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 II

Lecture note for course TDDC65

This series contains technical reports and tutorial texts from the project on
the Knowledge Representation Framework (KRF).

The present report, PM-krf-017, can persistently be accessed as follows:

Project Memo URL:	http://www.ida.liu.se/ext/caisor/pm-archive/krf/017/
AIP (Article Index Page):	http://aip.name/se/Sandewall.Erik.-/2010/015/
Date of manuscript:	2010-11-01

Related information can also be obtained through the following www sites:

KRFwebsite:	http://www.ida.liu.se/ext/krf/
AIP naming scheme:	http://aip.name/info/
The author:	http://www.ida.liu.se/∼erisa/

Chapter 1

More General Views of Actions

The preceding lecture note "Reasoning about Actions and Action Planning" described representations and planning methods that were based on the *state-transition schema,* where each action was characterized in terms of a *static precondition,* a *dynamic precondition,* and a *(dynamic) postcondition,* and where the dynamic pre- and postconditions could be characterized as partial states. A state is then a mapping from features to values that is intended to characterize the state of the microworld at hand at one point in time, and a partial state is a subset of a state whereby only some of the features are assigned values.

The state-transition schema has the advantage that it allows for several practically useful planning methods, but it has the disadvantage of being quite restrictive with respect to what kinds of actions can be described. The present lecture note shall therefore describe other, and more general ways for representing the effects of actions, and for reasoning about them.

The preceding lecture note described how planning could be done either progressively (searching forward from the given starting state), regressively (searching backward from the given goal state), or even middle-out (starting by assuming one or a few main actions in the middle of the forthcoming plan). The latter alternatives are not available in the more general cases that we shall consider here, or only with some difficulty, so the present note will focus on progressive planning. Progressive planning methods were not covered in the previous note, so the present note complements it with progressive planning methods even for the state-transition schema.

Three generalizations of the state-transition schema will be considered here:

- *State Transition with Conditional and Nondeterministic Actions*
- *State Transition with Immediate Causation* (STIC), which is similar to the previous case except that each action is considered to have both *direct* and *indirect* effects. The direct effects are represented like in the basic schema. Indirect effects are the results of cause-effect rules that apply immediately when the action has been performed.
- The *Trajectory Set Schema* (TSS), where a trajectory is a sequence of partial states of length 2 or greater, and each action is characterized

by a set of such trajectories. The combination of dynamic precondition and postcondition is considered as a trajectory of length 2. The Trajectory Set Schema is used for representing concurrent actions and actions with extended duration in time.

The first two cases will be treated together in Chapter 2; the case of TSS will be treated in Chapter 3.

Consider for example the action of pressing a light-switch which has the effect of turning on the lamp that is connected to the switch. Using the STIC schema, this action can be represented so that the direct effect consists of the change of mechanical configuration in the switch at hand, and the indirect effect includes the change in the lamp. One or more causation rules are used for relating the former effect to the latter one. Notice however that the exact decision as to what is to be considered as a direct effect or an indirect effect is up to the person designing the representation. For example, the possibility for passing an electric current through the switch changes as a result of this action, but is this considered to be a direct effect or an indirect effect? This is up to the designer to decide.

One may wonder, therefore, what is the advantage of making the distinction between direct and the indirect effects. Why not consider even the changes in the lamp, such as "beginning to shine" and "becoming warm" as direct effects? The answer for this example is that if there is only one lamp in the microworld at hand then it does not matter, maybe, but if several lamps are considered then the use of the STIC schema provides more modularity. For example, turning a switch will always have the mechanical effects in the switch itself, but the effects on one or more lamps, and on other devices that may be connected to the switch, depends on the structure of the electrical wiring. Therefore, instead of having a separate effect rule for each switch, one can have a single effect rule that applies to all switches, a declarative description of the electrical wiring, and one or a few causation rules that describe how switches influence currents and how currents influence lamps and other devices. Logical deduction can then be used for inferring indirect effects in each case.

In addition, the separation of information into distinct parts makes it possible to use it for more purposes, even in the case of a single switch and a single lamp. A description of wiring and electrical connections in a house may be used for other ends besides figuring out what happens when a switch is turned. Also, a diagnostic system that analyzes the reasons why a lamp or other device does not function as expected will also need a description of the various parts of the system at hand, their relationships and ways of functioning.

The STIC schema is simpler than the TSS schema since it is inherently sequential, like the basic state-transition schema. STIC assumes that one action is performed and all its indirect effects take place, and only then can the next action be performed. In other words, although in terms of physics there may be a time delay before the indirect effects are obtained, this time delay is so small compared to the duration of the actions themselves that it can be disregarded.

Consider by contrast the following example of causation. An agent performs the action of opening the faucet for a bathtub, whereby the immediate effect is that water starts coming out of the faucet and going into the bathtub.

Unless the agent closes the faucet in time, after a while the bathtub will be full and water starts coming out on the bathroom floor. [1] If the representation of this microworld includes actions that the agent can perform while the bathtub is filling up, such as making a phonecall, or putting bath-salt into the water, then the causation will introduce a concurrency *even if* the actual actions that are performed by the agent are entirely sequential. For this reason it is natural to consider *delayed causation* together with *concurrency between actions,* and the trajectory-set schema applies to each of those as well to their combination.

Besides these major cases, there are also some relatively simple extensions of the basic state-transition schema:

- *Nondeterministic Actions,* where a given prestate may result in several alternative poststates.
- *Action Failure,* where in some cases an action fails to have the intended effects. Cases where an action always fails for a given prestate are best handled by a more narrow definition of its prestate, and cases where an action may or may not fail for a given prestate is a special case of nondeterministic actions and may be handled as such.

Nondeterminism is also an issue for the STIC and TSS schemas. The simple case of basic state transition with the addition of nondeterminism will therefore be considered in the context of the STIC schema.

Throughout this memo we shall make the same assumption as for partial-order planning in the previous memo, namely, that actions are atomic entities without internal structure for the purpose of the representation. Therefore actions will be written as simple symbols in all examples, and without surrounding square brackets. Actions which originally are represented as an action verb followed by some arguments or parameters must therefore be replaced by the set of all their instances for different values of those arguments or parameters. It is furthermore assumed that the (resulting) set of actions is finite.

[1] This assumes that the bathtub does not have an escape outlet that will protect from overflow. Bathtubs in Europe often lack such an outlet whereas they are common in the U.S.

Chapter 2

State Transition with Immediate Causation

2.1 Generalized Semantics of Actions

The previous memo characterized actions using the prestate function `prest` and the poststate function `post` whereby `(prest .a)` and `(post .a)` are partial states when `.a` is an action. This is one particular choice of the *semantics* of actions, that is, a choice of assumptions about the structure and the properties of actions. This particular semantics is quite restricted since it does not allow for concurrent actions, nondeterministic actions, causation, and the other things that were mentioned in the previous chapter. In order to proceed to these planning problems we must therefore first define extensions to the semantics.

2.1.1 Semantics for Nondeterministic and Conditional Actions

We now replace the functions `prest` and `post` by an *action transition function* `atrans` whereby `(atrans .a)` is a set of sequences of length two, where each sequence consists of a possible prestate and the corresponding, resulting poststate. In other words, each action is defined by a set of prestate-poststate pairs, instead of just one such pair.

In this way it is possible to represent both conditional and nondeterministic actions. Consider for example a microworld where the current state has one single component, such as

```
{[: agent-has 22]}
```

expressing that the agent owns 22 units of money, and where the agent can engage in a coin-tossing action `play` whose effect is that this amount of money owned is either increased or decreased by 1. Then the value of `(atrans play)` will include the following twotuples, among others:

```
<{[: agent-has 22]} {[: agent-has 21]}>
<{[: agent-has 22]} {[: agent-has 23]}>
<{[: agent-has 23]} {[: agent-has 22]}>
```

```
<{[: agent-has 23]} {[: agent-has 24]}>
```

and so forth.

Notice that if `(atrans .a)` contains several twotuples with the same first element then the action is nondeterministic. If this is not the case then it is deterministic. If `(atrans .a)` for a deterministic action consists of more than one twotuple then the action is conditional, i.e. it will have different outcomes depending on the starting condition, and otherwise it is unconditional. [1]

The definition of the function `apply` is consequently modified as follows. If `.a` is an action and `.s` is a feature state, then

```
(Apply .a .s)
```

is a set of feature-states that is obtained by identifying all twotuples in `(atrans .a)` whose first element is a subset of `.s` , replacing that subset by the second element of the twotuple, and forming the set of all the resulting modifications of `.s`. The modified function is written with a capital first letter as a reminder that the value is a set of states, and not a single state.

Moreover, if the function `Apply` is used with a set of states as its second argument, then its value is obtained by using `Apply` on `.a` and each member of the second argument, and then taking the union of the resulting values.

For example,

```
(Apply play (Apply play {[: agent-has 22]} ))
```

has the following value, if `play` is defined as above:

```
{ {[: agent-has 20]} {[: agent-has 22]} {[: agent-has 24]} }
```

2.1.2 Semantics of Immediate Causation

In principle we distinguish between the direct effects of actions, which are characterized using the function `atrans,` and the indirect effects that are due to immediate causation. The combination of the direct and the indirect effects is called the *total effects.* However, rather than introducing one more function besides `atrans` it is sufficient, and simpler, to just introduce one more action which will represent effects done by nature through causation. This "action" will be called `causation.` Then `(atrans causation)` will be a set of twotuples each of which describes the prestate and the poststate of some spontaneous transition that takes place by itself.

There will of course be some states where no causation takes place, and for such states `(atrans causation)` shall contain a twotuple whose two elements are equal. Suppose an action is performed on a starting state s0 and its direct effects result in a state s1, and then there is a spontaneous transition to state s2 and another spontaneous transition to state s3, and then no more. In this case the steps from s1 to s2 and from s2 to s3 will be represented by twotuples in `(atrans causation),` and with respect to s3 there will be some partial state `.ps` that is a subset of s3 and such that <`.ps .ps`> is a member of `(atrans causation).`

[1] This is under the assumption that the definition of `(atrans a)` is written in the simplest possible way.

Now, for an action `.a` and a state `.s` consider the following sequence:

```
(Apply .a .s)
(Apply causation (Apply .a .s))
(Apply causation (Apply causation (Apply .a .s)))
   ...
```

If `causation` is defined in a reasonable way then there shall be some point in the sequence where one element equals its predecessor, which means that at that point there is no further change according to `causation` and all immediate causation has already taken place. If there is such a point then of course all following elements will also be the same, [2] so that the sequence has *converged* on a specific set of *stable states.* These stable states are the ones that represent the possible final outcomes of the action `.a` when applied in the state `.s`.

It is also possible to construct a formal definition for `causation` whereby there is no convergence since the state-sets in the sequence oscillate between different possibilities. For example, suppose the state contains a feature indicating that the agent is happy, and `causation` is defined so that if the agent is happy then it immediately becomes unhappy, and vice versa, and the set `(Apply .a .s)` that one starts with only contains states where the agent is happy. Then there will be no convergence. We therefore need the following concept:

A definition of `(atrans causation)` is *convergent* if and only if the sequence of states

```
(Apply causation .S)
(Apply causation (Apply causation .S))
  ...
```

converges to a non-empty set for every choice of `.S` as a set of states.

In this report we shall only be interested in cases where `(atrans causation)` is convergent.

The requirement that the sequence shall converge to a non-empty set guarantees that if the action `.a` is executable, so that `(atrans .a)` is non-empty, then it will not be possible for `causation` to make it look like the action is non-executable. Moreover, if the action is nondeterministic, so that there are several possible immediate effects, then it is not possible for `causation` to "hide away" any of those alternatives; each of them must be able to produce one or more alternatives for total effects.

2.2 Planning Methods

2.2.1 The Need for Conditional Plans

First of all we consider the simple case where `causation` is defined in such a way that

```
(Apply causation .s) = {.s}
```

[2] Since if x = (f x) then (f x) = (f (f x)) for any function f.

for every state `.s,` which means that the direct effects of an action are also the total effects. This is simply the basic state-transition schema, extended with conditional actions and nondeterminism.

The effect of nondeterminism on the planning problem is that one needs *conditional plans.* In the previous memo we assumed that the starting state for the planning problem was complete, so that the values of all the features was known, and we assumed that actions were deterministic. In this case it was possible to use plans that just specify a sequence of actions, without the need for conditionals. If some actions can have alternative outcomes *depending on the prestate* then it is still possible to use plans that are just a sequence of actions. This is the case of deterministic, conditional actions. However, if some actions can have alternative outcomes *for the same prestate,* which means that they are nondeterministic, then the outcome can not be predicted when the plan is made, so the plan must contain alternative branches that are chosen according to the outcome of these nondeterministic actions.

Actually, the same is true if the starting state for the planning problem is only partially specified, since then even actions that are just conditional on the prestate, without being nondeterministic, may also require conditional plans. [3]

It is natural to treat conditional actions and nondeterministic actions together since nondeterministic actions introduce the need for conditional subplans, and conditional actions can be viewed as a special case of conditional subplans.

2.2.2 Regressive Planning with Nondeterministic Actions

Consider first the regressive planning method using the situation calculus. Is it possible to adapt this method so that it applies for nondeterministic actions as well?

The immediate answer is no; there is no (known) reasonable way of modifying it so that it considers several nondeterministic outcomes of the actions on an equal basis. However, there is a fairly straightforward method using *primary outcomes* which is as follows. The term *point of nondeterminism* will refer to a combination of an action `.a` and a partial state `.s` where `(atrans .a)` specifies multiple outcomes, i.e. it contains several twotuples of the form $<$`.s .s'`$>$ for different `s'` . In each point of nondeterminism, select one of the outcomes as the primary one, and solve the planning problem using only primary outcomes. Then consider all the points in the obtained plan that rely on a primary outcome, and solve the planning problem again from the remaining, alternative outcomes to the given goal state for the entire problem. Continue doing this until all alternative outcomes have been considered.

This method is particularly attractive if some of the alternative outcomes are judged to be more likely than others, since in this case one will choose

[3] One assumes then of course that the missing state elements are known when the execution of the plan begins. If they are not then there is a need for introducing information-obtaining actions in the plan, which is an interesting and important problem, but outside the scope of the present text.

the most likely outcomes as the primary ones, so that they will be considered first. In fact, one may organize the planning process so that it starts with the most likely outcomes and proceeds with successively less and less likely outcomes, and stopping the process when it is no longer reasonable to spend more time on it.

Notice that in practice it is often not necessary to cover all the possible outcomes of all the nondeterministic actions with a continued plan. One may simply leave some of the cases open and arrange so that if these cases do occur during the execution of the plan then one will stop the execution and do new planning.

2.2.3 Progressive Planning of Conditional Plans without Causation

Before considering possible generalization of the other planning methods from the previous lecture note, we shall discuss progressive planning. The techniques that are introduced for this purpose will also be needed in those other methods.

The basic method for progressive planning is very simple, especially in the case of deterministic actions: you start with the given starting state for the planning problem; you generate the tree of possible sequences of actions; for every action sequence you identify the state(s) that may be result of performing that sequence; and you continue doing this until you have found a plan that achieves the goal.

The more general case that admits nondeterministic actions may be addressed using primary outcomes, in the way that was described in the previous subsection. However, it may also be addressed more directly by gradual extension of conditional plans. At this point we need to introduce a notation for such conditional plans.

Let `start` be a given starting state; it may be a partial or a complete state of the microworld at hand. An *evaluated partial plan* (epp) is an expression which is formed as either of the following kinds of KR expressions:

```
[soares a1 a2 ... an R]
[if c at af]
```

where each `ai` is either an action or an evaluated partial plan, recursively, `at` and `af` are `soares-` expressions, `R` is a set of states representing the possible outcomes of the sequence of actions preceding it, and `c` is a conditional expression that can be evaluated in states and that obtains the value `true` or `false` there.

Within the framework of KR expressions it is also possible to write `soares-` expressions using the general-purpose operator `soact` that has been introduced before for "sequence of actions," with the addition of a parameter `res` for "result states," as follows:

```
[soact a1 a2 ... an :res R]
```

However we use the separate operator `soares` in this lecture note.

The intended meaning of such an evaluated partial plan is the obvious one: `soares-` expressions are executed by doing the actions in it in sequence,

and `if-` expressions are executed by evaluating the conditional expression in the current state and then executing `at` if the value is `true` and `af` if the value is `false.` Just as planning in the deterministic case consists of keeping track of a number of alternative action sequences and extending one or the other of them in each step, so planning in the nondeterministic case consists of keeping track of a number of alternative evaluated-partial-plans and extending one or the other of them in each step, until one arrives at the point where one of these alternative epp's is such that it will achieve the given goal in all its branches, i.e. regardless of which path is taken in any of the `if-` expressions.

Two things have to be made clear then: (1) how can one extend an evaluated partial plan, and (2) how can one determine its outcomes. The inclusion of the `R` component at the end of a `soares-` expression is a help for the latter purpose.

First of all, if an epp consists of one single `soares-` expression, so that all the `ai` in that expression are actions and not subplans, then the `R` in

```
[soares a1 a2 ... an R]
```

must always be

```
(Apply an (Apply an-1 (Apply ... (Apply a1 s0) ... )))
```

It follows that the empty plan, containing no actions at all, can be written as

```
[soares {start}]
```

and that the plan shown above can be extended by adding the action `an+1` obtaining

```
[soares a1 a2 ... an an+1 (Apply an+1 R)]
```

provided that the prestate of the action `an+1` is satisfied in all the states in the set `R.` Therefore, if there is no need for conditionals, one can simply let each of the alternative plans have this form, and the planning process consists of gradually extending these sequences with several alternative choices of action in each step (thereby obtaining a tree search) until one finds a sequence where all members of the set `R` satisfy the given goal state. This is essentially the same as for the case without conditionals although expressed in terms of the `soares` operator.

However, if no plan can be obtained in this way then one must try obtaining a conditional plan. Consider the following simple example. The state of the world is characterized using two features, `hungry` and `has-money,` each of which may have the value `true` or `false.` The goal state is a partial state saying that `hungry` shall be `false` whereas `have-money` is not a goal. One considers an evaluated partial plan

```
[soares a1 a2 ... an R]
```

where `R` consists of two states, `hungry` is `true` in both, and `has-money` is true in one and false in another. Moreover one disposes of two actions, `buy-food` which is applicable if one has money, and `beg-food` that is a last resort which is only considered appropriate if one does not have any money.

The natural way of extending the evaluated partial plan in this case will be so as to obtain

```
[soares a1 a2 ... an
  [if [Hc has-money true]
      [soares buy-food {{[: hungry false][: have-money true]}
                        {[: hungry false][: have-money false]} }]
      [soares beg-food {{[: hungry false][: have-money false]}} ]]]
```

The details of how to proceed from having bought or begged food to actually having eaten it to satisfaction are not represented. The above also assumes that after having bought food one may or may not still have money, and that after begging for food one will still be without money. The predicate `Hc` was used in previous lecture notes and is true if the feature in its first argument has the value in its second argument, in the state at hand when the condition is evaluated. *(End of example).* The general rule is therefore as follows. Consider the evaluated partial plan

```
[soares a1 a2 ... an R]
```

and a proposition `c` that is true for some of the states in `R` and false for some other states. (If it is true in all those states, or false in all of them, then there is no point in introducing that particular condition). Let `Rt` be the set of the former and `Rf` is the set of the latter, so that `R` is the union of `Rt` and `Rf.` Let `at` be an action whose prestate is satisfied in all members of `Rt` and let `af` be an action whose prestate is satisfied in all members of `Rf.` Then construct the following expression

```
[if c [soares at (Apply at Rt)][soares af (Apply af Rf)]]
```

and extend the epp at hand so as to become

```
[soares a1 a2 ... an
  [if c [soares at (Apply at Rt)][soares af (Apply af Rf)]]
  RR ]
```

where `RR = (Apply at Rt)` $\cup$ `(Apply af Rf).` It is fairly easy to see that this is a correct extension. The execution of the actions `a1` to `an` will result in any of the states in the set `R.` The extended plan with the `if-` expression will proceed with either `at` or `af,` so `RR` is clearly the set of all possible states that may be the end result of the extended plan. Therefore it is correct to place it as the final element in the `soares-` expression.

When a conditional subplan has been introduced then one has several `soares-` expressions that can be extended, namely either on the top level or on the lower levels.

The rule for extending the evaluated partial plan with an `if-` expression can therefore be written in a more concise way, as follows. The epp

```
[soares a1 a2 ... an R]
```

may always be extended so as to become

```
[soares a1 a2 ... an [if c [soares Rt][soares Rf]] R]
```

where `Rt` and `Rf` are the two non-empty, complementary subsets of `R` that were defined above. This rule, together with the rule for adding an action at the end of a `soares-` expression on any level, can together provide the extension shown before.

The rule for modifying the `R` component of the evaluated partial plan becomes a bit more complex when the plan contains conditional sub-expressions.

It is as follows:

- If an action is added in the top-level `soares-` expression, then modify the `R` component as described above.
- If an `if-` expression containing no actions is added then the `R` component on the top level does not change, as described just above.
- If an action is added at the end of a `soares-` expression in a conditional one or more levels down, and if this conditional is the last element before the `R` component in the surrounding `soares-` expression, then that `R` component must be reset to become the union of the `R` components ending the two branches in the conditional. (Example below).
- If an action is added at the end of a `soares-` expression in a conditional one or more levels down, and if this conditional is *not* the last element before the `R` component in the surrounding `soares-` expression, then that `R` component must be recalculated by first taking the union set of the lower-level sets as described in the previous item, and then modifying it by applying again the successive actions following the conditional expression.
- If an action is added at the end of a `soares-` expression *more than* one level down, then in addition to the changes that are specified in the previous two items, one must also modify the `R` component in all `soares-` expressions on higher levels above the given one, proceeding upwards and recalculating the `R` on each level so that it is obtained from the `R` components on the level immediately below it.

As an example for the case in the third item, suppose one has

```
[soares a1 a2 ... an [if c [soares Rt][soares Rf]] R]
```

and it is extended to become

```
[soares a1 a2 ... an [if c [soares Rt][soares ak Rf']]  R']
```

then the primed components must be calculated as `Rf' = (Apply ak Rf)` and `R' = Rt ∪ Rf'`. Also, as for the fourth item, if one has

```
[soares a1 a2 ... am
        [if c [soares Rt][soares Rf]] an-1 an R]
```

and one extends it to

```
[soares a1 a2 ... am
        [if c [soares Rt][soares ak Rf']] an-1 an R']
```

then `Rf'` must be like in the previous example, and `R'` must be

```
(Apply an (Apply an-1 (union Rt Rf')))
```

where `union` is set union, i.e. ∪.

If an evaluated partial plan for a given planning problem has been started as

```
[soares {start}]
```

and it has been obtained by then extending the plan step by step using the above rule for recomputing the `R` component, then it is clear that in each step the top-level `R` component is a set that contains exactly those states that may be the result of executing the plan in question. Therefore, the criterium for whether a plan achieves the given goal, which has been specified as a partial state containing the required feature-value assignments, is simply that the goal must be a subset of each of the members of `R.`

This is essentially all; the rest is "only" the search process. The system has to consider alternative, evaluated partial plans, decide which of them to select for further extension, and to keep doing this until it has found an epp where all members of the final `R` set of the top-level expression include the given goal state.

In practice this method will only be feasible if one has quite small problems involving short plans, or if one can use so-called *heuristic* information that can effectively guide the search process. This heuristic information may consist of rules that indicate which actions and branches to try, or which ones not to try. It may also consist of numerical-valued functions that estimate the remaining "distance" to the goal or the probability of reaching it eventually. Such an estimate can be used for always expanding the most promising-looking branches in the search tree. The principles and the systematic methods for doing this must be left out from the present lecture notes.

2.2.4 Regressive and Progressive Planning for Nondeterministic Actions and Immediate Causation

We now proceed to the case where there is the additional possibility of immediate causation, so that the total effects of an action consist of the direct effects, as stated by the function `atrans` for the action in question, followed by the indirect effects which are stated by `(atrans causation).` The use of regressive planning may or may not be possible, depending on the structure of `causation.` If `causation` is deterministic then one can simply define a function that is similar to `Apply` but which takes both direct and indirect effects into account, and if the direct effects are nondeterministic one can use the method of primary outcomes that was described above. Even in the case where `causation` is nondeterministic it is in principle possible to identify one of its alternative outcomes as the primary one, for each of its points of nondeterminism. The practical feasibility of this has to be assessed for each application separately, but it is not a method that will work in all cases.

The use of progressive planning is very straightforward, on the other hand. The method of gradual extension of a set of evaluated partial plans can be applied just like for the case without causation, and the only change is that each time an epp is extended with one more action, one must consider not only the direct effects of that action, but also the possible causation chains leading up to stable states.

2.2.5 Island Planning for Nondeterministic Actions

It is possible to generalize partial-order planning so that it works for nondeterministic actions with or without immediate causation, but the description of this generalized planning process is a bit complicated and we shall not address it here. Let us however briefly describe a simpler case of middle-out planning for deterministic actions with or without causation. Please recall that the method of partial-order planning can be used for both progressive, regressive, and middle-out planning; what we shall describe now is a limited case of middle-out planning, called *island planning.*

The basic assumption now is that you have a planning problem where at least some of the actions are nondeterministic, and in in addition you can make an assumption on heuristic grounds that one or a few specific actions are likely to occur in the final plan, but probably inside the plan and not at its beginning or its end. This assumption about main actions may, for example, be based on a comparison between the starting state and the goal state in order to identify the differences, together with some rules which say that differences in certain features suggest the use of particular main actions.

Step 1: If there is more than one main action then make an assumption about the order in which they will occur in the final plan. The result of this is that we have decomposed the original planning problem into two or more planning problems that will hopefully be easier to solve, namely, from start to the preconditions of the first main action, from the outcome of the first main action to the preconditions of the second main action, and so forth, and then from the outcome of the last main action to the goal state.

Step 2: Consider the subproblem of finding a plan from the starting state to the first main action. Do a *limited* regressive planning in order to find out whether there are some actions that are likely to be used at the end of that subplan, namely, deterministic actions that achieve some of the preconditions that are required by the main action and that are not present in the starting state.

Step 3: Make progressive planning from the starting state to the first main action. Look primarily for a plan that leads up to some of the actions that were introduced in Step 2, but allow also for plans that go directly to the preconditions of the first main action. This is because it may happen that some action at the beginning of the subplan happen to achieve what the actions from Step 2 also do.

Step 4: When a plan from the starting state to and including the first main action has been found, then identify the result state of that plan, and use it as the starting state for a plan that goes to the second main action, or to the goal state if there was only one main action.

Following steps: Repeat steps 2 and 3 until a plan reaching the goal state has been found.

Backup step: If Step 3 fails in one of the cycles, then back up to an earlier cycle and find another plan there. If Step 2 fails in a cycle, in the sense that no relevant action is found, then proceed anyway since Step 3 can still be done in order to find a plan directly to the preconditions of the next main action.

This middle-out planning method shall be thought of as a kind of framework that can be adapted to various applications in various ways. For example, if the application is such that it will generate two or more *alternative* main actions, but only one of them is expected to be used, then it may be a good idea to first address the planning subproblem from each of them to the goal state in order to find out which of them is more likely to be relevant, and only then do the planning problem from the starting state to the selected main action.

2.3 Logic for Characterizing Causation and Nondeterministic Actions

The previous lecture note was based on the state-transition schema and described how the effects of actions can be described in two different ways, if that schema is used: with the `prest` and `post` functions which map actions to partial states, and with effect laws using the predicates `D` and `H`. These representations are exchangeable, so that it is possible to convert one of them into the other.

We have also seen how some planning methods rely on the use of effect laws, in particular for regressive planning using the situation calculus, and other planning methods are easily expressed using the `prest` and `post` functions. The difference is one of convenience rather than possibility, since it would also be possible to express e.g. partial-order planning in terms of effect laws, but this way of expressing it is more complicated.

The relationship between these two representations can be formally characterized as follows. Consider a simple plan consisting of a sequence of actions, and a starting state for that plan. Assign timepoints 0, 1, 2, 3 etc to the points where one action ends and the next action starts in the plan. Rewrite each step in the plan using the `D` predicate, so for example

```
[soares buy-food eat-food R]
```

is rewritten as

```
[D 0 1 buy-food]
[D 1 2 eat-food]
```

Then construct the states that result from the execution of the successive actions for a given starting state, associate these with the timepoints from 0 and up, and define the predicate `H` so that it can be evaluated in those states. Then, if the translation from the `prest` and `post` functions to the effect axioms has been done correctly, it must be the case that *the value of any of the effect laws must always be* `true`, for any choice of values for the variables that occur in it.

A definition of how one shall calculate the values (including truth-values) for formulas in a particular notation is called a *semantics* for that notation. Therefore the `prest` and `post` functions are the essential part of the *state-transition semantics* for plans (expressed using the `D` predicate) and for effect rules.

In the more general case of State Transition with Immediate Causation (STIC) which also allows for conditional and nondeterministic actions, it is

natural to use the same two perspectives. The `atrans` function provides the essential part of the semantics, and we have described partial-order planning in terms of that function.

There are also planning methods for the STIC schema that are based on the use of formulas for plans and effect rules, rather than using the `atrans` function directly. These techniques are not as well developed as for the basic state-transition schema, but they are an active area of research. In this report we shall only consider the first part of the problem, that is, how shall one represent action effect laws as logic formulas if the actions involve conditional effects, nondeterminism and immediate causation. The second part of the problem, which is how to organize the search for a plan will not be covered.

2.3.1 Actions with Conditional Effects

Please recall the following example of an effect rule from the previous lecture note:

```
(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] )
```

This shows how the dynamic precondition of the action occurs in a literal using the `H` predicate in the antecedent, that is, the first argument of `imp,` and the postcondition occurs in the consequent i.e. the second argument of `imp.` It is easy to generalize this way of writing the rule to the case where the action has conditional effects, most easily by having one rule for each of the possible cases and where each of the rules follows the same pattern as the above.

Consider for example the case of a toggling light-switch, which is such that if you press it when the light is on then it will go off, and vice versa. Let `light` be the feature representing whether a particular lamp is on, with the value `on` or `off.` This action can be characterized using the following two effect rules:

```
(imp (and [D .s .t toggle][H .s light on])
     [H .t light off] )

(imp (and [D .s .t toggle][H .s light off])
     [H .t light on] )
```

2.3.2 The Use of the Occlusion Predicate

The previous note mentioned the use of the occlusion predicate and we must return to it here. The problem is the following. Although it is quite possible to use action effect rules of the kind that were just shown, there is a particular disadvantage with this that arises because of the *frame problem,* namely, that it forces one to introduce a considerable number of direct or reverse frame axioms. (See Section 4 of the previous memo for the details). Therefore, a better solution is to write effect rules that specify which features

may change their value as the effect of the action, which will be as follows using the same example.

```
(imp (and [D .s .t toggle][H .s light on])
     (and [X .s .t light]
          [H .t light off] ))

(imp (and [D .s .t toggle][H .s light off])
     (and [X .s .t light]
          [H .t light on] ))
```

An equivalent formulation containing both cases is as follows and it may actually be more readable:

```
(imp [D .s .t toggle]
     (and [X .s .t light]
          (imp [H .s light on][H .t light off])
          (imp [H .s light off][H .t light on]) ))
```

A particularly simple case is for the action `toss-coin` which is as follows, assuming that the action affects the feature `upside` with the value `heads` or `tails:`

```
(imp [D .s .t toss-coin][X .s .t upside])
```

The point is that here it is only known that the value of `upside` may be either one of the possible values after the action has been performed, and that is what the `X` predicate expresses.

The previous lecture note described how the occlusion predicate `X` can be used together with the PMON axiom which says that if a feature is not occluded over a certain interval, then its value will not change there. There are actually also other ways of using the information about occlusion, other than PMON, but these are mostly useful in the context of the Trajectory Set Schema in the next chapter.

2.3.3 Immediate Causation Rules in Logic

Please recall that immediate causation is modelled using a causality transition from state to state, and that it is the stable state after potentially a number of such transitions that is of interest for representing the indirect effects of actions. This semantics was defined above in terms of an artificial action `causation` and the use of the `Apply` function.

From a practical point of view it is tempting to write causation laws as logic formulas. For example, if the toggling light-switch is modelled in such a way that the physical change in the switch is separated from the state of the lamp, using the features `(state myswitch)` and `(light mylamp)` respectively, for one particular switch and lamp, then one would express the direct effects of the action `toggle` as follows:

```
(imp [D .s .t [toggle .sw]]
     (and [X .s .t (state .sw)]
          (imp [H .s (state .sw) on][H .t (state .sw) off])
          (imp [H .s (state .sw) off][H .t (state .sw) on]) ))
```

and the effect on the lamp can be expressed as follows:

```
(imp (and [H .s (state .sw) on][controls .sw .lmp])
     [H .s (light .lmp) on])

(imp (and [H .s (state .sw) off][controls .sw .lmp])
     [H .s (light .lmp) off])
```

provided the following has also been stated:

```
[controls myswitch mylamp]
```

This example assumes a predicate `controls` specifying that the switch in the first argument entirely controls the device in the second argument. (If the lamp or other device is controlled by several switches that can be used independently then another predicate and a more complex rule will be needed). In all cases that use static preconditions, such as this one, it is convenient to use logic for expressing effect rules and causation rules. If rules are expressed directly in terms of the `atrans` predicate, on the other hand, then there is the problem that this only applies to the dynamic preconditions, and one has to introduce additional notation in order to allow for the static preconditions and the predicates that they require.

There are a few problems, however. One problem is with respect to the occlusion predicate that was discussed above. Given that we have the PMON axiom that says that if a feature is not occluded then it can not change, it seems that we must arrange for the feature `(light mylamp)` to also be occluded when `(state myswitch)` is occluded. There are three good ways of doing that. One possibility is to add axioms such as

```
(imp (and [controls .sw .lmp][X .s .t (state .sw)])
     [X .s .t (light .lmp)] )
```

Such axioms are called *occlusion causation rules.*

The other possibility is to observe that the light of the lamp is fully determined by the state of the switch, according to the effect rule above, and therefore to stipulate that the value of `(light .lmp)` is always occluded, technically speaking, if `.lmp` is controlled by a switch:

```
(imp [controls .sw .lmp][X .s .t (light .lmp)])
```

This works because the light-status of the lamp is always determined, for each timepoint separately, using the causation axioms above. Features that are handled in this way are called *dependent features.*

The third possibility is to introduce a predicate `is-dependent` on features, with axioms such as

```
(imp [controls .sw .lmp][is-dependent (light .lmp)])
```

and to modify the PMON axioms so that it becomes as follows:

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

Which of these three methods shall be used – occlusion causation rules, perennial occlusion, or the `is-dependent` predicate – must depend on the specific characteristics of the application and it is hard to give a general guideline.

2.3.4 Blinking and Back-and-Forth Causation

There is also a second problem with the use of logic formulas for expressing causation rules, as compared with the direct use of `(atrans causation)` as a set of twotuples.

Consider again the example with the light-switch and the lamp where the entire chain from switch to lamp, including the lamp turning on is considered as direct effects. Add a causation rule where, if the lamp is faulty in a particular way, it will begin to shine as the direct-effect rule specifies, but it will immediately overheat, the coil in it will melt, and the lamp will stop shining "instantly." This can be called a *blinking* causation relation. It can be represented using `(atrans causation)` without any problem, but the representation using a causation axiom that was shown in the previous subsection does not work, since in this case the effect rule for the action will state that the light will be on at the ending-time of the action, and the causation rule will state that the light will be off *at the same timepoint.*

The particular case where the direct effects and the indirect effects are incompatible can in fact be resolved by introducing extra timepoints, so that there is one timepoint for the end of the direct effects and a subsequent timepoint where the indirect effects have converged. However, this still does not work in all cases, since there may be incompatible effects *within* the causation chain. This can be called *back-and-forth causation.*

The following is an example of this. Suppose you have two features A and B whose values can be T or F, and a causation transformation that is defined as follows (with the obvious shorthand notation):

```
A:T, B:T  ->  A:F, B:F
A:F, B:F  ->  A:T, B:F
A:T, B:F  ->  A:F, B:T
A:F, B:T  ->  A:F, B:T
```

It is seen that `[A:F, B:T]` is the only stable state, so that an action whose direct effect is any of these states will anyway have that stable state as its total effect. However, the style of writing a causation rule in logic that was used above will not work in this case, since it is expressed so that the state of the microworld before and after one step in the causation transformation are referred to as being *at the same timepoint.* In the present example this will not work, and if one is going to characterize the values of these features in terms of timepoints then it has to be separate timepoints for the successive steps in the causation transformation.

Back-and-forth causation does not seem to be very common, fortunately, so the representation as logic formulas from the previous subsection can usually be used. with or without insertion of extra timepoints. However, this all means that when causation rules in an application are expressed in terms of logic then one must be observant on the structure of the causation transformation.

2.3.5 Endstate Description of Causation

Causation rules of the kind that were described above are written so that they characterize individual steps in the causation chain, and this is also

why they run into difficulties for blinking and back-and-forth causation. There is also another way of specifying extended causation chains, namely, using axioms that specify *some of* the changes that are obtained from the entire causation chain, but not all of them, but combined with *stable state axioms* that specify conditions that are true in all stable states.

The idea is as follows. In many practical cases one can see that there are a number of intermediate steps in causation chains, but they are so many and so complex that it is not realistic to document all of them. However, one does have a way of describing stable states which arise after all the causation steps have taken place. Therefore, one way of characterizing the indirect effects of an action is to specify some of the indirect effects, which in themselves will indicate a state that is not a stable state, and then adopt the stable state that is the "nearest" one to the obtained state according to some available metric. The commonly used metric is to find the stable state that differs from the obtained one in as few features as possible, so this method is commonly referred to as a method of *minimization of change.*

In the case of the burned-out lightbulb, for example, one could simply have a stable state axiom saying that a lightbulb having the particular fault that causes it to overheat will always be non-shining in stable states. The action that turns on the lamp will therefore have a direct effect that is not a stable state, and the nearest stable state, in terms of the set of features that have to be changed, is the state where the lamp in non-shining whereas all other features are unaffected.

The method of minimal change is often motivated on informal grounds in the literature, or based on the observation that it works as intended in a number of examples. However, a systematic study indicates that it does not work correctly in all cases, so it has to be used with a certain care. Just like when one works directly in terms of the causation transformation `(atrans causation)` it is necessary to be observant on the structure of causation in the application at hand.

Chapter 3

The Trajectory Set Schema

(To follow).