Printing Requisition / Graphic Services

20941

1. **Please complete unshaded areas** on form as applicable.
2. **Distribute copies** as follows: White and Yellow to Graphic Services. **Retain Pink Copies for your records.**
3. On completion of order the **Yellow copy will be returned** with the printed material.
4. **Please direct enquiries, quoting requisition number and account number,** to **extension 3451.**

TITLE OR DESCRIPTION
CS-84-48 Madame: A Planner for Ish

DATE REQUISITIONED: January 16, 1985 (20)
DATE REQUIRED: ASAP
(reimburse)
ACCOUNT NO.: 1 2 6 4 4 3 1 0 2

REQUISITIONER – PRINT: A. Harris
PHONE:
SIGNING AUTHORITY: D. McCracken per A Harris

MAILING INFO – NAME: D. McCracken DEPT.: Computer Science BLDG. & ROOM NO.: M&C 6081E [X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 220
NUMBER OF COPIES: 100 ~~50~~

TYPE OF PAPER STOCK
[X] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] ____

PAPER SIZE
[X] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] ____

PAPER COLOUR: [X] WHITE [ ] ____
INK: [ ] BLACK [ ] ____

PRINTING: [ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.
NUMBERING: FROM ____ TO ____

BINDING/FINISHING
[X] COLLATING [ ] STAPLING [ ] ____ HOLE PUNCHED [X] PLASTIC RING

FOLDING/PADDING
CUTTING SIZE

Special Instructions

Covers and Backs are enclosed

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

TYPESETTING — QUANTITY | OPER. NO. | TIME | LABOUR CODE

PROOF
P R F
P R F
P R F

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE
F L M
F L M
F L M
F L M
F L M

PMT
P M T
P M T
P M T

PLATES
P L T
P L T
P L T

STOCK

BINDERY

OUTSIDE SERVICES

$ ____ COST

TAXES PROVINCIAL [ ] FEDERAL [ ]

Madame:
A Planner for ISH

by

J.A.N. Andre Trudel

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

Research Report CS-84-48
December 1984

Madame: A Planner for ISH†

J. A. N. André Trudel

Logic Programming and Artificial Intelligence Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

An ongoing project at the University of Waterloo is the design and construction of an interactive UNIX‡ consultant, ISH (Intelligent Shell). The consultant has been designed to answer the type of questions normally posed to its human counterpart.

A planner called "Madame" was developed to be used as ISH's planning component. Madame is based on D.H.D. Warren's Warplan which is a goal directed planner that uses goal regression.

An important feature of Madame is a data structure called a "spider". The spider is used to store previously generated goal states. These states then serve as alternate start states for Madame, so Madame has many start states at its disposal instead of only one. Experiments show that the spider does increase the efficiency of Madame.

Madame required an axiomatization of the UNIX domain. The axiomatization developed is based on work done by J. Pakalns and J. Malito at the University of Waterloo. To encode the axioms, the method presented by R. Kowalski in "Logic for Problem Solving" is used.

This dissertation describes Madame, the spider, and the UNIX axiomatization.

---

† This report was originally submitted as a Master's thesis to the Department of Computer Science, Faculty of Mathematics, at the University of Waterloo by the author.

‡ UNIX is a Trademark of Bell Laboratories

Acknowledgements

The author would like to thank Randy G. Goebel and Cynthia "Madame" Trudel for their help and support.

Table of Contents

List of Figures

1

Introduction

1.1 Introduction

An ongoing project at the University of Waterloo is the design and construction of an interactive UNIX† consultant, ISH (Intelligent Shell). The consultant has been designed to answer the type of questions normally posed to its human counterpart.

The basic components of ISH are shown in Figure 1-1. A user will be able to pose English queries to ISH. For example, a query could be:

How do I create a file called "temp" in my working directory?

The component of ISH which receives user's queries is the natural language front-end called UWISH [MAL84]. UWISH will transform the English query into one which uses first order logic. This translated input is then passed to Madame, the planning component of ISH. An axiomatization (description) of the UNIX world called AHA [PAK83; MAL83] is used by Madame to find the correct sequence of UNIX commands to answer the user's query. This sequence of commands is then passed back to UWISH which in turn passes it to the user.

Madame, which is discussed in this dissertation, is a general purpose planner; that is, it can be used in domains other than the UNIX application. Furthermore, Madame uses a data structure called a spider that improves its efficiency. This spider is planner independent; it can be used with any planner.

This first chapter is an overview of planning. Chapter 2 describes the work done in producing a UNIX axiomatization to be used by Madame. The spider and how it is used by Madame is presented in Chapter 3. Madame's implementation is described in Chapter 4. Test results that show the increased efficiency of a planner when using the spider and suggestions for future research appear in Chapter 5.

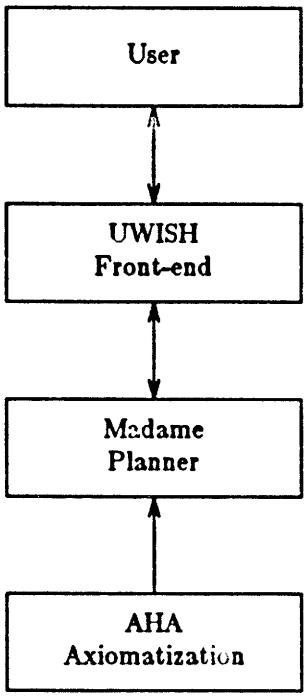
It is assumed that the reader is familiar with first-order logic and Prolog. Madame was implemented in Waterloo Unix Prolog (WUP) [EMD84]. An important feature of WUP, is it allows the user to break his program into modules. Each module resides in a different UNIX directory. These modules can then be accessed through WUP. After WUP has been invoked, the modules cannot be altered. WUP provides an auxiliary database that acts as a temporary workspace where axioms can be added and deleted.

1.2 Planning

Many problem domains can be represented as a collection of different states. State transitions can be modelled by a set of actions which operates on these states. An instance of the planning problem is: given a particular start state and goal state, construct a plan, i.e. a sequence of

† UNIX is a Trademark of Bell Laboratories.

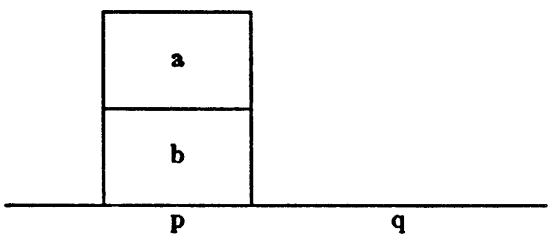
Figure 1-1: ISH Components



actions which transforms the start state into the goal state. A system that builds plans is called a *planner*.

The classical example of a problem domain is the blocks world. The blocks world consists of blocks which can be stacked one on top of another or placed on the floor at specific locations. A plan would consist of a sequence of move actions that transforms the start state into the goal state. Figure 1-2 depicts a state in a particular blocks world. This world consists of two movable blocks labelled "a" and "b". There are only three possible locations for a block in this world: on top of another block or, at location "p" or "q" on the floor.

Figure 1-2: The Blocks World



The planning problem is equivalent to finding a path in a graph. The states represent nodes,

and the actions that transform one state into another are the edges. The graph defined by the states and actions is called a *state space graph*. The problem is to find a path from the start state to the goal state. This path is usually not unique, because multiple plans may exist. If there is a unique plan, it can be modified by adding trivial actions (for example, move a block off a stack and then immediately move it back onto the same stack).

There are three basic approaches to path-finding; a forward search, a backward search, and a bi-directional search. A forward search begins at the start state and tries to find a path to the goal state. A backward search works in the opposite direction, from the goal state to the start state. The combination of the two, a bi-directional search, tries to find a path from the start state to the goal state and at the same time, a path from the goal state to the start state. It is hoped that both searches meet at a common state somewhere between the start and goal states.

Most planners use a backward search. One exception is BLOCKHEAD [KIB81], which uses a forward search. As the name implies, BLOCKHEAD was used in the blocks world and was found to be very effective. However, in contrast to the blocks world, there are domains that have many more possible actions and a larger number of objects in the start state. In this kind of situation, the forward planner would have many possible actions to apply to the start state. Each action would have variations because of the number of objects available and this could lead to a combinatorial explosion.

Bi-directional search is rarely used. The problem is, there is no guarantee that a common state will be found between the start and goal states. This can cause the planner to produce two distinct plans emanating from the start and goal states, thus doubling the work required.

Madame uses a combination of the three search techniques. First, a forward search is done to generate a partial plan, and then this partial plan is completed using a backward search. Since a forward and backward search is used, it appears that the search is bi-directional. However, the searches are not being done simultaneously; thus, the search is not purely bi-directional. A more complete description of the search used by Madame will be presented in Chapter 4.

Requests for plans are given to Madame, and many other planners, in the form of a list of goals. Sometimes, this list will define a unique goal state. However, more frequently the input list will describe a set of goal states. For example, if the list of goals to a particular planner were "block a is on block b, and nothing is on block a", then Figure 1-2 is a valid goal state. Another goal state would be to have the two blocks piled on position "q". Since the goal state description passed to a planner may not describe a goal state uniquely, the planner has the freedom of building a plan from the start state to any one of the valid goal states.

Some planners, such as Warplan [WAR74], only accept a positive input description of the goal state (i.e. none of the goals are negated). For example, Warplan would accept "a is on b" as a valid goal, but not "a is not on b". Madame has been designed to accept negative goals.

Actions are not limited to simple state change operators. Georgeff [GEO83] considers actions to be represented by sequences of states. For example, in the course of moving block "a" onto block "b", an intermediate state of the action is that block "a" is being held. The move action is therefore represented by three states; the traditional "before" and "after" states and one "intermediate" state. This method of representing actions is used to model what is happening in the world while the action is being executed. This additional information is very helpful when two plans are being executed simultaneously, for example, when two robots work together. The plans to be executed by both robots must be carefully examined to detect any conflicts. An example of a conflict is both robots trying to pick up the same block simultaneously. Since Madame has complete knowledge of the effects of an action and, no interaction of subplans occurs, this representation was not needed.

1.3 Planning Strategies

Planning is a very difficult problem, but there exists "a diverse bag of tricks for improving the efficiency" [SAC79] of a planner. A planner needs strategies to guide its traversal of the state space graph. Since there may be an infinite number of states, it is impossible to visit all of them. The following is a summary of the strategies presented by Sacerdoti [SAC79].

Hierarchical Planning: There are two ways of doing hierarchical planning. The first is to order the goals, and work on the most important one first. Thus, the planner always concentrates on the most important and critical aspect of the problem. There is a problem with this approach. There does not exist a general method to decide the relative ordering of the goals. Also, if a tie between goals occurs, there is no general heuristic available to decide how the tie should be broken. A dynamic scheme would have to be devised because often the relative ordering of goals depends on the particular problem being tackled. The second approach to hierarchical planning is to "abstract the descriptions of the actions" [SAC79]. A plan comprised of abstracted actions is produced much more easily. Parts of this plan are then recognized as requiring more work. The strategy is applied recursively to these sub-sections of the plan. The plan is developed in layers, each layer containing more detail than its predecessor.

Hierarchical Plan Repair: This strategy is very similar to hierarchical planning. It uses hierarchical planning, but if a sub-plan at any level causes unanticipated effects for the plan at a higher level, various patching methods are applied.

Bugging: The bugging strategy uses the fact that it is easier to find a plan for an oversimplified problem. Because of the oversimplifications, the plan will contain "bugs". "However, if the oversimplifications are designed properly, then only bugs of a limited number of types will be introduced, and relatively simple mechanisms can be implemented to remedy each expected type of bug" [SAC79].

Special-Purpose Subplanners: If the planner can recognize certain types of goals, it can pass these goals to subplanners which were designed to handle these very special cases. For example, if working with a robot application, a subplanner may be written for each room in a house. The appropriate subplanner would be called when the robot tries to move around in a particular room. But as Warren [WAR74] points out, "writing special purpose plan generators is at best tedious, and at worst near impossible, if the specification of the world is liable to change".

Constraint Satisfaction: If a planner uses a backward search, many goals will contain unbound (unassigned) variables. This occurs because only a partial description of the goal state is generated by each back chaining step. There are many alternatives available for binding these variables. For example, if a goal contains a variable which is to be bound to a block and there are many blocks in the world, then many different valid bindings exist for the one variable. The constraint satisfaction strategy chooses the correct binding of variables and thus, greatly improves the efficiency of a planner.

Relevant Backtracking: There are usually many different actions which can be applied to a particular state. The planner must choose between these actions and it is not guaranteed to make the correct choice. At some state, the planner may realize that there is an error in the subplan that has been developed. The planner must then return to a previously visited state and try another action. Relevant backtracking involves backtracking to a point such that the detected problem is removed. "Normal" backtracking would only backtrack to the first state where a choice of actions is available.

Disproving: Most planners, Madame included, are eternal optimists. They assume that any goal state can be reached and will work diligently towards a solution. An alternative strategy is to take a pessimistic view and try to disprove a goal. If the goal cannot be disproved, time and effort has not been wasted because insight to the solution of the goal will have been obtained. This planning strategy is nice in theory, but difficult to implement. The system must first acquire then exploit the "insight".

Pseudo-Reduction: The pseudo-reduction strategy is to work on each goal independently. Then, the individual plans for each goal are combined to make one master plan. There is a trade-off when this strategy is used; the planning phase is simplified but the savings are counteracted by the great effort needed to combine the plans.

Goal Regression: When a goal G1 is regressed over an action A1, a set of new goals G2 is produced. If G2 is true in a state S, and A1 is applied to state S, a new state T will be reached. The original goal G1 will be true in state T. For example, assume there are two goals and a plan has been found for the first goal. If an action exists such that when it is appended to the end of the plan, it produces the goal state, then the plan is finished. Otherwise, the second goal is regressed over the last action in the plan. A set of new goals is produced which describes a state, such that if the last action of the original plan is applied to this state, then the goal state is produced. Planning now continues with the set of new goals and the plan minus its last action. Goal regression can be applied again if required. This strategy is used in Warplan [WAR74] and Madame, and will be described in greater detail in a later chapter.

In addition to goal regression, Madame also uses a strategy that does not fit into one of the above categories. Because of this, another strategy is added to Sacerdoti's list.

Multiple Knowledge Sources Every planner will have at least one knowledge source, the axiomatization of the problem domain. In order to increase the planner's efficiency, other knowledge sources can be made available to the planner. In the case of Madame, a partial plan is passed that produces a state in which some of the goals are true. Thus, part of the planners work has been done; it need only modify this plan so that it produces the goal state. Other information which could possibly be passed to a planner includes special knowledge about the problem domain, the goals currently being worked on, which actions should be tried first, or which goals are more important. This information would also be useful to the other strategies. For example, hierarchical planning would use the information about the relative ordering of the goals.

1.4 Completeness

A planner is *complete* if it will generate a plan when one exists and there are no restrictions placed on that plan; that is, the plan is not required to be optimal. A plan is *optimal* if there is no other plan containing fewer actions which will transform the start state into the goal state. This definition of completeness is weaker than Warren's [WAR74] which is: "a plan generator is complete if it will eventually generate an optimal plan for any problem". The use of the weaker definition keeps the problems of completeness and optimality separate. Both are important, and should be considered separately.

Madame is neither complete nor optimal. The reason it is not optimal will be described in Chapter 3. The planning part of Madame is based on Warplan [WAR74], which is not complete.

To completely solve the planning problem, a planner would have to be complete, optimal, and efficient. As yet, no one strategy or a combination of strategies has been found to completely solve the planning problem.

1.5 Specifying the Problem Domain

Sacerdoti [SAC79] classifies the capabilities of a planner into three categories: management of state description models, deductive machinery, and action models. A state description model is defined as "a specification of the state of the world at some time" [SAC79]. The state description model is usually not complete, as all the information about every state cannot be encoded because of storage limitations and time inefficiencies. Therefore, some deductive mechanism is required to retrieve information about a state. In addition to the management of state description models and deductive machinery, a planner requires action models. Action models are a description of the effect an action has on a state. For example, when an action is applied to a state, the state is transformed into a new state. The action model must specify what is retained and what is deleted in the new state.

In Madame, the axioms used to describe the state description models, the action models, as well as the axioms used by the deductive machinery, are all based on Kowalski [KOW79]. This method will be referred to as *Kowalski's representation.*

The start state is the only state for which there is an explicit state description model. An example is the simplest way to explain the axioms used in the state description model of the start state. Assume the start state was the one shown in Figure 1-2. The axioms used to describe Figure 1-2 are:

(1) poss(0);
(2) holds(on(a b) 0);
(3) holds(on(b p) 0);
(4) holds(clear(a) 0);
(5) holds(clear(q) 0);
(6) manip(a);
(7) manip(b);

The start state is denoted by "0". Axiom (1) specifies that the start state is possible (i.e. it is a valid state). Axioms (2)-(5) describe how the blocks are arranged in the start state and axioms (6)-(7) specify that "a" and "b" are manipulatable.

The deductive machinery will use Kowalski's [KOW79] frame axiom to retrieve information about a state. The frame axiom is his solution to the frame problem. The frame problem arises when an action operates on a state to produce a new state. Some statements which were true in the old state remain true in the new state. This must be specified in some manner; this is the frame problem. There are two approaches to the frame problem; one is space inefficient and the other is time inefficient. The space inefficient solution is to keep a complete state description model for every state. This solution can also become time inefficient if the amount of data stored becomes too large. The alternative solution is to use the frame axiom:

(8) holds(Statement result(Action State)) <-
holds(Statement State)
preserves(Action Statement);

The above axiom says that a "Statement" is true in the new state produced by applying "Action" to "State" (first holds) if the "Statement" was true in "State" (second holds) and if "Action" does not delete "Statement" (preserves). The frame axiom backchains through the states and must often backchain all the way to the start state. This causes the time inefficiencies.

Certain preconditions must be true before an action can be applied to a state. These

preconditions are specified as part of Sacerdoti's action model. The action model must specify three things for every action:

1. Preconditions
2. Added statements
3. Deleted statements

For an example of an action model, consider the action of moving block A from B to C, trans(A B C). In Kowalski's representation, the action model is:

```
(9)   poss( result(trans(A B C) State) ) <-
          poss( State )
          manip( A )
          diff( A C )
          holds( clear(A) State )
          holds( clear(C) State )
          holds( on(A B) State );
(10)  holds( on(A C) result(trans(A B C) State) );
(11)  holds( clear(B) result(trans(A B C) State) );
(12)  preserves( trans(A B C) Statement ) <-
          diff( Statement on(A B) )
          diff( Statement clear(C) );
```

In the above, "diff(X Y)" is true if "X" and "Y" are not equal. Axiom (9) specifies the preconditions for the action. It says that "trans(A B C)" can be applied to "State" if the conditions in the body of (9) are true. After applying "trans(A B C)" to "State", certain statements will be "added" in the new state, they appear in axioms (10)-(11). The "deleted" statements in the new state are specified using axiom (12). Since the frame axiom is used to deduce what holds in a given state, nothing is actually added or deleted. It is merely deduced when necessary.

The *knowledge base* is defined to be the collection of all the axioms used in the state description models, the frame axiom and the action models. A synonym for knowledge base is *domain axiomatization*.

Negative knowledge, or false statements are encoded in a roundabout fashion in Kowalski's representation. The "not" axiom is never used to mean that a statement is false. For example, to specify the statements which are deleted by an action, "preserves" is used instead of

```
not( holds(Statement result(Action State)) );
```

or

```
holds( not(Statement) result(Action State) );
```

Madame makes the same closed-world assumption as SIPE [WIL83], that a negated statement is true unless the unnegated statement can be found or derived from the knowledge base. This impacts the construction of plans for negative goals, as explained in Chapter 4.

2

Madame and the UNIX Domain

2.1 Reasons for Choosing the UNIX Domain

Although Madame is domain independent, it was intended that Madame be used in the UNIX domain. One of the reasons for choosing this problem domain is that ISH required a planner. Another reason, is that the UNIX domain contains a large number of commands, which are difficult to encode regardless of the representation used. Thus, it is a non-trivial domain which makes it an ideal testbed for experimenting with new planning ideas. The new planning idea implemented in Madame is the use of multiple start states. This is possible through the use of a new data structure called a spider that is described in Chapter 3.

2.2 UNIX Queries

It is hoped that in the future, ISH will be able to accept most types of English queries. But at the moment, ISH's components restrict the range of acceptable queries. For example, Madame will only accept two types. The first are the queries that ask how the start state may be modified. A representative of this type is:

How do I delete the file "thesis" from my working directory?

The above query asks how to modify the start state so that an object in the start state (the file "thesis") no longer exists. UWISH would convert the above query into a goal and pass the following to Madame:

not(fexist(/u/anatrudel/thesis))

Madame's UWISH interface would pass the unaltered goal to the next component.

The second type of query are general questions about UNIX. This type commonly occurs with inexperienced users. An example is:

How do I delete a file?

For the above case, Madame expects UWISH to create a unique file name which does not already exist. Assume that the file name used is "xyz". The query is now interpreted as:

How do I delete the file "xyz"?

The information passed to the UWISH interface will be in two parts: a list of statements and a list of goals. The list of statements will be:

[holds(fexist(/u/anatrudel/xyz) 0)

holds(owner(anatrudel /u/anatrudel/xyz) 0)]

and the list of goals is:

[not(fexist(/u/anatrudel/xyz))]

The above means that if "/u/anatrudel/xyz" exists as a file in the start state and "anatrudel" owns this file, then build a plan to delete this file. The list of statements will be needed by the planning component of Madame; thus the list must be saved. The ideal location to store the list would be with the knowledge base. Since axioms cannot be added to a module while WUP is executing, the list of statements cannot be asserted with the knowledge base. Thus, the interface will assert the list of statements in Madame's auxiliary database. These statements will now be available to the planning component of Madame in addition to the axioms in the knowledge base. Since WUP will not allow modules to be modified during execution, the knowledge base will be in two places. After asserting the list of statements into the auxiliary database, the interface will pass the list of goals to the next component in Madame. Note that the list of goals may contain one or more goals and any number of these may be negated.

There are several types of queries not accepted by Madame. One of these is the "What happens if ..." type. In this type of query, a sequence of actions which represents a plan, is already given. For example:

What happens if I do an rm on "/u/anatrudel/thesis"?

On the other hand, a result may be included with the query:

Will "rm /u/anatrudel/thesis" delete this file?

In the above, along with the plan "rm /u/anatrudel/thesis", the result "does this plan delete the file" is included. This type of query asks if a certain result holds in the goal state after executing a given plan. The above two queries are best answered using a plan verifier, not a planner. A plan verifier would accept both a plan and results as input. The plan is executed and then the goal state produced is examined to see if the results hold in it.

Another type of query not handled by Madame is the "Why ..." type. For example:

Why has my file disappeared?

Answering the above query would require knowing the commands entered by the user during the session.

The plan produced by Madame and passed back to UWISH uses Kowalski's representation. For example, the plan produced for the query

How do I delete "/u/anatrudel/thesis"?

would be

result(rm(/u/anatrudel/thesis) 0)

When a plan is produced, the actions are applied from right to left starting with the start state. For example, the plan represented by the list of actions:

[action1 action2 action3]

is written in Kowalski's representation as:

result(action3 result(action2 result(action1 0)))

2.3 UNIX Knowledge Base

One of the major components of ISH is the UNIX axiomatization. The original axiomatization appears in [PAK83]; improvements were later done in [MAL83]. The axiomatization from [PAK83] and [MAL83] is called AHA. Further changes to AHA were required before the axiomatization could be used by Madame. The next few paragraphs outline those changes. For an example of the modifications made, Figure 2-1a shows the axiomatization of the "rm" command from AHA, and Figure 2-1b shows the modified version.

Figure 2-1a: AHA's Axiomatization of rm

```
possible( result( rm(Filename) S )) <-
        fullpath( Filename Pathname S )
        nonvar( Pathname )
        nonvar( S )
        holds( fexist(Pathname) S )
        not( holds( directory(Pathname) S ))
        rmperms( Pathname S )
        nonvar( S )
        possible( S );


preserves( rm(Filename) Rel S ) <-
        fullpath( Filename Pathname S )
        diff( Rel fexist(Pathname) )
        diff( Rel owner(_ Pathname) )
        diff( Rel perms(_ _ Pathname) )
        diff( Rel grpmember_file(Pathname _) )
        diff( Rel link(Pathname _) )
        diff( Rel link(_ Pathname) );
```

Renaming was a minor change that was required. For example, in Madame the start state is denoted by "0", as in Kowalski's representation. The notation used in AHA is "s_s" which stands for start state. Another change is that Madame uses the relation "poss" while AHA uses "possible". This second change can be noticed when comparing Figure 2-1a with Figure 2-1b.

A more substantial difference is in file name processing. A file in AHA is specified by taking any final portion of the full path name. For example, the file "/u/anatrudel/thesis" could be specified as any of the following:

Figure 2-1b: Madame's Axiomatization of rm

```
poss( result(rm(File) S )) <-
        poss( S )
        prouver( holds(fexist(File) S ))
        not( prouver( holds(directory(File) S )))
        rmperms( File S );


preserves( rm(File) Stat ) <-
        diff( Stat fexist(File) )
        diff( Stat owner(_ File) )
        diff( Stat perms(_ _ File) )
        diff( Stat grpmember_file(File _) )
        diff( Stat link(File _) )
        diff( Stat link(_ File) );
```

thesis
anatrudel/thesis
/u/anatrudel/thesis

This means that a file will have many valid names and the same name can represent many different files in different directories. This complicates the axiomatization unnecessarily. To resolve this problem, Madame only accepts full path names for files. Thus, each file will have a unique name. Also, by using this policy, Madame does not require the "fullpath" axiom used in AHA. This axiom computed the full path name of a file, given its partial path name. Another advantage of using full path names is that the "preserves" axiom has been changed so that it no longer has three arguments. In AHA, the current state was an argument to "preserves". This was required in order to be able to determine the full path names of files, which depended on the current directory. In Madame, the "preserves" axiom is in Kowalski's representation and so the current state is not an argument of it. It is obvious from comparing Figure 2-1a and Figure 2-1b that the use of the full path names has greatly simplified the "preserves" axiom. Since Madame only accepts full path names for files, the responsibility of guaranteeing full path names has been relegated to UWISH. UWISH will have to be modified so that it only accepts full path names or is given the ability to convert from partial to full path names.

As was mentioned in Section 2.2, the interface component of Madame sometimes asserts "holds" axioms in its auxiliary database. Whenever a "holds" term appears in the body of an axiom in the knowledge base, an alternative proof strategy called "prouver" is used with it. This is necessary because the "holds" can be either asserted in the knowledge base or the auxiliary database. Thus, "prouver" will check both locations. For example,

holds(X Y)

is written as:

prouver(holds(X Y))

Similarly with negated axioms, for example,

not(holds(X Y))

is written as:

not(prouver(holds(X Y)))

Examples of "prouver" can be seen in Figure 2-1b.

The "prouver" axiom indicates that its argument may appear in either the knowledge base or the auxiliary database. This result could also have been achieved by redefining "prove" to be:

```
prove( Module X ) <-
    prove( knowledge_base X );

prove( Module X ) <-
    prove( auxiliary_database X );
```

This modified version of "prove" first uses the knowledge base to prove "X". If this fails, the auxiliary database is then used to try to prove "X". The advantage of this form is that the "special" relation "prouver" would not have to be introduced. The disadvantage is that regardless of the axiom, "prove" will possibly check two locations. There are axioms which Madame knows can only exist in the auxiliary database (i.e. the axioms used to describe the spider), thus time would be wasted checking the knowledge base. Clearer code can be achieved by using "prouver" instead of redefining "prove". For example, if "prouver" is present as an axiom (i.e. "prove" is not redefined), and

prove(knowledge_base X)

appears in the body of an axiom, then the reader is assured that only the knowledge base will be used in trying to prove "X".

In addition to the changes in the axiomatization in AHA, three new relations have been defined: "only", "imposs" and "replace". The "only" relation takes one argument. It signifies that the only possibilities for its argument are what appear in the knowledge base. For example, assume that in the knowledge base, there are two "manip" and one "only" axiom:

```
only( manip(X) );
manip(a);
manip(b);
```

The axiom "manip" means that its argument, a block, can be moved (i.e. it is manipulatable). If Madame is passed the goal "manip(z)", first a check is done to see if the goal already holds. Since "manip(z)" is not asserted in the knowledge base, it does not hold. Next, Madame will check the knowledge base to see if an "only" relation exists for "manip". One does, so Madame immediately concludes that the goal "manip(z)" can never be satisfied. This conclusion is reached because "manip" appears in an "only" axiom and "manip(z)" does not appear in the knowledge base. Thus, Madame will fail, causing backtracking to a previous goal. Without the "only" axiom, Madame would try to modify the plan so that the goal "manip(z)" will hold. But since "manip(z)" can never be true, time will be wasted trying to modify the plan. Eventually, the planner would realise that no modifications can be made to the plan to make the goal hold. Thus, "only" is used to trap goals that can never hold and thus make the planner more efficient. "Only" is an implementation of the disproving planning strategy (presented in Chapter 1).

The second added axiom is "imposs", which comes from [WAR74]. The "imposs" axiom is used during planning in order to determine whether or not a set of goals is impossible. Thus,

"imposs" implements the disproving planning strategy for a set of goals. Presently, this axiom does not appear in the UNIX axiomatization though it is used in the blocks world. For example, in the blocks world, the following is used:

```
imposs( [on(X X)] );
```

The above says that it is impossible to have a block piled on itself.

The third added axiom is "replace". This axiom is best explained using the following example. The "poss" axiom for the UNIX command to create a directory (mkdir) is contained in Figure 2-2. The planning component of Madame will require the preconditions of an action. Using Figure 2-2, the preconditions for "mkdir" can be obtained; they are:

```
not( fexist(Path) )
parent( Path Parent )
fexist( Parent )
directory( Parent )
writeperms( Parent S )
```

The planning component of Madame uses the preconditions of an action as goals. Once these goals have all been satisfied, the action can be applied. The problem with the above list of preconditions is that "writeperms" is a goal which depends on the state. The planning component expects all goals to be state independent. This problem is resolved by using the "replace" axiom:

```
replace( writeperms(X Y) );
```

Whenever the "replace" axiom exists for a predicate and this predicate appears in the body of "poss", then special processing occurs when the list of preconditions is obtained. Continuing with the same example, the body of "writeperms" would replace "writeperms" in the list of preconditions for "mkdir". Therefore, "replace" causes an axiom to be replaced by its body whenever it is used in a list of preconditions.

Figure 2-2: Preconditions for mkdir

```
poss( result(mkdir(Path) S) ) <-
      poss( S )
      not( prouver(holds(fexist(Path) S)) )
      parent( Path Parent )
      prouver( holds(fexist(Parent) S) )
      prouver( holds(directory(Parent) S) )
      writeperms( Parent S );
```

An alternative solution to using "replace" is to put "writeperms" inside a "holds". For example, instead of having

```
writeperms( Parent S );
```

as the last statement in Figure 2-2, it could be replaced with

```
prouver( holds( writeperms(Parent) S ));
```

This was not done for two reasons. First, one goal when converting AHA to be used by Madame was to make as few changes as possible. This would allow Madame to use the bulk of the axioms in AHA with little or no modifications. Another goal, while converting the axiomatization, was that as few deviations as possible would be made from Kowalski's representation. Kowalski [KOW79] only uses the "holds" to describe the start state and the added statements of actions.

The UNIX axiomatization does not use a pure Kowalski representation since new axioms were added to the knowledge base. Two other deviations were made from Kowalski's representation. The first is that the "holds" axiom is allowed to have a body when used to represent an added statement. For example,

```
holds( owner(User Path) result(mkdir(Path) S) ) <-
    prouver( holds(current_user(User) S) );
```

is one of the add statements for "mkdir". The second deviation, involves negative knowledge. AHA freely allowed the use of negation. It was necessary to allow negations as it is used in the axiomatization of most of the actions. For example, one of the preconditions of deleting a file is that the file is "not" a directory. As a result of allowing negations in the knowledge base, the planning component of Madame had to be modified. The modifications made are discussed in Chapter 4.

A copy of the UNIX axiomatization used by Madame appears in Appendix A. Only a few UNIX commands appear in it as there are some unresolved problems. One of these problems is that most UNIX commands can be used in different situations. An example is the "mv" (move a file or directory) command. This command may be used with files or directories, and the object to where it is being moved, may or may not exist. To encode these variations, there would be more than one "poss" axiom for "mv" in the axiomatization. This also causes more than one "preserves" axiom to be present for "mv". Madame cannot handle these multiple "poss" and "preserves" axioms for the same command.

Figure 2-3: Preconditions for rm

```
poss( result(rm(File) S )) <-
    poss( S )
    prouver( holds(fexist(File) S ))
    not( prouver( holds(directory(File) S )))
    rmperms( File S );
```

Another problem with the axiomatization is that in UNIX, files and directories are considered to be almost the same. This similarity was encoded in the axiomatization. For example, assume "/u/anatrudel" is a directory and "/u/anatrudel/file" is a file, and both exist in the start state. The axioms used to describe them are:

```
holds( fexist(/u/anatrudel) 0 );
holds( fexist(/u/anatrudel/file) 0 );
```

Furthermore, in order to distinguish the directory from the file, the following is also asserted:

```
holds( directory(/u/anatrudel) 0 );
```

The "fexist" relation makes no distinctions between directories and files. But, although they are represented almost identically, different UNIX commands are used. For example, "rm" deletes a file while "rmdir" deletes a directory. This causes havoc with Madame. For example, when a directory is to be deleted, Madame will try "rm" first. Referring to Figure 2-3, the two first preconditions for "rm" are that the file exists and it isn't a directory. The first precondition is true but the second cannot be made true without violating the first (i.e. the directory would have to be deleted). Madame will backtrack to the first precondition to try to make the file exist by using the "mkdir" command. But, referring to Figure 2-2, a preconditon of "mkdir" is that the directory not exist. Therefore, Madame will try to use "rm" again to remove the directory. Madame is now in an infinite loop.

Due to the problems encountered in trying to axiomatize the UNIX world, it appears that changes to Kowalski's representation will be needed. These are presented in Chapter 5.

3

Spider: Data Structure for Multiple Start States

3.1 Introduction

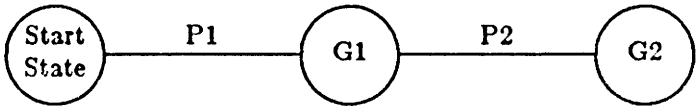
The planning problem involves finding a sequence of actions that transforms a unique start state into a goal state. A new approach to this problem would be to allow the planner to have many start states at its disposal. If the planner uses a forward search, it could pick a start state which would minimize the amount of work required. On the other hand, if the planner uses a backward search, it could terminate when one of the start states is reached. Therefore, many start states could be useful to a planner. Madame has been designed to work with many start states.

3.2 Acquisition and Retention of Multiple Start States

There are two major problems with having many start states: acquisition and retention. The first problem, acquisition, involves the question of how to obtain these extra start states. Retention involves storing the extra start states in such a way that they are easily accessible. Ease of addition and deletion of start states is also desirable.

When a plan is generated, it uniquely defines a goal state. This goal state may then be used as a state from which further planning can be pursued. That is, the plan used to reach this state might also serve as the initial portion of a larger plan. For example, in Figure 3-1, plan "P1" transforms the start state into the goal state "G1". Now, if a plan between the start state and a new goal state "G2" is required, it may be possible to use "G1" as the start state. If "G1" is used, "P2" is a plan which transforms "G1" into "G2". A plan between the start state and "G2" can now be easily built, by appending "P2" onto the end of "P1". So, "G1" was a goal state which was then used as a start state. Madame solves the acquisition problem by using the goal states from previously generated plans as shown in the above example.

Figure 3-1: Using G1 as the Start State



The retention problem has been further divided into three sub-problems: storage, addition, and deletion of start states. Madame solves the storage problem by using a *spider*. A spider is a directed tree with the original start state as its root. The arcs represent previously generated

plans and the nodes represent goal states associated with these plans. The tree can be of any depth. The spider is encoded using axioms; these will be described in Section 3.3. Every node in the spider is called a *p-s-s* (pseudo-start-state). The start state is also considered to be a p-s-s.

The following is the restatement of a previous example which used Figure 3-1. The example is now done with respect to the spider. Initially, the spider contains only the start state as in Figure 3-2a. Assume next that a goal state "G1" is passed to the planner and a plan "P1" is produced which transforms the start state into "G1". The spider would be updated as follows: an edge is added which represents the actions in plan "P1" and then a p-s-s representing "G1" is added. The updated spider is shown in Figure 3-2b. The next goal state passed to the planner is "G2". Assume, that a plan "P2" is built which transforms "G1" into "G2". The spider is then updated by adding "P2" as an edge and "G2" as a p-s-s. The final spider is shown in Figure 3-2c. Once again, a plan between the start state and "G2" can be easily built, by appending the actions in "P2" onto the end of the actions in "P1". This spider can now be written to a file to save it for future use. The next time Madame is used, the spider will contain three p-s-s's instead of only one.

Figure 3-2a: Initial Spider

Figure 3-2b: G1 is Added

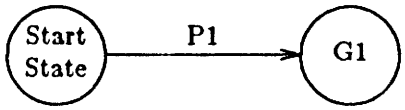
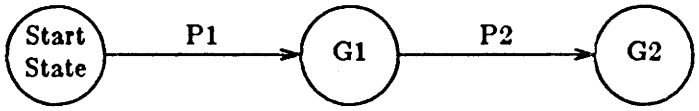


Figure 3-2c: Final Spider



Any p-s-s can be used as the start state. If a plan exists between one of the p-s-s's and a goal state, then a plan between the start state and that goal state can be easily constructed. For example, assume a plan has been found which transforms "p-s-s 4" into the "Goal State"; this is

Figure 3-3a: Plan between P-s-s 4 and the Goal State

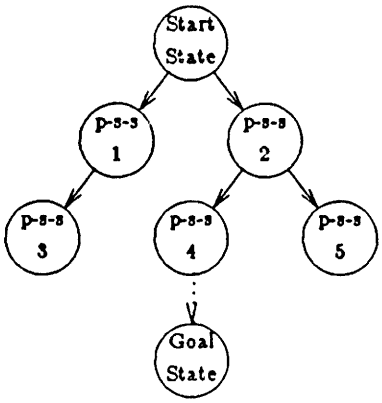
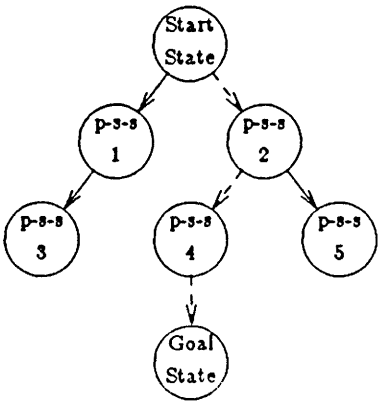


Figure 3-3b: Plan between Start State and Goal State



shown in Figure 3-3a. This "Goal State" with the newly generated plan is then added to the spider. A plan which transforms the "Start State" into the "Goal State" can now be easily constructed. It is built by appending the actions which are represented by the edges in the unique path between the "Start State" and the "Goal State". This path is represented by the dashed lines in Figure 3-3b.

Whenever Madame is passed a set of goals describing a goal state, the spider is checked to see if the goal state corresponds to one of the p-s-s's. If it does, then no planning is required. The required plan is built by concatenating the sub-plans represented by the edges in the unique path between the start state and the p-s-s corresponding to the goal state. This ensures that the spider will retain the properties of a tree. For example, assume we have a spider represented by Figure 3-4a and the goal state is equal to "p-s-s 3". If a check was not made to see if the goal state equals one of the p-s-s's, then "p-s-s 1" could be used for planning. An edge linking "p-s-s 1" to

Figure 3-4a: Initial Spider

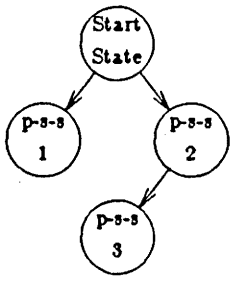
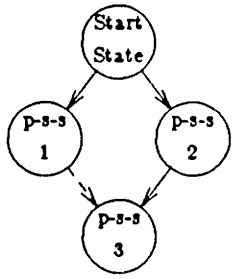


Figure 3-4b: Illegal Spider



"p-s-s 3" would then be added to the spider. The resulting spider (which is not a tree) is shown in Figure 3-4b. Therefore, the initial check to see if the goal state equals one of the p-s-s's, guarantees that when a new p-s-s is added to the spider, it will have exactly one parent.

The deletion of p-s-s's is not done by Madame. Although they can easily be deleted, the real problem is deciding which p-s-s to delete. A heuristic has not yet been developed for deciding which is the best candidate for deletion. As more p-s-s's are added, the spider will become very large. Thus, the time required to traverse, store and maintain the spider will eliminate its beneficial properties, making it a burden rather than an aid to the planner. The removal of p-s-s's could be used to control its growth. Possibly, heuristics such as Least Recently Used or FIFO, could be used for p-s-s removal.

3.3 Spider Encoding

As previously mentioned, the spider is encoded using axioms. The axioms can be divided into three categories, depending on what they describe. The three categories are: edges, p-s-s's and the spider in general.

3.3.1 Axiom Used to Describe Edges

Figure 3-5 shows a spider that contains two p-s-s's and one edge. The edge represents the plan which transforms "p-s-s 0" into "p-s-s 1". The axiom used to describe edges is "spi_plan". The general form of "spi_plan" is:

spi_plan(List N Pss)

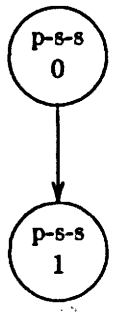
where "List" is a list of actions which makes up a plan, "N" is the number of actions in "List", and "Pss" is the unique goal state achieved as a result of executing the plan contained in "List". In Figure 3-5, "List" would contain the plan which transforms "p-s-s 0" into "p-s-s 1" and "Pss" would be bound to "1", which represents "p-s-s 1".

Let "p-s-s 0" in Figure 3-5 be the start state, and be represented by Figure 3-6a. Also, let "p-s-s 1" in Figure 3-5 be represented by Figure 3-6b. The actions needed to transform "p-s-s 0" into "p-s-s 1" are move block "a" onto "q", then move block "b" on top of "a". Thus, the axiom associated with the edge in Figure 3-5 is:

spi_plan([trans(a b q) trans(b c a)] 2 1)

Figure 3-7 shows the spider from Figure 3-5 with the spi_plan axiom added.

Figure 3-5: Spider



3.3.2 Axioms Used to Describe P-s-s's

Every p-s-s in the spider is identified by a unique non-negative integer. The axiom used to encode this information is:

spi_poss(N)

Every p-s-s will have a spi_poss axiom associated with it and "N" will be bound to its identifying integer. To verify if a certain p-s-s exists, a check to see if a spi_poss axiom exists for it is sufficient. The integer zero is used to represent the start state. The other p-s-s's are assigned integers in increasing order. Continuing the example of Figure 3-5 from Section 3.3.1, "p-s-s 0" is identified by

spi_poss(0)

Figure 3-6a: P-s-s 0

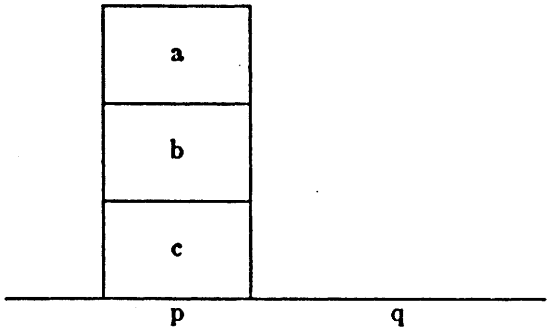
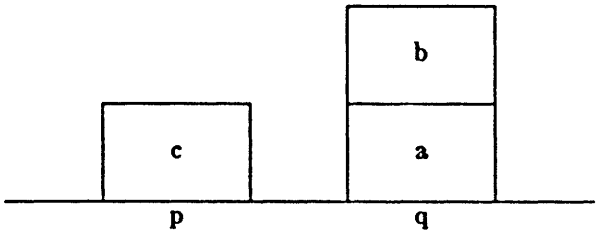


Figure 3-6b: P-s-s 1



and "p-s-s 1" by

spi_poss(1)

Since the spider is a directed tree, every p-s-s except for the start state will have exactly one parent. The parent of a p-s-s is specified using:

spi_parent(Parent Pss)

"Parent" and "Pss" will be bound to integers representing p-s-s's. There is one spi_parent axiom associated with each p-s-s except for the start state, since it does not have a parent. Again, using the example from Figure 3-5, "p-s-s 0", the start state, does not require a spi_parent axiom. The spi_parent axiom used for "p-s-s 1" is:

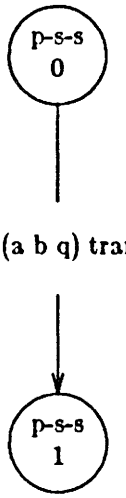
spi_parent(0 1)

The children of a p-s-s are specified using:

spi_children(List Pss)

where "List" is a list of integers. Each integer in this list represents a p-s-s which is a child of

Figure 3-7: Spider with spi_plan



"Pss". If a p-s-s has no children, then "List" will be equal to the empty list (i.e. [ ]). There is one spi_children axiom associated with each p-s-s including the start state. Although the spider is a directed tree, the axioms spi_parent and spi_children allow us to traverse the spider in either direction. Continuing with the example from Figure 3-5, the spi_children axiom for "p-s-s 0" is

spi_children([1] 0)

and the spi_children axiom for "p-s-s 1" is

spi_children([ ] 1)

Figure 3-8 shows the spider from Figure 3-5 with the axioms that have been presented.

When an action is applied to a state to produce a new state, most of the statements which held in the old state will still hold in the new state. Those which do not hold in the new state are said to be deleted. Also, there are statements which did not hold in the old state but now hold in the new state; these are called added statements. The application of a single action to a state can be generalized to a sequence of actions; this is a plan.

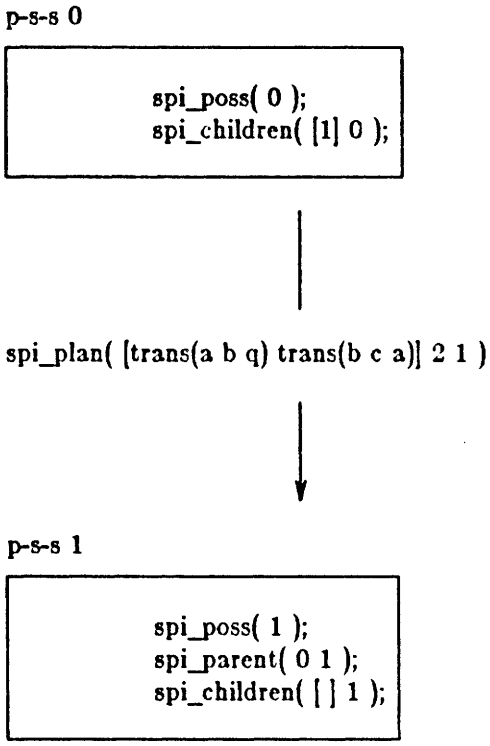
In the spider, a directed edge between any two p-s-s's represents a plan. This plan is made up of a sequence of actions. The added and deleted statements corresponding to an edge, are stored in the p-s-s which is at the head of the edge. The axioms used to describe the added and deleted statements will be presented in the remainder of this sub-section. The "spider frame axiom", which determines what held in the start state and what still holds in the goal state, will be presented in Section 3.3.3.

In every p-s-s, except for the start state, there will be some statements which are added by the actions in the plan; these added statements are specified using:

spi_holds(Statement Pss)

There will be one spi_holds axiom for every "Statement" which is added in the p-s-s represented by "Pss".

Figure 3-8: Spider with some P-s-s Axioms



The method used to obtain the added statements in a p-s-s is to look at the add list of each action in the p-s-s's spi_plan axiom and check to see if some statements in this add list are deleted by actions which follow in the plan. Only statements which are not deleted by future actions will appear in a spi_holds axiom. Figure 3-9 contains a pseudo-code description of the algorithm used for calculating the spi_holds axioms of a p-s-s.

As an example of how added statements are calculated, the algorithm in Figure 3-9 will be applied to the example from Figure 3-5. In this case, the spi_holds axiom will be generated for "p-s-s 1" so,

$$pss = 1$$

Step 1 of the algorithm says to get the plan from the spi_plan axiom:

$$Plan = [trans(a\ b\ q)\ trans(b\ c\ a)]$$

In step 2, "Plan" is not empty so:

Figure 3-9: Algorithm for spi_holds

Algorithm, written in pseudo-code, for calculating all the spi_holds axioms for the p-s-s: pss.

1. Get the spi_plan axiom for "pss" by executing the following query in Prolog:

   spi_plan(Plan N pss)

   "Plan" contains a list of actions.

2. % Examine each action in "Plan".
   If Plan is empty then
   % No more actions to look at.
   Terminate
   else
   % Look at the next action.
   Action = the first action in Plan
   Plan = Plan minus its first action
   endif

3. Get the add list for "Action" and put it in "Addlist".

4. % Examine each statement in "Addlist".
   While Addlist is not empty do
   Candidate = first statement in Addlist
   Addlist = Addlist minus its first statement
   If Candidate is not deleted by any actions in Plan then
   % "Candidate" is an added statement.
   Assert the following in the knowledge base:
   spi_holds(Candidate pss)
   endif
   endwhile

5. Go to Step 2.

Action = trans(a b q)
Plan = [trans(b c a)]

Next, in step 3, the add list for "Action" is calculated using the knowledge base:

Addlist = [on(a q) clear(b)]

The first iteration of the while loop of step 4 results in:

Candidate = on(a q)
Addlist = [clear(b)]

Since "Candidate" is not deleted by the action in "Plan", the following is asserted in the knowledge base:

spi_holds(on(a q) 1)

The second iteration of the while loop of step 4 results in:

Candidate = clear(b)
Addlist = [ ]

Once again, "Candidate" is not deleted by the action in "Plan", the following is asserted in the knowledge base:

spi_holds(clear(b) 1)

"Addlist" is now empty, and step 4 terminates. Execution returns to step 2. "Plan" still contains one action, so:

Action = trans(b c a)
Plan = [ ]

In step 3, the add list for "Action" is calculated:

Addlist = [on(b a) clear(c)]

Since "Plan" is empty, step 4 will assert the following in the knowledge base:

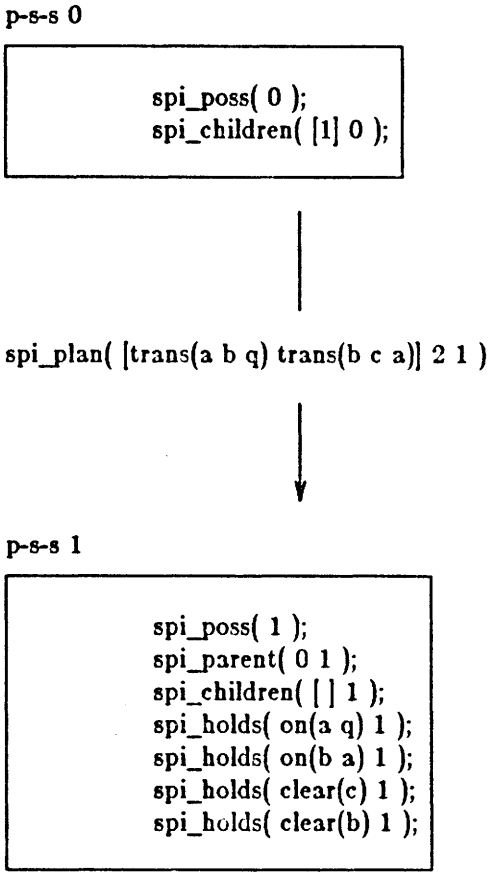
spi_holds(on(b a) 1)
spi_holds(clear(c) 1)

Next, step 5 returns execution to step 2. "Plan" is now empty and the algorithm terminates. The result of applying the algorithm is that four spi_holds axioms have been asserted in the knowledge base. The spider from Figure 3-8 is shown in Figure 3-10; it has been updated with the spi_holds axioms.

Usually, spi_holds specifies a statement which is true in the p-s-s and which is not true in the parent of p-s-s. This is true except when the plan undoes a statement and then redoes it. For example, assume that in a blocks world, on(a b) is true in the parent of a p-s-s. If the plan removes block "a" from block "b" and then later puts block "a" back onto block "b", then on(a b) will be an added statement in the p-s-s. The statement on(a b) is an added statement even though it was true in the parent because it is added by an action which appears in the plan.

The spi_holds axiom for the start state is written in a slightly different manner. It appears as:

Figure 3-10: Spider with spi_holds




```
spi_holds( Statement 0 ) <-
    prouver( holds(Statement 0) );
```

The special spi_holds axiom for the start state acts as an interface between the spider and the knowledge base. It would have been possible to code all the "holds" axioms for the start state as spi_holds axioms. But, this was not done in order to keep the spider and knowledge base separate. This is desirable in order to allow the planner to be run with or without the spider present. This in turn, allows the efficiency of the planner to be compared when it is and is not using the spider.

Actions can delete statements as well as add them. For example, when a block is unstacked (i.e. the action trans(a b c)), on(a b) is no longer true. The statements which are deleted by a p-s-s's plan are specified using:

```
spi_preserves( Statement pss ) <-
    diff( Statement statement1 )
        .
        .
    diff( Statement statementn );
```

Thus, "Statement" is true in the p-s-s represented by "pss" if it is true in the parent of "pss" and it is different from the "n" statements "statement1" through "statementn". The spi_preserves axiom is used by the "spider frame axiom" which is presented in Section 3.3.3. There is one spi_preserves axiom for each p-s-s except the start state.

"statement1" through "statementn" are the deleted statements for all the actions in spi_plan, less the statements which are re-added by any action which follows in the plan. If a statement is deleted and then later re-added by another action, this statement will appear in a spi_holds axiom. Figure 3-11 contains a pseudo-code description of the algorithm used to build the spi_preserves axiom for each p-s-s.

As an example of how deleted statements are calculated, the algorithm in Figure 3-11 will be applied to the example from Figure 3-5. In this case, the spi_preserves axiom will be generated for "p-s-s 1" so,

$$pss = 1$$

Step 1 of the algorithm says to get the plan from the spi_plan axiom:

$$Plan = [trans(a\ b\ q)\ trans(b\ c\ a)]$$

Also, "List" is initialized:

$$List = [\,]$$

In step 2, the following is done:

Figure 3-11: Algorithm for spi_preserves

Algorithm written in pseudo_code for calculating the spi_preserves axiom for the p-s-s: pss.

1. Get the spi_plan axiom for "pss" by executing the following query in Prolog:

   spi_plan(Plan N pss)

   "Plan" contains a list of actions. Initialize "List" to the empty list:

   List = [ ]

2. % Process the actions in "Plan" one at a time.
   Action = the first action in Plan
   Plan = Plan minus its first action

3. Get the delete list for "Action" and put it in "Deletelist".

4. % Examine each statement in "Deletelist".
   While Deletelist is not empty do
   Candidate = first statement in Deletelist
   Deletelist = Deletelist minus its first statement
   If Candidate is not added by any actions in Plan then
   Add Candidate to List
   endif
   endwhile

5. If Plan is not empty then
   % More work to be done
   Go to step 2
   endif

6. % At this point, "List" contains the deleted statements
   % "statement1" thru "statementn".
   Assert the following in the knowledge base:

   spi_preserves(Statement pss) <-
   diff(Statement statement1)
   .
   .
   diff(Statement statementn)

7. Terminate

Action = trans(a b q)
Plan = [trans(b c a)]

Next in step 3, the delete list for "Action" is calculated using the knowledge base:

Deletelist = [on(a b) clear(q)]

The first iteration of the while loop in step 4 results in:

Candidate = on(a b)
Deletelist = [clear(q)]

Since "Candidate" is not added by the action in "Plan", "Candidate" is added to "List":

List = [on(a b)]

The second iteration of the while loop in step 4 results in:

Candidate = clear(q)
Deletelist = [ ]

Once again, "Candidate" is not added by the action in "Plan", so it is added to "List":

List = [clear(q) on(a b)]

"Deletelist" is now empty and step 4 terminates. In step 5, "Plan" is not empty so execution returns to step 2. The result of executing step 2 is that:

Action = trans(b c a)
Plan = [ ]

In step 3, the delete list for "Action" is calculated:

Deletelist = [on(b c) clear(a)]

Since "Plan" is empty, step 4 will add "Deletelist" to "List":

List = [clear(a) on(b c) clear(q) on(a b)]

Next, execution continues at step 6 because "Plan" is empty. Step 6 will assert the following in the knowledge base:

```
spi_preserves( Statement 1 ) <-
      diff( Statement clear(a) )
      diff( Statement on(b c) )
      diff( Statement clear(q) )
      diff( Statement on(a b) )
```

Execution is then terminated by step 7. The spider from Figure 3-10 is shown in Figure 3-12; it has been updated with the spi_preserves axiom.

3.3.3 Axioms Used to Describe the Spider in General

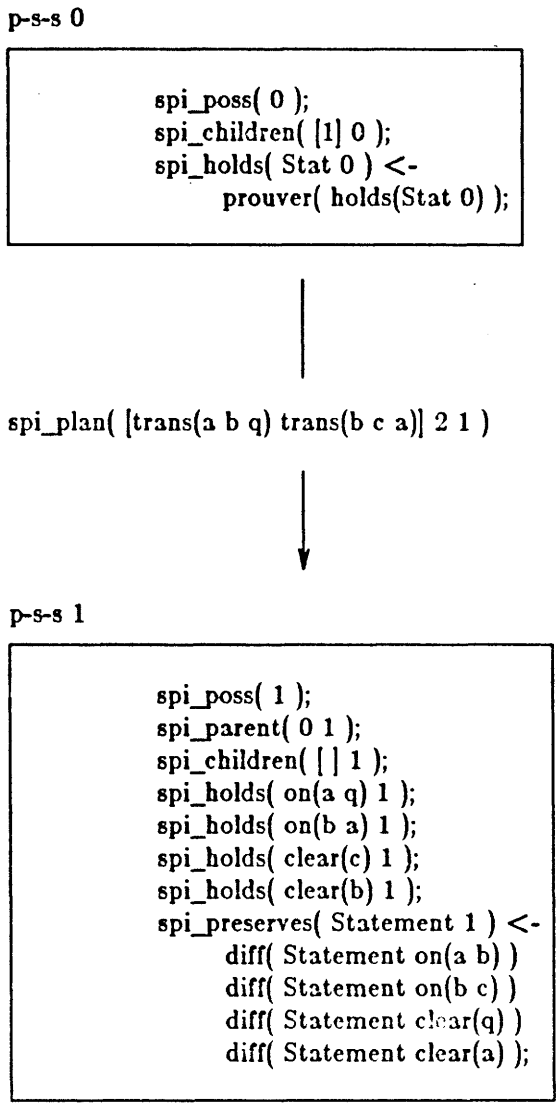
There are two global axioms which are used to describe the spider in general. The first is:

spi_no(Number)

The next integer available to be assigned to a new p-s-s is stored in "Number". Also, since the numbering of p-s-s's begins at zero, "Number" is a count of the p-s-s's in the spider.

The second global axiom is a special case of the spi_holds axiom, it is:

Figure 3-12: Spider with spi_preserves




```
spi_holds( Statement Pss ) <-
    spi_parent( Parent Pss )
    spi_holds( Statement Parent )
    spi_preserves( Statement Pss );
```

The above axiom is the "spider frame axiom". "Statement" is true in "Pss", if it is true in the parent of "Pss" and it has not been deleted by any actions in the plan between the parent of "Pss" and "Pss".

All the spider axioms have now been presented. Figure 3-13 shows a complete axiomatization of the example from Figure 3-5 which would be used by Madame.

Figure 3-13: Axiomatization Used by Madame

```
% spider frame axiom
spi_holds( Statement Cpss ) <-
      spi_parent( Ppss Cpss )
      spi_holds( Statement Ppss )
      spi_preserves( Statement Cpss );

% "spi_no" specifies how many p-s-s's there are.
spi_no(2);

% axioms for the start-state (p-s-s 0):
% {on(a b) on(b c) on(c p) clear(a) clear(q)}
spi_poss( 0 );
spi_children( [1] 0 );
spi_plan( [ ] 0 0 );
spi_holds( Stat 0 ) <-
      prouver( holds(Stat 0) );

% axioms for p-s-s 1:
% {on(c p) on(b a) on(a q) clear(c) clear(b)}
spi_poss( 1 );
spi_parent( 0 1 );
spi_children( [ ] 1 );
spi_plan( [trans(a b q) trans(b c a)] 2 1 );
spi_holds( on(a q) 1 );
spi_holds( on(b a) 1 );
spi_holds( clear(c) 1 );
spi_holds( clear(b) 1 );
spi_preserves( Statement 1 ) <-
      diff( Statement on(a b) )
      diff( Statement on(b c) )
      diff( Statement clear(q) )
      diff( Statement clear(a) );
```

The axioms used to encode the spider are very similar to Kowalski's representation. For example, the equivalent of spi_holds is holds, the equivalent of spi_preserves is preserves, and the spider frame axiom is very similar to Kowalski's frame axiom. Kowalski's representation is used for representing states of a world, which is also what the spider does. Therefore, it is natural to extend Kowalski's representation so that it describes the spider.

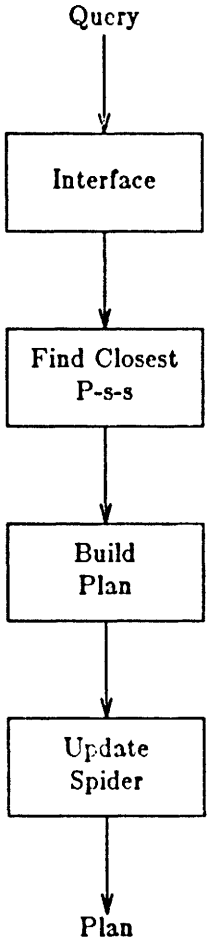
3.4 How the Spider is Used by Madame

Before Madame will accept a query, the spider is read into Madame's auxiliary database, where it can be modified. After Madame is finished answering queries, the spider is written to a file. Thus, all the changes made to the spider have been saved for the next session.

Figure 3-14 shows Madame's major components. First, the query is passed to the interface which will do some pre-processing on the query before passing it to the next component. Recall that the interface was discussed in Section 2.2. The next component will find the p-s-s which is

closest to the goal state described by the query. The method used to find the closest p-s-s will be discussed in Section 3.5. Next, the query and the closest p-s-s is passed to the planning component. The planning component, which is discussed in Chapter 4, will extend the plan between the start state and the closest p-s-s so that it reaches the goal state. The spider is then updated with the newly generated plan. This last component will be described in Section 3.7. After the spider has been updated, the plan is returned to UWISH. UWISH will then pass the plan to the user. The user now has a plan which transforms the start state into the required goal state. The user is never aware of the p-s-s's or the spider. All the plans passed back to the user are relative to the start state.

Figure 3-14: Madame's Components



3.5 Finding the Closest P-s-s

One of the p-s-s's in the spider must be chosen to be used by the planning component. Ideally, the p-s-s which is closest to the goal state should be chosen. By choosing the closest p-s-s, the planning phase is made simpler.

In order to find the closest p-s-s, Madame must examine each p-s-s in the spider. A depth-first traversal of the spider (which is a directed tree) is done. It would be possible to use any type of tree traversal algorithm, as long as it is guaranteed to visit every node in the spider.

As each p-s-s in the spider is visited, its distance from the goal state is calculated. The goal state is described by a list of goals "LG". The distance between a p-s-s and the goal state is equal to the number of goals in "LG" which do not hold in the p-s-s. To check if a goal "G" holds in a p-s-s "N", the following Prolog query is executed:

spi_holds(G N)

If the query above succeeds, then "G" holds in p-s-s "N". If a p-s-s is very close to the goal state (i.e. a short distance away), a large number of goals in "LG" will hold in the p-s-s. With the closest p-s-s, the planner will only have to work on the goals which do not hold in the p-s-s. For example, consider the list "LG" of goals to be:

[on(c b) on(b a)]

Figure 3-15a represents a p-s-s which is at a distance of 2 from the goal state. Figure 3-15b represents a p-s-s which is at a distance of 1 from the goal state. Thus, between the two, the p-s-s which corresponds to Figure 3-15b is closer to the goal state.

Figure 3-15a: Distance of 2 from the Goal State

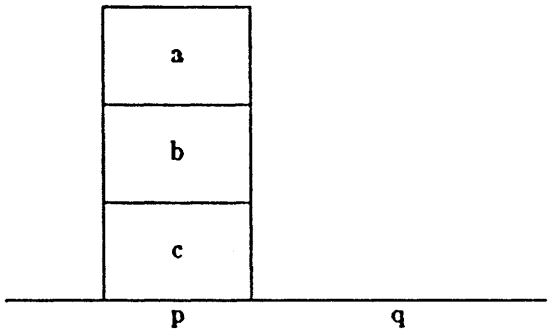
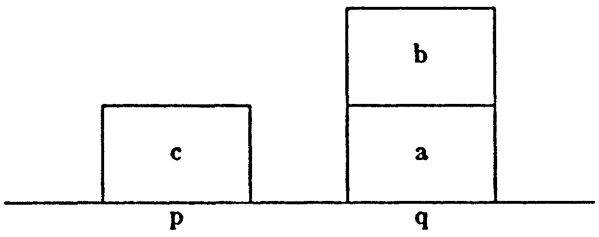


Figure 3-15b: Distance of 1 from the Goal State



A temporary axiom is placed in Madame's auxiliary database during the traversal of the spider to keep track of the closest p-s-s found to date. The axiom used is:

closest(ClosestPss Dist GoalsLeft)

where "ClosestPss" is bound to the integer representing the closest p-s-s found to date, "Dist" (an integer) is the distance between "ClosestPss" and the goal state, and the goals which do not hold in "ClosestPss" are contained in "GoalsLeft" (a list containing "Dist" elements). For example, if when examining p-s-s 1 (represented by Figure 3-15b), it is found to be the closest p-s-s to date to the set of goals:

[on(c b) on(b a)]

then the "closest" axiom would be updated to:

closest(1 1 [on(c b)])

Once the distance between a p-s-s and the goal state has been calculated, three outcomes are possible: the p-s-s is not the closest so far, the p-s-s is the closest so far, or the p-s-s is tied for closest. If the p-s-s is not the closest so far, the next p-s-s is examined. On the other hand, if the p-s-s is the closest one found so far, the "closest" axiom is updated to record this newly found closest p-s-s. If by chance, the distance between the p-s-s and the goal state is zero (i.e. all the goals hold in the p-s-s), then the p-s-s is equal to the goal state. Traversal of the spider is terminated because it is impossible to find a closer p-s-s. If the p-s-s does not equal the goal state, the spider traversal is continued.

If a tie occurs between two p-s-s's, Madame will pick the one which is closest to p-s-s 0 (the start state). The closest to p-s-s 0 will be the p-s-s with the least number of actions in the plan between itself and p-s-s 0. This choice of p-s-s helps to minimize the eventual plan between p-s-s 0 and the goal state.

It is possible that after traversing the spider, none of the goals in "LG" held in any of the p-s-s's. This results in a global tie, all the p-s-s's are equidistant from the goal state. The same rule that was used for breaking a tie between two p-s-s's is used; that is, the closest p-s-s to p-s-s 0 is chosen. This is a trivial problem, as the closest p-s-s is p-s-s 0. In this case, the spider does not aid the planner.

After the traversal of the spider is finished and the closest p-s-s to the goal state has been chosen, the "closest" axiom is removed from Madame's auxiliary database. The integer representing the closest p-s-s found and the goals which do not hold in this p-s-s are then passed on to the planning component of Madame.

3.6 A Problem with the Method Used for Finding the Closest P-s-s

The method used by Madame to find the closest p-s-s does not always succeed. Madame will sometimes pick a p-s-s which is not the closest. For example, let p-s-s 0 be represented by Figure 3-16a and p-s-s 1 be represented by Figure 3-16b. If the goal is "on(c a)", then both p-s-s's are at a distance of 1 from the goal state. This results in a tie, it is broken by choosing p-s-s 0 because the distance between p-s-s 0 and itself is zero. Thus, Madame will choose p-s-s 0 as the closest p-s-s. Madame has made the wrong choice because p-s-s 1 is actually the closest. The number of blocks which have to be moved to transform p-s-s 0 into the goal state is three while only one move is needed for p-s-s 1.

Although the method used by Madame does not always find the correct closest p-s-s, it was chosen for its simplicity and effectiveness. In many cases, the correct, closest p-s-s will be chosen. Tests were run which showed that the spider, and the way it is searched, made the planner more efficient. These results will be presented in Chapter 5.

Figure 3-16a: P-s-s 0

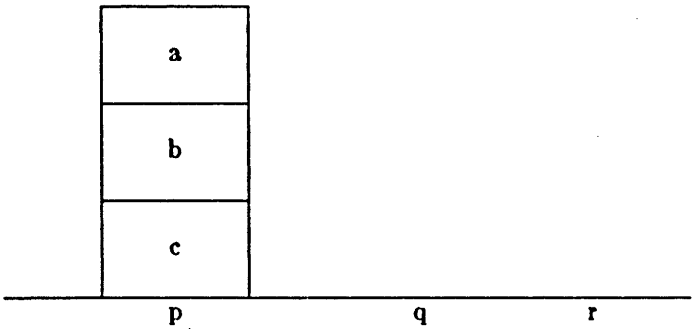
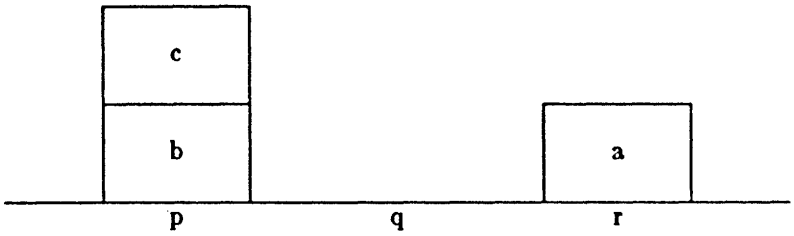


Figure 3-16b: P-s-s 1



3.7 Updating the Spider

After the planning component has extended the plan between the start state and the closest p-s-s so that it reaches the goal state, the goal state is added to the spider. There are two possible parents for the newly added p-s-s; the closest p-s-s to the goal state, or the start state. The planning component will sometimes insert actions in the plan which transforms the start state into the closest p-s-s; the reason this is done will be explained in Chapter 4. If actions were inserted, then the start state will be the parent. Otherwise, the closest p-s-s will be the parent of the newly added p-s-s.

Figure 3-17a shows the spider before the goal state is added to it. If the closest p-s-s were used as the parent, the spider would look like the one in Figure 3-17b after the update. On the other hand, if while building the plan between the start state and the goal state, actions were inserted in the plan represented by "Edge A", the spider would look like the one in Figure 3-17c after the update.

All newly generated plans are added to the spider. Currently, there is no criteria implemented for retiring or rejecting p-s-s's. A possible criteria would be to only accept a new p-s-s if the plan represented by the edge between the new p-s-s and its parent contains "n" or more actions. This would represent accepting new p-s-s's for which a lot of work had been expended to generate a plan. This method for p-s-s acceptance was not implemented as it was beyond the scope of this thesis.

Figure 3-17a: Before Update

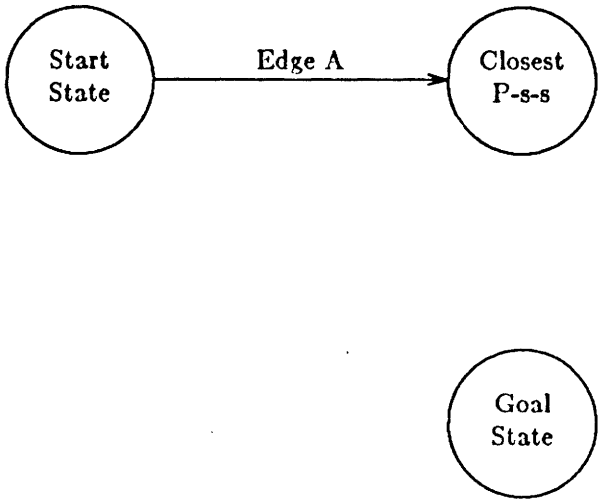
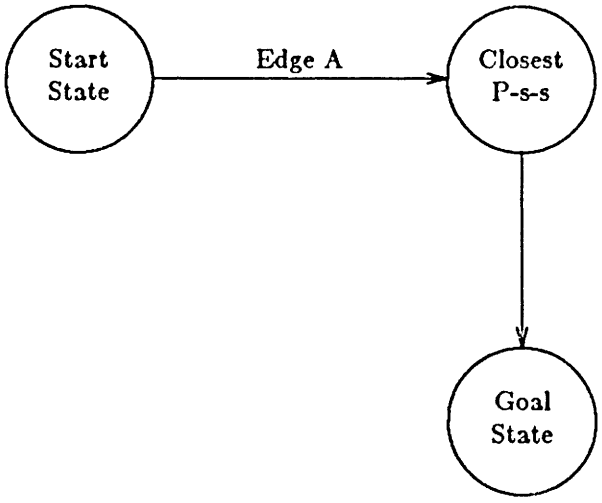


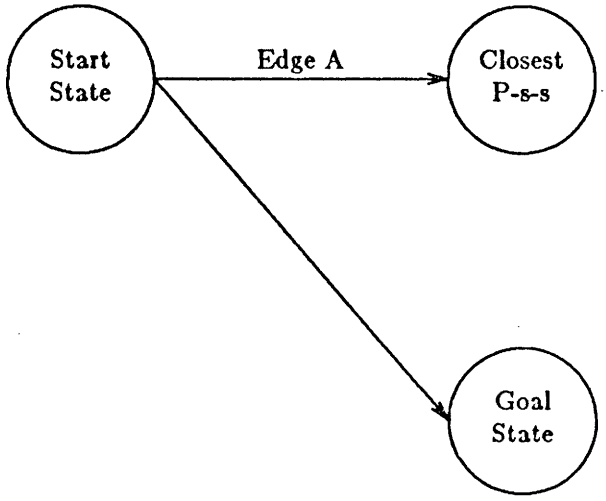
Figure 3-17b: Closest P-s-s is the Parent



3.8 Observations about the Spider

Madame sometimes produces a plan which is not optimal. For example, assume the spider contains two p-s-s's (p-s-s 0 and p-s-s 1) and the edge linking the p-s-s's represents a plan containing one hundred actions. Further assume that the list of goals "LG" passed to Madame hold in p-s-s 1. Madame will therefore return the plan linking p-s-s 0, the start state, with p-s-s 1. But, it

Figure 3-17c: Start State is the Parent



is possible, that an optimal plan could have been constructed by applying fewer actions to the start state. This optimal plan would not have been considered by Madame; plans that are found in the spider will be used before any other planning is done.

The spider obtains new p-s-s's from plans which have been generated. Since Madame uses the spider, it generates new plans from old plans. This means that as the user asks for plans, they are recorded in the spider. Every user will have his own spider. Therefore, the spider models in a rough way, what the user already knows about a problem domain. Also, the generated plans are in the user's "language"; that is, parts of the plan will already be familiar to him.

As the user asks for plans, the spider will grow larger because new p-s-s's will be added. This means that the spider will "know" more about the problem domain and be more helpful in planning. Therefore, the spider learns about the problem domain along with the user.

4

An Implementation Based on Warplan

4.1 Warplan

After the closest p-s-s to the goal state has been found, a plan linking them must be built. Any type of planner could be used for this task; Madame uses a WUP version of Warren's [WAR74] Warplan. This planner was chosen because it is coded in Prolog and uses a problem domain description similar to Kowalski's representation.

Warren's [WAR74] Warplan was modified by Kowalski and van Emden to run under WUP; this modified version will henceforth be referred to as Warplan. Warplan is a goal directed planner; that is, it uses a backward search. When trying to satisfy a goal, Warplan will first try extending the plan developed so far and if that fails, goal regression is used. The top level Prolog specification of Warplan is given in Figure 4-1. A complete listing of Warplan appears in Appendix B.

A request to produce a plan has the general form

plan(Plan LG)

where "Plan" is a free variable and "LG" is bound to a list of goals which describes the desired goal state.

When Warplan receives a request, "plan" will pass "LG" to the axiom "empower". "Empower" will separate the list "LG" into individual goals; "LG" is decomposed from left to right. Each goal in "LG" is passed to the axiom "empower1". "Empower1" will modify the plan built so far so that the goal passed to it holds. After the goal has been satisfied, "empower" will add the goal to the protected goal list. This list will also be referred to as the "list of goals already accomplished" since the list will only contain accomplished goals. The protected goal list guarantees that no future modifications to the plan will produce a state in which the goal does not hold. Before the plan being worked on is modified, a check is always made to see if the plan would delete any of the protected goals. The modification is not allowed if one or more of the goals in the protected goal list are deleted. In summary, "empower" decomposes a list of goals while "empower1" works on the plan to satisfy each goal one at a time.

"Empower1" has three methods at its disposal for making a goal hold. The first method simply determines if the goal already holds. The second method is empowerment by extension, and the third is empowerment by insertion. Empowerment by extension satisfies a goal by adding an action onto the end of the plan. Empowerment by insertion uses goal regression to satisfy a goal. The goal is regressed over the last action and planning continues with the shorter plan. Goal regression is called "empowerment by insertion" because the result of regressing a goal is the insertion of actions in the plan.

Given a goal "G" which does not hold, empowerment by extension will first find an action that adds "G". A check is then made to see if that action deletes any of the goals in the protected goal list. If none are deleted, the preconditions for the action are then obtained. These preconditions must hold before the action can be applied. "Empower" is called recursively with

Figure 4-1: Top Level of Warplan

```
plan( P G ) <- empower( [] [] P G );


% empower( P1 G1 P2 G2 )
%
%  G1:  list of goals achieved so far
%  P1:  plan to achieve G1
%  G2:  list of goals left to achieve
%  P2:  this is the plan we are looking for;
%                                          it will solve all the goals

empower( P G P [] );
empower( P G P1 [X|G1] ) <-
        empower1( P G P2 X )
        empower( P2 [X|G] P1 G1 );



% return P because X holds in state produced by P

empower1( P G P X ) <-
        holds( X P );

% empowerment by extension

empower1( P G :(P1 A) X ) <-
        addlist( A L )
        member( X L )
        preserves( A G )
        precond( A G1 )
        not ( inconsistent( G1 G ) )
        empower( P G P1 G1 )
        preserves( A G );

% empowerment by insertion

empower1( :(P0 P1) G :(P2 P1) X ) <-
        not( deletes( P1 X ) )
        addlist( P1 G1 )
        precond( P1 G2 )
        minus( G G1 G3 )
        union( G2 G3 G4 )
        empower1( P0 G4 P2 X )
        not( deletes( P1 X ) );
```

the plan built so far, the list of protected goals, and the list of preconditions which must be satisfied. "Empower" will modify the plan to ensure that all the preconditions of the action hold. This action is then appended to the modified plan. The goal "G" will now hold when the

modified plan is executed.

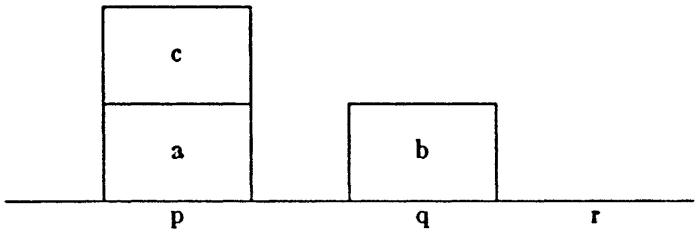
If empowerment by extension fails, empowerment by insertion will be attempted. The given goal "G" is regressed over the last action in the current plan; the regression will not be done if the last action deletes "G". Regression is done as follows. First, the last action in the current plan is examined; if any assertions from the addlist of this action appear in the protected goal list, they are removed. When the last action is re-executed, the removed assertions will automatically hold. Any preconditions of the last action which do not appear in the protected goal list are then added to it. This guarantees that if actions are inserted into the plan, the last action could still be applied because all its preconditions will hold. Then, "empower1" is called recursively. The parameters passed to "empower1" will be the plan built so far *less the last action,* the modified list of protected goals, and the same goal "G". The plan returned by the recursive call to "empower1" will satisfy the goal "G". The last action is then appended onto the end of the returned plan.

The remainder of this section contains an example of how Warplan works. Let "LG" be the list of goals passed to Warplan, "Plan" contains the plan built to-date, "PG" is the list of protected goals, and "Goal" is the current goal being attempted by "empower1". For this example, these variables will be initialized as follows:

LG = [on(a b) on(b c)]
Plan = 0
PG = [ ]
Goal is unassigned

Figure 4-2a depicts the start state in relation to the blocks world.

Figure 4-2a: Start State



There is only one action allowed, trans(X Y Z). If block "X" is resting on block or position "Y", trans(X Y Z) will remove block "X" from block or position "Y" and place it on block or position "Z".
The add list for trans(X Y Z) is:

clear(Y)
on(X Z)

its delete list is:

clear(Z)
on(X Y)

and its list of preconditions is:

on(X Y)
clear(X)
clear(Z)
diff(X Z)

The example will follow Warplan through its recursive calls. Level "X" will indicate that work is being done in the X'th level of recursion.

Level 1:

Warplan will pass "LG" to "empower" which will in turn remove the first goal from "LG", and pass the following to "empower1":

Plan = 0
PG = [ ]
Goal = on(a b)

"Empower1" will first check if "Goal" already holds. In this instance, the goal does not hold, so empowerment by extension is tried. The action trans(a Y b) ("Y" is an unbound variable) will be tried. "Empower1" will have ascertained that the current "Goal" is a member of this action's add list, and the action does not delete any of the protected goals (since, in this case, "PG" is empty).
The preconditions for trans(a Y b) are:

on(a Y)
clear(a)
clear(b)
diff(a b)

"Empower" is now called recursively to satisfy the list of preconditions so that trans(a Y b) can then be added to the end of the plan.
The values passed to "empower" are:

LG = [on(a Y) clear(a) clear(b) diff(a b)]
Plan = 0
PG = [ ]

Level 2:

The first goal will now be worked on. "Empower1" is called with:

Plan = 0
PG = [ ]
Goal = on(a Y)

"Goal" holds in the start state by binding "Y" to "p". Note that in level 1, the action trans(a Y b) has been instantiated to trans(a p b). Control is returned to "empower" which adds "on(a p)" to the protected goal list. Then "empower1" is called with the next goal, "clear(a)", and the following values:

Plan = 0
PG = [on(a p)]
Goal = clear(a)

"Goal" does not hold in the start state, so empowerment by extension will be tried. So as not to complicate this example any further, some steps will be omitted. The action chosen by empowerment by extension will be trans(c a r). The two remaining goals in "LG" already hold. Thus, the plan returned by "empower" to level 1 will be:

Plan = result(trans(c a r) 0)

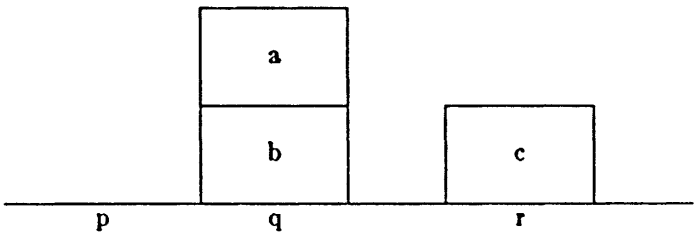
Level 1:

To recap, in level 1, empower1 was working on the first goal "on(a b)". This goal can be satisfied by the action trans(a p b). But not all the preconditions of trans(a p b) hold, so trans(c a r) must be applied first. Thus, the plan returned by "empower1" for the goal "on(a b)" is:

Plan = result(trans(a p b) result(trans(c a r) 0))

The state named by "Plan" is shown in Figure 4-2b.

Figure 4-2b: Intermediate State



Of the two original goals "on(a b)" and "on(b c)", "empower" has satisfied "on(a b)" and must now work on "on(b c)".
"Empower1" is again called with:

Plan = result(trans(a p b) result(trans(c a r) 0))
PG = [on(a b)]
Goal = on(b c)

"Goal" does not hold in the state produced by "Plan". Also, "Goal" cannot be satisfied by extension because moving "b" onto "c" first requires unstacking "a". This would violate the protected goal in "PG". The last alternative tried is empowerment by insertion. The last action in "Plan" is trans(a p b) which does not delete "Goal". Thus, "Goal" can be regressed over trans(a p b).
The add list of trans(a p b) is:

clear(p)
on(a b)

Any element in the above add list which also appears in the list of protected goals, is removed

from "PG". Thus, "PG" becomes:

PG = [ ]

The preconditions of trans(a p b) are:

on(a p)
clear(a)
clear(b)
diff(a b)

The above preconditions are added to the protected goal list:

PG = [on(a p) clear(a) clear(b) diff(a b)]

"Goal" has now been regressed over the last action. "Empower1" is then called recursively with the last action removed from "Plan".
The parameters are:

Plan = result(trans(c a r) 0)
PG = [on(a p) clear(a) clear(b) diff(a b)]
Goal = on(b c)

The recursive call to "empower1" will result in "Plan" being modified so that "Goal" holds.

Level 2:

"Goal" can now be satisfied using empowerment by extension. The action trans(b q c) is added onto the end of the plan. Thus, the plan returned to level 1 will be:

Plan = result(trans(b q c) result(trans(c a r) 0))

Level 1:

Empowerment by insertion has just succeeded in satisfying the goal "on(b c)" by modifying the plan. The action that had been regressed over is added to the end of the plan returned from level 2. The final plan produced by "empower" is:

Plan = result(trans(a p b) result(trans(b q c) result(trans(c a r) 0)))

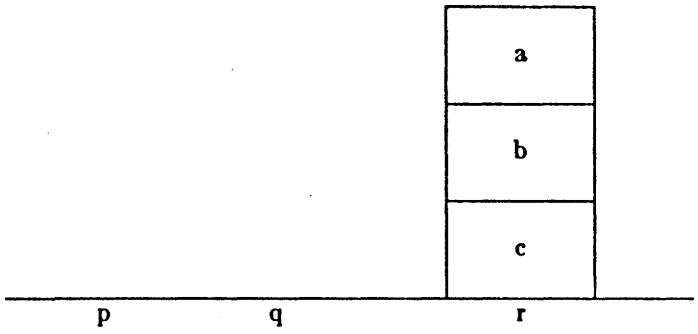
The result of doing empowerment by insertion is that the action trans(b q c) has been inserted into the plan. The goal state is shown in Figure 4-2c.

4.2 Changes Made to Warplan

The planner used by Madame is based on Warplan. Modifications for the conversion to the UNIX domain were carried out in three stages. First, Warplan was modified so that it would use Kowalski's representation; the resulting planner was named Warplan-Blocks. A listing of Warplan-Blocks appears in Appendix C. Since the axioms used to describe the UNIX world cannot be expressed in pure Kowalski representation, the second stage of the conversion involved modifying Warplan-Blocks so that it would use the UNIX axiomatization. The planner from the second stage was named Warplan-UNIX. In the third stage of the conversion, Warplan-UNIX was modified so that it would accept negated goals. The final planner was named Warplan-Not. A listing of Warplan-Not appears in Appendix D.

Originally, Warplan used a list description of the problem domain. For example, the "trans"

Figure 4-2c: Goal State



action in the blocks world was expressed by the following axioms:

```
precond( trans(X Y Z) [on(X Y) clear(X) clear(Z) diff(X Z)] )
addlist( trans(X Y Z) [clear(Y) on(X Z)] )
deletelist( trans(X Y Z) [clear(Z) on(X Y)] )
```

The major difference between the representation used by Warplan and Kowalski's representation is the use of the above three axioms. In Warplan, these axioms are used whenever the application of an action is being considered. For the conversion, no work was required for the "deletelist" axiom since its corresponding axiom in Kowalski's representation is the "preserves" axiom. The "deletelist" axiom is used to check if an action deletes a statement; this is easily done with the "preserves" axiom. The conversion of "precond" and "addlist" involved the removal of these two axioms from the knowledge base. General definitions for the "precond", and "addlist" axioms were then added to the definition of Warplan, and they provide the interface to the Kowalski representation. The axioms are shown in Figure 4-3a and Figure 4-3b. If the add list of an action is needed, the "addlist" axiom is executed. Since this axiom is no longer in the knowledge base, the add list information for the action is derived from its "holds" axioms in the knowledge base. Similarly, the "precond" axiom will access the knowledge base to get the information it requires from the "poss" axiom. The use of the "precond" and "addlist" axioms as an interface resulted in minimal changes to Warplan.

Some errors have been found in Warplan; they are discussed in [NOW84]. One error was corrected during the conversion to Warplan-Blocks. This correction required making a serious change to the definition of empowerment by insertion. The best way to describe the error in Warplan is with the following example. Assume the planner is working on a goal "G" which is true in the start state and no action in the plan developed so far deletes it. When faced with this goal "G", "empower1" will realize that it already holds and that no modifications to the plan are required. The planner will then continue to work on the remaining goals. Assume that at some point the planner must backtrack to the goal "G". Further assume that empowerment by extension cannot satisfy "G". If this occurs, "empower1" will try empowerment by insertion. The goal "G" will be regressed over the last action and "empower1" will be called recursively. "Empower1" checks to see if "G" already holds, which it does because "G" holds in the start state and no actions in the plan delete it. Therefore, the plan returned by the recursive call to "empower1" will be the plan originally passed to it. The result of empowerment by insertion is that a null action has been inserted in the plan. Planning will once again continue with the remaining goals. The same problem will occur again because the plan has not been changed. Therefore, the planner will again have to backtrack to the goal "G". The planner is now in an

Figure 4-3a: Redefined precond Axiom

```
% precond:       The preconditions for "Action" are
%           put in the list "PrecAct".
%
% precond( Action PrecAct )
%
% Input:   Action   - single action
%
% Output: PrecAct  - list of preconditions for "Action".


precond( Action PrecAct ) <-
      % Get the list of preconditions.
      retrieve( kwbase poss(result(Action X)) List )
      % Get rid of any unwanted preconditions.
      prune( List PrecAct );
```

Figure 4-3b: Redefined addlist Axiom

```
% addlist: Gets the addlist for a particular action.
%
% addlist( Action Temp List )
%
% Input:   Action  - single action
%          Temp    - initialized to [], temporary list
%
% Output: List    - contains the addlist for "Action"


addlist( Action Temp List ) <-
      % Get an added statement.
      clause( kwbase holds(X result(Action W)) Z )
      % Make sure we don't have the frame axiom.
      not( is_var(X) )
      % Have we looked at this one before?
      not( member(X Temp) )
      % No.
      cut
      addlist( Action [X|Temp] List );


addlist( Action List List );
```

infinite loop because it will keep on inserting null actions.

The solution to the problem was to check if an action was indeed inserted in the plan when empowerment by insertion is used. If a null action was inserted, then empowerment by insertion will fail. The problem with the insertion of null actions was present in Warren's [WAR74] original version of Warplan.

The next phase of the first conversion involved making two additional changes. These two changes were made to increase the efficiency of the planner. For the first change, a check was added in "empower1" to see if the goal is an element of the protected goal list. If it is, the goal has already been satisfied, so no work is required.

The second change, involves the use of the "only" axiom presented in Section 2.3. A check is done in "empower1" to see if the goal appears in an "only" axiom. If it does, it can be immediately determined from the knowledge base if the goal holds or could never hold. If the goal holds in the knowledge base, then no further work is required. Otherwise, since the goal appears in an "only" axiom and does not hold in the knowledge base, it can be immediately concluded that the goal will never hold. Therefore, "empower1" will fail, which causes backtracking to a previous goal. This concludes the first stage of the conversion.

The next stage of the conversion involved modifying Warplan-Blocks so that it would work in the UNIX domain. The axiomatization of the UNIX domain was described in Section 2.3. The interface to the knowledge base of Warplan-Blocks was changed so that the UNIX axioms could be used. No changes were made to the control structure of Warplan-Blocks. Thus, Warplan-UNIX is almost identical to Warplan-Blocks; the major difference is the form of the axioms in the knowledge base, and the access of these axioms. For example, the "precond" axiom in Figure 4-3a was rewritten; the "precond1" axiom appears in Figure 4-4. The "precond" axiom obtains the preconditions for the action from its "poss" axiom in the knowledge base. The "precond1" axiom obtains its preconditions from the "poss" axiom and an additional source. This source is the add list of the action. To use the add list of the action, "precond1" must be passed the action as well as the goal. "Precond1" uses these to access the "holds" axiom associated with the action that adds the goal. The body of this "holds" axiom is added to the list of preconditions. For example, one of the added statements for "mkdir" is:

```
holds( owner(User Path) result(mkdir(Path) S) ) <-
    prouver( holds(current_user(User) S) );
```

If the goal was

owner(User /u/anatrudel/dir)

and the action was

mkdir(/u/anatrudel/dir)

then

current_user(User)

would be added to the list of preconditions.

The third and final stage of the conversion was to modify Warplan-UNIX to accept negative goals and allow negations to appear in the knowledge base. As stated in Section 2.3, negation is required in the body of axioms that describe the UNIX domain. The major modification made to Warplan-UNIX was that the axiom "empower1_not" was added. "Empower1_not" accepts goals of the form "not(G)". The first thing done by "empower1_not" is to check if "not(G)" is a member of the protected goal list. If it is, the plan is not modified and "empower1_not" is finished with the goal. Next, it checks if "G" holds in the state produced by the plan developed so

Figure 4-4: Axiom Used by Warplan-UNIX

```
% precond1:     The preconditions for "Action" are put in the
%          list "PrecAct".
%          The preconditions are found by taking the body of
%          the following 2 axioms:
%               - The "poss" axiom for "Action".
%               - The "holds" axiom which says that "Action" adds
%                 "Goal". Using the body of this axiom will help
%                 to bind some of the variables in "Goal".
%
% precond1( Goal Action PrecAct )
%
% Input:  Goal     - The goal we are currently working on.
%                    It is added by "Action".
%          Action   - single action
%
% Output: PrecAct  - list of preconditions for "Action".



precond1( Goal Action PrecAct ) <-
     % Get the list of preconditions.
     retrieve( kwbase poss(result(Action X)) List1 )
     % Find the add statement used for "Goal".
     clause( kwbase holds(Goal result(X Y)) Body )
     % Make sure we don't have the frame axiom.
     not( is_var(X) )
     % Make sure we have the right add statement.
     eq( Action X )
     append( Body List1 List2 )
     % Get rid of any unwanted preconditions.
     prune( List2 PrecAct );
```

far. If "G" does not hold, this means that "not(G)" holds and "empower1_not" is finished. Notice here that the meaning of "not(G)" is that based on negation-as-failure [CLA78]. Otherwise, "G" holds and the plan must be modified so that "G" does not hold. Empowerment by extension is then used to modify the current plan so that "G" does not hold. An action which deletes "G" is found by looking at the "preserves" axiom for each action. Empowerment by extension will then continue as in "empower1". If empowerment by extension fails, then "empower1_not" tries empowerment by insertion. A check is made to see if the last action does not add "G". If it does not, the goal is regressed over the last action in the same way it is done in "empower1". Then, "empower1_not" is called recursively. In summary, "empower1_not" processes all the negated goals, and "empower1" processes the goals which are not negated. "Empower1" was not modified during this stage of the conversion. This concludes the conversion to Warplan-Not.

Surprisingly, the last conversion from Warplan-UNIX to Warplan-Not was the simplest of the three. The only major modification required was the addition of "empower1_not". It had been anticipated that the inclusion of negation would cause major difficulties; this was not the case.

4.3 How the Closest P-s-s is Used

Madame will find the closest p-s-s in the spider to the goal state. This was discussed in Chapter 3. This closest p-s-s is then passed to the planning component of Madame. The planner does not explicitly use the closest p-s-s as the start state. Instead, the plan "P" linking p-s-s 0 to the closest p-s-s is used as an initial approximation to the required plan. Plan "P", along with the list of goals which do not hold in the closest p-s-s, is given to the planner. The planner will modify this plan by inserting and adding actions to it. The result will be a plan which transforms the start state into the goal state. The advantage of passing plan "P" to the planner is that the planner will have a head start as some of the goals are satisfied by this plan.

Figure 4-5a: P-s-s 0

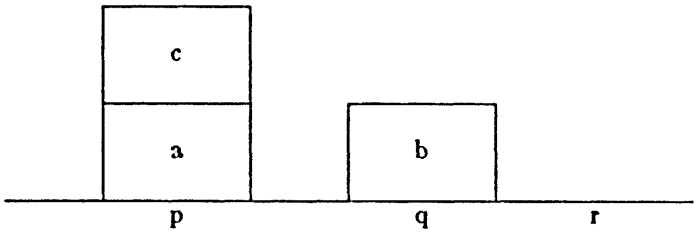
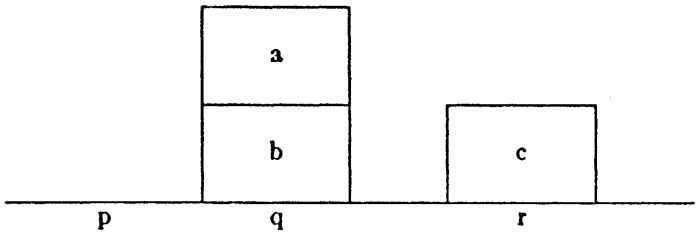


Figure 4-5b: P-s-s 1



Alternatively, the closest p-s-s could be explicitly used as the start state. This would not take advantage of Warplan's empowerment by insertion. For example, let the goals be "on(b c)" and "on(a b)", and p-s-s 0 be represented by Figure 4-5a; p-s-s 1 is represented by Figure 4-5b. The closest p-s-s is p-s-s 1 because it is at a distance of 1 from the goal state. If p-s-s 1 is explicitly used as the start state, then the goal "on(a b)" would first have to be undone in order to make "on(b c)" hold. The goal "on(a b)" would be undone by removing block "a" from "b". Then, planning could continue. The resulting plan between the start state and goal state would contain five actions. On the other hand, if the planner is passed the plan linking p-s-s 0 to p-s-s 1, it would insert an action in the plan and both goals would then be satisfied. The resulting plan would only contain three actions. Therefore, using the closest p-s-s explicitly as the start state could often result in some work having to be undone. This is the reason that the closest p-s-s was not explicitly used by Madame.

5

Results and Conclusions

5.1 Summary of Testing done with Madame

The disadvantage of using the spider is the overhead involved in searching and updating the spider. The advantage is that the planner will usually have less work to do. In most cases, the plan built between the closest p-s-s and the goal state is shorter than a direct plan built between the start state and the goal state. Tests were done to determine whether or not the advantages of using a spider outweigh the disadvantages.

Identical planning queries were passed to Madame and then only to the planning component of Madame. The former tests a planner which uses the spider while the latter tests a planner which does not. The two planners are identical except for the use of the spider; thus, any differences in performance between the two planners will be due to the spider.

Two sets of tests were run: test1 and test2. Test1 compared a version of Madame (which used Warplan-Blocks as the planning component) with Warplan-Blocks. Test2 compared a version of Madame (which used Warplan-Not as the planning component) with Warplan-Not. During each set of tests, total inferences, total unifications and total time were measured.

5.2 Input Data and Results

All the files related to test1 are contained in Appendix E and the files for test2 are contained in Appendix F. Each of these appendices contain listings of the following files:

1. The initial spider
2. The axiomatization of the problem domain
3. The input queries to Madame
4. The input queries to the planning component which did not use the spider
5. The final spider
6. Output from Madame
7. Output from the planning component which did not use the spider

The planning queries used in the tests consisted of twenty lists of goals. Each goal list represented a goal state. The lists of goals for test1 were from the blocks world, and from the UNIX domain for test2. In both cases, an effort was made to choose queries at random. This was done so that the queries would not bias the tests. The queries could have been generated so that one planner was favoured over the other. For example, queries could be generated where the closest p-s-s in the spider for each query would be p-s-s 0 (the start state). Thus, the spider would be useless, making the planner inefficient. At the other extreme, the same query could be repeated twenty times. The planner using the spider would produce a plan once while the other planner would re-generate the same plan twenty times. The planner using the spider would be the

clear winner in this case. A few queries were repeated; this models a user who asks for a plan, then forgets the plan and repeats the query.

In both tests, the spider initially contained one p-s-s: p-s-s 0 (i.e. the start state). By having only one p-s-s in the spider initially, Madame was not given an unfair advantage. When the planning component was run by itself, it had the same start state defined in its knowledge base.

5.2.1 Input Data Used in Test1

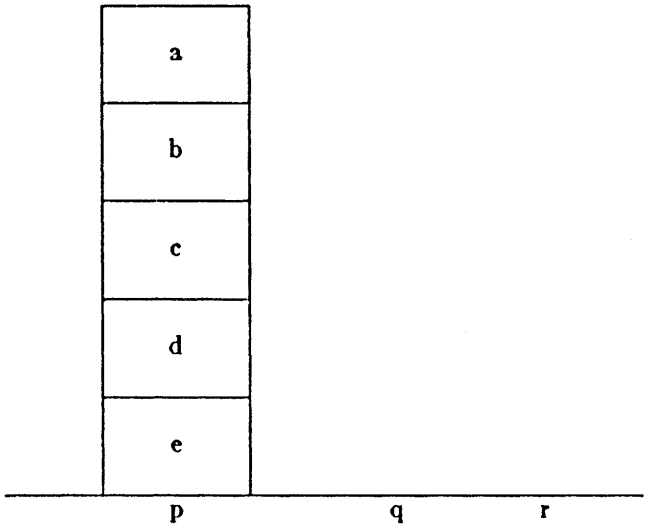
Test1 used Warplan-Blocks as the planning component. First, twenty lists of goals from the blocks world were passed to Madame. Then, the same twenty lists of goals were passed to Warplan-Blocks, the planning component of Madame. In the latter case, Warplan-Blocks had no access to the spider. Figure 5-1 shows the twenty lists of goals used for test1.

Figure 5-1: Goals Used as Input for Test1

1. [clear(q) clear(p)]
2. [on(a r)]
3. [on(e d) on(d c) on(c b) on(b a) on(a q)]
4. [on(e r)]
5. [on(b q) on(c r) on(a p)]
6. [on(c b)]
7. [on(a b)]
8. [on(d e) on(e p) on(c b) on(b a) on(a q)]
9. [clear(a)]
10. [on(a q) on(b a) on(d b) on(c d) on(e p)]
11. [clear(b)]
12. [on(c d) on(d e) on(e p) on(b q) on(a r)]
13. [clear(c)]
14. [clear(d)]
15. [on(c r) on(d c) on(a p) clear(d) clear(a)]
16. [clear(e)]
17. [on(d a) clear(c) on(e b)]
18. [on(d q)]
19. [clear(c) clear(b) clear(a)]
20. [on(e a)]

The spider initially used by Madame during test1 contained only one p-s-s; the start state. The start state is represented by the diagram in Figure 5-2. This is also the start state used by Warplan-Blocks.

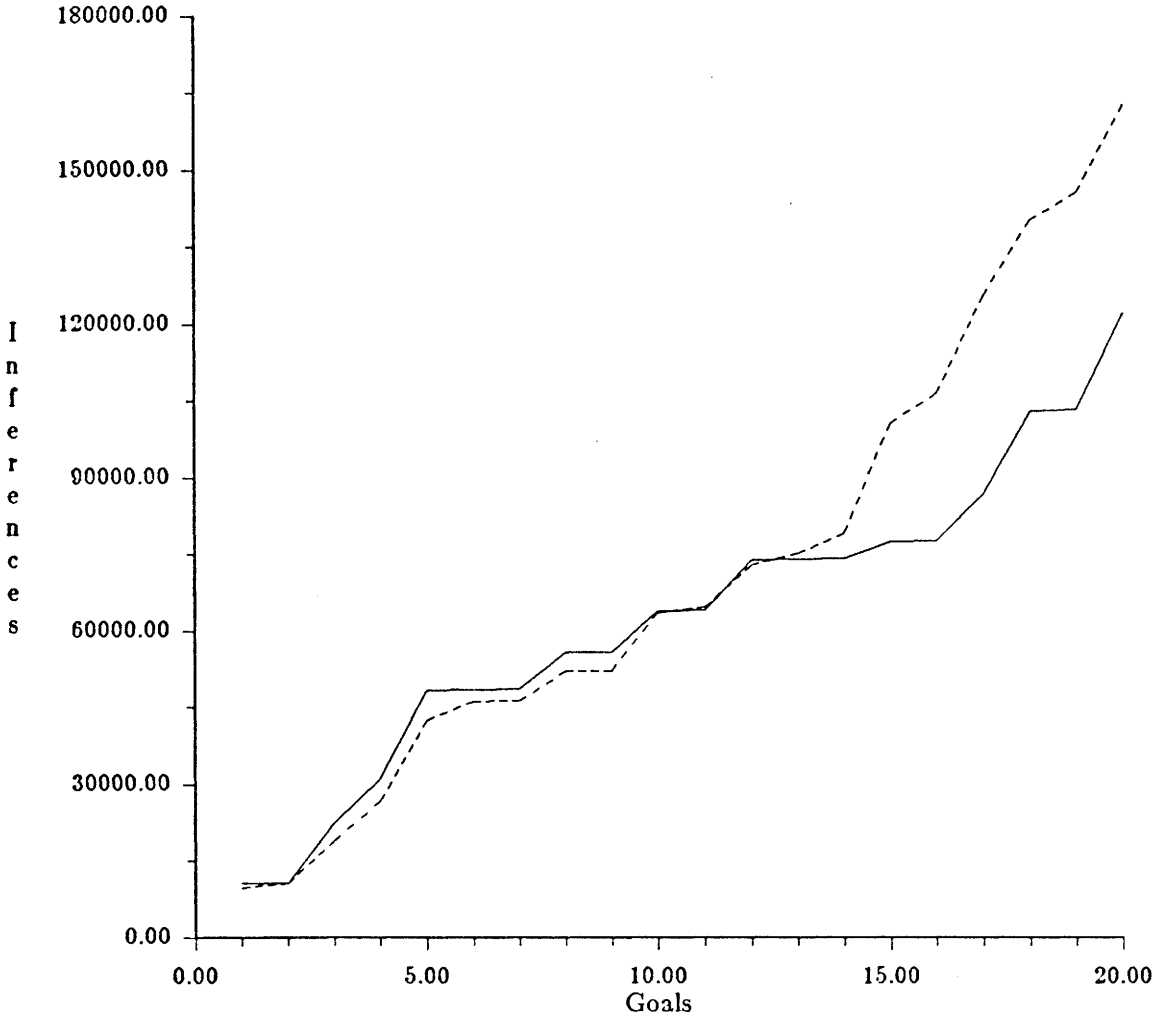
Figure 5-2: Start State for Test1



5.2.2 Results from Test1

Graphs comparing the number of inferences, unifications and total time for the planners which did and did not use the spider are contained in Figure 5-3a, Figure 5-3b and Figure 5-3c.

Figure 5-3a: Test1 - Total Inferences

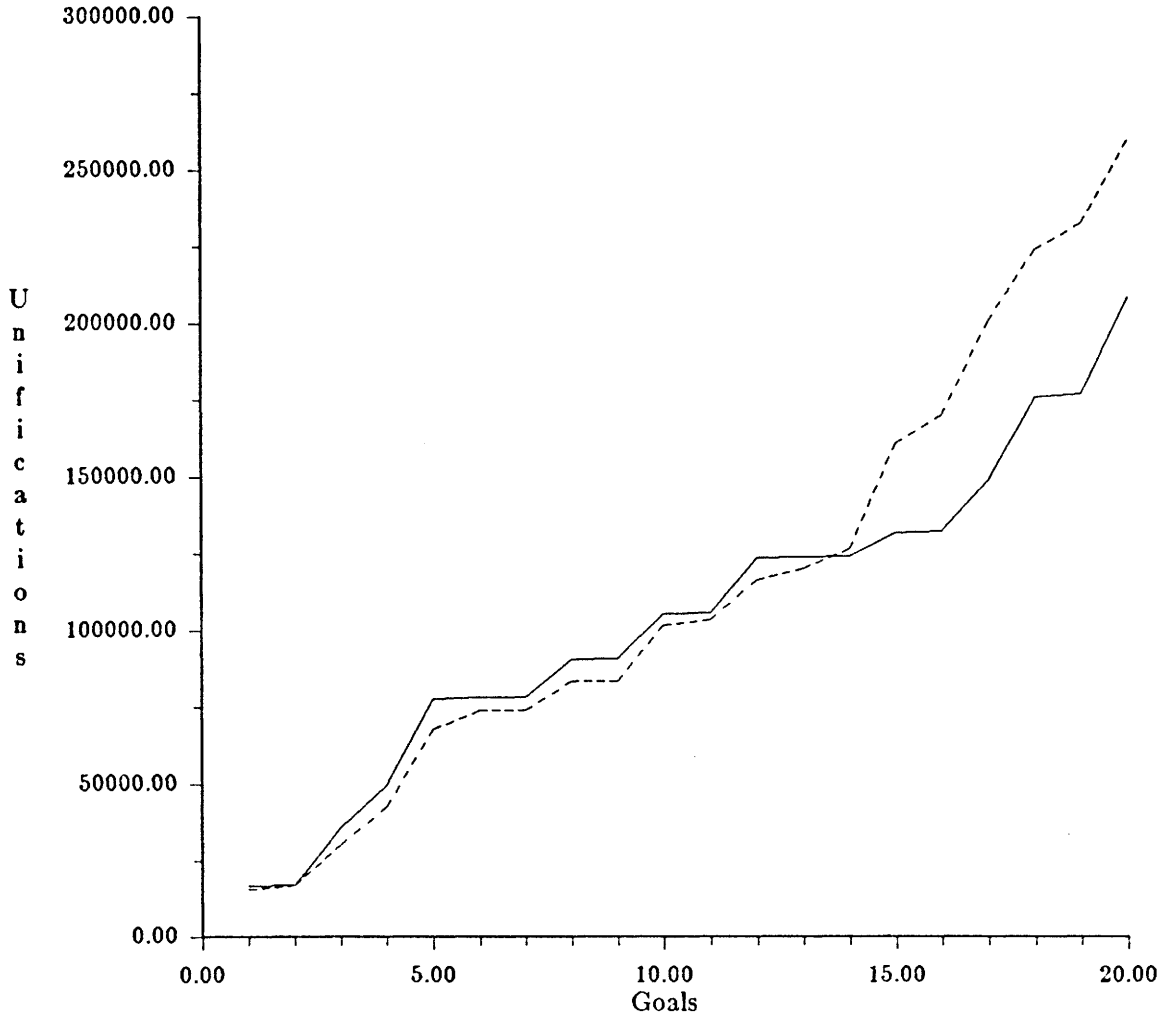


Test was run in the Blocks World

Solid Line: Madame

Dashed Line: Warplan-Blocks

Figure 5-3b: Test1 - Total Unifications

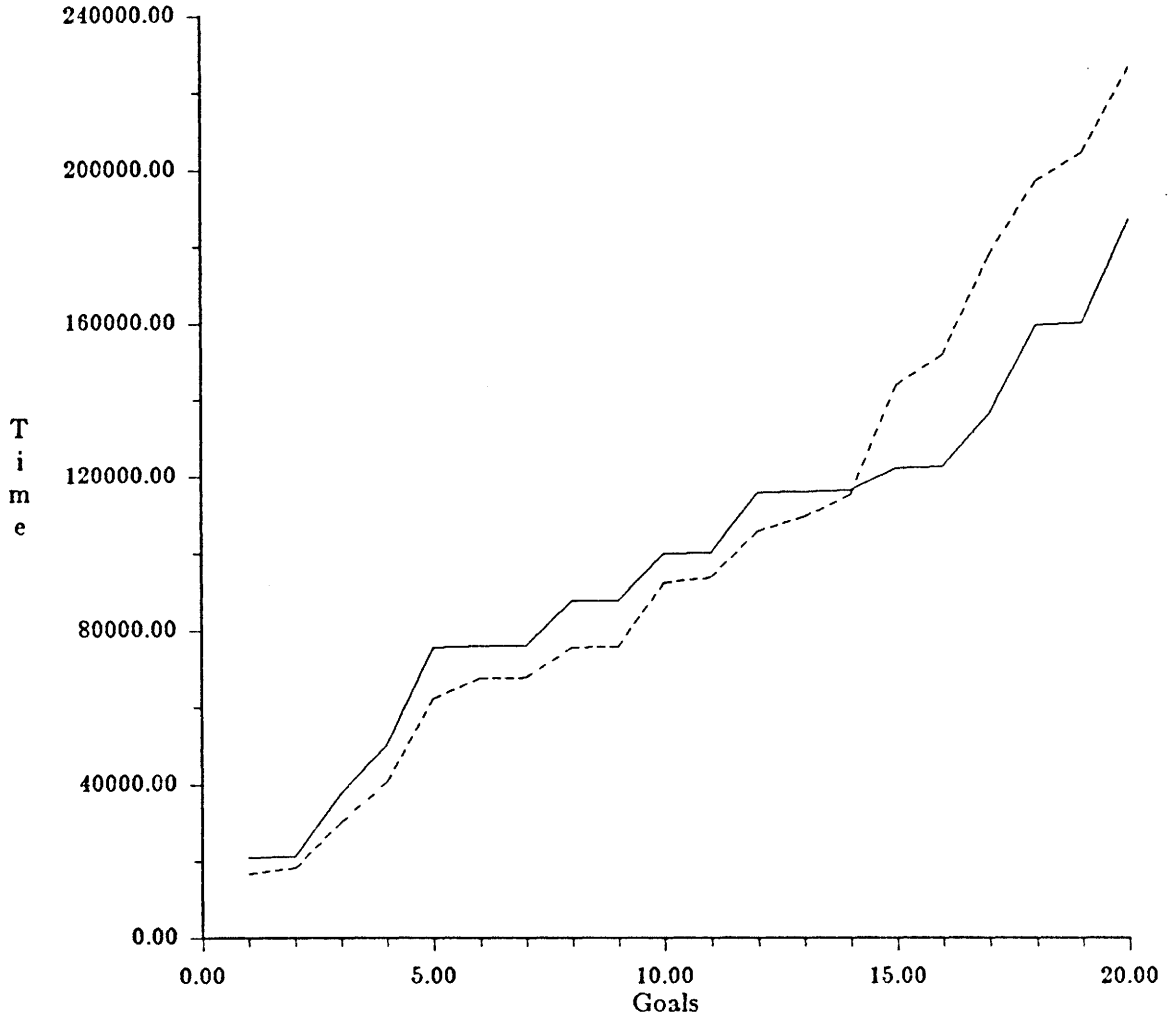


Test was run in the Blocks World

Solid Line: Madame

Dashed Line: Warplan-Blocks

Figure 5-3c: Test1 - Total Time (msec)



Test was run in the Blocks World

Solid Line: Madame

Dashed Line: Warplan-Blocks

5.2.3 Input Data Used in Test2

Test2 used Warplan-Not as the planning component. First, twenty lists of goals from the UNIX domain were passed to Madame. Then, the same twenty lists of goals were passed to Warplan-Not, the planning component of Madame. In the latter case, Warplan-Not had no access to the spider. Figure 5-4 shows the twenty lists of goals used for test2.

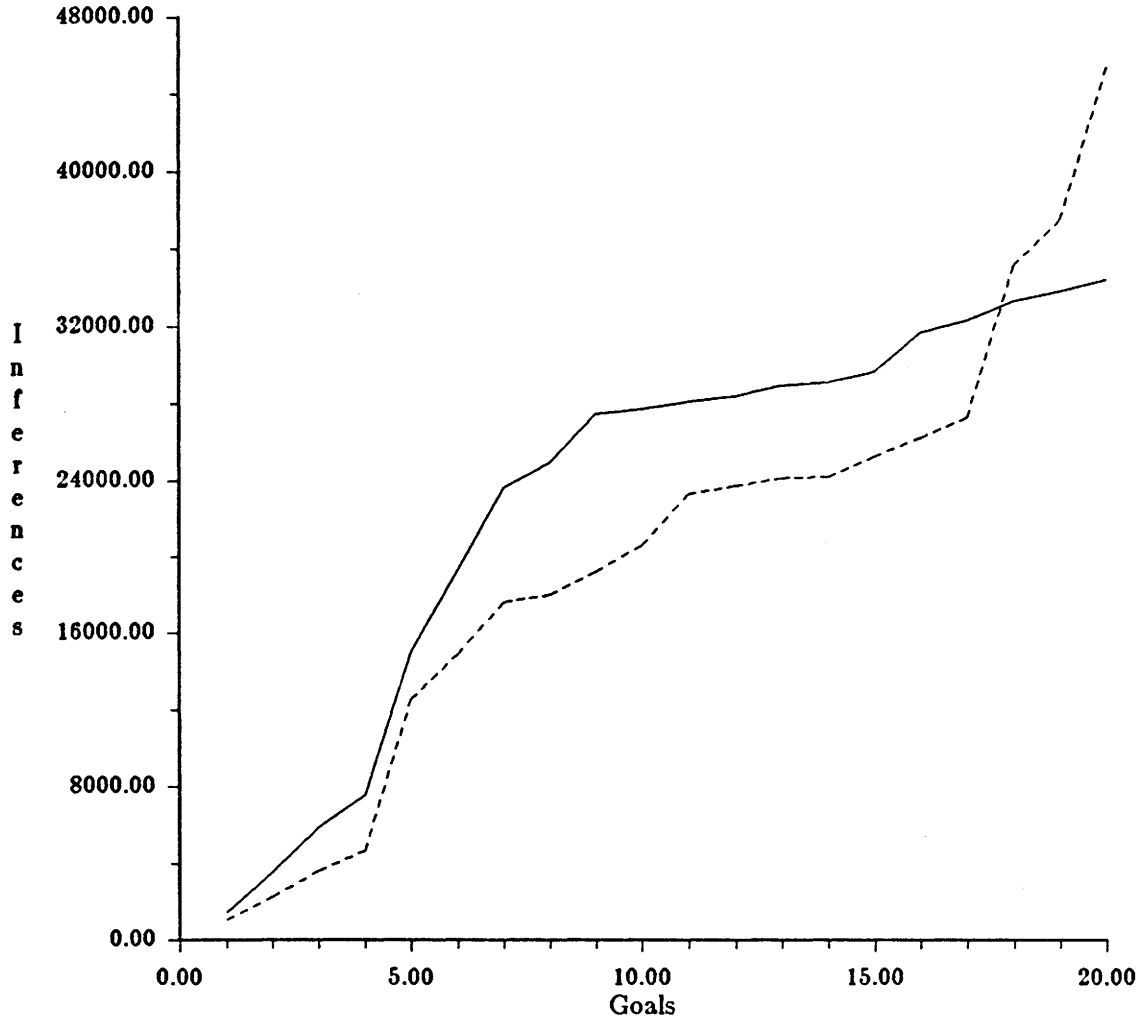
Figure 5-4: Goals Used as Input for Test2

1. [perms(group w /u/anatrudel/junk)]
2. [directory(/u/anatrudel/dir1)]
3. [owner(anatrudel /u/anatrudel/dir2)]
4. [not(fexist(/u/anatrudel/thesis/madame))]
5. [directory(/u/anatrudel/dir1) perms(group w /u/anatrudel/dir1)]
6. [directory(/u/anatrudel/junk)]
7. [perms(group w /u/anatrudel/dir2)]
8. [not(perms(group r /u/anatrudel))]
9. [fexist(/u/anatrudel/dir3)]
10. [grpmember_file(/u/anatrudel/dir3 the_group)]
11. [perms(group w /u/anatrudel/dir2)]
12. [not(perms(group r /u/anatrudel))]
13. [not(perms(group r /u/anatrudel/thesis/madame))]
14. [not(fexist(/u/anatrudel/dir5))]
15. [not(fexist(/u/anatrudel/thesis/madame))]
16. [not(owner(anatrudel /u/anatrudel/junk))]
17. [not(owner(anatrudel /u/anatrudel/thesis/madame))]
18. [directory(/u/anatrudel/dir1) perms(group w /u/anatrudel/dir1)]
19. [directory(/u/anatrudel/junk)]
20. [fexist(/u/anatrudel/dir2) perms(group w /u/anatrudel/dir2)]

5.2.4 Results from Test2

Graphs comparing the number of inferences, unifications and total time for the planner which did and did not use the spider are contained in Figure 5-5a, Figure 5-5b and Figure 5-5c.

Figure 5-5a: Test2 - Total Inferences

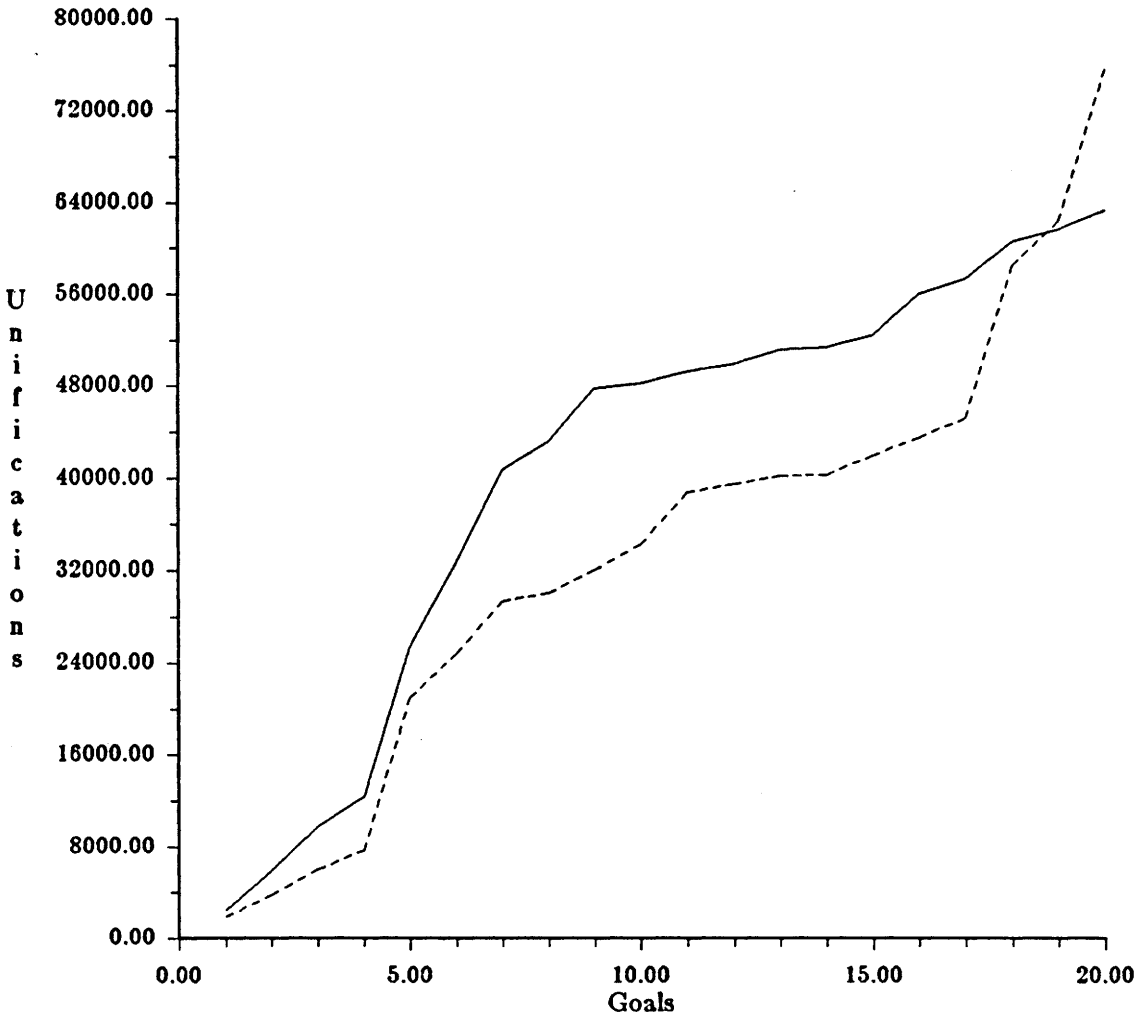


Test was run in the UNIX Domain

Solid Line: Madame

Dashed Line: Warplan-Not

Figure 5-5b: Test2 - Total Unifications

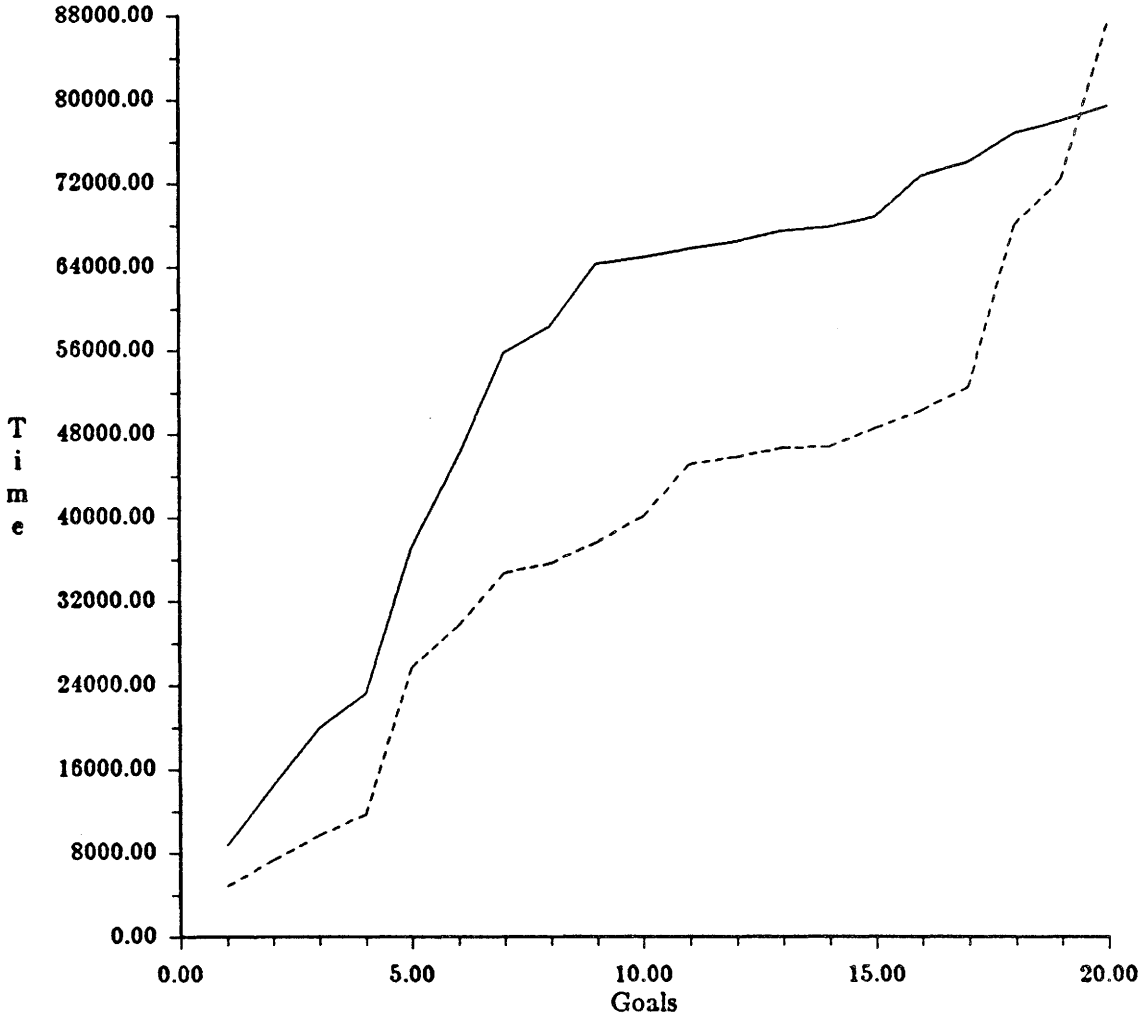


Test was run in the UNIX Domain

Solid Line: Madame

Dashed Line: Warplan-Not

Figure 5-5c: Test2 - Total Time (msec)



Test was run in the UNIX Domain

Solid Line: Madame

Dashed Line: Warplan-Not

5.3 Conclusions Drawn from the Tests

The results of test1 and test2 show that once the spider has grown to a certain size, Madame becomes more efficient. At the beginning of the test, the overhead of keeping the spider makes the planner less efficient. Eventually, the spider will attain a size where the probability of finding a p-s-s which is closer to the goal state than the start state is relatively high. This can be seen in the preceding graphs as the point where the two curves cross. Without a p-s-s removal function, it is probable that Madame will become less efficient as more goal states are added to the spider. This is due to the fact that spider traversal and updating would take a considerable amount of time. Thus, if the curves were extended, they would probably cross again. From that point on, Madame would be less efficient because of the large spider it would have to deal with.

It is interesting to note that the spider from test1 was more useful to the planner than the spider for test2. The p-s-s found to be the closest to the goal state was different from p-s-s 0 (the start state) more often in test1. Thus, the spider was more beneficial in test1 than in test2.

5.4 Problems with Madame and the Representation Used

As mentioned in Chapter 1, an "ideal" planner would be complete, optimal and very efficient. Madame fails in all three categories. A basic reason why Madame could not be changed into an "ideal" planner is that it uses Kowalski's representation.

Due to the problems encountered in trying to axiomatize the UNIX world, one can conclude that Kowalski's representation does not have complete expressive power. The representation cannot be used to fully axiomatize the UNIX domain. The major flaw with the representation is that is uses an oversimplified view of actions. Kowalski assumes that when the right conditions occur in a state, the action can be applied to generate a new state. In the UNIX domain, actions are much more complicated. Factors that should be considered are: time, and the different possibilities for each action. Each UNIX command requires a different amount of time to execute. If while producing a plan, the planner has the choice between two commands, it should choose the quicker of the two. For example, it is faster to move a file rather than re-copy it. The representation should also consider the different possibilities for each action. For example, almost all the UNIX commands can be used with parameters, and depending on the parameters used, the result of applying an action differs.

5.5 Directions for Future Work

Some interesting work with the spider still remains to be done. For example, heuristics for the acceptance and deletion of p-s-s's could be implemented. At the present, Madame accepts all goal states as new p-s-s's, and p-s-s's are never deleted from the spider. Also, new heuristics for determining the distance between a p-s-s and the goal state could be developed and tested.

Further tests could also be run to measure the efficiency of the spider. For example, tests similar to test1 and test2 could be run using other problem domains.

Since no p-s-s's are presently being deleted from the spider, the spider can be made to grow arbitrarily large. It would be interesting to measure the true efficiency of the planner when it has a very large spider with which to deal.

There are other areas besides the spider where work needs to be done. Because of the problems with Kowalski's representation, a new representation should be developed. The new representation should be powerful enough to allow the complete axiomatization of the UNIX domain. This would then serve as an ideal building block for a planner. Based on this representation, an efficient, optimal and complete planner could possibly be built.

References

[CLA78] Clark, K.L. *Negation as Failure,* in Logic and Databases, H. Gallaire and J. Minker, Plenum Press, N.Y., 1978. Pp. 293-322.

[EMD84] van Emden, M.H., and Goebel, R.G. Waterloo Unix Prolog User's Manual. Logic Programming and AI Group, Department of Computer Science, University of Waterloo, Waterloo, Canada, July 1984.

[GEO83] Georgeff, M. *Communication and Interaction in Multi-Agent Planning.* Proceedings of The National Conference on Artificial Intelligence, Washington D.C., Aug 1983. Pp. 125-129.

[KIB81] Kibler, D., and Morris, P. *Don't be Stupid.* Proceedings of the Seventh International Joint Conference on Artificial Intelligence, 1981. Vol 1, Pp. 345-347.

[KOW79] Kowalski, R. Logic for Problem Solving. New York: Elsevier North Holland, 1979.

[MAL83] Malito, J.L. *Project Report for CS786.* Department of Computer Science, University of Waterloo, Waterloo, Canada. 1983.

[MAL84] Malito, J.L. *UWISH: The UNIX Wizard Component of the Intelligent Shell.* MMath dissertation. Department of Computer Science, University of Waterloo, Waterloo, Canada. 1984.

[NOW84] Nowlan, S. *Development of an Adaptive Planner.* Report for SD362. Department of Systems Design, University of Waterloo, Waterloo, Canada. 1984.

[PAK83] Pakalns, J.L. *AHA, A UNIX Consultant!* MMath dissertation. Department of Computer Science, University of Waterloo, Waterloo, Canada. 1983.

[SAC79] Sacerdoti, E.D. *Problem Solving Tactics.* Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Japan, 1979. Pp. 1077-1085.

[WAR74] Warren, D.H.D. Warplan: A System for Generating Plans. Memo No. 76, Department of Computational Logic, University of Edinburgh, Edinburgh, June 1974.

[WIL83] Wilkins, D.E. *Representation in a Domain-Independent Planner.* Proceedings of the Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, Aug 1983. Vol 2, Pp. 733-740.

Appendix A: Listing of UNIX Axiomatization

This appendix contains the axiomatization of the UNIX domain which allows negation. The axiomatization is used by Warplan-Not which appears in Appendix D. The majority of the axioms in this appendix were derived from [MAL83]. Some modifications were made so that the axiomatization could be used by Madame (the modifications are described in Section 2.3).

```
contains_slash( ['/'|List] ) <-
        cut;

contains_slash( [Char|Rest] ) <-
        contains_slash( Rest );
```

```
diff( X Y ) <-
        ne( X Y );
```

```
% Check if we have exec
% permissions for "Pathname".


% "User" owns the "Pathname".

execperms( Pathname S ) <-
        prouver( holds( current_user(User) S ) )
        prouver( holds( owner(User Pathname) S ) )
        prouver( holds( perms(user x Pathname) S ) );


% "User" belongs to a group which has exec
% permission for "Pathname".

execperms( Pathname S ) <-
        prouver( holds( current_user(User) S ) )
        not( prouver( holds(owner(User Pathname) S) ) )
        prouver( holds( grpmember_file(Pathname Group) S ) )
        prouver( holds( grpmember_user(User Group) S ) )
        prouver( holds( perms(group x Pathname) S ) );


% Everybody has exec permission for "Pathname".

execperms( Pathname S ) <-
        prouver( holds( current_user(User) S ) )
        not( prouver( holds(owner(User Pathname) S) ) )
        prouver( holds( grpmember_file(Pathname Group) S ) )
        not( prouver( holds(grpmember_user(User Group) S) ) )
        prouver( holds( perms(general x Pathname) S ) );
```

```
% Describe start state

holds( fexist(/u/anatrudel)          0 );
holds( fexist(/u/anatrudel/junk)    0 );
holds( fexist(/u/anatrudel/thesis)          0 );
holds( fexist(/u/anatrudel/thesis/madame) 0 );

holds( directory(/u/anatrudel)          0 );
holds( directory(/u/anatrudel/thesis) 0 );

holds( owner(anatrudel /u/anatrudel)            0 );
holds( owner(anatrudel /u/anatrudel/junk)      0 );
holds( owner(anatrudel /u/anatrudel/thesis)    0 );
holds( owner(anatrudel /u/anatrudel/thesis/madame) 0 );

holds( valid_group(the_group) 0 );

holds( grpmember_file(/u/anatrudel the_group)              0 );
holds( grpmember_file(/u/anatrudel/junk the_group)        0 );
holds( grpmember_file(/u/anatrudel/thesis
      the_group)          0 );
holds( grpmember_file(/u/anatrudel/thesis/madame
      the_group) 0 );

holds( grpmember_user(anatrudel the_group) 0 );

holds( perms(user r /u/anatrudel)                0 );
holds( perms(user r /u/anatrudel/junk)                  0 );
holds( perms(user r /u/anatrudel/thesis)        0 );
holds( perms(user r /u/anatrudel/thesis/madame)  0 );

holds( perms(user w /u/anatrudel)                0 );
holds( perms(user w /u/anatrudel/junk)                  0 );
holds( perms(user w /u/anatrudel/thesis)        0 );
holds( perms(user w /u/anatrudel/thesis/madame)  0 );

holds( perms(user x /u/anatrudel)                0 );
holds( perms(user x /u/anatrudel/thesis)        0 );
holds( perms(user x /u/anatrudel/thesis/madame)  0 );

holds( perms(group r /u/anatrudel)               0 );
holds( perms(group r /u/anatrudel/junk)          0 );
holds( perms(group r /u/anatrudel/thesis)        0 );
holds( perms(group r /u/anatrudel/thesis/madame) 0 );

holds( perms(group w /u/anatrudel) 0 );

holds( perms(group x /u/anatrudel)               0 );
holds( perms(group x /u/anatrudel/thesis)    0 );
holds( perms(group x /u/anatrudel/thesis/madame) 0 );

holds( perms(general r /u/anatrudel)             0 );
holds( perms(general r /u/anatrudel/junk)       0 );
holds( perms(general r /u/anatrudel/thesis)     0 );
```

```
holds( perms(general r /u/anatrudel/thesis/madame) 0 );

holds( perms(general x /u/anatrudel)              0 );
holds( perms(general x /u/anatrudel/thesis)       0 );
holds( perms(general x /u/anatrudel/thesis/madame) 0 );

holds( current_user(anatrudel) 0 );

holds( current_group(the_group) 0 );

holds( home(/u/anatrudel) 0 );

holds( working_directory(/u/anatrudel) 0 );

holds( umask(user r) 0 );
holds( umask(user w) 0 );
holds( umask(user x) 0 );

holds( umask(group r) 0 );
holds( umask(group x) 0 );

holds( umask(general x) 0 );



holds( fexist(Pathname) result(mkdir(Pathname) S) );

holds( directory(Pathname) result(mkdir(Pathname) S) );

holds( owner(User Pathname) result(mkdir(Pathname) S) ) <-
      prouver( holds(current_user(User) S) );

holds( perms(Class Op Pathname)
      result(mkdir(Pathname) S) ) <-
      prouver( holds(umask(Class Op) S) );

holds( grpmember_file(Pathname Group)
      result(mkdir(Pathname) S) ) <-
      prouver( holds(current_group(Group) S) );



holds( perms(user x Pathname) result(chmod(100 Pathname) S) );
holds( perms(user w Pathname) result(chmod(200 Pathname) S) );
holds( perms(user r Pathname) result(chmod(400 Pathname) S) );

holds( perms(group x Pathname) result(chmod(10 Pathname) S) );
holds( perms(group w Pathname) result(chmod(20 Pathname) S) );
holds( perms(group r Pathname) result(chmod(40 Pathname) S) );

holds( perms(general x Pathname) result(chmod(1 Pathname) S) );
```

```
holds( perms(general w Pathname) result(chmod(2 Pathname) S) );
holds( perms(general r Pathname) result(chmod(4 Pathname) S) );



holds( U result(V W) ) <-
        prouver( holds(U W) )
        preserves( V U );

-------------------------------------


only( contains_slash(X) );
only( diff(X Y) );
only( parent(X Y) );
only( parent_list(X Y) );
only( valid_chmod_num(X) );

-------------------------------------


% parent( /u/usr/file Parent ) succeeds with Parent = /u/usr

parent( Pathname Dir ) <-
        is_atom( Pathname )
        ne( Pathname / )
        name( Pathname List1)
        parent_list( List1 List2 )
        name( Dir List2 );

-------------------------------------


% same as parent except the arguments are lists

parent_list( ['/'|Rest] [] ) <-
        not( contains_slash(Rest) );

parent_list( [Char|Rest1] [Char|Rest2] ) <-
        contains_slash( Rest1 )
        parent_list( Rest1 Rest2 );

-------------------------------------


poss( 0 );                      % start state.
```

```
poss( result(chmod(Num Pathname) S) ) <-
      poss(S)
      valid_chmod_num( Num )
      prouver( holds(current_user(User) S) )
      prouver( holds(owner(User Pathname) S) );


poss( result(rm(File) S) ) <-
      poss( S )
      prouver( holds(fexist(File) S) )        % Make sure
                                              % "File" exists.
      not( prouver( holds(directory(File) S) ) )  % Make sure
                                              % "File" is not
                                              % a directory.
      rmperms( File S );                      % Do we have permission
                                              % to delete "File"?


poss( result(mkdir(Pathname) S) ) <-
      poss( S )
      not( prouver( holds(fexist(Pathname) S) ) )
      parent( Pathname Parent )
      prouver( holds(fexist(Parent) S) )
      prouver( holds(directory(Parent) S) )
      writeperms( Parent S );


%poss( result(rmdir(Pathname) S) ) <-
      %poss( S )
      %prouver( holds(fexist(Pathname) S) )
      %prouver( holds(directory(Pathname) S) )
      %not( prouver( holds(working_directory(Pathname) S) ) )
      %writeperms( Pathname S );

-------------------------------------


preserves( chmod(Num Pathname) Stat ) <-
      diff( Stat perms(X Y Pathname) );


preserves( rm(File) Stat ) <-
      diff( Stat fexist(File) )
      diff( Stat owner(_ File) )
      diff( Stat perms(_ _ File) )
```

```
        diff( Stat grpmember_file(File _) )
        diff( Stat link(File _) )
        diff( Stat link(_ File) );


preserves( mkdir(Pathname) Stat );


%preserves( rmdir(Pathname) Stat ) <-
        %diff( Stat fexist(Pathname) )
        %diff( Stat directory(Pathname) )
        %diff( Stat owner(_ Pathname) )
        %diff( Stat perms(_ _ Pathname) )
        %diff( Stat grpmember_file(Pathname _) );

-------------------------------------

% This axiom is used by 'prune' in the planning phase.

replace( execperms(X Y) );
replace( rmperms(X Y) );
replace( writeperms(X Y) );

-------------------------------------


% "rmperms" checks to see if we have the
% correct permissions to "rm"
% "Pathname". We have the correct permissions
% if we have write permission
% on the parent directory.

rmperms( Pathname S ) <-
        parent( Pathname Parent )
        writeperms( Parent S );

-------------------------------------


% validate the number used with 'chmod'.

valid_chmod_num( 100 );
valid_chmod_num( 200 );
```

```
valid_chmod_num( 400 );
valid_chmod_num( 10  );
valid_chmod_num( 20  );
valid_chmod_num( 40  );
valid_chmod_num( 1   );
valid_chmod_num( 2   );
valid_chmod_num( 4   );
```

```
% Check if we have write permissions for "Pathname".


% "User" owns the "Pathname".

writeperms( Pathname S ) <-
      prouver( holds( current_user(User) S ) )
      prouver( holds( owner(User Pathname) S ) )
      prouver( holds( perms(user w Pathname) S ) );


% "User" belongs to a group which has
% write permission for "Pathname".

writeperms( Pathname S ) <-
      prouver( holds( current_user(User) S ) )
      not( prouver( holds(owner(User Pathname) S) ) )
      prouver( holds( grpmember_file(Pathname Group) S ) )
      prouver( holds( grpmember_user(User Group) S ) )
      prouver( holds( perms(group w Pathname) S ) );


% Everybody has write permission for "Pathname".

writeperms( Pathname S ) <-
      prouver( holds( current_user(User) S ) )
      not( prouver( holds(owner(User Pathname) S) ) )
      prouver( holds( grpmember_file(Pathname Group) S ) )
      not( prouver( holds(grpmember_user(User Group) S) ) )
      prouver( holds( perms(general w Pathname) S ) );
```

Appendix B: Listing of Warplan

This appendix contains the WUP version of Warplan as modified by Kowalski and van Emden.

```
% "append" appends the first two lists together to produce the third
% argument.
%
% ie. append( [a b] [c d] X ) returns X = [a b c d]
%
%
append( [] Y Y );
append( [U|X] Y [U|Z] ) <-
        append( X Y Z );
```

```
%
% "cons" breaks the second argument into a list.
%
% ie.  cons( X foo(a) ) returns X = [foo a]
%      cons( X a      ) returns X = [a]
%
%
cons( [F] F ) <-
        atomic( F )
        cut;
cons( L F ) <-
        functor( F L );
```

```
deletes( A X ) <-
        instantiate( [ A X ] 0 )
        deletelist( A L )
        member( X L );
```

```
%
% empower( P1 G1 P2 G2 )
%
%  G1:  list of goals achieved so far
%  P1:  plan to achieve G1
%  G2:  list of goals left to achieve
%  P2:  this is the plan we are looking for, it will solve all the goals
%
%
empower( P G P [] );
empower( P G P1 [X|G1] ) <-
    empower1( P G P2 X )
       empower( P2 [X|G] P1 G1 );
```

```
% return P because X holds in state produced by P

empower1( P G P X ) <-
        holds( X P );

% empowerment by extension

empower1( P G :(P1 A) X ) <-
        addlist( A L )
        member( X L )
        preserves( A G )
        precond( A G1 )
        not ( inconsistent( G1 G ) )
        empower( P G P1 G1 )
        preserves( A G );

% empowerment by insertion

empower1( :(P0 P1) G :(P2 P1) X ) <-
        not( deletes( P1 X ) )
        addlist( P1 G1 )
        precond( P1 G2 )
        minus( G G1 G3 )
        union( G2 G3 G4 )
        empower1( P0 G4 P2 X )
        not( deletes( P1 X ) );
```

```
holds( [] P );
holds( X :(P A) ) <-
        addlist( A L )
        member( X L );

% the famous frame axiom

holds( X :(P A) ) <-
        holds( X P )
        not( deletes( A X ) );
holds( X [] ) <-
        initially( X );
holds( X P ) <-
        always( X );
```

```
inconsistent( C G ) <-
```

```
        imposs( S )
        not(not( intersect( C S ) ))
        append( C G G1 )
        instantiate( G1 0 )
        subset( S G1 );
```

```
inst( qqq(N1) N1 N2 ) <-
        cut
        add( N1 1 N2 );
inst( _1 N N ) <-
        is_atom( _1 )
        cut;
inst( _1 N N2 ) <-
        cons( [X|Y] _1 )
        instargs( Y N N2 );
```

```
%
% Initially N1 = 0 and the first argument is a list.
%
% "instantiate" will bind all variables to the form X=qqq(int) where
% int is a unique integer.
%
%
instantiate( [X|L] N1 ) <-
        cons( [Y|Z] X )
        instargs( Z N1 N2 )
        instantiate( L N2 );
instantiate( [] N );
```

```
instargs( [X|L] N1 N3 ) <-
        inst( X N1 N2 )
        instargs( L N2 N3 );
instargs( [] N N );
```

```
%
% "intersect" will succeed if the intersection of the lists Y and Z
% are not empty.
%
```

```
% ie. intersect( [a b] [b c] ) succeeds
%     intersect( [a b] [c d] ) fails
%
%
intersect( Y Z ) <-
        member( X Y )
        member( X Z );
```

```
%
% member(a [a b c]) will succeed
% member(a [d b c]) will fail
%
%
member( X [X|Y] );
member( X [Y|Z] ) <-
        member( X Z );
```

```
%
% minus( List1 List2 List3 ) returns List3 = the elements in List1 which
% do not appear in List2.
%
% ie. minus( [a b] [b c] X ) returns X = [a]
%
%
minus( [X|L1] L2 L3 ) <-
        member( X L2 )
        cut
        minus( L1 L2 L3 );
minus( [X|L1] L2 [X|L3] ) <-
        minus( L1 L2 L3 );
minus( [] L2 [] );
```

```
plan( P G ) <- empower( [] [] P G );
```

```
preserves( A [X|G] ) <-
        not( deletes( A X ) )
        preserves( A G );
preserves( A [] );
```

```
%
% "subset" succeeds if the first list is contained in the second list
%
% ie. subset([a b] [a c b d]) succeeds
%     subset([a b] [  c b d]) fails
%
%
subset( [A|X] Y ) <-
        member( A Y )
        subset( X Y );
subset( [] _ );
```

```
%
% "union" will perform the union of the first two lists and put this
% union in the third list.
%
% ie. union( [a] [b] X ) returns X = [a b]
%
%
union( [X|L1] L2 L3 ) <-
        member( X L2 )
        cut
        union( L1 L2 L3 );
union( [X|L1] L2 L3 ) <-
        union( L1 [X|L2] L3 );
union( [] L1 L1 );
```

Appendix C: Madame and Warplan-Blocks

This appendix contains the version of Madame which uses Warplan-Blocks as its planning component.

```
% setup:   Execute this predicate before using the planner, it
%          reads in the spider and the knowledge base.

setup <-
      consult( usr block/block5/.spider )  % put the spider in the aux. db
      import( kwbase block/block5 );
```

```
% laFin will rewrite all the spider axioms back into the .init file.
% This saves the modified spider.

laFin <-
      rewrite( block/block5/.spider )
      listaux( usr spi_holds )
      listaux( usr spi_no )
      listaux( usr spi_poss  )
      listaux( usr spi_parent)
      listaux( usr spi_children )
      listaux( usr spi_plan  )
      listaux( usr spi_preserves )
      close( block/block5/.spider );
```

```
% interface:      This is the interface between madame
%           and the outside world.
%           It formats the input and output for madame.
%
% interface( Goals Plan )
%
% Input:   Goals - list of goals
%
% Output: Plan  - Plan which produces a state that satisfies
%                 all the goals.
%                 Plan = result(Act1 result(Act2 ... result(Actn 0)...))


interface( [] [] ) <-     % need no plan to satisfy an empty set of goals
      cut;

interface( Goals Plan ) <-
      is_list( Goals )
      madame( Goals Plan );
```

```
% madame:      madame controls the planner by calling the procedures in
```

```
%          the proper order so as to construct a plan.
%
% madame( Goals Plan )
%
% Input:   Goals - non-empty list of goals
%
% Output: Plan  - Plan which produces a state that satisfies
%                 all the goals.
%                 Plan = result(Act1 result(Act2 ... result(Actn 0)...))


madame( Goals Plan ) <-
      find_closest( Goals 0 GoalsLeft ClosestPSS ) % Find the closest p-s-s.
      build_plan( Goals GoalsLeft ClosestPSS Plan )% Build a plan starting
                                        % from the closest p-s-s.
      add_plan( Plan ClosestPSS );              % Update the spider with
                                        % this new plan.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Find the closest p-s-s %%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% find_closest:  find the closest p-s-s to the set of goals
%
% find_closest( Goals Root GoalsLeft ClosestPSS )
%
% Input:   Goals      - non-empty list of goals
%          Root       - only the Root and its descendants will be
%                       considered as candidate p-s-s's
%
% Output:        GoalsLeft  - list of goals which don't hold in the
%                    "ClosestPSS"
%          ClosestPSS - integer representing the
%                       closest p-s-s


find_closest( Goals Root GoalsLeft ClosestPSS ) <-
      assert( usr closest(9999 9999 []) [] )  % this axiom will be used to
                                        % keep track of the closest pss
                                        % found so far
      look_all( Goals Root )                % look at all the p-s-s's to
                                        % find the closest
      retract( usr closest(ClosestPSS Dist GoalsLeft) )
      nl display("The closest p-s-s is:   ") display(ClosestPSS)
      nl display("The distance is:        ") display(Dist)
      nl display("The goals which do not hold in the p-s-s are: ")
      nl display(GoalsLeft) nl;
```

```
% look_all:      look at all the p-s-s's to find the closest one to the set
%          of goals.
%          A depth-first search of the subtree rooted at "Root" is done.
%
% look_all( Goals Root )
%
% Input:  Goals - non-empty list of goals
%          Root  - root of a subtree (it is an integer identifying a p-s-s)


look_all( Goals Root ) <-                          % It is no use to keep on
                                                   % looking
     prove( usr closest(ClosestPss 0 []) )         % if we find a p-s-s where
     cut;                                          % all the "Goals" hold.


look_all( Goals Root ) <-
     check_min( Goals Root )                       % Check if this is the
                                                   % closest p-s-s so far.
     prove( usr spi_children(Children Root) )      % Get its children.
     look_all_list( Goals Children );              % Now check the children.
```

```
% look_all_list: Break up the "List" and call "look_all" with the
%                individual elements in the list.
%
% look_all_list( Goals List )
%
% input:          Goals  - non-empty list of goals
%                 List   - list of p-s-s numbers


look_all_list( Goals List ) <-                     % It is no use to keep on
     prove( usr closest(ClosestPss 0 []) )         % looking if we find a p-s-s
     cut;                                          % where all the "Goals" hold.


look_all_list( Goals [] );


look_all_list( Goals [Pss|Rest] ) <-
     look_all( Goals Pss )
     look_all_list( Goals Rest );
```

```
% check_min:   Find the distance between the set of goals and the p-s-s,
%          then check if this is the closest p-s-s found so far.
%
% check_min( Goals PSS )
%
% Input:   Goals - nom-empty list of goals
%          PSS   _ integer identifying a p-s-s


check_min( Goals PSS ) <-
     distance( Goals PSS 0 Dist GoalsLeft )  % find the distance
                                             % between the
                                             % set of goals and the p-s-s
     is_it_closer( PSS Dist GoalsLeft );     % is this the closest p-s-s so
                                             % far?
```

```
% distance:      Find the distance between the set of goals and the p-s-s
%          The distance will be the number of elements in the list
%          "GoalsLeft". Therefore, if the distance is small, then
%          a lot of goals are satisfied in the p-s-s.
%
% distance( Goals PSS Count Dist GoalsLeft )
%
% Input:       Goals - non-empty list of goals
%          PSS   - integer identifying a p-s-s
%          Count - initialized to zero, used to calculate the distance
%
% Output:      Dist  - distance between "Goals" and "PSS"
%          GoalsLeft - list of goals which don't hold in PSS


distance( [] PSS Count Count [] );                  % No more goals to look
                                                    % at.

distance( [H|Goals] PSS Count Dist GoalsLeft ) <-
     prove( usr spi_holds(H PSS) )                  % See if H holds in the
                                                    % p-s-s.
     cut
     distance( Goals PSS Count Dist GoalsLeft );    % Yes it does, look at
                                                    % the rest of the goals.

distance( [H|Goals] PSS Count Dist [H|GoalsLeft] ) <-  % H does not
                                                    % hold in PSS.
     plus( Count 1 CountP1 )                        % Add 1 to Count.
     distance( Goals PSS CountP1 Dist GoalsLeft );  % Look at the rest
                                                    % of the goals.
```

```
% is_it_closer    Check if this is the closest p-s-s found so far.  If it is,
%          update the "closest" axiom in the aux. database.
%
% is_it_closer( Pss Dist GoalsLeft )
%
% Input:   Pss          - candidate p-s-s (integer)
%          Dist         - distance from the set of Goals to "Pss"
%          GoalsLeft    - list of goals which don't hold in "Pss"


is_it_closer( Pss Dist GoalsLeft ) <-
     prove( usr closest(ClosestPss ClosestDist X) )  % Get the closest p-s-s.
     lt( Dist ClosestDist )                          % Is this p-s-s closer.
     cut                                        % Yes it is.
     retract( usr closest(ClosestPss ClosestDist X) )
     assert( usr closest(Pss Dist GoalsLeft) [] );

is_it_closer( Pss Dist GoalsLeft ) <-
     prove( usr closest(ClosestPss ClosestDist X) )  % Get the closest p-s-s.
     eq( Dist ClosestDist )                          % Are they equidistant?
     len_plan( ClosestPss 0 Clength )                     % Check which has the
     len_plan( Pss        0 Length  )                     % simplest plan.
     lt( Length Clength )                            % Is plan to "Pss"
                                                %  shorter?
     cut                                        % We now have a closer
                                                %  p-s-s.
     retract( usr closest(ClosestPss ClosestDist X) )
     assert( usr closest(Pss Dist GoalsLeft) [] );

is_it_closer( Pss Dist GoalsLeft );                  % This is not the
                                                %  closest p-s-s so far.
```

```
% len_plan:      Find the number of actions in the plan which takes you
%          from the start state to "Pss".  ie find the path length
%          in the spider.  The spider is traversed backwards,
%          from the "Pss" we travel up towards the root.
%
% len_plan( Pss Count Length )
%
% Input:   Pss    - integer
%          Count  - initialized to zero, used as temporary counter
%
% Output: Length - # of actions in plan from start state to "Pss"


len_plan( 0 Count Count ) <-                    % We have reached start state.
     cut;

len_plan( Pss Count Length ) <-
```

```
    prove( usr spi_plan(X Len Pss) )    % Get length from parent p-s-s to
                                        % "Pss".
    plus( Count Len Sum )
    prove( usr spi_parent(ParPss Pss) ) % Get parent p-s-s.
    len_plan( ParPss Sum Length );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Build a plan %%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% build_plan:   build a plan which takes you from the start state to the
%          goal state.
%
% build_plan( Goals GoalsLeft ClosestPss Plan )
%
% Input:   Goals - non-empty list of goals
%          GoalsLeft  - list of goals which don't hold in the
%                   "ClosestPss"
%          ClosestPss - the closest p-s-s
%
% Output: Plan - Plan which takes you from the start
%                state to the goal state
%                Plan = result(Act1 result(Act2 ... result(Actn 0)...))
```

```
build_plan( Goals GoalsLeft ClosestPss Plan ) <-
    plan_to_pss( ClosestPss [] Plan_to_Pss )      % get the plan which
                                                  % takes you from the
                                                  % start state to the
                                                  % ClosestPss
    convert_plan( Plan_to_Pss 0 Con_plan )        % Convert the plan to
                                                  % Kowalski form
    minus( Goals GoalsLeft ProtectedGoals )       % Get the goals which
                                                  % are already satisfied
    empower(Con_plan ProtectedGoals Plan GoalsLeft); % WARPLAN
```

```
% convert_plan:      Convert a list of actions into a plan in Kowalski's
%              representation.  i.e. convert [Act1 Act2] to
%              result( Act2 result( Act1 0 ))
%
% convert_plan( InPlan TempPlan OutPlan )
%
% Input:       InPlan    - list of actions
%              TempPlan  - initialized to zero
```

```
%
% Output:       OutPlan   - "InPlan" converted to Kowalski's repres.


convert_plan( [] Plan Plan );

convert_plan( [Act|InPlan] TempPlan OutPlan ) <-
      convert_plan( InPlan    result( Act TempPlan )    OutPlan );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% WARPLAN-BLOCKS %%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% empower:      Build a plan to satisfy a list of goals.
%
% empower( PlanSoFar GoalsSoFar Plan GoalsLeft )
%
% Input:   PlanSoFar - This is the plan we have so far.
%                      PlanSoFar =
%                      result(Act1 result(Act2 ...result(Actn 0)...))
%          GoalsSoFar      - This list of goals hold in the state
%                      produced by "PlanSoFar".
%          GoalsLeft  - List of goals that we have to work at.
%
% Output: Plan        - This is the plan we are looking for.
%                      The goals in "GoalsSoFar" union
%                      "GoalsLeft" will hold in the state
%                      produced by "Plan".
%                      Plan =
%                      result(Act1 result(Act2 ...result(Actn 0)...))


empower( Plan Goals Plan [] );                    % no more goals to
                                                  % look at

empower( PlanSoFar GoalsSoFar Plan [Goal|GoalsLeft] ) <-
      empower1( PlanSoFar GoalsSoFar P1 Goal )    % work on the
                                                  % first goal
      empower(  P1 [Goal|GoalsSoFar] Plan GoalsLeft );  % work on the
                                                  % rest of the goals
```

```
% empower1:    empower1 is the same as empower except that there is
%              a single goal left to satisfy instead of a list of them.
%
% empower1( PlanSoFar GoalsSoFar Plan Goal )
%
% Input:  PlanSoFar - This is the plan we have so far.
%         GoalsSoFar      - This list of goals hold in the state
%                         produced by "PlanSoFar".
%         Goal        - goal that we have to work at.
%
% Output: Plan        - This is the plan we are looking for.
%                       The goals in "GoalsSoFar" union
%                       "Goal" will hold in the state
%                       produced by "Plan".


empower1( Plan ListOfGoals Plan [] ) <-          % No work required.
     cut;



% Try and catch impossible goals like diff(a a).

empower1( Plan1 ListOfGoals Plan2 Goal ) <-
     has_no_var( Goal )                  % Make sure that all variables
                                    % in "Goal" are bound.
     prove( kwbase only(Goal) )          % Is it a "Goal" with no
                                    % alternatives?
     not( prove(kwbase Goal) )           % "Goal" better be true.
     cut                            % "Goal" is false.
     fail;



% Check to see if "Goal" is already contained in "ListOfGoals".
% If it is, no planning is required.

empower1( Plan ListOfGoals Plan Goal ) <-
     has_no_var(Goal)                         % Make sure that all
                                         % variables
                                         % in "Goal" are bound.
     not(not(is_member(Goal ListOfGoals)))    % Is "Goal" contained
                                         % in "ListOfGoals"?
     cut;                                % yes



% Return the existing "Plan" because "Goal" holds in the state produced
% by "Plan".

empower1( Plan ListOfGoals Plan Goal ) <-
     already_holds( Goal Plan );
```

```
% empowerment by extension

empower1( PlanSoFar GoalsSoFar result(Action RestPlan) Goal ) <-
      clause( kwbase holds( Goal result(Action W)) Z )
                                  % Check if this "Action"
                                  % adds "Goal".
      not( is_var(Action) )           % Make sure we don't have
                                  % the frame axiom.
      conserves( Action GoalsSoFar )        % Make sure "Action" does
                                  % not delete any of the
                                  % protected goals.
      precond( Action PrecAct )       % Get the list of
                                  % preconditions for
                                  % "Action".
      not( inconsistent(PrecAct GoalsSoFar) )   % Check for conflicts
                                  % between the two sets
                                  % of goals.
      empower( PlanSoFar GoalsSoFar RestPlan PrecAct )
                                  % Modify the plan so that
                                  % the preconditions hold.
      conserves( Action GoalsSoFar );       % Do the check again
                                  % because some of the
                                  % variables might have
                                  % been bound.



% empowerment by insertion

empower1( result(Act Plan) ListGoals result(Act ModPlan) Goal ) <-
      not( deletes(Act Goal) )        % Make sure "Act" does not
                                  % delete "Goal".
      addlist( Act [] AddAct )        % Get the addlist for "Act".
      precond( Act PreAct )           % Get the preconditions for
                                  % "Act".
      minus( ListGoals AddAct List1 )       % Remove the addlist from
                                  % the protected goals.
      union( PreAct List1 List2 )     % Add the preconditions to
                                  % the protected goals.
      empower1( Plan List2 ModPlan Goal )   % Perform the insertion.
      not( deletes(Act Goal) )        % Repeat the check because
                                  % some variables might
                                  % have been bound.
      cut
      not( equiv(Plan ModPlan) );     % Make sure plans are
                                  % different.
```

```
% has_no_var(X) :    Succeeds if "X" contains no variables.
%                   "X" is not a list.
```

```
has_no_var( X ) <-
      cons( List X )                % break "X" into a list
      not( has_list_var(List) );    % make sure list contains no variables
```

```
% has_list_var(List) :        Succeds if "List" contains a variable.

has_list_var( [X|List] ) <-
      is_var( X )
      cut;

has_list_var( [X|List] ) <-
      has_list_var( List );
```

```
% is_member( X List ) :          Succeeds if "X" is a member of "List".
%                       "X" has no unbound variables.  The variables
%                       in "List" will be bound.

is_member( X List ) <-
      instantiate( List 0 )
      member( X List )
      cut;
```

```
% already_holds:      Succeeds if "Goal" holds in the state produced
%                by "Plan".
%
% already_holds( Goal Plan )


already_holds( Goal Plan ) <-
      prove( kwbase Goal );          % "Goal" is a state-independent
                                     % assertion.

already_holds( Goal Plan ) <-
      prove( kwbase holds(Goal Plan) ); % Use the frame axiom.
```

```
% conserves:     Succeeds if all the goals in "ListOfGoals" are not deleted
%           by "Action".
%
% conserves( Action ListOfGoals )
%
% Input:   Action
%           ListOfGoals
%


conserves( Action [Goal|List] ) <-
      not( deletes(Action Goal) )  % Make sure "Action" does not delete
                                   % "Goal".
      conserves( Action List );


conserves( Action [] );
```

```
% deletes: Succeeds if "Action" deletes "Goal".
%
% deletes( Action Goal )
%
% Input:   Action - single action
%           Goal   - single goal


deletes( Action Goal ) <-
      has_no_var( Goal )          % Make sure all variables in "Goal"
                                  % are bound.
      instantiate( [Action] 0 )     % Bind all variables in "Action".
      not( prove( kwbase preserves(Action Goal) ) );
```

```
% precond:       The preconditions for "Action" are put
%           in the list "PrecAct".
%
% precond( Action PrecAct )
%
% Input:   Action   - single action
%
% Output: PrecAct  - list of preconditions for "Action".
```

```
precond( Action PrecAct ) <-
      retrieve( kwbase poss(result(Action X)) List )   % Get the list of
                                                       % preconditions.
      prune( List PrecAct );                           % Get rid of any
                                                       % unwanted
                                                       % preconditions.
```

```
% prune:   Gets rid of any unwanted preconditions and does some
%          editing of preconditions.
%
% prune( InList OutList )
%
% Input:   InList   - list of preconditions
%
% Output: OutList  - pruned list of preconditions.


prune( [holds(Stat X)|InList] [Stat|OutList] ) <-
      cut
      prune( InList OutList );     % for each "holds" only keep "Stat".


prune( [poss(X)|InList] OutList ) <-
      cut
      prune( InList OutList );     % get rid of "poss"


prune( [X|InList] [X|OutList] ) <-
      prune( InList OutList );


prune( [] [] );
```

```
inconsistent( C G ) <-
      prove( kwbase imposs(S) )
      not(not( intersect( C S ) ))
      append( C G G1 )
      instantiate( G1 0 )
      subset( S G1 );
```

```
% "intersect(Y Z)" will succeed if the intersection of the lists Y and Z
```

```
% are not empty.
%
% ie. intersect( [a b] [b c] ) succeeds
%     intersect( [a b] [c d] ) fails
%
%


intersect( Y Z ) <-
      member( X Y )
      member( X Z );
```

```
% "subset" succeeds if each element in  the first list is:
%         1) contained in the second list
%         OR
%         2) can be proven.
%
% ie. subset([a b] [a c b d]) succeeds
%     subset([a b] [  c b d]) fails


subset( [A|X] Y ) <-
       member( A Y )
       subset( X Y );

subset( [A|X] Y ) <-
      not( is_var(A) )
      prove( kwbase A )
      subset( X Y );

subset( [] _ );
```

```
equiv( [] [] );

equiv( result(P1 P0) result(P3 P2) ) <-
    . cut
      equiv( P0 P2 )
      equiv( P1 P3 );

equiv( [] P2 ) <-
      cut
      fail;

equiv( P1 [] ) <-
      cut
```

```
        fail;

equiv( P1 P2 ) <-
        instantiate( [P1 P2] 0 )
        eq(P1 P2);
```

```
%
% Initially N1 = 0 and the first argument is a list.
%
% "instantiate" will bind all variables to the form X=qqq(int) where
% int is a unique integer.
%
%
instantiate( [X|L] N1 ) <-
        cons( [Y|Z] X )
        instargs( Z N1 N2 )
        instantiate( L N2 );
instantiate( [] N );
```

```
%
% "cons" breaks the second argument into a list.
%
% ie.  cons( X foo(a) ) returns X = [foo a]
%      cons( X a      ) returns X = [a]
%
%
cons( [F] F ) <-
        atomic( F )
        cut;
cons( L F ) <-
        functor( F L );
```

```
instargs( [X|L] N1 N3 ) <-
        inst( X N1 N2 )
        instargs( L N2 N3 );
instargs( [] N N );
```

```
inst( qqq(N1) N1 N2 ) <-
```

```
        cut
        add( N1 1 N2 );
inst( _1 N N ) <-
        is_atom( _1 )
        cut;
inst( _1 N N2 ) <-
        cons( [X|Y] _1 )
        instargs( Y N N2 );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% Add plan to spider %%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% add_plan:     Update the spider with the new plan.
%
% add_plan( Plan ClosestPss )
%
% Input:  Plan       - Plan which takes you from the start state
%                      to the goal state.  ie. in the form:
%                          result(Act1 result(Act2 ... result(Actn 0)...))
%         ClosestPss - The closest p-s-s to the set of goals.


add_plan( 0 ClosestPss ) <-
      cut
      nl
      display( "Spider not modified - Goals hold in start state" )
      nl;


add_plan( Plan ClosestPss ) <-
      convert_to_list( Plan List )
      reverse_order( List RList )
      add_plan1( RList ClosestPss );
```

```
% add_plan1:    Update the spider with the new plan.
%
% add_plan1( List ClosestPss )
%
% Input:  List       - List of actions representing the plan.
%                      The list of actions take you from the start
%                      state to the goal state.
%         ClosestPss  - Integer representing closest p-s-s.
```

```
% add a child to a p-s-s

add_plan1( List ClosestPss ) <-
      plan_to_pss( ClosestPss [] ListPss )      % Get the plan which takes
                                                % you from the start state
                                                % to the "ClosestPss".
      remove_subset( List ListPss List2 )       % See if any actions were
                                                % inserted in the plan to
                                                % get to the "ClosestPss".
      cut                                       % All we did was add actions
                                                % onto the "ClosestPss".
      add_plan2( ClosestPss List2 );



% Add a child to the start state because the planning stage modified the
% plan between the start state and the "ClosestPss".

add_plan1( List ClosestPss ) <-
      add_plan2( 0 List );
```

```
% add_plan2:    Add a new p-s-s to the spider.
%
% add_plan2( Parent Plan )
%
% Input:  Parent  - Integer representing the parent of the new p-s-s.
%                   Parent is an existing p-s-s.
%
%         Plan    - List of actions which take us from "Parent" to
%                   the new p-s-s.


add_plan2( Parent [] ) <-
      cut
      nl
      display( "Spider not modified - Goals hold in the closest p-s-s" )
      nl;



% Write some new spider axioms to the aux. db. for this new p-s-s and
% update the parent.

add_plan2( Parent Plan ) <-
      get_new_pss_no( Pss )
```

```
        nl
        display( "The Spider has been modified" )
        nl
        display( "    New p-s-s added is:     " ) display( Pss )
        display( "    It's parent is:         " ) display( Parent )
        nl
        display( "    The list of actions which take us from the parent to ")
        display(     "the new p-s-s is: " )
        nl
        display( "    " ) display( Plan )
        nl

%       write the axioms for the new p-s-s

        assert( usr spi_poss( Pss ) [] )
        assert( usr spi_parent(Parent Pss) [] )
        assert( usr spi_children([] Pss) [] )
        count_list( Plan 0 Count )
        assert( usr spi_plan(Plan Count Pss) [] )
        assert_spi_holds( Plan Pss )
        assert_spi_preserves( Plan Pss )

%       update the parent

        retract( usr spi_children(Children Parent) )
        assert( usr spi_children([Pss|Children] Parent) [] );
```

```
% convert_to_list:     Convert "Plan" to a list of actions.
%
% convert_to_list( Plan List )
%
% Input:          Plan  - Plan in Kowalski form, ie. Plan =
%                       result(Act1 result(Act2 ... result(Actn 0)...))
%
% Output:         List  - list of actions contained in "Plan".
%                       List = [Act1 Act2 ... Actn]


convert_to_list( 0 [] );

convert_to_list( result(Act State) [Act|Rest] ) <-
      convert_to_list( State Rest );
```

```
% reverse_order:Reverse the order of the input list.
%
```

```
% reverse_order( In Out )
%
% Input:   In    - list
%
% Output: Out   - "In" in reverse order.


reverse_order( [] [] );

reverse_order( [Hd|Tl] Z ) <-
      reverse_order( Tl Y )
      append( Y [Hd] Z );
```

```
% remove_subset:      Remove "List2" from the beginning of "List1".
%
% remove_subset( List1 List2 List3 )
%
% Input:          List1
%                 List2
%
% Output:         List3 -  List3 = List1 minus List2


remove_subset( List [] List );


remove_subset( [X|List1] [X|List2] List3 ) <-
      remove_subset( List1 List2 List3 );
```

```
% get_new_pss_no:    Get the next available p-s-s identification number
%                   and update "spi_no".
%
% get_new_pss_no( Pss )
%
% Output:         Pss - Next available p-s-s number.


get_new_pss_no( Pss ) <-
      retract( usr spi_no(Pss) )
      add( Pss 1 PssP1 )
      assert( usr spi_no(PssP1) [] );
```

```
% count_list:    Count the number of elements in a list.
%
% count_list( List Temp Count )
%
% Input:  List  - list to be counted
%         Temp  - initialized to zero, temporary counter
%
% Output: Count - Number of elements in "List".


count_list( [] Count Count );

count_list( [Hd|Tl] Temp Count ) <-
      add( Temp 1 TempP1 )
      count_list( Tl TempP1 Count );
```

```
% assert_spi_holds:   Assert the "spi_holds" axioms for "Pss".
%                    They are asserted in the aux. db. of usr.
%
% assert_spi_holds( Plan Pss )
%
% Input:        Plan  - List of actions which take us from the
%                       parent of "Pss" to "Pss".
%               Pss   - Integer representing the new p-s-s being added.


assert_spi_holds( [] Pss );

assert_spi_holds( [Action|Plan] Pss ) <-
      addlist( Action [] Addlist )              % Get the addlist for
                                                % "Action".
      remove_from_addlist( Plan Addlist Newlist )   % Remove
                                                % the members
                                                % of the addlist which
                                                % are deleted by future
                                                % actions in the "Plan"
      assert_spi_holds1( Newlist Pss )          % Write a "spi_holds"
                                                % axiom for each member
                                                % of "Newlist".
      assert_spi_holds( Plan Pss );
```

```
% assert_spi_holds1:  Assert a new "spi_holds" axiom for each member
%                    of "List".
%
% assert_spi_holds1( List Pss )
%
```

```
% Input:         List  - List of statements to be asserted with a
%                       "spi_holds".
%               Pss   - Integer representing a p-s-s.


assert_spi_holds1( [Statement|List] Pss ) <-
      clause( usr spi_holds(Statement Pss) [] )
      cut         % this "Statement" has already been asserted.
      assert_spi_holds1( List Pss );


assert_spi_holds1( [Statement|List] Pss ) <-
      assert( usr spi_holds( Statement Pss ) [] )
      assert_spi_holds1( List Pss );


assert_spi_holds1( [] Pss );
```

```
% remove_from_addlist:    Remove the statements in the
%               "Addlist" which are
%               deleted by actions in "Plan".
%
% remove_from_addlist( Plan Addlist Newlist )
%
% Input:        Plan      - List of actions.
%               Addlist   - The addlist of an action.
%
% Output:       Newlist   - List containing the members of "Addlist"
%                           which are not deleted by the actions
%                           in "Plan".


remove_from_addlist( [] Addlist Addlist ) <-
      cut;

remove_from_addlist( Plan [] [] ) <-
      cut;

remove_from_addlist( [Action|Plan] Addlist Newlist ) <-
      remove_from_addlist1( Action Addlist Newlist1 )  % See which
                                                       % statements are deleted by
                                                       % "Action".
      remove_from_addlist( Plan Newlist1 Newlist );
```

```
% remove_from_addlist1:   Remove the statements in "Addlist" which are
%                 deleted by "Action".
%
% remove_from_addlist1( Action Addlist Newlist )
%
% Input:   Action  - a single action
%          Addlist - The addlist of an action.
%
% Output: Newlist - List containing the members of "Addlist" which
%                   are not deleted by "Action".


remove_from_addlist1( Action [] [] );

remove_from_addlist1( Action [Stat|Addlist] [Stat|Newlist] ) <-
     prove( kwbase preserves(Action Stat) )          % "Action" does not
                                          % delete "Stat".
     cut
     remove_from_addlist1( Action Addlist Newlist );

remove_from_addlist1( Action [Stat|Addlist] Newlist ) <-
     remove_from_addlist1( Action Addlist Newlist ); % "Stat" is deleted
                                          % by "Action".
```

```
% assert_spi_preserves:          Assert the "spi_preserves"
%                       axioms for "Pss".
%
% NOTE:       "assert_spi_holds" must be executed before this procedure
%          because "assert_spi_preserves" uses the "spi_holds" axioms
%          asserted by "assert_spi_holds".
%
% assert_spi_preserves( Plan Pss )
%
% Input:   Plan   - List of actions which take us from the parent of
%                   "Pss" to "Pss".
%          Pss    - Integer representing the new p-s-s being added.


assert_spi_preserves( Plan Pss ) <-
     delete_list( Plan DelList )              % Get the delete list
                                          % for all the
                                          % actions in the plan.
     remove_from_deletelist( Pss DelList DeleteList ) % Remove elements
                                          % from the delete list
                                          % which are later
                                          % added by other
                                          % actions.
     assert_spi_preserves1( DeleteList Pss );  % Assert everything
                                          % in "DeleteList".
```

```
% assert_spi_preserves1:    Assert everything in "DeleteList"
%                          using one "spi_preserves" axiom.
%
% assert_spi_preserves1( DeleteList Pss )
%
% Input:  DeleteList  - List of deleted statements.
%         Pss         - int. repres. a p-s-s


assert_spi_preserves1( DeleteList Pss ) <-
     convert( List1 DeleteList )    % Convert "DeleteList" so that it
                                    % contains diff statements.
     make_same( List1 X )           % Make all the variables in "List1"
                                    % the same.
     assert( usr spi_preserves(X Pss) List1 );
```

```
% delete_list:    Build the delete list of all the actions in "Plan".
%
% delete_list( Plan DeleteList )
%
% Input:  Plan       - List of actions.
%
% Output: DeleteList - Delete list of all the actions in "Plan".


delete_list( [] [] );

delete_list( [Action|Plan] DeleteList ) <-
     clause( kwbase preserves(Action X) List1 )   % get the delete list
                                                  % for "Action".
     convert( List1 List2 )                       % Edit the delete list.
     delete_list( Plan List3 )                    % Process the rest of
                                                  % the actions.
     union( List2 List3 DeleteList );
```

```
% convert: Convert between:
%               In = [diff(X1 Y1)  ...  diff(Xn Yn)]
%         and:
%               Out = [Y1  ...  Yn]
%
% convert( In Out )
```

```
convert( [] [] );

convert( [diff(X Y)|In] [Y|Out] ) <-
      convert( In Out );

-------------------------------------------


% remove_from_deletelist:  Remove elements from "DeleteList" which are
%                          later added by other actions.
%
% remove_from_deletelist( Pss DeleteList OutList )
%
% Input:   Pss         - int. repres. a p-s-s
%          DeleteList  - List of deleted statements.
%
% Output: OutList      - "DeleteList" minus some elements.


remove_from_deletelist( Pss [] [] );


remove_from_deletelist( Pss [Statement|DeleteList]
      [Statement|OutList] ) <-
      not( clause(usr spi_holds(Statement Pss) []) )    % "Statement" is not
                                                        % added.
      cut
      remove_from_deletelist( Pss DeleteList OutList );


remove_from_deletelist( Pss [Statement|DeleteList] OutList ) <-
      remove_from_deletelist( Pss DeleteList OutList );  % "Statement" is
                                                        % added.

-------------------------------------------


% make_same:  Given a list:
%                List = [diff(X1 1)  ...  diff(Xn n)]
%          make all the Xi variables equal to "Y".
%
% make_same( List Y )
%
% Input:   List
%          Y


make_same( [diff(X Z)|Rest] Y ) <-
      eq( X Y )
      make_same( Rest Y );
```

```
make_same( [] Y );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Utilities %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

utilities(X);    % this is a dummy axiom
```

```
% addlist: Gets the addlist for a particular action.
%
% addlist( Action Temp List )
%
% Input:   Action  - single action
%          Temp    - initialized to [], temporary list
%
% Output: List    - contains the addlist for "Action"


addlist( Action Temp List ) <-
      clause( kwbase holds(X result(Action W)) Z )   % Get an
                                        % added statement.
      not( is_var(X) )                  % Make sure we don't
                                        % have the frame axiom.
      not( member(X Temp) )                     % have we looked at this
                                        % one before?
      cut                               % no
      addlist( Action [X|Temp] List );


addlist( Action List List );
```

```
%
% "append" appends the first two lists together to produce the third
% argument.
%
% ie. append( [a b] [c d] X ) returns X = [a b c d]
%
%
append( [] Y Y );
append( [U|X] Y [U|Z] ) <-
       append( X Y Z );
```

```
%
% member(a [a b c]) will succeed
% member(a [d b c]) will fail
%
%

member( X [X|Y] );

member( X [Y|Z] ) <-
        member( X Z );
```

```
%
% minus( List1 List2 List3 ) returns List3 = the elements in List1 which
% do not appear in List2.
%
% ie. minus( [a b] [b c] X ) returns X = [a]
%
%
minus( [X|L1] L2 L3 ) <-
        member( X L2 )
        cut
        minus( L1 L2 L3 );
minus( [X|L1] L2 [X|L3] ) <-
        minus( L1 L2 L3 );
minus( [] L2 [] );
```

```
% plan_to_pss:  Get the plan which takes you from the start state to
%           "Pss". The spider is traversed backwards, from the
%           "Pss" we travel up towards the root.
%
% plan_to_pss( Pss Temp Plan )
%
% Input:    Pss   - integer representing a p-s-s.
%           Temp  - initialized to [], used as a temporary list
%
% Output: Plan  - List of actions which take you from the start
%                 state to "Pss".


plan_to_pss( 0 Plan Plan ) <-              % Reached the start state.
      cut;
```

```
plan_to_pss( Pss Temp Plan ) <-
      prove( usr spi_plan(Plan1 N Pss) )    % Get the plan which got us to
                                            % this "Pss".
      append( Plan1 Temp Plan2 )
      prove( usr spi_parent(ParPss Pss) )   % Get the parent.
      plan_to_pss( ParPss Plan2 Plan );
```

```
%
% "union" will perform the union of the first two lists and put this
% union in the third list.
%
% ie. union( [a] [b] X ) returns X = [a b]
%
%
union( [X|L1] L2 L3 ) <-
      member( X L2 )
      cut
      union( L1 L2 L3 );
union( [X|L1] L2 L3 ) <-
      union( L1 [X|L2] L3 );
union( [] L1 L1 );
```

Appendix D: Madame and Warplan-Not

This appendix contains the version of Madame which uses Warplan-Not as its planning component. An axiomatization of the UNIX domain which is used with this version of Madame is contained in Appendix A.

```
% setup:   Execute this predicate before using the planner, it
%          reads in the spider and the knowledge base.

setup <-
      consult( usr unix_axioms/unix_not/.spider )  % Put the
                                                   % spider in the aux db.
      import( kwbase unix_axioms/unix_not );
```

```
% laFin will rewrite all the spider axioms back into the .spider file.
% This saves the modified spider.

laFin <-
      rewrite( unix_axioms/unix_not/.spider )
      listaux( usr spi_holds )
      listaux( usr spi_no )
      listaux( usr spi_poss  )
      listaux( usr spi_parent)
      listaux( usr spi_children )
      listaux( usr spi_plan  )
      listaux( usr spi_preserves )
      close( unix_axioms/unix_not/.spider );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% Interface %%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% interface:      This is the interface between the planner
%          and the NL front-end.
%          It formats the input and output for madame.
%
% interface( Stats Goals Plan )
%
% Input:   Stats  - "Stats" will be a list of the form:
%                     [ holds( Stat1, Start_State )
%                           .
%                           .
%                           .
%                       holds( Statn, Start_State ) ]
%          Goals  - "Goals" will be a list of the form:
%                     [ Goal1 ... Goaln ]
%
% Output: Plan   - Plan which produces the goal state.
%                  Plan = result(Act1 result(Act2 ... result(Actn 0)...))
```

```
interface( Stats [] 0 ) <-          % no goals, Plan = 0
      cut;

interface( Stats Goals Plan ) <-
      nl(3)
      display("The list of goals to satisfy is:")
      nl
      display(Goals)
      nl(3)

      clean_up                % Clean up the aux. db. of usr.
      assert_usr( Stats )           % Assert "Stats"
                              % in the aux. db. of usr.
      madame( Goals Plan )    % Build plan.

      nl(4)
      display("**** The plan generated is: ****")
      nl(3);
```

```
% clean_up:      Gets rid of all 'holds' axioms in the aux. db. of usr.


clean_up <-
      delax( usr holds(X Y) )
      fail;

clean_up;
```

```
% assert_usr:    Assert everything in "List" into the aux. db. of usr.
%
% assert_usr( List )
%
% Input:   List = [holds(Stat1,0) ... holds(Statn,0)]


assert_usr([]);

assert_usr( [Head|Tail] ) <-
      assert( usr Head [] )
      assert_usr( Tail );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% madame %%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% madame:      madame controls the planner by calling the
%          procedures in
%          the proper order so as to construct a plan.
%
% madame( Goals Plan )
%
% Input:   Goals - non-empty list of goals
%
% Output: Plan  - Plan which produces a state that satisfies
%                 all the goals.
%                 Plan = result(Act1 result(Act2 ... result(Actn 0)...))


madame( Goals Plan ) <-
     find_closest( Goals 0 GoalsLeft ClosestPSS ) % Find the
                                        % closest p-s-s.
     build_plan( Goals GoalsLeft ClosestPSS Plan )% Build a plan
                                        % starting
                                        % from the closest p-s-s.
     add_plan( Plan ClosestPSS );              % Update the spider
                                        % with this new plan.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%% Find the closest p-s-s %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% find_closest:  find the closest p-s-s to the set of goals
%
% find_closest( Goals Root GoalsLeft ClosestPSS )
%
% Input:   Goals      - non-empty list of goals
%          Root       - only the Root and its descendants will be
%                     considered as candidate p-s-s's
%
% Output:        GoalsLeft  - list of goals which don't hold in the
%                    "ClosestPSS"
%          ClosestPSS - integer representing the
%                     closest p-s-s


find_closest( Goals Root GoalsLeft ClosestPSS ) <-
     assert( usr closest(9999 9999 []) [] )  % this axiom will be used
                                        % to keep track of the closest pss
                                        % found so far
```

```
    look_all( Goals Root )              % look at all the p-s-s's to
                                        % find the closest
    retract( usr closest(ClosestPSS Dist GoalsLeft) )

    nl(3)
    display("The closest p-s-s is:   ") display(ClosestPSS)
    nl
    display("The distance is:        ") display(Dist)
    nl
    display("The goals which do not hold in the p-s-s are: ")
    nl
    display(GoalsLeft)
    nl(3);
```

```
% look_all:      look at all the p-s-s's to find the closest one to
%         the set of goals.
%         A depth-first search of the subtree rooted at "Root"
%         is done.
%
% look_all( Goals Root )
%
% Input:  Goals - non-empty list of goals
%         Root  - root of a subtree (it is an integer identifying
%                 a p-s-s)


look_all( Goals Root ) <-                   % It is no use to keep on
                                            % looking
    prove( usr closest(ClosestPss 0 []) )   % if we find a p-s-s
    cut;                                    % where all the "Goals" hold.


look_all( Goals Root ) <-
    check_min( Goals Root )                 % Check if this is the
                                            % closest p-s-s so far.
    prove( usr spi_children(Children Root) ) % Get its children.
    look_all_list( Goals Children );        % Now check the children.
```

```
% look_all_list: Break up the "List" and call "look_all" with the
%                individual elements in the list.
%
% look_all_list( Goals List )
%
% input:         Goals  - non-empty list of goals
%                List   - list of p-s-s numbers
```

```
look_all_list( Goals List ) <-                    % It is no use to keep on
      prove( usr closest(ClosestPss 0 []) )    % looking if we find a
      cut;                                  % p-s-s where all the "Goals" hold.


look_all_list( Goals [] );


look_all_list( Goals [Pss|Rest] ) <-
      look_all( Goals Pss )
      look_all_list( Goals Rest );
```

```
% check_min:   Find the distance between the set of goals and the
%          p-s-s,
%          then check if this is the closest p-s-s found so far.
%
% check_min( Goals PSS )
%
% Input:   Goals - nom-empty list of goals
%          PSS   _ integer identifying a p-s-s


check_min( Goals PSS ) <-
      distance( Goals PSS 0 Dist GoalsLeft )  % find the distance
                                        % between the
                                         % set of goals and the p-s-s
      is_it_closer( PSS Dist GoalsLeft );     % is this the closest
                                        % p-s-s so far?
```

```
% distance:      Find the distance between the set of goals and
%          the p-s-s.
%          The distance will be the number of elements in the list
%          "GoalsLeft". Therefore, if the distance is small, then
%          a lot of goals are satisfied in the p-s-s.
%
% distance( Goals PSS Count Dist GoalsLeft )
%
% Input:       Goals - non-empty list of goals
%          PSS   - integer identifying a p-s-s
%          Count - initialized to zero, used to calculate the distance
%
% Output:      Dist  - distance between "Goals" and "PSS"
%          GoalsLeft - list of goals which don't hold in PSS


distance( [] PSS Count Count [] );                % No more goals
```

```
                                        % to look at.

distance( [not(H)|Goals] PSS Count Dist GoalsLeft ) <- % The new
                                        % goal is a negation.
      not( prove(usr spi_holds(H PSS)) )        % See if 'H' does not
                                        % hold in the p-s-s.
      cut
      distance( Goals PSS Count Dist GoalsLeft );


distance( [H|Goals] PSS Count Dist GoalsLeft ) <-
      ne( H not(X) )
      prove( usr spi_holds(H PSS) )             % See if H holds
                                        % in the p-s-s.
      cut
      distance( Goals PSS Count Dist GoalsLeft );   % Yes it does,
                                        % look at the rest of the goals.


distance( [H|Goals] PSS Count Dist [H|GoalsLeft] ) <-  % H does
                                        % not hold in PSS.
      plus( Count 1 CountP1 )               % Add 1 to Count.
      distance( Goals PSS CountP1 Dist GoalsLeft ); % Look at the
                                        % rest of the goals.
```

```
% is_it_closer   Check if this is the closest p-s-s found so far.
%         If it is,
%         update the "closest" axiom in the aux. database.
%
% is_it_closer( Pss Dist GoalsLeft )
%
% Input:  Pss         - candidate p-s-s (integer)
%         Dist        - distance from the set of Goals to "Pss"
%         GoalsLeft   - list of goals which don't hold in "Pss"


is_it_closer( Pss Dist GoalsLeft ) <-
      prove( usr closest(ClosestPss ClosestDist X) ) % Get the
                                        % closest p-s-s.
      lt( Dist ClosestDist )                % Is this p-s-s closer.
      cut                               % Yes it is.
      retract( usr closest(ClosestPss ClosestDist X) )
      assert( usr closest(Pss Dist GoalsLeft) [] );

is_it_closer( Pss Dist GoalsLeft ) <-
      prove( usr closest(ClosestPss ClosestDist X) ) % Get the
                                        % closest p-s-s.
      eq( Dist ClosestDist )                % Are they
                                        % equidistant?
```

```
      len_plan( ClosestPss 0 Clength )              % Check which
                                          % has the
      len_plan( Pss       0 Length  )               % simplest plan.
      lt( Length Clength )                    % Is plan to "Pss"
                                          % shorter?
      cut                                 % We now have a closer
                                          % p-s-s.
      retract( usr closest(ClosestPss ClosestDist X) )
      assert( usr closest(Pss Dist GoalsLeft) [] );

is_it_closer( Pss Dist GoalsLeft );           % This is not the
                                          % closest p-s-s so far.
```

```
% len_plan:      Find the number of actions in the plan which
%          takes you
%          from the start state to "Pss". ie find the path length
%          in the spider. The spider is traversed backwards,
%          from the "Pss" we travel up towards the root.
%
% len_plan( Pss Count Length )
%
% Input:  Pss    - integer
%          Count  - initialized to zero, used as temporary counter
%
% Output: Length - # of actions in plan from start state to
%                  "Pss".


len_plan( 0 Count Count ) <-                % We have reached
      cut;                          % start state.

len_plan( Pss Count Length ) <-
      prove( usr spi_plan(X Len Pss) )    % Get length from
                                    % parent p-s-s to "Pss".
      plus( Count Len Sum )
      prove( usr spi_parent(ParPss Pss) ) % Get parent p-s-s.
      len_plan( ParPss Sum Length );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%% Build a plan %%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% build_plan:    build a plan which takes you from the start
%          state to the goal state.
```

```
%
% build_plan( Goals GoalsLeft ClosestPss Plan )
%
% Input:   Goals - non-empty list of goals
%          GoalsLeft  - list of goals which don't hold in the
%                 "ClosestPss"
%          ClosestPss - the closest p-s-s
%
% Output: Plan - Plan which takes you from the start
%               state to the goal state
%               Plan = result(Act1 result(Act2 ... result(Actn 0)...))


build_plan( Goals GoalsLeft ClosestPss Plan ) <-
      plan_to_pss( ClosestPss [] Plan_to_Pss )        % get the plan
                                          % which takes you from the
                                          % start state to the
                                          % ClosestPss
      convert_plan( Plan_to_Pss 0 Con_plan )          % Convert
                                          % the plan to Kowalski form
      minus( Goals GoalsLeft ProtectedGoals )         % Get the goals
                                          % which are already satisfied
      empower(Con_plan ProtectedGoals Plan GoalsLeft);
```

```
% convert_plan:       Convert a list of actions into a plan
%                 in Kowalski's
%                 representation.  i.e. convert [Act1 Act2] to
%                 result( Act2 result( Act1 0 ))
%
% convert_plan( InPlan TempPlan OutPlan )
%
% Input:          InPlan    - list of actions
%                 TempPlan  - initialized to zero
%
% Output:         OutPlan   - "InPlan" converted to Kowalski's
%                               repres.


convert_plan( [] Plan Plan );

convert_plan( [Act|InPlan] TempPlan OutPlan ) <-
      convert_plan( InPlan    result( Act TempPlan )    OutPlan );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%% WARPLAN-NOT %%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% empower:      Build a plan to satisfy a list of goals.
%
% empower( PlanSoFar GoalsSoFar Plan GoalsLeft )
%
% Input:   PlanSoFar - This is the plan we have so far.
%                      PlanSoFar =
%                      result(Act1 result(Act2 ...result(Actn 0)...))
%          GoalsSoFar      - This list of goals hold in the state
%                      produced by "PlanSoFar".
%          GoalsLeft  - List of goals that we have to work at.
%
% Output: Plan        - This is the plan we are looking for.
%                      The goals in "GoalsSoFar" union
%                      "GoalsLeft" will hold in the state
%                      produced by "Plan".
%                      Plan =
%                      result(Act1 result(Act2 ...result(Actn 0)...))


empower( Plan Goals Plan [] );                    % no more goals to
                                                  % look at


empower( PlanSoFar GoalsSoFar Plan [Goal|GoalsLeft] ) <-
     ne( Goal not(X) )                            % 'Goal' is not a
                                                  % negation.
     empower1( PlanSoFar GoalsSoFar P1 Goal )     % work on
                                                  % the first goal
     empower(  P1 [Goal|GoalsSoFar] Plan GoalsLeft );  % work
                                                  % on the rest of the goals


empower( PlanSoFar GoalsSoFar Plan [not(Goal)|GoalsLeft] ) <-
     empower1_not(PlanSoFar GoalsSoFar P1 not(Goal))   % Work
                                                  % on the first
                                                  % goal which is a
                                                  % negation.
     empower(P1 [not(Goal)|GoalsSoFar] Plan GoalsLeft);% Work
                                                  % on the rest of the goals.

-------------------------------------


% empower1:    empower1 is the same as empower except that
%          there is
%               a single goal left to satisfy instead of a list
%          of them.
```

```
%
% empower1( PlanSoFar GoalsSoFar Plan Goal )
%
% Input:   PlanSoFar - This is the plan we have so far.
%          GoalsSoFar      - This list of goals hold in the state
%                            produced by "PlanSoFar".
%          Goal        - goal that we have to work at.
%
% Output: Plan         - This is the plan we are looking for.
%                        The goals in "GoalsSoFar" union
%                        "Goal" will hold in the state
%                        produced by "Plan".


empower1( Plan ListOfGoals Plan [] ) <-              % No work
     cut;                                 % required.


% Try and catch impossible goals like diff(a a).

empower1( Plan1 ListOfGoals Plan2 Goal ) <-
     has_no_var( Goal )                        % No vars in 'Goal'.
     prouver( only(Goal) )                     % Is it a "Goal" with no
                                          % alternatives?
     not( prouver(Goal) )                      % "Goal" better be true.
     cut                                  % "Goal" is false.
     fail;


% Check to see if "Goal" is already contained in "ListOfGoals".
% If it is, no planning is required.

empower1( Plan ListOfGoals Plan Goal ) <-
     has_no_var(Goal)                          % Make sure that all
                                          % variables
                                          % in "Goal" are bound.
     not(not(is_member(Goal ListOfGoals)))     % Is "Goal"
                                          % contained in "ListOfGoals"?
     cut;                                 % yes



% Return the existing "Plan" because "Goal" holds in the state
% produced by "Plan".

empower1( Plan ListOfGoals Plan Goal ) <-
     already_holds( Goal Plan );
```

```
% empowerment by extension

empower1( PlanSoFar GoalsSoFar result(Action RestPlan)
      Goal ) <-
    clause( kwbase holds(Goal result(Action W)) Z )
                                % Find an 'Action' which
                                % adds 'Goal'.
    not( is_var(Action) )           % Make sure we don't
                                % have the frame axiom.
    conserves( Action GoalsSoFar )          % Make sure
                                % "Action" does
                                % not delete any of the
                                % protected goals.
    precond1( Goal Action PrecAct )         % Get the list of
                                % preconditions for
                                % "Action".
    not( inconsistent(PrecAct GoalsSoFar) )   % Check for
                                % conflicts
                                % between the two sets
                                % of goals.
    empower( PlanSoFar GoalsSoFar RestPlan PrecAct )
                                % Modify the plan so that
                                % the preconditions hold.
    conserves( Action GoalsSoFar );         % Do the check
                                % again because some of the
                                % variables might have
                                % been bound.



% empowerment by insertion

empower1( result(Act Plan) ListGoals result(Act ModPlan)
      Goal ) <-
    not( deletes(Act Goal) )            % Make sure "Act" does not
                                % delete "Goal".
    addlist( Act [] AddAct )            % Get the addlist for "Act".
    precond2( Act PreAct )                  % Get the preconditions
                                % for "Act".
    minus( ListGoals AddAct List1 )         % Remove the addlist
                                % from the protected goals.
    union( PreAct List1 List2 )         % Add the preconditions to
                                % the protected goals.
    empower1( Plan List2 ModPlan Goal )     % Perform the
                                % insertion.
    not( deletes(Act Goal) )            % Repeat the check
                                % because
                                % some variables might
                                % have been bound.
    cut
    not( equiv(Plan ModPlan) );         % Make sure plans are
                                % different.
```

```
% empower1_not:     empower1_not is the same as empower1
%          except that the goal is a negation.
%
% empower1_not( PlanSoFar GoalsSoFar Plan Goal )
%
% Input:   PlanSoFar - This is the plan we have so far.
%          GoalsSoFar      - This list of goals hold in the state
%                            produced by "PlanSoFar".
%          Goal        - Negated goal that we have to work at.
%                            (ie. of the form 'not(foo(X))')
%
% Output: Plan         - This is the plan we are looking for.
%                            The goals in "GoalsSoFar" union
%                            "Goal" will hold in the state
%                            produced by "Plan".


empower1_not( Plan ListOfGoals Plan [] ) <-          % No work
     cut;                                  % required.


% Check to see if "Goal" is already contained in "ListOfGoals".
% If it is, no planning is required.

empower1_not( Plan ListOfGoals Plan not(Goal) ) <-
     has_no_var(Goal)                          % Make sure that all
                                          % variables
                                          % in "Goal" are bound.
     not(not(is_member(not(Goal) ListOfGoals)))  % Is "Goal"
                                          % contained in "ListOfGoals"?
     cut;                                 % yes



% Return the existing "Plan" because "Goal" does not hold
% in the state produced
% by "Plan".

empower1_not( Plan ListOfGoals Plan not(Goal) ) <-
     not( already_holds(Goal Plan) );



% empowerment by extension

empower1_not( PlanSoFar GoalsSoFar result(Action RestPlan)
          not(Goal) ) <-
```

```
    find_act_del( Goal Action )          % Find an 'Action' which
                                 % deletes 'Goal'.
    conserves( Action GoalsSoFar )       % Make sure
                                 % "Action" does
                                 % not destroy any of the
                                 % protected goals.
    precond2( Action PrecAct )           % Get the list of
                                 % preconditions for
                                 % "Action".
    not( inconsistent(PrecAct GoalsSoFar) )   % Check for
                                 % conflicts
                                 % between the two sets
                                 % of goals.
    empower( PlanSoFar GoalsSoFar RestPlan PrecAct )
                                 % Modify the plan so that
                                 % the preconditions hold.
    conserves( Action GoalsSoFar );      % Do the check again
                                 % because some of the
                                 % variables might have
                                 % been bound.


% empowerment by insertion

empower1_not( result(Act Plan) ListGoals result(Act ModPlan)
        not(Goal) ) <-
    not( adds(Act Goal) )                % Make sure "Act"
                                 % does not add "Goal".
    addlist( Act [] AddAct )             % Get the addlist for "Act".
    precond2( Act PreAct )               % Get the
                                 % preconditions for "Act".
    minus( ListGoals AddAct List1 )      % Remove the
                                 % addlist from
                                 % the protected goals.
    union( PreAct List1 List2 )          % Add the preconditions to
                                 % the protected goals.
    empower1_not(Plan List2 ModPlan not(Goal))  % Perform the
                                 % insertion.
    not( adds(Act Goal) )                % Repeat the check
                                 % because some variables might
                                 % have been bound.
    cut
    not( equiv(Plan ModPlan) );          % Make sure plans are
                                 % different.
```

```
% find_act_del:      Finds an action which deletes 'Goal'.
%
```

```
% find_act_del( Goal Action )
%
% Input:        Goal     - A single goal.
%
% Output:       Action   - A single action which deletes 'Goal'.


find_act_del( Goal Action ) <-
      delete_list( [Action] List )   % Get the delete list for an
                                     % action.
      member( Goal List );           % Check if "Goal" is in the
                                     % delete list.
```

```
% is_member( X List ) :          Succeeds if "X" is a member of "List".
%                       "X" has no unbound variables. The variables
%                       in "List" will be bound.


is_member( X List ) <-
      instantiate( List 0 )
      member( X List )
      cut;
```

```
% already_holds:      Succeeds if "Goal" holds in the state
%                 produced by "Plan".
%
% already_holds( Goal Plan )


already_holds( Goal Plan ) <-
      prouver( Goal );           % "Goal" is a state-independent
                                 % assertion.

already_holds( Goal Plan ) <-
      prouver( holds(Goal Plan) );      % Use the frame axiom.
```

```
% conserves:     Succeeds if all the goals in "ListOfGoals" are not
%           destroyed by "Action".
%
% conserves( Action ListOfGoals )
%
% Input:   Action
```

```
%           ListOfGoals
%


conserves( Action [not(Goal)|List] ) <-
      cut
      not( adds(Action Goal) )     % Make sure 'Action' does not
                                   % add 'Goal'
      conserves( Action List );


conserves( Action [Goal|List] ) <-
      cut
      not( deletes(Action Goal) )  % Make sure "Action" does not
                                   % delete "Goal".
      conserves( Action List );


conserves( Action [] );
```

```
% deletes: Succeeds if "Action" deletes "Goal".
%
% deletes( Action Goal )
%
% Input:   Action - single action
%          Goal   - single goal


deletes( Action Goal ) <-
      has_no_var( Goal )           % Make sure all variables in "Goal"
                                   % "Goal" are bound.
      instantiate( [Action] 0 )    % Bind all variables in "Action".
      not( prouver(preserves(Action Goal)) );
```

```
% adds:          Succeeds if 'Action' adds 'Goal'.
%
% adds( Action Goal )
%
% Input:   Action  - single action
%          Goal    - single goal


adds( Action Goal ) <-
      has_no_var( Goal )           % Make sure all variables in 'Goal'
                                   % are bound.
```

```
    instantiate( [Action] 0 )      % Bind all variables in 'Action'.
    clause( kwbase holds(Goal result(Act W)) Z )
                                   % Find an action which adds 'Goal'.
    not( is_var(Act) )             % Make sure we don't have the frame
                                   % axiom.
    eq( Act Action );              % 'Action' adds 'Goal'.
```

```
% precond1:     The preconditions for "Action" are put in the list
%          "PrecAct".
%          The preconditions are found by taking the body of
%          the following 2 axioms:
%               - The 'poss' axiom for 'Action'.
%               - The 'holds' axiom which says that 'Action' adds
%                 'Goal'. Using the body of this axiom will help
%                 to bind some of the variables in 'Goal'.
%
% precond1( Goal Action PrecAct )
%
% Input:  Goal     - The goal we are currently working on. It is
%                    added by 'Action'.
%          Action   - single action
%
% Output: PrecAct  - list of preconditions for "Action".


precond1( Goal Action PrecAct ) <-
    retrieve( kwbase poss(result(Action X)) List1 )  % Get the
                                   % list of preconditions.
    clause( kwbase holds(Goal result(X Y)) Body )    % Find
                                   % the add statement used for
                                   % 'Goal'.
    not( is_var(X) )               % Make sure we don't
                                   % have the frame axiom.
    eq( Action X )                 % Make sure we have the
                                   % right add statement.
    append( Body List1 List2 )
    prune( List2 PrecAct );                % Get rid of any
                                   % unwanted
                                   % preconditions.
```

```
% precond2:     The preconditions for "Action" are put in the list
%          "PrecAct".
%          Only the 'poss' axiom for 'Action' is looked at.
%
% precond2( Action PrecAct )
```

```
%
% Input:   Action   - single action
%
% Output: PrecAct  - list of preconditions for "Action".


precond2( Action PrecAct ) <-
      retrieve( kwbase poss(result(Action X)) List )   % Get the list of
                                                       % preconditions.
      prune( List PrecAct );                           % Get rid of any
                                                       % unwanted
                                                       % preconditions.
```

```
% prune:   Gets rid of any unwanted preconditions and does some
%          editing of preconditions.
%
% prune( InList OutList )
%
% Input:   InList   - list of preconditions
%
% Output: OutList  - pruned list of preconditions.


prune( [not(prouver(holds(Stat X)))|InList]
      [not(Stat)|OutList] ) <-
      cut
      prune( InList OutList );


prune( [prouver(holds(Stat X))|InList] [Stat|OutList] ) <-
      cut
      prune( InList OutList );     % for each "holds" only keep
                                   % "Stat".


prune( [poss(X)|InList] OutList ) <-
      cut
      prune( InList OutList );     % get rid of "poss"


prune( [Stat|InList] OutList ) <-
      prouver( replace(Stat) )     % 'Stat' is to be replaced
                                   % by its body.
      clause( kwbase Stat Body )
      append( Body InList List )
      prune( List OutList );
```

```
prune( [X|InList] [X|OutList] ) <-
      prouver( not(replace(X)) )
      prune( InList OutList );


prune( [] [] );
```

```
inconsistent( C G ) <-
       prouver( imposs(S) )
       not(not( intersect( C S ) ))
       append( C G G1 )
       instantiate( G1 0 )
       subset( S G1 );
```

```
% "intersect(Y Z)" will succeed if the
% intersection of the lists Y and Z
% are not empty.
%
% ie. intersect( [a b] [b c] ) succeeds
%     intersect( [a b] [c d] ) fails
%
%


intersect( Y Z ) <-
      member( X Y )
      member( X Z );
```

```
% "subset" succeeds if each element in  the first list is:
%         1) contained in the second list
%         OR
%         2) can be proven.
%
% ie. subset([a b] [a c b d]) succeeds
%     subset([a b] [  c b d]) fails


subset( [A|X] Y ) <-
      member( A Y )
      subset( X Y );

subset( [A|X] Y ) <-
```

```
        not( is_var(A) )  % 'A' is not a variable.
        prouver( A )
        subset( X Y );

subset( [] _ );
```

```
equiv( [] [] );

equiv( result(P1 P0) result(P3 P2) ) <-
        cut
        equiv( P0 P2 )
        equiv( P1 P3 );

equiv( [] P2 ) <-
        cut
        fail;

equiv( P1 [] ) <-
        cut
        fail;

equiv( P1 P2 ) <-
        instantiate( [P1 P2] 0 )
        eq(P1 P2);
```

```
%
% Initially N1 = 0 and the first argument is a list.
%
% "instantiate" will bind all variables
% to the form X=qqq(int) where
% int is a unique integer.
%
%
instantiate( [X|L] N1 ) <-
        cons( [Y|Z] X )
        instargs( Z N1 N2 )
        instantiate( L N2 );
instantiate( [] N );
```

```
%
% "cons" breaks the second argument into a list.
```

```
%
% ie.  cons( X foo(a) ) returns X = [foo a]
%      cons( X a      ) returns X = [a]
%
%
cons( [F] F ) <-
        atomic( F )
        cut;
cons( L F ) <-
        functor( F L );
```

```
instargs( [X|L] N1 N3 ) <-
        inst( X N1 N2 )
        instargs( L N2 N3 );
instargs( [] N N );
```

```
inst( qqq(N1) N1 N2 ) <-
        cut
        add( N1 1 N2 );
inst( _1 N N ) <-
        is_atom( _1 )
        cut;
inst( _1 N N2 ) <-
        cons( [X|Y] _1 )
        instargs( Y N N2 );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%% Add plan to spider %%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% add_plan:     Update the spider with the new plan.
%
% add_plan( Plan ClosestPss )
%
% Input:  Plan       - Plan which takes you from the start state
%                      to the goal state.  ie. in the form:
%                         result(Act1 result(Act2 ... result(Actn 0)...))
%         ClosestPss - The closest p-s-s to the set of goals.

add_plan( 0 ClosestPss ) <-
```

```
      cut
      nl
      display( "Spider not modified - Goals hold in start state" )
      nl;


add_plan( Plan ClosestPss ) <-
      convert_to_list( Plan List )
      reverse_order( List RList )
      add_plan1( RList ClosestPss );
```

```
% add_plan1:    Decide who the proud parent
%          of the new p-s-s will be.
%
% add_plan1( List ClosestPss )
%
% Input:   List         - List of actions representing the plan.
%                         The list of actions take you from the start
%                         state to the goal state.
%          ClosestPss   - Integer representing closest p-s-s.


% add a child to a p-s-s

add_plan1( List ClosestPss ) <-
      plan_to_pss( ClosestPss [] ListPss )      % Get the plan which
                                                % takes you from the start state
                                                % to the "ClosestPss".
      remove_subset( List ListPss List2 )       % See if any actions
                                                % were inserted in the plan to
                                                % get to the "ClosestPss".
      cut                                       % All we did was add actions
                                                % onto the "ClosestPss".
      add_plan2( ClosestPss List2 );



% Add a child to the start state because
% the planning stage modified the
% plan between the start state and the "ClosestPss".

add_plan1( List ClosestPss ) <-
      add_plan2( 0 List );
```

```
% add_plan2:    Add a new p-s-s to the spider.
%
% add_plan2( Parent Plan )
%
% Input:   Parent  - Integer representing the parent of the new
%                    p-s-s.  Parent is an existing p-s-s.
%
%          Plan    - List of actions which take us from "Parent" to
%                    the new p-s-s.


add_plan2( Parent [] ) <-
      cut
      nl
      display( "Spider not modified - Goals
            hold in the closest p-s-s" )
      nl;




% Write some new spider axioms to the aux. db.
% for this new p-s-s and
% update the parent.

add_plan2( Parent Plan ) <-
      get_new_pss_no( Pss )
      nl(3)
      display( "The Spider has been modified" )
      nl
      display( "    New p-s-s added is:     " ) display( Pss )
      display( "    It's parent is:         " ) display( Parent )
      nl
      display( "    The list of actions which
                        take us from the parent to ")
      display(      "the new p-s-s is: " )
      nl
      display( "    " ) display( Plan )
      nl(3)

%     write the axioms for the new p-s-s

      assert( usr spi_poss( Pss ) [] )
      assert( usr spi_parent(Parent Pss) [] )
      assert( usr spi_children([] Pss) [] )
      count_list( Plan 0 Count )
      assert( usr spi_plan(Plan Count Pss) [] )
      assert_spi_holds1( Parent Plan Pss )
      assert_spi_preserves( Plan Pss )

%     update the parent
```

```
      retract( usr spi_children(Children Parent) )
      assert( usr spi_children([Pss|Children] Parent) [] );
```

```
% convert_to_list:     Convert "Plan" to a list of actions.
%
% convert_to_list( Plan List )
%
% Input:          Plan  - Plan in Kowalski form, ie. Plan =
%                       result(Act1 result(Act2 ... result(Actn 0)...))
%
% Output:         List  - list of actions contained in "Plan".
%                       List = [Act1 Act2 ... Actn]


convert_to_list( 0 [] );

convert_to_list( result(Act State) [Act|Rest] ) <-
      convert_to_list( State Rest );
```

```
% reverse_order:Reverse the order of the input list.
%
% reverse_order( In Out )
%
% Input:   In    - list
%
% Output: Out   - "In" in reverse order.


reverse_order( [] [] );

reverse_order( [Hd|Tl] Z ) <-
      reverse_order( Tl Y )
      append( Y [Hd] Z );
```

```
% remove_subset:     Remove "List2" from the beginning of "List1".
%
% remove_subset( List1 List2 List3 )
%
% Input:          List1
%                 List2
```

```
%
% Output:       List3 - List3 = List1 minus List2


remove_subset( List [] List );


remove_subset( [X|List1] [X|List2] List3 ) <-
      remove_subset( List1 List2 List3 );
```

```
% get_new_pss_no:   Get the next available p-s-s identification
%                   number and update "spi_no".
%
% get_new_pss_no( Pss )
%
% Output:       Pss - Next available p-s-s number.


get_new_pss_no( Pss ) <-
      retract( usr spi_no(Pss) )
      add( Pss 1 PssP1 )
      assert( usr spi_no(PssP1) [] );
```

```
% count_list:    Count the number of elements in a list.
%
% count_list( List Temp Count )
%
% Input:   List   - list to be counted
%          Temp   - initialized to zero, temporary counter
%
% Output: Count  - Number of elements in "List".


count_list( [] Count Count );

count_list( [Hd|Tl] Temp Count ) <-
      add( Temp 1 TempP1 )
      count_list( Tl TempP1 Count );
```

```
% assert_spi_holds1:  Assert the "spi_holds" axioms for "Pss".
```

```
%               They are asserted in the aux. db. of usr.
%
% assert_spi_holds1( Parent Plan Pss )
%
% Input:        Parent - Integer repres. the parent of 'Pss'.
%               Plan   - List of actions which take us from
%                        'Parent' to 'Pss'.
%               Pss    - Integer representing the
%                            new p-s-s being added.


assert_spi_holds1( Parent Plan Pss ) <-
     plan_to_pss( Parent [] ListPlan )  % Get the plan which
                                        % takes us from
                                        % the start state to 'Parent'.
     reverse_order( ListPlan RPlan )
     convert_to_list( ParPlan RPlan )   % Convert the plan
                                        % from a list to
                                        % the form containing 'result'.
     assert_spi_holds2( ParPlan Plan Pss );
```

```
% assert_spi_holds2:  Assert the "spi_holds" axioms for "Pss".
%               They are asserted in the aux. db. of usr.
%
% assert_spi_holds2( ParentPlan Plan Pss )
%
% Input:        ParentPlan - Plan which takes us from the start
%                       state to the parent of 'Pss'.  It is
%                       in the form:
%                         result(Act1 ... result(Actn 0)...)
%                       When 'assert_spi_holds2' calls itself
%                       recursively, 'ParentPlan' is modified
%                       so that it contains the latest action
%                       processed.
%               Plan    - List of actions which take us from the
%                       parent of "Pss" to "Pss".
%               Pss     - Integer representing the new p-s-s being.
%                       added.


assert_spi_holds2( ParentPlan [] Pss );

assert_spi_holds2( ParentPlan [Action|Plan] Pss ) <-
     addlist_all( Action ParentPlan Addlist )       % Get the addlist
                                        % for "Action".
     remove_from_addlist( Plan Addlist Newlist )    % Remove
                                        % the members
                                        % of the addlist which
                                        % are deleted by future
                                        % actions in the "Plan"
```

```
    assert_spi_holds3( Newlist Pss )        % Write a "spi_holds"
                                            % axiom for each member
                                            % of "Newlist".
    assert_spi_holds2( result(Action ParentPlan) Plan Pss );
```

```
% assert_spi_holds3:  Assert a new "spi_holds" axiom for each
%                     member of "List".
%
% assert_spi_holds3( List Pss )
%
% Input:        List  - List of statements to be asserted with a
%                       "spi_holds".
%               Pss   - Integer representing a p-s-s.


assert_spi_holds3( [Statement|List] Pss ) <-
    clause( usr spi_holds(Statement Pss) [] )
    cut         % this "Statement" has already been asserted.
    assert_spi_holds3( List Pss );


assert_spi_holds3( [Statement|List] Pss ) <-
    assert( usr spi_holds( Statement Pss ) [] )
    assert_spi_holds3( List Pss );


assert_spi_holds3( [] Pss );
```

```
% addlist_all:   Gets the addlist for a particular action.
%
% addlist_all( Action Plan List )
%
% Input:  Action  - A single action whose addlist we are building.
%         Plan    - Plan which contains 'result'.
%
% Output: List    - Contains the addlist for 'Action'.


addlist_all( Action Plan List ) <-
    addlist( Action [] AddList )        % Get 'Action's addlist.
    bind_all( Action Plan AddList List ); % Find all the
                                        % possible bindings
                                        % for the vars in 'AddList'.
```

```
% bind_all:      Finds all the possible bindings for the
%          vars in 'AddList'.
%          These different bindings exist because
%          the add statements
%          have bodies.
%
% bind_all( Action Plan AddList List )
%
% Input:   Action   - Single action.
%          Plan     - Plan containing 'result'.
%          AddList  - Addlist of 'Action'.
%
% Output: List


bind_all( Action Plan [] [] );

bind_all( Action Plan [Stat|Rest] List ) <-
      all-of( List1 Stat prouver(holds(Stat result(Action Plan))) )
                              % Get all bindings for 'Stat'.
      bind_all( Action Plan Rest List2 )
      append( List1 List2 List );
```

```
% remove_from_addlist:    Remove the statements in the
%                "Addlist" which are
%                deleted by actions in "Plan".
%
% remove_from_addlist( Plan Addlist Newlist )
%
% Input:         Plan      - List of actions.
%                Addlist   - The addlist of an action.
%
% Output:        Newlist   - List containing the
%                            members of "Addlist"
%                            which are not deleted by the actions
%                            in "Plan".


remove_from_addlist( [] Addlist Addlist ) <-
      cut;

remove_from_addlist( Plan [] [] ) <-
      cut;

remove_from_addlist( [Action|Plan] Addlist Newlist ) <-
      remove_from_addlist1( Action Addlist Newlist1 )  % See
                                          % which statements
```

```
                                        % are deleted by
                                        % "Action".
    remove_from_addlist( Plan Newlist1 Newlist );
```

```
% remove_from_addlist1:   Remove the statements in
%                 "Addlist" which are
%                 deleted by "Action".
%
% remove_from_addlist1( Action Addlist Newlist )
%
% Input:   Action  - a single action
%          Addlist - The addlist of an action.
%
% Output: Newlist - List containing the members of "Addlist"
%                   which are not deleted by "Action".


remove_from_addlist1( Action [] [] );

remove_from_addlist1( Action [Stat|Addlist] [Stat|Newlist] ) <-
    prouver( preserves(Action Stat) )          % "Action" does not
                                        % delete "Stat".
    cut
    remove_from_addlist1( Action Addlist Newlist );

remove_from_addlist1( Action [Stat|Addlist] Newlist ) <-
    remove_from_addlist1( Action Addlist Newlist ); % "Stat"
                                        % is deleted by "Action".
```

```
% assert_spi_preserves:          Assert the "spi_preserves"
%                       axioms for "Pss".
%
% NOTE:       "assert_spi_holds" must be executed
%         before this procedure
%         because "assert_spi_preserves" uses
%         the "spi_holds" axioms
%         asserted by "assert_spi_holds".
%
% assert_spi_preserves( Plan Pss )
%
% Input:   Plan   - List of actions which take us from the parent of
%                   "Pss" to "Pss".
%          Pss    - Integer representing the new p-s-s being added.


assert_spi_preserves( Plan Pss ) <-
```

```
    delete_list( Plan DelList )                    % Get the delete list
                                                   % for all the
                                                   % actions in the plan.
    remove_from_deletelist( Pss DelList DeleteList ) % Remove
                                                   % elements from
                                                   % the delete list
                                                   % which are later
                                                   % added by other
                                                   % actions.
    assert_spi_preserves1( DeleteList Pss );  % Assert
                                                   % everything
                                                   % in "DeleteList".
```

```
% assert_spi_preserves1:    Assert everything in "DeleteList"
%                          using one "spi_preserves" axiom.
%
% assert_spi_preserves1( DeleteList Pss )
%
% Input:   DeleteList  - List of deleted statements.
%          Pss         - int. repres. a p-s-s


assert_spi_preserves1( DeleteList Pss ) <-
    convert( List1 DeleteList )    % Convert "DeleteList" so that it
                                   % contains diff statements.
    make_same( List1 X )           % Make all the variables in "List1"
                                   % the same.
    assert( usr spi_preserves(X Pss) List1 );
```

```
% remove_from_deletelist:  Remove elements from
%                          "DeleteList" which are
%                          later added by other actions.
%
% remove_from_deletelist( Pss DeleteList OutList )
%
% Input:   Pss         - int. repres. a p-s-s
%          DeleteList  - List of deleted statements.
%
% Output: OutList      - "DeleteList" minus some elements.


remove_from_deletelist( Pss [] [] );
```

```
remove_from_deletelist( Pss [Statement|DeleteList]
      [Statement|OutList] ) <-
      not( has_no_var(Statement) )                  % "Statement"
                                          % contains vars.
      cut
      remove_from_deletelist( Pss DeleteList OutList );


remove_from_deletelist( Pss [Statement|DeleteList]
      [Statement|OutList] ) <-
      not( clause(usr spi_holds(Statement Pss) []) )
                                          % "Statement" is not
                                          % added.
      cut
      remove_from_deletelist( Pss DeleteList OutList );


remove_from_deletelist( Pss [Statement|DeleteList] OutList ) <-
      remove_from_deletelist( Pss DeleteList OutList );
                                          % "Statement" is
                                          % added.
```

```
% make_same:  Given a list:
%              List = [diff(X1 1)  ...  diff(Xn n)]
%         make all the Xi variables equal to "Y".
%
% make_same( List Y )
%
% Input:  List
%         Y


make_same( [diff(X Z)|Rest] Y ) <-
      eq( X Y )
      make_same( Rest Y );

make_same( [] Y );
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%% Utilities %%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
utilities(X);    % this is a dummy axiom
```

```
% addlist: Gets the addlist for a particular action.
%
% addlist( Action Temp List )
%
% Input:   Action  - single action
%          Temp    - initialized to [], temporary list
%
% Output: List     - contains the addlist for "Action"


addlist( Action Temp List ) <-
      clause( kwbase holds(X result(Action W)) Z )   % Get an
                                          % added statement.
      not( is_var(X) )                    % Make sure we don't
                                          % have the frame axiom.
      not( member(X Temp) )                       % Have we looked at
                                          % this one before?
      cut                                 % No!
      addlist( Action [X|Temp] List );


addlist( Action List List );
```

```
%
% "append" appends the first two lists together to produce the
% third argument.
%
% ie. append( [a b] [c d] X ) returns X = [a b c d]
%
%
append( [] Y Y );
append( [U|X] Y [U|Z] ) <-
       append( X Y Z );
```

```
% convert: Convert between:
%                In = [diff(X1 Y1)  ...  diff(Xn Yn)]
%          and:
%                Out = [Y1  ...  Yn]
%
% convert( In Out )
```

```
convert( [] [] );

convert( [diff(X Y)|In] [Y|Out] ) <-
      convert( In Out );
```

```
% delete_list:    Build the delete list of all the actions in "Plan".
%
% delete_list( Plan DeleteList )
%
% Input:   Plan       - List of actions.
%
% Output: DeleteList - Delete list of all the actions in "Plan".


delete_list( [] [] );

delete_list( [Action|Plan] DeleteList ) <-
      clause( kwbase preserves(Action X) List1 )    % get the
                                                    % delete list
                                                    % for "Action".
      convert( List1 List2 )                        % Edit the delete list.
      delete_list( Plan List3 )                     % Process the rest of
                                                    % the actions.
      append( List2 List3 DeleteList );
```

```
% has_list_var(List) :        Succeds if "List" contains a variable.


has_list_var( [X|List] ) <-
      is_var( X )
      cut;

has_list_var( [X|List] ) <-
      has_list_var( List );
```

```
% has_no_var(X) :     Succeeds if "X" contains no variables.
%                    "X" is not a list.


has_no_var( X ) <-
      cons( List X )                % break "X" into a list
      not( has_list_var(List) );    % make sure list contains
```

```
                    % no variables

-------------------------------------------


%
% member(a [a b c]) will succeed
% member(a [d b c]) will fail
%
%

member( X [X|Y] );

member( X [Y|Z] ) <-
        member( X Z );

-------------------------------------------


%
% minus( List1 List2 List3 ) returns List3 = the
% elements in List1 which
% do not appear in List2.
%
% ie. minus( [a b] [b c] X ) returns X = [a]
%
%
minus( [X|L1] L2 L3 ) <-
        member( X L2 )
        cut
        minus( L1 L2 L3 );
minus( [X|L1] L2 [X|L3] ) <-
        minus( L1 L2 L3 );
minus( [] L2 [] );

-------------------------------------------


% plan_to_pss:  Get the plan which takes you
%           from the start state to
%           "Pss". The spider is traversed backwards, from the
%           "Pss" we travel up towards the root.
%
% plan_to_pss( Pss Temp Plan )
%
% Input:    Pss   - integer representing a p-s-s.
%           Temp  - initialized to [], used as a temporary list
%
% Output: Plan  - List of actions which take you from the start
%                 state to "Pss".
```

```
plan_to_pss( 0 Plan Plan ) <-              % Reached the start state.
      cut;

plan_to_pss( Pss Temp Plan ) <-
      prove( usr spi_plan(Plan1 N Pss) )   % Get the plan which
                                           % got us to this "Pss".
      append( Plan1 Temp Plan2 )
      prove( usr spi_parent(ParPss Pss) )  % Get the parent.
      plan_to_pss( ParPss Plan2 Plan );
```

```
% prouver:Prove an axiom using the knowledge
%          base (kwbase) and the
%          aux. db. of usr.  The reason the aux. db. of usr is
%          checked is because "interface" asserts axioms in it
%          which can also be found in kwbase.
%
% prouver( Axiom )
%
% Input:   Axiom  - a single axiom to be proven.


prouver( Axiom ) <-
      is_var( Axiom )        % 'Axiom' is a variable.
      cut
      fail;


prouver( Axiom ) <-
      ne( Axiom not(X) )     % 'Axiom' is not a negation.
      prove( kwbase Axiom );


prouver( Axiom ) <-
      ne( Axiom not(X) )     % 'Axiom' is not a negation.
      prove( usr    Axiom );


prouver( not(Axiom) ) <-   % 'Axiom' is a negation.
      prove( and( not(kwbase Axiom) not(usr Axiom) ) );
```

```
%
% "union" will perform the union of the first two lists and put this
% union in the third list.
%
% ie. union( [a] [b] X ) returns X = [a b]
%
```

```
%
union( [X|L1] L2 L3 ) <-
        member( X L2 )
        cut
        union( L1 L2 L3 );
union( [X|L1] L2 L3 ) <-
        union( L1 [X|L2] L3 );
union( [] L1 L1 );
```

Appendix E: Test1

This appendix contains all the files related to test1; they are:

1. The initial spider
2. The axiomatization of the problem domain
3. The input queries to Madame
4. The input queries to the planning component which did not use the spider
5. The final spider
6. Output from Madame
7. Output from the planning component which did not use the spider

1. **The following page is a listing of the initial spider.**

```
% frame axiom for the spider
% This specifies which statements which held in the parent state still hold.
spi_holds( Statement Cpss ) <-
        spi_parent( Ppss Cpss )
        spi_holds( Statement Ppss )
        spi_preserves( Statement Cpss );


% "spi_no" specifies how many p-s-s's there are.
% This is also the next available int. to be used as an identification number
% for a p-s-s.
spi_no(1);


% axioms for the start-state:
spi_poss( 0 );
spi_children( [] 0 );
spi_plan( [] 0 0 );
spi_holds( Stat 0 ) <-
        holds( Stat 0 );
```

2. The following pages contain a listing of the axiomatization of the blocks world used in test1.

```
holds( on(a b) 0 );
holds( on(b c) 0 );
holds( on(c d) 0 );
holds( on(d e) 0 );
holds( on(e p) 0 );
holds( clear(a) 0 );
holds( clear(q) 0 );
holds( clear(r) 0 );

holds( on(X Z) result(trans(X Y Z) W ));
holds( clear(Y) result(trans(X Y Z) W) );

% frame axiom
holds( U result(V W) ) <-
        holds(U W)
        preserves(V U);
```

```
manip(a);
manip(b);
manip(c);
manip(d);
manip(e);
```

```
poss(0);

poss( result(trans(X Y Z) W) ) <-
        poss(W)
        holds( on(X Y)  W )
        holds( clear(X) W )
        holds( clear(Z) W )
        diff(X Z)
        manip(X);
```

```
preserves( trans(X Y Z) U) <-
        diff( U on(X Y) )
        diff( U clear(Z) );
```

```
impossible( |on(X Y) clear(Y)| );
```

```
imposs( [on(X Y) on(X Z) diff(Y Z)] );
imposs( [on(X X)] );
```

```
% diff(X Y) succeeds if X and Y are different

diff( X Y ) <- ne( X Y );
```

```
only( diff(Y Z) );
only( manip(X) );
```

---

3. **The following page contains the input queries passed to Madame.**

```
?time;

?setup;

?interface([clear(q) clear(p)] P);

?interface([on(a r)] P);

?interface([on(e d) on(d c) on(c b) on(b a) on(a q)] P);

?interface([on(e r)] P);

?interface([on(b q) on(c r) on(a p)] P);

?interface([on(c b)] P);

?interface([on(a b)] P);

?interface([on(d e) on(e p) on(c b) on(b a) on(a q)] P);

?interface([clear(a)] P);

?interface([on(a q) on(b a) on(d b) on(c d) on(e p)] P);

?interface([clear(b)] P);

?interface([on(c d) on(d e) on(e p) on(b q) on(a r)] P);

?interface([clear(c)] P);

?interface([clear(d)] P);

?interface([on(c r) on(d c) on(a p) clear(d) clear(a)] P);

?interface([clear(e)] P);

?interface([on(d a) clear(c) on(e b)] P);

?interface([on(d q)] P);

?interface([clear(c) clear(b) clear(a)] P);

?interface([on(e a)] P);

?laFin;

?quit;
```

4. The following page contains the input queries passed to Warplan-Blocks.

```
?time;

?setup;

?empower(0 [] P [clear(q) clear(p)] );

?empower(0 [] P [on(a r)] );

?empower(0 [] P [on(e d) on(d c) on(c b) on(b a) on(a q)] );

?empower(0 [] P [on(e r)] );

?empower(0 [] P [on(b q) on(c r) on(a p)] );

?empower(0 [] P [on(c b)] );

?empower(0 [] P [on(a b)] );

?empower(0 [] P [on(d e) on(e p) on(c b) on(b a) on(a q)] );

?empower(0 [] P [clear(a)] );

?empower(0 [] P [on(a q) on(b a) on(d b) on(c d) on(e p)] );

?empower(0 [] P [clear(b)] );

?empower(0 [] P [on(c d) on(d e) on(e p) on(b q) on(a r)] );

?empower(0 [] P [clear(c)] );

?empower(0 [] P [clear(d)] );

?empower(0 [] P [on(c r) on(d c) on(a p) clear(d) clear(a)] );

?empower(0 [] P [clear(e)] );

?empower(0 [] P [on(d a) clear(c) on(e b)] );

?empower(0 [] P [on(d q)] );

?empower(0 [] P [clear(c) clear(b) clear(a)] );

?empower(0 [] P [on(e a)] );

?quit;
```

5. **The following pages contain a listing of the final spider produced from test1.**

```
spi_holds(Statement,Cpss) <-
        spi_parent(Ppss,Cpss),
        spi_holds(Statement,Ppss),
        spi_preserves(Statement,Cpss);

spi_holds(Stat,0) <-
        holds(Stat,0);

spi_holds(on(a,r),1);
spi_holds(on(b,a),1);
spi_holds(on(c,b),1);
spi_holds(clear(e),1);
spi_holds(on(d,c),1);
spi_holds(clear(p),1);
spi_holds(on(e,d),1);
spi_holds(clear(r),2);
spi_holds(on(a,q),2);
spi_holds(on(b,a),2);
spi_holds(on(c,b),2);
spi_holds(clear(e),2);
spi_holds(on(d,c),2);
spi_holds(clear(p),2);
spi_holds(on(e,d),2);
spi_holds(on(a,q),3);
spi_holds(on(b,a),3);
spi_holds(clear(d),3);
spi_holds(on(c,b),3);
spi_holds(clear(e),3);
spi_holds(on(d,c),3);
spi_holds(clear(p),3);
spi_holds(on(e,r),3);
spi_holds(on(b,q),4);
spi_holds(on(c,r),4);
spi_holds(clear(e),4);
spi_holds(on(d,c),4);
spi_holds(on(e,d),4);
spi_holds(clear(b),4);
spi_holds(on(a,p),4);
spi_holds(clear(r),5);
spi_holds(on(e,p),5);
spi_holds(clear(c),5);
spi_holds(on(d,e),5);
spi_holds(clear(e),6);
spi_holds(on(d,b),6);
spi_holds(clear(r),6);
spi_holds(on(c,d),6);
spi_holds(clear(b),7);
spi_holds(on(a,r),7);
spi_holds(clear(c),7);
spi_holds(on(b,q),7);
spi_holds(clear(d),8);
spi_holds(on(e,b),8);
spi_holds(clear(e),9);
spi_holds(on(d,a),9);
```

```
spi_holds(on(c,d),9);
spi_holds(clear(p),9);
spi_holds(on(e,b),9);
spi_holds(clear(d),10);
spi_holds(on(c,r),10);
spi_holds(clear(a),10);
spi_holds(on(b,c),10);
spi_holds(on(a,b),10);
spi_holds(clear(e),10);
spi_holds(on(d,q),10);
spi_holds(on(a,q),11);
spi_holds(clear(e),11);
spi_holds(on(d,r),11);
spi_holds(clear(b),11);
spi_holds(on(c,d),11);
spi_holds(on(b,c),11);
spi_holds(clear(p),11);
spi_holds(on(e,a),11);


spi_no(12);


spi_poss(0);
spi_poss(1);
spi_poss(2);
spi_poss(3);
spi_poss(4);
spi_poss(5);
spi_poss(6);
spi_poss(7);
spi_poss(8);
spi_poss(9);
spi_poss(10);
spi_poss(11);


spi_parent(0,1);
spi_parent(0,2);
spi_parent(0,3);
spi_parent(0,4);
spi_parent(3,5);
spi_parent(5,6);
spi_parent(0,7);
spi_parent(4,8);
spi_parent(7,9);
spi_parent(0,10);
spi_parent(0,11);


spi_children([],1);
spi_children([],2);
spi_children([5],3);
spi_children([],6);
```

```
spi_children([6],5);
spi_children([],8);
spi_children([8],4);
spi_children([],9);
spi_children([9],7);
spi_children([],10);
spi_children([],11);
spi_children([11,10,7,4,3,2,1],0);


spi_plan([],0,0);
spi_plan([trans(a,b,r),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(e,p,d)],5,1);
spi_plan([trans(a,b,r),trans(a,r,q),trans(b,c,a),trans(c,d,b),
    trans(d,e,c),trans(e,p,d)],6,2);
spi_plan([trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(e,p,r)],5,3);
spi_plan([trans(a,b,q),trans(a,q,r),trans(b,c,q),trans(a,r,b),
    trans(c,d,r),trans(d,e,c),trans(e,p,d),trans(a,b,p)],8,4);
spi_plan([trans(e,r,p),trans(d,c,e)],2,5);
spi_plan([trans(c,b,r),trans(d,e,b),trans(c,r,d)],3,6);
spi_plan([trans(a,b,q),trans(a,q,r),trans(b,c,q)],3,7);
spi_plan([trans(e,d,b)],1,8);
spi_plan([trans(c,d,b),trans(d,e,a),trans(c,b,d),trans(e,p,b)],4,9);
spi_plan([trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(c,b,r),
    trans(b,a,c),trans(a,q,b),trans(d,e,q)],7,10);
spi_plan([trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(d,e,c),
    trans(d,c,r),trans(c,b,d),trans(b,a,c),trans(e,p,a)],8,11);


spi_preserves(_39440,1) <-
      diff(_39440,clear(r)),
      diff(_39440,on(a,b)),
      diff(_39440,clear(a)),
      diff(_39440,on(b,c)),
      diff(_39440,clear(b)),
      diff(_39440,on(c,d)),
      diff(_39440,clear(c)),
      diff(_39440,on(d,e)),
      diff(_39440,clear(d)),
      diff(_39440,on(e,p));
spi_preserves(_42782,2) <-
      diff(_42782,on(a,b)),
      diff(_42782,clear(q)),
      diff(_42782,on(a,r)),
      diff(_42782,clear(a)),
      diff(_42782,on(b,c)),
      diff(_42782,clear(b)),
      diff(_42782,on(c,d)),
      diff(_42782,clear(c)),
      diff(_42782,on(d,e)),
      diff(_42782,clear(d)),
      diff(_42782,on(e,p));
spi_preserves(_32463,3) <-
      diff(_32463,clear(q)),
      diff(_32463,on(a,b)),
```

```
        diff(_32463,clear(a)),
        diff(_32463,on(b,c)),
        diff(_32463,clear(b)),
        diff(_32463,on(c,d)),
        diff(_32463,clear(c)),
        diff(_32463,on(d,e)),
        diff(_32463,clear(r)),
        diff(_32463,on(e,p));
spi_preserves(_63971,4) <-
        diff(_63971,on(a,q)),
        diff(_63971,clear(q)),
        diff(_63971,on(b,c)),
        diff(_63971,on(a,r)),
        diff(_63971,clear(r)),
        diff(_63971,on(c,d)),
        diff(_63971,clear(c)),
        diff(_63971,on(d,e)),
        diff(_63971,clear(d)),
        diff(_63971,on(e,p)),
        diff(_63971,clear(p)),
        diff(_63971,on(a,b));
spi_preserves(_25866,5) <-
        diff(_25866,clear(p)),
        diff(_25866,on(e,r)),
        diff(_25866,clear(e)),
        diff(_25866,on(d,c));
spi_preserves(_29446,6) <-
        diff(_29446,on(c,b)),
        diff(_29446,clear(b)),
        diff(_29446,on(d,e)),
        diff(_29446,clear(d)),
        diff(_29446,on(c,r));
spi_preserves(_35453,7) <-
        diff(_35453,on(a,b)),
        diff(_35453,clear(r)),
        diff(_35453,on(a,q)),
        diff(_35453,clear(q)),
        diff(_35453,on(b,c));
spi_preserves(_12289,8) <-
        diff(_12289,clear(b)),
        diff(_12289,on(e,d));
spi_preserves(_34273,9) <-
        diff(_34273,clear(a)),
        diff(_34273,on(d,e)),
        diff(_34273,clear(d)),
        diff(_34273,on(c,b)),
        diff(_34273,clear(b)),
        diff(_34273,on(e,p));
spi_preserves(_60466,10) <-
        diff(_60466,on(c,d)),
        diff(_60466,clear(r)),
        diff(_60466,on(c,b)),
        diff(_60466,clear(c)),
        diff(_60466,on(b,a)),
```

```
        diff(_60466,clear(b)),
        diff(_60466,on(a,q)),
        diff(_60466,clear(q)),
        diff(_60466,on(d,e));
spi_preserves(_70558,11) <-
        diff(_70558,clear(q)),
        diff(_70558,on(a,b)),
        diff(_70558,on(d,e)),
        diff(_70558,clear(r)),
        diff(_70558,on(d,c)),
        diff(_70558,clear(d)),
        diff(_70558,on(c,b)),
        diff(_70558,clear(c)),
        diff(_70558,on(b,a)),
        diff(_70558,clear(a)),
        diff(_70558,on(e,p));
```

6. **The following pages contain the output produced by Madame during test1.**

```
Waterloo Unix Prolog [Release 1.2 -- Apr 9,1984]

?     time;
timing on

Inferences  =     5
Unifications =    6
Time (msec)  =    83
Speed (LIPS) =    60
yes

?     setup;

Inferences  =    11
Unifications =    14
Time (msec)  =   183
Speed (LIPS) =    60
yes

?     interface([clear(q),clear(p)],P);
*UNKNOWN PREDICATE* spi_parent(Ppss,Cpss)
*UNKNOWN PREDICATE* spi_parent(Ppss,Cpss)

The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[clear(p)]
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(c,_21363)
*UNKNOWN PREDICATE* on(d,_21550)
*UNKNOWN PREDICATE* on(e,_21737)
*UNKNOWN PREDICATE* on(_21775,_21774)
*UNKNOWN PREDICATE* on(_22798,_22798)
*UNKNOWN PREDICATE* on(_22913,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_22896)
*UNKNOWN PREDICATE* clear(_16770)
*UNKNOWN PREDICATE* clear(_11447)
*UNKNOWN PREDICATE* clear(_6893)
*UNKNOWN PREDICATE* clear(_3074)

The Spider has been modified
   New p-s-s added is:    1   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,r),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(e,p,d)]

Inferences  = 10631
Unifications = 16901
Time (msec)  = 21033
Speed (LIPS) =  505
P = result(trans(e,p,d),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,r),0)))))yes

?     interface([on(a,r)],P);
```

```
The closest p-s-s is:  1
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences   =   140
Unifications =   183
Time (msec)  =   250
Speed (LIPS) =   560
P = result(trans(e,p,d),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,r),0)))))yes

?     interface([on(e,d),on(d,c),on(c,b),on(b,a),on(a,q)],P);

The closest p-s-s is:  1
The distance is:       1
The goals which do not hold in the p-s-s are:
[on(a,q)]
*UNKNOWN PREDICATE* on(a,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* on(a,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_28391,_28390)
*UNKNOWN PREDICATE* on(_29693,_29693)
*UNKNOWN PREDICATE* on(a,_29838)
*UNKNOWN PREDICATE* clear(q)

The Spider has been modified
   New p-s-s added is:    2   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,r),trans(a,r,q),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(e,p,d)]

Inferences   =  11616
Unifications =  18656
Time (msec)  =  16433
Speed (LIPS) =    706
P = result(trans(e,p,d),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,r,q),
result(trans(a,b,r),0))))))yes

?     interface([on(e,r)],P);

The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[on(e,r)]
*UNKNOWN PREDICATE* on(e,r)
```

```
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_968,_967)
*UNKNOWN PREDICATE* on(e,_1450)
*UNKNOWN PREDICATE* on(_1501,_1500)
*UNKNOWN PREDICATE* on(_1919,_1919)
*UNKNOWN PREDICATE* on(e,_2000)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(_14216)
*UNKNOWN PREDICATE* clear(_9398)
*UNKNOWN PREDICATE* clear(_5332)
*UNKNOWN PREDICATE* clear(r)

The Spider has been modified
   New p-s-s added is:    3   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(e,p,r)]

Inferences   =  8688
Unifications =  13906
Time (msec)  =  12600
Speed (LIPS) =  689
P = result(trans(e,p,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))yes

?      interface([on(b,q),on(c,r),on(a,p)],P);

The closest p-s-s is:  0
The distance is:       3
The goals which do not hold in the p-s-s are:
[on(b,q),on(c,r),on(a,p)]
*UNKNOWN PREDICATE* on(b,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_1389,_1388)
*UNKNOWN PREDICATE* on(b,_1871)
*UNKNOWN PREDICATE* on(_1922,_1921)
*UNKNOWN PREDICATE* on(_2340,_2340)
*UNKNOWN PREDICATE* on(b,_2421)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* on(qqq(0),_49376)
*UNKNOWN PREDICATE* on(e,_49615)
*UNKNOWN PREDICATE* on(a,_49815)
*UNKNOWN PREDICATE* on(c,_50002)
*UNKNOWN PREDICATE* on(b,_50189)
*UNKNOWN PREDICATE* on(_50221,_50220)
*UNKNOWN PREDICATE* on(_51244,_51244)
*UNKNOWN PREDICATE* on(_51352,e)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(_51335)
*UNKNOWN PREDICATE* clear(_44936)

The Spider has been modified
   New p-s-s added is:    4   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
```

```
    [trans(a,b,q),trans(a,q,r),trans(b,c,q),trans(a,r,b),trans(c,d,r),
     trans(d,e,c),trans(e,p,d),trans(a,b,p)]

Inferences   = 17285
Unifications = 28138
Time (msec)  = 25433
Speed (LIPS) =  679
P = result(trans(a,b,p),result(trans(e,p,d),result(trans(d,e,c),
result(trans(c,d,r),result(trans(a,r,b),result(trans(b,c,q),
result(trans(a,q,r),result(trans(a,b,q),0)))))))yes

?     interface([on(c,b)],P);

The closest p-s-s is:  3
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences   =  170
Unifications =  348
Time (msec)  =  333
Speed (LIPS) =  510
P = result(trans(e,p,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0))))yes

?     interface([on(a,b)],P);

The closest p-s-s is:  0
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in start state

Inferences   =  56
Unifications =  69
Time (msec)  =  116
Speed (LIPS) =  480
P = 0yes

?     interface([on(d,e),on(e,p),on(c,b),on(b,a),on(a,q)],P);

The closest p-s-s is:  3
The distance is:       2
The goals which do not hold in the p-s-s are:
[on(d,e),on(e,p)]
*UNKNOWN PREDICATE* on(d,e)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(q)
*UNKNOWN PREDICATE* on(_4430,_4429)
```

```
*UNKNOWN PREDICATE* on(d,_5082)
*UNKNOWN PREDICATE* on(c,_5297)
*UNKNOWN PREDICATE* on(b,_5460)
*UNKNOWN PREDICATE* on(a,_5623)
*UNKNOWN PREDICATE* on(_5653,_5652)
The Spider has been modified
   New p-s-s added is:    5   It's parent is:        3
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(e,r,p),trans(d,c,e)]

Inferences  =  7215
Unifications  =  12521
Time (msec)  =  11483
Speed (LIPS)  =  628
P = result(trans(d,c,e),result(trans(e,r,p),result(trans(e,p,r),
result(trans(d,e,c),result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0)))))))yes

?     interface([clear(a)],P);

The closest p-s-s is:  0
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in start state

Inferences  =  56
Unifications  =  69
Time (msec)  =  83
Speed (LIPS)  =  672
P = 0yes

?     interface([on(a,q),on(b,a),on(d,b),on(c,d),on(e,p)],P);

The closest p-s-s is:  5
The distance is:       2
The goals which do not hold in the p-s-s are:
[on(d,b),on(c,d)]
*UNKNOWN PREDICATE* on(d,b)
*UNKNOWN PREDICATE* on(_23072,_23071)
*UNKNOWN PREDICATE* on(_23766,_23766)
*UNKNOWN PREDICATE* on(c,_23847)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)

The Spider has been modified
   New p-s-s added is:    6   It's parent is:        5
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(c,b,r),trans(d,e,b),trans(c,r,d)]

Inferences  =  8075
Unifications  =  14541
Time (msec)  =  12250
```

```
Speed (LIPS) =   659
P = result(trans(c,r,d),result(trans(d,e,b),result(trans(c,b,r),
result(trans(d,c,e),result(trans(e,r,p),result(trans(e,p,r),
result(trans(d,e,c),result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0))))))))))yes

?     interface([clear(b)],P);

The closest p-s-s is:  4
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences  =   179
Unifications =   279
Time (msec)  =   316
Speed (LIPS) =   565
P = result(trans(a,b,p),result(trans(e,p,d),result(trans(d,e,c),
result(trans(c,d,r),result(trans(a,r,b),result(trans(b,c,q),
result(trans(a,q,r),result(trans(a,b,q),0)))))))yes

?     interface([on(c,d),on(d,e),on(e,p),on(b,q),on(a,r)],P);

The closest p-s-s is:  0
The distance is:       2
The goals which do not hold in the p-s-s are:
[on(b,q),on(a,r)]
*UNKNOWN PREDICATE* on(_21633,_21632)
*UNKNOWN PREDICATE* on(_22656,_22656)
*UNKNOWN PREDICATE* on(_22764,q)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_22747)
*UNKNOWN PREDICATE* on(a,r)

The Spider has been modified
   New p-s-s added is:    7    It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,q),trans(a,q,r),trans(b,c,q)]

Inferences  =  9795
Unifications =  18003
Time (msec)  =  15500
Speed (LIPS) =   631
P = result(trans(b,c,q),result(trans(a,q,r),result(trans(a,b,q),0)))yes

?     interface([clear(c)],P);

The closest p-s-s is:  7
The distance is:       0
The goals which do not hold in the p-s-s are:
[]
```

```
Spider not modified - Goals hold in the closest p-s-s

Inferences   =   119
Unifications =   247
Time (msec)  =   266
Speed (LIPS) =   446
P = result(trans(b,c,q),result(trans(a,q,r),result(trans(a,b,q),0)))yes

?     interface([clear(d)],P);

The closest p-s-s is:  3
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences   =   200
Unifications =   449
Time (msec)  =   383
Speed (LIPS) =   521
P = result(trans(e,p,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))yes

?     interface([on(c,r),on(d,c),on(a,p),clear(d),clear(a)],P);

The closest p-s-s is:  4
The distance is:       1
The goals which do not hold in the p-s-s are:
[clear(d)]
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(r)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(_5811,_5810)
*UNKNOWN PREDICATE* on(qqq(0),_6797)
*UNKNOWN PREDICATE* on(c,_7024)
*UNKNOWN PREDICATE* on(d,_7199)
*UNKNOWN PREDICATE* on(a,_7374)
*UNKNOWN PREDICATE* on(_7411,_7410)
*UNKNOWN PREDICATE* on(_8365,_8365)
*UNKNOWN PREDICATE* on(_8473,d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(_8456)

The Spider has been modified
   New p-s-s added is:    8   It's parent is:       4
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(e,d,b)]

Inferences   =  3268
Unifications =  7699
Time (msec)  =  5866
```

```
Speed (LIPS) =   557
P = result(trans(e,d,b),result(trans(a,b,p),result(trans(e,p,d),
result(trans(d,e,c),result(trans(c,d,r),result(trans(a,r,b),
result(trans(b,c,q),result(trans(a,q,r),
result(trans(a,b,q),0)))))))))yes

?     interface([clear(e)],P);

The closest p-s-s is:  4
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences  =   209
Unifications =   410
Time (msec)  =   416
Speed (LIPS) =   501
P = result(trans(a,b,p),result(trans(e,p,d),result(trans(d,e,c),
result(trans(c,d,r),result(trans(a,r,b),result(trans(b,c,q),
result(trans(a,q,r),result(trans(a,b,q),0)))))))yes

?     interface([on(d,a),clear(c),on(e,b)],P);

The closest p-s-s is:  7
The distance is:       2
The goals which do not hold in the p-s-s are:
[on(d,a),on(e,b)]
*UNKNOWN PREDICATE* on(d,a)
*UNKNOWN PREDICATE* on(qqq(0),_19561)
*UNKNOWN PREDICATE* on(e,_19801)
*UNKNOWN PREDICATE* on(d,_19976)
*UNKNOWN PREDICATE* on(_20013,_20012)
*UNKNOWN PREDICATE* on(_20954,_20954)
*UNKNOWN PREDICATE* on(_21061,b)
*UNKNOWN PREDICATE* clear(_21044)

The Spider has been modified
   New p-s-s added is:   9   It's parent is:       7
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(c,d,b),trans(d,e,a),trans(c,b,d),trans(e,p,b)]

Inferences  =  9168
Unifications =  16795
Time (msec)  =  13733
Speed (LIPS) =   667
P = result(trans(e,p,b),result(trans(c,b,d),result(trans(d,e,a),
result(trans(c,d,b),result(trans(b,c,q),result(trans(a,q,r),
result(trans(a,b,q),0)))))))yes

?     interface([on(d,q)],P);

The closest p-s-s is:  0
```

```
The distance is:       1
The goals which do not hold in the p-s-s are:
[on(d,q)]
*UNKNOWN PREDICATE* on(d,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_2276,_2275)
*UNKNOWN PREDICATE* on(d,_2758)
*UNKNOWN PREDICATE* on(_2809,_2808)
*UNKNOWN PREDICATE* on(_3227,_3227)
*UNKNOWN PREDICATE* on(d,_3308)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* on(b,_33820)
*UNKNOWN PREDICATE* on(a,_33995)
*UNKNOWN PREDICATE* on(d,_34183)
*UNKNOWN PREDICATE* on(_34214,_34213)
*UNKNOWN PREDICATE* on(_35168,_35168)
*UNKNOWN PREDICATE* on(_35269,b)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(_35252)
*UNKNOWN PREDICATE* clear(_29786)
*UNKNOWN PREDICATE* clear(_25043)

The Spider has been modified
   New p-s-s added is:    10   It's parent is:       0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(c,b,r),trans(b,a,c),
    trans(a,q,b),trans(d,e,q)]

Inferences   = 16257
Unifications = 27034
Time (msec)  = 23350
Speed (LIPS) =   696
P = result(trans(d,e,q),result(trans(a,q,b),result(trans(b,a,c),
result(trans(c,b,r),result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0)))))))yes

?      interface([clear(c),clear(b),clear(a)],P);

The closest p-s-s is:  7
The distance is:       0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

Inferences   =  277
Unifications =  821
Time (msec)  =  600
Speed (LIPS) =  461
P = result(trans(b,c,q),result(trans(a,q,r),result(trans(a,b,q),0)))yes

?      interface([on(e,a)],P);
```

```
The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[on(e,a)]
*UNKNOWN PREDICATE* on(e,a)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_2490,_2489)
*UNKNOWN PREDICATE* on(e,_2972)
*UNKNOWN PREDICATE* on(_3023,_3022)
*UNKNOWN PREDICATE* on(_3441,_3441)
*UNKNOWN PREDICATE* on(e,_3522)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(_33378)

The Spider has been modified
   New p-s-s added is:    11   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [trans(a,b,q),trans(b,c,a),trans(c,d,b),trans(d,e,c),trans(d,c,r),
    trans(c,b,d),trans(b,a,c),trans(e,p,a)]

Inferences  = 18961
Unifications = 31591
Time (msec)  = 27133
Speed (LIPS) =  698
P = result(trans(e,p,a),result(trans(b,a,c),result(trans(c,b,d),
result(trans(d,c,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))))yes

?    laFin;

Inferences  =    10
Unifications =    10
Time (msec)  =  1200
Speed (LIPS) =     8
yes

?    quit;
```

7. **The following pages contain the output from Warplan-Blocks.**

```
Waterloo Unix Prolog [Release 1.2 -- Apr 9,1984]

?     time;
timing on

Inferences  =     5
Unifications =    6
Time (msec)  =    66
Speed (LIPS) =    75
yes

?     setup;

Inferences  =    11
Unifications =    14
Time (msec)  =   200
Speed (LIPS) =    55
yes

?     empower(0,[],P,[clear(q),clear(p)]);
*UNKNOWN PREDICATE* clear(q)
*UNKNOWN PREDICATE* on(e,_10477)
*UNKNOWN PREDICATE* on(_10513,_10512)
*UNKNOWN PREDICATE* on(_11398,_11398)
*UNKNOWN PREDICATE* on(_11499,d)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(_14160,_14159)
*UNKNOWN PREDICATE* on(qqq(0),_15146)
*UNKNOWN PREDICATE* clear(_16805)
*UNKNOWN PREDICATE* clear(_11482)
*UNKNOWN PREDICATE* clear(_6928)
*UNKNOWN PREDICATE* clear(_3109)

Inferences  =  9814
Unifications =  15616
Time (msec)  =  16783
Speed (LIPS) =   584
P = result(trans(e,p,d),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,r),0)))))yes

?     empower(0,[],P,[on(a,r)]);
*UNKNOWN PREDICATE* on(a,r)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(a,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(a,_1692)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(r)
```

```
Inferences  =  859
Unifications  =  1381
Time (msec)  =  1466
Speed (LIPS)  =  585
P = result(trans(a,b,r),0)yes

?     empower(0,[],P,[on(e,d),on(d,c),on(c,b),on(b,a),on(a,q)]);
*UNKNOWN PREDICATE* on(e,d)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_9019,_9019)
*UNKNOWN PREDICATE* on(_9107,d)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(_11436,_11435)
*UNKNOWN PREDICATE* on(qqq(0),_12361)
*UNKNOWN PREDICATE* on(c,_12576)
*UNKNOWN PREDICATE* on(d,_12739)
*UNKNOWN PREDICATE* clear(_5024)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* on(d,c)
*UNKNOWN PREDICATE* on(c,b)
*UNKNOWN PREDICATE* on(b,a)
*UNKNOWN PREDICATE* on(a,q)

Inferences  =  8267
Unifications  =  13255
Time (msec)  =  11900
Speed (LIPS)  =  694
P = result(trans(e,p,d),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))yes

?     empower(0,[],P,[on(e,r)]);
*UNKNOWN PREDICATE* on(e,r)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(e,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(e,_1692)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(_3188,_3187)
*UNKNOWN PREDICATE* on(qqq(0),_3981)
*UNKNOWN PREDICATE* on(e,_4172)
*UNKNOWN PREDICATE* on(_4200,_4199)
*UNKNOWN PREDICATE* on(_4960,_4960)
*UNKNOWN PREDICATE* on(_5041,e)
*UNKNOWN PREDICATE* on(_19529,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_19512)
```

```
*UNKNOWN PREDICATE* clear(_13908)
*UNKNOWN PREDICATE* clear(_9090)
*UNKNOWN PREDICATE* clear(_5024)
*UNKNOWN PREDICATE* clear(r)

Inferences   =  7765
Unifications =  12380
Time (msec)  =  10566
Speed (LIPS) =  734
P = result(trans(e,p,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))yes

?     empower(0,[],P,[on(b,q),on(c,r),on(a,p)]);
*UNKNOWN PREDICATE* on(b,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(b,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(b,_1692)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_3188,_3187)
*UNKNOWN PREDICATE* on(qqq(0),_3981)
*UNKNOWN PREDICATE* on(b,_4172)
*UNKNOWN PREDICATE* on(_4200,_4199)
*UNKNOWN PREDICATE* on(_4960,_4960)

Inferences   =  15761
Unifications =  25285
Time (msec)  =  21666
Speed (LIPS) =  727
P = result(trans(a,b,p),result(trans(e,p,d),result(trans(d,e,c),
result(trans(c,d,r),result(trans(a,r,b),result(trans(b,c,q),
result(trans(a,q,r),result(trans(a,b,q),0)))))))yes

?     empower(0,[],P,[on(c,b)]);
*UNKNOWN PREDICATE* on(c,b)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(c,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(c,_1692)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)

Inferences   =  3715
Unifications =  5922
Time (msec)  =  5283
Speed (LIPS) =  703
P = result(trans(c,d,b),result(trans(b,c,a),result(trans(a,b,q),0)))yes
```

```
?      empower(0,[],P,[on(a,b)]);
*UNKNOWN PREDICATE* on(a,b)

Inferences  =    57
Unifications =    99
Time (msec)  =    83
Speed (LIPS) =   684
P = 0yes

?      empower(0,[],P,[on(d,e),on(e,p),on(c,b),on(b,a),on(a,q)]);
*UNKNOWN PREDICATE* on(d,e)
*UNKNOWN PREDICATE* on(e,p)
*UNKNOWN PREDICATE* on(c,b)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* on(_11091,_11090)
*UNKNOWN PREDICATE* on(qqq(0),_12082)
*UNKNOWN PREDICATE* on(b,_12309)
*UNKNOWN PREDICATE* on(c,_12484)
*UNKNOWN PREDICATE* on(e,_12659)
*UNKNOWN PREDICATE* on(d,_12834)
*UNKNOWN PREDICATE* on(_12865,_12864)
*UNKNOWN PREDICATE* on(_13832,_13832)
*UNKNOWN PREDICATE* on(_13934,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_13917)
*UNKNOWN PREDICATE* clear(_8313)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* on(b,a)
*UNKNOWN PREDICATE* on(a,q)

Inferences  =  5907
Unifications =  9439
Time (msec)  =  8000
Speed (LIPS) =   738
P = result(trans(c,d,b),result(trans(b,c,a),result(trans(a,b,q),0)))yes

?      empower(0,[],P,[clear(a)]);
*UNKNOWN PREDICATE* clear(a)

Inferences  =    53
Unifications =    93
Time (msec)  =   100
Speed (LIPS) =   530
P = 0yes

?      empower(0,[],P,[on(a,q),on(b,a),on(d,b),on(c,d),on(e,p)]);
*UNKNOWN PREDICATE* on(a,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
```

```
*UNKNOWN PREDICATE* on(a,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(a,_1692)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(q)
*UNKNOWN PREDICATE* on(b,a)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(q)
*UNKNOWN PREDICATE* on(_4284,_4283)
*UNKNOWN PREDICATE* on(b,_4804)
*UNKNOWN PREDICATE* on(a,_4995)
*UNKNOWN PREDICATE* on(_5023,_5022)
*UNKNOWN PREDICATE* on(_5510,_5510)
*UNKNOWN PREDICATE* on(b,_5570)
*UNKNOWN PREDICATE* on(c,_36853)
*UNKNOWN PREDICATE* on(d,_37068)
*UNKNOWN PREDICATE* on(b,_37231)
*UNKNOWN PREDICATE* on(a,_37394)
*UNKNOWN PREDICATE* on(_37424,_37423)
*UNKNOWN PREDICATE* on(_38049,_38049)
*UNKNOWN PREDICATE* on(c,_38123)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* on(e,p)

Inferences   =  11439
Unifications =  18337
Time (msec)  =  16583
Speed (LIPS) =    689
P = result(trans(c,r,d),result(trans(d,e,b),result(trans(c,b,r),
result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0))))))yes

?     empower(0,[],P,[clear(b)]);
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* on(_898,_897)
*UNKNOWN PREDICATE* on(qqq(0),_1653)
*UNKNOWN PREDICATE* on(_1704,_1703)
*UNKNOWN PREDICATE* on(_2395,_2395)
*UNKNOWN PREDICATE* on(_2497,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_2480)

Inferences   =  1050
Unifications =  1679
Time (msec)  =  1366
Speed (LIPS) =   768
P = result(trans(a,b,q),0)yes

?     empower(0,[],P,[on(c,d),on(d,e),on(e,p),on(b,q),on(a,r)]);
*UNKNOWN PREDICATE* on(c,d)
*UNKNOWN PREDICATE* on(d,e)
```

```
*UNKNOWN PREDICATE* on(e,p)
*UNKNOWN PREDICATE* on(b,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* on(_2642,_2641)
*UNKNOWN PREDICATE* on(b,_3294)
*UNKNOWN PREDICATE* on(e,_3509)
*UNKNOWN PREDICATE* on(d,_3672)
*UNKNOWN PREDICATE* on(c,_3835)
*UNKNOWN PREDICATE* on(_3865,_3864)
*UNKNOWN PREDICATE* on(_4490,_4490)
*UNKNOWN PREDICATE* on(b,_4564)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* on(qqq(0),_17214)
*UNKNOWN PREDICATE* on(b,_17466)
*UNKNOWN PREDICATE* on(e,_17653)
*UNKNOWN PREDICATE* on(d,_17840)
*UNKNOWN PREDICATE* on(c,_18027)
*UNKNOWN PREDICATE* on(_18059,_18058)
*UNKNOWN PREDICATE* on(_19082,_19082)
*UNKNOWN PREDICATE* on(_19190,q)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_19173)
*UNKNOWN PREDICATE* on(a,r)

Inferences  =  8235
Unifications  =  13124
Time (msec)  =  11950
Speed (LIPS)  =  689
P = result(trans(b,c,q),result(trans(a,q,r),result(trans(a,b,q),0)))yes

?     empower(0,[],P,[clear(c)]);
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_898,_897)
*UNKNOWN PREDICATE* on(qqq(0),_1653)
*UNKNOWN PREDICATE* on(_1704,_1703)
*UNKNOWN PREDICATE* on(_2395,_2395)
*UNKNOWN PREDICATE* on(_2497,c)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_3989,_3988)
*UNKNOWN PREDICATE* on(qqq(0),_4782)
*UNKNOWN PREDICATE* on(b,_4973)
*UNKNOWN PREDICATE* on(_5001,_5000)
*UNKNOWN PREDICATE* on(_5761,_5761)
*UNKNOWN PREDICATE* on(_5842,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_5825)
*UNKNOWN PREDICATE* clear(_2480)
```

```
Inferences  =  2337
Unifications  =  3725
Time (msec)  =  3750
Speed (LIPS)  =  623
P = result(trans(b,c,a),result(trans(a,b,q),0))yes

?     empower(0,[],P,[clear(d)]);
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* on(_898,_897)
*UNKNOWN PREDICATE* on(qqq(0),_1653)
*UNKNOWN PREDICATE* on(_1704,_1703)
*UNKNOWN PREDICATE* on(_2395,_2395)
*UNKNOWN PREDICATE* on(_2497,d)
*UNKNOWN PREDICATE* clear(_2480)

Inferences  =  3906
Unifications  =  6221
Time (msec)  =  5733
Speed (LIPS)  =  681
P = result(trans(c,d,b),result(trans(b,c,a),result(trans(a,b,q),0)))yes

?     empower(0,[],P,[on(c,r),on(d,c),on(a,p),clear(d),clear(a)]);
*UNKNOWN PREDICATE* on(c,r)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_9017,_9017)
*UNKNOWN PREDICATE* on(_9105,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_9088)
*UNKNOWN PREDICATE* clear(_5024)
*UNKNOWN PREDICATE* clear(r)
*UNKNOWN PREDICATE* on(d,c)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* clear(r)
*UNKNOWN PREDICATE* on(_15053,_15052)
*UNKNOWN PREDICATE* on(d,_15573)
*UNKNOWN PREDICATE* on(_72749,_72748)
*UNKNOWN PREDICATE* on(_73716,_73716)
*UNKNOWN PREDICATE* on(_73818,b)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(_73801)
*UNKNOWN PREDICATE* clear(_67835)
*UNKNOWN PREDICATE* clear(a)

Inferences  =  21623
Unifications  =  34572
Time (msec)  =  28733
Speed (LIPS)  =  752
P = result(trans(b,d,e),result(trans(e,b,q),result(trans(a,q,p),
result(trans(e,p,b),result(trans(b,e,d),result(trans(b,a,e),
result(trans(d,e,c),result(trans(c,d,r),result(trans(b,c,a),
result(trans(a,b,q),0)))))))))yes

?     empower(0,[],P,[clear(e)]);
```

```
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* on(_898,_897)
*UNKNOWN PREDICATE* on(qqq(0),_1653)
*UNKNOWN PREDICATE* on(_1704,_1703)
*UNKNOWN PREDICATE* on(_2395,_2395)
*UNKNOWN PREDICATE* on(_2497,e)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* on(_3991,_3990)
*UNKNOWN PREDICATE* on(qqq(0),_4784)
*UNKNOWN PREDICATE* on(d,_4975)
*UNKNOWN PREDICATE* on(_5003,_5002)
*UNKNOWN PREDICATE* on(_13733,_13732)
*UNKNOWN PREDICATE* on(_14631,_14631)
*UNKNOWN PREDICATE* on(_14726,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_14709)
*UNKNOWN PREDICATE* clear(_9892)
*UNKNOWN PREDICATE* clear(_5827)
*UNKNOWN PREDICATE* clear(_2480)

Inferences  =  5763
Unifications  =  9176
Time (msec)  =  7850
Speed (LIPS)  =  734
P = result(trans(d,e,c),result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0))))yes

?     empower(0,[],P,[on(d,a),clear(c),on(e,b)]);
*UNKNOWN PREDICATE* on(d,a)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(d,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(d,_1692)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* on(_3188,_3187)
*UNKNOWN PREDICATE* on(qqq(0),_3981)
*UNKNOWN PREDICATE* on(d,_4172)
*UNKNOWN PREDICATE* on(_4200,_4199)
*UNKNOWN PREDICATE* on(_4960,_4960)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* on(_56022,_56021)
*UNKNOWN PREDICATE* on(qqq(0),_56935)
*UNKNOWN PREDICATE* on(e,_57150)
*UNKNOWN PREDICATE* on(d,_57326)
*UNKNOWN PREDICATE* on(_57356,_57355)
*UNKNOWN PREDICATE* on(_58241,_58241)
*UNKNOWN PREDICATE* on(_58335,e)
```

```
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(_58318)
*UNKNOWN PREDICATE* clear(b)

Inferences   = 19176
Unifications = 30546
Time (msec)  = 26133
Speed (LIPS) =  733
P = result(trans(e,p,b),result(trans(b,e,d),result(trans(b,c,e),
result(trans(d,e,a),result(trans(b,a,c),result(trans(c,b,r),
result(trans(c,d,b),result(trans(b,c,a),.
result(trans(a,b,q),0)))))))))yes

?     empower(0,[],P,[on(d,q)]);
*UNKNOWN PREDICATE* on(d,q)
*UNKNOWN PREDICATE* clear(qqq(0))
*UNKNOWN PREDICATE* on(_660,_659)
*UNKNOWN PREDICATE* on(d,_1142)
*UNKNOWN PREDICATE* on(_1193,_1192)
*UNKNOWN PREDICATE* on(_1611,_1611)
*UNKNOWN PREDICATE* on(d,_1692)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* on(_3188,_3187)
*UNKNOWN PREDICATE* on(d,_32567)
*UNKNOWN PREDICATE* on(_32598,_32597)
*UNKNOWN PREDICATE* on(_33552,_33552)
*UNKNOWN PREDICATE* on(_33653,b)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(_33636)
*UNKNOWN PREDICATE* clear(_28170)
*UNKNOWN PREDICATE* clear(_23427)

Inferences   = 14719
Unifications = 23410
Time (msec)  = 19533
Speed (LIPS) =  753
P = result(trans(d,e,q),result(trans(a,q,b),result(trans(b,a,c),
result(trans(c,b,r),result(trans(c,d,b),result(trans(b,c,a),
result(trans(a,b,q),0)))))))yes

?     empower(0,[],P,[clear(c),clear(b),clear(a)]);
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_898,_897)
*UNKNOWN PREDICATE* on(qqq(0),_1653)
*UNKNOWN PREDICATE* on(_1704,_1703)
*UNKNOWN PREDICATE* on(_2395,_2395)
*UNKNOWN PREDICATE* on(_2497,c)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* on(_3989,_3988)
```

```
*UNKNOWN PREDICATE* on(qqq(0),_4782)
*UNKNOWN PREDICATE* on(b,_4973)
*UNKNOWN PREDICATE* on(_5001,_5000)
*UNKNOWN PREDICATE* on(_5761,_5761)
*UNKNOWN PREDICATE* on(_5842,b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(_5825)
*UNKNOWN PREDICATE* clear(_2480)
*UNKNOWN PREDICATE* clear(b)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* clear(a)
*UNKNOWN PREDICATE* on(_10676,_10675)
*UNKNOWN PREDICATE* on(qqq(0),_11553)
*UNKNOWN PREDICATE* on(_11618,_11617)
*UNKNOWN PREDICATE* on(_12421,_12421)
*UNKNOWN PREDICATE* on(_12549,a)
*UNKNOWN PREDICATE* clear(_12532)

Inferences   =  5395
Unifications =  8605
Time (msec)  =  7416
Speed (LIPS) =  727
P = result(trans(b,a,r),result(trans(b,c,a),result(trans(a,b,q),0)))yes

?     empower(0,[],P,[on(e,a)]);
*UNKNOWN PREDICATE* on(e,_8161)
*UNKNOWN PREDICATE* on(_8190,_8189)
*UNKNOWN PREDICATE* on(_9019,_9019)
*UNKNOWN PREDICATE* on(_9107,d)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(c)
*UNKNOWN PREDICATE* clear(d)
*UNKNOWN PREDICATE* clear(e)
*UNKNOWN PREDICATE* clear(p)
*UNKNOWN PREDICATE* on(_11436,_11435)
*UNKNOWN PREDICATE* on(qqq(0),_12361)
*UNKNOWN PREDICATE* clear(_31548)

Inferences   =  17120
Unifications =  27205
Time (msec)  =  22200
Speed (LIPS) =  771
P = result(trans(e,p,a),result(trans(b,a,c),result(trans(c,b,d),
result(trans(d,c,r),result(trans(d,e,c),result(trans(c,d,b),
result(trans(b,c,a),result(trans(a,b,q),0)))))))yes

?     quit;
```

Appendix F: Test2

This appendix contains all the files related to test2; they are:

1. The initial spider
2. The axiomatization of the problem domain
3. The input queries to Madame
4. The input queries to the planning component which did not use the spider
5. The final spider
6. Output from Madame
7. Output from the planning component which did not use the spider

1. **The following page is a listing of the initial spider.**

```
% frame axiom for the spider
% This specifies which statements which held in the parent state still hold.
spi_holds( Statement Cpss ) <-
      spi_parent( Ppss Cpss )
      spi_holds( Statement Ppss )
      spi_preserves( Statement Cpss );


% "spi_no" specifies how many p-s-s's there are.
% This is also the next available int. to be used as an identification number
% for a p-s-s.
spi_no(1);


% axioms for the start-state:
spi_poss( 0 );
spi_children( [] 0 );
spi_plan( [] 0 0 );
spi_holds( Stat 0 ) <-
      prouver( holds(Stat 0) );
```

2. For a listing of the UNIX axiomatization used in test2, see Appendix A.

3. The following page contains the input queries passed to Madame.

```
?time;

?setup;

?interface( [] [perms(group w /u/anatrudel/junk)] Plan );

?interface( [] [directory(/u/anatrudel/dir1)] Plan );

?interface( [] [owner(anatrudel /u/anatrudel/dir2)] Plan );

?interface( [] [not(fexist(/u/anatrudel/thesis/madame))] Plan );

?interface([] [directory(/u/anatrudel/dir1)
    perms(group w /u/anatrudel/dir1)] Plan);

?interface([] [directory(/u/anatrudel/junk)] Plan );

?interface([] [perms(group w /u/anatrudel/dir2)] Plan );

?interface([] [not(perms(group r /u/anatrudel))] Plan );

?interface([] [fexist(/u/anatrudel/dir3)] Plan);

?interface([] [grpmember_file(/u/anatrudel/dir3 the_group)] Plan);

?interface([] [perms(group w /u/anatrudel/dir2)] Plan );

?interface([] [not(perms(group r /u/anatrudel))] Plan );

?interface([] [not(perms(group r /u/anatrudel/thesis/madame))] Plan);

?interface([] [not(fexist(/u/anatrudel/dir5))] Plan );

?interface([] [not(fexist(/u/anatrudel/thesis/madame))] Plan);

?interface([] [not(owner(anatrudel /u/anatrudel/junk))] Plan);

?interface([] [not(owner(anatrudel /u/anatrudel/thesis/madame))] Plan);

?interface([] [directory(/u/anatrudel/dir1)
    perms(group w /u/anatrudel/dir1)] Plan);

?interface([] [directory(/u/anatrudel/junk)] Plan );

?interface([] [fexist(/u/anatrudel/dir2)
    perms(group w /u/anatrudel/dir2)] Plan);

?laFin;

?quit;
```

4. The following page contains the input queries passed to Warplan-Not.

```
?time;

?setup;

?empower(0 [] P [perms(group w /u/anatrudel/junk)] );

?empower(0 [] P [directory(/u/anatrudel/dir1)] );

?empower(0 [] P [owner(anatrudel /u/anatrudel/dir2)] );

?empower(0 [] P [not(fexist(/u/anatrudel/thesis/madame))] );

?empower(0 [] P [directory(/u/anatrudel/dir1)
    perms(group w /u/anatrudel/dir1)] );

?empower(0 [] P [directory(/u/anatrudel/junk)] );

?empower(0 [] P [perms(group w /u/anatrudel/dir2)] );

?empower(0 [] P [not(perms(group r /u/anatrudel))] );

?empower(0 [] P [fexist(/u/anatrudel/dir3)] );

?empower(0 [] P [grpmember_file(/u/anatrudel/dir3 the_group)] );

?empower(0 [] P [perms(group w /u/anatrudel/dir2)] );

?empower(0 [] P [not(perms(group r /u/anatrudel))] );

?empower(0 [] P [not(perms(group r /u/anatrudel/thesis/madame))] );

?empower(0 [] P [not(fexist(/u/anatrudel/dir5))] );

?empower(0 [] P [not(fexist(/u/anatrudel/thesis/madame))] );

?empower(0 [] P [not(owner(anatrudel /u/anatrudel/junk))] );

?empower(0 [] P [not(owner(anatrudel /u/anatrudel/thesis/madame))] );

?empower(0 [] P [directory(/u/anatrudel/dir1)
    perms(group w /u/anatrudel/dir1)]);

?empower(0 [] P [directory(/u/anatrudel/junk)] );

?empower(0 [] P [fexist(/u/anatrudel/dir2)
    perms(group w /u/anatrudel/dir2)]);

?quit;
```

5. **The following pages contain a listing of the final spider produced from test2.**

```
spi_holds(Statement,Cpss) <-
        spi_parent(Ppss,Cpss),
        spi_holds(Statement,Ppss),
        spi_preserves(Statement,Cpss);

spi_holds(Stat,0) <-
        prouver(holds(Stat,0));

spi_holds(perms(group,w,"/u/anatrudel/junk"),1);
spi_holds(grpmember_file("/u/anatrudel/dir1",the_group),2);
spi_holds(perms(user,r,"/u/anatrudel/dir1"),2);
spi_holds(perms(user,w,"/u/anatrudel/dir1"),2);
spi_holds(perms(user,x,"/u/anatrudel/dir1"),2);
spi_holds(perms(group,r,"/u/anatrudel/dir1"),2);
spi_holds(perms(group,x,"/u/anatrudel/dir1"),2);
spi_holds(perms(general,x,"/u/anatrudel/dir1"),2);
spi_holds(owner(anatrudel,"/u/anatrudel/dir1"),2);
spi_holds(directory("/u/anatrudel/dir1"),2);
spi_holds(fexist("/u/anatrudel/dir1"),2);
spi_holds(grpmember_file("/u/anatrudel/dir2",the_group),3);
spi_holds(perms(user,r,"/u/anatrudel/dir2"),3);
spi_holds(perms(user,w,"/u/anatrudel/dir2"),3);
spi_holds(perms(user,x,"/u/anatrudel/dir2"),3);
spi_holds(perms(group,r,"/u/anatrudel/dir2"),3);
spi_holds(perms(group,x,"/u/anatrudel/dir2"),3);
spi_holds(perms(general,x,"/u/anatrudel/dir2"),3);
spi_holds(owner(anatrudel,"/u/anatrudel/dir2"),3);
spi_holds(directory("/u/anatrudel/dir2"),3);
spi_holds(fexist("/u/anatrudel/dir2"),3);
spi_holds(perms(group,w,"/u/anatrudel/dir1"),5);
spi_holds(grpmember_file("/u/anatrudel/junk",the_group),6);
spi_holds(perms(user,r,"/u/anatrudel/junk"),6);
spi_holds(perms(user,w,"/u/anatrudel/junk"),6);
spi_holds(perms(user,x,"/u/anatrudel/junk"),6);
spi_holds(perms(group,r,"/u/anatrudel/junk"),6);
spi_holds(perms(group,x,"/u/anatrudel/junk"),6);
spi_holds(perms(general,x,"/u/anatrudel/junk"),6);
spi_holds(owner(anatrudel,"/u/anatrudel/junk"),6);
spi_holds(directory("/u/anatrudel/junk"),6);
spi_holds(fexist("/u/anatrudel/junk"),6);
spi_holds(grpmember_file("/u/anatrudel/dir2",the_group),7);
spi_holds(owner(anatrudel,"/u/anatrudel/dir2"),7);
spi_holds(directory("/u/anatrudel/dir2"),7);
spi_holds(fexist("/u/anatrudel/dir2"),7);
spi_holds(perms(group,w,"/u/anatrudel/dir2"),7);
spi_holds(perms(user,x,"/u/anatrudel"),8);
spi_holds(grpmember_file("/u/anatrudel/dir3",the_group),9);
spi_holds(perms(user,r,"/u/anatrudel/dir3"),9);
spi_holds(perms(user,w,"/u/anatrudel/dir3"),9);
spi_holds(perms(user,x,"/u/anatrudel/dir3"),9);
spi_holds(perms(group,r,"/u/anatrudel/dir3"),9);
spi_holds(perms(group,x,"/u/anatrudel/dir3"),9);
spi_holds(perms(general,x,"/u/anatrudel/dir3"),9);
spi_holds(owner(anatrudel,"/u/anatrudel/dir3"),9);
```

```
spi_holds(directory("/u/anatrudel/dir3"),9);
spi_holds(fexist("/u/anatrudel/dir3"),9);


spi_no(11);


spi_poss(0);
spi_poss(1);
spi_poss(2);
spi_poss(3);
spi_poss(4);
spi_poss(5);
spi_poss(6);
spi_poss(7);
spi_poss(8);
spi_poss(9);
spi_poss(10);


spi_parent(0,1);
spi_parent(0,2);
spi_parent(0,3);
spi_parent(0,4);
spi_parent(2,5);
spi_parent(0,6);
spi_parent(0,7);
spi_parent(0,8);
spi_parent(0,9);
spi_parent(0,10);


spi_children([],1);
spi_children([],3);
spi_children([],4);
spi_children([],5);
spi_children([5],2);
spi_children([],6);
spi_children([],7);
spi_children([],8);
spi_children([],9);
spi_children([],10);
spi_children([10,9,8,7,6,4,3,2,1],0);


spi_plan([],0,0);
spi_plan([chmod(20,"/u/anatrudel/junk")],1,1);
spi_plan([mkdir("/u/anatrudel/dir1")],1,2);
spi_plan([mkdir("/u/anatrudel/dir2")],1,3);
spi_plan([rm("/u/anatrudel/thesis/madame")],1,4);
spi_plan([chmod(20,"/u/anatrudel/dir1")],1,5);
spi_plan([rm("/u/anatrudel/junk"),mkdir("/u/anatrudel/junk")],2,6);
spi_plan([mkdir("/u/anatrudel/dir2"),chmod(20,"/u/anatrudel/dir2")],2,7);
spi_plan([chmod(100,"/u/anatrudel")],1,8);
```

```
spi_plan([mkdir("/u/anatrudel/dir3")],1,9);
spi_plan([rm("/u/anatrudel/junk")],1,10);


spi_preserves(_5075,1) <-
      diff(_5075,perms(_4981,_4982,"/u/anatrudel/junk"));
spi_preserves(_7423,2);
spi_preserves(_8234,3);
spi_preserves(_5069,4) <-
      diff(_5069,fexist("/u/anatrudel/thesis/madame")),
      diff(_5069,owner(_,"/u/anatrudel/thesis/madame")),
      diff(_5069,perms(_,_,"/u/anatrudel/thesis/madame")),
      diff(_5069,grpmember_file("/u/anatrudel/thesis/madame",_)),
      diff(_5069,link("/u/anatrudel/thesis/madame",_)),
      diff(_5069,link(_,"/u/anatrudel/thesis/madame"));
spi_preserves(_31101,5) <-
      diff(_31101,perms(_31007,_31008,"/u/anatrudel/dir1"));
spi_preserves(_14295,6) <-
      diff(_14295,owner(_,"/u/anatrudel/junk")),
      diff(_14295,perms(_,_,"/u/anatrudel/junk")),
      diff(_14295,grpmember_file("/u/anatrudel/junk",_)),
      diff(_14295,link("/u/anatrudel/junk",_)),
      diff(_14295,link(_,"/u/anatrudel/junk"));
spi_preserves(_15755,7) <-
      diff(_15755,perms(_15660,_15661,"/u/anatrudel/dir2"));
spi_preserves(_4434,8) <-
      diff(_4434,perms(_4340,_4341,"/u/anatrudel"));
spi_preserves(_9269,9);
spi_preserves(_6842,10) <-
      diff(_6842,fexist("/u/anatrudel/junk")),
      diff(_6842,owner(_,"/u/anatrudel/junk")),
      diff(_6842,perms(_,_,"/u/anatrudel/junk")),
      diff(_6842,grpmember_file("/u/anatrudel/junk",_)),
      diff(_6842,link("/u/anatrudel/junk",_)),
      diff(_6842,link(_,"/u/anatrudel/junk"));
```

6. **The following pages contain the output produced by Madame during test2.**

```
Waterloo Unix Prolog [Release 1.2 -- Apr 9,1984]

?     time;
timing on

Inferences  =     5
Unifications =    6
Time (msec)  =    50
Speed (LIPS) =   100
yes

?     setup;

Inferences  =    11
Unifications =   14
Time (msec)  =   183
Speed (LIPS) =    60
yes

?     interface([],[perms(group,w,"/u/anatrudel/junk")],Plan);


The list of goals to satisfy is:
[perms(group,w,/u/anatrudel/junk)]


*UNKNOWN PREDICATE* spi_parent(Ppss,Cpss)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/junk),0)


The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[perms(group,w,/u/anatrudel/junk)]


*UNKNOWN PREDICATE* only(perms(group,w,/u/anatrudel/junk))
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/junk)
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/junk)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* imposs(_3277)
*UNKNOWN PREDICATE* imposs(_3304)
*UNKNOWN PREDICATE* only(valid_chmod_num(20))
*UNKNOWN PREDICATE* current_user(_3024)
*UNKNOWN PREDICATE* current_user(_3024)
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel/junk))
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/junk)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/junk)


The Spider has been modified
```

```
   New p-s-s added is:    1    It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [chmod(20,/u/anatrudel/junk)]


*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/junk),0)



*** The plan generated is: ***


Inferences   =  1424
Unifications =  2467
Time (msec)  =  8783
Speed (LIPS) =  162
Plan = result(chmod(20,"/u/anatrudel/junk"),0)yes

?      interface([],[directory("/u/anatrudel/dir1")],Plan);


The list of goals to satisfy is:
[directory(/u/anatrudel/dir1)]


*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/dir1),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/dir1),0)



The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[directory(/u/anatrudel/dir1)]



The Spider has been modified
   New p-s-s added is:    2    It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [mkdir(/u/anatrudel/dir1)]



*** The plan generated is: ***
```

```
Inferences  =  2129
Unifications  =  3487
Time (msec)  =  5633
Speed (LIPS)  =  377
Plan = result(mkdir("/u/anatrudel/dir1"),0)yes

?     interface([],[owner(anatrudel,"/u/anatrudel/dir2")],Plan);


The list of goals to satisfy is: .
[owner(anatrudel,/u/anatrudel/dir2)]


*UNKNOWN PREDICATE* holds(owner(anatrudel,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* holds(owner(anatrudel,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* holds(owner(anatrudel,/u/anatrudel/dir2),0)


The closest p-s-s is:  0
The distance is:      1
The goals which do not hold in the p-s-s are:
[owner(anatrudel,/u/anatrudel/dir2)]


*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel/dir2))
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* holds(owner(anatrudel,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_1150))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_1150))
*UNKNOWN PREDICATE* imposs(_1460)
*UNKNOWN PREDICATE* imposs(_1487)
*UNKNOWN PREDICATE* only(current_user(anatrudel))
*UNKNOWN PREDICATE* current_user(anatrudel)
*UNKNOWN PREDICATE* current_user(anatrudel)


The Spider has been modified
   New p-s-s added is:    3   It's parent is:      0
   The list of actions which take us from the parent to the new p-s-s is:
   [mkdir(/u/anatrudel/dir2)]


*** The plan generated is: ***
```

```
Inferences  =  2357
Unifications  =  3895
Time (msec)  =  5516
Speed (LIPS)  =  427
Plan = result(mkdir("/u/anatrudel/dir2"),0)yes

?     interface([],[not(fexist("/u/anatrudel/thesis/madame"))],Plan);


The list of goals to satisfy is:
[not(fexist(/u/anatrudel/thesis/madame))]



The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[not(fexist(/u/anatrudel/thesis/madame))]


*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* imposs(_1674)
*UNKNOWN PREDICATE* imposs(_1701)


The Spider has been modified
   New p-s-s added is:    4   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [rm(/u/anatrudel/thesis/madame)]



*** The plan generated is: ***


Inferences  =  1674
Unifications  =  2587
Time (msec)  =  3333
Speed (LIPS)  =  502
Plan = result(rm("/u/anatrudel/thesis/madame"),0)yes

?     interface([],[directory("/u/anatrudel/dir1"),
perms(group,w,"/u/anatrudel/dir1")],Plan);
```

The list of goals to satisfy is:
[directory(/u/anatrudel/dir1),perms(group,w,/u/anatrudel/dir1)]

UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)

The closest p-s-s is: 2
The distance is: 1
The goals which do not hold in the p-s-s are:
[perms(group,w,/u/anatrudel/dir1)]

The Spider has been modified
New p-s-s added is: 5 It's parent is: 2
The list of actions which take us from the parent to the new p-s-s is:
[chmod(20,/u/anatrudel/dir1)]

UNKNOWN PREDICATE holds(umask(group,w),0)

*** The plan generated is: ***

Inferences = 7474
Unifications = 13092
Time (msec) = 13950
Speed (LIPS) = 535
Plan = result(chmod(20,"/u/anatrudel/dir1"),
result(mkdir("/u/anatrudel/dir1"),0))yes

? interface([],[directory("/u/anatrudel/junk")],Plan);

The list of goals to satisfy is:
[directory(/u/anatrudel/junk)]

```
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)


The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[directory(/u/anatrudel/junk)]



The Spider has been modified
   New p-s-s added is:    6   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [rm(/u/anatrudel/junk),mkdir(/u/anatrudel/junk)]



*** The plan generated is: ***


Inferences   =  4279
Unifications =  7234
Time (msec)  =  8633
Speed (LIPS) =  495
Plan = result(mkdir("/u/anatrudel/junk"),
result(rm("/u/anatrudel/junk"),0))yes

?      interface([],[perms(group,w,"/u/anatrudel/dir2")],Plan);


The list of goals to satisfy is:
[perms(group,w,/u/anatrudel/dir2)]


*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)


The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[perms(group,w,/u/anatrudel/dir2)]
```

```
*UNKNOWN PREDICATE* only(perms(group,w,/u/anatrudel/dir2))
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_2376))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_2376))


The Spider has been modified
   New p-s-s added is:    7   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [mkdir(/u/anatrudel/dir2),chmod(20,/u/anatrudel/dir2)]



*** The plan generated is: ***


Inferences   =   4301
Unifications =   8012
Time (msec)  =   9933
Speed (LIPS) =    432
Plan = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

?      interface([],[not(perms(group,r,"/u/anatrudel"))],Plan);


The list of goals to satisfy is:
[not(perms(group,r,/u/anatrudel))]



The closest p-s-s is:  0
The distance is:       1
The goals which do not hold in the p-s-s are:
[not(perms(group,r,/u/anatrudel))]



The Spider has been modified
   New p-s-s added is:    8   It's parent is:        0
   The list of actions which take us from the parent to the new p-s-s is:
   [chmod(100,/u/anatrudel)]
```

*** The plan generated is: ***

Inferences = 1302
Unifications = 2424
Time (msec) = 2550
Speed (LIPS) = 510
Plan = result(chmod(100,"/u/anatrudel"),0)yes

? interface([],[fexist("/u/anatrudel/dir3")],Plan);

The list of goals to satisfy is:
[fexist(/u/anatrudel/dir3)]

UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)
UNKNOWN PREDICATE holds(fexist(/u/anatrudel/dir3),0)

The closest p-s-s is: 0
The distance is: 1
The goals which do not hold in the p-s-s are:
[fexist(/u/anatrudel/dir3)]

The Spider has been modified
New p-s-s added is: 9 It's parent is: 0
The list of actions which take us from the parent to the new p-s-s is:
[mkdir(/u/anatrudel/dir3)]

*** The plan generated is: ***

Inferences = 2550
Unifications = 4629
Time (msec) = 6000

```
Speed (LIPS) =   425
Plan = result(mkdir("/u/anatrudel/dir3"),0)yes

?      interface([],[grpmember_file("/u/anatrudel/dir3",the_group)],Plan);


The list of goals to satisfy is:
[grpmember_file(/u/anatrudel/dir3,the_group)]


The closest p-s-s is:  9
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences   =   248
Unifications =   427
Time (msec)  =   600
Speed (LIPS) =   413
Plan = result(mkdir("/u/anatrudel/dir3"),0)yes

?      interface([],[perms(group,w,"/u/anatrudel/dir2")],Plan);


The list of goals to satisfy is:
[perms(group,w,/u/anatrudel/dir2)]


*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)


The closest p-s-s is:  7
The distance is:       0
The goals which do not hold in the p-s-s are:
[]
```

```
Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences   =   374
Unifications =  1021
Time (msec)  =   866
Speed (LIPS) =   431
Plan = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

?      interface([],[not(perms(group,r,"/u/anatrudel"))],Plan);


The list of goals to satisfy is:
[not(perms(group,r,/u/anatrudel))]

*UNKNOWN PREDICATE* holds(perms(group,r,/u/anatrudel),0)


The closest p-s-s is:  8
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences   =   315
Unifications =   675
Time (msec)  =   683
Speed (LIPS) =   460
Plan = result(chmod(100,"/u/anatrudel"),0)yes

?      interface([],[not(perms(group,r,"/u/anatrudel/thesis/madame"))],Plan);
```

```
The list of goals to satisfy is:
[not(perms(group,r,/u/anatrudel/thesis/madame))]


The closest p-s-s is:  4
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences   =   549
Unifications =  1221
Time (msec)  =  1033
Speed (LIPS) =   531
Plan = result(rm("/u/anatrudel/thesis/madame"),0)yes

?      interface([],[not(fexist("/u/anatrudel/dir5"))],Plan);


The list of goals to satisfy is:
[not(fexist(/u/anatrudel/dir5))]

*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir5),0)


The closest p-s-s is:  0
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in start state


*** The plan generated is: ***
```

```
Inferences   =   170
Unifications =   232
Time (msec)  =   400
Speed (LIPS) =   425
Plan = 0yes

?     interface([],[not(fexist("/u/anatrudel/thesis/madame"))],Plan);


The list of goals to satisfy is:
[not(fexist(/u/anatrudel/thesis/madame))]


*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/thesis/madame),0)


The closest p-s-s is:  4
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences   =   537
Unifications =  1069
Time (msec)  =   916
Speed (LIPS) =   585
Plan = result(rm("/u/anatrudel/thesis/madame"),0)yes

?     interface([],[not(owner(anatrudel,"/u/anatrudel/junk"))],Plan);


The list of goals to satisfy is:
[not(owner(anatrudel,/u/anatrudel/junk))]


*UNKNOWN PREDICATE* holds(owner(anatrudel,/u/anatrudel/junk),0)


The closest p-s-s is:  0
The distance is:       1
```

The goals which do not hold in the p-s-s are:
[not(owner(anatrudel,/u/anatrudel/junk))]

UNKNOWN PREDICATE owner(anatrudel,/u/anatrudel/junk)
UNKNOWN PREDICATE owner(anatrudel,/u/anatrudel/junk)
UNKNOWN PREDICATE replace(parent(/u/anatrudel/junk,_3283))
UNKNOWN PREDICATE replace(parent(/u/anatrudel/junk,_3283))
UNKNOWN PREDICATE imposs(_3553)
UNKNOWN PREDICATE imposs(_3580)
UNKNOWN PREDICATE only(fexist(/u/anatrudel/junk))
UNKNOWN PREDICATE fexist(/u/anatrudel/junk)
UNKNOWN PREDICATE fexist(/u/anatrudel/junk)
UNKNOWN PREDICATE directory(/u/anatrudel/junk)
UNKNOWN PREDICATE directory(/u/anatrudel/junk)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/junk),0)
UNKNOWN PREDICATE current_user(_3482)

The Spider has been modified
New p-s-s added is: 10 It's parent is: 0
The list of actions which take us from the parent to the new p-s-s is:
[rm(/u/anatrudel/junk)]

*** The plan generated is: ***

Inferences = 2058
Unifications = 3601
Time (msec) = 3983
Speed (LIPS) = 516
Plan = result(rm("/u/anatrudel/junk"),0)yes

? interface([],[not(owner(anatrudel,"/u/anatrudel/thesis/madame"))],
Plan);

The list of goals to satisfy is:
[not(owner(anatrudel,/u/anatrudel/thesis/madame))]

The closest p-s-s is: 4
The distance is: 0
The goals which do not hold in the p-s-s are:

[]

Spider not modified - Goals hold in the closest p-s-s

*** The plan generated is: ***

Inferences = 639
Unifications = 1314
Time (msec) = 1283
Speed (LIPS) = 497
Plan = result(rm("/u/anatrudel/thesis/madame"),0)yes

? interface([],[directory("/u/anatrudel/dir1"),
perms(group,w,"/u/anatrudel/dir1")],Plan);

The list of goals to satisfy is:
[directory(/u/anatrudel/dir1),perms(group,w,/u/anatrudel/dir1)]

UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(directory(/u/anatrudel/dir1),0)
UNKNOWN PREDICATE holds(perms(group,w,/u/anatrudel/dir1),0)

The closest p-s-s is: 5
The distance is: 0
The goals which do not hold in the p-s-s are:
[]

Spider not modified - Goals hold in the closest p-s-s

*** The plan generated is: ***

```
Inferences  =  1021
Unifications =  3270
Time (msec)  =  2833
Speed (LIPS) =  360
Plan = result(chmod(20,"/u/anatrudel/dir1"),
result(mkdir("/u/anatrudel/dir1"),0))yes

?     interface([],[directory("/u/anatrudel/junk")],Plan);


The list of goals to satisfy is:
[directory(/u/anatrudel/junk)]


The closest p-s-s is:  6
The distance is:       0
The goals which do not hold in the p-s-s are:
[]


Spider not modified - Goals hold in the closest p-s-s


*** The plan generated is: ***


Inferences  =  492
Unifications =  1061
Time (msec)  =  1116
Speed (LIPS) =  440
Plan = result(mkdir("/u/anatrudel/junk"),
result(rm("/u/anatrudel/junk"),0))yes

?     interface([],[fexist("/u/anatrudel/dir2"),
perms(group,w,"/u/anatrudel/dir2")],Plan);


The list of goals to satisfy is:
[fexist(/u/anatrudel/dir2),perms(group,w,/u/anatrudel/dir2)]


The closest p-s-s is:  7
The distance is:       0
The goals which do not hold in the p-s-s are:
```

[]

Spider not modified - Goals hold in the closest p-s-s

*** The plan generated is: ***

Inferences = 573
Unifications = 1674
Time (msec) = 1500
Speed (LIPS) = 382
Plan = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

? laFin;

Inferences = 10
Unifications = 10
Time (msec) = 1533
Speed (LIPS) = 6
yes

? quit;

7. The following pages contain the output from Warplan-Not.

```
Waterloo Unix Prolog [Release 1.2 -- Apr 9,1984]

?     time;
timing on

Inferences  =     5
Unifications =    6
Time (msec)  =    66
Speed (LIPS) =    75
yes

?     setup;

Inferences  =    11
Unifications =   14
Time (msec)  =   200
Speed (LIPS) =    55
yes

?     empower(0,[],P,[perms(group,w,"/u/anatrudel/junk")]);
*UNKNOWN PREDICATE* only(perms(group,w,/u/anatrudel/junk))

Inferences  =  1057
Unifications =  1885
Time (msec)  =  4866
Speed (LIPS) =   217
P = result(chmod(20,"/u/anatrudel/junk"),0)yes

?     empower(0,[],P,[directory("/u/anatrudel/dir1")]);
*UNKNOWN PREDICATE* only(directory(/u/anatrudel/dir1))
*UNKNOWN PREDICATE* directory(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* directory(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/dir1),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir1,_382))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir1,_382))
*UNKNOWN PREDICATE* imposs(_678)
*UNKNOWN PREDICATE* imposs(_705)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir1),0)
*UNKNOWN PREDICATE* only(fexist(/u/anatrudel))
*UNKNOWN PREDICATE* fexist(/u/anatrudel)

Inferences  =  1204
Unifications =  1950
Time (msec)  =  2516
Speed (LIPS) =   478
P = result(mkdir("/u/anatrudel/dir1"),0)yes

?     empower(0,[],P,[owner(anatrudel,"/u/anatrudel/dir2")]);
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel/dir2))

Inferences  =  1373
Unifications =  2229
```

```
Time (msec)  =  2333
Speed (LIPS) =  588
P = result(mkdir("/u/anatrudel/dir2"),0)yes

?     empower(0,[],P,[not(fexist("/u/anatrudel/thesis/madame"))]);
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* current_user(_640)
*UNKNOWN PREDICATE* current_user(_640)
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel/thesis))
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/thesis)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/thesis)
*UNKNOWN PREDICATE* only(perms(user,w,/u/anatrudel/thesis))
*UNKNOWN PREDICATE* perms(user,w,/u/anatrudel/thesis)
*UNKNOWN PREDICATE* perms(user,w,/u/anatrudel/thesis)

Inferences  =  1049
Unifications =  1656
Time (msec)  =  2016
Speed (LIPS) =  520
P = result(rm("/u/anatrudel/thesis/madame"),0)yes

?     empower(0,[],P,[directory("/u/anatrudel/dir1"),
perms(group,w,"/u/anatrudel/dir1")]);
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir1,_382))
*UNKNOWN PREDICATE* imposs(_678)
*UNKNOWN PREDICATE* imposs(_705)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir1),0)
*UNKNOWN PREDICATE* only(fexist(/u/anatrudel))

Inferences  =  7893
Unifications =  13304
Time (msec)  =  14000
Speed (LIPS) =  563
P = result(chmod(20,"/u/anatrudel/dir1"),
result(mkdir("/u/anatrudel/dir1"),0))yes

?     empower(0,[],P,[directory("/u/anatrudel/junk")]);
*UNKNOWN PREDICATE* only(directory(/u/anatrudel/junk))
*UNKNOWN PREDICATE* directory(/u/anatrudel/junk)
*UNKNOWN PREDICATE* directory(/u/anatrudel/junk)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_382))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_382))
*UNKNOWN PREDICATE* imposs(_678)
*UNKNOWN PREDICATE* imposs(_705)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/junk)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/junk)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_1168))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_1168))

Inferences  =  2353
```

```
Unifications = 3837
Time (msec) = 3933
Speed (LIPS) = 598
P = result(mkdir("/u/anatrudel/junk"),
result(rm("/u/anatrudel/junk"),0))yes

? empower(0,[],P,[perms(group,w,"/u/anatrudel/dir2")]);
*UNKNOWN PREDICATE* only(perms(group,w,/u/anatrudel/dir2))
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_438))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_438))
*UNKNOWN PREDICATE* imposs(_752)
*UNKNOWN PREDICATE* imposs(_779)
*UNKNOWN PREDICATE* only(umask(group,w))
*UNKNOWN PREDICATE* umask(group,w)
*UNKNOWN PREDICATE* umask(group,w)
*UNKNOWN PREDICATE* holds(umask(group,w),0)
*UNKNOWN PREDICATE* imposs(_1339)
*UNKNOWN PREDICATE* imposs(_1366)
*UNKNOWN PREDICATE* only(umask(group,w))
*UNKNOWN PREDICATE* umask(group,w)
*UNKNOWN PREDICATE* umask(group,w)

Inferences = 2711
Unifications = 4531
Time (msec) = 5083
Speed (LIPS) = 533
P = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

? empower(0,[],P,[not(perms(group,r,"/u/anatrudel"))]);
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel)
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel)
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* imposs(_493)
*UNKNOWN PREDICATE* imposs(_520)
*UNKNOWN PREDICATE* current_user(_293)
*UNKNOWN PREDICATE* current_user(_293)
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel))
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel)

Inferences = 413
Unifications = 703
Time (msec) = 916
Speed (LIPS) = 450
P = result(chmod(100,"/u/anatrudel"),0)yes

? empower(0,[],P,[fexist("/u/anatrudel/dir3")]);
*UNKNOWN PREDICATE* only(fexist(/u/anatrudel/dir3))
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir3)
```

```
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir3)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir3),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir3,_380))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir3,_380))
*UNKNOWN PREDICATE* imposs(_674)
*UNKNOWN PREDICATE* imposs(_701)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir3)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir3)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir3),0)
*UNKNOWN PREDICATE* only(fexist(/u/anatrudel))
*UNKNOWN PREDICATE* fexist(/u/anatrudel)
*UNKNOWN PREDICATE* fexist(/u/anatrudel)
*UNKNOWN PREDICATE* only(directory(/u/anatrudel))
*UNKNOWN PREDICATE* directory(/u/anatrudel)
*UNKNOWN PREDICATE* directory(/u/anatrudel)
*UNKNOWN PREDICATE* current_user(_603)
*UNKNOWN PREDICATE* current_user(_603)

Inferences  =  1204
Unifications  =  1952
Time (msec)  =  2033
Speed (LIPS)  =  592
P = result(mkdir("/u/anatrudel/dir3"),0)yes

?     empower(0,[],P,[grpmember_file("/u/anatrudel/dir3",the_group)]);
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir3,_408))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir3,_408))
*UNKNOWN PREDICATE* imposs(_725)

Inferences  =  1373
Unifications  =  2229
Time (msec)  =  2416
Speed (LIPS)  =  568
P = result(mkdir("/u/anatrudel/dir3"),0)yes

?     empower(0,[],P,[perms(group,w,"/u/anatrudel/dir2")]);
*UNKNOWN PREDICATE* only(perms(group,w,/u/anatrudel/dir2))
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* perms(group,w,/u/anatrudel/dir2)
*UNKNOWN PREDICATE* holds(perms(group,w,/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_438))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_438))
*UNKNOWN PREDICATE* imposs(_752)
*UNKNOWN PREDICATE* imposs(_779)
*UNKNOWN PREDICATE* only(umask(group,w))
*UNKNOWN PREDICATE* umask(group,w)
*UNKNOWN PREDICATE* umask(group,w)
*UNKNOWN PREDICATE* holds(umask(group,w),0)
*UNKNOWN PREDICATE* imposs(_1339)
*UNKNOWN PREDICATE* imposs(_1366)
*UNKNOWN PREDICATE* only(umask(group,w))

Inferences  =  2711
Unifications  =  4531
```

```
Time (msec)  =  5033
Speed (LIPS) =  538
P = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

?     empower(0,[],P,[not(perms(group,r,"/u/anatrudel"))]);
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel)
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel)
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* imposs(_493)
*UNKNOWN PREDICATE* imposs(_520)
*UNKNOWN PREDICATE* current_user(_293)
*UNKNOWN PREDICATE* current_user(_293)
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel))
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel)

Inferences  =  413
Unifications =  703
Time (msec)  =  716
Speed (LIPS) =  576
P = result(chmod(100,"/u/anatrudel"),0)yes

?     empower(0,[],P,[not(perms(group,r,"/u/anatrudel/thesis/madame"))]);
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* perms(group,r,/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* replace(valid_chmod_num(_253))
*UNKNOWN PREDICATE* imposs(_493)
*UNKNOWN PREDICATE* imposs(_520)
*UNKNOWN PREDICATE* current_user(_293)

Inferences  =  413
Unifications =  709
Time (msec)  =  866
Speed (LIPS) =  476
P = result(chmod(100,"/u/anatrudel/thesis/madame"),0)yes

?     empower(0,[],P,[not(fexist("/u/anatrudel/dir5"))]);
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir5)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir5)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir5),0)

Inferences  =  73
Unifications =  135
Time (msec)  =  133
Speed (LIPS) =  547
P = 0yes

?     empower(0,[],P,[not(fexist("/u/anatrudel/thesis/madame"))]);
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/thesis/madame)
```

```
Inferences  =  1049
Unifications  =  1656
Time (msec)  =  1766
Speed (LIPS)  =  593
P = result(rm("/u/anatrudel/thesis/madame"),0)yes

?     empower(0,[],P,[not(owner(anatrudel,"/u/anatrudel/junk"))]);
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/junk)

Inferences  =  995
Unifications  =  1592
Time (msec)  =  1666
Speed (LIPS)  =  597
P = result(rm("/u/anatrudel/junk"),0)yes

?     empower(0,[],P,[not(owner(anatrudel,"/u/anatrudel/thesis/madame"))]);
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* owner(anatrudel,/u/anatrudel/thesis/madame)
*UNKNOWN PREDICATE* only(owner(anatrudel,/u/anatrudel/thesis))

Inferences  =  1052
Unifications  =  1661
Time (msec)  =  2216
Speed (LIPS)  =  474
P = result(rm("/u/anatrudel/thesis/madame"),0)yes

?     empower(0,[],P,[directory("/u/anatrudel/dir1"),
perms(group,w,"/u/anatrudel/dir1")]);
*UNKNOWN PREDICATE* only(directory(/u/anatrudel/dir1))
*UNKNOWN PREDICATE* directory(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* directory(/u/anatrudel/dir1)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/dir1),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir1,_382))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir1,_382))
*UNKNOWN PREDICATE* imposs(_678)

Inferences  =  7893
Unifications  =  13304
Time (msec)  =  15616
Speed (LIPS)  =  505
P = result(chmod(20,"/u/anatrudel/dir1"),
result(mkdir("/u/anatrudel/dir1"),0))yes

?     empower(0,[],P,[directory("/u/anatrudel/junk")]);
*UNKNOWN PREDICATE* only(directory(/u/anatrudel/junk))
*UNKNOWN PREDICATE* directory(/u/anatrudel/junk)
*UNKNOWN PREDICATE* directory(/u/anatrudel/junk)
*UNKNOWN PREDICATE* holds(directory(/u/anatrudel/junk),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_382))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_382))
*UNKNOWN PREDICATE* imposs(_678)
*UNKNOWN PREDICATE* imposs(_705)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/junk)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/junk)
```

```
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_1168))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/junk,_1168))
*UNKNOWN PREDICATE* imposs(_1438)

Inferences  =  2353
Unifications  =  3837
Time (msec)  =  4316
Speed (LIPS)  =  545
P = result(mkdir("/u/anatrudel/junk"),
result(rm("/u/anatrudel/junk"),0))yes

?     empower(0,[],P,[fexist("/u/anatrudel/dir2"),
perms(group,w,"/u/anatrudel/dir2")]);
*UNKNOWN PREDICATE* only(fexist(/u/anatrudel/dir2))
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir2)
*UNKNOWN PREDICATE* fexist(/u/anatrudel/dir2)
*UNKNOWN PREDICATE* holds(fexist(/u/anatrudel/dir2),0)
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_380))
*UNKNOWN PREDICATE* replace(parent(/u/anatrudel/dir2,_380))

Inferences  =  7902
Unifications  =  13324
Time (msec)  =  14883
Speed (LIPS)  =  530
P = result(chmod(20,"/u/anatrudel/dir2"),
result(mkdir("/u/anatrudel/dir2"),0))yes

?     quit;
```