KRF

Knowledge Representation Framework Project
Department of Computer and Information Science, Linköping University,
and Unit for Scientific Information and Learning, KTH, Stockholm

Erik Sandewall

Reasoning about Actions and Action Planning

The present text is preliminary and fairly dense, and is intended as a lecture note in the literal sense that it accompanies an actual suite of two successive lectures in my course TDDC65 – Artificial Intelligence and Lisp.

1 Describing the Effects of Actions

1.1 Representation of Actions

The lecture note "Principles of Domain Modelling for Knowledge Representation" in the present course materials introduced the following representation using features.

```
[Hc .f .v]
[H .t .f .v]
```

Here, `[Hc .f .v]` expresses that the current value of the feature `.f` is `.v` and `[H .t .f .v]` expresses that the of the feature `.f` at time `.t` is `.v`. The same lecture note also introduced the use of relationship expressions for describing actions. As an extension of what was specified there, we introduce the following notation

```
[D .s .t .a]
```

where `[D .s .t .a]` expresses that the action `.a` is performed from time `.s` to time `.t`.

1.2 Characterization of Actions Using Prestate and Poststate Expressions

Based on the introduction of these predicates, we shall furthermore use the `Precond` property of verbs that form actions, and we use an effect definition in the `Performdef` property of the same verb. The contents of these properties is not restricted to using the operators shown above: it is also possible to use predicates such as `equal, member,` or `lt` (for 'less than') in the `Precond` property, for example.

In many situations it is anyway the case that a large part of the precondition information can be expressed using the `Hc` predicate, that is, using preconditions of the form "the value of feature ***f*** shall be ***v*** when the action starts." The present lecture note addresses methods for that type of situation. More specifically we use the following restriction on the applications being considered:

State-Transition Schema. The preconditions and effects of actions are considered to consist of two parts: a *static precondition* that only uses conditions that are unaffected by the actions, so that they are constant over time, and a *dynamic precondition* that only consists of one or more feature-value restrictions of the kind that can be expressed using the predicate `Hc`. Moreover, it is required that the *effects* of an action consist only of the assignment of new values to the features that occur in the dynamic precondition.

The static preconditions can be used for conditions that characterize the general environment of the microworld under consideration, for example the distance between two locations, or the existence or non-existence of a road between them.

We extend the notation for actions in a way that applies to actions that satisfy the state-transition schema. Two new properties are introduced, namely `Prestate` and `Poststate` where in both cases the value shall be a mapping i.e. a set of maplets, and where variables are allowed in both of them. The following is an example of a simple definition for the action `moveto` to be used as in

```
    [moveto :r the-robot :fr start-location :to destination]
```

The definition is as follows in the new framework:

```
@Precond
[existsroad .fr .to]

@Prestate
{[: (the position of .r) .fr]}

@Poststate
{[: (the position of .r) .to]}
```

In more complex cases it is of course possible to use a composite expression for the `Precond` property, using propositional connectives such as `and` and `or,` and it is possible to use more than one feature in the `Prestate` and `Poststate` properties. It is important however that *the same set of features shall be assigned in those two properties.*

The `Prestate` and `Poststate` expressions may contain variables, as this example has shown, which means that in any specific situation where these expressions are used, they must be *instantiated* (i.e. made specific) by inserting values for those variables according to the current context. These expressions should therefore normally contain variables corresponding to the parameters that are used by the verb being defined.

There is a straightforward transformation from this new notation to the one used before. Each component of the prestate can be translated to a literal using the predicate `Hc,` and the poststate can be translated to a set of assignment statements using the `sav` operator, connected together using the `soact` operator if there is more than one of them. For the example above the translation will be

```
@Precond
(and [existsroad .fr .to]
     [Hc (the position of .r) .fr] )

@Performdef
[sav .o position .to]
```

The new notation is therefore redundant in the sense that it can be reduced to the earlier, and more general notation, but it is useful since it provides the basis for standard methods for action planning.

However, although the `Poststate` expression can be translated to a script consisting of assignment statements, it is also possible to view the combination of the `Prestate` and `Poststate` expressions as a definition of a *state transition.* In this view, the current state of the environment (a microworld, for example) where the actions are performed is considered as a mapping from all the defined features to their corresponding values. Each combination of an object and a changeable attribute for that object defines a feature; a mapping that assigns a value to each of the features is a *state,* and the set of all possible states (with all the possible choices of values, but for the same set of features) is called the *state space.*

In this conceptual framework, an instantiated `Prestate` and `Poststate` defines a transformation in the state space: each state that contains the instantiated prestate as a subset, is mapped to a similar state where that subset is replaced by the instantiated poststate, and everything else is unchanged.

In this view, *action planning* can be seen as a search problem: given a starting state and an ending state, find a path in the search space that leads from starting state to ending state, and where each step along the path is characterized by one of the available actions. We shall return to this shortly, but first we shall introduce yet another representation for the effects of actions.

1.3 Characterization of Actions Using Effect Rules

The `D` predicate can be used for the log of actions in an episode, and for specifying scripts consisting of several actions. For some purposes it is convenient to represent the preconditions and effects of actions using propositions (logic formulas). The following is how the `moveto` action will then be described:

```
(imp (and [existsroad .fr .to]
          [D .s .t [moveto .o .fr .to]]
          [H .s (the position of .r) .fr] )
     [H .t (the position of .r) .to] )
```

A proposition of this kind is called an *effect rule* for the verb, which in this case is the verb `moveto.` Notice that we now use the predicate `H` rather than `Hc` since this proposition describes and relates the state of the world at two different timepoints, namely `.s` for the starting-time of the action, and `.t` for its ending-time. Some of the methods for action planning operate directly the representation using the `Prestate` and `Poststate` properties, and some use the logic-based representation that is shown in this example.

In some cases it is more convenient to separate the static precondition from the effect rule, so that the remaining effect rule is merely:

```
(imp (and [D .s .t [moveto .o .fr .to]]
          [H .s (the position of .r) .fr] )
     [H .t (the position of .r) .to] )
```

The choice between these variants of the notation depends on the particular planning method or reasoning method that is being used.

2 Prediction and Postdiction

If one is given a set of propositions that characterize the state of a microworld at a specific time (time 0, for example), a sequence of actions at successive timepoints after the starting time, and effect rules for the verbs in those actions, then it is in principle straightforward to deduce the state of the microworld at the successive timepoints where the actions start and end. An example of this will follow below.

The reverse operation of *postdiction* is also sometimes of interest: one is given the state of the microworld at a particular timepoint and a sequence of actions that were done *before* that time, and one is interested in deducing the state of the world at earlier timepoints. This is an important problem for *diagnosis* in cases where the system being diagnosed changes over time, and the problem is to figure out what earlier change (for example, a component breaking) may have led to the presently observed symptoms.

Some effect rules can be used both forward and backward in time, and in those cases postdiction is no more difficult than prediction, but if effect rules are written in such a way that several different starting states may lead to the same ending state, for a given action, then postdiction becomes nondeterministic and additional information is needed (for example, information about the state of the microworld at *some* of the earlier timepoints).

3 Planning Using the Situation Calculus

We observed above that action planning can be viewed as a straightforward search problem. This means that general graph-search algorithms *can* be used, but it does not exhaust the topic, since the action planning problem has additional characteristics that can be exploited in order to make the search practically feasible even for relatively large plans.

One of the most important action planning techniques is *regressive planning using the situation calculus.* Regressive planning means that one starts the search with the goal and tries to identify the *last* action in the plan, which will be the action after all the other actions and the one that finally achieves the goal. Then one considers the preconditions of that action and tries to identify the second-last action, and so on. The opposite method where one starts with the starting situation and works forward into the plan is called *progressive planning.*

3.1 The Situation Calculus

The situation calculus is based on the idea of *branching time.* Instead of considering a totally ordered set of timepoints, like the non-negative integers, it defines a tree of possible points called *situations,* starting with the initial situation which is usually written `s0` (pronounced ess-zero). Furthermore there is a *successor function* `succ` where `(succ .a .s)` is the successor of situation `.s` that is obtained by performing the action `.a` where of course `.s` can itself be formed using the successor function, recursively.

In this way, the plan of doing the actions M, N, and P in succession, starting in the initial situation, will be written as

```
(succ (succ (succ s0 M) N) P)
```

All the possible plans, in the sense of sequences of actions, can therefore be expressed as situations.

The successor function `succ` is related to the action performance predicate `D` through the following *execution axiom:*

```
[all .s .a [D .s (succ .s .a) .a]]
```

which means that in the tree of possible situations, the step from a node `.s` in the tree to one of its successors, `(succ .s .a)` , the action `.a` is performed. One should think of that tree as the tree of all possible futures from the given starting situation.

3.2 Prediction in the Situation Calculus

Let us consider a concrete example, the plan

```
[soact [moveto Robbie A B][moveto Robbie B C][moveto Robbie C D]]
```

where `Robbie` is one particular robot (or person), and where we now use the arguments without tags. In situation-calculus form this plan will be written

```
(succ (succ (succ s0 [moveto Robbie A B])
            [moveto Robbie B C] )
      [moveto Robbie C D] )
```

Now suppose we also know

```
[H s0 (the position of Robbie) A]
```

which says that the robot is at position `A` at the initial timepoint or situation `s0.` We should expect to be able to prove

```
[H (succ s0 [moveto Robbie A B]) (the position of Robbie) B]
```

In fact this comes out directly from the effect axiom for `moveto.` By instantiation it gives

```
(imp (and [D s0 (succ s0 [moveto Robbie A B]) [moveto Robbie A B]]
          [H s0 (the position of Robbie) A] )
     [H (succ s0 [moveto Robbie A B]) (the position of Robbie) B] )
```

The second argument of the conjunction (and-expression) has already been stated, and its first argument is obtained by instantiating the execution axiom. This gives the expected conclusion, and it is clear that similar conclusions can be drawn for succeeding situations along a path from the root `s0` of the situation tree.

This has shown that the situation-calculus representation of actions and plans could be used for prediction, albeit in a very simple case. We shall later discuss what are the restrictions on this approach: when is it applicable, and when is it not applicable. However, let us first consider how this approach can be used for planning, since this is after all our main topic.

3.3 An Example of Regressive Planning in the Situation Calculus

For this example, we use the larger version of the effect rule for the `moveto` verb, where there is also a requirement of a road between the two roadpoints:

```
(imp (and [existsroad .fr .to]
          [D .s .t [moveto .o .fr .to]]
          [H .s (the position of .r) .fr] )
     [H .t (the position of .r) .to] )
```

We consider the following facts as given:

```
   [existsroad A B]
   [existsroad B C]
   [existsroad C D]
   [H s0 (the position of Robbie) A]
```

The problem is the one of finding the plan whereby the position of Robbie is D at the end of the plan. It must be obtained in a systematic way given these facts, but it will of course be the same plan as we saw above.

The first step for doing this using regressive planning in the situation calculus is to rewrite all the given facts as clauses. This is trivial for the given atomic propositions. The effect law becomes

```
    {[-existsroad .fr .to]
     [-D .s .t [moveto .o .fr .to]]
     [-H .s (the position of .r) .fr]
     [H .t (the position of .r) .to] }
```

Finally, there is the desired goal of the deduction. In this case we don't just want to prove that there *exists* a plan, we also want to have it, but technically we begin by just making the statement that there exists a plan:

```
    [exist .u [H .u (the position of Robbie) D]]
```

This is negated, using the method that was described in the logic lecture note, obtaining

```
    [all .u [-H .u (the position of Robbie) D]]
```

which in resolution terms becomes the clause

```
    {[-H .u (the position of Robbie) D]}
```

At this point we introduce a technical trick: we add one more literal to this *goal clause,* so as to obtain

```
    {[-H .u (the position of Robbie) D]
     [Answer .u] }
```

This *answer literal* will be carried along in the deduction process and it will give us the desired answer at the end.

Now proceed as follows. The goal clause is resolved against the effect law, obtaining

```
    {[-existsroad .fr D]
     [-D .s .u [moveto Robbie .fr D]]
     [-H .s (the position of Robbie) .fr]
```

```
  [Answer .u] }
```

This is resolved against the execution axiom, obtaining

```
{[-existsroad .fr D]
 [-H .s (the position of Robbie) .fr]
 [Answer (succ .s [moveto Robbie .fr D])] }
```

One of our premises was `[existsroad C D]` so resolving against it we obtain a clause with one literal less, and more specific information:

```
{[-H .s (the position of Robbie) C]
 [Answer (succ .s [moveto Robbie C D])] }
```

However, this clause is entirely analogous to the original goal clause, which means that the same sequence of resolution against the effect law, the execution axiom, and a known `existsroad` fact can be performed. This obtains

```
{[-H .s (the position of Robbie) B]
 [Answer (succ (succ .s [moveto Robbie B C]) [moveto Robbie C D])] }
```

Just like in the first round it is the *situation variable* in the answer literal that gets to be substituted, which is why the situation representing the move from B to C appears as a subexpression of the situation for moving from C to D. Informally, what we have at this point is a plan saying that if Robbie is at position B at some time (= situation), then there is plan consisting of first going from B to C, and then going from C to D that will take Robbie to D.

With the given premises we must however perform the same steps one time more, obtaining

```
{[-H .s (the position of Robbie) A]
 [Answer (succ (succ (succ .s [moveto Robbie A B])
                     [moveto Robbie B C] )
               [moveto Robbie C D] )]}
```

This clause can finally be resolved against

```
[H s0 (the position of Robbie) A]
```

obtaining

```
{[Answer (succ (succ (succ s0 [moveto Robbie A B])
                     [moveto Robbie B C] )
               [moveto Robbie C D] )]}
```

At this point we can not come any further, since there is no clause containing a negated literal with the predicate `Answer.` Now consider what has happened. If we *had not* introduced the answer predicate then at this point we would have had the empty clause, representing a contradiction, which means that we have proved that

```
(not [exist .u [H .u (the position of Robbie) D]])
```

leads to a contradiction, which means that we have proved

```
[exist .u [H .u (the position of Robbie) D]]
```

Moreover, as a result of attaching the answer literal that does not do anything in the proof besides following along and collecting substitutions, we have obtained a *constructive proof:* not only knowing that there exists a plan, but we have obtained one such plan. We have seen above that one can make a proof that the plan leads to the desired effect, but it is not necessary to carry out that proof; the correctness of the plan is guaranteed by the way it was constructed.

This example shows the basic idea in regressive planning using the situation calculus, although the `Answer` predicate is not used in actual systems today. It was used historically and it is a nice way of showing the idea, but in practice one does the proof without this device, one keeps track of the substitutions that are made in the proof, and one extracts the resulting plan from those substitutions.

3.4 More General Cases of Situation Calculus

We have used a particularly simple example in order to make the details manageable in the written text, but practical uses of this method will of course be much larger, which is possible when a computer does the manipulation of the formulas. There can also be more than one action, and more than one way of choosing the arguments of an action in each step, which means that the process of finding a plan from the initially given goal is a genuine search process. Actions can have more than one effect, i.e. they can change the value of more than one feature, and they can be conditional so that there are different effects depending on some aspect of the action's starting state.

Other variants on this theme can not be represented, or require additional devices in order to be represented. Plans involving concurrent actions can not be represented in the basic notation, and extensions to the notation that may seem to allow concurrent actions will introduce additional difficulties. Nondeterministic actions that lead to a random choice between different, alternative assignments to the same feature can also not be represented in the basic notation.

Finally, since the situation calculus is based on a notion of successive, discrete actions that are characterized merely in terms of their starting state and ending state, it is not very suitable for applications requiring the use of continous time and continous change within the duration of an action.

It should be emphasized however that several of these restrictions of the situation calculus are shared with other approaches that will be discussed later. Also, it is not the case that one is always interested in the most general available method: a method that is based on certain restrictions on the problem being addressed may also be more efficient for problems that satisfy those restrictions.

4 The Frame Problem

The *frame problem* is a classical problem in the logical representation of actions and their effects. It applies in one way or another to virtually all logic-based methods for reasoning about actions and change, but we shall describe it here in terms of the situation calculus.

4.1 An Example

Let us consider again the example from the previous section, but now we add to the goal that when Robbie arrives to the destination then he (or she, or it) shall be carrying a food supply for use there. We extend the known facts with one more of them:

```
[H s0 (the carries of Robbie) foodsupply]
```

and we modify the goal statement so as to be

```
[exist .u (and [H .u (the position of Robbie) D]
               [H .u (the carries of Robbie) foodsupply] )]
```

After the negation, conversion to clause form, and addition of the answer literal this becomes

```
{[-H .u (the position of Robbie) D]
 [-H .u (the carries of Robbie) foodsupply]
 [Answer .u] }
```

Exercise: Please verify that this clause is in fact obtained. Since the revised goal statement is a conjunction (and-expression), why do we obtain one single clause here containing effectively the disjunction between the two literals using `H` ?

Finally, in order not to have to do all three deduction steps, let us assume in addition to the previous that there is a direct road from A to D:

```
[existsroad A D]
```

At the end of the first round of three resolutions we shall now have the following clause:

```
{[-H .s (the position of Robbie) A]
 [-H (succ .s [moveto Robbie A D]) (the carries of Robbie) foodsupply]
 [Answer (succ .s [moveto Robbie A D])] }
```

Resolving against the known initial position of Robbie we obtain, similar to before:

```
{[-H (succ s0 [moveto Robbie A D]) (the carries of Robbie) foodsupply]
 [Answer (succ s0 [moveto Robbie A D])] }
```

We have now identified the correct solution in the argument of the answer predicate, but there is still a restriction that needs to be verified, namely, the requirement that Robbie shall be carrying the food supply at the end of the plan. We can not resolve the first literal in this clause against the known fact that Robbie carries the food supply in situation `s0` since `s0` is a constant and not a variable, from the point of view of the logic.

The problem is therefore: since we know that Robbie carries the food supply in situation `s0,` how can we obtain the relatively obvious conclusion that he carries the food supply after having gone to location D, but doing the conclusion in a systematic way? This is referred to as the *persistence* of feature values, and the problem of characterizing persistence in a good way is referred to as the *frame problem* in reasoning about actions and change. We must do it in a way that will fit into the formal machinery, and we should also do it in a way that allows for restrictions on the persistence.

For example, it may be that some burdens are too heavy to carry, or that some burdens are not allowed on certain means of transport, and so forth.

4.2 Simple Solutions

The simplest way of doing this is using *forward frame axioms.* The following is an example of such an axiom.

```
(imp [H .s (the carries of .r) .v]
     [H (succ .s [moveto .r .p]) (the carries of .r) .v] )
```

It says that if `.r` carries `.v` at the beginning of a `moveto` action that it does itself, then it will also carry it at the end of that action. In fact this axiom can be written more generally as

```
(imp [H .s (the carries of .r) .v]
     [H (succ .s [moveto .q .p]) (the carries of .r) .v] )
```

saying that if `.r` carries `.v` at the beginning of *someone else's* moving action, it will still carry it at the end of that action.

We can see at once that there will be a very large number of such forward frame axioms, and this is not very convenient. Another possibility is to use *reverse frame axioms,* such as the following schematic one:

```
(imp (and [H .s (the carries of .r) .v]
          [H (succ .s .a) (the carries of .r) .w]
          [/= .v .w] )
     (or [= .a [drop .r .v]]
         [exist .q [= .a [swap .r .q .v .w]]]
               ... ))
```

where the three dots indicate space for adding more alternatives. What this axiom says is that if the value of the `carries` attribute for a particular `.r` is different before and after an action, then that action *must be* one of those that are listed in the second half of the axiom: either it is an action where `.r` drops `.v`, or it is an action where `.r` makes a swap with another agent `.q` whereby they exchange what they are carrying, and so forth. The reverse frame axiom must therefore enumerate *all the possible alternatives* for how the value of that particular feature may get to be changed.

4.3 The Occlusion Predicate

More advanced solutions to the frame problem and in general to the representation of actions and change are based on two essential devices: the use of *nonmonotonicity* in the logic being used, and the use of the *occlusion predicate.* Nonmonotonicity is a complex topic and we shall only touch on it superficially here, and we start with occlusion.

Occlusion is introduced for several reasons: it provides a systematic representation for nondeterministic actions, it also provides a solution for actions with extended duration in time where changes occur successively during the action, and finally it is the basis for some solutions to the frame problem. The basic idea is to introduce a predicate `X` (pronounced as 'occludes') as in, for example

```
[imp [D .s .t [throw-dice .d]]
     (and [X .s .t (the face of .d)]
          (or [H .t (the face of .d) 1]
              [H .t (the face of .d) 2] ...
              [H .t (the face of .d) 6] ))]
```

This effect rule states that after the dice has been thrown, its face will show one of the numbers from 1 to 6, and furthermore the value of that feature is occluded from time `.s` to `.t` which is used by the following axiom, called the *PMON axiom:*

```
(imp (and [H .s .f .v]
          [H (succ .s .a) .f .w]
          [/= .v .w] )
     [X .s (succ .s .a) .f] )
```

This axiom is similar to the reverse frame axioms but it is entirely general, whereas when reverse frame axioms are used one has to write one of them for each verb. Here it is left to the separate effect rules to specify which features are being occluded, and the PMON axiom simply says that if a feature changes its value from one situation to its successor then it must have been occluded during that interval.

The treatment of the occlusion predicate has a peculiar property in the sense that is is *assumed not to hold unless it is proved to hold.* This is a kind of default rule, and it is a natural one from a computational point of view, but a nonstandard one from the point of view of formal logic where the truth of a proposition is usually considered to be *unknown* unless itself or its negation has been stated or proved explicitly. It is this peculiarity of the occlusion predicate that leads to the notion of nonmonotonicity in the logic.

Nonmonotonicity is important in the representation of knowledge. There are also other ways of introducing and using nonmonotonicity for reasoning about actions and their effects, and there are additional representation topics where it is also used. More will be said about this later in the course.

5 Partial-Order Planning

We proceed now to the method of *partial-order planning* which is one of the methods that make direct use of the state-space formulation of effect rules for actions. This method is often described as a nondeterministic algorithm, but it can also be described as a satisfiability problem, and that is the formulation that we shall use here since it provides a more compact description and since it highlights the relationship between the methods involved. Satisfiability problems are introduced in the lecture note 'Computational Engines in Artificial Intelligence' is the present set of course materials.

5.1 Problem Specification and Setup

A specific partial-order planning problem is defined by a set of features, a set of actions whose `Prestate` and `Poststate` properties are partial mappings from features to corresponding values, a *starting state* `ss` that is simply a mapping from features to values, and a *goal state* `gs` that is likewise. The starting state must be a total mapping, i.e. it must assign values to all the features. The goal state is given as a partial mapping, which assigns values to some of the features in the starting state. The requirement is to find a plan that obtains these assignments of feature values. It is not required that the other features values shall be unchanged; it is permitted for the plan to change some of them as well. If one should wish to require that some of feature-value assignments in the starting state are to remain unchanged then one shall simply include them in the goal state as well.

As we showed in an earlier section it is natural to consider actions that are formed using an action verb and its arguments, and where each verb is associated with `Prestate` and `Poststate` properties that are expressions containing variables. For each action using the verb in question, one substitutes the actual arguments in the action expression in the place of those variables. This obtains variable-free expressions for the prestate and poststate of the action in question.

However, for the present section we shall disregard the verbs and the substitutions, and simply assume that there is a *set of actions,* without caring about the internal structure of each action in terms of verbs, arguments, or the like. If a microworld contains many objects that can occur as arguments in action expressions then the set of actions will be quite large, but the description of the planning methods becomes simpler in this way. [1]

With this framework, we introduce the functions `prest` and `post` such that `(prest .a)` and `(post .a)` are sets of feature-value assignments, called *conditions,* for each action `.a` . Conditions are written as maplets in the KRE notation, for example

```
[: (the hunger of Groucho) big]
```

A mapping is then a set of maplets, as usual.

Notice that the term "condition" is here used in a slightly different way compare to how the term was used in earlier sections where conditions where entire logic formulas.

For example, the verb `moveto` that was used above will be characterized by

```
[= (prest [moveto .r .fr .to]) = {[: (the position of .r) .fr]}
[= (post [moveto .r .fr .to]) = {[: (the position of .r) .to]}
```

If `.s` is a feature state and `.a` is an action, then we write

```
(apply .a .s)
```

for the new feature state that is obtained by performing the action in the given feature state. It is defined if and only if `(prest .a)` is a subset of `.s` and in this case it is obtained as the union of `.s - (prest .a)` and `(post .a)` – that is, remove from `.s` all the conditions in the prestate of the action, and then add all the conditions in its poststate. This definition is in line with the persistence assumption that the state of the microworld under consideration will only change as the direct result of actions that are performed there.

[1]Implemented systems will however usually need to handle the arguments of action expressions separately, in particular because the choice of those arguments is often restricted by static preconditions of the actions. Compare Subsection 8.2.

As a part of the setup of a planning problem and before starting the main planning process, one introduces two artificial actions, called `start` and `finish,` with the following definitions.

```
[= (prest start) 0]
[= (post start) ss]
[= (prest finish) gs]
[= (post finish) 0]
```

where `0` represents the empty set, and where `ss` and `gs` are the starting state and goal state, like before. These two actions are added to the actions that have been provided by the application as such. Notice that technically speaking, `start` and `finish` are actions and not verbs, which explains why they will never occur surrounded by square brackets.

5.2 Planning Process

The partial-order planning method constructs plans by adding actions successively to a current *partial plan.* The current plan is initialized as the empty plan, not containing any actions at all except `start` and `finish,` and actions can be added to it at any point during the planning process. It is therefore possible to do both progressive and regressive planning within its framework, but it is also possible to being by selecting some actions in the middle of the forthcoming plan. The process stops when a satisfactory non-partial plan has been found, or when the process fails for one reason or another.

Since it is possible that the same action instance will occur several times in the same plan one introduces a function `ai` (for “action instance”) that takes an action and a number as arguments, for example

```
(ai [moveto Groucho stable1 stable2] 1)
```

The numbering is local to each action. We shall use an abbreviation where such a term can also be written as

```
[moveto Groucho stable1 stable2]#1
```

The functions `prest` and `post` will apply to action instances in the obvious way, so that

```
[= (prest (ai .a .n)) (prest .a)]
```

and similarly for `post.` A partial plan contains a set of such action instances, and a partial order on them that specifies that some action instances must be done before others, for example

```
[before [moveto Groucho stable1 stable2]#1 [shower Groucho]#3]
```

The partial-order planning process (POP) operates by successively adding action instances and literals using `before` to its current partial plan. At each point in this process there are several alternative additions that can be made, which means that this is a search process and that POP is a nondeterministic algorithm.

The action number is omitted on `start` and `finish` since there will never be more than one occurrence of these. (Technically, the same symbol is used for both the action and the action instance, according to context).

The essential requirement on a correct plan is that the preconditions of each action instance must be satisfied when the plan is performed, that is, when the actions are performed in some order that is permitted by the `before` literals in the plan. The preconditions of a later action instance must then have been produced by the postconditions of preceding action instances, unless they were already present in the starting state. However, they need not be a result of the *immediately* preceding action instance, and in fact it is not always the case that there is merely one action instance that can be the preceding one. The method must therefore be able to administrate the possibility that an earlier action instance has an effect that is used by one or more action instances that occur much later. However, this is only permitted if there is no *other* action instance that intervenes and undoes the effect of the first action instance.

In order to manage this issue one introduces an additional predicate which is written `achieves` and that has three arguments, for example

```
[achieves [moveto Robbie A B]#2
          [: (the position of Robbie) B]
          [moveto Robbie B C]#1 ]
```

This statement expressed that the maplet in the middle argument is part of the poststate of the first argument and is part of the prestate of the third argument, and that it is part of the plan that the first-argument action instance shall enable this precondition for the third-argument action instance. Propositions of this kind are also made part of the partial plan as it is being built up, but they are only an auxiliary construct so they are not used in the final plan.

Now we can express the requirements on a correct partial plan for the given planning task. Each partial plan consists of two things:

- A set of action instances.
- A set of literals using the predicates `before` and `achieves,` where the arguments are chosen from the action instances in the first component.

Each step in the planning process proceeds from such a partial plan to an extended one. A correct partial plan must satisfy the following requirements.

- The `before` relationship must not be circular.
- The middle argument of every `achieves` literal must be a member of the poststate of the action instance in the first argument, and of the prestate of the action instance of the third argument.
- Every condition in the prestate of every action instance in the plan must occur as the second argument of an `achieves` literal that has that action as its third argument.
- The first argument in any `achieves` literal must be stated as `before` the third argument.
- For every `achieves` literal, if there is some other action instance in the plan whose poststate contains an assignment of a different value to the maplet in the literal's middle argument, then that action instance

must either precede the first argument, or succeed the third argument in the literal in question.

- Each added action instance must be stated to occur after `start` and before `finish`.

These conditions can be precisely expressed in logic, as follows (universal quantification understood and omitted):

```
1.  (imp (and [before .a .b][before .b .c]) [before .a .c])
2.  [-before .a .a]
3.  (imp [achieves .b .m .a]
         (and [member .m (post .b)][member .m (prest .a)]) )
4.  [all .m .a (imp [member .m (prest .a)]
                    [exist .b [achieves .b .m .a]] )]
5.  (imp [achieves .a .m .b][before .a .b])
6.  (imp (and [achieves .a [: .p .v] .b]
              [/= .v .w]
              [member [: .p .w] (post .c)] )
         (or [before .c .a][before .b .c][= .c .a][= .c .b]) )
7.  [all .a (imp [/= .a start] [before start .a])]
8.  [all .a (imp [/= .a finish] [before .a finish])]
```

The partial-order planning method consists simply of successively adding action instances and literals to the current partial plan until all of these conditions are satisfied. In this sense it is a satisfaction problem.

Some of these restrictions can be considered as “automatic” in the sense that they should always be applied immediately when some addition has been made to the current partial plan. This applies for Restrictions 7 and 8 which are applied each time an action instance has been added, and for Restriction 5 which is applied each time an `achieves` literal has been added. In principle it also applies for restriction 1, which expresses the transitivity of the `before` relation, although in that case one will probably not draw the conclusion explicitly. The transitivity is used together with Restriction 2 for guaranteeing that the `before` relation is not circular.

Restriction 3 has another character: it is a real restriction, since it expresses that an `achieves` literal *may only* be added if its middle argument is a post-condition of the first argument and a precondition of the third argument. – We shall return to the roles of Restrictions 4 and 6 after the following example.

5.3 An Example Using Regression

Let us now do the same planning example using partial-order planning as we did above using the situation-calculus method. We shall do the example twice, using different search strategies, in order to demonstrate that partial-order planning is not restricted to plain regressive and plain progressive planning. However, first we show how regressive planning can be done in the partial-order planning framework.

The given problem of getting from A to D is set up as follows, using the rules that were just described. The feature `(the position of Robbie)` will be abbreviated as `por`. The initial partial plan is as follows.

```
{start, finish}
start before finish
```

Please recall that a partial plan consists of a set of objects and a set of literals. We write the set of objects on the first one or a few lines of a partial plan, and the literals on succeeding lines with one literal per line.

The actions `start` and `finish` and the verb `moveto` are defined as follows, thereby also encoding the start and end conditions of the planning problem.

```
(prest start) = 0
(post start) = {[: por A]}
(prest finish) = {[: por D]}
(post finish) = 0
(prest [moveto .r .x .y]) = {[: (the position of .r) .x]}
(post [moveto .r .x .y]) = {[: (the position of .r) .y]}
```

The definitions for `moveto` can be specialized as follows, using the abbreviation.

```
(prest [moveto Robbie .x .y] = {[: por .x]}
(prest [moveto Robbie .x .y] = {[: por .y]}
```

The initial partial plan does not satisfy Restriction 4, if one selects `.m` as `finish`, since there is not any `achieves` literal in this partial plan that has the single maplet in `(prest finish)` as its middle argument. This can be remedied by adding an instance of `[moveto Robbie C D]` to the initial partial plan and stating that it achieves the precondition in question. This obtains

```
{start, finish, [moveto Robbie C D]#1}
[achieves [moveto Robbie C D]#1 [: por D] finish]
start before finish
start before [moveto Robbie C D]#1
[moveto Robbie C D]#1 before finish
```

Now Restriction 4 is satisfied for `finish` but it is not satisfied for `[moveto Robbie C D]` so the process has to be repeated. This is done twice and one then obtains

```
{start, finish, [moveto Robbie C D]#1, [moveto Robbie B C]#1,
   [moveto Robbie A B]#1}
[achieves [moveto Robbie C D]#1 [: por D] finish]
[achieves [moveto Robbie B C]#1 [: por C] [moveto Robbie C D]#1]
[achieves [moveto Robbie A B]#1 [: por B] [moveto Robbie B C]#1]
[moveto Robbie B C]#1 before [moveto Robbie C D]#1
[moveto Robbie A B]#1 before [moveto Robbie B C]#1
start before finish
start before [moveto Robbie C D]#1
start before [moveto Robbie B C]#1
start before [moveto Robbie A B]#1
[moveto Robbie C D]#1 before finish
[moveto Robbie B C]#1 before finish
[moveto Robbie A B]#1 before finish
```

At this point the regression has arrived to the given starting state, and one can add

```
[achieves start [: por A] [moveto Robbie A B]#1]
```

With that final addition we are ready to check all the requirements. Restriction 4 is now satisfied since it is satisfied for `finish,` each of the `move` actions has it satisfied from its predecessor, and the `start` action has an empty set of preconditions. Requirement 1 is trivially obtained by adding literals for the transitivity of `before,` for example

```
[moveto Robbie A B]#1 before [moveto Robbie C D]#1
```

There is no need to write out all of these transitivity conclusions; the important thing is that restrictions 1 and 2 together specify that there is no circularity with respect to `before,` and this is the case in our solution.

Restriction 5 says that if x achieves y then x must be before y, and this is satisfied in the final partial plan.

As for Restriction 6, consider it for the case where `.a` is the move from B to C and `.b` in that condition is the move from C to D. Then `.p` in that condition will be `por,` and the condition says that any other action – besides `.a` and `.b` – that assigns `por` in its postcondition must either come before `.a` or come after `.b.` In the present case this applies to three action instances, since all the action instances that are included in the first component of the partial plan must be considered; thus it applies to `start` and `finish` and `[moveto Robbie A B]` . It is seen at once that there are applicable `before` literals for all of these, so Restrition 6 is satisfied for this particular choice of `.a` and `.b.` The other choices for the various variables are verified in the same way.

From this one can also see how the general machinery works. Restriction 4 and Restriction 6 are the most interesting ones. Restriction 4 enforces that all preconditions of all considered actions are satisfied, and the construction of using the "action" `finish` has the effect that the various goal conditions are obtained in the same way. The construction of using the "action" `start` likewise establishes the starting state as being available for satisfying preconditions.

Restriction 6, on the other hand, is the one that keeps actions from disturbing the connection from postcondition to precondition. Suppose, in our example, that the first move action had had some additional effects, and that these were needed by the third move action but not by the second one. This would have been in order, unless the second action had had some additional effect that changed the feature that the first action had set and the third action had needed. Restriction 6 has the effect of precluding partial plans having that structure.

5.4 The Same Example, Using Middle-Out Planning

Now consider the same example, but assuming a context where there was additional information to the effect that any reasonable plan ought to contain the action `[moveto B C]` without saying anything more. This is typical of many practical situations where, for a given problem, there is good advise about one or a few major actions that should be taken, but the details remain. For example, for having lunch, the main action may be to go the cafeteria, but the details of getting there may have to be worked out: take on a coat? bring an umbrella? put one's laptop in a locker? and so on.

In this case there is the same initial partial plan as above, namely

```
{start, finish}
start before finish
```

Adding the suggested action obtains

```
{start, finish, [moveto Robbie B C]#1}
start before finish
start before [moveto Robbie B C]#1
[moveto Robbie B C]#1 before finish
```

There is still no possibility of adding an `achieves` literal, but this partial plan contains two "open preconditions," that is, preconditions that are not included in an `achieves` literal, namely the precondition of `finish` and of `[moveto Robbie B C]` . This means that the original planning problem has been *decomposed* into two separate planning problems that are (hopefully) simpler than the given one.

The strategy of assuming an action in the middle of a forthcoming plan is called *middle-out planning,* as an alternative to progressive planning, which is also sometimes called *forward planning,* and to regressive planning. The efficiency of middle-out planning depends entirely on the quality of the advise for the initial choice of action. If good advise of this kind is available then it can reduce the search very much, but if no such advise is available and the initial middle action has to be chosen at random then it does not have any clear advantage.

Lab 2b during course TDDC65 used a notation for specifying precondition repair methods in the course of action execution. A similar technique may be used for specifying the advise that will take one from the occurrence of a particular goal condition, or combination of goal conditions, to a proposed initial middle action in a plan for achieving the goal.

6 Planning Graphs and the Graphplan Method

6.1 Layered plans and execution states

We proceed now to the use of *planning graphs* and the first method using them, namely, the *Graphplan* method. This approach is based on the use of *layered plans* which are organized as follows. Each layered plan consists of a number of *plan steps* that are identified by numbers 1, 2, etc., and a non-empty set of *action instances* in each step. The number of the plan step is used as the postfix for the action instance, so plan step 2 could for example contain the following two action instances

```
[moveto Robbie B C]#2
[moveto Ronnie D A]#2
```

These methods use the state-space formulation of the effects of actions, so each action instance has a prestate and a poststate. A set of action instances has a *joint prestate* and *joint poststate;* these are simply the union of the prestates and the poststates, respectively. One must assume that the actions in the set are not conflicting; more about this later.

Consider now a given starting state and a given layered plan. A sequence of *execution states* is defined as follows. The starting state is the execution state for step 0. If the joint prestate of the plan for step *k* is a subset of the execution state for step *k-1* then the execution state for step *k* is obtained by subtracting the joint prestate and then adding the joint poststate for step *k.* If the joint prestate for step *k* does not have this property then there is no execution state for step *k.* A layered plan of length *n* is said to be *executable* from a given starting state if the execution state exists for step *n.*

The idea with a layered plan is that all action instances in one plan step must be executed before the action instances in later steps, but the order of execution within the plan step is arbitrary. The action instance within the plan step may be performed concurrently, if the computational and physical resources permit so, or they may be performed sequentially in any order.

6.2 Planning graphs

A planning graph is an auxiliary structure that is used in the planning process. Graphplan and other, similar methods operate by first constructing a planning graph, and then trying to identify an executable plan within the planning graph. More precisely, the method is to first construct a planning graph of length 1 and look for an executable layered plan of length 1, if no such plan is found then extend the planning graph to length 2 and look for executable layered plans of length 2, and so forth.

The planning graph consists of *action layers* and *state layers* which are similar to plan steps and execution states, respectively, except that action layers may contain many incompatible action instances, and state layers may contain incompatible value assignments to features. For example, given the `moveto` verb that we have used above, and assuming that there are roads both from A to B and from A to D, action layer number 1 will contain both the following action instances:

```
[moveto Robbie A B]#1
[moveto Robbie A D]#1
```

and state layer number 1 will contain all of the following feature-value assignments, which we shall call conditions here:

```
[: (the position of Robbie) A]
[: (the position of Robbie) B]
[: (the position of Robbie) D]
```

given that the starting state contains the first one of those. In general, the action layer number *k* will consist of all the action instances whose prestate is a subset of the state layer number *k-1,* that is, all the actions that are possible to execute from it, and the state layer number *k* will be the *union* of the state layer number *k* and the joint poststate of all those actions.

This means that the state layer is highly inconsistent, in normal cases, since it contains all the possible ways of performing all the possible actions. If you consider the sequence of state layer number 1, 2, and so on, it is clear that there is some number *N* such that no more additions can be made by this procedure, that is, all state layers with number *N* or greater will be equal. This is called the *saturated* state layer. Two observations can be made at once:

- If the given goal state is not a subset of the saturated state layer then the given planning problem does not have any solution.
- If the given goal state is a subset of state layer number k but not of any earlier state layer, then any plan for achieving the goal must have at least length k.

6.3 Annotation of the Planning Graph

The problem with the sequence of action layers and state layers (i.e. the planning graph) is of course that it is inconsistent. However, by adding annotations (markup information) that characterizes the inconsistencies, it becomes possible to use it as the basis for finding a plan. The annotation uses two predicates, with arguments as in the following examples:

```
[mutex [moveto Robbie A B]#1 [moveto Robbie A D]#1]
[incomp 1 [: (the position of Robbie) A]
          [: (the position of Robbie) D]]
```

The predicate `mutex` takes two arguments which must both be action instances for the same plan step, and expresses that these two actions are *mutually exclusive:* it is not possible to use both during that timestep. The predicate `incomp` takes three arguments where the first one must be a plan-step number, and the other two must be conditions, not necessarily for the same feature. It expresses that these conditions are incompatible; it is not possible to have both at the same time in that plan step.

In addition we use the auxiliary predicate `dinc` to express that two conditions are *directly incompatible,* i.e. they assign different values to the same feature, so that

```
(equiv [dinc [: .f .v][: .f .u]] [/= .v .u])
[-dinc [: .f .v][: .g .u]]
```

Positive literals using `mutex` and `incomp` are added to the planning graph according to the following rules.

```
1.  (imp [dinc .f .g] [incomp .t .f .g])
2.  (imp (and [member .u (post .a)] [member .v (post .b)]
              [dinc .u .v] )
         [mutex (ai .a .k) (ai .b .k)] )
3.  (imp (and [member .u (prest .a)] [member .v (prest .b)]
              [incomp .u .v .k] )
         [mutex (ai .a .k) (ai .b .k)] )
4.  (imp (and [member .u (prest .a)] [member .v (post .b)]
              [dinc .u .v] )
         [mutex (ai .a .k) (ai .b .k)] )
5.  [forall .u .v .k
       (imp [all .a .b
                 (imp (and [exist .x [= .a (ai .x .k)]]
                           [member .u (post .a)]
                           [member .v (post .b)] )
                      [mutex .a .b] )]
            [incomp .k .u .v] )
```

These rules shall be read as follows. Rule 1 says that if two directly incompatible conditions occur in a particular layer, then they are incompatible there. Rule 5 says, in addition, that if you have two conditions in a layer, and you choose one action instance with that layer number whose poststate includes the first condition, and you also choose another action instance in that layer whose poststate includes the other condition, and if those two action instances exclude each other *for all available choices of the first one and the second one,* and in all combinations of those choices, then the two conditions are also incompatible there, meaning that they can not both hold there.

Furthermore, rule number 2 says that if you have two actions whose poststates contain incompatible conditions, then those actions are mutually exclusive in that layer. Rules number 3 and 4 say similarly that two actions are mutually exclusive if the prestates of the two actions are directly incompatible, or if the prestate of one is directly incompatible with the poststate of the other.

In fact, rule number 3 goes further, by stating that if two actions instances are such that there is a combination of one condition from each of their prestates and where these two conditions are incompatible in the layer preceding the layer of these two actions (that is, in the layer where the preconditions of the actions must be satisfied), then these two action instances are also mutually exclusive. Notice that they do not have to be *directly* incompatible; the rule applies also if the incompatibility is indirect.

In this way one can assign conflict information to the actions and to the conditions in each layer. One thing has to be added, however: suppose you have two actions that can not be done at the same time, but it is possible to do them one after the other, then with the definition of Rule 5 it would seem that neither of their effects is achievable, since in every layer both will be present, and they will block each other. For this reason one adds one more action to the set of actions, namely the `no-op` action that has the empty set as prestate and as poststate. The use of this action will prevent the undesirable effect in Rule 5.

6.4 Plan Extraction

As described above, the overall procedure for the Graphplan method is to first construct the planning graph for length 1 and try to extract a correct plan from it; if this does not succeed then it tries with a planning graph for length 2, and so forth. Suppose now that no executable plan was found from the planning graph of length *k-1* and one has constructed and annotated the planning graph of length *k.* The task is then to extract an executable plan from it.

This is done by regression, that is, starting from the desired goal, which is expressed as a partial state. If the desired goal is not a subset of the current state layer (the layer for number *k*) then no plan can be found. Moreover, if the goal consists of more than one condition and two of the conditions in the goal are incompatible in the present layer, according to the `incomp` predicate, then it is also not possible to find a plan. In both of these cases it is necessary to add one more layer to the plan graph.

If all goal conditions are pairwise compatible, then one obtains a set of action instances in the current action layer (the action layer for number *k*) that contributes to the goal set, i.e. each of the action instances in that set must have at least one condition in its poststate that is a member of the goal state. This set must not contain two action instances that exclude each other, according to the `mutex` predicate.

For each possible choice of such a set of action instances for layer *k,* the method constructs a revised goal set to be used for layer *k-1,* namely, by removing those conditions that were achieved by the poststates of the selected actions, and combining the unachieved goal conditions with the joint preconditions of the actions in the chosen set of action instances. This revised goal set is now applied to layer *k-1* in the plan graph. Notice that layer *k-1* has previously been worked on using the original goal set, and now one returns to it using a revised set of conditions.

This process is repeated recursively until one gets back to level 0 where, if one is successful, the revised goal set will be a subset of the given starting state for the planning problem.

The plan extraction process is therefore just regressive planning using the state-space representation of actions, but with two important modifications:

- The availability of the plan graph makes it possible to restrict the choices of possible actions in each regression step.
- The use of layered plans makes it possible to treat two or more actions as an unordered set in those cases where the order between these actions does not actually matter.

As an additional efficiency consideration one may impose the following restrictions on the choice of the set of action instances for a particular layer. The set can be required to be *maximal* in the sense that it is not possible to add one more action and thereby increase the number of conditions in the goal set that are obtained. It can also be required to be *non-redundant* in the sense that it is not possible to remove one action and still obtain the same set of conditions in the goal set. (This may occur if several actions can produce the same condition). The use of these restrictions will not lead to the loss of any solution to the given problem, and it will tend to increase the efficiency of the search by reducing the number of action combinations that do essentially the same thing.

In summary, the Graphplan method is characterized by a progressive phase, where one constructs the plan graph by proceeding forward from the given starting state, and a regressive phase that consists of regressive planning where the search is strictly constrained by the plan graph.

7 General Considerations for State-Space Planning Methods

7.1 Logic and Specification vs. Algorithms and Datastructures

We have described two planning methods that are based on state-space characterization of actions in terms of their prestate and poststate. Both methods have been described as constraint problems, where the solution is obtained by successively adding literals to a working set in such a way that certain given constraints are satisfied. The case of having rules that directly specify what literals need to be added in specific situations is a special case of having constraints.

Our descriptions of these methods should be understood as high-level specifications and not as detailed descriptions of possible implementations. In practice it would not be efficient to represent partial plans or plan-graph layers by enumerating a set of literals, and one will instead use an implementation in terms of datastructures and pointers. There are also additional considerations that have been omitted here and that must be taken into account in order that the methods will work satisfactorily in all cases.

7.2 Static Preconditions

We have described Partial-Order Planning and Graphplan as methods that only take dynamic preconditions of actions into account. Actual applications usually require the use of static preconditions as well. The introduction of static preconditions is quite straightforward on the formal level since both the methods themselves and the static preconditions can be expressed in logic, and since the methods are viewed as satisfaction problems. The static preconditions will simply be additional restrictions for these satisfaction problems.

Notice, however, that there is a contradiction between the requirement of using static preconditions, and the desire to use efficient, conventional algorithmic implementations that was mentioned in the previous subsection. The implementation issue suggests that the explicit use of logic formulas should be replaced by the use of traditional data structures, but if applications involve static preconditions that differ from one application to another, then it becomes necessary to have an implementation that allows the use of logic formulas in addition to the optimized data structures, and it also becomes necessary to interface and connect those two representations. This is not impossible, of course, but it may account for additional complexity in the resulting software systems.

7.3 Accomodable Extensions

We have described planning methods that apply for problems that conform to the state-transition schema that was definedin Subsection 1.1. Some moderate generalizations of that schema can also be accomodated in the same methods. In particular, there is the case of *conditional actions* whose effects depend on some conditions in the starting state. Formally such actions can be represented by extending the state-transition schema so that each action is characterized by a *set of* prestate-poststate pairs, namely, one pair for each branch of the action. Furthermore, problems using such conditional actions can be reduced to the simple form simply by considering the different branches as separate actions.

8 Progressive Planning Methods

Progressive planning methods are those where one considers the tree of possible action sequences, starting in the given starting state and proceeding forward from it, searching the tree until one has found a sequence of actions that will produce the desired goal state, according to the available descriptions of the actions. The challenge for this approach is how to avoid the combinatorial explosion, and how to direct the search so that actions that are likely to lead towards the goal are favored. Progressive methods are therefore appropriate in applications where there is additional, so-called *heuristic* information that can guide the search from starting state to goal state. This includes, in particular,

- Applications where one has a good estimator for the remaining distance to the goal. In this case one can organize the search so that it always proceeds in the direction where the estimated remaining distance to the goal is minimized.
- Applications where there is sufficiently much heuristic information in the form of rules that can guide the search, for example, rules saying that certain actions are improductive in certain world states, or rules saying that particular subsequences of two or three actions are usually not meaningful.

Furthermore, in applications that involve additional kinds of actions besides those that fit into the state-transition schema, there is usually no alternative to using progressive planning methods, simply because no regressive or other methods are known, and it seems unlikely that any will be found. This includes, in particular,

- Applications involving nondeterministic actions.
- Applications involving concurrent actions, actions with extended duration in time so that one has to take their duration into account for the planning, and actions whose duration is conditional on the starting state or on the effects of other, concurrent actions.
- Real-world applications where the persistence assumption does not hold, so that the planner must combine consideration of a possible sequence of actions with predictions about what *other* things are likely to occur in the world at the same time as these plan actions are performed. Regressive and middle-out methods are very hard to use in such applications.

These kinds of applications are very important in practice, in particular for real-word robot applications, so progressive planning methods are quite significant. They usually require reasoning about cause and effect, besides reasoning about actions, so the progressive planning mechanism becomes a part of a prediction mechanism that simulates likely developments in the agent's immediate future, although also taking possible actions of the agent itself into account. We shall therefore describe progressive methods for action planning in the context of reasoning about causes and effects.

9 Planning in Cognitive Architectures

The methods that have been described in this note are dedicated to the problem of action planning. They should be compared with descriptions of actual software systems that do planning and that are used in real-life applications, such as O-Plan, TÆMS, and other systems that are described in Chapter 2 of our report "Autonomous Intelligent Agents" that is also used for the present course TDDC65. These systems contain many additional facilities and it is not so easy to discern the "pure" action planning methods in them.

This illustrates that action planning in artificial intelligence should be understood as a kind of "engine" that is sometimes used in a free-standing way, but quite often it is closely integrated into a larger system. The character of the surrounding system has implications for many aspects of the planning mechanism, including the need for having additional features and for permitting more types of actions, and even for the choice which of the basic planning methods shall be used.

Here is one example of how the surrounding system may affect the choice of planning method. The descriptions of planning methods in earlier sections refer to a particular "goal state" for the planning task, and this can easily be read as though there is one single objective that one strives for, although this objective is characterized in terms of a set of conditions that together make up the goal state. Some applications are like that, in particular in those cases where the planning task is performed in response to a request from a user who has a single objective for his or her request.

However, there are also applications where the goal state represents the combination of several objectives. For example, the task of planning what to do during one's lunch break may include both getting something to eat and doing a number of errands. In such cases the major challenge in the planning task is maybe not so much in identifying what actions are to be performed (each objective may map to one or a few corresponding actions in a quite simple way), but rather in *scheduling* those actions in a suitable order while taking various constraints into account, such as the walking distance or travel time between different places that one is to visit, opening hours for services, and expected length of waiting lines in the cafeteria or elsewhere. The scheduling activity is similar in some ways to planning inasmuch as it has to match postconditions with preconditions, but usually it also involves quantitative considerations for the "cost" of the actions being considered.

None of the action planning methods that have been described here is directly applicable to such an extended and modified planning task, but it may be more or less difficult to adapt them to those needs. Pure regressive planning methods, such as the method based on the situation calculus is not very well suited in this case; progressive planning is more promising, and methods that use layered plans, such as Graphplan, are also likely to be good starting-points for building a system of this kind.

In summary: we have described a number of planning methods; the choice of planning method depends on the requirements of the application at hand, but in many cases one shall view the planning method as a kernel that has to be extended and modified in order to fit the system into which it will be integrated. The planning methods are described in terms of logic, but they use it in two different ways: there are *inference-based* planning methods where a plan is obtained as the result of deduction or abduction, [2] and there are also *state-space based* planning methods that operate directly on the actions' prestates and poststates. The state-space based methods can be specified as constraint satisfaction problems.

[2]Abduction-based methods have not been described in this report, however.