VICTORIA UNIVERSITY OF WELLINGTON
*Te Whare Wananga o te Upoko o te Ika a Maui*

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

# Agents That Plan and Act in a Dynamic and Only Partly Observable World

Isaac Sansom

Supervisor: Dr. Peter Andreae

October 17, 2005

Submitted in partial fulfilment of the requirements for
Bachelor of Information Technology.

**Abstract**

An intelligent agent who acts in a dynamic and only partly observable world must have some rules or guidelines which define what actions are possible to take and enable it to choose approapriate actions in any given situation. With these rules comes the ability to create plans to solve various tasks.

This report investigates a combination of the traditional planning approach and the alternative reactive planning approach. The combination of the two planning approaches has the potential to combine the strengths of each approach, while mitigating the individual weaknesses.

This report describes how the components are combined, and the example plans/tasks show how that combination can solve problems effectively, which neither planner could alone.

# Contents

# Chapter 1

# Introduction

An intelligent agent who acts in a dynamic and only partly observable world must have some rules or guidelines which define what actions are possible to take and enable it to choose approapriate actions in any given situation. With these rules comes the ability to create plans to solve various tasks.

Traditional approaches to planning require rules which describe the effects of actions, and attempt to construct plans containing actions whose effects satisfy the given task. Given that the world is dynamic, the agent will need to learn new rules to cope with changing aspects of the world. During the learning process, rules being learned will necessarily be incomplete. Unfortunately, these traditional approaches do not work when the provided rules are incomplete, or there are missing rules. Additionally, traditional planning approaches require complete knowledge of the current situation, and this is not possible in a partly observable world.

An alternative approach to planning is reactive planning. Reactive planning performs actions, not by examining the action's effects, but by knowing that an action is beneficial, given the current state and some goal. The reative planning approach unfortunately is not directed enough, such that there is large potential for the reactive planner to perform needless actions, and no guarantee that the reactive planner will ever reach the goal.

The goal of this report is to investigate a combination of the traditional planning approach and the alternative reactive planning approach. The combination of the two planning approaches has the potential to combine the strengths of each approach, while mitigating the individual weaknesses.

The report will begin by giving a more detailed background into traditional planning, reactive planning, and the CraneWorld simulation. Chapter two contains the overview of the the new CPR system that combines the two planning approaches. Chapters three, four, and five contain details about the various system components. Chapter six contains example plans produced by system, and sample tasks solved, and chapter seven contains concluding remarks.

## 1.1 Background

### 1.1.1 Traditional Planning

Traditional planners such as the GoalStack[1] and Partial Order[1] planners work backwards from the goal towards the current state. The actions used by these traditional planners are STRIPS rules, and consist of action name, effects, and preconditions. These planners need to backtrack — undoing previously made choices — which can resolve complicatations, but also results in significant increases in planning time. Partial Order planning also introduces the principle of least commitment — when a variable can be resolved in several different ways, there is no need to choose immediately, but instead wait until the choice must be made.

Learning STRIPS rules is a complicated process, due to the complexity of discovering hidden effects. An interesting system for learning STRIPS rules was developed by M. Wojnar in 2004 [2].

### 1.1.2 Reactive Planning

Reactive planners[3] can be considered to be 'non-planning' planners, providing actions which are considered to be helpful given the current state of the world and some goal. The main difference between reactive planners and traditional planners lies in reactive planners lack of consideration for the effects of actions — if the action is seen as beneficial, the action is performed regardless of the effects of that action.

Reactive planners use reactive rules instead of STRIPS rules. Reactive rules consist of a current state, a goal state, and an action to perform if the current and goal states are satisfied. Reactive rules are potentially very powerful, but learning them satisfactorily is very difficult. An system for learning reactive rules was developed by A. Longhurst [4].

### 1.1.3 CraneWorld Simulation

CraneWorld is a simulation of a discrete world in which intelligents agents can perform various tasks. The CraneWorld simulation was developed by P. Andreae and D. Gilligan, and reimplemented by M. Wojnar.

## 1.2 Approaches

When combining both a traditional and a reactive planner, there are several different alternative approaches that could be taken to solve a problem. Listed below are several approaches to combining the two planners.

- **Plan then react:** The planner would begin by planning as far from the goal to the current state as it can, and if the planner finds that the solution is unreachable, pass the best available plan forward to the executor/monitor. The executor/monitor will now perform reactive actions upon the initial state of the world so that those parts that the planner could not solve are now different, and hopefully closer to the goal. The executor/monitor will then return the new current state to the planner and the process beings again. Eventually the current state will be transformed into a state that is solvable, given the goal state, or the executor/monitor will determine that the current state is unsolvable.

$$\textit{Plan(Goal)} \rightarrow \textit{State}_n\textit{; Reaction(Current State)} \rightarrow \textit{State}_m\textit{; Plan(goal)} \rightarrow \textit{State}_m$$

- **React then plan:**A reverse of the above planner, this planner would begin by having the executor/monitor use reactive rules on the current state, given the goal state, until there are no more actions that are appropriate, or the goal state is reached. If there are no more appropriate actions, and the current state is not the goal state, then give the planner the current state and the goal state and begin planning. If a complete plan is available, use it, otherwise return the best available plan to the executor/monitor and repeat applying reactive rules on those current state facts that the planer could not solve.

$$\textit{Reaction(Initial State)} \rightarrow \textit{State}_n\textit{; Plan(Goal)} \rightarrow \textit{State}_m\textit{; Reaction(State}_n\textit{)} \rightarrow \textit{State}_m$$

- **Plan and react:**The planner could also approach the goal from both directions, creating a plan from the goal state in parallel with the reactive rules being applied to the initial state. When a subgoal of the plan becomes part of the current state, the planner will stop improving that part of the plan and no more reactive rules will be applied to that part of the current state. This could be an extremely effective way of solving problems, but would require very good communication between the executor/monitor and the planner, and the planner would need to be very flexible to cope with an ever changing initial state.

- **Interleaved planning and reacting:**The final possibility for the planner is that it limit the depth of how far it will try and plan, and when that depth is reached, the best current plan would be given to the executor/monitor. From here, reactive rules will be examined to determine if there is any rule that will change the current state to satisfy a subgoal of the plan. Once the plan's subgoals are satisfied the plan is complete. If not all subgoals are satisfied, then the planner extends the planning depth for those subgoals that were not satisfied, and the process is repeated.

- **Reactive rules as heuristic guide:**Any of the above planners could, in addition to their other methods of solving, use the reactive rules as a heuristic for selecting which action to try first when creating a plan. If the reactive rules are correct, then the planner could avoid exhaustively attempting to solve a subgoal with poor choices of action, when the correct action is suggested by the reactive rules.

- **Multiple plans:**Another approach that is similar to the above is to produce several different plans, if possible, and use the reactive rules to choose which plan to follow now, given the next action suggested by each plan.

# Chapter 2

# System Overview

To investigate the potential of a combined standard planning and reactive planning system, three key components are needed.

- A standard planner — CPRPlan, whose role is to produce the best possible plan, given the STRIPS rules available. The planner is a greedy varient of a Partial Order planner, capable of creating conflict-prone tree-like plans.
- A reactive planner — CPRReact, whose role is to provide potentially benificial actions, given the current state of the world and a set of outstanding goals and subgoals from the plan.
- A Plan Executor — CPRExec, whose role is to execute the provided plan inside the CraneWorld environment, and call upon CPRReact when that plan fails to suggest a feasible action.

2.1 gives an overview of the system components and their relationships.

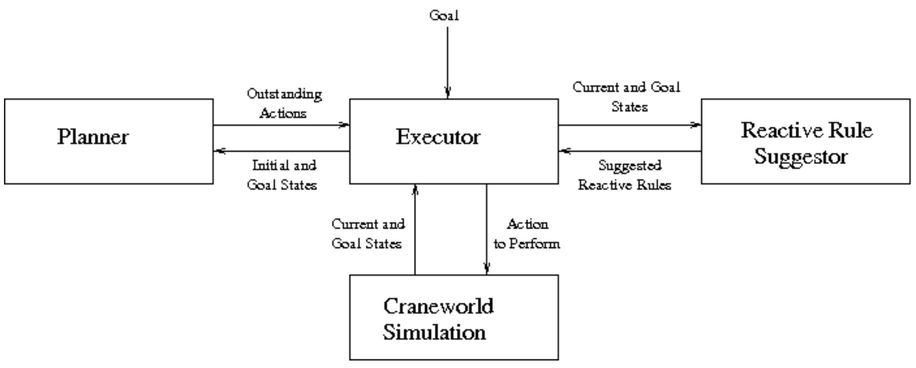

Figure 2.1: System Overview

The key approach in the design is to make limited, greedy, efficient planners which, by themselves, would be quite inadqequate, but in combination, can solve more difficult tasks than the components could individually.

The following three chapters provide detailed descriptions of each system component.

Chapter 3

# Planner — CPRPlan

## 3.1 Planner Approach

The main aim of the planner is not to create perfect plans everytime, but instead to produce a mostly complete plan in a short amount of time, even with incomplete knowledge. It should be understood that these plans can potentially contain conflicting actions and incomplete branches but, with the inclusion of reactive planning, these defects can be resolved without the need of time intensive planning techniques such as backtracking and conflict resolution.

CPRPlan is a simplified Partial Order planner, creating a tree-like plan with each subgoal creating a new branch in the plan. CPRPlan is greedy, in that it chooses an action to satisfy a subgoal with the assumption that this is the right choice, and does not backtrack if the choice is subsequently proved to be poor.

There are four major areas that must be addessed in the design of CPRPlan:

- The representation of possible actions,
- The representation of states in the world,
- The representation of the contructed plan,
- The algorithm for creating a plan from an initial state to a goal state.

## 3.2 Action Representation

The action representation used in CPRPlan is that of STRIPS rules. A STRIPS rule consists of four parts:

- Action name and parameters. Action name is the name of the action, e.g. Pickup(...), while action parameters describe upon which object(s) the action is being performed. For example, Pickup(spoon1) where Pickup(...) is an action, and spoon1 is the parameter being used in the Pickup action.
- Preconditions. Preconditions consist of a set of terms which must hold true in the state of the world for the action to be possible. For example, if the precondition of Pickup(x) was that the object x must not be under or supporting anything, then a precondition of Pickup(x) would be Clear(x).
- Add Effects. Add effects describe the ways in which the state of the world will be altered by the action being performed, in particular those alterations which add a fact to the state of the world. For example, by performing the Pickup(x) action, the state of

the world would now need to include the fact that x is now being held, and hence the fact Holding(x) would be added to the state of the world.

- Delete Effects. Delete effects describe the ways in which the state of the world will be altered by the action being performed, in particular those alterations which remove a fact from the state of the world. For example, by performing the Pickup(x) action, the state of the world would now need to include the fact x is no longer on y, and hence the fact On(x,y) would be removed from the state of the world.

For example, the complete STRIPS rule for Pickup is as follows:

```
# Pickup X from Y
Action: Pickup(X,Y)
Preconditions: clear(X)
Add: holding(X), clear(Y)
Delete: clear(X), on(X,Y)
```

In CPRPlan, the rules contain all of the above described components, although the delete effects are not currently used. This is because no action currently requires that a fact about the State of the World be not true, and because the planner is greedy and does not check for conflicts between parts of the plan — where one action deletes facts needed by another action. This may not always be the case, as a different planning algorithm could quite conceivably replace the current algorithm, one which might require the delete effects in order to plan successfully.

## 3.3 State Representation

The state of the world is represented as a collection of facts which hold true at the current time. The facts which make up a description of the world include types of objects, properties of those objects, and relationships between objects. For example, the description of a simple world which includes a blue plate on a brown table would look like this:

```
Type(plate1,Plate), Type(table1,Table), Colour(plate1, blue),
Colour(table1, brown), On(plate1, table1), Clear(plate1).
```

Although there are three types of facts, the representation used within CPRPlan does not distinguish in any way between the three types of facts available. This allows for simple and consistant representation, but does make representing a few actions awkward.

It is important to note that the CraneWorld simulation in which the plan is executed uses a different state representation. The representation in the CraneWorld is object centered. For example, a world which includes a blue plate on a brown table:

```
plate1: {Type=[Plate], Colour=[blue]} {On=[table1]}
table1: {Type=[Table], Colour=[brown]} {In=[kitchen1], Supports=[plate1]}
```

In essence the difference between the two representations is that the first describes facts about the World, while the second describes each object as a set of properties and relationships. It will be necessary to translate between the two methods of representation in order to implement the reactive planner(CPRReact), and to be able to check if the plan created for a specific goal has reached that goal state. The translation process is described in the section on the executor(CPRExec).

## 3.4 Plan Representation

The plan representation used within the planner is a tree, with the goal state as the root node and the children being the actions who satisfy subgoals of the parent, continuing until either the parent's requirements are satisfied by the current state, or there is no way they can be satisfied.

## 3.5 Building The Plan

The algorithm used to build a plan from the initial state to the goal state is a breadth first backwards search, building the plan from the goal state back towards the initial state. Starting at the goal, each subgoal is satisfied by either the initial state, or the first action whose effects satisfies the subgoal. If the subgoal was satisfied by an action, that action is added to the goal queue. Once the goal state has been satisfied with actions, the head of the goal queue is removed, and an attempt is made to satisfy all of its preconditions — subgoals. This continues, with new actions being added and satisfied actions being removed, until the queue is empty, at which point all branches of the plan are either satisfied by the initial state, or are unsatisfiable.

During planning each subgoal requiring satisfaction is compared with all available actions, and with the initial state. If an action does satisfy the subgoal it is connected to it, via the satisfying term. A subgoal can be satisfied by an action if an effect of that action is the same as the subgoal, eg if the subgoal is Holding(x), then the action Pickup(x) will satisfy the subgoal, as one of the effects of Pickup(x) is Holding(x).

Pseudo-code for the algorithm described above is as follows:

```
Push goal state onto goal queue;
While objects remain in goal queue {
  Remove object from head of goal queue;
  For each unsatisfied subgoal of object {
    Attempt to satisfied subgoal with initial state;
    If subgoal unsatisfied {
      Find first action which satisfies subgoal;
      Push satisfying action onto goal queue;
    }
  }
}
```

For example, a simple plan generated by this algorithm for the goal of holding two forks which currently lie in a closed drawer might be:

```
Subgoals:                        Holding(fork1) Holding(fork2)
                                /                             \
Effects:                  Holding(fork1)                Holding(fork2)
Actions:                  Pickup(fork1)                  Pickup(fork2)
Preconditions:In(fork1, drawer1) Open(drawer1)   Open(drawer1) In(fork2, drawer1)
                 |                      |               |               |
Effects:         |                Open(drawer1)   Open(drawer1)         |
Actions:         |                Open(drawer1)   Open(drawer1)         |
Preconditions:   |               Closed(drawer1) Closed(drawer1)        |
                 |                         \          /                 |
Initial State:In(fork1, drawer1)          Closed(drawer1)      In(fork2, drawer1)
```

## 3.6 Using The Plan

When the executor requests actions to perform, the planner could simply return the plan in a tree-structure. However, it is more convenient for the planner to return the plan as a set or list of actions which are present in the plan. There are several ways the list of actions can be retrieved, and each way provides a different ordering of actions to perform.

The list of actions to be performed can be generated by retrieving only the leaf nodes of the tree. This returns actions which are very likely to be performable, and was initially the method used. However, not including all actions in the list elimintates the possibility of serendipitous acting, where an action higher up the tree is possible to perform before all the actions assigned to satisfy that action's subgoals have been performed. This situation can readily occur when two branches of a plan deal with similar objects, such as the plan shown above, where getting the two forks from the same drawer results in a plan which contains Open(drawer1) twice.

The list of actions to be performed could also be generated by retrieving all the actions in a breadth-first manner, such that all leaf nodes are added first, followed by the parent nodes of those leaf nodes, and so on up to the root node. This method results in the complete set of actions required to fulfil the plan, and allows for seredipitous acting. However, when executing this list of actions, the executor will be jumping between branches of the plan frequently. Given the lack of conflict-checking in the planner, this jumping is quite likely to result in the subgoal's of one branch being satisfied, and then undone as another branch's subgoal's are satisfied.

One last method of generating the list of actions to perform is to traverse the plan using either pre-order traversal, or post-order traversal. This depth-first ordering results in the list of actions being grouped by branch, reducing the potential for conflict during execution. Post-order traversal is currently used, as this provides the grouping by branch, with the actions within each branch ordered leaf first, then parent. Pre-order traversal would be better in principle, but interacts in unfortunate ways with the current implementation of CraneWorld, which permits infeasible actions and then fails on them silently.

## 3.7 Planner Issues

There were two additional issues faced when designing and implementing the planner. The first is the ordering of subgoals during the satisfaction process. Each action has a set of preconditions, which must be fufilled before the action is possible to undertake. Although the planner is greedy and ignores conflicts between branches, the presence of unbound variables in subgoals means that the order in which subgoals are satisfied matters. The optimum order for satisfying these preconditions can change during the satisfacion process, based upon which variables have been bound, and which are still unbound. Due to the greedy nature of the planner, it is entirely possible to satisfy some subgoal of an action, binding a variable in the process, and as a side effect, binding the same variable in another subgoal in such a way that it becomes more difficult or impossible to satisfy. The following example illustrates the problem.

The current state of the world:

```
Clear(chair1) Clear(spoon1) On(spoon1,table1) Supports(table1,spoon1)
```

The goal state:

```
Clear(table1)
```

Possible Actions:

```
Action: Pickup(x,y)
Effects: Clear(y) Holding(x)
Preconditions: Clear(x) On(x,y)
```

The planner selects the Pickup action in order to make Clear(table1) true, binding the second parameter of Pickup to table1:

```
Action: Pickup(x,table1)
Effects: Clear(table1) Holding(x)
Preconditions: Clear(x) On(x,table1)
```

The planner must now satisfy the preconditions of the Pickup action. If it first chooses to work on the Clear(x) precondition, which has an unbound variable, it may satisfy it using the chair1 object, which happens to be clear. The resulting Pickup action is:

```
Action: Pickup(chair1,table1)
Effects: Clear(table1) Holding(chair1)
Requirements: Clear(chair1) On(chair1,table1)
```

As can be seen, the final instantiation of the Pickup action does not in fact help — as chair1 is not on table1, taking it off will not further the solving of the initial goal of Clear(table1). In addition, the planner would waste further effort, placing chair1 on table1 in order to satisfy the preconditions of Pickup(chair1,table1).

To deal with this, two heuristics were introduced to determine which subgoal should be satisfied next. The first heuristic is to first satisfy those subgoals which can only be satisfied by the initial state, as there are no actions which can satisfy them. The second heuristic is to satisfy those subgoals which are most 'complete' first. A subgoal is 0% complete if none of the variables involved in the subgoal are bound, and 100% complete if all the subgoal's variables are bound. To illustrate the difference this makes, here is the same example, but using the two heuristics.

The current state of the world:

```
Clear(chair1) Clear(spoon1) On(spoon1,table1) Supports(table1,spoon1)
```

The goal state:

```
Clear(table1)
```

Possible Actions:

```
Action: Pickup(x,y)
Effects: Clear(y) Holding(x)
Preconditions: Clear(x) On(x,y)
```

The planner selects the Pickup action in order to make Clear(table1) true, binding the second parameter of Pickup to table1:

```
Action: Pickup(x,table1)
Effects: Clear(table1) Holding(x)
Preconditions: Clear(x) On(x,table1)
```

The planner must now satisfy the preconditions of the Pickup action. Using the heuristics described above, the most complete precondition is determined to be On(x,table1), and so On(x,table1) is choosen to be satisfied first. There is only one object on the table, spoon1, and so x is bound to spoon1. The variable bindings result in:

```
Action: Pickup(spoon1,table1)
Effects: Clear(table1) Holding(spoon1)
Requirements: Clear(spoon1) On(spoon1,table1)
```

As can be seen, these heuristics result in an action which is useful for achieving the real goal of having Clear(table1) true.

The second issued was that of partial constraints. When satisfying some requirement, such as Type(x,Fork), which we can satisfy with more than one different object, such as fork1 or fork2, we can immediately choose one object over another but this may lead to conflict if we choose the wrong object, such as attempting to satisfying several mutually exclusive subgoals with fork1. It it more beneficial to neither choose neither, but constrain the possible bindings of the variable to those objects which could be used, and leave the choice of which object is most appropriate to later steps in the planner, or even to the executor. To illustrate, consider the following example.

The current state of the world, where two forks are on a bench:

```
Type(table1,Table) Type(fork1,Fork) Type(fork2,Fork)
Type(bench1,Bench) On(fork1, bench1) On(fork2,bench1)
```

The goal is to place two forks on the table:

```
On(x,table1) On(y,table1) Type(x,Fork) Type(y,Fork)
```

A potential plan:

```
Goal:    On(x,table1) On(y,table1)
           /               \
    PutDown(x,table1)  PutDown(y,table1)
           |                |
       Pickup(x)          Pickup(y)
```

By immediately selecting which object to use, and realizing that neither branch will check for conflict with another branch when planning, it is quite possible that both x and y will resolve to be fork1. In this case fork1 will be picked up, put down on table1, picked up again, and put down on table1 again. This does not solve the goal of having two forks placed on the table.

If instead we leave object selection up to the executor, both x and y can be resolved to 'either fork1 or fork2' in the plan. The executor could then resolve x to fork1 and y to fork2, and thus solve the goal correctly.

## 3.8 Planner Implementation

The implementation of the planner was a fairly complicated matter, due to the complex data-structures it contains and the behaviour required of it. The planner is required to produce plans, which are composed of actions connected together.

3.1 shows a UML diagram of the classes in the planner. Each Action object is composed of two groups of Terms: Effects and Prerequesites. The group of Terms contained in the

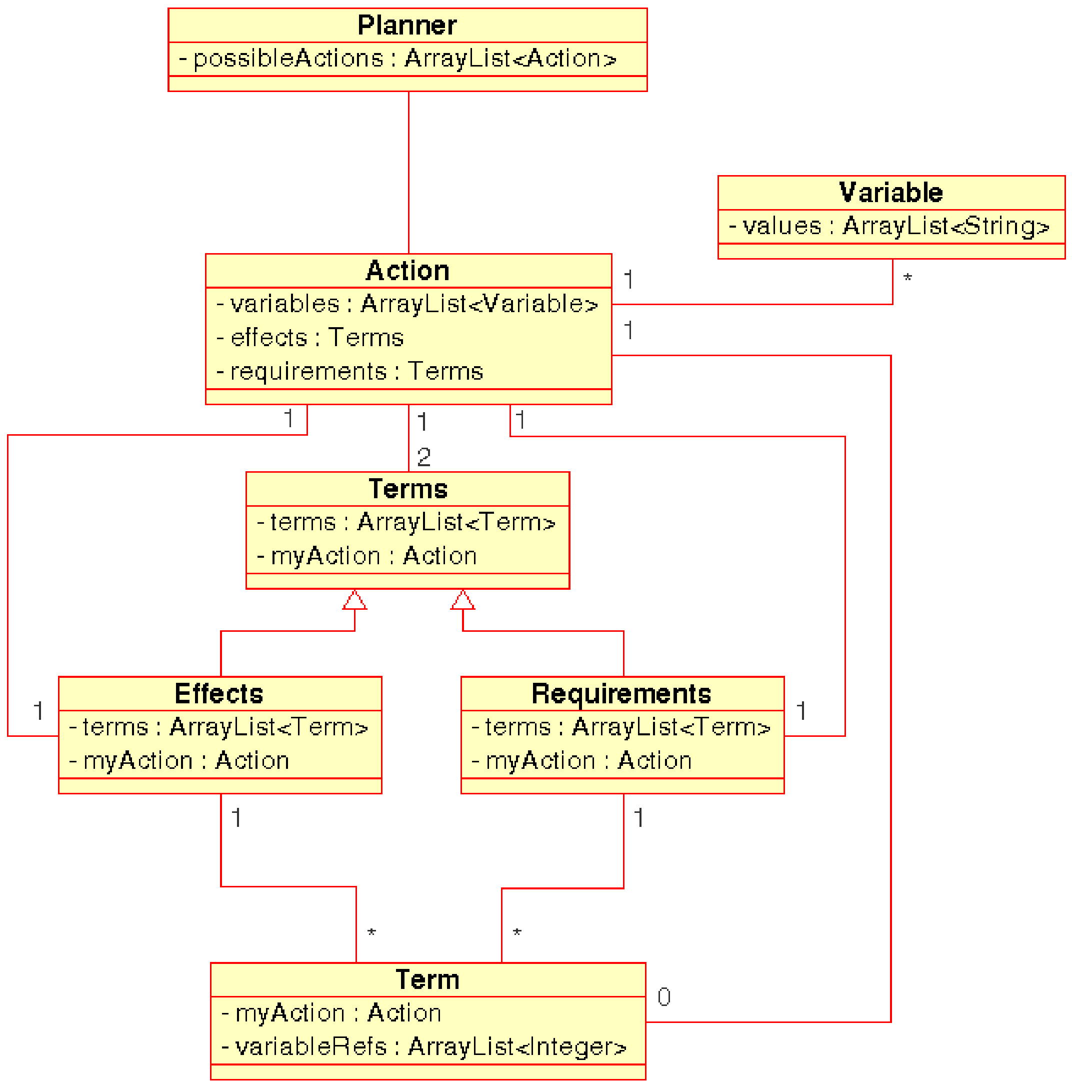
Planner
- possibleActions : ArrayList<Action>
Variable
- values : ArrayList<String>
Action
- variables : ArrayList<Variable>
- effects : Terms
- requirements : Terms
Terms
- terms : ArrayList<Term>
- myAction : Action
Effects
- terms : ArrayList<Term>
- myAction : Action
Requirements
- terms : ArrayList<Term>
- myAction : Action
Term
- myAction : Action
- variableRefs : ArrayList<Integer>
1
*
1
1
1
1
2
1
1
1
1
*
*
0

Figure 3.1: Class Structure of the Planner

Prerequesites consists of all those facts which must be satisfied before the action can be performed, while the group of Terms contained in the Effects consists of all those facts which will be made true by performing the action. A Term represents one fact about the world, such as Clear(table1). Each Term is connected to one or more Variable, which reside in the Action object. Several Terms in an Action can point to the same Variable, and when one of those Term's binds the Variable to a value, this will affect all the references to that Variable, and not just the Term in question.

When a sub-goal is satisfied by an Action, the Variables values' of both the satisfied and satisfying Terms are considered. If both are unset or both are set, then there is no change. If one is set and the other is not, the values of the set Variable are propogated to the unset Variable. Additional Variable binding is needed, however, in the form of propogating newly set values back up the tree to those Variables whose value should also change. To illustrate this, consider a simple plan with two actions.

```
   On(x,table1)
        |
 PutDown(x,table1)
```

```
    |
Pickup(x)
```

When Pickup(x) is satisfied, x becomes bound. In addition to this, the x Variables refered to in PutDown(x,table1) and On(x,table1) also need to be bound. The current implementation unfortunately lacks this functionality. However the effect of this is minimal as all goals coming from CraneWorld are fully bound, and so Variable bindings tend to travel down the tree as actions are added, rather than up the tree as is the case in the example.

As mentioned in the Planner Issues section, an improvement over commiting immediately to object selection is to instead merely constrain the possible values to those that are possible, and allow the executor to decide which object to use. The two areas this affects are the setting of Variables and the storage of Variable values. When setting Variables, if both the satisfied and satisfying Terms' Variables are set, then the intersection of the two Variable values is found, and the result replaces the previous values. To store the values of Variables, a list of strings is now required, rather than a simple string field.

Chapter 4

# Reactive Planner — CPRReact

The key feature of reactive planners is their simplicity. The reactive planner simply finds an applicable action and performs it.

Reactive rules, in general, are very powerful since they are equivalent to production rules, which are turing equivalent. However, we assume that the reactive rules have been learned from experience, and so are simple, and neither complete nor entirely reliable.

Reactive rules, as described earlier, consist of a current state, a goal state, and a related action. A reactive rule is applicable if the reactive rule's current state is a sub-state of the current state of the world, and the reactive rule's goal state is a sub-state of the provided goal state. If both these conditions are met, the action associated with the reactive rule is potentially beneficial in moving towards the goal state, and should be performed.

The purpose of CPRReact is to accept a current state and goal state from the Executor, and to provide actions which are likely to advance the current state towards the goal state. The current state used in CPRReact is that of the current state in the CraneWorld simulation. The goal states used are the overall goal state of CraneWorld, as well as the requirements of each action in the plan.

## 4.1 Reactive Rule Representation

Reactive rules have three components.

- The required current state the world must be in for the action to be possible.
- The requred goal state which must be present for the action to be useful.
- The action to perform, if the current and goal states are true.

Both the current and goal states of reactive rules are represented in the same manner as described in the Planner section. The action is represented by a string, containing the name of the action and that actions arguments. The following are some example reactive rules.

When a container is locked, and the goal is to be holding that which is inside, pickup any keys available, as they might unlock the container.

```
Current State: Type(x,Key) Locked(y) In(z,y)
Goal State:    Holding(z)
Action:        Pickup(x)
```

If holding a dirty object, place it in the sink for cleaning.

```
Current State: Holding(x) UnClean(x)
Goal State:
Action:        PutDown(x,sink1)
```

## 4.2 Reactive Rule Suggestor Issues

One additional issue was faced during the design and implementation of CPRReact. When CPRReact is required to suggest an action to perform, CPRReact currently returns only the action associated with the first reactive rule to be satisfied by the goal and current states. However, this action may not always be performable in the current state of the world, and so no reactive rules are applied and the plan is simply remade. An improvement to this would be to return not just one action, but the actions associated with all reactive rules that are satisfied by the current and goal states, thereby increasing the chance that one of the provided actions will be performable in the current state of the world.

# Chapter 5

# Executor — CPRExec

## 5.1 Executor Approach

The role of the executor is to execute the plan created by CPRPlan. If the plan fails at some point before the goal is reached, the executor must take reasonable action, whether this be replanning from the current state, or requesting reactive rules from CPRReact.

Since the plans created by this system are to be executed in the CraneWorld simulation, the executor must interact in some way with the CraneWorld simulation. For this reason, CPRExec is in fact a controlling agent within the CraneWorld simulation. When the agent is required to perform an action, CraneWorld provides a set of plausible actions that the executor must choose from. The executor performs the first action provided by the planner that is in the list of plausible actions. If there is no feasible action, the executor must choose a sensible alternative response.

## 5.2 Executor Algorithm

The main goal of the executor is to perform the actions present in the plan, and when unable to do so, perform reactive rules instead.

It is important to consider the behaviour of the executor when no actions in a plan are executible, and the CraneWorld simulation has not arrived at the goal state. There are three alternative behaviours to consider here.

- Replan given the current state of the world, and the goal state. This might be considered the conventional approach, throwing away a 'bad' plan and replanning from the current state. However, while this may work in situations where the plan has been partially executed before failing, this is not always the case. The system is intended to work with rulesets which are incorrect — such as rules missing requirements — or incomplete — such as rules missing entirely. In these cases it is quite possible for the planner to be unable to even partially execute a plan, and so each replan will result in identical plans, which fail identically. In addition to this, replanning can throw away potentially good plans, thus wasting effort.

- Perform reactive rules until the plan can be executed again. This approach takes advantage of the reactive rules to advance the current state. By using reactive rules, the resulting state is likely to be closer to the goal state then the previous state was, and the plan may well be once again executible. This method was used by the executor initially, but poor execution resulted, as the executor would undo results of reactive rules when the initial plan continued execution.

- Perform a reactive rule and then replan, given the new state of the world and the goal state. This hybrid approach takes advantage of the strengths of replanning, while also reducing the impact of incomplete or incorrect rulesets with the execution of a relevant reactive rule. This single reactive rule advances the current state, thereby avoiding the potential for constantly producing the same 'bad' plan. This is the approach currently used in the executor, and performs quite effectively.

The algorithm, therefore, consists of an attempt first to perform any of the actions present in the plan. If no action is possible, the executor creates a list of goals and provides this list to the reactive planner. The reactive planner then returns an action which will be potentially beneficial to achieving one of the goals in the list provided. With due consideration to the discussion above, replanning is performed both after a reactive rule is performed, and when neither action nor reactive rule is possible.

```
While goal is not satisfied {
  For each action in the plan {
    If the action is performable {
      Remove action from plan;
      Perform action;
      Repeat from top;
    }
  }
  #If no action in the plan is possible:
  Extract a list of outstanding goals from plan;
  For each goal in the list {
    If there is an appropriate reactive rule {
      If reactive rule is performable {
        Perform associated action;
        Replan;
        Repeat from top;
      }
    }
  }
  #If no reactive rules are possible:
  Perform the 'DoNothing' action;
  Replan;
  Repeat from top;
}
```

## 5.3 Craneworld Integration

As described earlier, CraneWorld is a simulation of a world in which an agent can move and act. All input to the world comes in the form of actions performed by that agent, and so the executor resides within an agent and contains both the planner, and the reactive planner.

At initialization, the executor retrieves the current state of the world from CraneWorld, along with the goal state, and provides these to the planner. The planner then creates a plan to get from the current state to the goal state, given the actions that the planner knows about. It is important to realize that it is quite possible that the planner either does not know about some action which is vital in achieving the goal state, or that one or more actions that it does know about are in fact incorrect in some way. It is because of this that execution of a plan is not always as simple as performing each action in the plan one after another.

## 5.4 State Translation

As detailed earlier, the state representations of CraneWorld and the planner differ. CraneWorld describes the world as a set of objects, with properties, and relationships with other objects. The state representation in the planner is a collection of facts about the world, with no objects explicitly defined, but rather implied through their presence in the facts. Converting the states from CraneWorld representation to planner representation is necessary, as planning with CraneWorld states would complicate the planning process.

The process of conversion is fairly straightforward. For each object in the CraneWorld representation, the set of facts related to that object are extracted from the properties and relationships, creating fact-tuples of predicate(object) or predicate(object,value). Boolean properties in the CraneWorld object description, such as Open and Clean, are translated to predicate(object) facts. All other properties, and all relationships, are translated to predicate(object,value) facts. To illustrate this, consider the following example.

A CraneWorld state, in which there is a plate and a door.

```
plate1: {Type=[Plate] Clean=[Yes]} {In=[kitchen1] On=[table1]}
door1:  {Type=[Door] Colour=[brown] Open=[No]} {In=[kitchen1]}
```

The same state, represented in CPRPlan format, as a collection of facts.

```
Type(plate1,Plate) Clean(plate1) In(plate1, kitchen1) On(plate1,table1)
Type(door1, Door) Colour(door1,brown) UnOpen(door1) In(door1,kitchen)
```

Initially the CraneWorld states themselves were required to perform the conversion into planner states. This approach was untidy however, as any change in simulation environments would require this functionality to be added to each new simulation. Instead, the conversion process was extracted into a class of it's own, which accepts a CraneWorld state, and returns a planner state.

A further consideration to the conversion process was whether to assume implicit facts. If an object in the CraneWorld simulation does no support any objects, it can be considered to be clear. This is not explicitly stated, but is implicit in the lack of the On relationship. For some states, such as the initial state in the planner, this implicit fact must be explicitly stated to facilitate proper planning. In other states where the state only represents those properties and relationships which are important, such as the goal state, the inclusion of implicit facts introduces additional, and potentially incorrect, constraints which in fact hinder planning. It was therefore necessary to introduce two slightly different methods of state conversion, one which includes implicit facts and one which does not.

A final consideration to the conversion process was the special relationship of holding. In the CraneWorld simulation, if the agent picks an object up, it is implicitly understood that the agent is now holding that object. This is represented by placing the object in the agent. As this is prepresented no differently than any other In relationship, special consideration was taken to convert all of the In relationships contained in the agent into Holding facts.

## 5.5 Changes to CraneWorld

Two main additions to CraneWorld were made. The first was the addition of a goal state to the world. This state was loaded in as a CraneWorld state at initialization, and was present to allow the executor to determine whether the goal state had been reached. This goal state was also converted and used in the planner and reactive rule suggestor as the goal state.

The second addition to CraneWorld was the ability to compare states and determine if one was a sub-state of another. This was necessary to determine if the goal was a sub-state of the current state. The goal is, in itself, a sub-state as it does not necessarily contain all the information about the world, but only those details which are important.

## 5.6 Action Ordering

A further issue faced when designing and implementing CPRExec was the execution ordering of plan actions. As described earlier in the planner section, when the executor requests a list of actions from the planner, a heuristically ordered list is returned. However, suppose the first branch of the plan is unexecutable in the current state of the world, and so the executor moves to the next branch of the plan and executes the first available action. In the new current state, it may now be possible to execute the first branch of the plan, and in doing so undo some important effect of the previous action. This will then prevent the plan from being executed completely. A solution to this problem is to move the branch which has most recently been acted upon to the head of the list of actions, thereby reducing the potential for conflicting plan branches to undo one another. This is unfortunately not currently implemented, but the implementation of such a heuristic would be fairly trivial.

## 5.7 Executor Implementation

The implementation of the executor is fairly simple. The executor contains the planner and the reactive rule suggestor, and responds to action requests with the most appropriate action as described earlier in this chapter. The algorithm for providing the most appropriate action

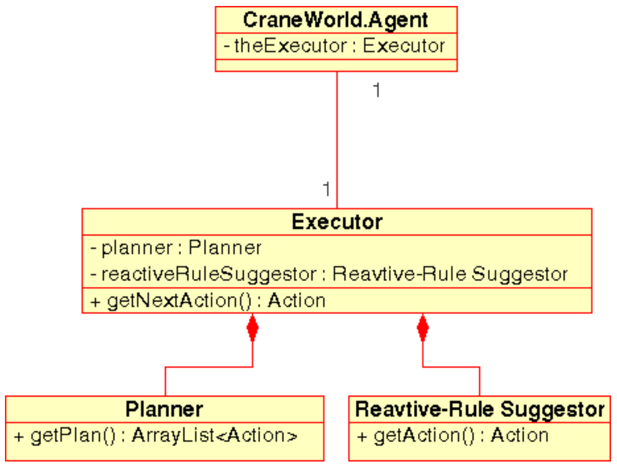

Figure 5.1: Executor

is also detailed earlier, and can be represented by the data-flow diagram shown in 5.2.

One issue that arose during the implementation of the executor was how to deal with partially constrained variables. Currently the actions from the planner are compared with CraneWorld actions through a simple string comparison. If the strings match, the actions match. However, when the plan contains an action which is only partially constrained, such as Pickup(fork1 or fork2), this simple method fails. There are two potential approaches to deal with these partially constrained actions.

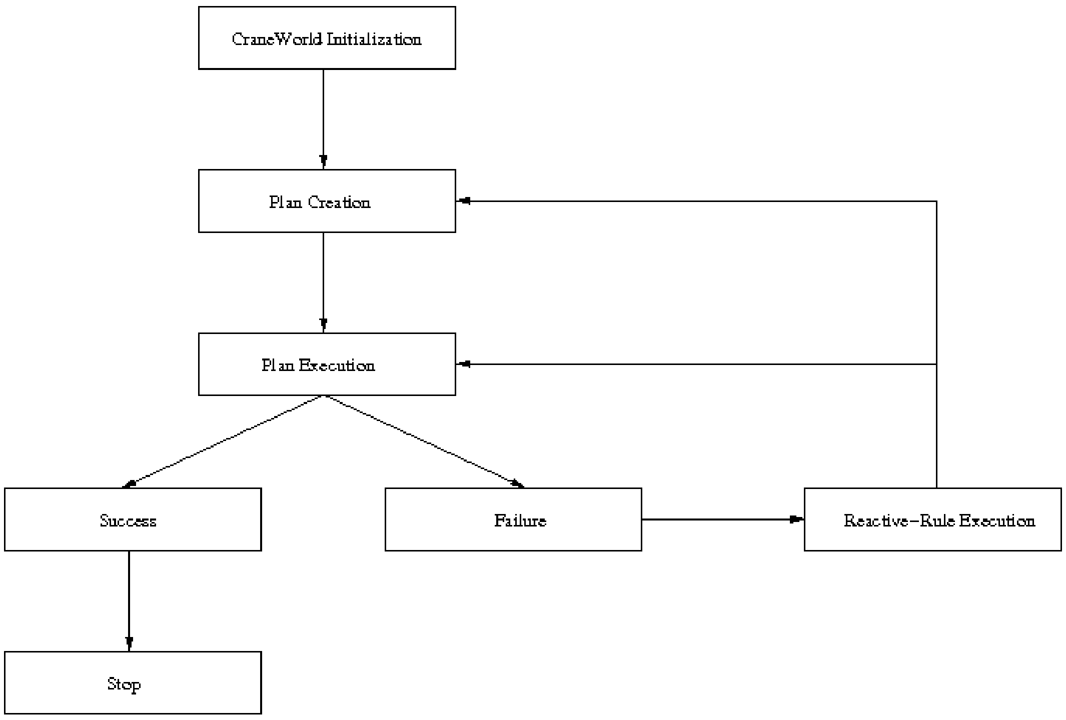

Figure 5.2: Data-Flow Diagram of Executor

- Choose the first object which is possible in the current state of the world. If the action is Pickup(fork1 or fork2) and fork2 cannot currently be picked up, then the action could be further constrained to Pickup(fork1), which is a valid action. If both can be picked up, then then the executor could behave in a greedy manner, choosing the first object with the assumption that this is correct. This approach is trivial to implement, but can potentially undo previous actions which have involved that object.

- Choose the most appropriate object, given the current state of the world. If the action is again Pickup(fork1 or fork2) and both objects are available for picking up, then the executor must consider which object should be used. If fork1 has already been moved and now satisfies some goal, it makes no sense to pick it up again and once more move it, and so fork2 should be chosen instead. This approach is more complicated to implement, as there needs to be some method for determining if an object is satisfying a goal, but would result is improved plan execution.

Currently the executor implements neither, and so unfortunately cannot make use of partially constrained variables. This is something which is vital for future work.

A further issue that arose is what the goal state provided to the reactive rule suggestor should consisted of. There are three alternative approachs.

- Provide only the main goal state of CraneWorld to the reactive rule suggestor. This conservative approach attempts to find any reactive rules which further the current state towards the overall goal state, but does not attempts to find any reactive rules which will further the execution of the current plan. While sensible, in that moving towards the overall goal state must be benificial, this approach does not take into account the current plan, and so ignoring potentially necessary itermediate steps towards the goal.

- Provide a composite goal state. This approach combines the overall goal with the sub-goals of each action in the plan. While an improvement on ignoring the plan entirely, the drawback to this approach is the potential for unbenefical actions to be suggested. Independant action's sub-goals', when combined, may mislead the reactive rule suggestor into suggesting an action which does not in fact benifit the situation.

- Provide multiple goal states. This approach presents the reactive planner with multiple goal states, one for the overall goal and one for each action in the plan. This avoids the potential for misleading the reactive planner with impossible combinations of sub-goals, while still taking due consideration of the current plan. The implementation of the reactive planner currently uses this method.

# Chapter 6

# Example Plans

This chapter describes two example tasks that the system solved, to illustrate how the it works.

## 6.1 Table Setting

The Table Setting problem is a trivial problem, chosen to demonstrate the speed of the planner when it has complete information and there are no conflicts within the plan. The goal of the Table Setting problem is to set four places at a table. A place is considered set when it contains a knife and a fork. The knives and forks begin scattered across the kitchen.

The initial state of the Table Setting problem consists of a kitchen containing a table, a bench and a dishwasher. The table contains four places. The dishwasher contains two knives and two forks. Two additional forks and knives are on the bench. Below is the CraneWorld state representing the initial state of the problem.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
bench1: {Type=[Bench]} {In=[kitchen1], Supports=[fork2, fork4, knife1, knife3]}
dishwasher1: {Type=[Container, Dishwasher]}
             {In=[kitchen1], Contains=[fork1, fork3, knife2, knife4]}
fork1: {Type=[Fork]} {In=[dishwasher1]}
fork2: {Type=[Fork]} {On=[bench1]}
fork3: {Type=[Fork]} {In=[dishwasher1]}
fork4: {Type=[Fork]} {On=[bench1]}
kitchen1: {Type=[Container, Kitchen, Room]}
          {Contains=[PlanningPete, bench1, dishwasher1, table1]}
knife1: {Type=[Knife]} {On=[bench1]}
knife2: {Type=[Knife]} {In=[dishwasher1]}
knife3: {Type=[Knife]} {On=[bench1]}
knife4: {Type=[Knife]} {In=[dishwasher1]}
place1: {Type=[Container, Place], Open=[Yes]} {On=[table1]}
place2: {Type=[Container, Place], Open=[Yes]} {On=[table1]}
place3: {Type=[Container, Place], Open=[Yes]} {On=[table1]}
place4: {Type=[Container, Place], Open=[Yes]} {On=[table1]}
table1: {Type=[Table]} {In=[kitchen1], Supports=[place1, place2, place3, place4]}
```

The goal state of the Table Setting problem consists of a kitchen containing a table. The table contains four places, and each place contains a knife and a fork. Below is the CraneWorld state representing the goal state of the problem.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
fork1: {Type=[Fork]} {In=[place1]}
fork2: {Type=[Fork]} {In=[place2]}
```

```
fork3: {Type=[Fork]} {In=[place3]}
fork4: {Type=[Fork]} {In=[place4]}
kitchen1: {Type=[Container, Kitchen, Room]} {Contains=[PlanningPete, table1]}
knife1: {Type=[Knife]} {In=[place1]}
knife2: {Type=[Knife]} {In=[place2]}
knife3: {Type=[Knife]} {In=[place3]}
knife4: {Type=[Knife]} {In=[place4]}
place1: {Type=[Container, Place]} {On=[table1], Contains=[fork1, knife1]}
place2: {Type=[Container, Place]} {On=[table1], Contains=[fork2, knife2]}
place3: {Type=[Container, Place]} {On=[table1], Contains=[fork3, knife3]}
place4: {Type=[Container, Place]} {On=[table1], Contains=[fork4, knife4]}
table1: {Type=[Table]} {Supports=[place1, place2, place3, place4], In=[kitchen1]}
```

The actions available to the planner were Pickup(x) and PutDown(x,y). Note, unlike traditional Blocks World problems, the agent can holding multiple objects at one time.

```
#Pickup(x) - Pickup object x
Action: Pickup(x)
Effects: Holding(x)
Preconditions: Clear(x)

#PutDown(x,y) - PutDown object x on object y
Action: PutDown(x,y)
Effects: On(x,y) Clear(y)
Preconditions: Holding(x)
```

It is immediately clear that the plan required to solve this problem is very simple. No conflicts between objects are present, and the only actions required, Pickup and PutDown, are available. Unsurprising, the plan — 6.1created is both straightforward, and correct. The time taken to create this simple plan was 39milliseconds, which was comparable with the time taken for a GoalStack planner to perform the same task. The execution of the plan was

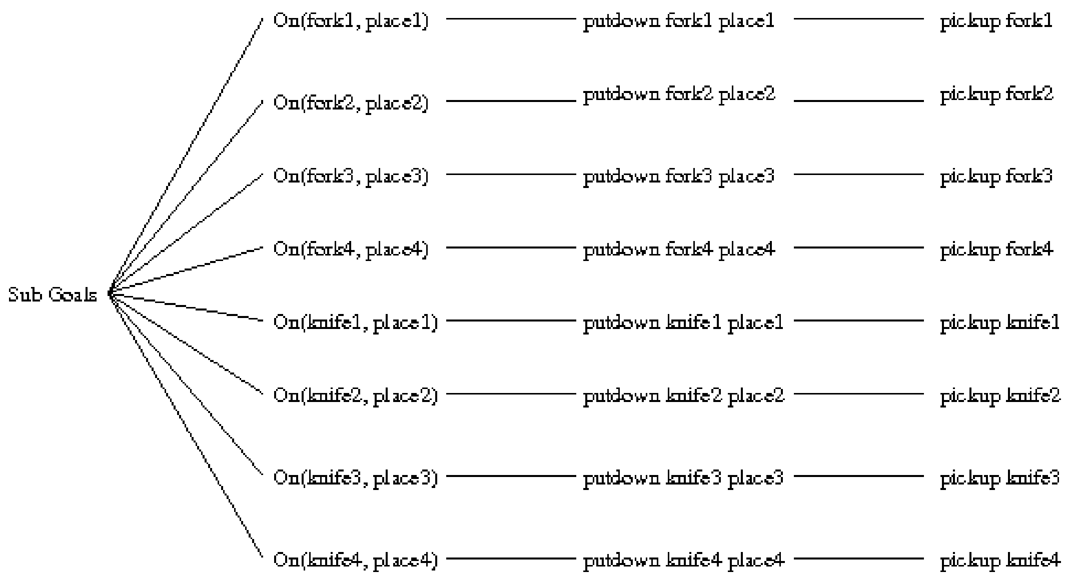

Figure 6.1: Table Setting Plan

also an uncomplicated matter — no branch of the plan conflicted with any other, and there was complete knowledge. No reactive rules were required during the execution.

## 6.2 Blocks World

The Blocks World problem is a much more complex problem, with a large potential for conflicting plan branches. The Blocks World problem was chosen to demonstrate the system's ability to cope with complex situations. The goal of the Blocks World problem is to stack four blocks on top of each other.

The initial state of the Blocks World problem consists of four blocks in two seperate stacks, on a table. Blocks A and B are on the table. Block C is stacked on block A, and block D is stacked on block B. Below is the CraneWorld state representing the initial state of the problem.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
blockA: {Type=[Block]} {In=[kitchen1], Supports=[blockC], On=[table1]}
blockB: {Type=[Block]} {In=[kitchen1], Supports=[blockD], On=[table1]}
blockC: {Type=[Block]} {In=[kitchen1], On=[blockA]}
blockD: {Type=[Block]} {In=[kitchen1], On=[blockB]}
kitchen1: {Type=[Container, Kitchen, Room]}
          {Contains=[PlanningPete, blockA, blockB, blockC, blockD, table1]}
table1: {Type=[Table]} {In=[kitchen1], Supports=[blockA, blockB]}
```

The goal state of the Blocks World problem consists of the same four blocks and the table, but Block D is now on the table, block C is stacked on block D, block B is stacked on block C, and block A is stacked on block B. Below is the CraneWorld state representing the goal state of the problem.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
blockA: {Type=[Block]} {On=[blockB]}
blockB: {Type=[Block]} {On=[blockC], Supports=[blockA]}
blockC: {Type=[Block]} {On=[blockD], Supports=[blockB]}
blockD: {Type=[Block]} {On=[table1], Supports=[blockC]}
kitchen1: {Type=[Container, Kitchen, Room]} {Contains=[PlanningPete, table1]}
table1: {Type=[Table]} {In=[kitchen1], Supports=[blockD]}
```

The actions available to the planner were the same Pickup(x) and PutDown(x,y) as in the previous example. The reactive rule available to the reactive rule suggestor was PutDown(x,y)

```
#PutDown(x,y) - PutDown object x on object y
Action: PutDown(x,y)
Initial State: Holding(x)
Goal State: On(x,y)
```

As can be seen, there is great potential for actions within the plan to conflict. For example, it is possible to satisfy the requirement that block C is on block D immediately, but in doing so block A becomes trapped under both blocks. To then satisfy the requirement that block A is on block B requires that both block C and block D be picked up, undoing the effects of the previous stacking action. The plan initial generated — 6.2, has the potential to solve each of the subgoals individually, but doing so will undo other subgoals which have previously been satisfied.

This initial plan is executed, and unsurprisingly several subgoals are undone by other branches in the plan. The resulting state does not satisfy the goal, as block B is not on block C, and block C is being held. The reactive rule PutDown(blockC,blockD) is performed, and the resulting state is as follows.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
blockA: {Type=[Block]} {On=[blockB]}
```

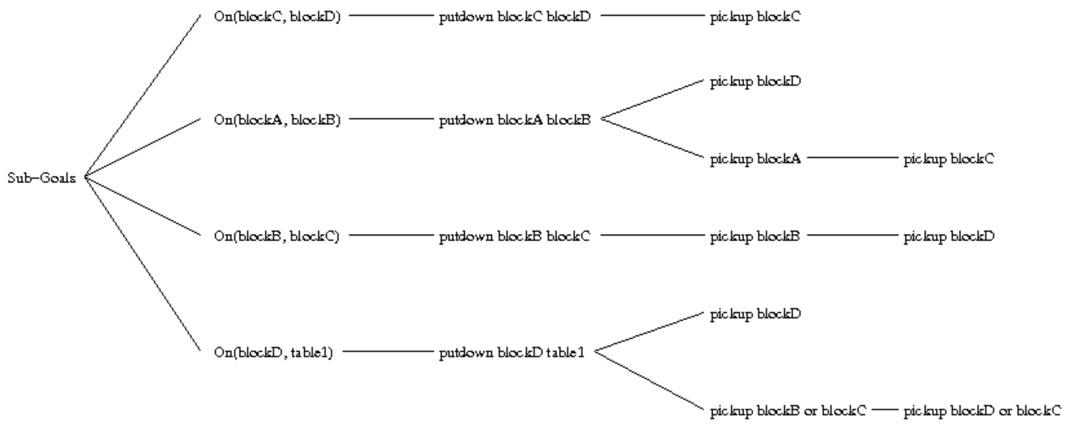

Figure 6.2: Blocks World Plan, Iteration One

```
blockB: {Type=[Block]} {On=[table1], Supports=[blockA], In=[kitchen1]}
blockC: {Type=[Block]} {On=[blockD]}
blockD: {Type=[Block]} {On=[table1], Supports=[blockC]}
kitchen1: {Type=[Container, Kitchen, Room]} {Contains=[PlanningPete, blockB, tabl
table1: {Type=[Table]} {Supports=[blockB, blockD], In=[kitchen1]}
```

As can be seen, several of the subgoals are now satisfied — On(blockD,table1), On(blockC,blockD) and On(blockA,blockB). The planner now creates a new plan — 6.3, from the current state to the goal state. This new plan is executed, satisfying the requirement On(blockB,blockC),

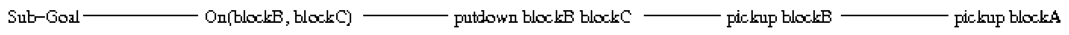

Figure 6.3: Blocks World Plan, Iteration Two

but in doing so, it undoes On(blockA,blockB). The reactive rule PutDown(blockA,blockB) is performed, and the resulting state is as follows.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
blockA: {Type=[Block]} {On=[blockB]}
blockB: {Type=[Block]} {On=[blockC], Supports=[blockA]}
blockC: {Type=[Block]} {On=[blockD], Supports=[blockB]}
blockD: {Type=[Block]} {On=[table1], Supports=[blockC]}
kitchen1: {Type=[Container, Kitchen, Room]} {Contains=[PlanningPete, table1]}
table1: {Type=[Table]} {Supports=[blockD], In=[kitchen1]}
```

The goal state is now satisfied, after two iterations of planning. In this problem, reactive rules were present, but played a fairly minor role, fulfilling trivial subgoals. This problem was designed to test the system's ability to cope with complex situations, and conflicting plans.

## 6.3 Locked Box

The Locked Box problem is a simple problem, but with lack of knowledge and incorrect rules. The Locked Box problem was chosen to demonstrate the system's ability to cope with missing information. The goal of the Locked Box problem is to be holding a crown, which is currently inside a locked box.

Below is the CraneWorld state representing the initial state of the problem.

```
PlanningPete: {Type=[Robot]} {In=[kitchen1]}
box1: {Open=[No], Locked=[Yes], Type=[Box, Container]}
      {Contains=[crown1], In=[kitchen1], LockedBy=[key1]}
crown1: {Type=[Crown]} {In=[box1]}
key1: {Type=[Key]} {Locks=[box1], In=[kitchen1]}
kitchen1: {Type=[Container, Kitchen, Room]}
          {Contains=[PlanningPete, box1, key1]}
```

Below is the CraneWorld state representing the goal state of the problem.

```
Goal State:
PlanningPete: {Type=[Robot]} {Contains=[crown1], In=[kitchen1]}
crown1: {} {}
kitchen1: {Type=[Container, Kitchen, Room]} {Contains=[PlanningPete]}
```

The actions available to the planner were the same Pickup(x) and PutDown(x,y) as in the previous examples. The reactive rules available to the reactive planner were Pickup(x), Unlock(x), and Open(x).

```
#Pickup(x) - Pickup object x
Action: Pickup(x)
Initial State: Locked(y) LockedBy(y,x) In(z,y)
Goal State: Holding(z)

#Unlock(x) - Unlock object x
Action: Unlock(x)
Initial State: Locked(x) LockedBy(x,y) In(z,x)
Goal State: Holding(z)

#Open(x) - Open object x
Action: Open(x)
Initial State: UnOpen(x) In(y,x)
Goal State: Holding(y)
```

The initial plan created consisted of Pickup(x). This is because the Pickup action is missing the preconditions required to ensure that the container which holds object x is in fact unlocked and open. This plan is impossible to perform, and so the reactive rule Pickup(key1) is performed.

The plan is recreated, but is identical to the previous plan. The plan still cannot be performed, and so another reactive rule is performed — Unlock(box1).

Once more the plan is recreated, and again the plan is identical, and a failure. The reactive rule Open(box1) is performed, and the plan is created for the last time. Pickup(crown1) is now possible and the goal is achieved.

This simple example demonstrates the system's ability to cope with rules which lack preconditions.

# Chapter 7

# Conclusion

This report has described an investigation of greedy, limited planning. Three planner components were created: a greedy planner, a reactive planner, and an executor.

The greedy planner has several key features:

- The greedy planner is capable of producing plans, even when provided with rulesets that are missing rules, or contain incorrect rules.
- The greedy planner is fast. By ignoring potential conflicts within the plan, and by selecting the first action available, the planner does not need to backtrack. However, the plans produced are conflict prone.
- The greedy planner has heuristics. By behaving in a greedy manner, the planner can potentially produce stupid plans. To reduce the impact of this, heuristics were created which guide the planner to make better choices, while still being greedy.
- The greedy planner contains variables which are to be instantiated during the execution, when there is sufficient information to make a sensible choice. It can also place partial constraints on variables to prevent premature commitment to any particular object, while still ruling out objects that are known to be inappropriate.

The reactive planner, capable of suggesting potentially beneficial actions to perform for a given situation, has two key features:

- The reactive planner can operate with minimal knowledge. The reactive planner can provide actions which will be beneficial to the current situation, even if the reactive planner does not know how the action will be helpful.
- The reactive planner's rules can be learned from experience. Although reactive rules are potentially very powerful, the rules used within this system's reactive planner are straightforward, and so are easily learned.

The executor combines both the greedy planner and the reactive planner in order to solve tasks. The approach described in this report prioritises the planner, executing until failure is detected, at which point the reactive planner takes over. Although the report focused on this approach, there are several other approaches that were outlined and which could be investigated in the future.

This report has described how the components are combined, and the example plans/tasks have shown how that combination can solve problems effectively, which neither planner could alone.

There are several limitations of the combined system.

- If the required plan must be perfect the first time, this system will be unlikely to succeed.
- If irreversable actions are present, they may be performed too early, thereby preventing the task from being solved correctly.
- The action chosen to satisfy a subgoal is the first available, and not necessarily the most appropriate.

## 7.1 Future Work

With regards to the current system, several components still require either implementation, or improvement.

- When an action is performed from the plan, the branch which contained that plan should be given priority in the plan.
- The executor must contain the functionality to correctly deal with partially constrained actions and reactive rules.
- The reactive planner should return all appropriate reactive rules for a given current state and goal state, rather than just the first.

The three experiments carried out demonstrate the system's speed, flexibility, and ability to cope with incomplete rulesets. However, more experimentation is needed.

Two heuristics could potentially increase the planner's ability to cope with complex situations. The first recommended heuristic is to choose not the first, but the most likely, action which satisfies some subgoal. This could be done by considering how many of the action's preconditions are met in the current state, and to choose the action with most preconditions already satisfied.

The second recommended heuristic is needed at the action ordering stage. In order to reduce the potential for conflicting actions to undo one another, the planner could perform a once-over conflict check, similar to the Partial Order Planner, to determine which branches should have priority.

Currently reactive rules are only applied when no action in the plan is available. Another approach is to attempt to apply reactive rules when an action is found to be unperformable. This assumes that the action is necessary for the plans execution, but does potentially deal with rules which have incorrect/incomplete preconditions.

The system's executor currently plans, executes the plan, and then performs reactive rules. There are other alternatives however, such as performing the planning and reactive rules in parallel, or performing all appropriate reactive rules first, before planning. These alternative approaches were outlined earlier.

# Bibliography

[1] S. J. Russell and P. Norvig, *Artificial Intelligence. A ModernApproach*. Prentice-Hall, 2003.

[2] M. Wojnar, “An intelligent agent that learns in complex, novel, environments,” 2004.

[3] P. Agre and D. Chapman, “Pengi: An implementation of a theory of activity,” 1987.

[4] A. Longhurst, “An agent that acts based on past experience,” 2004.