COMPLETENESS AND IRREDUNDANCY	16
8. CONCLUSIONS.	23
APPENDIX I. - LISTING OF THE PROGRAM	28
APPENDIX II. - TEST PROBLEMS	31
APPENDIX III. - SUMMARY OF PROLOG	41
ACKNOWLEDGEMENTS	44
REFERENCES.	45

ABSTRACT

The system is intended to be a general purpose plan generator for domains described in a formalism close to that of STRIPS. The method used is in many ways similar to STRIPS, but the search space is complete, unlike STRIPS and several similar systems. However the present purely depth-first search strategy is obviously incomplete although it produces good solutions to many problems.

The system is implemented in PROLOG, an elegant programming language essentially identical in syntax and semantics to Predicate Calculus in clausal form. The entire WARPLAN program comprises 46 clauses and aims at conciseness and clarity rather than efficiency. Nevertheless the present implementation solves some "standard" problems roughly 8 times faster than STRIPS, or roughly 5 times slower than LAWALY (a system designed primarily to be efficient).

WARPLAN is perhaps interesting as a system implemented in First Order Logic which solves problems in First Order Logic.

PREFACE

This memo is an interim report on a program I implemented and tested during the last two weeks of a visit to the University of Marseille. I am obviously hoping to develop the program further, but as a number of people have expressed interest in the work, it seems worthwhile to present the basic ideas now.

The most important part of this memo is Appendix I.

1. INTRODUCTION

Many problem domains can naturally be formalised as a world with a set of actions which transform that world from one state to another. A particular problem is then specified by describing an initial state and a desired goal state. The problem solver is required to generate a plan, a simple sequence of actions which transforms the world from the initial state to the goal state (strictly speaking, from any state satisfying the initial description to some state satisfying the goal description).

Such problem domains include, but are not restricted to, applications to robot planning. More generally the problem can be regarded as one of compiling a high level goal description into a low level program, albeit of an extremely simple structure, given a formal description of the target language (ie. target machine, ie. "world").

Writing special purpose plan generators (for a particular world or target machine) is at best tedious, and at worst near impossible, if the specification of the world is liable to change. A number of systems (including STRIPS,[3] LAWALY,[14] HACKER[16]) have been implemented which attempt in varying degrees to be general, ie. applicable to any domain.

A common failing of present systems is that they require that a conjunction of goals can be solved under the linear assumption. That is, goals 'X_1 & X_2' can be solved starting from some state 'start' by a plan of the form 'start; T_1; T_2' where T_1 is the action sequence of a minimal* plan to achieve X_1 from 'start' and T_2 is the sequence of a minimal plan to achieve X_2 from the state resulting from the plan 'start; T_1'. If 'X_1 & X_2' is unsolvable in that order, the goals are typically permuted to 'X_2 & X_1' and another attempt is made. There is no facility to interleave subplans. This frequently means that an optimal solution is not even theoretically attainable - it is not in the search space. At worst, there is no solution in the search space even though there is a solution in the intended interpretation (see the 3 Blocks Problem below). We shall call such systems linear.

WARPLAN can be regarded as a simple extension to STRIPS sufficient to attain completeness. The extension obviates the need to permute goals in a conjunction. The extension is also irredundant, that is the identical plan cannot be generated in two different ways. However there is still the underlying redundancy that re-ordering independent actions produces a distinct plan.

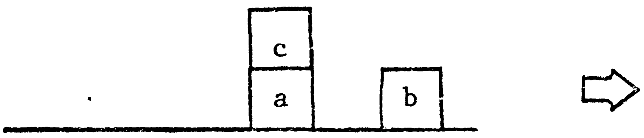
---

* For a definition of minimal (not to be confused with optimal) see Section 7.1.

2. THE 3 BLOCKS PROBLEM (of Austin Tate)

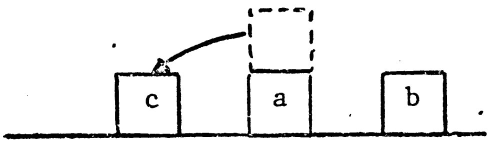
This is possibly the simplest example of a problem which is not solvable (optimally) by a linear planning systems. It was first noted as such by Austin Tate, although it appears in Sussman's thesis{16} as an "anomalous situation".

Given an initial state of 3 blocks as shown below, we want to achieve a final state in which 'a' is on 'b' and 'b' is on 'c' :-

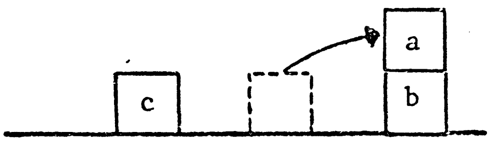


on (a,b) & on (b,c)

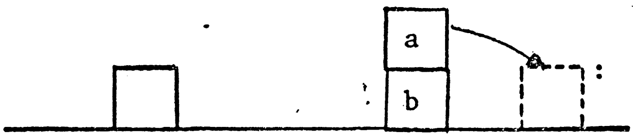
Typically, a STRIPS-like system would go through the following steps:-



:

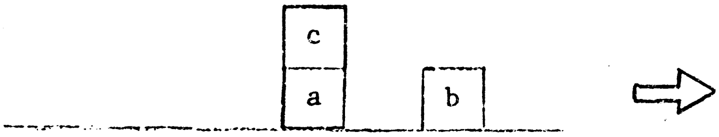


: FIRST GOAL ACHIEVED

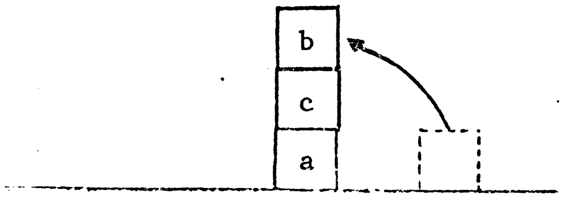


: PROTECTION VIOLATION

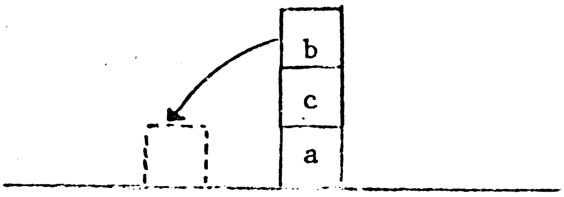
The system is now trying to clear the top of 'b' so that it can put 'b' on 'c'. But this would destroy an already achieved subgoal: 'on(a,b)'. Typically, the system tries again with permuted subgoals:-



on(b,c) & on(a,b)



: FIRST GOAL ACHIEVED



: PROTECTION VIOLATION

The system is trying to clear the top of 'a' so that it can put 'a' on 'b', and to do this it first needs to remove 'b' - another protection violation, this time of 'on(b,c)'.

The system is now in a dilemma since all paths in its search space lead to a protection violation. It either concludes, falsely, that the problem is unsolvable, or else permits a protection violation which (for this problem) allows a non-optimal solution to be obtained.

The optimal solution is of course:-

<u>move</u> c <u>from</u> a <u>to</u> floor;
<u>move</u> b <u>from</u> floor <u>to</u> c;
<u>move</u> a <u>from</u> floor <u>to</u> b

The reason linear planning systems get into difficulties is that the solutions to the two main goals are interleaved in the optimal plan. ie. the first and third steps achieve 'on(a,b)' and the second achieves 'on(b,c)'.

3. SPECIFYING A PROBLEM

Note: Refer to Appendix III for details of PROLOG syntax.

All data items manipulated by WARPLAN are represented as PC (Predicate Calculus, First Order Logic) terms. The main data types are a conjunction of facts (or goals) and a plan. A conjunction is constructed from certain primitive data items called facts using the binary function '&'. Similarly a plan is constructed from primitive data items called initial states and actions using the binary function ';'. '&' and ';' are declared as infix operators so that:-

X & Y & Z = X & (Y & Z) = &(X, & (Y,Z))

T ; U ; V = (T ; U) ; V = ; (; (T,U) , V)

PROLOG treats the above identities as different external representations of the same internal object.

Interpret: 'X & Y' as 'X and Y'
and 'T ; U' as 'the state after doing U in T'.

Note that, for brevity of exposition, we will identify a conjunction of a single fact with that fact and will frequently not distinguish between a plan and the state resulting from that plan.

A problem domain is specified to WARPLAN as a set of clauses, the problem database. This contains essentially the same information as STRIPS add-lists, delete-lists, preconditions and initial world wff. Certain information in the problem database may be represented procedurally, ie. as non-unit clauses. Usually, however, the clauses will be unit assertions. The following predicates are used:-

+ add(X,U) : fact X is added by action U; ie. X is true in any state resulting from U (and U is a possible action in some state in which X is not true).

+ del(X,U) : fact X is "deleted" by action U; ie. it is not the case that X is preserved by U. (X is preserved by U if and only if X is not added by U and X is true in a state resulting from U whenever X is true in the preceding state.)

+ can(U,C) : the conjunction of facts C is the <u>preconditions</u> of action U; ie. U is possible in any state in which C is true.

+ always(X) : fact X is true in any state.

+ imposs(C) : the conjunction of facts C is impossible in any state.

+ given(T,X) : fact X is true in the initial state T (but it is not the case that X is true in all states).

Note the following points:-

(1) Only actions which are sufficiently "primitive" can be formalised using the above predicates.

(2) In particular, any variable in an action's preconditions must appear as a parameter of that action. This is so that each action has a unique preconditions.

(3) The intended interpretation of 'del' is rather broader than that of STRIPS' delete-lists. <u>Any</u> fact may be regarded as deleted by an action so long as it is not positively preserved by that action. For efficiency, it is desirable to make 'del' assertions as powerful as possible. For example, we might assert 'del(on(robot,Z), climboff(B))' if we knew that no fact of the form 'on(robot,Z)' could remain true after the robot has climbed off a box B. It would not be necessary to instantiate Z to B, the particular box climbed off.

(4) Similarly, although it is not necessary to specify that certain conjunctions of facts are impossible, this can yield a huge improvement in performance for certain domains.

(5) Finally, notice that a fact cannot be both added and preserved by an action, so if a fact already holds before an action which adds the same fact, the previous instance should be regarded as deleted. This is important to avoid redundancy - see Section 7.

A particular problem is posed to WARPLAN by a procedure call of the following form:

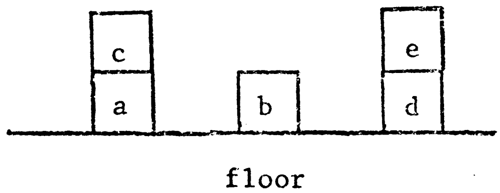
-plans(C,T) : output any plans which achieve the conjunction of facts C from initial state T.

4. A 5 BLOCKS PROBLEM AND ITS SOLUTION

As an illustration of the preceding points, we will give the database for a blocks world, and then outline how WARPLAN would solve the problem of stacking up 5 blocks. In obtaining the solution, WARPLAN also solves the 3 blocks problem of Austin Tate.

4.1 The Database

```
+ add ( on(U,W) ,     move(U,V,W) ).
+ add ( clear(V),     move(U,V,W) ).

+ del ( on (U,Z),     move(U,V,W) ).
+ del ( clear(W),     move(U,V,W) ).

+ can ( move(U,V,floor),  on(U,V)& V ≠ floor & clear (U) ).
+ can ( move(U,V,W)    ,  clear(W)& on (U,V)& U ≠ W &clear(U) ).

+ imposs ( on(X,Y) &clear(Y) )
+ imposs ( on(X,Y) &on(X,Z) &Y ≠ Z ).
+ imposs ( on(X,X)  ).

+ given (start, on(a,floor)).
+ given (start, on(b,floor)).
+ given (start, on(c,a) ).
+ given (start, on(d,floor)).
+ given (start, on(e,d) ).
+ given (start, clear(b)).
+ given (start, clear(c)).
+ given (start, clear(e)).
```



4.2 The Problem

```
- plans (on(a,b) & on(b,c) & on(c,d) & on(d,e), start).
```

The computation performed by WARPLAN in obtaining a solution can be interpreted as generating a sequence of approximations to the final solution. The first approximation is the initial state:-

(0) Start

We now attempt to solve the first goal 'on(a,b)'. It is not true in 'start', so we attempt to achieve it using the only operator available 'move(a,V,b)', abbreviated to 'm(a,V,b)':-

(1) start ; m(a,V,b)

(Changes to the plan are underlined.) We must now make sure that the preconditions of 'm(a,V,b)' are satisfied, inserting extra actions if necessary. The first precondition, 'clear(b)', is true in 'start', and the second, 'on(a,V)', is true if 'V := floor', giving the second approximation:-

(2) start ; m(a,floor,b)

'a ≠ b' is true (formal inequality of PC terms), so we have one remaining precondition to satisfy : 'clear(a)'. This is not already true in 'start', so we attempt to achieve it using the only available operator:-

(3) start ; m(U,a,W) ; m(a,floor,b)

All the preconditions are satisfied immediately if we choose suitable instantiations:-

(4) start ; m(c,a,floor) ; m(a,floor,b)

We have now achieved the first main goal and so far the steps have been identical to STRIPS. The second goal 'on(b,c)' is not true in the state produced by the plan so far, so we attempt to achieve it using the only available operator 'm(b,V_1,c)'. As STRIPS, we first try to introduce the new action at the end of the current plan. However, we notice that a precondition of 'm(b,V_1,c)' is 'clear(b)' and this is inconsistent with the already achieved goal 'on(a,b)'. Hence the action cannot be introduced at that point. We next try tracing back through the current plan trying to find a suitable point to insert the action, taking care that the goal we are achieving, 'on(b,c)', is not deleted by any already generated actions to the right. We find that a possible point to insert is immediately before the last action:-

(5) start ; m(c,a,floor) ; m(b,V_1,c) ; m(a,floor,b)

All the preconditions of 'm(b,V_1,c)' are satisfied at the point of insertion if 'V_1 := floor' :-

(6) start ; m(c,a,floor) ; m(b,floor,c) ; m(a,floor,b)

We have now solved the first two main goals, so we have a solution to the 3 Blocks Problem. The principal steps remaining to the final solution of the 5 Blocks Problem are listed below:-

(7) start ; m(c,a,floor) ; m(c,floor,d) ; m(b,floor,c) ; m(a,floor,b)

(8) start ; m(c,a,floor) ; m(e,d,floor) ; m(c,floor,d) ;
m(b,floor,c) ; m(a,floor,b)

(9) start ; m(c,a,floor) ; m(e,d,floor) ; m(d,floor,e) ;
m(c,floor,d) ; m(b,floor,c) ; m(a,floor,b)

The present implementation reaches this solution in a total of 52 seconds CPU time. The search strategy used at present is purely depth first, with conventional back tracking. The steps described above, assume that the order of main goals, operator preconditions, and '+can' assertions are as presented in Section 4.1. Notice that the above solution is obtained essentially without backtracking, and is not quite optimal. Further solutions, including the optimal one, could be obtained if different choices were made at the choice points. Notice that the original goals are achieved in the plan in the reverse order to that in which they are stated, but they are solved by the plan generator in the original order.

5. IMPLEMENTATION OF THE SYSTEM

Note: The reader is recommended to examine the examples in Appendix II before proceeding with this Section. Refer to Appendix I for a complete listing of the program and to Appendix III for details of PROLOG. It may well be easier to understand the program listing that this explanation!

The central predicate is 'plan (C,P,T,T_1)', the procedure entry point to the main recursive loop.

It has four arguments:-

- C is a conjunction of goals to be solved;
- T is an (already generated) partial plan;
- P is a conjunction of goals already solved by T which must be protected;
- T_1 is a new plan, which contains T as a subplan and preserves the already solved goals P, and which also solves the new goals C.

As interpreted by PROLOG, C,P,T behave as input variables and T_1 as an output variable. The clauses defining 'plan' are:-

+plan $(X\,\&\,C,P,T,T_2)$ -/- solve (X,P,T,P_1,T_1) - plan (C,P_1,T_1,T_2).
+plan (X,P,T,T_1) - solve (X,P,T,P_1,T_1).

Essentially this states that a 'plan' can be produced by 'solve'-ing each goal in the order given. The effect of the '-/' is to tell PROLOG not to consider the second clause if it has successfully "matched" the first literal of the first clause. In this case it could be omitted without affecting the semantics of the program; it is needed to prevent the substantial inefficiencies of trying to 'solve' a conjunction of goals, which is in fact impossible.

'solve(X,P,T,P_1,T_1)' is true if:-

- X is an atomic goal;
- T is a partial plan;
- P is a conjunction of goals achieved by T;
- T_1 is a plan, containing T as a subplan, which solves P_1 ;
- P_1 is a conjunction comprising P and X where X is not repeated.

X,P,T will be input variables and P_1,T_1 output.

There are three ways in which a goal may be 'solve'-d:-

+ solve (X,P,T,P,T) - always (X).
+ solve (X,P,T,P_1,T) - holds (X,T) - and(X,P,P_1).
+ solve $(X,P,T,X \& P,T_1)$ - add(X,U) - achieve (X,U,P,T,T_1).

It may be 'always' true in the world. It may be that it already 'holds' in the state produced by the current partial plan. Finally we may look in the database for an action U which 'add'-s the goal X and then 'solve' X by 'achieve'-ing U .

There are two methods to 'achieve' an action, which we will call <u>extension</u> and <u>insertion</u>. If we were to omit the clause for insertion, we would get a system almost identical to STRIPS without the ability to permute goals. The clause for extension is:-

+ achieve$(X,U,P,T,T_1;U)$
  - preserves(U,P)
  - can(U,C)
  - consistent(C,P)
  - plan(C,P,T,T_1)
  - preserves(U,P).

We first check that the action U preserves (ie. does not delete) the protected facts P. Then we lookup the preconditions C in the database and check that C is consistent with the protected facts. All being well, we call 'plan' recursively to modify the current plan T to a new plan T_1 which produces a state in which C is attained as well as P. U can then be applied in T_1, corresponding to the plan resulting from this call of 'achieve'. Finally, we repeat the check that U preserves P. The reason for this is that U and P may not have been instantiated to ground terms at the time of the original check.

Lacking the ability to co-routine in PROLOG at present, we have to be satisfied with an incomplete first check followed by a second check "to make sure". Even so there is still a slight flaw in the program, as U and P may still not be fully instantiated by the time of the second check.

The clause for the second method of 'achieve'-ing an action, insertion, is:-

```
+ achieve(X,U,P,T;V,T1;V)
    - preserved(X,V)
    - retrace(P,V,P1)
    - achieve(X,U,P1,T,T1)
    - preserved(X,V).
```

If the last action V in the current partial plan doesn't delete the current goal X, we can try to insert the action U somewhere before V, provided we 'retrace' the set of protected facts to the point before V. P is 'retrace'-d to P_1 before V if

$$P_1 = P - (\text{addset of } V) + (\text{preconditions of } V).$$

As mentioned previously, in WARPLAN, plans and states of the world are virtually synonymous. Everything that 'holds' in a state of the world can be determined from the plan which produces that state of the world. The system chains backwards through the sequence of actions, so long as none of these actions deletes the sought-for fact, until the fact is found in the 'add'-set of an action or was 'given' in the initial state:-

```
+ holds (X,T;V) - add (X,V).
+ holds (X,T;V) - /
    - preserved(X,V)
    - holds (X,T)
    - preserved(X,V).
+ holds (X,T) - given (T,X).
```

This method avoids the overhead of generating a net set of facts for each state of the world considered, as do STRIPS, PLANNER {6}, etc. albeit in a structure-shared form. However, to balance against this, there is more computation involved in accessing a fact (see Section 6 for possible improvements).

To prove that a fact X is preserved by an action V, WARPLAN essentially tries to satisfy itself that it can't prove that V deletes X :-

```
+ preserved(X,V) - mkground (X & V,0,N) - del (X,V)-/- fail.
+ preserved(X,V) .
```

'mkground' substitutes "arbitrary constants" for any variables in the terms currently bound to X and V. If we can now prove that X is 'del'-eted by V, we call '/' to prevent any further choices being taken for 'preserved' and then call 'fail'. This is an arbitrary predicate which can't be proved true, since we supply no definition for it. The net effect is that the original attempt to prove 'preserved(X,V)' fails, and subsequent backtracking of course "undoes" the effects of 'mkground'. In the other case we can't prove that X is 'del'-eted by V and the second clause allows the proof of 'preserved(X,V)' to succeed. Unlike previous uses of '/', this use actually changes the meaning of the two clauses. As a "hack", the technique is rather powerful and not without a certain appeal.

The remainder of the WARPLAN axioms define some fairly straightforward auxiliary procedures.

6. DEFICIENCIES OF THE SYSTEM

Some deficiencies of the system are listed below. They range from relatively minor details to problems which suggest that a totally new approach is needed.

(1) The 'holds' axioms could be made more efficient by using knowledge of action preconditions to avoid always chaining back to the point at which a fact was added, as is done for 'inroom(robot,room(1))' in the fourth STRIPS problem (Appendix II, 1.3) for example. If a fact occurs as a precondition of an action in a plan, we know that the fact must hold immediately prior to that action in the plan. So it is only necessary to chain back as far as the last time the fact was "used". Thus every time a fact were "used" it would become more "accessible" for further use. However it is difficult to do this without introducing redundancy.

(2) There is a similar possible efficiency improvement associated with the consistency checks. When an inconsistency has been found, the system should immediately retrace as far as is necessary to

remove one of the already-achieved goals which "cause" the inconsistency.

(3) In the current PROLOG, there is only direct access to a group of clauses with the same leftmost predicate, and not to clauses within such a group. Thus WARPLAN is continually chaining through its entire list of 'add' clauses, for example, to find a suitable action to add a certain fact. More direct access could be achieved with the current PROLOG at the expense of some loss of clarity in the WARPLAN axioms. This should yield a substantial improvement in speed.

(4) The system needs a more intelligent search strategy. Most of the problems which it has solved involved little or no backtracking. Backtracking frequently results in crazy alternatives being tried next. This arises particularly because there is nothing to prevent WARPLAN from constructing a plan to achieve a fact which is already true in the current state. (This facility is needed, in some cases at least, for completeness.)

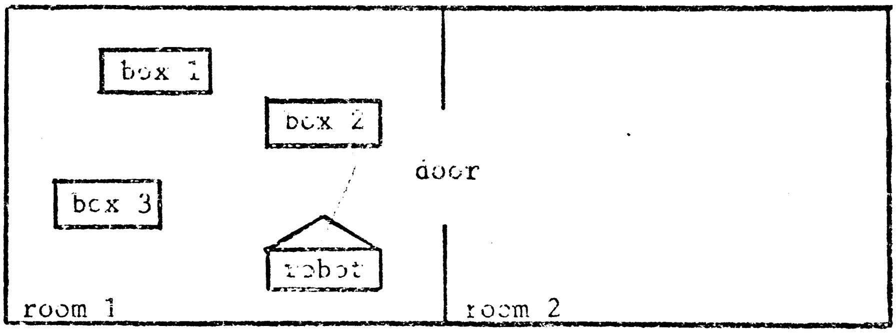
(5) An automatic check for loops could alleviate this and other problems, but would probably slow the system down substantially.

(6) At the moment, goals, action preconditions and 'add' clauses can be pre-ordered by hand to give the best results. It would be nice if the system could do the analysis necessary for this (cf. the way LAWALY determines its hierarchies). It would of course be better if the orderings were determined dynamically.

(7) Actions which add several facts frequently need "augmented" preconditions for some of them. (Two versions of 'take' were used in the Keys and Boxes Problem (Appendix II.2) to bypass this problem.) It should not be too difficult to provide this facility.

(8) There are a number of ways in which PROLOG might be enhanced to WARPLAN's benefit. The inability to put "restrictions" on variables results in some flaws in the program. This is a special case of the need for co-routining - more flexible choice of which literal to cancel next (ie. which goal to solve next).

(9) ABSTRIPS {13} may be regarded as a technique to enable STRIPS to perform co-routining as it generates a plan, and something similar is obviously needed in WARPLAN. For example, consider a world similar to Appendix II.1 :-



with the goal:-

inroom(robot,room 2) & nextto(robot,box3)

for which the <u>optimal</u> solution might be:-

start ;
goto(box 3) ;
shuntthru(box 3,door,room 1,room 2) ;
goto(box 3, room 2)

where 'shuntthru' does not leave the robot 'nextto' the shunted object. WARPLAN has to first produce a complete solution to one of the two top-level goals, and, for it to subsequently find the optimal solution, it is necessary that this partial solution be a subplan of the optimal plan . Suppose the top level goals are ordered as above. Then the initial subplan needed for 'inroom(robot, room 2)' is:-

start; goto(box 3); shuntthru(box 3, door, room 1, room 2)

a rather unlikely solution! (We are assuming there is a 'gothru' operator, and of course there are several boxes.) If, on the other hand, 'nextto(robot, box 3)' is ordered first, then the partial plan needed is the complete plan itself! The problem is that WARPLAN is generating too much detail in its solution to one subgoal before going on to consider a dependent subgoal.

(10) Like STRIPS, WARPLAN generates a plan by a mixture of backward (from the goal) and forward (from the initial state) analysis. For many problems, particularly "difficult" ones such as in (9) above or

block stacking or impossible tasks, it appears that a completely backward analysis solves the problem better. One starts with the given conjunction of goals and applies operators "in reverse" to generate a new conjunction of goals. Each new conjunction of goals is checked for consistency, and possibly for subsumption by other conjunctions of goals which have been generated. A solution is found when the conjunction of goals is satisfied in the initial state. The problem with this technique is that more and more variables get introduced into successive goals and these need to be restricted in complex ways; the initial state only gets "used" in the final step (although it would presumably direct the search strategy). The advantage of the SIRIPS-like analysis is that variables get quickly instantiated making the system much more amenable to implementation in the present PROLOG.

(11) WARPLAN is unable to generate conditional plans; nor is it therefore able to generate iterative plans.

(12) Plans generated by WARPLAN are totally ordered, often arbitrarily. Besides unnecessarily restricting the freedom of the plan executer, this also means that there is potential redundancy in the search space.

7. COMPLETENESS AND IRREDUNDANCY

7.1 Preliminary Definitions

For definitions of added, preserved, preconditions, see Section 3.

A plan of length N comprises an initial state T_0 and a sequence of actions $A_1, A_2, \ldots, A_N$. It will be written '$T_0;A_1;A_2;\ldots;A_N$'. We require that the plan be executable, ie. the state resulting from '$T_0;\ldots;A_{I-1}$' satisfies the preconditions of A_I, for I from 1 to N.

A problem is a pair $\langle C,T_0 \rangle$ comprising a conjunction C of facts which are main goals and an initial state T_0. A plan T solves the problem $\langle C,T_0 \rangle$ if T_0 is the initial state of T and C holds in the state resulting from T.

A plan is optimal for a problem if there is no plan of lower length which solves the problem.

A plan generator is complete if it will eventually generate an optimal plan for any problem.

A plan is minimal (for a problem) if every action is needed. An action in a plan is needed if it adds a fact which is a main goal or a precondition of a needed action.

A plan T is a subplan of a plan T_1 if

(i) T and T_2 have the same initial state

and (ii) the actions of T are a subset of the actions of T_1.

T is a proper subplan of T_1 is the actions of T are a proper subset of the actions of T_2.

7.2 Corollaries of the Definitions

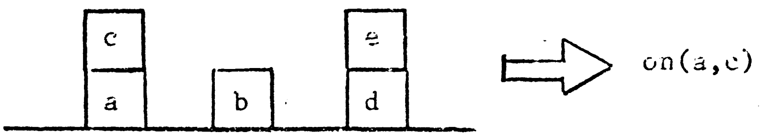
Any plan which solves a problem has a unique minimal subplan which solves the problem. (It is unique because we do not allow a fact to be both added and preserved by an action.)

An optimal plan must be minimal.

A plan generator which can generate all minimal plans is complete.

7.3 Example

Given the problem:-



a far from optimal plan which does in fact solve the problem is:-

start ; move(e,d,floor) ; move(d,floor,b) ; move(c,a,d) ; move(a,floor,c)

The corresponding minimal subplan is:-

start ; move(e,d,floor) ; move(c,a,d) ; move(a,floor,c)

['move(a,floor,c)' is needed since it achieves 'on(a,c)'.
'move(c,a,d)' is needed since it achieves 'clear(c)' which is a precondition of 'move(a,floor,c)' which is needed.
'move(e,d,floor)' is needed since it achieves 'clear(d)' which is a precondition of 'move(c,a,d)'.
The only action not needed is 'move(d,floor,b)'.]

7.4 Outline of a Completeness Proof

The completeness of WARPLAN follows from the fact that 'plan (C, true, T_o, T)' is a valid deduction from the WARPLAN axioms and problem database if and only if T is a minimal plan which solves the problem $\langle C, T_o \rangle$. To get a complete implementation of WARPLAN, one would need to

(1) provide a proper implementation of formal inequality ("restrictions" on variables) in PROLOG and modify the clauses for 'preserved' etc. to take advantage of this;

(2) make the PROLOG search strategy complete. For instance, every time a choice point is encountered (more than one input clause matches the current literal) set up the different choices as "parallel (independent) processes".

In outlining the proof, we shall only discuss the part played by the primary clauses, those labelled P1, P2, S1, S2, S3, A1, A2, H1, H2, H3 in Appendix I. We assume that the secondary clauses (the remaining WARPLAN clauses and the problem database) are a complete and correct formulation of their intended interpretations.

We wish to show that, for any problem $\langle C, T_o \rangle$, given a minimal plan T, there exists a derivation of 'plan (C, true, T_o, T)' from the WARPLAN axioms and problem database.

A derivation of a fact X from a set of Horn clauses S is a tree of instances of clauses from S, such that

(1) the top (or root) clause instance has $+X$ as its positive literal;

(2) for each negative literal $-L$ occurring in a clause instance, there is exactly one subtree at that node which is a derivation of L.

A clause is a Horn clause if it has at most one positive literal.

The proof proceeds by induction on the length of the minimal plan T. The proposition for proof by induction is:-

For any $C = G_1 \& G_2 \& \ldots \& G_M$, T_o, and minimal plan T of length N, there exist derivations of 'solve(G_I, P_{I-1}, T_{I-1}, P_I, T_I)' for I from 1 to M where P_0 = true,

$P_I = G_1 \& \ldots \& G_i$ [i from 1 to M] ,

T_I is the minimal subplan of T which solves P_I.

(From this proposition and clauses P1, P2 the final conclusion follows trivially.)

Case $N = 0$.

The proposition holds trivially using instances of clauses S1 and S2.

We must now assume the proposition holds for N and prove if for $N+1$. We shall merely describe the construction needed for this step. A rigorous proof would involve checking all the details.

Let $T = (T' ; U)$.

By minimality, U achieves at least one of the goals C.

Let G_I be the first such goal.

Let $K_1 \& \ldots \& K_J$ be the goals of C which appear after G_I in the conjunction and which are not achieved by U.

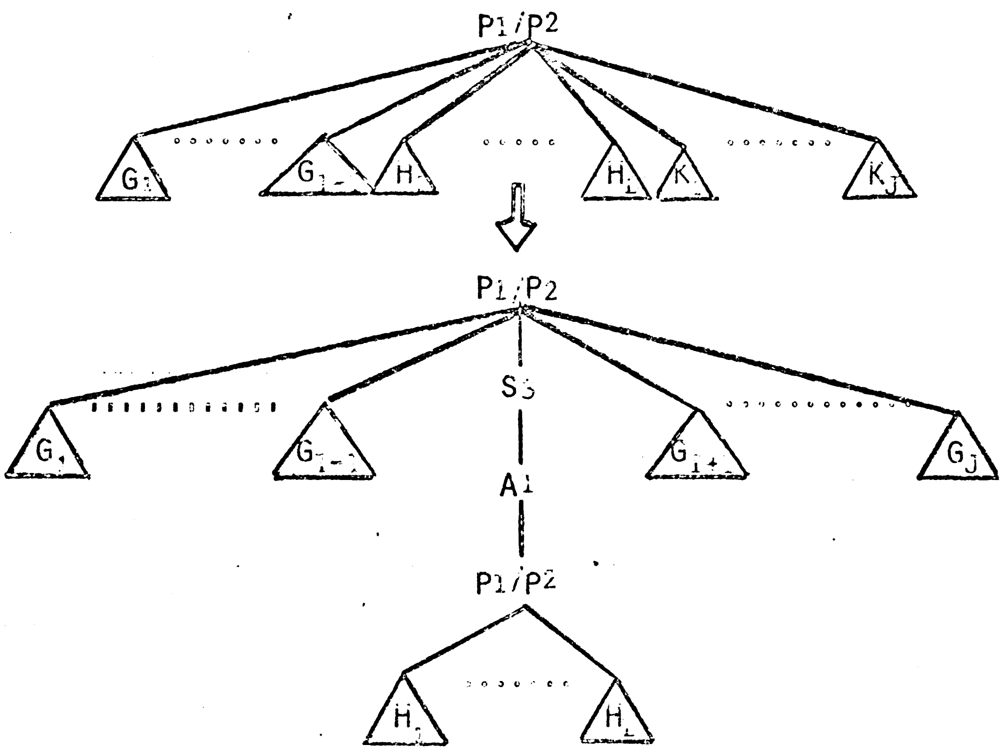
Let $H_1 \& H_2 \& \ldots \& H_L$ be the preconditions of U.

Let $C' = G_1 \& \ldots \& G_{I-1} \& H_1 \& \ldots \& H_L \& K_1 \& \ldots \& K_J$

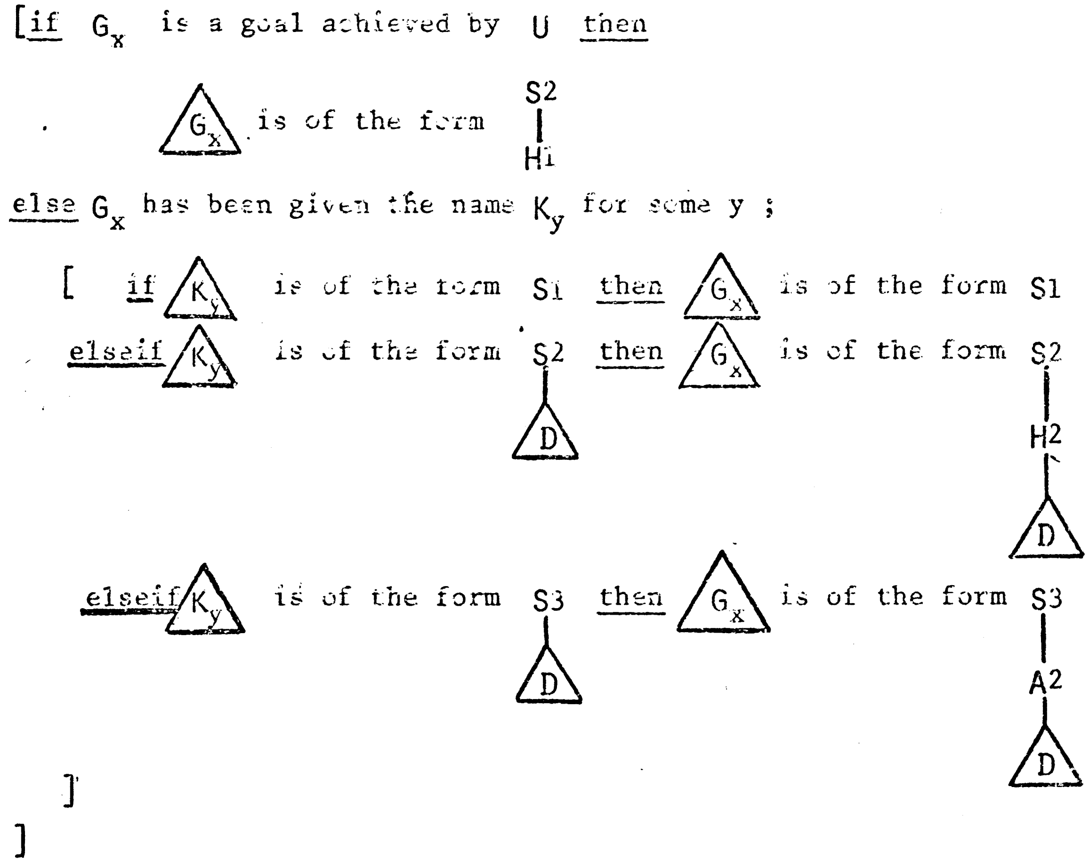
Then clearly T' is a minimal plan for the problem $< C', T_0 >$.

By the inductive hypothesis, there exists a derivation of 'plan (C', true, T_0, T')'. We shall show how to use this derivation to construct a derivation of 'plan (C, true, T_0, T)'.

The construction is illustrated below:-



A derivation is a tree of instances of WARPLAN clauses.
We ignore instances of secondary clauses.
It is clear that clauses P1 and P2 serve only to link together a number of derivations of 'solve' literals. We indicate such an occurrence of a set of P1 clauses with one P2 clause by 'P1/P2'.
A subtree is indicated by a triangle. Those in the diagram above represent derivations of 'solve(X,...)' where X is the label inside the triangle. Most of the subtrees in the "output" derivation are copied from the corresponding subtree in the "input" derivation. It remains to show how the subtrees for G_{I+1} to G_J are constructed:-



That completes the outline of the proof.

7.5 Irredundancy

A plan generator is *irredundant* if, in generating plans, it never generates the same plan more than once.

A set of (Horn) clauses S is *irredundant*, if, for any fact X, there exists at most one derivation of X from S.

The irredundancy of WARPLAN as a plan generator is equivalent to the irredundancy of a set of clauses comprising the WARPLAN axioms and an irredundant problem database. In fact WARPLAN is irredundant provided that

(1) 'always(X)' precludes 'add(X,U)' or 'given(T,X)' ;

(2) 'add(X,U)' and 'preserved(X,U)' are mutually exclusive ;

(3) facts of the term 'given(T,X)' are only supplied where T is the initial state.

Once again we shall only consider the primary clauses, assuming the secondary clauses are irredundant. We have to show that it is impossible for there to be two distinct derivations of 'plan(C,true,T_0,T)' where C,T_0,T are ground terms.

For most of the primary predicates, it is clear by inspection that, for ground instantiations of the first and last arguments, there is only one clause which can be used to derive that literal, given the assumptions listed above.

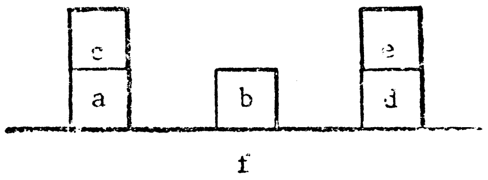
The only case in which there is difficulty is for the clauses S2 and S3. Suppose they can both be used to derive the same literal 'solve(X,P,T,P_1,T_1)', where X,P,T,P_1,T_1 are ground terms. From S2, $T = T_1$. But considering S3, it is clear that a derivation of the 'achieve' literal is only possible if the length of T_1 is at least one greater than T - a contradiction.

Therefore the irredundancy has been demonstrated.

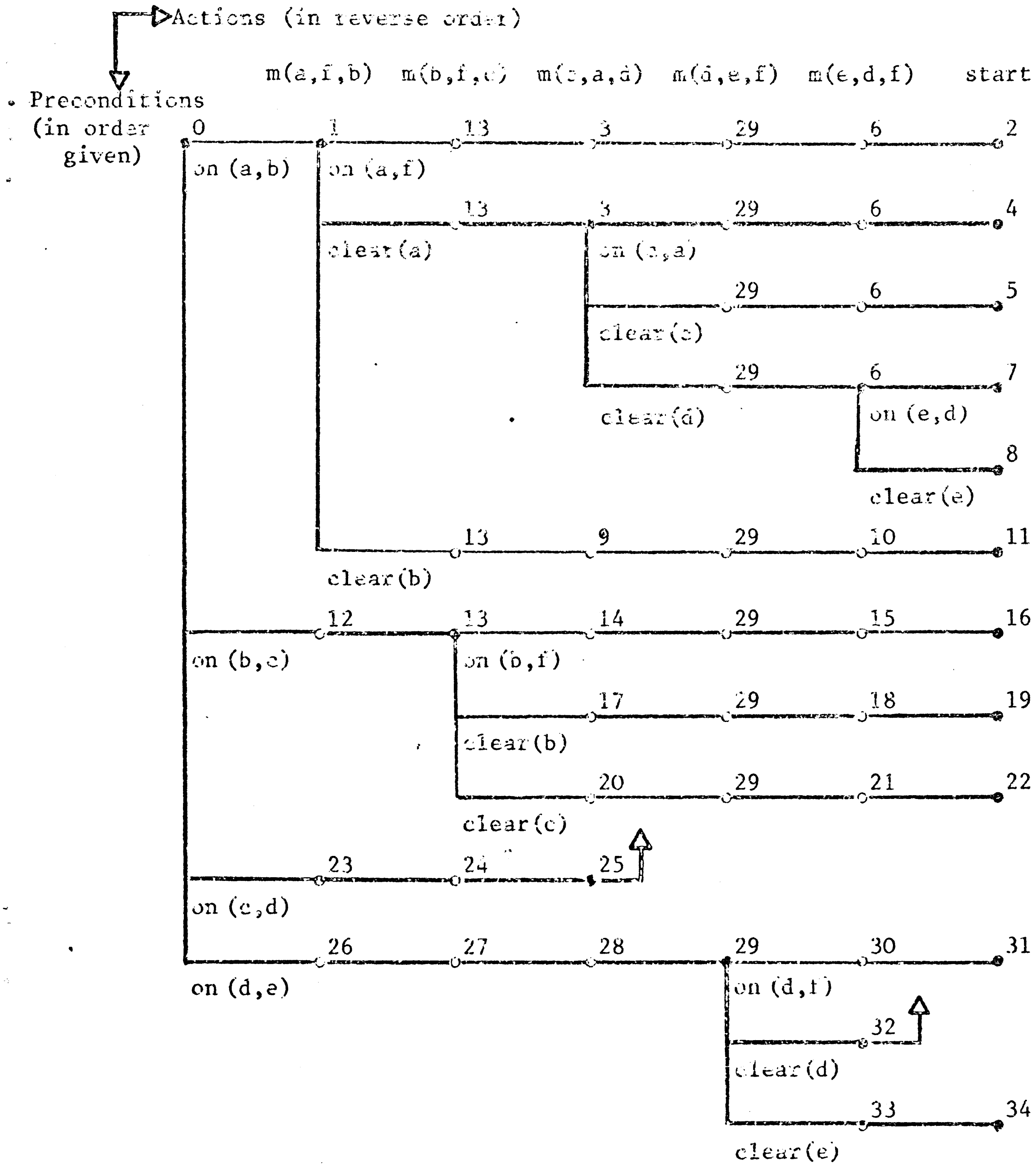
7.6 Another Example

To further elucidate the completeness of WARPLAN, the diagram below indicates how WARPLAN would arrive at the optimal solution to the 5 Blocks Problem (the solution described in Section 4 is the one which would be found first by a depth-first search strategy).

The diagram represents the <u>situation calculus</u> proof of the optimal plan which is <u>implicitly</u> generated by WARPLAN. A black dot represents an add/preconditions axiom whereas a white dot represents a frame axiom. The numbers labelling each dot indicate the order in which the proof is built up. The proof of the preconditions of an operator is only generated once; hence the proof of preconditions of a later use of the action is indicated by an upward arrow.



⇨ on(a,b) & on(b,c) & on(c,d) & on(d,e)



In order to find this proof, WARPLAN has to elect to 'clear(a)' by a 'move(c,a,d)' resulting in a need to 'clear(d)'. It later "discovers" that 'move(c,a,d)' happens to achieve the main goal 'on(c,d)'. Thus, although the optimal solution <u>is</u> in the search space, it is discovered by a rather unnatural process.

8. CONCLUSIONS

8.1 Predicate Calculus as Programming Language

WARPLAN is a program implemented in Predicate Calculus. I believe the use of this language made the program shorter, clearer and easier to implement, with little (if any) sacrifice of efficiency. I doubt whether a shorter program could be produced by rewriting the algorithm in another programming language.

More generally, I support Kowalski's {8},{10} contention that PC is a superior high-level programming language suitable for wide use, particularly in AI. Experience with PROLOG suggests that any program that can be formulated in LISP can be formulated at least as easily in PROLOG. Frequently the naturalness of PC can suggest a more elegant formulation.

So far this is only assuming an interpreter such as PROLOG with an unsophisticated (but fast) proof procedure, which gives PC programs a rather conventional control structure. Inference systems such as Kowalski's Connection Graphs {9} suggest how, in principle, control structures might be liberalised to great potential advantage, but much remains to be done to make this a practical reality.

Wherein lies the advantage of PC over other languages? One might attribute it to the features which PC doesn't provide. High-level languages have progressed by successively transferring responsibilities from the user to the software engineer. The user is relieved of certain machine-oriented tasks which would make his program clumsy to formulate and difficult to debug, leaving him free to concentrate on the problem-oriented aspects. One of the first such tasks to come under software control was the mapping of data into actual machine addresses. Later advances made the early machine-oriented concept of a goto redundant (though still widely used to the detriment of clear programming).

I believe a similar situation exists with regard to assignment (the ability to "re-use" a variable to denote more than one value during the course of a computation). Experience with ALGOL 68 convinced me that true assignment is rarely needed, and its avoidance results in programs which are easier to understand. There is of course no assignment in PC, and it's all the better for it!

Among the nice features of PC which might be mentioned are:-

(1) one language for program and data;

(2) the usual tree data-structures, but with a beautifully natural way to manipulate them via the generality of unification;

(3) no explicit distinction between input and output variables, so that a single predicate may function as several different procedures.

As an illustration of the elegance of PC programming, consider the definition of 'intersect' from Appendix I :-

$$+ \text{intersect}(S_1, S_2) - \text{elem}(X, S_1) - \text{elem}(X, S_2)$$

This simply makes precise the natural language statement that "two sets intersect if they have an element in common". The definition is conceived <u>before</u> consideration is given to its procedural aspects ie. how it will be used by the PROLOG interpreter. In fact, given full instantiations for S_1 and S_2, PROLOG will proceed to <u>generate</u> non-deterministically the elements of S_1 and then <u>check</u> to see whether they are elements of S_2 (ie. 'elem' is functioning as two distinct procedures). This example demonstrates the closeness of PC to natural language. The much-maligned clausal form seems quite natural when PC is given a procedural interpretation.

An interesting aspect of PC is the way one set of clauses really represents an equivalence class of different algorithms. For example, in Appendix I, if we interchange the two clauses for 'achieve', PROLOG gives a different "version" of WARPLAN in which actions are (first) inserted as far left as possible in the current plan, rather than first trying to tag the new action on to the end of a plan. Similarly, changing the order of literals in a clause can produce different but reasonable algorithms.

A factor hindering the progress of AI seems to be that AI programs are not produced in a form suitable for easy comprehension by humans. The essential ideas underlying the program become obscured when it is "coded". Consequently the programs themselves are rarely published, but are only described, leading inevitably to vagueness and ambiguity. A case in point is the literature on STRIPS, which by comparison with most papers is a model of clarity, and yet the number of hours I know I and others have spent in arguing about "what STRIPS really does" ...

Programs need to be published. For this one requires a language with the simplicity and universality of PC. The aim of programming should be towards an ideal of beauty and clarity. A program should not have to be "explained" by flowcharts or interspersed comments.

8.2 Logic in Logic

The WARPLAN program uses PC terms to represent conjunctions of facts (or goals). Thus the objects of discourse include sentences of logic. In addition, there is a one-to-one correspondence between plans and the proofs that these plans achieve the desired goals*. (The "proof" of a plan is, as in Section 7.6, a tree constructed from instances of 'add' axioms, 'can' axioms and implicit frame axioms derived from the 'del' axioms.) Thus the terms representing plans can equally well be said to represent proofs of these plans; the objects of discourse include proofs of sentences of logic.

Accordingly, WARPLAN may be described as a theorem-prover implemented in a theorem-prover. (cf. for example, an ALGOL compiler written in ALGOL.) I believe this technique may have considerable potential and answers to some extent the criticism that Higher Order Logic is needed to express certain problems.

8.3 The Utility of Inconsistency Tests in Planning

WARPLAN can make effective use of negative statements about the world. Typically these statements of impossibility reflect the "physics" of the world whereas the actions represent "engineering" in accordance with this physics. In particular the "delete list" of an action ought to be computable from a knowledge of the physics of the world, rather than being spelled out explicitly. (cf. the 'del' facts for the Machine Code Generation problem in Appendix II.3.2.)

The positive and negative aspects of problem statement and problem solving merit further investigation. The system DISPROVER of Siklossy and Roach {15} tries to show that a goal is an engineering impossibility whereas WARPLAN can only look for physical impossibilities. (A man on the Moon was an engineering impossibility last century but not a physical

---

*This requirement is the root reason why certain actions need parameters which appear "unnecessary" from the STRIPS standpoint. (See Section 3) impossibility, like perpetual motion; it only needed a few more operators to be added to the world!)

8.4 The Frame Problem

The frame problem {12} {5} has frequently been cited as an argument against the use of "uniform proof procedures". Since it is hard to imagine a proof procedure more "uniform" than PROLOG, it is interesting to consider whether or not the frame problem arises when the WARPLAN clauses are interpreted.

As Kowalski has pointed out {10}, there are two aspects to the frame problem as it applies to the traditional situation calculus formulation. The first is the inconvenience of having to explicitly state all the frame axioms. The second is the problem of inefficiency arising from the frame axioms.

The approach to the first problem in WARPLAN is much the same as in STRIPS. One specifies only which facts are deleted by an action and assumes that all other facts are preserved. Moreover, in WARPLAN, the specification doesn't have to be explicit - delete information may be represented procedurally. Notice however that frame axioms are still used implicitly in WARPLAN's deductions. For example, the second 'holds' clause in Appendix I is clearly a universal frame axiom which can become instantiated to different frame axioms for various actions.

The efficiency problem is tackled by:-

(1) only using the frame axioms "top-down", and

(2) the ability to (implicitly) insert frame axioms into the proof of the current partial plan.

The "top-down" use of a frame axiom contrasts with a "bottom-up" use in which it is used to generate facts about a new state. In WARPLAN a fact about a state is only deduced when needed from the history of the state. To see how WARPLAN inserts frame axioms into a proof, consider say SL resolution {7} operating with the goal state clause of a situation calculus problem as top clause. In cancelling the selected literal, it has to choose between an "add" axiom or a frame axiom, and the choice is irrevocable (without backtracking). In the analagous situation in WARPLAN, only an "add" axiom could be chosen. Later, and without backtracking, any number of frame axioms could be inserted between the negative goal literal and the positive literal of the "add" axiom.

There are still, however, difficulties which might be ascribed to the frame problem, in particular the redundancies and inefficiencies in solving a conjunction of goals relating to "independent sub-worlds". I think it is important to note that the frame problem is not peculiar to PC formulations. There are analogues of frame axioms in all systems which maintain a record of more than one state or context with information structure-shared between them.

APPENDIX I. LISTING OF THE PROGRAM.

NOTE: the labels "P1:" etc. are referenced in Section 7 and are not part of the program.

```
     + operator ('&', right to left, 1).
     + operator (';', left to right, 3).
P1:  + plan (X & C,P,T,T2) -/- solve(X,P,T,P1,T1)- plan(C,P1,T1,T2).
P2:  + plan (X,P,T,T1) - solve (X,P,T,P1,T1).

S1:  + solve(X,P,T,P,T) - always (X).
S2:  + solve(X,P,T,P1,T) - holds(X,T) - and (X,P,P1).
S3:  + solve(X,P,T,X & P,T1) - add (X,U) - achieve(X,U,P,T,T1).

A1:  + achieve (X,U,P,T,T1 ; U)
          - preserves (U,P)
          - can (U,C)
          - consistent(C,P)
          - plan (C,P,T,T1)
          - preserves (U,P).
A2:  + achieve (X,U,P,T;V,T1;V)
          - preserved (X,V)
          - retrace (P,V,P1)
          - achieve (X,U,P1,T,T1)
          - preserved (X,V).

H1:  + holds(X,T;V) - add (X,V).
H2:  + holds(X,T;V) - /
          - preserved (X,V)
          - holds ( X,T )
          - preserved (X,V).
H3:  + holds(X,T) - given (T,X).

     + preserved (X,V) - mkground (X & V,0,N) - del (X,V) -/- fail.
     + preserved (X,V).

     + preserves (U,X & C) - preserved (X,U)-preserves (U,C).
     + preserves (U, true).
```

```
+ retrace (P,V,P2)
     - can (V,C)
     - retrace 1 (P,V,C,P1)
     - append (C,P1,P2).
+ retrace 1 (X & P,V,C,P1)
     - add (Y,V) - equiv (X,Y)-/- retrace 1 (P,V,C,P1).
+ retrace 1 (X & P,V,C,P1)
     - elem (Y,C) - equiv (X,Y) -/- retrace 1 (P,V,C,P1).
+ retrace 1 (X & P,V,C,X & P1) - retrace 1 (P,V,C,P1).
+ retrace 1 (true, V,C, true).

+ consistent (C,P)
     - mkground (C & P,O,N)
     - imposs (S)
     - unless ( unless ( intersect (C,S)))
     - implied (S,C & P) .
     -/- fail.
+ consistent (C,P).

+ plans (C,T) - unless (consistent(C , true)) -/- output (impossible)-newline.
+ plans (C,T) - plan(C,true,T,T1) - output(T1) - newline.

+ and (X,P,P) - elem (Y,P) - equiv (X,Y) -/.
+ and (X,P,X & P).

+ append(X & C,P,X & P1) -/- append (C,P,P1).
+ append(X,P,X & P).

+ elem (X,Y & C) - elem (X,Y).
+ elem (X,Y & C) -/- elem (X,C).
+ elem (X,X).

+ intersect (S1,S2)- elem (X,S1) - elem (X,S2) .

+ implied (S1 & S2,C) -/- implied (S1,C) - implied (S2,C).
+ implied (X,C) - elem (X,C).
+ implied (X,C) - X.
```

```
+ equal (X,X).
+ notequal (X,Y) - unless (equal(X,Y))
      - unless ( equal ( X,qqq(N1)))
      - unless ( equal ( Y,qqq(N2))).
+ equiv (X,Y) - unless ( nonequiv(X,Y)).
+ nonequiv (X,Y) - mkground (X & Y,0,N) - equal (X,Y) -/- fail .
+ nonequiv (X,Y).

+ mkground(qqq(N1),N1,N2) -/- plus(N1,1,N2).
+ mkground(qqq(N),N1,N1) -/.
+ mkground(X,N1,N2) - univ (X,.(F,L)) - mkgroundlist(L,N1,N2).
+ mkgroundlist ( .(X,L),N1,N3)
      -mkground (X,N1,N2)
      -mkgroundlist (L,N2,N3).
+ mkgroundlist(nil,N1,N1).

+ unless (X) - X -/- fail.
+ unless (X).
```

APPENDIX II. TEST PROBLEMS

.1 First STRIPS World

.1.1 Description

The problem domain and its formalisation are essentially the same as that given in the original STRIPS paper {3}. The chief differences are that the delete lists and certain other facts are represented procedurally and various "types" checks are handled automatically by the unification algorithm. The domain is of interest for comparing the performance of WARPLAN with similar planning systems:-

System	Implemented In	Machine
WARPLAN	PROLOG (interpreted in Fortran)	IBM 360-67
STRIPS	LISP (partially compiled)	PDP 10
LAWALY	LISP (interpreted)	CDC 6600

The times quoted below are total CPU times. The STRIPS times exclude garbage collection. PROLOG does no garbage collection (although space is reclaimed on backtracking) as the implementation is carefully designed to conserve storage. What weightings, if any, to apply to the different figures seems to be largely a subjective matter.

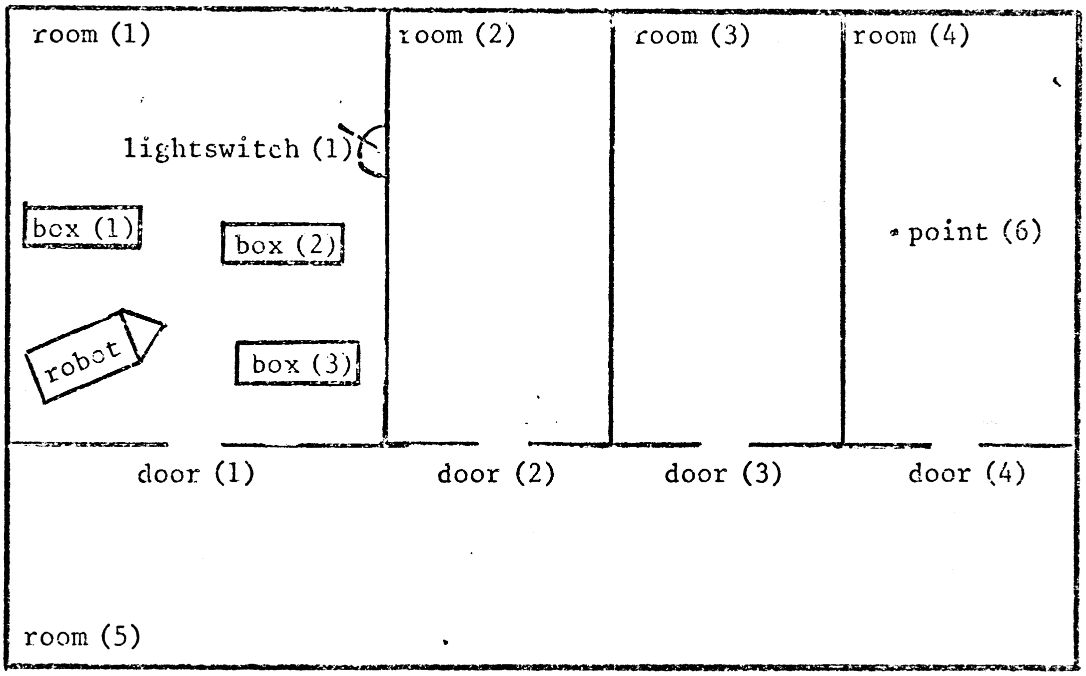
.1.2 Database

```
+ add( at(robot,P),           goto1(P,R)        ).
+ add( nextto(robot,X),       goto2(X,R)        ).
+ add( nextto(X,Y) ,          pushto(X,Y,R)     ).
+ add( nextto(Y,X) ,          pushto(X,Y,R)     ).
+ add( status(S,on),          turnon(S)         ).
+ add( on(robot,B),           climbon(B)        ).
+ add( onfloor,               climboff(B)       ).
+ add( inroom(robot,R2),      gothru(D,R1,R2)   ).
```

```
+ del ( at (X,Z),U) - moved (X,U).
+ del ( nextto(Z,robot),U) -/- del(nextto(robot,Z),U).
+ del ( nextto(robot,X),pushto(X,Y,R)) -/- fail.
+ del ( nextto(robot,B), climbon(B))-/- fail.
+ del ( nextto(robot,B),climboff (B))-/- fail.
+ del ( nextto(X,Z),U) - moved (X,U).
+ del ( nextto(Z,X),U) - moved (X,U).
+ del ( on (X,Z),U) - moved (X,U).
+ del ( onfloor,climbon (B)).
+ del ( inroom(robot,Z),gothru (D,R1,R2)).
+ del ( status(S,Z),turnon (S)).

+ moved (robot,goto 1 (P,R)).
+ moved (robot,goto 2 (X,R)).
+ moved (robot,pushto (X,Y,R)).
+ moved (X,pushto (X,Y,R)).
+ moved (robot,climbon (B)).
+ moved (robot,climboff (B)).
+ moved (robot,gothru(D,R1,R2)).

+ can ( goto 1 (P,R),
        locinroom (P,R) & inroom(robot,R) & onfloor).
+ can (goto 2 (X,R),
        inroom (X,R) & inroom(robot,R) & onfloor).
+ can ( pushto(X,Y,R),
        pushable (X) &inroom (Y,R) & inroom (X,R) &
        nextto(robot,X) & onfloor).
+ can ( turnon(lightswitch (S)) ,
        on(robot,box(1)) & nextto(box(1),lightswitch(S))).
+ can ( climbon(box(B)) ,
        nextto(robot,box(B)) & on floor).
+ can ( climboff(box(B)),
        on(robot,box(B) )).
+ can ( gothru(D,R1,R2),
        connects(D,R1,R2) & inroom(robot,R1) &
        nextto(robot,D) & onfloor).
```

```
+ always ( connects (D,R1,R2)) - connects 1(D,R1,R2).
+ always ( connects (D,R2,R1)) - connects 1(D,R1,R2).
+ always ( inroom (D,R) ) - always ( connects (D,R,R1)).
+ always ( pushable ( box (N))).
+ always ( locinroom ( point (6) , room (4) )) .
+ always ( inroom ( lightswitch (1), room (1) )).
+ always ( at ( lightswitch (1), point (4) )).

+ connects 1 (door (N) ,room (N) ,room (5) ) - range (N,1,4).
+ range (M,M,N) .
+ range (M,L,N) - notequal (L,N) - plus (L,1,L1)- range (M,L1,N) .

+ given ( strips 1 ,   at ( box (N),point (N) )) - range (N,1,3).
+ given ( strips 1,    at ( robot, point (5) )) .
+ given ( strips 1,    inroom ( box (N) , room (1) )) - range (N,1,3).
+ given ( strips 1 ,   inroom ( robot,room (1) )).
+ given ( strips 1,    onfloor).
+ given ( strips 1,    status ( lightswitch (1), off)).
```

strips 1:



.1.3 Problems

	Goal(s)	Plan length	CUP time in secs. WARPLAN	STRIPS	LAWALY
(1)	status(lightswitch(1),on)	4	9	65*	1.6
(2)	nextto(box(1),box(2)) & nextto(box(2),box(3))	4	21	122	4.1
(3)	at(robot,point(6))	5	9	125	2.6
(4)	nextto(box(2),box(3)) & nextto(box(3),door(1)) & status(lightswitch(1),on) & nextto(box(1),box(2)) & inroom(robot,room(2))	15++	95+	-	10.3**

* :non-optimal solution

** :to produce a similar 15-step plan.

\+ :time to produce all but the last step, at which point, apparently, an error in PROLOG caused the system to crash.

++ :solution produced by WARPLAN is:-

```
strips 1 ;
goto 2 (box(3),room(1) ) ;
pushto ( box(3),door(1),room(1) );
goto 2 (box(2),room(1) ) ;
pushto ( box(2),box(3),room(1) ) ;
goto 2 (box(1),room(1) );
pushto(box(1),lightswitch(1),room(1) ) ;
climbon(box(1) ) ;
turnon(lightswitch(1) );
climboff(box(1) );
*goto 2 (box(1),room(1) ) ;
pushto(box(1),box(2),room(1) );
goto 2 (door(1),room(1) ) ;
gothru(door(1),room(1),room(5) );
goto 2 (door(2),room(5) );
gothru(door(2),room(5),room(2) )
```

.1.4 Comments

The times quoted are for orderings of axioms, goals and action preconditions exactly as above. Apart from the goals, and preconditions of 'turnon', these orderings have been carefully chosen to produce best results with the present depth-first search strategy.

It is interesting to note that, for this problem domain, there is a single "natural" order for action preconditions, corresponding to the order in which one would expect them to be achieved. Thus the preconditions for 'turnon' are not stated in the natural order of:-

nextto(box(1),lightswitch(S)) & on(robot,box(1))

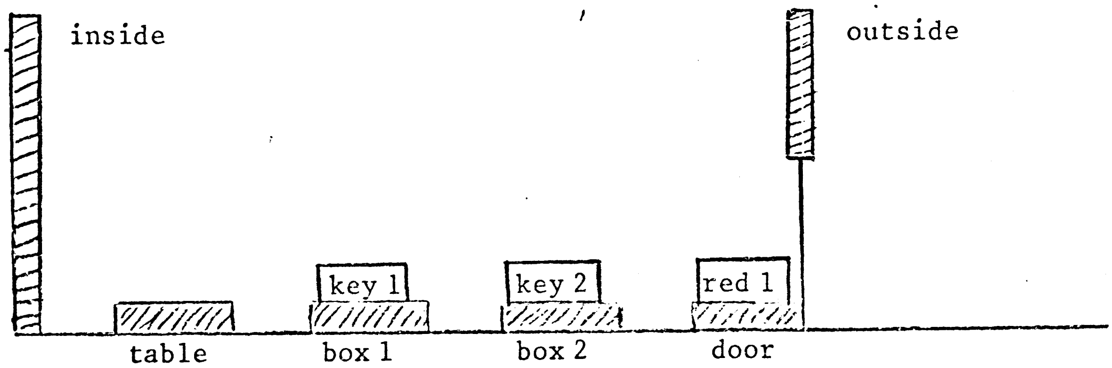
Nevertheless WARPLAN still manages to find the optimal solution to 'status(lightswitch(1),on)', unlike STRIPS.

In the solution for problem (4), WARPLAN generates the superfluous step marked with an asterisk, owing to the flaw mentioned in Section 5 concerning non-ground terms.

.2 A Version of the Keys and Boxes Problem (of D. Michie).

.2.1 Description

This is a substantially simplified version of a "benchmark test" of Michie {11}.



The world comprises two areas 'inside' and 'outside'. There are four distinct locations 'inside', namely 'table', 'box 1', 'box 2', 'door'. There is a robot which is able to move about and transport objects. If the robot attempts to pickup an object at a location, all that can be ascertained is that it will be holding one of the objects, if any, at that location. In our simplified formulation, the robot is only allowed to pickup an object if it is the only object at a location. There are two actions 'go' and 'take'; (for technical reasons, explained later, there are two versions of 'take'). The robot is only allowed to 'go' or 'take' something 'outside' if both the objects 'key 1' and 'key 2' are at the 'door'. In the initial state 'key 1', 'key 2', 'red 1' are the only objects at, respectively, 'box 1', 'box 2', 'door' and nothing is at the 'table'. The goal is to have 'red 1' 'outside'.

.2.2 Database

```
+ operator('is',righttoleft,4).
+ operator('set',prefix,5).
+ operator('placed',prefix,5).
+ operator('only',prefix,5).
+ operator('at',righttoleft,6).

+ add( position is P,              go (P)          ).
+ add( X is placed Q,              take(X,P,Q)     ).
+ add( set placed P is nothing,    take(X,P,Q)     ).
+ add( set placed Q is only X,     take 1(X,P,Q)   ).

+ add( Z,take 1 (X,P,Q) ) - add(Z,take(X,P,Q)).

+ del( position is Z,              go (P)          ).
+ del( X is placed Z,              take (X,P,Q)    ).
+ del( position is Z,              take (X,P,Q)    ).
+ del( set placed Q is Z,          take (X,P,Q)    ).
+ del( set placed P is Z,          take (X,P,Q)    ).

+ del( Z ,take 1 (X,P,Q) - del(Z,take(X,P,Q) ).

+ can( go(inside at L), true).
+ can( take(X,inside at L1,inside at L2),
          set placed inside at L1 is only X &
          position is inside at L1 ).
+ can( take 1 (X,inside at L1,inside at L2),
          set placed inside at L2 is nothing &
          set placed inside at L1 is only X &
          position is inside at L1 ).
+ can( take(X,inside at L, outside),
          set placed inside at L is only X &
          position is inside at L &
          key 1 is placed inside at door &
          key 2 is placed inside at door ).

+ always(true).
```

```
+ given(start kb,  set placed inside at table is nothing    ).
+ given(start kb,  set placed inside at box 1 is only key 1 ).
+ given(start kb,  set placed inside at box 2 is only key 2 ).
+ given(start kb,  set placed inside at door is only red 1  ).

+ imposs(position is P & position is Q & P ≠ Q) .
```

.2.3 The Problem

```
- plans( red 1 is placed outside, startkb).
```

Ans:

```
startkb;
go(inside at door);
take 1(red 1, inside at door, inside at table);
go(inside at box 1);
take(key 1, inside at box 1, inside at door);
go(inside at box 2);
take(key 2, inside at box 2, inside at door);
go(inside at table);
take(red 1 ,inside at table, outside)
```

Time: 29 secs.

.2.4 Commentary

With the present depth-first search strategy, getting a solution in an acceptable time was dependent on:-

(1) the right orderings of action preconditions etc.;

(2) omitting certain unnecessary axioms, e.g.

+ add(position is Q, take (X,P,Q)).

(3) a formulation using 'take' rather than 'pickup' and 'letgo'.

The reason for the two versions of 'take' is that an action must have a unique set of preconditions, and the effects of a 'take' depend on the set of objects at the destination. This is symptomatic of an oversimplification in WARPLAN which can and should be rectified.

.3 Machine Code Generation

.3.1 Description

I invented the following example to demonstrate that the system is general purpose. The example suggests an interesting area for future applications.

There is a very simple computer comprising an accumulator and an unspecified number of general purpose registers. There are just four instructions 'load', 'store', 'add', 'subtract'. To axiomatise the domain for WARPLAN, it was necessary to follow each instruction in an assembly language program by a comment. The comment is introduced by '!' and states the value which will be in the accumulator after the instruction has been executed. Such comments are often needed by human programmers too!

.3.3 Database

```
+ operator ('!',righttoleft,4).
+ operator ('is',righttoleft,4).
+ operator ('+',lefttoright,5).
+ operator ('-',lefttoright,5).
+ operator ('load',prefix,5).
+ operator ('add',prefix,5).
+ operator ('subtract',prefix 5).
+ operator ('store',prefix,5).
+ operator ('reg',prefix,5).
```

+ add (acc is $V_1 + V_2$, add R ! $V_1 + V_2$).
+ add (acc is $V_1 - V_2$, subtract R ! $V_1 - V_2$).
+ add (acc is V, load R ! V).
+ add (reg R is V, store R ! V).

+ del (acc is Z,U) - add (acc is V,U) .
+ del (reg R is Z,U)- add (reg R is V,U) .

+ can (load R ! V, reg R is V).
+ can (store R ! V, acc is V).
+ can (add R ! $V_1 + V_2$, reg R is V_2 & acc is V_1).
+ can (subtract R ! $V_1 - V_2$, reg R is V_2 & acc is V_1).

```
+ given (init,   reg 1 is c1 ).
+ given (init,   reg 2 is c2 ).
+ given (init,   reg 3 is c3 ).
+ given (init,   reg 4 is c4 ).
```

.3.3 Problems

(1) - plans(acc is (c1-c2)+(c3-c4), init).

Ans:

```
init;
load 3        !c3;
subtract 4    !c3-c4;
store X1      !c3-c4;
load 1        !c1;
subtract 2    !c1-c2;
add X1        !(c1-c2)+(c3-c4)
```

(2) - plans(acc is (c1-c2)+(c1-c2), init).

Ans:

```
init;
load 1        !c1;
subtract 2    !c1-c2;
store X1      !c1-c2;
add X1        !(c1-c2)+(c1-c2)
```

(3) - plans(reg 1 is c1+(c2-c3) &
reg 2 is c2-c3 &
reg 3 is c4+c4, init).

Ans:

```
init;
load 2        !c2;
subtract 3    !c2-c3;
store 2       !c2-c3;
load 1        !c1;
add 2         !c1+(c2-c3);
store 1       !c1+(c2-c3);
load 4        !c4;
add 4         !c4+c4;
store 3       !c4+c4
```

Time: 18 secs

(4) - plans(reg 1 is c1+(c2-c3) &
reg 2 is c2-c3 &
acc is c1, init).

```
Ans:    init;
        load 2         !c2;
        subtract 3     !c2-c3;
        store 2        !c2-c3;
        load 1         !c1;
        store X1       !c1;
        add 2          !c1+(c2-c3);
        store 1        !c1+(c2-c3);
        load X1        !c1
```

Note. The first branch in WARPLAN's search space for this problem is infinite. Interactive intervention to block this branch resulted in the above solution being found without any further assistance.

.3.4 Comments

Here WARPLAN is behaving as a simple "compiler-compiler" from PC to machine code using a machine definition also written in PC. It is interesting to note that a very uniform proof procedure with no special domain-dependent heuristics can generate nicely "optimised" code automatically, as first solution.

Of course, once again I have "cheated" slightly by taking advantage of the freedom to order clauses and the terms in a conjunction to get the best results.

APPENDIX III. SUMMARY OF PROLOG.

.1 Introduction

PROLOG is an elegant and powerful programming language developed at the University of Marseille. It bears certain similarities with PLANNER {6}. Alternatively, one may regard the PROLOG interpreter as an efficient PC theorem prover for problems renameable as Horn clauses. The present system {1} is implemented partly in FORTRAN, partly in PROLOG itself, and running on an IBM 360-67 achieves roughly 200 unifications per second. (An earlier version of PROLOG is described in {2}.)

The features of PROLOG used in WARPLAN are summarised below. I have modified the syntax in certain inessential respects to improve legibility and to assist Anglo-saxon readers. The chief differences in the implemented form are:-

(1) only upper-case is used, so variables are prefixed by an '*' (asterisk);

(2) certain names (of evaluable predicates) are, of course, their French equivalents (and may soon be changed anyway).

.2 Syntax

The syntax is essentially the same as PC in clausal form.

Positive literals are preceded by '+', negative by '-'. The literals of a clause are simply concatenated with no explicit sign for disjunction. Each clause is terminated by a '.' .

An identifier is either a sequence of alpha-numeric characters or a <u>single</u> non-alpha-numeric character. Identifiers for variables commence with an upper case letter; other identifiers denote predicates, functions or constants according to context.

Certain functions may be specified as infix or prefix operators to improve readability. Terms containing such functions are converted to the standard form on input and (nice point) are converted back on output. An operator must be declared before it is first used, by an axiom of the form:-

+ operator (F,D,P).

where

F is the name of the operator (function);

D is either 'prefix' or else specifies, for an infix operator, the direction of association (default bracketing).

P is a number indicating the level of the operator in a precedence hierarchy.

Thus using the operators specified in Appendix I and Appendix III.3.2, the following terms are equivalent:-

reg 1 is c1+c2-c3 & reg 2 is c4 & acc is c1
reg 1 is (c1+c2)-c3 & (reg 2 is c4 & acc is c1)
&(is(reg(1),-(+(c1,c2),c3)),&(is(reg(2),c4), is(acc,c1)))

.3 Semantics

The "denotational" semantics is essentially the same as PC in clausal form.

The "operational" semantics, or proof procedure of the PROLOG interpreter, is as follows. The interpreter attempts to cancel (by resolution) the literals of a clause left-to-right, depth-first with backtracking. When attempting to cancel a literal, the candidate complementary literals are restricted to the leftmost literals of input clauses. The candidate input clauses are tried in the order in which they appear in the list of input clauses. Thus it is a linear inference system which may be described as SL resolution {7}, without merging, or ancestor resolution, with leftmost literal in a cell as selected, and with a depth-first search strategy. There is no "occur check" in unification. (Ancestor resolution and merging can be achieved by explicit programmer action.)

Certain predicates are defined as "evaluable", and behave as built-in procedures (frequently with side effects):-

'/' (slash): when this predicate is cancelled, it has the effect of prohibiting backtracking on any choices since (and including) the time the literal complementary to the leftmost literal in the input clause containing the '/' was cancelled. Thus it makes the choice of the current clause deterministic.

'plus (X,Y,Z)': adds the integers X and Y to yield result Z.

'output(X)': prints the term X on the user's terminal, 'newline': causes a new line to be started on the user's terminal.

'univ(X,Y)' : takes a term X and returns a list Y of which the first element is the name of the principal function of X (represented as a list of characters), and the remaining elements are that function's arguments in X. A list is constructed from the function '.' and constant 'nil' just as LISP uses CONS and NIL. e.g:-

```
univ ( fun (a, X, foo(Y)) ,
        . ( . (f, . (u, . (n,nil))),
        . ( a,
        . ( x,
        . ( foo (Y),nil)))))
```

or given that '.' is declared as a right-to-left infix operator:-

```
univ (fun (a, X, foo (Y)),
        (f.u.n.nil).a.X .foo (Y) .nil)
```

A variable may appear in place of a literal in a clause. This is analagous to allowing procedures to be a data-type in other programming languages. For an example of its use, see 'unless' in Appendix I.

ACKNOWLEDGEMENTS

to Professor Donald Michie, my superviser, for encouraging me to work on theorem provers and robot planning;

to Austin Tate, who is working on similar problems, for much stimulating exchange of ideas;

to Philippe Roussel, Bob Pasero, Alain Colmerauer and other members of the Groupe d'Intelligence Artificielle, Marseille, for their hospitality, both personal and professional;

and especially

to Robert Kowalski, my second superviser, from whom I first learnt the wonders of Predicate Calculus as programming language, for his help and encouragement and for arranging for me to visit Marseille.

I have obviously drawn heavily on the ideas of Cordell Green and the designers of STRIPS.

The work was supported by an SRC research studentship. My visit to Marseille was financed under a NATO grant.

REFERENCES

{1} Battani, G. and Meloni, H.
'Interpreteur du langage de programmation PROLOG'.
Université d'Aix Marseille, 1973.

{2} Colmerauer, A., Kanoui, H., Pasero, R. and Roussel, P.
'PROLOG: Un système de communication homme-machine en français'.
Université d'Aix Marseille.

{3} Fikes, R.E. and Nilsson, N.J.
'STRIPS: A new approach to the application of theorem proving to problem solving'.
Proceedings of IJCAI 2, pp. 608-620, 1971.

{4} Green, C.
'Application of theorem proving to problem solving'.
Proceedings of IJCAI 1, pp. 219-239, 1969.

{5} Hayes, P.
'A logic of actions'.
Machine Intelligence 6, pp. 495-520, Edinburgh, 1971.

{6} Hewitt, C.
'Description and theoretical analysis of PLANNER'.
AI Memo 251, MIT, 1972.

{7} Kowalski, R. and Kuehner, D.
'Linear resolution with selection function'.
Artificial Intelligence 2, pp. 227-260, 1971.

{8} Kowalski, R.
'Predicate Logic as programming language'.
DCL Memo 70, University of Edint rgh, 1973.

{9} Kowalski, R.
'A proof procedure using connection graphs'.
DCL Memo 74, University of Edinburgh, 1973.

{10} Kowalski, R.
'Logic for problem solving'.
DCL Memo 75, University of Edinburgh, 1974.

{11} Michie, D.
'On Machine Intelligence',
pp. 149-152, Edinburgh, 1974.

{12} Raphael, B.
'The frame problem in problem solving systems'.
in Artificial Intelligence and Heuristic Programming,
pp. 159-169, Elsevier 1971.

{13} Sacerdoti, E.D.
'Planning in a hierarchy of abstraction spaces'.
Proceedings of IJCAI 3, pp. 412-422, 1973.

{14} Siklossy, L. and Dreussi, J.
'An efficient robot planner which generates its own procedures'.
Proceedings of IJCAI 3, pp. 423-430, 1973.

{15} Siklossy, L. and Roach J.
'Proving the impossible is impossible is possible: disproofs based on hereditary partitions'.
Proceedings of IJCAI 3, pp. 383-392, 1973.

{16} Sussman, G.J.
'HACKER: A computational model of skill acquisition'.
Ph.D. Thesis, MIT, 1973.