A formal specification and rigorous implementation of an AI planner

T.L.McCluskey,
School of Computing and Maths,
Univ of Huddersfield, UK

August 1, 2000

Abstract

This is an abridged version of chapters 6 and 7 in [Turner and McCluskey 94], which were originally derived from [McCluskey 88]. It introduces the reader to Planning as viewed from an Artificial Intelligence perspective. It also shows how a planning algorithm can be formally specified and then transformed into executable code. Note that the specification was developed without the use of any tools, hence it may contain some syntactic bugs. The planner implementation (and its derivatives) have been used frequently for nearly ten years: in that time no semantic bugs have been found. Hence the application could also be viewed as case study evidence supporting the power of formal techniques.

After introducing the reader to planning in Sections 1 and 2, in Section 3 we describe an abstract specification of a planner, In Section 4 we introduce a design level solution to planning in the form of a goal directed algorithm whose basic operation is to achieve goals within a developing, partially ordered plan. We then progress to modelling the plan as a VDM state, and then specify the 'achieve' operations on this state. In Sections 5,6 and 7 we show how the design specification may be implemented faithfully using Logic Programming.

1. Introduction

One interesting application area in which formal specification can be used is that of *Artificial Intelligence* (AI). This involves the creation of computer systems which perform tasks normally associated with human intelligence, such as planning, reasoning, vision, natural language understanding and learning. AI applications tend to be complex, leading to huge implementations. In some cases, scientists create theories and models of intelligence with which to guide or base their computational models. With such large engineering tasks an interesting question arises: how do scientists know that their computer models have been implemented faithfully? One approach is to use a formal specification as a *bridge* that links the high level model at one extreme, and the implementation at the other. That way, the model can be mapped to the specification, which can then itself be prototyped, or used as a contract with respect to which the correctness of the implementation is checked. An easy mistake to make, especially in complex application areas such as those in AI, is to assume that if the problem is ill-defined, formal specification techniques are not applicable. "Such and such an area is not capable of being fully captured, therefore formal specification is not appropriate" one might say. This misses the point: if a complex program is to be written to simulate an un-fathomable application then the program itself can (and should) be formally specified, even though the area it is approximating cannot.

In the first part of this paper we will develop a VDM specification of the main procedure in a *planning* program, that is a program which generates plans automatically. Specifically, our design level specification in section 4 captures the goal achievement procedure in a "Constraint Posting Non-Linear Conjunctive Planner" (the reader is referred to [Chapman 87] for the background on this). In the second part of the paper we will prototype this specification using Prolog. As with any substantial case study, however, the reader must become familiar with the application area, and we devote several pages to giving a simple introduction to planning. Those who need more information on automatic planning could consult textbooks on Artificial Intelligence[1].

2. Automatic Planning

Planning is what we do when we assemble orderings of actions to achieve goals. These orderings are *temporal* - they involve the concept of time. The idea of using a partial ordering to represent temporal relations between actions is quite common. As well as actions, we must represent objects that are being acted on, and properties and relationships between these objects. In the model developed here, we will often refer to the actions, objects and relationships which are relevant the *planning world* or simply *the world* for short. Going on holiday requires a simple form of planning - actions are packing suitcases, going to the travel agent, going to the airport, booking a hotel and so on. Orderings include 'obtain tickets before flying' and 'pack suitcases before going to the airport'. Objects related to the actions are travel tickets, passports, currency, people, planes, baggage and so on, and these objects may have a myriad of important properties and relationships.

Computer programming is another type of planning. Typical actions are programming commands such as assignment, procedure call and iteration; in most programming languages, commands are applied (or *executed*) sequentially, so here we have a *total ordering* of actions in time. For example, the code fragment:

$z{:}=0;\ \mathit{while}\ (z+1)^2 \leq x\ \mathit{do}\ z{:}=z+1$

means apply action $z{:}=0$ *before* the iterative action $\mathit{while}\ (z+1)^2 \leq x\ \mathit{do}\ z{:}=z+1$. Objects are modelled by data types, and relations between objects are the relational operators of the data types used. Goals may be posed by stating conditions on output data. Here the goals may be given by a program specification, and in VDM these would be the post-conditions of operators. The goal of our program example is:

$(z^2 \leq x) \wedge (x < (z+1)^2)$

which is the post-condition of a function to return the integer square root. The form of planning we will model here is called *generative* because the planner proceeds to work out a complete plan to achieve some given goals, assuming it has a fixed, correct representation of the planning world. Where the interaction of a plan execution mechanism with a largely unknown environment is the most important factor, a different kind of planning, call *reactive* planning, may be called for. Our model will be restricted by a number of other simplifying assumptions, to make the case study small enough to fit into a paper. As a working example we use a world often referred to in the planning literature as *the blocks world* (see figure 6.1(a)). Here a robot is given a goal in the form of an arrangement of stacked blocks, and has to work out a plan to achieve that goal. The plan must consist of actions to be applied by a robot arm (a *gripper*).

2.1 Objects in the Blocks World

Objects in the blocks world are the blocks, the table and the gripper. Typical goals involve block stacking, using the gripper. We express properties and relationships in planning worlds in the form of *literals*, such as 'block a is on b' and 'block d has a clear top'. The imaginary world given in figure 6.1(a) could be represented by asserting the following literals, which we will refer to as state $S1$:

'block a is on block b', 'block c is on the table', 'block b is on the table',
'block d is on the table', 'block d has a clear top', 'block a has a clear top',
'block c has a clear top', and 'the gripper is free'.

[1]Refer to [Rich and Knight 91] for a good introduction to Planning.

```
                               |
                               |
     --------              ---------
     |block |              |gripper|
     |   a  |
     --------
     --------   --------   --------
     |block |   |block |   |block |
     |   b  |   |  c   |   |  d   |
     --------   --------   --------
==================================================================table
```

Figure 6.1: The blocks world (state S1)

A set of literals that is assumed to capture a snapshot of the world adequately is called a *state*. To keep the representation of a state simple, we will insist that each literal making up a state asserts a single, positive fact about the world. Negative literals such as:

'block b has not got a clear top'

can be represented implicitly: we assume that whatever is *not* asserted is false: hence, if it is not asserted that 'block b has a clear top', then we assume it is not the case.

Certain facts such as:

'the table always has space for a block'

can be also be represented implicitly, within the actions that model block stacking. The assumed infinite size of the table, for example, can be represented by assuming that a block can always be put down onto it. Note also that the choice of which literals to use depends totally on the user and the sort of tasks that the user has in mind for the planner. For instance we have chosen not to record any shape information about the blocks, or the fact that they are all the same size.

2.2 Actions in the Blocks World

We choose to model four types of action in the blocks world, all performed by the robot gripper: grasping a block, picking up a block, putting a block down onto another, and putting a block down onto the table. Our planner will embody the assumption that the effect of actions changing states can be stated by defining actions via pre-conditions and post-conditions, in a similar form to a VDM operation. An action which models the gripper getting hold of block a is defined as follows:

pre-conditions - literals that must be true before the action can be applied:
'block a has a clear top', 'the gripper is free'
post-conditions -
literals made true by the effect of the action:
'gripper grasps block a'
literals made false by the effect of the action:
'block a has a clear top', 'the gripper is free'

The pre-conditions are assumed to be those facts needed to be true in a state for the action to be applicable. So before this action can be exected on a state, the literals 'block a has a clear top' and 'the gripper is free' must be asserted in the state description. The post-condition of an action is traditionally split into two separate structures: the *add-set*, holding those literals made true by the effect of the action, and the *delete-set*, holding only those literals made false by the effect of the action. Complete with a name, *grasp a*, a shortened form of this action is then (we assume a, b, c and d denote blocks from now on):

name: *grasp a*
pre-conditions: 'a has a clear top', 'gripper is free' ;
add-set: 'gripper grasps a' ;
delete-set: 'a has a clear top', 'gripper is free' ;

In the same way we can model lifting up block a from another block, lifting up block a from the table, putting a down onto another block, and putting a down onto the table:

name: *liftup a from b* ;
pre-conditions: 'gripper grasps a', 'a is on b' ;
add-set: 'a lifted up', 'b has a clear top' ;
delete-set: 'a is on b';

name: *liftup a from table* ;
pre-conditions: 'gripper grasps a', 'a is on table' ;
add-set: 'a lifted up' ;
delete-set: 'a is on table';

name: *putdown a onto c*
pre-conditions: 'a lifted up', 'c has a clear top' ;
add-set: 'a is on c ', 'gripper is free', 'a has a clear top' ;
delete-set: 'a lifted up', 'gripper grasps a', 'c has a clear top' ;

name: *putdown a onto table*;
pre-conditions: 'a lifted up' ;
add-set: 'a is on table ', 'gripper is free', 'a has a clear top ;
delete-set: 'a lifted up', 'gripper grasps a' ;

The actions moving the other blocks b, c, d can all be written in exactly the same form, giving a total of 36 action instances (4 *grasp*'s, 4 *liftup*'s from the table, 4 *putdown*'s onto the table, 12 *putdown*'s between blocks and 12 *liftup*'s between blocks). In fact, we could reduce the action set to just 5 actions if we used parameters for the block names a, b, c, d (see Exercise 2 no. 2) but this would over-complicate the VDM specification we are about to develop.

2.3 Action Application

Next, we define how actions change states. An action can be *applied* to a state if its pre-conditions are literals contained in that state. For example, *grasp a*'s pre-conditions are contained in state A, so *grasp a* can be applied to it. The effect of applying an action to a state is that any literals in the action's *delete-set* are deleted from the old state, and all the literals in the action's *add-set* are 'unioned' to the result, generating a new state. In summary, applying action A to state S, denoted $apply(A, S)$, is given by:

$apply(A, S) = (S \setminus A$'s delete-set$) \cup A$'s add-set

Important note: In the rest of the paper the name of an action will often be identified with the full action representation that it stands for. This is a shorthand device, because when we refer to actions, we do not want to keep repeating their full representation, including their pre- and post-conditions. For example, when we write "apply action *grasp a*", we actually mean apply the action *named grasp a*.

Examples 1

1. Applying *grasp a* to state $S1$ results in a new state, called state $S2$, as follows (see figure 6.1(b)):

```
        |
        |
    ----|----
    |gripper |
    |--------|
     |block |
     |   a  |
     --------
     --------    --------    --------
     |block |    |block |    |block |
     |   b  |    |  c   |    |  d   |
     --------    --------    --------
===================================================================table
```

Figure 6.1(b): The blocks world (state S2)

```
                   |
                   |
                   |
              -----|----
              | gripper |
              |-------- |
               |block |
               |   a  |
               --------


     --------    --------    --------
     |block |    |block |    |block |
     |   b  |    |  c   |    |  d   |
     --------    --------    --------
===================================================================table
```

Figure 6.1(c): The blocks world (state S3)

state $S2$
= apply *grasp a* to state $S1$,
= (state $S1\backslash$ delete-set of *grasp a*) $\cup$ add-set of *grasp a*
= (state $S1\backslash$ { 'a has a clear top', 'gripper is free' }) $\cup${ 'gripper grasps a' },
= { 'a is on b', 'c is on the table', 'b is on the table',
'd is on the table', 'd has a clear top', 'gripper grasps a' 'c has a clear top'},

Note that *grasp a* is applicable to state $S1$ because its pre-conditions ('a has a clear top', 'gripper is free') are contained in $S1$.

2. The pre-conditions of *liftup a from b* ('gripper grasps a', 'a is on b') are contained in state $S2$ so we may apply this action to it. Calling the new state $S3$ (see figure 6.1(c)), we have:

state $S3$
= apply *liftup a from b* to state $S2$
= (state $S2\backslash$ delete-set of *liftup a from b*) $\cup$ add-set of *liftup a from b*
= (state $S2\backslash$ { 'a is on b' }) $\cup${ 'a lifted up', 'b has a clear top' },
= { 'a lifted up', 'b has a clear top', 'c is on the table', 'b is on the table',
'd is on the table', 'd has a clear top', 'gripper grasps a' 'c has a clear top'}.

Exercise 1

Apply the action *putdown a onto c* to state $S3$ to obtain a new state, $S4$.

2.4 Plans

A solution plan to a planning problem is an ordering of actions which achieves a set of goals, and the ordering may be total or partial. A plan is executed by applying each of its individual actions in turn. Planning problems can be posed by describing:

- an initial state: this is the state from which the solution must start execution,
- a set of goal literals: these are the literals which must be *achieved*. A goal literal is said to be achieved by an action sequence if it is contained in the final state after the actions have all been applied (a more general definition of goal achievement is given in section 4).

To be able to solve a planning problem, a planner must have access to a set of actions. A subset of these actions will be used to form the solution. A solution to a planning problem is simply a correct plan, defined as follows:

A correct plan is a complete plan which when applied sequentially to the initial state produces a state which contains all the goal literals.

A complete plan is a total order of actions which can be applied sequentially to an initial state to produce a final state.

Note that the correctness of a plan can only be checked if we have a set of goals in mind. These definitions generalise easily to partially ordered plans, since a partially ordered plan can be thought of as specifying a set of totally ordered plans. Hence we have:

A partially ordered plan is complete (correct) if all the totally ordered plans it specifies are complete (correct).

This model of planning sidesteps many considerations such as the use of resources and the passage of time intervals. For example, in exercise 6.2 no. 1 we model the Painting World. With our restricted model, it is impossible to consider such questions as "have we enough paint to cover the wall?" and "has the first coat of paint dried?". It would be interesting to extend the model to cope with this kind of reasoning, but to keep the case study to a reasonable size we have to limit the planner's application.

Examples 2

1. Consider 'going on a foreign holiday' as a planning scenario. A plan with the goal 'Holiday in Spain' which got us to an airport without bringing our passport would be an incomplete plan. The pre-condition of one of the actions, going through passport control, would not be met. Similarly, a plan which was complete, but landed us in Bermuda rather than Torremolinos would be an incorrect (although perhaps more desirable) plan.

2. Consider the sequence of actions:

grasp a, liftup a from b, putdown a on c

Examples 1 and Exercise 1 show that each of these three actions can be applied in sequence starting from state $S1$, hence it is a complete plan. If the goal was { 'a is on c', 'b has a clear top' }, then the plan is correct with respect to this goal. This sequence is *not* correct with respect to goals { 'c is on d', 'b has a clear top' }, because it does not achieve one of the literals in the goal set.

3. The sequence:

grasp a, putdown a onto c, liftup b from table

is not a complete plan starting at state $S1$, because *putdown a onto c* cannot be applied after *grasp a*, according to our definition of action application. After *grasp a* has been applied, the pre-condition 'a lifted up' of *putdown a onto c* is not in the resulting state $S2$.

Exercises 2

1. We will capture a simple Painting World in our planning model. As a start, we model the operator *paint_ceiling* as follows:

name: *paint_ceiling*
pre-conditions: 'have ladder', 'ladder functional', 'have paint'
add-set: 'ceiling painted'
delete-set: empty

Within our simplification of reality *paint_ceiling* does not delete any facts, and we state this as 'empty'. The other four actions are:

name: *paint_wall*
pre-conditions: 'have paint', 'celing painted'
add-set: 'wall painted'
delete-set: empty

name: *paint_ladder*
pre-conditions: 'have ladder', 'have paint'
add-set: 'ladder painted'
delete-set: 'ladder functional'

name: *get_paint*
pre-conditions: 'have credit card'
add-set: 'have paint'
delete-set: empty

name: *get_ladder*
pre-conditions: 'have credit card', 'own large car'

```
 ----------                                  -----------
|get_ladder|--\                      -------> |paint_wall|
|          |   \                              |          |
-----------     \                    /        -----------
                 \                  /
                  \                /
                   \              /
  ---------         --> --------------          ------------
 |get_paint|---------> | paint_ceiling| --------->|paint_ladder|
 |         |           |              |          |            |
  ---------             --------------            ------------
```

figure 6.1(d): A Partial Ordering of Actions in the Painting World

add-set: ‘have ladder’, ‘ladder functional’
delete-set: empty

From initial state:

{ ‘have credit card’, ‘own large car’ }

we show that the plan illustrated in figure 6.1(d) is correct, with respect to the goal set:

{‘ladder painted’, ‘ceiling painted’, ‘ wall painted’ }.

Note: To show correctness of the plan, you should show that every totally ordered plan conforming to the partial order, of which there are four, is correct.

2. Generalise the blocks world action definitions so that you may use parameters, and express actions such as *putdown X onto Y* and *liftup Z from the table*. Are there any special problems arising when parameters are introduced? (for example, consider the case where $X = Y$ in the definition of *putdown X onto Y*)

3. An Abstract Specification of the Planner

The overall requirement of our planning program is to input a problem posed correctly in its input language, and output a correct plan. How are we to specify such a program? We will choose two levels on which to pose the specification. The first specification, given in section 3.2, is more abstract, implicit and a good deal shorter than the more concrete specification developed in section 4. It relies on a formalisation of the input and captures the idea that the output must be an ordering of actions which is correct with respect to the input goal set. The second level of specification incorporates a *goal directed solution method*, and is concrete enough for us to prototype later in the paper. We start, however, by creating a model of the input language to the planning program, which will be used for both levels. This input will contain the actions, the initial state, and the goal.

3.1 A VDM Representation of Planning Problems

All components of our simplified planning system have as their basis the literal, so we shall start by modelling it. Order is important in a literal (for example ‘block a is on block b’ is not the same as ‘block b is on block a'), and it can be of variable length, hence we will use a sequence of tokens or identifiers to represent it.

$$Literal = Token^*$$

Furthermore, if the literal contains a relation name or property name, we let it be the head element in the sequence, and the objects related by it, the tail. For example, in state $S1$, '*on*' is a relation name and '*clear*' a property name.

Both a state and a goal can be modelled as sets of Literals. In both cases ordering of Literals is not assumed to be important, and there is no limit to the size of goals and states. The Set type is therefore chosen[2]:

$$State = Literal\text{-}\mathbf{set}$$

$$Goal = Literal\text{-}\mathbf{set}$$

Actions have a fixed number of different components (a name, a pre-condition and so on) which leads us to choose a model using a composite:

$$\begin{aligned} Action :: \; & name : Literal \\ & pre \quad : Literal\text{-}\mathbf{set} \\ & add \quad : Literal\text{-}\mathbf{set} \\ & del \quad : Literal\text{-}\mathbf{set} \end{aligned}$$

Finally, we put the three components together in a composite type

$$\begin{aligned} Planning\_Problem :: \; & AS : Action\text{-}\mathbf{set} \\ & I \quad : State \\ & G \quad : Goal \end{aligned}$$

Here the component variables represent:

- the set of actions
- the initial state
- the goal expression

Examples 3

1. State $S1$ is explicitly represented in VDM as:

{ [on, a, b], [on, c, table], [on, b, table], [on, d, table],
[clear, d], [clear, a], [clear, c], [free, gripper] }

2. The goal set in Exercise 2 no. 1 is represented in VDM as:

{ [painted, ladder], [painted, ceiling], [painted, wall] }.

3. Action *grasp a* is represented in VDM by the expression:

$$\begin{aligned} & mk\text{-}Action([grasp, a], \\ & \{[clear, a], [free, gripper]\}, \\ & \{[grasp, a]\}, \\ & \{[clear, a], [free, gripper]\}) \end{aligned}$$

[2] Note that our interpretation of *Goal* and *State* are different. A state is interpreted with the implicit assumption that anything not asserted in it is assumed to be false. A goal, on the other hand, specifies a set of states - exactly all those which contain all the goal's literals.

4. In the Painting World, as in the Blocks World, when we translate the literals into VDM, we use the convention that the properties of objects head the literal sequence. Action *paint ceiling* is represented by

$$
\begin{array}{l}
\textit{mk-Action}([\textit{paint}, \textit{ceiling}],\\
\{[\textit{have}, \textit{ladder}], [\textit{functional}, \textit{ladder}], [\textit{have}, \textit{paint}]\},\\
\{[\textit{painted}, \textit{ceiling}]\},\\
\{\})
\end{array}
$$

Exercises 3

Using the explicit VDM representations of sequences, sets and composites, represent the following:

1. states $S2$ and $S3$;

2. all the Blocks World actions;

3. the planning problem consisting of the actions in 2., the initial state $S1$ and the goal expression:

{ [on, a, c], [clear, a] }

3.2 Invariants for the Planning Problem

Many invariants can be captured to ensure the validity of the planning problem. We will state several here, in terms of problem components I, G and AS. The reader is invited to develop the model further in exercise 6.2.

(1) 'every literal in the goal set appears in either the initial state or in some action's add-set'. This is required because the only way that literals can be added to the initial state in our model is through the application of actions. If it were not satisfied, then the goal set would not be achievable. The condition formalises to:

$\forall\, l \in G \cdot (l \in I \vee \exists\, A \in AS \cdot l \in A.add)$

(2) An invariant that invalidates trivial problems is: 'the goal set is not a subset of the initial state', captured by:

$\neg\,(G \subseteq I)$

(3) Actions need to be restricted to disallow futile ones: 'No action can both add and delete the same literal'. This formalises to:

$\forall\, A \in AS \cdot \neg\,(\exists\, p \cdot p \in A.add \wedge p \in A.del)$

Putting these together we obtain the data type:

$$
\begin{array}{ll}
\textit{Planning\_Problem} :: & AS : \textit{Action}\textbf{-set}\\
& I \;\; : \textit{State}\\
& G \;\; : \textit{Goal}
\end{array}
$$

$$
\begin{array}{l}
\textbf{inv } \textit{mk-Planning\_Problem}(G, I, AS) \triangleq\\
\quad \forall\, l \in G \cdot (l \in I \vee \exists\, A \in AS \cdot l \in A.add) \wedge\\
\quad \neg\,(G \subseteq I) \wedge\\
\quad \forall\, A \in AS \cdot \neg\,(\exists\, p \cdot p \in A.add \wedge p \in A.del)
\end{array}
$$

Exercises 4

1. Check that your answer to Exercise 3, no. 3, satisfies the data type invariant.

2. Formalise the condition: 'every action has at least one literal in its add-set'.

3. Formalise the condition: 'every pre-condition literal is either in the initial state or in some action's add-set'.

The conditions in questions 2. and 3. may well be worthy of inclusion in the invariant. If 2. were not satisfied by an action then one might ask why that action were included (because it could not help in achieving a goal). If 3. were not satisfied by an action, that action would not be able to be used in a plan, as its pre-conditions could never be achieved in (or added to) a state. On the other hand we may not want to be too strict, because we may pose different problems by changing the initial state and the goal, while keeping the action set fixed.

3.3 A First Specification of the Planner

In this section we build up an abstract specification using functions as building blocks. The planner itself is *implicitly* defined via an operation which inputs a *Planning Problem* and outputs a solution in the form of an ordering of actions. No notion of VDM state is required, because of the very abstract nature of the specification.

We start by formalising the application of actions in VDM, with the following function which applies an action to a state:

$$apply : Action \times State \rightarrow State$$

The definition of *apply* immediately follows from the definition given in section 2.3. To make the function total, however, we introduce the idea of an 'error' state, and regard this as the empty set of literals. Applying an action to an error state should also result in an error state:

$$apply : Action \times State \rightarrow State$$

$$\begin{array}{l} apply\ (a, s) \ \underline{\triangle} \\ \quad \mathbf{if}\ a.pre \subseteq s \\ \quad \mathbf{then}\ (s \setminus a.del) \cup a.add \\ \quad \mathbf{else}\ \{\,\} \end{array}$$

A planning problem is solved by the application of a *sequence* of action applications, and so we need to formalise the idea of the application of an action sequence to a state. This is done in terms of *apply*: the function *apply_seq* applies the head of an action sequence to obtain an advanced state, and recursively calls itself with the advanced state and with the tail of the action sequence:

$$apply\_seq : Action^{*} \times State \rightarrow State$$

$$\begin{array}{l} apply\_seq\ (as, s) \ \underline{\triangle} \\ \quad \mathbf{if}\ as = [\,] \\ \quad \mathbf{then}\ s \\ \quad \mathbf{else}\ apply\_seq(\mathbf{tl}\ as, apply(\mathbf{hd}\ as, s)) \end{array}$$

Note:

- if the action sequence is empty, the state is returned unchanged.
- if *as* is an incomplete plan, then the definition of *apply* ensures that the function evaluates to the empty set, signifying an error state.

Next the notion of completeness of a plan is formalised, again using *apply* within a recursive function. $complete(as, s)$ returns true if and only if every action in the sequence *as* is applicable, starting with state *s*. The function is boolean valued:

$$complete : Action^{*} \times State \rightarrow \mathbf{B}$$

If the action sequence is empty it is considered complete, otherwise it is complete if

(a) the pre-conditions of the head of the sequence are contained in the current state, *and*

(b) the tail of the sequence is complete when applied to the advanced state obtained by applying the head of the sequence to the current state.

This is summed up by the VDM function:

$$complete : Action^{*} \times State \to \mathbf{B}$$

$$\begin{array}{l} complete\,(as, s) \triangleq \\ \quad \mathbf{if}\ as = [\,] \\ \quad \mathbf{then}\ true \\ \quad \mathbf{else}\ (\mathbf{hd}\ as).pre \subseteq s \wedge complete(\mathbf{tl}\ as, apply(\mathbf{hd}\ as, s)) \end{array}$$

Using the functions *complete* and *apply_seq*, an implicit specification of a planner can be written. Essentially, the specification states that for an input *Planning_Problem*, a correct plan in the shape of a sequence of actions is output:

$$\begin{array}{l} PLANNER\,(pp : Planning\_Problem)\ soln : Action^{*} \\ \mathbf{pre}\ \ true \\ \mathbf{post}\ \mathbf{elems}\ soln \subseteq pp.AS \wedge \\ \qquad complete(soln, pp.I)) \wedge \\ \qquad pp.G \subseteq apply\_seq(soln, pp.I) \end{array}$$

The post-condition asserts that

- only actions defined in the *Planning_Problem* are allowed in the action sequence;
- the plan is complete
- execution of the plan outputs a state which contains the goal

Hence the final two conjunctions formalise the correctness criterion given in section 2. Note that *PLANNER* is not a function as there may be many plans that satisfy this specification, given a particular planning problem. Even if we added an extra constraint to the post condition which insists on a minimum length solution, there still may be more than one correct plan.

Exercises 5

If the first three exercises, let $A1$ be the action called $[grasp, a]$, $A2$ the action called $[liftup, a, b]$, and $A3$ the action called $[putdown, a, c]$.

1. Check that the expression:

$complete([A1, A2, A3], S1)$

evaluates to true.

2. Evaluate the expression:

$apply\_seq([A1, A2, A3], S1)$

using the formal definition of *apply_seq*, verefying that it coincides with our informal notion of section 2.

3. Assume *PLANNER* has been input with the planning problem of Exercise 3, no. 3. Using the results of exercises 1. and 2. above, deduce that

$soln = [A1, A2, A3]$

makes the post-condition of *PLANNER* true.

4. Generalise the specification of *PLANNER* to one which outputs a partially ordered set of actions as a solution. Hint: *apply* must be re-defined to apply a *set* of action sequences and return a *set* of states.

5. Consider the following planning problem, where p and q are literals (this example is due to Yogesh Naik):

$$\begin{array}{l} mk\text{-}Planning\_Problem(\\ \{mk\text{-}Action([bill], \{[p]\}, \{[q]\}, \{[p]\}), mk\text{-}Action([ben], \{[q]\}, \{[p]\}, \{[q]\})\}, \\ \{p\}, \\ \{p, q\}) \end{array}$$

It has two actions, called *bill* and *ben*; its initial state is simply the set of one literal, p, and its goal is the set of two literals, p and q. Verify that this problem satisfies *Planning_Problem*'s invariant. With this problem as input, can you find an action sequence which satisfies *PLANNER*'s post-condition? What conclusions can you draw about the Satisfiability of *PLANNER*? In fact, this exercise shows that we can pose problems in the planning language for which there are no solution - planning is hard!.

4. A Design Level Specification

In this section we construct a more concrete specification for the planner, which incorporates a goal directed procedure for solving planning problems. Most non-trivial problems can be usefully specified at one or more 'design levels', in which commitments to particular solution techniques and data structures are progressively made. After having completed a more detailed design level, it is up to the designer to check it is adequate with respect to the more abstract level. VDM encompasses a well developed process called *reification*, in which operators and data types are re-expressed at a more detailed level after their initial specification, and the detailed level is checked for adequacy using a *retrieve function*. Showing how the design level of the planner described in this section conforms to the abstract level given above is beyond the scope of this book, and is left as a project for the interested reader.

The specification, certainly towards the end, becomes rather complicated and may be difficult for some readers. In this case, the reader is encouraged to move on to the next section, where the prototyping of the planner may shed more light on its specification, or to consult appendix 3, where sample inputs and outputs of the planner are given, as well as its implementation.

4.1 A Technique for Solving Planning Problems

The more concrete specification commits our planner to a particular solution method in which the planning program generates plans in a systematic manner and then terminates when it finds a plan that is correct. The solution method runs as follows:

- start with an initial plan which only contains the initial state and goal set, viewed as special actions;
- incrementally achieve goal literals by:
  - identifying an action already in the plan which achieves the goal literal; or
  - adding a new action to the plan to achieve the goal literal (in which case the new action's pre-conditions themselves must be achieved).

When an action is added to the plan, rather than storing it in a sequence, it is stored within a partially ordered set of actions, such as the action set in the Painting World example.

```
START: Initialise the Planning Problem
by generating the initial plan,
and put the initial plan in a Store;


LOOP:

  1. --Choose and remove a plan 'pp' from the Store;

  2. --Choose a goal instance Gi from 'pp';

  3. --Generate plans to achieve Gi in 'pp' in all ways possible;

  4. --Add all new plans generated by step 3 to the Store

UNTIL there exists a plan in Store which has no unachieved goal instances.
```

figure 6.2: The Top Level Loop of a Naive Planning Algorithm

The planner's job is to find a plan in which all literals in the goal set and in every action's pre-condition are achieved, in the sense that the final solution is complete (the pre-conditions of each action are met as they are applied) and correct (the final state produced contains the goal set). This means that an action's pre-conditions must be achieved at an earlier time than the goal set, and we will identify this time with the position of the action in the plan. The combination of a goal literal with a position in the temporal ordering at which it must be achieved we call a *goal instance*.

In figure 6.2 we present a top level algorithm of a planning program. If we assume that the choices in steps 1. and 2. are made randomly, then the heart of this algorithm is step 3. - generating new plans which achieve previously unachieved goal instances. The specification developed here will consist of the initialisation operation carried out before the loop, and two operations which perform goal achievement necessary for step 3. The terminating condition of the loop is dependent on a plan being found in Store which has no unachieved goal instances. In fact, the specification of the goal achievement operations ensures that once a plan is found with an empty set of unsolved goal instances, that plan will contain a solution as defined by our abstract specification in section 3.

4.2 An Introduction to Goal Achievement in Planning

A goal instance is any pre-condition literal and action pairing in a plan. To use a concrete example, consider figure 6.2(b).

We know 'have ladder' is a pre-condition of action *paint ceiling*, hence the pair ('have ladder', *paint ceiling*) is a goal instance in the plan represented by this figure. One of the goal literals of the Painting World is 'ladder painted', so the pair ('ladder painted', *goal*) is another goal instance. In this example, *goal* is considered a special kind of action, whose pre-conditions are the literals in the goal set. The same can be done for the initial state - it can be considered an action which has an add-set containg all the literals in the initial state.

For a more abstract example, refer to figure 6.3. It is a bounded poset representing an abstract plan containing some imaginary actions which we call $C1$, $C2$, $C3$, $C4$, A and O. Assume 'p' in the diagram is a literal contained

```
         ----------                              -----------
        |get_ladder|                        ---> |paint_wall|
       /|          |-->-                   /     |          |->-
      /  -----------     \                 /      -----------    |
    ^                     \                                      |
    |                      \               /                     |
 ------                     \             /                    ------
|init  |                     \           /                    |goal  |
|      |--                     \         /                     |      |
 ------  |                      \       /                      /------
         V                       \     /                     /
         ---------           --------------    ------------  /
        |get_paint|-------> | paint_ceiling| -->|paint_ladder|->
        |         |         |              |    |            |
         ---------           --------------     ------------
```

figure 5.4: The Painting World plan: A Partial Order with Bounds

in the pre-conditions of O; then the pair $(p,\ O)$ is an example of a goal instance.

Now we can give a full definition of **goal achievement**:

A goal instance $(p,\ O)$ is achieved in a plan if some action X is constrained to be necessarily before O, X contains p in its add-set, and no action that could possibly occur between X and O contains p in its delete-set. In this case X is said to be the achiever of p at O

We can define a *complete plan* in terms of goal achievement: a complete plan is a bounded ordering of actions in which every goal instance is achieved by some action (compare this definition with our earlier definition of completeness for sequential plans in section 3). A complete plan in this sense is also correct because a subset of these goal instances are those taken from the goal set and combined with dummy action *goal*. We will finish this section with several examples and exercises, so that the reader may get an intuitive feel for our goal achievement definition.

Examples 4

1. The goal instance ('ladder functional', *paint ceiling*) is achieved in the Painting World plan of figure 5.4, by action *get ladder*. We can check the conditions are true by our definition of goal achievement:

- *get ladder* is necessarily before *paint ceiling*, as there is a path of directed arcs (in this case just one) from the former to the latter;
- *get ladder* contains 'ladder functional' in its add-set (see Exercise 2);
- the only action that can possibly occur between *get ladder* and *paint ceiling* is *get paint*. It does not contain 'ladder functional' is its delete-set, and so this condition is met.

Note that if *paint ladder* was *not* ordered to be after *paint ceiling*, then goal achievement would not be necessarily true, because *paint ladder* contains 'ladder functional' is its delete-set.

2. The goal instance ('have paint', *paint wall*) is achieved by action *get paint*. The conditions are true as follows: *get paint* is necessarily before *paint wall*; *get paint* contains 'have paint' in its add-set (see Exercise 2); none of the three actions that could be between *get paint* and *paint wall* contain 'have paint' in their delete-set.

3. In figure 6.3, action instance A *could* be the achiever of goal instance (p, O) if

- A contains p in its add-set
- $C1$ does not contain p in its delete-set.
- Either $C4$ does not contain p in its delete-set OR an arc is added from $C4$ to A to constrain $C4$ to be before the achiever, A.

4. $C4$ could be the achiever of goal instance (p, O) if all these conditions are made true:

- An arc was added from $C4$ to O, to ensure $C4$ was executed before O in an application of the completed plan;
- $C4$ contains p in its add-set;
- Either $C1$ does not contain p in its delete-set OR an arc is added from $C1$ to $C4$ to constrain $C1$ to be before the achiever, $C4$.
- Either A does not contain p in its delete-set OR an arc is added from A to $C4$ to constrain A to be before the achiever, $C4$.

Actions that could not possibly be used to achieve p are those that do not contain p in their add-sets, those which are necessarily after O (for example $C2$), and those which have an action in between them and the goal instance which deletes p.

Of course, another way to achieve a pre-condition literal p is to add another action to the plan to achieve it. In this case we must go through the same procedure to make sure p is not 'undone'. Note that although any further temporal constraints on the plan (that is additional arcs) will not invalidate the achievement of p, the addition of an action to achieve some other goal instance may well undo its achievement (we return to this point later).

Exercises 6

1. Using the example plan in figure 6.3, state the conditions under which the following operators achieve (p, O):

(a) $C1$

(b) $C3$

(c) *init*

2. Assume another action Y is added to the plan in figure 6.3, and constrained to be before O in the new plan (see figure 6.4). State the conditions under which Y achieves (p,O).

3. (This follows on from question 2.). Assume the conditions of Y being an achiever for p in question 2. are true. Now add extra temporal constraints to figure 6.4, for example an arc from Y to $C1$, and another from $C4$ to O. Is Y still an achiever for p? Form an argument showing that for any X which is an achiever for a literal p at action instance O, then no legal additions of temporal constraints (that is arcs) will affect X's achievement of p.

4. List all the goal instances in the Painting World plan of figure 5.4. Convince yourself, using our definition of goal achievement, that the plan contains an achiever for each one.

```
             ---            ----       p ---
            | A | --->--   | C1 |  -->--| O |->-
             ---            ----         ---    |
              |                                 |
              |                                 |
              |                                 V
              |
 ------     --|--        ----                ----         ------
| init |->-|  C3 |--->--| C4 |------->-------| C2 | -->-- | goal |
 ------     -----        ----                ----         ------
```

figure 6.3: An abstract plan (C1,C2,C3,C4,O,A are arbitrary action
identifiers, init and goal identify the initial state and goal
conditions viewed as special actions)

```
                             ---
          -------->-------- | Y |-->-----
         /                   ---         |
        /                                V
       /                                 |
      /       ---            ----       p ---
     |       | A | --->--   | C1 |  -->--| O |->-
     |        ---            ----         ---   |
     ^        |                                 |
     |        |                                 |
     |        |                                 V
     |        ^                                 |
 ------     --|--        ----                ----         ------
| init |->-|  C3 |--->--| C4 |------->-------| C2 | -->-- | goal |
 ------     -----        ----                ----         ------
```

figure 6.4: A new abstract plan

4.3 Modelling the VDM State

The VDM state will represent a developing plan, as discussed above. The developing plan consists of some actions, a temporal ordering on those actions, some outstanding goal instances to be achieved, and some goal instances already achieved. The 'achieve' operations defined later will change the state by achieving one of the unsolved goal instances either through the addition of an action to the plan, or through an existing action in the plan.

Although it will also contain the planning problem itself, we will call the whole system state a *Partial_Plan*, and give it five state components named *pp*, *Os*, *Ts*, *Ps*, *As* as follows:

state *Partial_Plan* **of**
  $pp : Planning\_Problem$
  $Os : Action\_instances$
  $Ts : Bounded\_Poset$
  $As : Goal\_instances$
  $Ps : Goal\_instances$
end

Actions, states and the structure of *Planning_Problem* were all defined in section 3.

4.3.1 The *Action_instances* Data Structure

This component holds the actions occurring in a plan - called the *action instances*. It is necessary to allocate each action instance a unique identifier as it is added to the developing plan because the same action may occur more than once in the plan. In this case *Os* needs to be represented using a mapping from identifiers to actions, and an invariant is used to constrain the range of *Os* to be members of *pp.AS* (recall from 6.3 that *AS* represents the component of *Planning_Problem* in which the action definitions are held). This map captures the constraint that no identifier can point to more than one action, but allows one action to be pointed to by more than one identifier.

For the sake of uniformity, the initial state and the goal set will be modelled as special actions, occurring in every plan. More importantly, they will become the lower and upper bounds, respectively, of the bounded poset in *Ts* below. These special actions are formed explicitly as:

$$mk\text{-}Action([init], \{\,\}, pp.I, \{\,\})$$
$$mk\text{-}Action([goal], pp.G, \{\,\}, \{\,\})$$

and will be identified by *init* and *goal* respectively. The translation of these two literal sets into the actions above is intuitively sound: the initial state needs no pre-conditions, does not delete any literals but has the effect of 'adding' all its literals; the goal does not have any post-condition effects, but to achieve its pre-conditions a plan must have asserted all the literals in the goal.

The type of *Os* is *Action_instances*, and is defined as follows:

$$Action\_id = Token$$

$$Action\_instances = Action\_id \xrightarrow{m} Action$$

4.3.2 The *Bounded_Poset* Data Structure

Ts holds a strong partial order relation on the action identifiers in *Os*, bounded by *init* and *goal* which identify the special initial state and goal actions. A poset specification developed is used below to provide the temporal structure for the plan (for the reader new to formal methods, this specification is introdced in chapter 5 of "The Construction of Formal Specifications" by Turner and McCluskey, published in 1994 by McGraw-Hill):

$$\begin{aligned} &Arc :: source : Action\_id \\ &\qquad\quad dest \quad : Action\_id \end{aligned}$$

$$Bounded\_Poset = Arc\textbf{-set}$$

$$\begin{aligned} &\textbf{inv } mk\text{-}Bounded\_Poset(p) \triangleq \forall\, x, y \in get\_nodes(p) \cdot \\ &\quad \neg\,(before(x, y, p) \wedge before(y, x, p)) \wedge \\ &\quad x \neq init \;\Rightarrow\; before(init, x, p) \wedge \\ &\quad x \neq goal \;\Rightarrow\; before(x, goal, p) \end{aligned}$$

The associated functions change are:

$$get\_nodes : Arc\textbf{-set} \to Action\_id$$

$$\begin{aligned} &get\_nodes\,(p) \triangleq \\ &\quad \{a.source \mid a \in p.arcs\} \cup \{a.dest \mid a \in p.arcs\} \end{aligned}$$

$$before : Action\_id \times Action\_id \times Arc\textbf{-set} \to \mathbf{B}$$

$$\begin{aligned} &before\,(x, z, p) \triangleq \\ &\quad mk\text{-}Arc(x, z) \in p \vee \\ &\quad \exists\, y \in get\_nodes(p) \cdot before(x, y, p) \wedge before(y, z, p) \end{aligned}$$

$$possibly\_before : Action\_id \times Action\_id \times Arc\textbf{-set} \to \mathbf{B}$$

$$\begin{aligned} &possibly\_before\,(x, y, p) \triangleq \\ &\quad x \neq y \wedge \neg\, before(y, x, p) \end{aligned}$$

Likewise, the three operations *init_poset*, *add_node* and *make_node* are easily adapted to fit this application:

$$init\_poset : \to Bounded\_Poset$$

$$\begin{aligned} &init\_poset\,() \triangleq \\ &\quad \{mk\text{-}before(init, goal)\} \end{aligned}$$

$$add\_node : Action\_id \times Bounded\_Poset \to Bounded\_Poset$$

$$\begin{aligned} &add\_node\,(u, p) \triangleq \\ &\quad p \cup \{mk\text{-}Arc(init, u), mk\text{-}Arc(u, goal)\} \end{aligned}$$

$$make\_before : Action\_id \times Action\_id \times Bounded\_Poset \to Bounded\_Poset$$

$$\begin{aligned} &make\_before\,(u, v, p) \triangleq \\ &\quad \textbf{if } possibly\_before(u, v, p) \wedge \{u, v\} \subseteq get\_nodes(p) \\ &\quad \textbf{then } p \cup \{mk\text{-}Arc(u, v)\} \end{aligned}$$

4.3.3 The *Goal_instances* Data Structure

Ps represents a collection of unsolved goal instances, those which are still to be achieved by some action. A goal instance is a relation between goal literals and action identifiers, therefore we represent the set of *Ps* as a set of ordered pairs as follows:

$$Goal\_instance :: gl : Literal$$
$$\qquad ai : Action\_id$$

$$Goal\_instances = Goal\_instance\text{-}\mathbf{set}$$

Action instances that are added to the plan to *achieve* the goal instances in Ps may themselves have pre-condition literals: these will then be added to Ps as goal instances. At initialisation, Ps will record the goal set $pp.G$, and will take the form:

$\{mk\text{-}Goal\_instance(g1, goal), ..., mk\text{-}Goal\_instance(gn, goal)\}$

where $pp.G = \{g1, ..., gn\}$. This can be written more concisely using set comprehension:

$\{mk\text{-}Goal\_instance(g, goal) \mid g \in pp.G\}$

Likewise, As holds a collection of achieved goal instances. This set will be initially empty, but each execution of the 'achieve' operations defined below will result in a literal being achieved at some point in the plan, and so this goal instance will be added to As.

4.4 The State Invariant

Some possible states of the *Partial_Plan* structure we have defined are clearly not valid, and our discussion has already thrown up some useful invariants. We first start by listing the easier ones:

(1) Os always contains the two special actions formed from the initial state and goal literals.

(2) The range of the map Os (the *Action*-**set**) is a subset of actions posed in the *Planning_Problem* augmented with the *init* and *goal* actions.

(3) The nodes in Ts and the identifiers from Os are the same: this ensures that Ts is a partial order on all (and only) those action instances in Os.

(4) Ps and As are disjoint: no goal instance can be both achieved and not achieved.

(5) All the pre-conditions of the action instances in Os are recorded as goal instances in either As or Ps (which means they have been achieved or are not achieved).

Finally we express the most important invariant of a plan:

(6) Every goal instance $mk\text{-}Goal\_instance(p, O)$ (for a goal literal p and an action instance O) in As is *achieved* using the definition in section 4.2:

- there is an action instance A in the plan which is necessarily before O and contains p in its add-set; AND
- no action in the plan that could possibly occur between A and O contains p in its delete-set.

Notice that O could itself be *goal*, in which case p would be one of the goal literals, or again, A could be the *init* action, in which case p would have to be contained in the initial state. Invariant (6) corresponds to the informal definition of goal achievement described at the beginning of section 4.

The first five conditions are expressed in VDM as follows. Readers are encouraged to try to formalise the conditions themselves before reading on.

(1) $Os(init) = mk\text{-}Action([init], \{\,\}, pp.I, \{\,\}) \wedge\ Os(goal) = mk\text{-}Action([goal], pp.G, \{\,\}, \{\,\})$

(2) $\mathbf{rng}\ Os \subseteq pp.AS \cup \{Os(init), Os(goal)\}$

(3) $\mathbf{dom}\ Os = get\_nodes(Ts)$

(4) $As \cap Ps = \{\,\}$

(5) $\forall\, A \in \mathbf{dom}\ Os \cdot (p \in Os(A).pre \;\Rightarrow\; mk\text{-}Goal\_instance(p, A) \in (Ps \cup As))$

The first part of (6):

"there is an action instance A in the plan which is necessarily before O and contains p in its add-set ... "

formalises to

$$\begin{array}{l} \exists\, A \in \mathbf{dom}\ Os \cdot \\ before(A, O, Ts) \wedge \\ p \in Os(A).add \end{array}$$

To formalise the second part, we make use of the partial order's *possibly_before* function. The expression:

$$possibly\_before(A, C, Ts) \wedge possibly\_before(C, O, Ts)$$

means that A could be ordered to be before C, and C could be ordered to be before O, with respect to the partial order Ts. This captures the idea that C could be ordered to be *between* A and O in partial order Ts. In figure 6.3, for example, this expression is true for $C = C4$, and for A and O as they actually appear in the figure. Hence the second part:

"no action in the plan that could possibly occur between A and O contains p in its delete-set.

formalises to:

$$\begin{array}{l} \neg\,(\exists\, C \in \mathbf{dom}\ Os \cdot \\ possibly\_before(C, O, Ts) \wedge \\ possibly\_before(A, C, Ts) \wedge \\ p \in Os(C).del) \end{array}$$

We put these conditions together into a function definition, which defines what it means for A to be an achiever of p at O in Ts:

$$achieve : Action\_instances \times Bounded\_Poset \times Action\_id \times Goal\_instance \to \mathbf{B}$$

$$\begin{array}{l} achieve\,(Os, Ts, A, mk\text{-}Goal\_instance(p, O)) \;\underline{\triangle} \\ \quad before(A, O, Ts) \wedge \\ \quad p \in Os(A).add \wedge \\ \quad \neg\,(\exists\, C \in \mathbf{dom}\ Os \cdot \\ \quad possibly\_before(C, O, Ts) \wedge \\ \quad possibly\_before(A, C, Ts) \wedge \\ \quad p \in Os(C).del) \end{array}$$

and finally we can write condition (6) as:

$\forall\, gi \in As \cdot \exists\, A \in \mathbf{dom}\ Os \cdot achieve(Os, Ts, A, gi)$

Hence the final VDM state definition is:

$$\begin{array}{l} \mathbf{state}\ Partial\_Plan\ \mathbf{of} \\ \quad pp : Planning\_Problem \\ \quad Os : Action\_instances \\ \quad Ts : Bounded\_Poset \\ \quad As : Goal\_instances \\ \quad Ps : Goal\_instances \end{array}$$

$$
\begin{array}{l}
\textbf{inv } \textit{mk-Partial\_Plan}(pp, Os, Ts, As, Ps) \triangleq \\
\quad Os(init) = \textit{mk-Action}([init], \{\,\}, pp.I, \{\,\}) \land \\
\quad Os(goal) = \textit{mk-Action}([goal], pp.G, \{\,\}, \{\,\})\}) \land \\
\quad \textbf{rng } Os \subseteq pp.AS \cup \{Os(init), Os(goal)\} \land \\
\quad \textbf{dom } Os = get\_nodes(Ts) \land \\
\quad As \cap Ps = \{\,\} \land \\
\quad \forall A \in \textbf{dom } Os \cdot (p \in Os(A).pre \;\Rightarrow\; \textit{mk-Goal\_instance}(p, A) \in (Ps \cup As)) \land \\
\quad \forall gi \in As \cdot \exists A \in \textbf{dom } Os \cdot achieve(Os, Ts, A, gi) \\
\textbf{end}
\end{array}
$$

Exercise 7

1. Check that the Painting World examples in Examples 6, no. 1 and no. 2, satisfy the formal definition of goal achievement.

2. The definition of *achieve* may be logically transformed to make prototyping more straightforward later in the paper. Using the definitions of *possibly_before* and *before* already supplied, show that:

$$
\begin{array}{l}
\neg\, (\exists\, C \in \textbf{dom } Os \cdot \\
possibly\_before(C, O, Ts) \land \\
possibly\_before(A, C, Ts) \land \\
p \in Os(C).del)
\end{array}
$$

transforms to:

$$
\begin{array}{l}
\forall\, C \in \textbf{dom } Os \cdot \\
C = O \lor \\
C = A \lor \\
before(O, C, Ts) \lor \\
before(C, A, Ts) \lor \\
\neg\, (p \in Os(C).del)
\end{array}
$$

4.5 VDM Operations

4.5.1 Operation *INIT*

As usual we will construct the initialisation operation first. This inputs a *Planning_Problem* and outputs the first plan. The only action instances in Os are *init* and *goal*, there are no achieved goal instances, and the only goal instances to be achieved are those from the goal set itself, pp.G.

$$
\begin{array}{l}
INIT\ (ppi : Planning\_Problem) \\
\textbf{ext wr } \ pp : Planning\_Problem \\
\quad\;\; \textbf{wr } \ Os : Action\_instances \\
\quad\;\; \textbf{wr } \ Ts : Partial\_order \\
\quad\;\; \textbf{wr } \ Ps : Goal\_instances \\
\quad\;\; \textbf{wr } \ As : Goal\_instances \\
\textbf{pre } \ true \\
\textbf{post } \ pp = ppi \land \\
\quad Os = \{init \mapsto \textit{mk-Action}([init], \{\,\}, ppi.I, \{\,\}), goal \mapsto \textit{mk-Action}([goal], ppi.G, \{\,\}, \{\,\})\} \land \\
\quad Ts = init\_poset() \land \\
\quad Ps = \{\textit{mk-Goal\_instance}(g, goal) \mid g \in ppi.G\} \land \\
\quad As = \{\,\}
\end{array}
$$

4.5.2 Operation $ACHIEVE\_1$

The first operation we shall specify to achieve a goal instance uses an action instance already present in Os to be the achiever. Labelled '*ext rd*' below, Os is for access only and does not change. The first post-condition predicate we need is therefore:

$\exists A \in \mathbf{dom}\ Os \cdot achieve(Os, Ts, A, gi)$

It is not just a question of verefying that the *achieve* function is true for this instance, however; it may be necessary to change the partial order Ts to make sure that the *achieve* predicate is true. The variable Ts is therefore labelled '*ext wr*' below.

We leave the specification of

$is\_completion\_of(Ts, \overleftarrow{Ts}),$

which defines a relation between two posets, as an exercise. The relation is true if the two posets contain the same set of nodes, and any nodes that are ordered in $\overleftarrow{Ts}$ are also ordered in Ts. This relation will be used to provide the constraint on the change to the input partial order that we require: Ts must be a completion of $\overleftarrow{Ts}$.

Finally, the goal instance being achieved is essentially passed from Ps to As:

$$Ps = \overleftarrow{Ps} \setminus \{gi\} \wedge$$
$$As = \overleftarrow{As} \cup \{gi\}$$

Putting these predicates together, we get:

$$\begin{array}{l}
ACHIEVE_1\,(gi : Goal_instance) \\
\mathbf{ext\ rd}\quad Os : Action_instances \\
\qquad\ \ \mathbf{wr}\quad Ts : Partial_order \\
\qquad\ \ \mathbf{wr}\quad Ps : Goal_instances \\
\qquad\ \ \mathbf{wr}\quad As : Goal_instances \\
\mathbf{pre}\ \ gi \in Ps \\
\mathbf{post}\ \exists A \in \mathbf{dom}\ Os \cdot achieve(Os, Ts, A, gi) \wedge \\
\qquad is_completion_of(Ts, \overleftarrow{Ts}) \wedge \\
\qquad Ps = \overleftarrow{Ps} \setminus \{gi\} \wedge \\
\qquad As = \overleftarrow{As} \cup \{gi\}
\end{array}$$

4.5.3 Operation $ACHIEVE\_2$

The second achieve operator achieves a goal instance by the introduction of a new action instance into Os (recall exercise 6.4 no. 2). This is necessary if no action already in the plan can be found to achieve the goal.

The new action will need a new identifier, and to create a *unique* one for the action instance we assume the existence of a function which inputs a set of $Action\_id$ and outputs one not in the set:

$$\begin{array}{l}
newid\,(is : Action_id\text{-}\mathbf{set})\ i : Action_id \\
\mathbf{pre}\ \ true \\
\mathbf{post}\ \ i \notin is
\end{array}$$

The introduction of a new action instance can be written using the 'let' construct as follows:

$$
\begin{aligned}
&\textbf{let } NewA = newid(\textbf{dom } \overleftarrow{Os}) \textbf{ in}\\
&\exists\, A \in pp.AS \cdot Os = \overleftarrow{Os} \dagger \{NewA \mapsto A\}
\end{aligned}
$$

and in fact we will use the 'let' construct to bind NewA to the new identifier throughout the whole post-condition.

Now that there is a new action in the developing plan, we have the added problem that certain goal instances in As may be rendered un-achieved (the technical term in planning for the plight of such unfortunate goal instances is that they have been *clobbered*). Consider figure 6.4 of exercise 2 in 6.4.2. Assume that a goal instance $(q, C1)$ had been achieved by action A, and that q was a member of Y's delete-set. Then the addition of Y to achieve goal instance (p, O) would clobber q, as in this case the the 'achieve' predicate:

$$achieve(Os,\, Ts,\, A,\, mk\text{-}Goal\_instance(q,\, C1))$$

would evaluate to false, because the following expression is true:

$$
\begin{aligned}
&possibly\_before(Y,\, C1,\, Ts) \wedge\\
&possibly\_before(A,\, Y,\, Ts) \wedge\\
&q \in Os(Y).del
\end{aligned}
$$

The preceding argument necessitates the 'declobber' condition in the post-condition of $ACHIEVE\_2$ (we will define it fully later).

The relationship between the old and new temporal orders ($\overleftarrow{Ts}$ and Ts) is a little more subtle in this second achieve operation. Here, Ts must be a completion of $add\_node(NewA, \overleftarrow{Ts})$, which is the ordering with the new node added. The constraint we need on the output temporal order is therefore:

$$is\_completion\_of(Ts,\, add\_node(NewA,\, \overleftarrow{Ts}))$$

Finally, $\overleftarrow{Ps}$ is augmented with the pre-conditions of the new action, and the achieved goal instance gi is passed to As:

$$
\begin{aligned}
&Ps = (\overleftarrow{Ps} \setminus \{gi\}) \cup \{mk\text{-}Goal\_instance(p,\, NewA) \mid p \in A.pre\} \wedge\\
&As = \overleftarrow{As} \cup \{gi\}
\end{aligned}
$$

Putting these conditions together, gives us the following operation:

$$
\begin{aligned}
&ACHIEVE\_2\,(gi : Goal\_instance)\\
&\textbf{ext rd}\;\; pp : Planning\_Problem\\
&\quad\;\;\textbf{wr}\;\; Os : Action\_instances\\
&\quad\;\;\textbf{wr}\;\; Ts : Partial\_order\\
&\quad\;\;\textbf{wr}\;\; Ps : Goal\_instances\\
&\quad\;\;\textbf{wr}\;\; As : Goal\_instances\\
&\textbf{pre}\;\; gi \in Ps\\
&\textbf{post let } NewA = newid(\textbf{dom } \overleftarrow{Os}) \textbf{ in}\\
&\qquad \exists\, A \in pp.AS \cdot Os = \overleftarrow{Os} \dagger \{NewA \mapsto A\}\\
&\qquad achieve(Os,\, Ts,\, NewA,\, gi) \wedge\\
&\qquad \forall\, gj \in \overleftarrow{As} \cdot declobber(Os,\, Ts,\, NewA,\, gj) \wedge\\
&\qquad is\_completion\_of(Ts,\, add\_node(NewA,\, \overleftarrow{Ts})) \wedge\\
&\qquad Ps = (\overleftarrow{Ps} \setminus \{gi\}) \cup \{mk\text{-}Goal\_instance(p,\, NewA) \mid p \in A.pre\} \wedge\\
&\qquad As = \overleftarrow{As} \cup \{gi\}
\end{aligned}
$$

The *declobber* condition effectively insists that each gi in As remains achieved after the addition of $\{NewA \mapsto A\}$ to Os. If we let $gi = mk\text{-}Goal\_instance(q, C)$, then this is so if one of the following conditions is met:

- C is necessarily before $NewA$ in Ts:

  $before(C, NewA, Ts)$

- $NewA$ does not contain q in its delete-set:

  $\neg\,(q \in Os(NewA).del)$

- there is an achiever for gi called W which is constrained to be between $NewA$ and C:

  $\exists\, W \in Os \cdot$
  $(before(NewA, W, Ts) \wedge$
  $before(W, C, Ts) \wedge$
  $q \in Os(W).add)$

Let us return to the example in figure 6.4 of exercise 2 in section 4.2. We had assumed above that a goal instance $(q, C1)$ had been achieved by action A, and that q was a member of Y's delete-set. After the addition of Y to the plan (to achieve goal instance (p, O)), the goal instance $(q, C1)$ therefore had been clobbered by Y. Consideration of the first condition above leads us to one way of declobbering: the temporal order can have an arc added from $C1$ to Y, to make sure that the clobbering action would be applied *after* $C1$.

Putting the disjunction of conditions together, the definition of function declobber is formed:

$declobber : Action\_instances \times Bounded\_Poset \times Action\_id \times Goal\_instance \to \mathbf{B}$

$declobber\ (Os, Ts, NewA, mk\text{-}Goal\_instance(q, C)) \;\underline{\triangle}$
$\quad before(C, NewA, Ts) \vee$
$\quad \neg\,(q \in Os(NewA).del) \vee$
$\quad \exists\, W \in Os \cdot$
$\quad (before(NewA, W, Ts) \wedge$
$\quad before(W, C, Ts) \wedge$
$\quad q \in Os(W).add)$

Exercise 8

Consider our running example using figure 4 of exercise 2 in section 4.2. Under what other conditions (apart from the one we have given above) could goal instance $(q, C1)$ be declobbered?

4.5.4 Proof Obligations

We discharge the proof obligation for $ACHIEVE\_1$, while leaving the proof obligations for $INIT$ and $ACHIEVE\_2$ as an exercise.

Firstly, we make the observation that $ACHIEVE\_1$ as it stands is *not* satisfiable! In other words, there is there is a binding of inputs that will result in no output state. This is demonstrated by considering the VDM state output from $INIT$ and input to $ACHIEVE\_1$: the only 'action' in the plan before *goal* is *init*, and if gi cannot be achieved by *init*, then $ACHIEVE\_1$ will fail. Exercise 5 no.5 asks you to find a pre-condition that renders this operation satisfiable.

We can show, however, that if $ACHIEVE\_1$ outputs a state (and it is non-deterministic in that there may be many states satisfying the post-condition) then the state is valid with respect to the invariant. In the informal proof below, we refer to the invariant's components (1) through to (6):

- (1) and (2) remain true because Os is unchanged.
- the truth of (3) is preserved over the state change because the nodes in Ts remain constant, and Os is unchanged.
- the final two conditions in the post-condition take a gi from Ps and put it in As. These two sets stay disjoint, and so (4) is true.
- The specification preserves the sets $Ps \cup As$ and Os, hence (5) is preserved.

To finish, we must show (6) is true in the output state. $ACHIEVE\_1$'s post-condition asserts that the new goal instance is achieved, and in the old state we have (asserted by the invariant) that all the other gi's were achieved. It remains, therefore, to show that these gi's are still achieved in the new state. By exercise 6.4 no 3, the addition of extra temporal constraints into Ts does not invalidate any achieved goal instances, and so, also using the fact that Os is unchanged, we argue that the final part of the invariant remains intact.

Additional Problems 6

Some of the Problems below are quite hard. The reader who wants to get a better feel for the specification is encouraged to precede to the next chapter, where it is prototyped.

Problem 1. Specify a function which inputs a *Partial_Plan* and returns true if the problem it contains has been solved (hint: the problem is solved if all the goal instances in the plan have been achieved).

Problem 2. Perform the proof obligations for $INIT$ and $ACHIEVE\_2$.

Problem 3. (project) The four Blocks World *grasp* actions may be written as one paramerterised action (refer to Exercise 2, no. 2):

name: *grasp* X
pre-conditions: 'X is a block', 'X has a clear top', 'gripper is free' ;
add-set: 'gripper grasps X' ;
delete-set: 'X has a clear top', 'gripper is free' ;

and in general it is much more expedient to pose actions as paramerterised structures. Re-specify both design levels of the planner with the extension that actions can be posed with parameters.

Problem 4. (hard) Show that a *Partial_Plan* which has an empty Ps necessarily contains a plan which satisfies the post-condition of $PLANNER$. Hence demonstrate the connection between the abstract planning specification of $PLANNER$ and the design level planner of section 4.

Problem 5. Add a pre-condition to $ACHIEVE\_2$ to make it satisfiable.

5. Protoyping the Specification

In this second part of the paper we we will describe some of the principles of prototyping model-based specifications, and illustrate the idea by constructing prototypes in the Prolog programming language. Our main example will be prototyping the case study of the last chapter, resulting in a full implementation of the design level specification of the Non-linear Planner, which is supplied in Appendix 1.

Prototyping a model based specification S means translating S into a program P which, though not necessarily satisfying efficiency or other non-functional constraints, is correct with respect to S. This allows the specification to be *animated*, which enables developers and end users to check early in the development stage that the specification is a valid representation of their requirements. Other advantages include:

- the promise of a working prototype lessens the risk factor involved for a software purchaser: instead of waiting untill a full implementation is available, the prototype is a working model which gives a good indication of what the final product will be like;
- constructing a prototype helps in debugging the specification: even after proof obligations have been successfully discharged there may remain errors in the logic;
- it is pleasing, after expending much effort on a static, mathematical specification, to be able to get something working!

If the programming language in which P is created (call it PL) is well chosen, then prototyping P can be a semi-automatic process with a very high degree of certainty that P will be correct with respect to S. To be a good prototyping language, PL should contain constructs that are similar to the specification language (say SL) in which S is written. In effect, this means that the *semantic gap* between SL and PL should be small with respect to:

- operations: it should be straightforward to translate operations and functions written in SL into PL.
- data types: PL should either support the same data types as SL, or it should be easy to implement them.

The main difference between an SL and a PL is that a PL has a *procedural semantics*. This means that a program P written in PL is executable in the sense that it (together with an interpreter for PL) can accept input data and should output data. If P is a correct implementation this input-output relationship will always conform to specification S. For example, with SL = VDM, PL = Prolog, our prototype planning program in Appendix 3 should input planning problems and output solutions conforming to the specification in section 4. The relationship between implicit and explicit function definitions is analogous to the relationship between a specification and its prototype.

The main deficiency of prototyped specifications is that they tend to be *inefficient* in time and space. This is due to two factors:

- a naive implementation: if P is written following the same structure as S, the program itself may be grossly inefficient.
- compilers producing slow code: good candidates for PL (such as Prolog) have compilers which tend to produce less efficient code than imperative languages (such as Ada or Modula-2). This is not surprising, as imperative languages reflect the prevailing computer architecture.

Another problem faced when prototyping many specifications which concern the functional aspects of software is that an interface must be constructed to handled input and output. In the simplest terms, procedures for efficiently allowing the input of data, and presenting the output in an intelligible form, need to be written.

6. Prolog as a Prototyping Language for VDM

6.1 Prolog

Prolog is a general purpose logic programming language that was developed in the 1970's for use in Artificial Intelligence, especially in the area of Natural Language Processing. Although many interpreters, compilers and software development environments exist for it, most dialects conform to a standardised version called Edinburgh Prolog. We will assume that the reader is familiar with Edinburgh Prolog, and use it as our prototyping language (in what follows by *Prolog* we mean *Edinburgh Prolog*). Interested readers will find a good introduction to Prolog in [Clocksin and Mellish 84].

There are many advantages in the use of Prolog, including its:

- simple form: A Prolog program is a list of clauses, each clause being a *fact* or a *rule*. Facts are predicate structures of the form:
  $fact\_name(t_1, ..., t_n)$.
  where $n \geq 0$, and each t_i is a *term* defined as

  - a constant or
  - a variable or
  - a function symbol containing zero or more terms as arguments.

  Rules are of the form
  a :- $b, c,$ ... , d.
  where

  - '$a, b, c, ..., d$' are predicates structures,
  - 'a' is called the *head* of the rule,
  - '$b, c, ..., d$.' is called the *tail* of the rule,
  - ':-' is read as 'follows from'.

  A group of clauses which share the same head predicate is called a procedure.
- 'dual' semantics: Prolog programs can be interpreted in two ways:

  - *declaratively*: each clause can be interpreted as a logic formula, generally a fact or a rule. Our typical rule a :- $b, c,$ ... , d. corresponds to the logical formula:
    $(b \wedge c \wedge ... \wedge d) \;\Rightarrow\; a$
    where all variables in the rule are universally quantified.
  - *procedurally*: each clause can be read as a goal oriented procedure, which asserts 'to solve the head, solve all the predicates in the tail'. The head predicate is akin to a procedure's heading, whereas the predicates in the tail can be interpreted as procedure calls. One consequence of the goal-oriented interpretation of Prolog's clauses is that a VDM post-condition can be modelled as a set of Prolog goals to be achieved.

- high level data structures: Prolog's term and list data structures can be easily adapted to implement VDM data structures.
- backtracking: Prolog's backtracking mechanism is a form of control which resembles naive search. The search is for instances of a goal predicate's variables which make a goal succeed. Later we will see how backtracking can be used to prototype existentially quantified conditions in VDM operations.

The first two advantages pointed out above just apply to *Pure Prolog*, a subset of Prolog which does not contain any side effects. Of course, as a programming language Prolog contains expressions which cannot readily be given a logical interpretation, such as *read* and *write* procedures. The language has various other disadvantages of which the user should be aware:

- Prolog is not a strongly typed language, and in fact variables in predicates are not type-restricted in any way. This means that there is a danger of procedures being called with unsuitable parameter values. With a strongly typed language (Pascal, for example) if a procedure is called with too many or the wrong type of argument values, the implementation would automatically flag a type mismatch at or before run-time, and the user would be aware of and could pin-point the error. In Prolog the run-time behaviour in this case would be unpredictable, and the error would be difficult to detect and fix.
- Prolog has no (standard) module mechanism. This means data type encapsulation and information hiding are not supported.
- Prolog does not support functional evaluation, except in some special circumstances such as numerical evaluations. This means that every functional expression in a VDM condition must be translated into a Prolog procedure which has an extra slot for an output value. This extra slot contains a dummy

parameter which carries the function value to the next evaluation. For example, consider the following VDM expression:

$S \cap \mathbf{dom}\ M$

where S is a set, and M a map. Both functions '$\cap$' and '*dom*' must be implemented as Prolog procedures. If we let these procedures be called 'dom_map' and 'intersect_set', then this expression translates to the Prolog predicates:

```
dom_map(M,  DomM), intersect_set(S,DomM,  Result)
/* post-condition: Result = S intersect dom(M) */
```

'DomM' is the dummy parameter carrying the result of the first function evaluation to the next evaluation.

- Prolog is case sensitive in that all variables have to start with a capital letter, and all predicate, function and constant names start with a lower case letter.

6.2 VDM to Prolog Translation

The prototyping process needs to be supported with a set of tools and techniques specific to Prolog. Firstly, we will show how a set of tools can be created to support VDM data types; then we will devise a method to translate systematically VDM operations into Prolog clauses. The fact that the logic of pre- and post-conditions can be reflected within the declarative semantics of Prolog, makes the translation relatively straightforward, although one must also be aware of certain pitfalls.

VDM has been successfully prototyped using other programming languages, and various support kits have been written. For example, 'me too' [Henderson and Minkowitz 86] was an early system for prototyping VDM, using LISP as the target language. As a preliminary to our implementation fragments below, we set out some conventions for Prolog:

- code will be put in a bold typeface (unlike *pseudo-code* which appears in italics);
- procedures in the toolkits implementing the Set, Sequence, Map and Composite will have 'set', 'seq', 'map', and 'comp' tagged on to the end of their names accordingly;
- primitive recursive procedures will be headed with pre- and post-conditions, relating input and output parameters.
- output parameters will be placed to the right in a procedure heading, separated from the input parameters by three spaces (there may be occasional exceptions to this, as some parameters may be used for input *or* output);
- where we represent VDM states as parameters below, the input state parameters will be tagged with the letter 'I', and output state parameters with the letter 'O'.

6.3 Data Type Implementation

VDM's Set, Sequence, Map and Composite types can all be implemented quite easily. In this section we show the reader how to implement the Set type only, although implementations of all the four types are given in appendix 3. First it is necessary to represent the Set with a Prolog structure, and to do this we will use Prolog's List. The examples of sets in chapter 3:

$\{1, 4, 9, 16, 25, 36, 49, 64, 81\}$
$\{1, 2, 3, 5, 7, 11\}$
$\{\ \}$

can be written as Prolog lists as follows

$[1, 4, 9, 16, 25, 36, 49, 64, 81]$
$[1, 2, 3, 5, 7, 11]$
$[\,]$

Recall that whereas in a set representation the left to right ordering of elements is irrelevant, in a list (which is similar to a sequence) different orderings of the same elements denote different lists. Hence a special set equality predicate must be defined, and we do this in terms of the subset ($\subseteq$) relation, which in turn will be defined using the 'element of' ($\in$) relation:

$X = Y$ if and only if $X \subseteq Y \land Y \subseteq X$
$X \subseteq Y$ if and only if $\forall\, e \in X \cdot e \in Y$

To implement set equality, we first implement the '$\in$' relation using a simple list processing procedure:

```
/* element_of_set(E,Y)                          */
/* pre: Y is a set                              */
/* post: E is an element of Y                   */
/*       iff element_of_set(E,Y) succeeds       */
/*(1)*/ element_of_set(E,[E|Y]).
/*(2)*/ element_of_set(E,[_|Y]) :-
           element_of_set(E,Y).
```

(1) reads as: "element_of_set' succeeds if the element to be tested is at the head of the list'. The rule in (2) reads as: 'if (1) is not the case, then apply 'element_of_set' to the tail of the list. The empty slot '_' in 'element_of_set(E,[_|Y])' represents a variable which we do not need to give a name to (as it is not used elsewhere in the rule). Next we can define subset, and finally the equality predicate (which we call 'eq_set'):

```
/* sub_set(X,Y)                                 */
/* pre: X,Y sets                                */
/* post: X is a subset of Y                     */
         iff sub_set(X,Y) succeeds              */
sub_set([First_Element|Rest],Y) :-
     element_of_set(First_Element,Y),
     sub_set(Rest,Y),!.
sub_set([],Y).

/* eq_set(X,Y)                                  */
/* pre: X,Y sets                                */
/* post: X=Y iff eq_set(X,Y) succeeds           */
eq_set(X,Y) :-
     sub_set(X,Y),
     sub_set(Y,X),!.
```

Note the use of Prolog's 'cut' ('!') in the last two procedures. The 'cut' is a device to stop Prolog's backtracking mechanism: procedures which are implementing *functions* only have one output value that satisfy their inputs, therefore backtracking within their procedural definition is worthless. Using the 'cut' in the manner above stops backtracking *within* the function defintion.

The subset procedure shows the disadvantage of Prolog not being a strongly typed language: the second clause asserts that the empty list (or empty set in our interpretation) is a subset of *anything*. We have to rely on the commented preconditions being followed for the integrity of this procedure. An alternative suggested in exercise 7.2 would be to create the procedure $is\_a\_set(X)$ which checks variables, returning true only if its argument is a set. This way a type mechanism can be implemented on top of Prolog.

Set functions are implemented in terms of the 'element_of_set' procedure:

- The union operator '∪':

```
/* union_set(X,Y,Z)                              */
/* pre: X,Y sets                                 */
/* post: Z = X union Y                           */

union_set([E|X],Y,[E|Z]) :-
     not(element_of_set(E,Y)),
     union_set(X,Y,Z),!.
union_set([E|X],Y,Z) :-
     element_of_set(E,Y),
     union_set(X,Y,Z),!.
union_set([],Y,Y).
```

- The intersection operator '∩':

```
/* intersect_set(X,Y,Z)                          */
/* pre: X,Y sets                                 */
/* post: Z = X intersect Y                       */

intersect_set([E|X],Y,[E|Z]) :-
     element_of_set(E,Y),
     intersect_set(X,Y,Z),!.
intersect_set([E|X],Y,Z) :-
     not(element_of_set(E,Y)),
     intersect_set(X,Y,Z),!.
intersect_set([],Y,[]).
```

- The set difference operator '-':

```
/* minus_set(X,Y,Z)                              */
/* pre: X,Y sets                                 */
/* post: Z = X minus Y                           */

minus_set([E|X],Y,Z) :-
     element_of_set(E,Y),
     minus_set(X,Y,Z),!.
minus_set([E|X],Y,[E|Z]) :-
     not(element_of_set(E,Y)),
     minus_set(X,Y,Z),!.
minus_set([],Y,[]).
```

0.0.1 Exercises 9

1. The VDM primitive function *tl* can be implemented with just one Prolog fact:

```
/* tl_seq(List_in, List_out):                    */
/* post:  List_out = tl(List_in)                 */
tl_seq([H|T], T).
```

Implement the rest of the Sequence data structure's functions in Prolog.

2. Implement the Composite and Map types in Prolog, ensuring that the representation of these structures is insulated from the rest of the program. To do this you need to provide the implementations for the procedures specified below, which initialise, add and retrieve values to and from these structures.

```
/* init_comp(Name, List_of_slot_names, List_of_slot_values,   Comp ):  */
/* post: Comp is a composite structure with name Name, and a list      */
/* of component names given in List_of_slot_names, with corressponding  */
/* values in List_of_slot_values                                        */

/* put_comp(Comp, Slot_name, Value,   NewComp):                         */
/* post: NewComp = Comp except Slot_name(NewComp) = Value               */

/* get_comp(Comp, Slot_name,   Value):                                  */
/* post: Value = Slot_name(Comp)                                        */

/* init_map(  Map)                                                      */
/* post: Map is an empty map                                            */

/* overwrite_map(Map, Dom, Value,   NewMap):                            */
/* post: NewMap = Map + [Dom -> Value]                                  */

/* apply_map(Map, Dom,   Value):                                        */
/* post: Value = Map(Dom)                                               */

/* dom_map(Map,   Dom_Map):                                             */
/* post: Dom_Map = dom(Map)                                             */

/* ran_map(Map,   Ran_Map):                                             */
/* post: Ran_Map = ran(Map)                                             */
```

6.4 Operator Translation

Some basic rules for the translation of VDM operators into Prolog are given below. We start with the assumption that a VDM operator has the following form:

$OP\_NAME\ (i_1 : ti_1, ..., i_n : ti_n)\ o : to$
ext wr $s : state\_type$
pre $pre(i_1, ..., i_n, s)$
post $post(i_1, ..., i_n, \overleftarrow{s}, o, s)$

where $i_1, .., i_n$ is a list of input parameters, o an output parameter, and $ti_1, .., ti_n, to$ their respective types.

The head of the Prolog procedure that is created from this will have the following form:

$op(i_1, .., i_n, \overleftarrow{s}, o, s)$

Input and output states are represented with explicit parameters, to avoid the problem of managing global data. Otherwise, the VDM state can be represented in Prolog as a collection of facts, or as a single fact, and state changes performed by Prolog's *assert* and *retract* predicates. The ad hoc use of global data effected by *assert* and *retract* tends to make Prolog programs unmaintainable.

The *op* procedure can have a number of clauses, depending on the logical form of the pre- and post-conditions. Disjunction or implication in an operator's conditions leads to a definition with several clauses. We will assume for the moment that the conditions are conjunctions of predicates without any quantified variables, that is:

$pre(i_1, ..., i_n, s) = p_1 \wedge p_2 \wedge ... \wedge p_k$
$post(i_1, ..., i_n, \overleftarrow{s}, o, s) = q_1 \wedge q_2 \wedge ... \wedge q_l$

Now each of the p_i and q_j could be either:

- primitive predicates, pre-defined in VDM, such as '$\in$' and '$=$', or
- user-defined predicates, such as *achieve* and *before* of chapter 6.

We let each of these types of predicates be implemented by a corresponding Prolog procedure: $proc\_p_1$ for p_1, $proc\_q_1$ for q_1, and so on. The names $proc\_p_1, proc\_p_2, ...$ represent procedures which may contain input parameters (including the input state parameter). The names $proc\_q_1, proc\_q_2, ..$ each represent procedures, but they may contain references to any input or output parameters or states. A simple monolithic translation, combining pre-condition procedures with those in the post-condition into one clause, gives:

$op(i_1, .., i_n, o_1, ..., o_m, \overleftarrow{s}, s)$:-
$proc\_p_1$,
$proc\_p_2$,
...,
$proc\_p_k$,
$proc\_q_1$,
$proc\_q_2$,
...,
$proc\_q_l$.

The specification of each $proc\_p_i$ is given essentially by treating p_j as its post-condition. The specification of each $proc\_q_i$ is a little more complicated since these procedures effectively have to achieve the post-condition of op. In achieving a predicate, there is always the possibility that an already achieved predicate may be un-achieved! The reader should see the parallel with *clobbering* in the planning application of chapter 6, where the effect of one action could undo the achievement of one goal. The similarity is not surprising since programming to achieve a specification is a form of planning.

In fact the post-condition of $proc\_q_j$ is $q_1 \wedge q_2 \wedge ... \wedge q_j$, since we require $proc\_q_j$ to achieve condition q_j, while preserving conditions $q_1 \wedge q_2 \wedge ... \wedge q_{j-1}$. Although strictly speaking this is the case, it is more expedient to consider q_j as the post-condition of $proc\_q_j$, and keep a watchful eye that previous predicates are not undone.

A more elaborate way to translate operators into Prolog procedures is to first create a procedure which executes the post-condition procedures only if the pre-condition is met:

$op(i_1, .., i_n, o_1, ..., o_m, \overleftarrow{s}, s)$:-
$proc\_p_1$,
$proc\_p_2$,
...,
$proc\_p_k$,
$execute\_op\_name(i_1, .., i_n, o_1, ..., o_m, \overleftarrow{s}, s)$.
$op(i_1, .., i_n, o_1, ..., o_m, \overleftarrow{s}, s)$:-
write('operator pre-condition failure').

If the pre-condition procedures succeed, the first clause calls *execute_op* which 'executes' the operator. The second clause outputs an error message if the pre-condition procedures fail, which means the operator cannot be executed. *execute_op_name* is defined:

$execute\_op\_name(i_1, .., i_n, o_1, ..., o_m, \overleftarrow{s}, s)$:-
$proc\_q_1$,
$proc\_q_2$,
...,
$proc\_q_l$.

This second form, although longer, is more secure since errors in the implementation of the post-condition will not be confused with failing pre-conditions. To keep the length of the code fragments in this chapter to a minimum, however, we adopt the first form.

Examples 5

1. The estate agent database operator *MAKEOFFER* is defined thus:

$$\begin{array}{l} MAKEOFFER\,(addr : Address) \\ \textbf{ext wr}\;\; forsale, underoffer : Address\textbf{-set} \\ \textbf{pre}\;\; addr \in forsale \\ \textbf{post}\; forsale = \overleftarrow{forsale} \setminus \{addr\} \wedge underoffer = \overleftarrow{underoffer} \cup \{addr\} \end{array}$$

Prototyping *MAKEOFFER* is straightforward, because its pre- and post-conditions are conjunctions of predicates. It translates into one clause consisting of primitive procedures:

```
makeoffer(Addr, ForsaleI, UnderofferI, ForsaleO,   UnderofferO) :-
      element_of_set(Addr,   ForsaleI),
      minus_set(ForsaleI, [Addr],   ForsaleO),
      union_set(UnderofferI, [Addr],   UnderofferO).
```

2. Consider the *ADD* operator of the *Symbol_table* of chapter 4:

$$\begin{array}{l} ADD\;(i : Identifier, a : Attribute) \\ \textbf{ext wr}\;\; s : Symbol\_table \\ \textbf{pre}\;\; s \neq [\,] \wedge i \notin \textbf{dom}\;(\textbf{hd}\; s) \\ \textbf{post}\;\; s = [(\textbf{hd}\; \overleftarrow{s}\,) \dagger \{i \mapsto a\}] \frown \textbf{tl}\; \overleftarrow{s} \end{array}$$

Again, the implementation is made up completely from data structure primitives:

```
add(I, A, SI,   SO) :-
       not(SI = [ ]),                            /* pre-conditions:  */
       hd_seq(SI,   SIhd),
       dom_map(SIhd,   DomSIhd),
       not(element_of_set(I,   DomSIhd)),
       overwrite_map(SIhd,I,A,   SIhd1),         /* post-conditions: */
       tl_seq(SI,   SItl),
       append_seq([SIhd1],SItl,   SO).
```

Notice how the dummy variables (SIhd, DomSIhd, SIhd1) are required to store the results of operator evaluation.

Exercise 10

1. Prototype the *DELETE_HOUSE*, *PUSH* and *POP* operations of chapters 3 and 4.

2. Create a general translation method for VDM operations which includes conditions containing '$\vee$', the logical 'or' operator.

3. Implement the type checking predicate $is\_a\_set(X)$, and hence create a type checking mechanism for the Prolog implementation of VDM data structures.

6.4 The Existential Quantifiers

The quantifier '$\exists$' invariably occurs in a VDM condition in the following type of expression:

$\exists\, X \in \mathit{some\_set} \cdot p \ldots$

where X occurs as a free variable in p. This can be simulated in Prolog using a free variable for X in the 'element_of_set' procedure. Assuming procedure 'proc' achieves predicate p (that is 'proc' is the the prototyped version of p), the expression above can be translated to the piece of code:

```
 element_of_set(X, Some_set), proc, ...
```

X will become instantiated with the first element of 'Some_set' by the execution of 'element_of_set'. If 'proc' fails with that instantiation, Prolog's backtracking mechanism will re-call 'element_of_set' which will succeed with another instance for X. This will continue systematically (because 'Some_set' is implemented as a list) until 'proc' eventually succeeds. Hence Prolog's backtracking mechanism persists in backtracking to 'element_of_set' until the correct element is picked. If all elements are exhausted and 'element_of_set' eventually fails, the condition cannot be satisfied.

Example 6

A RETRIEVE operation can be written to rely on an existentially quantified variable b in its pre-condition:

> $\mathit{RETRIEVE}\ (i : \mathit{Identifier})\ a : \mathit{Attribute}$
> **ext rd** $s : \mathit{Symbol\_table}$
> **pre** $\exists\, b \in \textbf{elems}\ s \cdot i \in \textbf{dom}\ b$
> **post** $a = \mathit{get\_from\_table}(s, i)$

This translates into:

```
retrieve(I,S,   A) :-
     elems_seq(S,   SetS),
     element_of_set(B,   SetS),
     dom_map(B,   domB),
     element_of_set(I,   domB),
     get_from_table(I,S,   A).
```

In this example, backtracking actually occurs across the middle procedure 'dom_map'. Whatever instance of 'B' the procedure 'element_of_set' produces, 'dom_map', being a total function, succeeds. The fourth procedure call in the clause fails until the correct choice of 'B' has been picked.

6.5 The Universal Quantifier

The universal quantifier appears in a VDM condition as the existential quantifier above:

$\forall\, X \in \mathit{some\_set} \cdot p \ldots$

where X occurs as a free variable in p. To implement this expression in Prolog the procedure (call it 'Proc') corresponding to the predicate p has to be called repeatedly for all instances of X in *some_set*. This contrasts with the case of the existentially quantified variable, where only one successful procedure call needs to be made. Also, if the scope of X is more than one predicate, then we let 'Proc' correspond to all these predicates.

For this implementation we need two special Prolog predicates (the reader who is not too concerned about low level implementation may simply examine the post-condition of the implementing procedure below and move to the next section):

- the meta-predicate *call*(X), which executes its argument X as a normal Prolog goal.
- the infix operator '=..', which succeeds if its right hand argument is a list of terms making up the structure of its left hand argument. For example the Prolog goal 'f(x) =.. X' would succeed with X bound to [f, x].

Using *call*(X) we can create an iterative procedure with the name 'for_all_els' which repeatedly calls the procedure 'Proc'. This procedure call will have a different value from *some_set* on each invocation:

```
\* post: for_all_els( Some_set, Proc) is true
        iff for all X in Some_set : p is true */
for_all_els( [ ], Proc).
for_all_els( [E|L], Proc) :-
     Proc =.. OL,
     append(OL,[E], OL1),
     ProcE = ..OL1,
     call(ProcE),
     for_all_els( L, Proc).
```

For this to succeed, the procedure 'Proc' must be defined so that its last argument expects a value from the set S. 'for_all_els' is then supplied with the 'Proc' procedure instance without its last argument. Then, the three procedure calls

```
     Proc =.. OL,
     append(OL,[E], OL1),
     ProcE =.. OL1,
```

succeed in gluing on an element of 'Some_set' to the end of the procedure 'Proc', which is then called with the correct number of arguments.

Example 7

The specification for MAX_IN in chapter 3 contained a universal quantifier:

$MAX\_IN\ (s : \mathbf{N\text{-}set})\ m : \mathbf{N}$
pre $s \neq \{\}$
post $m \in s \land \forall\, i \in s \cdot i \leq m$

It translates as follows:

```
max_in(S,   M) :-
     not( S = [ ]),
     element_of_set(M, S),
     for_all_els(S, less_eq(M)).
less_eq(M, E) :- M =< E.
```

The output parameter 'M' here is systematically instantiated with elements of 'S', until eventually one is found which satisfies the universally quantified condition. Procedures like 'for_all_els' that invoke meta-predicates act in a similar fashion to *higher order functions* in functional programming languages.

Exercise 11

A complication arises if the universal quantifier occurs in the post-condition, and the predicate p relates input and output (state) parameters. As each time 'Proc' is invoked, its output parameters need to be supplied as input to the next call of 'Proc' with a new set element. In this case, the procedure implementing 'for all' must keep track of the changing input parameter as each version of 'Proc' is called.

Implement a new version of 'for_all_els' called 'for_all_elsIO' along these lines. It should include two extra arguments for the input and output parameters, as follows:

```
\* post: for_all_elsIO(S,Proc,In,Out) is true iff
         forall X in S: p(In,Out,X) is true       */
```

The answer to this exercise also appears in Appenix 1.

6.6 Prototyping Pitfalls

The main problem in producing naive prototypes along the lines above is the threat of producing grossly inefficient code. The implementation of MAX_IN was such an example. A more extreme case arises from the specification of a sort function below, which inputs an arbitrary sequence of integers and outputs the ordered sequence:

$SORT\,(in : \mathbf{N}^*)\ out : \mathbf{N}^*$
pre *true*
post $permutation(in, out) \wedge ordered(out)$

Assuming *permutation* and *ordered* are defined elsewhere, the naive translation to the top level procedures would be:

```
sort(In,   Out) :-
     permutation(In,Out),
     ordered(Out).
```

The implementation that this leads to is very inefficient: the first procedure *permutation* will blindly generate permutations of the initial sequence, until one is eventually found that is ordered. Though the specification of *sort* seems to be a natural one, an implementation that follows its structure is inadequate.

Output state variables and the output parameter in implicitly defined operations can cause related problems. In theory, they are dealt with in a similar way to existentially quantified variables: Prolog's backtracking mechanism iterates until values are found for them which satisfy the post-condition. Unfortunately, in very abstract definitions, this is impractical. An example which we met in chapter 6, was the top level specification of the Planner:

$PLANNER\,(pp : Planning\_Problem)\ soln : Action^*$
pre *true*
post elems $soln \subseteq pp.AS \wedge$
$complete(soln, pp.I)) \wedge$
$pp.G \subseteq apply\_seq(soln, pp.I)$

Taking the predicates in the order they are written, to produce a prototype we might construct a procedure to generate actions sequences from the set of all actions in the planning problem, but this would lead to a hopeless implementation!

One general way of improving efficiency in prototypes is by judicious re-ordering of its procedures, although any ordering of the procedures produced from $SORT$ and $PLANNER$ would result in at least a very inefficient implementation[3]. Specifications such as these would have to be transformed or refined before the kind of method we are advocating would be worthwhile.

We end the section on a more optimistic note: this final example will show how a computationally 'explosive' prototype can be rescued and made efficient with a correctness preserving transformation. The *before* function in chaper 5 was defined with the use of an existential quantifier:

$$\mathit{before} : \mathit{Nodes} \times \mathit{Nodes} \times \mathit{Arc}\textbf{-set} \rightarrow \mathbf{B}$$

$$\begin{aligned}&\mathit{before}\,(x, z, p) \triangleq \\ &\quad \mathit{mk\text{-}Arc}(x, z) \in p \vee \\ &\quad \exists\, y \in \mathit{get\_nodes}(p) \cdot \mathit{before}(x, y, p) \wedge \mathit{before}(y, z, p)\end{aligned}$$

Disjunction in the function definition means that the Prolog procedure will have to be split into a list of clauses, one for each formulae connected by the disjunction. In this case we will have two clauses: the first constructs the arc composite, and checks whether it already occurs in the poset p:

```
before(X,Z,P) :-
       init_comp(arc, [source,dest], [X,Y],    ARC),
       element_of_set(ARC,    P),!.
```

The second clause contains the recursive calls, to find arc paths:

```
before(X,Z,P) :-
       get_nodes(P,   Nodes),
       element_of_set(Y, Nodes),
       before(X,Y,P),
       before(Y,Z,P).
```

The problem is that the double call to procedure 'before' does not lead to a systematic algorithm. A logically equivalent definition is as follows:

$$\mathit{before} : \mathit{Nodes} \times \mathit{Nodes} \times \mathit{Arc}\textbf{-set} \rightarrow \mathbf{B}$$

$$\begin{aligned}&\mathit{before}\,(x, z, p) \triangleq \\ &\quad \mathit{mk\text{-}Arc}(x, z) \in p \vee \\ &\quad \exists\, y \in \mathit{get\_nodes}(p) \cdot \mathit{mk\text{-}Arc}(x, y) \in p \wedge \mathit{before}(y, z, p)\end{aligned}$$

This version leads to a workable prototype because the search for a path is defined in a systematic manner:

```
before(X,Z,P) :-
       get_nodes(P,   Nodes),
       element_of_set(Y, Nodes),
       init_comp(arc, [source,dest], [X,Y],   ARC),
       element_of_set(ARC,  P),
       before(Y,Z,P).
```

[3] In fact, systematically generating instances of *soln* from $complete(soln, I(pp))$ in a planner corresponds to very inefficient search strategies such as 'breadth-first search'.

Exercises 12

1. Prototype the rest of the poset specifications in chapter 5, in Prolog (note that the solution can be found in appendix 1).

2. Using either a formal or diagramatic argument, prove that the two versions of *before* given above are equivalent.

7. The Planner Prototype

The method described above is used now to prototype the Planner specified in 6.4. Although the implementation of some of the functions is left as an exercise for the reader, the answers can be found in a full listing of the whole prototype in Appendix 1. We develop the prototype in a top-down fashion, rooting the procedures eventually in the primitive data structure functions of Section 6.

Each of the three operations *INIT*, *ACHIEVE_1* and *ACHIEVE_2* will be translated to a top level Prolog procedure, using the methods outlined above. The efficiency problem is not so acute in this application, because of the fairly concrete, and goal directed nature of the specification. Nevertheless, re-ordering the goals in some of the resulting Prolog clauses certainly makes the prototype more efficient, while preserving the correctness of the code.

7.1 The *INIT* Operator

The initialisation operator can be translated using the notation introduced in Exercise 9 no. 2 for the Composite and Map types. One problem remains - that of set comprehension. Rather than producing a general procedure to deal with set comphrension, we present a specialised solution for the expression:

$\{mk\text{-}Goal\_instance(g, goal) : g \in G(pp)\}$

This translates into Prolog using a recursive procedure which builds up a set of goal instances as follows:

```
/* make_goal_instances(A, Gs,   Gi)                        */
/* pre: Gs is a literal set, A is an action identifier     */
/* post: Gi = {mk-Goal_instances(g,A) : g is in Gs}        */
make_goal_instances(Action_Id, [G|G_rest],   [Gi|Gi_rest]) :-
     init_comp(goal_instance, [gl, ai], [Action_Id, G],   Gi),
     make_goal_instances(Action_Id, G_rest,   Gi_rest).
make_goal_instances(_, [],   []).
```

Making use of this predicate, the translation of the *INIT* operator is then:

```
init(PPI,   PPO) :-
     get_comp(PPI, planning_problem, i,   IPP),
     get_comp(PPI, planning_problem, g,   GPP),
     init_comp(action, [name,pre,add,del], [ init ,[], IPP, [] ],   INIT),
     init_comp(action, [name,pre,add,del], [ goal , GPP, [], [] ],   GOAL),
     init_map(  OS),
     overwrite_map(OS, init, INIT,   OS1),
     overwrite_map(OS1, goal, GOAL,   OS2),
     make_goal_instances(goal, GPP,   GIs),
     initPO(Ts),
     init_comp(partial_plan, [pp,os,ts,ps,as], [PPI,OS2,Ts,GIs,[]],   PPO).
```

The proliferation in variable names (OS1, OS2, for example) is not only due to the lack of functional evaluation in Prolog, but also the price of abstraction: whereas we chose to make the representation of sequences and maps visible (as lists), the representation of maps and composites has been hidden within the definition of these structures' primitive functions.

7.2 The *ACHIEVE* Operators

ACHIEVE_1 can now be implemented by a procedure called 'achieve1', in which all the procedures are primitive data functions except for the 'achieve' predicate itself. As an external state is not accessed, the operation must use an access function (get_comp) to break up the input plan:

```
achieve1(PlanI, Gi,   PlanO) :-
     get_comp(PlanI, partial_plan, os,   Os),
     get_comp(PlanI, partial_plan, ts,   Ts),
     get_comp(PlanI, partial_plan, ps,   Ps),
     get_comp(PlanI, partial_plan, as,   As),

     element_of_set(Gi, Ps),    /* pre-condition */

     dom_map(Os,   DomOs),      /* post-condition: */
     element_of_set(A, DomOs),
     achieve(Os,Ts,A,Gi,   Ts_new),
     minus_set(Ps, [Gi],   Ps_new),
     union_set(As, [Gi],   As_new),

     put_comp(PlanI, partial_plan, ts, Ts_new,   Plan1),
     put_comp(Plan1, partial_plan, ps, Ps_new,   Plan2),
     put_comp(Plan2, partial_plan, as, As_new,   PlanO).
```

Notice how these implementations follow the specification virtually line by line, except that Prolog is more longwinded because of its need for procedures to return function values explicitly. Our 'predicate to procedure' correspondence is broken in that the implementation of the 'achieve' procedure has as its post-condition two predicates:

$$\begin{array}{l} achieve(Os, Ts, A, gi) \wedge \\ is\_completion\_of(Ts, \overleftarrow{Ts}) \end{array}$$

The last conjunction is effectively met if we constrain the implementation of 'achieve' so that it only adds constraints to Ts, and adds no new nodes (that is actions). The definition of 'achieve' in chapter 6 is:

$$achieve : Action\_instances \times Bounded\_Poset \times Action\_id \times Goal\_instance \to \mathbf{B}$$

$$\begin{array}{l} achieve\ (Os, Ts, A, mk\text{-}Goal\_instance(p, O)) \ \underline{\triangle} \\ \quad before(A, O, Ts) \wedge \\ \quad p \in Os(A).add \wedge \\ \quad \neg\, (\exists\, C \in \mathbf{dom}\ Os \cdot \\ \quad possibly\_before(C, O, Ts) \wedge \\ \quad possibly\_before(A, C, Ts) \wedge \\ \quad p \in Os(C).del) \end{array}$$

It will need to be transformed somewhat, to give a simpler implementation. We move the negation '*not*' inwards, using the result of exercise 6.7 no. 2, and introduce an auxilliary predicate *declobber_achieve* to produce the new form:

$achieve : Action\_instances \times Bounded\_Poset \times Action\_id \times Goal\_instance\ To\mathbf{B}$

$achieve\ (Os, Ts, A, mk\text{-}Goal\_instance(p, O)) \triangleq$
$\quad before(A, O, Ts) \wedge$
$\quad p \in Os(A).add \wedge$
$\quad \forall\, C \in \mathbf{dom}\ Os \cdot declobber\_achieve(p, A, O, Os, C, Ts)$

where:

$declobber\_achieve(p, A, O, Os, C, Ts) \triangleq$
$C = O \vee$
$C = A \vee$
$before(O, C, Ts) \vee$
$before(C, A, Ts) \vee$
$\neg\,(p \in Os(C).del)$

Now we have two distinct cases to consider: one in which *achieve* succeeds without changing the input state's temporal order Ts at all; and another, in which constraints have to be added to Ts. Thus we have the following cases:

- an achieving action 'A' is found for p, and no actions present in Os clobber (that is undo) this achievement. Therefore no constraint need be added to the temporal order (Ts would remain unchanged).
- an achieving action 'A' is found for p, but there is at least one action which clobbers p, and there is at least one way of declobbering it (that is putting constraints into Ts which avoid the goal literal p being clobbered).

It is desirable to make the Planner take a *least commitment* approach to forming plans, and so if the first case is true then the second case need not be explored. On the other hand, if the first case is false then we want $ACHIEVE\_1$ to be able to make a non-deterministic choice among the set of possible declobbering constraints (potentially therefore, a backtracking mechanism could generate every possible choice). The first part of the *achieve* implementation is (in the code below we put in comments next to a procedure its the corresponding predicate post-condition, where possible):

```
achieve(Os,Ts,A,GI,   New_Ts) :-
     get_comp(GI,goal_instance, ai,  O),
     get_comp(GI,goal_instance, gi,  P),
     apply_map(Os,A,   ActionA),
     get_comp(ActionA,action,add,   AddA),
     element_of_set(P,   AddA),              /* P is in Os(A).add  */
     make_before(A,O,Ts,   Ts1),             /* before(A,O,Ts1)    */
     dom_map(Os,   DomOs),
     for_all_elsIO(DomOs, declobber_achieve(P,A,O,Os), Ts1,   New_Ts).
```

Again, the reader should notice how the code mirrors the specification. One change in ordering we have made is in switching around the 'apply_map' and 'make_before' procedures - this improves the efficiency of the planner.

The 'declobber_achieve' procedure has two parts: the first, where the partial order Ts remains unchanged, and the second in which it is necessary to add a constraint. If the first part succeeds we do not require any other alternatives involving temporal constraint additions, therefore Prolog's 'cut' will be used to cut down the alternatives in that case:

```
declobber_achieve(P,A,O,Os,O,Ts,   Ts) :- !. /* C = O V */
declobber_achieve(P,A,O,Os,A,Ts,   Ts) :- !. /* C = A V */
```

```
declobber_achieve(P,A,O,Os,C,Ts,   Ts) :-
     before(O,C,Ts),!.                     /* before(O,C,Ts) V */
declobber_achieve(P,A,O,Os,C,Ts,   Ts) :-
     before(C,A,Ts),!.                     /* before(C,A,Ts) V */
declobber_achieve(P,A,O,Os,C,Ts,   Ts) :-
     apply_map(Os,C,   CA),
     get_comp(CA,action,del,   CAD),
     not(element_of_set(P,CAD)),!.         /* not( p in Os(C).del) */

declobber_achieve(P,A,O,Os,C, Ts,   New_Ts) :-
     make_before(O,C,Ts,   New_Ts).        /* make  before(O,C,Ts) */
declobber_achieve(P,A,O,Os,C, Ts,   New_Ts) :-
     make_before(C,A,Ts,   New_Ts).        /* make  before(C,A,Ts) */
```

Finally, the *ACHIEVE_2* operation is prototyped in a similar manner:

```
achieve2(PlanI, Gi, PlanO) :-
     get_comp(PlanI, partial_plan, pp,   PP),
     get_comp(PlanI, partial_plan, os,   Os),
     get_comp(PlanI, partial_plan, ts,   Ts),
     get_comp(PlanI, partial_plan, ps,   Ps),
     get_comp(PlanI, partial_plan, as,   As),
     element_of_set(Gi, Ps),                  /* pre-condition */

     dom_map(Os,   DomOs),                    /* post-condition: */
     newid(DomOs,   NewA),
     add_node(NewA,Ts,   Ts2),
     get_comp(PP,planning_problem, as,  ASpp),
     element_of_set(Action, ASpp),
     overwrite_map(Os,NewA,Action,   Os_new),

     achieve(Os_new,Ts2,NewA,Gi,   Ts3),
     for_all_elsIO(As, declobber(Os_new,NewA), Ts3,   Ts_new),

     get_comp(Action,action,pre,   PreA),
     make_goal_instances(NewA, PreA,   GIs),
     minus_set(Ps, [Gi],   Ps_new1),
     union_set(Ps_new1, GIs,   Ps_new2),
     union_set(As, [Gi],   As_new),
     put_comp(PlanI, partial_plan, os, Os_new,   Plan1),
     put_comp(Plan1, partial_plan, ts, Ts_new,   Plan2),
     put_comp(Plan2, partial_plan, ps, Ps_new2,  Plan3),
     put_comp(Plan3, partial_plan, as, As_new,   PlanO).
```

Exercise 13

Continue the prototyping exercise. You will need to implement procedures corresponding to the predicates *declobber* and *newid*, using the specifications in the last chapter, and also a top levelprocedure conforming to the algorithm in figure 6.2. Compare your answers with the implementation in the appendix.

5 4–Add all new partial plans generated by step 3 to the Store

7.3 Summary

Prototyping VDM specifications using Prolog involves, firstly, building a set of tools in Prolog to support the Set, Sequence, Map and Composite data structures. VDM operators are then translated into Prolog clauses, roughly by considering each predicate in an operator's pre- and post-condition as the post-condition of its corresponding Prolog procedure.

The advantages in using Prolog is that there is a correspondence between the Prolog's declarative semantics and the logic of an operator's conditions, and that Prolog's backtracking mechanism can be used to find the correct choice of existentially quantified variables. The chief disadvantages are in the insecurities of Prolog (it is not strongly typed, it has no data encapsulation mechanisms), and in its lack of functional evaluation, causing long winded implementations.

Additional Problems: Improvements to the Planner and the Prototype

We offer some suggestions for improving and expanding both the specification and the prototype, which the reader may like to take up as a project (these suggestions range from 'extended coursework' upwards..)

– After extending the planner so that it accepts parameterised actions (see the additional problems at the end of chapter 6) prototype your new specification.

– A hierarchical planner is one whose domain model must be specified at several levels of abstraction. The planner we have presented is 'flat', but it can be extended to plan within a hierarchy with the addition of a *refine* operation, which can be specified in much the same way as the 'achieve' operations. By consulting the A.I. planning literature, try to extend the design level specification to that of a hierarchical planner (for some help in this consult the PhD thesis in [Fox 90]).

– The planning algorithm involves three types of choice: which partial plan to choose from Store, which goal instance to achieve in that plan, and which way to achieve the goal instance. Letting these choices be random is very inefficient. Try to construct *heuristic* rules which influence these choices.

– Re-arrange the procedures in the achieve procedures to increase the planner's efficiency. For example, try to cut down the amount of backtracking Prolog has to perform to find an action to achieve a goal.

Bibliography

[Chapman 87] Chapman, D: 'Planning for Conjunctive Goals' *Artificial Intelligence*, vol. 32, pp. 333-377, July 1987.

[Clocksin and Mellish 84] Clocksin, W., and Mellish, C., 'Programming in Prolog', Second Edition, Springer Verlag, 1984.

[Fox 90] Fox, M., 'A Constructive Paradigm of Hierarchical Planning', PhD Thesis, The University of Hertfordshire, 1990.

[Henderson and Minkowitz 86] Henderson, P., and Minkowitz, C. J., 'Tme too methods of software design', ICL Technical Journal, vol 5, no 4, 1986.

[McCluskey 88] McCluskey, T. L., 'Deriving a Correct Logic Program from a Formal Specification of a Non-linear Planner' *Methodologies for Intelligent Systems*, Volume 3, pp351-358, North-Holland, 1988.

[Rich and Knight 90] Rich, E. and Knight, K: *Artificial Intelligence*, Second Edition, McGraw-Hill, 1990.

[Turner and McCluskey 94] Turner, J. and McCluskey, T. L., 'The Construction of Formal Specifications: an Introduction to the Model-Based and Algebraic Approaches', McGraw-Hill, 1994.