PLANKEE -
a Planner with Replanning Capability

by

Magnus Christerson
Abstract

PLANKEE is a planner which is able to do replanning when execution of the original plan has failed. In the report we first summarize the history of planners and then develop an own planner for the blocks world. The planner uses slightly modified ideas from [Sacerdoti 77]. PLANKEE is implemented in KEE†.

PLANKEE is extended to be able to replan failed plans. A new initial state is asserted from which the new plan should be applied. By associating information from the planning procedure to the operators in the plan, PLANKEE can use this information to do effective replanning. The information needed for effective replanning is (1) to divide the operators into different categories and (2) to extend the use of preconditions.

† KEE is a trademark of IntelliCorp, Inc.
Contents

Abstract ........................................................................................................................................ 3
Contents ....................................................................................................................................... 4
Preface ......................................................................................................................................... 6

1. Summary and Conclusions ...................................................................................................... 7
2. Introduction .......................................................................................................................... 9

Part I: Planning

3. The Development of Planners .............................................................................................. 12
   3.1 Means-ends Analysis and GPS ..................................................................................... 12
   3.2 Forward and Backward Searching ............................................................................... 13
   3.3 State Description and STRIPS .................................................................................... 13
   3.4 The Frame Problem ...................................................................................................... 16
   3.5 Hierarchical Plans and ABSTRIPS ............................................................................. 17
   3.6 Interacting Goals .......................................................................................................... 18
   3.7 The Non-linearity of Plans and NOAH ....................................................................... 19
   3.8 Other Approaches to Planning .................................................................................... 21

4. The Implementation of PLANKEE ...................................................................................... 23
   4.1 World Modelling ........................................................................................................... 23
   4.2 Default Rules and Truth Maintenance ...................................................................... 25
   4.3 Operators ...................................................................................................................... 26
   4.4 Control Strategy for Plan Generation ....................................................................... 27
   4.5 Representation of Plans .............................................................................................. 31
   4.6 Execution of Plans ........................................................................................................ 32

Part II: Replanning

5. To Modify and Reuse Completed Plans ............................................................................. 34
   5.1 The Triangle Table in STRIPS .................................................................................. 34
   5.2 Case-Based Planning ................................................................................................. 36
   5.3 Analogical Problem Solving ....................................................................................... 37
   5.4 Marker-Passing ........................................................................................................... 37
   5.5 Replanning Through Abstraction ............................................................................ 39
   5.6 Dependency Information ............................................................................................ 39

6. Making PLANKEE Able to Replan .................................................................................... 41
   6.1 A Dynamic World Description .................................................................................. 41
   6.2 Different Kinds of Operators and Their Intentions .................................................. 41
   6.3 Extending the Preconditions ....................................................................................... 43
   6.4 The Replanning Strategy ............................................................................................ 44
6.5 Two Replanning Examples ........................................ 45
6.6 Execution of the New Plan ..................................... 47

Appendix
A GPS - The General Problem Solver .......................... 49
B STRIPS - STanford Research Institute Problem Solver .... 50
C NOAH - Nets Of Action Hierarchies ...................... 52
D.1-F in the Prototype library, SICS/PA89/89002A ........ 54 - 85

References .................................................................. 86
Preface

This report describes a system, PLANKEE. The system is a planner and a replanner. The planning domain is the blocks world. The work was done as a Master of Science thesis in Computer Engineering at the Department of Computer Science in Lund. The actual work was done at SICS, Swedish Institute of Computer Science in the laboratory of Knowledge Based Systems.

The main work was done during the summers of '88 and '89. In '88 it was mainly theoretical studies of planning and replanning and studies in the literature of the same areas. The theory described in this report was outlined during that time. In '89 the implementation of PLANKEE was done. The theory that was developed was also tested, and, of course, slightly modified.

The main purpose of the project was twofold:

- Immersing the AI field of planning
- Testing the KEE knowledge based system tool

The first thing I noticed when studying the literature, was that AI as a practically useful science hasn't evolved very far, even though it started of in the 50's. Still today a planner of the PLANKEE type is state of the art in planning. This shows that trying to be intelligent in planning situations is extremely hard.

The next thing I noticed when trying to do it all by myself was that it is really hard to learn a system to be intelligent, or at least able to do some flexible planning. My conclusion from this is that we are far from building any truly intelligent system, at least in the planning problem space.

Finally acknowledgements. Rune Gustavsson, who is both the professor of AI at the Department of Computer Science at Lund University and the head of the KBS-laboratory at SICS, has been my advisor on this project. Our discussions have been, sometimes quite wild though, always very helpful to me, both in working with PLANKEE and otherwise. Rune's "domain knowledge" has been of much help working on this project.

I also want to thank the people at SICS for their help during the project time, and the people at the Computer Science department in Lund for helping me early in this work. The idea of attaching intentions to operators is due to Tobias Rydén.
1. Summary and Conclusions

This report describes a system, PLANKEE, that is able to develop plans in a limited world domain, the blocks world. It also has the feature of replanning when the initial state has changed and the original plan is impossible to invoke.

The report is divided in two parts, one for planning - chapter three and four - and one for replanning - chapter five and six. The first chapter in each part is a historical perspective in the field, and the second chapter describes the ability implemented in PLANKEE.

In chapter two a short discussion on the planning problem is given. It also gives a broad outline description of the PLANKEE system.

In chapter three a historical perspective on planners is given. Certain concepts that are used in the planning literature, such as hierarchical - nonhierarchical planning and linear - nonlinear planning, are explained. The main ideas of the planning research f today is also given.

Chapter four describes the planning facility of PLANKEE. We show how the knowledgebase for the blocks world is built, using the facilities KEE supports. The world modelling and the rulesystem is specially investigated. The algorithm for planning, used by PLANKEE, is outlined and explained while viewing an example in blocks world.

The next chapter, chapter five, shows some of the early attempts of replanning. We also take a look at the replanning research of today. Similar methods are used to do planning, such as Case Based Planning and Analogical Problem Solving, and some of these are also glanced upon.

Finally, in chapter six we outline a strategy for doing replanning in the blocks world. This is also implemented in the PLANKEE system. It is showed that to put more information into the operators in the plan, we can do an effective replanning. The strategy for replanning is used in two examples.

The work reported here was purposed to view on methods to do replanning in the blocks world. A method was found which put constraints to the planning information associated with the operators. Mainly it is two kinds of information needed.

Firstly, we need to know what kind an operator really is and what's its intention. We show that there are two kinds of operators. The ones with intention of satisfying a subgoal and the ones with intention of satisfying an operator's precondition. We also asserts the intention of the operators. When this information is added to the operators, it is easier to reason about them in the planning situation.

Secondly, we extend the preconditions used in most of the planners developed. The new preconditions keeps more information of when an operator is applicable. We also define protected preconditions which are preconditions specially critical to the success of the plan.

The work was also purposed of testing the KEE system. It worked very well. KEE supports a lot of features that can be used in many different situations. By using these different features in different combination, KEE gives you a very flexible system to use. Sometimes it is hard not to use too many features in too many combinations. KEE also supports the Lisp environment when needed. This makes KEE a stronger and more flexible environment to work with.
KEE is very large. When putting PLANKEE in it, it is even larger. This makes the system to sometimes run quite slowly and doing garbage collection frequently. PLANKEE was developed on a Sun with 12 Mb of memory. Even with this KEE is sometimes overloaded.

To summarize, KEE is a large system to work on. It has many features and it takes quite a while to manage all of them. (I often experienced that the reference manuals were better than the user's manual to read first.) When you think you started to manage the system, there are always feature you find later, never dreaming of them existing. This makes KEE quite fun to work with.
2. Introduction

The ability of problem solving is an important feature of every intelligent system. Problem solving is maybe that feature we associate most with artificial intelligence. An intelligent system should be able to think, whatever that is, and by doing that, solve problems. Problem solving is a very wide area. Typical for it is to find one or more solutions to a given problem. The problem domain may vary a lot. From natural language understanding to pattern recognition and vision, robotics, reasoning, learning, and planning.

Planning is maybe the most abstract part of problem solving. It is argued [Wilensky 83] that by doing effective planning we must understand the domain. Thus planning and understanding have very much in common. This means that if we understand a problem, we are also able to plan in the problem domain. Most planners implemented show this very strongly. In the planners, a lot of information about the domain are inserted. The planner relies heavily on this information when doing planning. If the information is wrong, the planner will fail.

In this report we shall look at some classical attempts of doing planning. We shall also develop a planner for a very limited domain, the blocks world. By doing this we shall see what kinds of information the planner must have to do effective planning. We shall see that the planner is very dependent upon this information.

We live in a changeable world. This means that nothing is ever constant. The world around us changes all the time. When a plan is invoked, the problem space may have changed so much since the plan was developed that it is impossible to execute. Then a new plan must be developed. One possibility is to invoke the planner again to develop a new plan according to the new world model. This is maybe the easiest way of solving the problem. Another possibility is to keep the old plan and just modify it to the new world. This is called replanning.

If the world has only changed slightly it is often more effective to use the old plan, because often most of it is ok even for the new world. Then replanning can be used for the old plan. But if the world has been through a dramatic change, it might be best to do a completely new plan. Then the planner will be invoked again. Which is the most effective of these two ways? Where is the breaking point between these two alternatives? This depends how effective the planner and replanner really is. This report will describe a system that can be used for examining this.

The report also describes some earlier attempts of doing planning. We will look at some of the big planners of the history of artificial intelligence. We will see how they are evolved from actually quite simple ideas. The report also describes some replanning systems that are built on different, sometimes more trickier ideas.

The report also describes a planner with the ability to replan. The planner is called PLANKEE and it is implemented in KEE¹, which is a knowledge based system shell written in Lisp. PLANKEE uses some of the earlier ideas of replanning, mainly the ideas of Sacerdoti [Sacerdoti 77]. It also uses new ideas for the replanning. These ideas are described in the report.

In figure 2.1 a descriptive model of PLANKEE is shown. The numbers refer to the text.

¹ KEE is a trademark of IntelliCorp, Inc.
Figure 2.1 The PLANKEE system.

Initially we have two states, the starting state and the goal state (1). The problem is to find an optimal plan that will lead from the starting state to the goal state when invoked. The planner takes these two states as input (2). The planner develops, from these two states and with its domain knowledge, a plan (3). When PLANKEE has developed the plan it will be showed to the user. This plan is taken as input to the execute unit (4). The execute unit executes the plan and thus transforms the starting state to the goal state (5).

PLANKEE also supports a replanner. The replanner can be invoked whenever the user wants it to. It takes as input the old plan and the new starting state (6). The new starting state is explicitly given by the user. The replanner replans and develops a new plan (7). This new plan can be fed into the execute unit which executes the new plan (8).
Part I:
Planning
3. The Development of Planners

A planning problem can be viewed as getting from a current situation (an initial state) to a desired situation (a goal state) by using a set of operators. Regard for instance a construction of a house. Our initial state is (hopefully) a flat ground to put the house on, and our goal state is a fine, rigorous house on the ground. Our operators may be on different abstraction levels. Examples of operators could be "dig a hole to put the basement in" or "hammer a certain nail". Maybe we even have an operator like "call a construction company to do the job". The granularity of our operators determine the complexity of our plan.

3.1 Means-ends Analysis and GPS

To get from our initial state to our goal state we must select the right kind of operators at the right time. This is our planning problem. One way to do this, is to select operators according to their ability to reduce the difference between the current state and the goal state. This approach is called means-ends analysis. This is the most common approach among automatic planners.

In means-ends analysis it may occur that in a current state, S1, it is impossible for any operator to reach our goal state. But in an adjacent state, S2, there is an operator that leads to the goal (figure 3.1). We then apply an operator to take us to S2, and there we apply the operator that leads to the goal. We say that the adjacent state satisfies the pre-requisite conditions for the goal-leading operator.

![Figure 3.1. When no operator is applicable in our current state, S1, we apply an operator to go to an adjacent state, S2. In our new adjacent state we can apply a goal-leading operator.](image)

There may be situations where we reach a dead-end and no operator can be found that is relevant to reaching the goal state. When this occur, the last operation must be withdrawn and a new path must be sought from our previous state. Thus, we backtrack. Some planners do not allow getting to a state more distant to the goal than our current state, while others may allow this since it may be necessary for reaching the goal state.

The first planner to use means-ends analysis was the General Problem Solver, GPS [Newell 60]. For a more thorough description of GPS, see Appendix A. GPS selected operators from a difference-operator table. Herein operators were related to differences in the states. GPS needed a difference-operator table for each domain of application and was therefore quite inflexible. The operators associated with each difference were ordered in according to relevance. Each operator had associated preconditions. If the precondition is satisfied, the operator can be applied. If it is not, we must satisfy the precondition before the operator can be applied.
Once an operator was selected, GPS worked recursively on its preconditions. When these were satisfied, the operator was applied to the current state description, and the process continued.

3.2 Forward and Backward Searching

When a planner works from the initial state toward the goal state it is said to do forward searching (called F-rules in [Nilsson 80] and forward chaining in [Winston 84a]). Sometimes it is better to work backwards from the goal state to the initial state. This is called backward searching (called B-rules in [Nilsson 80] and backward chaining in [Winston 84a]). Deciding whether forward or backward searching is preferable depends highly on the shape of the problem space, see figure 3.2. In figure 3.2.a, forward searching is better because every motion toward the goal state will eventually lead to the desired goal without backup. In figure 3.2.b, forward searching can lead to dead ends, forcing backup. In figure 3.2.b backward searching is better. GPS normally works in a forward searching fashion.

![Diagram of state space](image)

**Figure 3.2.** The shape of the statespace determines whether forward or backward searching is preferable. In figure a, forward searching is better and in figure b, backward searching.

3.3 State Description and STRIPS

The complexity of our domain determines what our state description will look like. A state description of our house-building problem could be very complex. If we want to catch everything in our state description, we would end up with an infinite number of different states. This, our computers can't handle today. Therefore we must have idealized states. These will only be approximations of the real world. Still today most problem solving systems run in a quite limited world description.

The easiest description of states can be constructed from predicate calculus wffs (well formed formulas). For an early description of state-transformation methods using predicate calculus, see [Cordell 69]. As an example, consider the blocks-world in figure 3.3. That situation can be represented by the conjunction of following formulas:
clear(B), clear(C), on(B,A), on(A,TABLE), on(B,TABLE)

Figure 3.3 A state of blocks-world.

The predicate clear(B) simply means that the top of block B is clear. On(B,A) means that B is directly supported by A. The table is treated as an ordinary object. Note that the formal representation doesn't say e.g. on which side block C is on in relation to the two other objects.

In our state description, we might also include "default" formulas such as:

$$(\forall x)(\forall y)(\forall z)[(\text{on}(x,y) \land (y\neq z)) \rightarrow \neg\text{on}(x,z)]$$

which simply says that a block in one place cannot be in a different place.

STRIPS [Fikes 71a&b] represented a world model in this way. The problem space for STRIPS was defined by

- the initial state,
- the set of available operators and their effects on world models,
- the goal state.

The initial state and the goal state were described as above.

Each operator were defined by an operator description consisting of two main parts:

- the preconditions of the operator,
- the effects of the operator.

The effects of the operators are called postconditions. The postconditions are defined by two lists, one list of clauses that must be added to the model, the add-list, and one list of clauses that are no longer true and therefore must be deleted, the delete-list.

Imagine a blocks-world state like in figure 3.3. If we want to move block B on top of block C, we could use the following operator:

- **operator**: puton(B,C) ; puts block B on top of block C
- **precondition**: clear(B) \& clear(C) ; both blocks have to be clear
- **postcondition**: 
  - **add list**: on(B,C) \& clear(A)
  - **delete list**: clear(C) \& on(B,A)

Each new state produced by STRIPS was defined by two clause lists. The first list, Deletions, names all those clauses from the initial state that are no longer valid in the state being defined. The second list, Additions, names all those clauses in the state being defined that are not also in the initial state. When an operator is applied to a state, we produce a new state. In our new state the Deletions list is a copy of the Deletions list of our previous state plus any clauses that are deleted by the operator. The Additions list of the new state consists of the clauses from the old state's
Additions list plus the clauses from the operator's add list. Note that the order we add and delete clauses in, is important.

Another approach, which is more common, is to keep a whole world description in every state. In this way we take our old world description from the previous state and delete the clauses from our delete list and add our clauses from our add list. In this way, we do not have to bother about order of additions and deletions in different states.

An example of a state-space is shown in figure 3.4. The arrows relate to possible operators.

Figure 3.4 The state-space of a blocks-world. The arrows determine possible transformations.
3.4 The Frame Problem

The frame problem is one of the most fundamental problems in Artificial Intelligence. It was first defined in [McCarthy 69]. The intrinsic problem lies in the fact that we cannot expect to exhaustively list for every possible operator and for every possible state of our world, how the operator changes the truth or falsity of each individual fact. McCarthy gave the following example: "If we want to do a plan for starting a car, we probably would not involve an operator that would check the tailpipe for a potato. But then, if we had a potato in our tailpipe, our plan wouldn't work." (slightly modified).

The frame problem is actually that of specifying what doesn't change when an event occurs. This problem is harder than the opposite, deciding what does change. The problem with the latter approach is that we never can define postconditions that are infinite detailed. Therefore we reformulate the problem to determine what doesn't change instead.

McDermott divides, in [McDermott 87], the frame problem into three related problems, namely:

- The qualification problem: The problem of making sound predictions about the future without taking into account everything about the past. (Cf. McCarthy's "potato in the tailpipe"-problem.)

- The inertia problem: Deciding how long a fact stays true or a process continues, once it starts. (This is the original frame problem. In [Shoham 87 a&b] this problem is called extended prediction.)

- The ramification problem: The problem of computing all the effects of an operator.

In [Elgot-Drapkin 87] it is argued that the frame problem is actually two problems, one mathematical and one commonsense. The mathematical frame problem pertains to an ideal world with a finite number of states and operators. The problem is a severe computational difficulty for deciding what and what has not changed as state transformation occur. The commonsense problem is based on the notion that it might not even be possible to axiomatize any significant portion of the real world. The problem is that commonsense reasoning deals with illdefined concepts.

One approach to the frame problem is to use what McCarthy and Hayes called frame axioms. Say for instance, [Shoham 87b], you want to rearrange the furniture in your house. This will not change the color of your house and therefore we have a frame axiom that says something like: "Rearranging furniture in a house doesn't change the color of the furniture". The problem with frame axioms is that we need many axioms: rearranging the furniture doesn't change the color of your house, it doesn't change the King of Sweden, and the list can be continued infinitely. If we would allow concurrent actions, the frame axioms would be simply wrong. Someone might actually paint your house while you are busy rearranging your furniture. So we must add an exception to the rule: "rearranging furniture doesn't change the color of the house, unless in the meanwhile someone paints the house in a different color". But even this isn't quite right, since although someone might paint your house in a different color, he might be using a paint that fades away immediately. Therefore we must state an exception to the exception and so on.

Back to our blocks-world. If we move a block, we do not change the position of the other blocks because the change of a block's status must involve an event which involves the mover, which, like all objects, has a unique position at any moment. Likewise a block's color will be unchanged until an operator repaints it. (What happens if the mover overturns a pile of blocks while moving another block?)

The most common approach to the frame problem is to assume that the effects of the operators are local, and the only effects are described by the postconditions, thus simply ignoring it.

For more about the frame problem see [Hayes 73], [Brown 87], [Shoham 87a] or [Brachman 85].
3.5 Hierarchical Plans and ABSTRIPS

The idea of developing a plan "hierarchically" arose early. Already in STRIPS the idea had been discussed. Other planning projects at the same time were also discussing these ideas, see for instance [Siklóssy '73].

A confusing ambiguity in the concept of hierarchical planning was widespread though. In [Wilkins '86] a definition is given, and we will follow that here. We quote:

"..., the essence of hierarchical planning (and a necessary defining condition) is the use of different levels of abstraction both in the planning process and in the description of the domain. An abstraction level is distinguished by the granularity [Hobbs '85], or fineness of detail, of the discriminations it makes in the world." [Wilkins '86]

The vast majority of planners have nested sub-goal structures - hierarchical structures. However the word has another interpretation. The distinction lies in that hierarchical planners generate a hierarchy of representations of the plan in which the highest is a simplification, an abstraction, of the plan, and the lowest is a detailed plan.

STRIPS, for instance, generated sub-goals to the main goals and stacked them. It used a hierarchical structure, but the planner was still non-hierarchical. The major disadvantage of non-hierarchical planners, like STRIPS, is that it does not distinguish between operators that are critical to the success of a plan and those that are simply details.

The overall strategy of hierarchical planning is first to sketch a plan that is complete but too vague. Then refine the vague parts of the plan, into more detailed subplans, until finally the plan has been refined to a complete sequence of detailed operators.

The first planner to invoke these ideas was actually a further development of STRIPS, namely ABSTRIPS [Sacerdoti '73]. For an excellent description of hierarchical planning and ABSTRIPS see [Cohen '82]. ABSTRIPS stands for Abstraction-Based STRIPS. ABSTRIPS plans in a hierarchy of abstraction levels. An abstracted level is a simplified representation of the problem space in which unimportant details are ignored. When a solution to the problem in the abstraction level is generated, all that remains is to account for the details between the operators that represent the plan.

An abstraction level contains all of the objects and operators given in the initial problem space, but some preconditions of some operators are judged to be more important than others. The operator's preconditions are given criticalities, an ordering of the preconditions according to their importance. All preconditions whose truth values cannot be changed by an operator are given a maximum criticality. For each other preconditions, if a sub-plan can be found to achieve it, it is assumed to be a detail and is given a criticality equal to that specified in the partial ordering. If no such plan can be found, it is given a criticality greater than the highest one in the partial order.

ABSTRIPS solves problems with much less searching and backtracking than STRIPS because it is a hierarchical planner. It generates a hierarchy of plans in which the highest level plans are very sketchy and the lowest level plans are detailed. Since a complete plan is formulated at each level of abstraction before the next level is considered, ABSTRIPS can find dead ends early, since important preconditions that can't be satisfied, are detected early in the planning process.

Sacerdoti calls his search strategy length-first search, since it pushes the planning process in each abstraction level all the way to the original goal state before beginning to plan in a lower abstraction level. The major defect of this approach is that when a failure occurs in a lower level, the context of the level where the failure was generated is no longer available. So ABSTRIPS relies heavily on being able to produce good plans at the higher levels.
3.6 Interacting Goals

Consider the problem in figure 3.5. From our initial state we want to rearrange the block according to the goal state. This problem is often called the Sussman anomaly (though "invented" by Allan Brown).

![Initial State](image1)

![Goal State](image2)

Figure 3.5. The problem is to rearrange the block in the initial state according to the goal state. This problem is sometimes called the Sussman anomaly.

The goal state may be expressed like:

\[ \text{on}(A,B) \land \text{on}(B,C) \]

Our hierarchical problem-solver would generate a plan containing the following operators (in some order):

- puton(A,B)
- puton(B,C)

Our planner has to order the operators in some way. Since puton(B,C) is the only operator we can do directly, we might select the following order:

1. puton(B,C)
2. puton(A,B)

We see that doing the first operator will make the second operator impossible. If we ordered them in the opposite way it would be the same thing. The problem lies in that the two sub-goals interact to each other.

This was a very irritating problem for all developers of problem-solvers in the mid 70's. One approach proposed by [Hewitt 75] was to use both forward and backward searching in a means-ends analysis fashion. If it didn't work, a reordering of the sub-goals was tried. He called this constraint analysis, since while working backwards (or forwards) from a situation, attempt to eliminate the constraints that don't hold in reality.

Another approach, proposed by [Tate 75], used protection of goals and reordering of them to achieve the desired main goal. He used the time for which an achieved goal must remain true to determine where to insert the goal in the plan.

All of this methods suffers from having to assume a chronological plan and then modify it so that all goals are satisfied. This approach is called linear planning. For further discussion on this, see [Sussman 73], [Sacerdoti 75] or [Waldinger 77].

The solution to the linear planning problem is to assume the development of plans being non-linear. This, we shall see in the next section.
3.7 The Non-Linearity of Plans and NOAH

Sacerdoti proposed in [Sacerdoti 75] that the plans can be developed in a correct way by assuming them non-linear. He used a plan-description where the operators were parallel. The idea is that if we do not know how to order the operators, we assume that they can be executed in parallel.

See again figure 3.5. The planner can begin with an oversimplified plan (higher abstraction level) that considers the sub-goals of putting A on B and putting B on C as parallel, independent operations. When it looks at the subplans in more detail, a simple analysis will show the interaction between them. Conflicts can then be resolved by imposing linear constraints on some of the detailed operators.

NOAH, which was the first planner to work in a non-linear way, is described in appendix C. We shall here look at the more important parts.

The plan is developed in a procedural net. The procedural net is a network of nodes, each of which contains information about a particular operator in the plan. The nodes are linked together to form a hierarchical description of the plan. By using critics problems are solved, for instance interacting goals. Associated with each node is an add list and a delete list just like in STRIPS. Let's consider an example, the development of the plan for solving the Sussman anomaly (figure 3.5).

In our highest level we have our main conjunctive goal:

\[
on(A,B) \land \neg on(B,C)\]

Figure 3.6 Level 1 in our procedural net.

This conjunctive goal-node of our procedural net is now expanded into two parallel goals. They are not ordered until there is a reason for ordering them.

\[
\begin{align*}
on(A,B) \\
on(B,C)
\end{align*}
\]

Figure 3.7 Level 2 in our procedural net. The subgoals aren't ordered until there is a reason for ordering them.

Each on-node is now expanded to an operator. The operator for on-goals is the puton-operator (see appendix C). This operator include the precondition of the blocks being clear. These are therefore accomplished before the puton-node. Each branch of our previous net is now expanded to a new, more detailed subplan. See figure 3.8.
One of the critics, the resolve-conflicts critic, resolve conflicts in conjunctive goals by ordering them partially. The critic now notices that doing puton(A,B) will delete a precondition of puton(B,C), namely clear(B). NOAH uses this conflict as an opportunity to order the operators; it decides to accomplish puton(A,B) after it has done puton(B,C). See figure 3.9.

After the ordering, a new critic acts. The eliminate-redundant-preconditions critic eliminates redundant preconditions, i.e. preconditions that occur twice when only need to be done once. The critic notices that clear(B) appears as precondition in two places. One is eliminated. After all criticism is applied at level 3, our procedural net will look like in figure 3.10.

NOAH now expands the clear(A) goal at level 3. That is the last goal that remains to be expanded, since both block B and block C have been cleared from the start of the problem.
Now let's examine level 4. To achieve clear(A), NOAH needs to move C off of it and put C someplace; it does not know where so it doesn't say it yet. We have the situation shown in figure 3.11.

![Diagram](image)

Figure 3.11 Level 4 before criticism.

NOAH now notices that the node puton(B,C) interferes with the goal clear(C). Therefore it is placed after that branch. The eliminate-redundant-preconditions critic now acts and notices that clear(C) is mentioned twice in the plan. It eliminates one of the nodes.

The last critic is the use-existing-objects critic. It binds the unbound variables. In our example we have to "bind" where to put block C before execution of the plan. We "bind" it to the table. The final plan is shown in figure 3.12.

![Diagram](image)

Figure 3.12 Level 4 - the final plan.

NOAH have now solved the problem with interacting goals. It assumes the conjuncts are independent, but the non-linear representation of its subgoals frees it from ordering them until it explicitly knows how to order them.

Further developments of NOAH have been done. One example is NONLIN [Tate 77]. NONLIN keeps the choice points where NOAH made a decision. These choice points are used if a selection shows to be wrong. NONLIN can in that case backtrack to this point and make a new choice of the alternatives.

A distributed version of NOAH is described in [Corkill 79]. Here the critics and the state description are distributed.

### 3.8 Other Approaches to Planning

Tate developed the idea of NOAH a bit further. In his planner, NONLIN, he made some improvements of NOAH. The major differences were that he kept points wherever choices were made. These points were used later, if a wrong choice had been made. He also differed between interactions between "important" effects of operators and "nonimportant". NONLIN is described in [Tate 77].

The problem with the Sussman anomaly, figure 3.13, was solved in another way by Waldinger. He passed goals over operators. One ordering of the operators is chosen arbitrarily. When a conflict occurs, the interacted goal is passed back over the operator that caused the interaction. Consider we have first decided to do puton(A,B). Then we must first apply puton(C,Table). When we want to accomplish the goal on(B,C), we notice that this is impossible since block B is not clear. We therefore pass the goal on(B,C) over the operator puton(A,B). Thus, we will first
accomplish on(B,C) and then apply the operator puton(A,B). Hence, the final plan becomes putontable(C), puton(B,C), puton(A,B), which is correct. Passing goals over operators are described in [Waldinger 77].

Figure 3.13 The Sussman anomaly.

Case-based planning (CBP) is a method where a number of old plans are kept in a library. New problems are compared with the set of old plans. If a plan can be matched to the current problem we apply that as our solution. HACKER [Sussman 75] used CBP as it kept libraries of old plans. The organization of old plans is here the big problem. It is argued, [Hammond 83, 86, 87] and [Swartout 88], that what you learn from constructing a new plan is much more than the resultant plan; you learn about how and when to apply the plan; how to debug similar plans; when and where to apply your general knowledge about the world; and, finally, how to index your newly found knowledge in such a way that you are likely to find it the next time you need it. In [Hammond 83, 86, 87] the plans are ordered by the problems they avoid. Another description of a case-based planner appears in [Kolodner 85]. Discussions on how to organize knowledge in a planner appears in [Hayes 75]

Analogical problem solving (APS) [Carbonell 81, 82, 83], [Burstein 83], is a similar way of planning as CBP. Here we try to draw analogies between the current problem and our experiences from previous problems. It is, just like in CBP, important to save planning information for later use. If we cannot find any analogies, we try to divide the problem to sub-problems and try to find analogies to them.

Other approaches to planning are opportunistic planning, see [Hayes-Roth 79], interactive planning, see [Broverman 87], possible worlds planning, described in [Ginsberg 86] and to interleave plan formation and execution, [Chien 75] and [Durfee 86].
4. The Implementation of PLANKEE

We have seen in the previous chapter the many difficulties that arise in developing a planner. Taking this into account, how should we now build our own planner? We wish the planner to be flexible and able to solve problems with interacting goals. We want it also to be thorough and only develop optimal plans. We never want the planner to fail in developing a plan. Another requirement is that the planner should be easy to understand and be flexible to experiment with.

In this chapter we will outline a planner, PLANKEE. PLANKEE is implemented in KEE'. Our domain will be the traditional blocks world.

4.1 World Modelling

Consider the blocks world in figure 4.1.

![Figure 4.1 A blocks world.](image)

To describe our blocks world we first need to define what kind of objects exist in our world. One way is to class the objects in two different categories, BLOCKS and TABLES. In the world we have four blocks and one table. The blocks are called A, B, C and D and the table is called T1. Representing this in KEE will look like figure 4.2.

![Figure 4.2 The objects of the world model in KEE.](image)

All objects in figure 4.2 is represented within KEE by units. Thus OBJECTS is a unit just as well as D is a unit. Every unit can have associated certain features. These features are called slots in KEE. The slots might be inherited by other units. The hierarchy of the units determine which slots are inherited by whom.

The line from OBJECTS to BLOCKS is continuous while the line from BLOCKS to D is dotted. Continuous lines represents subclasses while dotted lines represents members. The difference is that subclasses might have own descendants but members mightn't. This can be viewed as the members are the real, physical, objects while the subclasses are just structures to represent the members relationships and their features.

To represent the world model in figure 4.2 in PLANKEE we need certain features to the objects. For this purpose BLOCKS has two slots. These are SUPPORTER and SUPPORTEE. The slots are inherited by A, B C and D. The values of these slots will repre-

---

'KEE is a trademark of IntelliCorp, Inc.
sent what is on top of the block and what is directly under the block. The unit TABLES have only one slot to maintain the knowledge of the table. The slot is BLOCK-ON-TABLE and it will maintain which blocks that are on the table. Thus to represent the state in figure 4.1 we have the following knowledgebase:

<table>
<thead>
<tr>
<th>unit</th>
<th>slot</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>SUPPORTER</td>
<td>T1</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>SUPPORTER</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>SUPPORTER</td>
<td>T1</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>NONE</td>
</tr>
<tr>
<td>D</td>
<td>SUPPORTER</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>NONE</td>
</tr>
<tr>
<td>T1</td>
<td>BLOCK-ON-TABLE</td>
<td>A, C</td>
</tr>
</tbody>
</table>

This is by no means a full description of the state, e.g. nothing is said about on which side of the other blocks block C is on or what color the blocks are, but it will do for our purpose at time being.

To assert facts in our knowledge base KEE supports several alternatives. One is to use the TellAndAsk language. TellAndAsk is a Lisp-oriented language developed especially to assert facts in KEE. It is, as we shall see, very similar to English. To assert a value to a certain slot the TellAndAsk syntax looks like:

the <slot> of <unit> is <value>

Thus, to assert the fact that C is on top of D, we assert the wffs:

the SUPPORTER of C is D
the SUPPORTEE of D is C

We can also use TellAndAsk to query in the knowledgebase. This is done by giving one of the parameters as a variable. A variable in TellAndAsk is represented by a word with a question-mark in front of it. We can thus query:

the SUPPORTER of C's ?X

and KEE will instantiate ?X to whatever is appropriate and give us the answer. Another wff of TellAndAsk is for asserting a class hierarchy. This wff looks like:

<unit> is in <class>

An example would be the query:

?Z is in BLOCKS

and KEE will give us all units that is members of BLOCKS.
4.2 Default Rules and Truth Maintenance

One alternative to set up the states would be to assert all the values to all the slots by hand. Another alternative is to use the already asserted values to draw conclusions. For instance, if we know the SUPPORTER of A to be C we can draw the conclusion that the SUPPORTTEE of C is A. KEE supports a rulesystem for doing this. The rulesystem supports three kind of rules. PLANKEE uses two of these ruletypes.

The first ruletype is the most common. It asserts conclusions when certain premises are true. The syntax of these rules are:

\[
\text{if } \langle \text{premises} \rangle \text{ then do } \langle \text{conclusions} \rangle
\]

When the \langle premises \rangle are true the rule asserts all the facts in \langle conclusion \rangle. If the \langle premises \rangle at a later time turns false, the rule does not withdraw the asserted \langle conclusions \rangle. Another ruletype of KEE can do exactly this. More about that later. The rules are fired every time we change a value to a slot. KEE allows us to combine the rules with the TellAndAsk language described in the previous chapter.

To assert the SUPPORTTEE of the blocks PLANKEE uses two kind of rules, OVER and EMPTY-TOP. The rule OVER is used when a block is on top of another block and it looks like this:

\[
\text{if } (\text{and} \ (Z \text{is in BLOCKS}) \ (X \text{is in BLOCKS}) \ (\text{the SUPPORTER of } Z \text{is } X)) \text{ then do (change.to (the SUPPORTTEE of } X \text{is } Z))}
\]  

(4.1)

Change.to is a function that changes the value of a slot. Thus the rules tells us that if the SUPPORTER of one block is another block then change the other blocks SUPPORTTEE to the first block. The code is quite similar to Lisp.

The other rule, EMPTY-TOP is used to assert the fact that a block does not carry another block, thus it has an empty top. Here is the rule:

\[
\text{if } (\text{and} \ (Z \text{is in BLOCKS}) \ (\text{cant.find (the SUPPORTER of } X \text{is } Z))) \text{ then do (change.to (the SUPPORTTEE of } Z \text{is NONE}))}
\]  

(4.2)

These two rules will automatically maintain the SUPPORTTEE slot of the blocks. Whenever we change the SUPPORTER the SUPPORTTEE slot are updated.

A third rule is used to put the block on the table initially, unless anything else is said. PLANKEE has a rule called START-ON-TABLE to see for this.

\[
\text{if } (\text{and} \ (Z \text{is in BLOCKS}) \ (\text{cant.find (the SUPPORTER of } Z \text{is } X)) \ (T \text{is in TABLES})) \text{ then do (change.to (the SUPPORTER of } Z \text{is } T))}
\]  

(4.3)

The last rule of PLANKEE is of another type. KEE supports (in version 3.0) a rule type called deduction rules. This ruletype is used to support the Truth Maintenance System,
TMS, in KEE. For more about TMS see [Doyle 79] or [Dressler 88]. Deduction rules have the syntax:

if <premises> then deduce <conclusions>

The deduction rules sets up justifications so that while the premises it asserts the conclusions and when the premises turns false it KEE withdraw the same conclusions. This rule type is used in PLANKEE by the rule ON-TABLE. ON-TABLE are used to maintain what blocks are standing on the table, thus the BLOCK-ON-TABLE slot in TABLES. ON-TABLE looks like:

\[
\text{(if (and (the SUPPORTER of } ?Z \text{ is } ?T \\
?T \text{ is in TABLES))}
\]
\[
\text{then deduce (the BLOCK-ON-TABLE of } ?T \text{ is } ?Z))}
\]

(4.4)

With these rules now described, PLANKEE maintains all the slots but the SUPPORTER slot. Hence, when we move the blocks we only need to assert what block we move and where we move it to, then PLANKEE automatically updates the knowledge-base.

4.3 Operators

Our final plan will be represented by an ordered set of operators. The situation before we apply the plan is the initial situation and after execution of the plan we have the final situation. Associated with each operator in our plan is its input state, which is the state before we apply the operator, and its output state which is correspondingly the state just after we have applied the operator. The initial situation is the first operators input situation and the final situation is the output situation of the last operator in our plan.

Every operator has associated to it preconditions which must be true in its input state for the operator to be applicable. An operator can only be executed if all its preconditions are true in the input state. Every operator adds new facts (cf. the add-list in STRIPS) and deletes some facts when applied (cf. the delete-list in STRIPS). To get the output state of an operator, it is simply the input state minus the facts deleted plus the facts asserted. Note that every state must be consistent, that is, if we have the fact p in a state, we must not have \(-p\) in the same state.

Now let us consider the blocks world again. The only operator we need is an operator to move a block, puton. puton takes two argument, the first is what object to move, and the second is where to put the object. The syntax is:

\[(\text{puton } <x> <y>)\]

which means: take object \(<x>\) and put it (directly) on top of \(<y>\). We can only move one block at a time. Thus, the preconditions for puton are that both \(x\) and \(y\) are clear. We also need to know the old SUPPORTER of \(<x>\), since we want that object clear afterwards. Thus the preconditions for (puton \(<x> \text{ y}>\)) is:

\[
(\text{clear } <x>)
\]
\[
(\text{clear } <y>)
\]
\[
(\text{on } <x> <z>)
\]

After we have applied the operator, we want to 1) add that object \(<x>\) is on top of \(<y>\), 2) add that object \(<z>\) is clear, 3) deny that object \(<y>\) is clear and 4) deny that object \(<x>\) is on top of \(<z>\). Hence, our postconditions are:
(on \(<x> \:<y>\))
(clear \(<z>\))
\neg(clear \(<y>\))
\neg(on \(<x> \:<z>\))

To use a good description of the operators we will use the template as Chapman had in [Chapman 87] to describe them. The preconditions are written to the left of the operator box and the postconditions to the right of it. We also specify arrows to describe in which "direction" an operator takes place.

![Diagram of the puton operator](image)

Figure 4.3 The puton operator.

PLANKEE generates this kind of operators for the plan. In the replanning we shall see that even more features must be added to the operator. The replanning problem also gives new constraints on the precondition. The problem with replanning we shall discuss in a later chapter. For now, we will look how PLANKEE generates the right operators at the right time.

### 4.4 Control Strategy for Plan Generation

We have now seen how to describe the states and how to describe the operator. In this section we will develop a way to concatenate the operators to a plan, given an initial state and a goal state. The algorithm developed in this section are implemented in KEE. We will mainly use the ideas of hierarchical planning and a non-linear assumption of the plan-operators. These methods are described in chapter 3.

To reason about two different states, the initial state and the goal state, PLANKEE uses something called KEEworlds. KEEworlds is supported by KEE as a method to allow representation of alternative states of knowledge. PLANKEE initially sets up the two states START and GOAL. These are represented in PLANKEE by two windows displaying the blocks configuration in the actual state. The user of PLANKEE can simply define the states by clicking and moving the objects in the states.

Both states have the same structured representation showed in figure 4.2. What differs are the values of the slots. Thus by changing the SUPPORTER slot value in different KEEworlds, we define the states. We will look at an example. See figure 4.4. The problem is to switch the blocks C and D.

![Initial and goal states](image)

Figure 4.4 Two states in the Blocks world.
To represent the states showed in figure 4.4 we would have two KEEworlds, START and GOAL, with value as:

<table>
<thead>
<tr>
<th>unit</th>
<th>slot</th>
<th>value in START</th>
<th>value in GOAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>SUPPORTER</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>NONE</td>
<td>NONE</td>
</tr>
<tr>
<td>B</td>
<td>SUPPORTER</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>SUPPORTER</td>
<td>T1</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>SUPPORTER</td>
<td>C</td>
<td>T1</td>
</tr>
<tr>
<td></td>
<td>SUPPORTEE</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>T1</td>
<td>BLOCK-ON-TABLE</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

The planning problem is to develop a plan that is feasible and will change all the slot values in the START world to the ones in the GOAL world. Since we have the rules defined in the previous section, we only need to consider the SUPPORTER slot.

Step 1 in PLANKEE's planning-algorithm is thus to generate the operators for changing the values. We thus go through the blocks and wherever the value of the slot SUPPORTER is not equal we generate a puton-operator. Hence after this step we have the following outline to our plan:

\[
\begin{align*}
\text{(puton B C)} \\
\text{(puton C D)} \\
\text{(puton D T1)}
\end{align*}
\]

Notice that we do not order the operator in this step, we are assuming them being non-interacting, thus non-linear, as we learnt from Sacerdotti and NOAH in the previous chapter.

Step 2 and 3 in the planning algorithm tries to order the operators generated. PLANKEE assumes that the most important is to not deny each others precondition. Thus step 2 is looking for any operator that would deny another operators precondition. We have the situation shown in figure 4.5 (with only the interesting pre- and postconditions for these steps in the algorithm).

![Diagram](image)

Figure 4.5 The plan after step 1.
Thus step 2 looks for any precondition denied by another postcondition. In the example in figure 4.5 the operator (puton B C) will deny (clear C) which is needed for (puton C D). Thus we do the partial ordering of:

\[(\text{puton C D}) \prec (\text{puton B C})\]

Further PLANKEE notices that (puton C D) denies the fact (clear D) which is a precondition of (puton D T1). Therefore the latter operator must be executed before the former. Thus the plan now looks like:

\[(\text{puton D T1}) \prec (\text{puton C D}) \prec (\text{puton B C})\]

Notice that (puton D T1) denies (clear T1) which is rather horrible. The precondition (clear \(<x>\)) is represented by the value of the slot SUPPORTEE. Since tables do not have this slot the operators cannot deny that fact, thus it is no problem in the reality.

When PLANKEE cannot order any more operators in step 2 it calls step 3 to try to order them further. Step 3 checks whether any operator asserts a precondition needed by another operator. Only not already ordered operators can be ordered in step 3. In our example we have ordered all the operators in step 2, thus our plan is not affected by step 3. The order between step 2 and 3 is determined by the opinion that it is better to not destroy for someone than to help someone.

In performing step 2 and step 3 we may run into symmetric partial solutions. If we do so, we remember the solutions for later use and simply choose one for the time. The other solution might show out to be better for the replanning. If it does, we should get the other solution instead for the replanning. This remembrance is not yet implemented in PLANKEE.

When trying to order the operators in step 2 and step 3, we might come up with that an operator do not at all interact with any other operator. It is totally independent. If this occurs we simply add that operator at the end of the plan.

We now have the situation showed in figure 4.6. The plan is now a chain, an assembled set of operators. The preconditions that are interesting at this stage is also marked.

![Figure 4.6 The plan after step 3 with preconditions to each operator.](image)

In step 4 we look for precondition not satisfied in the initial state. We start from the beginning of the plan. PLANKEE now adds operators to satisfy these preconditions. The way PLANKEE knows to accomplish a (clear \(<x>\)) precondition is by asserting the operator (puton \(<z>\) T1) when (on \(<z>\) \(<x>\)).

In the example when starting from the beginning, PLANKEE finds (clear D) not to be satisfied. Since B is on top of D we generate the operator (puton B T1). This operator has these preconditions:

\[(\text{clear B}), (\text{clear T1}), (\text{on B D})\]

The precondition (clear B) is not satisfied. PLANKEE generates new operators to satisfy the new preconditions. This is done in a recursive way. Thus PLANKEE generates the operator (puton A T1). The preconditions for this operators are satisfied, hence we don't need to add any new operators for the moment. PLANKEE has now generated a preplan to satisfy the precondition (clear D). The preplan is shown in figure 4.7.
This preplan is now inserted just before the operator that caused it, i.e. (puton D T1). Now we will look at the next operator in our plan, (puton C D). Its precondition are now satisfied too, and so are the precondition of the last operator, (puton B C). Thus, the plan after step 4 in the planning algorithm are:

(puton A T1), (puton B T1), (puton D T1), (puton C D), (puton B C).

If this plan would be run on our initial state we would not accomplish the goal state. The reason for this is when generating the preplan, we would destroy some sub-goals that was true in the initial state. The subgoal (on A B) are not accomplished by the plan.

Step 5 in the algorithm is to accomplish all subgoals that are destroyed by the plan. Thus we simulate the plan and see what goals are not fulfilled. Thus in our example this step generates the operator (puton A B). This operator are inserted with the method in step 2 in the algorithm. Step 3 is not interesting since the operators preconditions are always satisfied (why?). If step 2 couldn't insert the operator we insert it at the end of the plan. The final plan for our example is now:

(puton A T1), (puton B T1), (puton D T1), (puton C D), (puton B C), (puton A B).

This plan is correct and optimal for our example problem. To summarize, the PLANKEE algorithm for planning is:

1) For every goal not satisfied in the initial state, generate an operator.

2) If operator A denies operator B's precondition, operator B is placed before operator A.

3) If operator A and operator B are not relative ordered then if A asserts one of B's precondition then A is placed in front of B.

   • If during step 2 and 3 arbitrary choices occur, register this and choose one of them.

4) For every precondition not satisfied in the appropriate place, generate a preplan that asserts this precondition. Insert the new operator in accordance to step 2 or at the end of the plan.

5) Generate and insert operators to satisfy any sub-goals destroyed by the plan.

The planner now works for planning normal situations. We shall see when developing the replanner that the planner needs to assert a lot of information into the plan to cope with a good replanning. The replanner needs to know, e.g. why a certain operator was asserted. It also needs to know the structures of the plan. Which operator led to which preplan? Which operators are preplans? In the chapter about the replanner we shall show what information is needed.
4.5 Representation of Plans

The plan is internally represented by a list in the slot THE.PLAN in the unit PLAN. Each step in the planning algorithm takes as input a list from the step previous to it. One step, step 3, needs the plan ordered in a way that the operators are ordered in inserted operators and not inserted operators.

To accomplish different values of the slot THE.PLAN we use the KEEworlds. With this facility, a slot may have one value in every world. Thus while the planner runs it generates a new world for each step in the algorithm. Each new world are generated as a descendant form the START world. By doing like this we can investigate how the plan looks after each step. We can look in the THE.PLAN slot in the PLAN unit in every world and see how the plan is developed to the final plan.

The replanner needs, as we shall see, certain features associated to each operator. Therefore the planner generates one new unit for each operator. The required features are asserted as values in the slots in these units. The units are generated as members to the PLAN unit. The name of the units are similar to the operator, though parenthesis and spaces cannot be part of unit names. Therefore a simple transform are made to the name. For example, the operator (puton A B) generates a unit called puton-A-B.

After running the example, we have the situation shown in figure 4.8. The operators are ordered in alphabetical order.

![Diagram showing operator units](image)

Figure 4.8 The operator units are members of the PLAN unit.

In the plan each operator is represented by a list which has has four atoms. The three first are what we have used to symbolize the operators. The fourth atom is the name of the operator unit. For instance, an operator can be represented by the list:

(puton C D #:PUTON-C-D)

where #:PUTON-C-D is the symbol of the unit. #: is the symbol KEE uses to denote a unit name.

To summarize, the slot THE.PLAN in the unit PLAN has this value in world STEP_5:

[((puton A T1 #: PUTON-A-T1)
 (puton B T1 #: PUTON-B-T1)
 (puton D T1 #: PUTON-D-T1)
 (puton C D #: PUTON-C-D)
 (puton B C #: PUTON-B-C)
 (puton A B #: PUTON-A-B)).
4.6 Execution of Plans

To simulate a plan, PLANKEE has a feature to run the plan developed. This is done by invoking the method in the slot EXECUTE in the OPERATORS unit. The simulation is done in the INITIAL.STATE window.

To support the simulation each object has certain features to maintain information about the object. A block has features which tells the block's position in the window. It also has features of the name of the picture representing the block. These features are in the slots POSITION and PICTURE.NAME. The tables has, except for the features in the blocks, a feature to maintain which positions on the table that are free. This slot is called FREEPOSITIONS.

The EXECUTE method uses these features to do the moving around. It also uses the method slot ON in the same unit. This method checks to see whether a certain operator is practicable in the current state. It also changes the values in the blocks' units in the START world. If it went ok the the new position is calculated and the move is performed by the KEE function MOVE!, supported by the KEEPPICTURES knowledge-base.

The EXECUTE function uses the START world to do the moving in. Therefore this world is changed in accordance to the plan. After a plan is performed the START state can be used as a new start state. Then the user only has to change the GOAL world to continue the planning simulation.
Part II: Replanning
5. To Modify and Reuse Completed Plans

When we have developed the plan and are ready to execute it, it may be the case that the original world has changed. Shall we then throw away our newly developed plan and start all over and develop a new plan? This is of course possible, but is it optimal? We still have the same goal. It is also likely that our original world has only changed slightly. To use the original plan and modify it, intuitively seems better. This, we call replanning.

Replanning may also occur while we are executing the plan. Maybe we don’t discover the change until we are far up in the execution. Then we may also apply replanning. We shall in this chapter look at some previous efforts of replanning.

5.1 The Triangle Table in STRIPS

In STRIPS the plans were represented in triangle tables [Fikes 71b], [Fikes 72]. A triangle table is a triangular matrix. The columns correspond to operators and the rows to facts in the states. See figure 5.1.

<table>
<thead>
<tr>
<th></th>
<th>preconds-1</th>
<th>operator-1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>preconds-2</td>
<td>Add-1</td>
<td>operator-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>preconds-3</td>
<td>(Add-112)</td>
<td>Add-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>preconds-4</td>
<td>(Add-112,3)</td>
<td>(Add-213)</td>
<td>Add-3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>(Add-112,3,4)</td>
<td>(Add-213,4)</td>
<td>(Add-314)</td>
<td>Add-4</td>
</tr>
</tbody>
</table>

Figure 5.1 The triangle table.

The columns of the table, with exception of column 0, are headed with operators. In our example the plan is:

Operator-1 o Operator-2 o Operator-3 o Operator-4.

For each column we place in the top cell the add-list of the operator that heads the column. Thus operator-2’s add-list is Add-2. Going down each column, we place in the consecutive cells the part of the add-list that survives the application of the subsequent operators. Thus, \{Add-213\} denotes those facts in Add-2 not deleted by operator-3 and \{Add-213,3,4\} is the part of Add-2 we can assume after applying operator-3 and operator-4.

We see now that the union of a specific row (excluding column 0) specifies the additions to the world model that the plan had done so far. Thus the union of the cells in the bottom row specifies what the complete plan has accomplished to the world model.

Column 0 we haven’t mentioned yet. This column specifies the preconditions for the current operator, not added by any previous operators. Thus, they must be true in the initial state and must not be deleted by any operator.

Let’s look at an example. Since many facts will reappear in cells beneath each other, we will just specify them in the last row in which they hold, as described in [Nilsson 80].

34
We have the operator:

\[(\text{puton } <x> <y>); \text{ put block } <x> \text{ on top of block } <y>\]

Our initial state and the desired goal state are described in figure 5.2.

![Initial state and Goal state diagram](image)

Figure 5.2 The problem is to rearrange the blocks in the initial state to the configuration in the goal state.

The planner, however it works, should generate the plan:

\[(\text{puton } D \ T1) \ o \ (\text{puton } C \ D) \ o \ (\text{puton } B \ A)\.

The triangle table will look something like in figure 5.3.

![Triangle table](image)

Figure 5.3 The triangle table for the problem in figure 5.2.

In the table we can read, for instance that (puton D T1) will generate the postconditions (clear D), (clear B) and (on D T1) (the cells just under (puton D T1)). We see also that (clear D) holds until the operator (puton C D) is applied.

When a plan is generated we can store it and use it again if we come to a similar problem another time. If the plan is generalized and the variables are not bound to certain boxes, the plan is more useful. We can then bump into the same problem where only the boxes have changed names, and use the generalized plan. These generalized plans are called MACROPS in STRIPS.

Notice that we can determine the world model after each operator. The union of the cells in the outlined field in figure 5.3 is the state description after applying (puton C D). This field is called a kernel.

In real-world problems we compare the kernel to the real world. If they do not match a change has occurred in the real world (since we assume our algorithm for developing the kernel is correct). We can use the kernel for the replanning problem. If a kernel can be found which starts at a lower level, i. e. lower row, we can skip the operators until that kernel and start executing from there.

This approach is quite optimistic and will only work for very special disturbances. In most cases we will not find any matching kernels. Then, a sub-plan are generated to get from the current state "back on the track" to the original plan. MACROPS can be used for this. Since we have the MACROP from the original plan, a new plan is generated by appending an appropriate tail of the MACROP to the subplan.
5.2 Case-Based Planning

In Case-based planning (CBP) a library of old plans are used to select a new plan that is appropriate to the current problem. The essence in CBP is to learn what kind of problems a plan can solve and what kind of problems it avoids. When the best plan for the problem has been chosen and modified for the new situation, the plan is simulated in the new context. Errors may now occur since we may be in a new situation the planner have no experience of. The plan must now be repaired (replanned). This is the most complex problem of CBP.

One approach [Hammond 86, 87] is to find out why the failure occurred. This explanation can be built by back chaining from the failure to the state that caused the plan to go wrong, using a set of causal rules that describe the effects of operators. The explanation is used for finding a strategy for solving the problem. The strategy is then used to build a description of a change in the plan for the current problem. This description is now used as an index to the library of plan modifiers. Thus, the algorithm is [Hammond 87]:

1) Notice the failure.

2) Build a causal explanation of why it happened.

3) Use the explanation to find a repair strategy.

4) Apply each of the general repair strategies using the specifics of the problem.

5) Choose and implement the best repair.

6) Save the new plan.

The idea behind the strategy is simple. The planning information purchased from previous planning, e.g. about interactions between plans and goals, is used to learn what kind of problems a plan is good for. The strategy is indexed by an abstracted problem that we want to match to the current problem. Thus, the explanation gives the planner the knowledge it needs to choose those repairs that will fix a plan without introducing new problems to it.

Once a plan is repaired it can be described as a plan that now avoids the problem that has just been fixed. When it is stored in the memory, it is stored as a plan that avoids this problem and can be used whenever a similar problem occurs.

Case-based planners are described in [Kolodner 85], [Hammond 83, 86, 87]. Experiments with word pronunciation as the problem, using a large knowledge base, is described in [Lehnert 87].

A similar approach is that of behaviors, [Cromarty 84]. Behaviors are sub-plans that the planner knows of. They are similar to the MACROPS in STRIPS. When a problem arise, the planner looks for a behavior that solves the problem. When an error occurs a quick change of the whole behavior can be invoked. This, Cromarty calls "shallow reasoning".

Macro-operators, or macros, are still another name for the same idea. An outline of a theory of macros is described in [Korf 83]. Here is presented a knowledge structure of the problem space that consists of a small number of operator sequences.

A method of transforming (macro-)operators that are not explicit in their state-transformation into operators that are more explicit is described in [Porter 84]. The reason why it is better with more "explicit" operators is also argued herein.

36
Saving plans or part of plans may gradually decrease the system’s performance since searching for an appropriate (sub-)plan can take too much time. It is therefore important to save the right (sub-)plans. This is discussed in [Minton 85].

5.3 Analogical Problem Solving

A similar approach to planning takes the analogical problem solving, APS. The idea is similar to case-based planning. APS assumes there are old plans, developed in a means-ends analysis fashion. Instead of solving problems in the original problem space, in which the states are descriptions of the external world, APS tries solving the problem by starting with a solution to a similar problem and transforming it into a solution for the new problem. APS draws analogies to previous problems.

Just as in case-based planning, learning is an essential part of APS. Useful experiences are saved from the reasoning process. As much valuable information as possible is saved in the problem solving process. Examples on useful information could be: each decision made, reasons for the decisions, false paths, alternatives considered and rejected and of course the resultant solution itself.

When a new problem is encountered that does not lend itself to direct recognition of a previous solution pattern, the planner starts to analyze the problem. If any parallels can be drawn to previous solution, retrieve the full reasoning traces and proceed with the derivational transformation process.

When a new plan has been developed, store it as a divergence to the "parent" plan as another potentially useful source of analogies.

Analogical problem solving is described in [Carbonell 81, 82, 83] and [Burstein 83].

5.4 Marker-Passing

Much of the problem in planning is due to achieve interacting goals. When a plan is developed in a hierarchical fashion, step-wise refinements are made to the higher level plan. The problem with interacting goals is to detect the interactions early and avoid situations where they occur. One approach of doing this is to use marker passing. The very idea behind marker passing is to identify all possible relationships in the current situation and use this information to develop plans or to avoid conflicts.

This is of course impossible in the real world. An approximation is, though, to mark every related node to any central node in the memory. Nodes related to them are then marked and so on. We then try to find nodes that are marked twice (or more times) to find paths between them and to detect conflicts. Consider the horrible example from [Hendler 85]:

Commit suicide while holding a gun.

The knowledgebase knows that to commit suicide you can either hang, shoot or poison yourself. All of these nodes are marked. These nodes continue to mark related nodes, thus, e.g. rope, gun and arsenic are marked. While this is going on markers are also passed from the gun node. An intersection will be found that goes over shoot, and thus we know how to commit suicide, to shoot ourselves. Refer to figure 5.4
Figure 5.4. In a. markers are passed from "commit suicide" and in b. markers are passed from gun. The node "shoot" is marked twice, thus that is a possible solution.

In this fashion a lot of paths will be developed between the nodes in the knowledge-base. Therefore rules are needed to see if any useful information can be gleaned from the paths. Hendler suggests in [Hendler 85] two purposes for these value-rules, first, to quickly reject paths that are not going to yield useful information, and second, extract information from those parts that are useful. This is done by passing the paths through a series of rules in a certain order. The early rules are designed to reject paths known not to be useful, the later rules are designed to quickly check for a certain feature and pass the path on to the next rule if it is not found.

If too many paths are found by the marker-passers, efficiency is lost since every path must be examined. If too few paths are reported the correct information may be missed. The marker-passers must be designed so that it constrains the number of paths reported. Hendler suggests here a propagation limitation on markers to restrict the number of false paths found. Firstly the markers are invoked with a certain strength at the start node. That node divides the strength by the number of nodes it has as neighbors, and passes a marker to each of them with the new strength. These nodes continues to split the strength among their adjacent nodes. Once strength falls below a certain limit, marker-passing stops along that path.

A further development of this method is described in [Hendler 87]. Here Hendler suggests that a single object may be decomposed into a set of smaller features to recognize similar characteristics. He called these microfeatures. Markers are passed from one object to the microfeatures and then to objects that also carry these microfeatures. For instance, to understand that a letter-opener can be used as a knife, markers are passed to microfeatures such as pointed, metallic and sharp. These microfeatures passes the marker to the knife, and are summed there. In this way we can realize that a knife and a letter-opener are almost the same thing.

This latter way of identifying properties reminds of a neurological process. It can be used in replanning to identify similar actions that do not conflict.
5.5 Replanning Through Abstraction

Microfeatures, described in the last section, are used to identify similar properties to different actions by decomposing the actions into smaller features. Instead of decomposing the actions, we can abstract them to more abstract versions of the action. When a failure in the plan is found, we rise to a higher, more abstracted level. Alternatives for the current operator are then looked for.

By organization of the planning knowledge in a "is-a"-hierarchy, we can try to match new situation to old ones. Alterman calls this adaptive planning and describes it in [Alterman 85,86]. From there this example is taken (modified): We have a plan for riding the subway in California, BART (Bay Area Rapid Transfer). The steps in the plan are:

• Buy a BART-ticket from a machine
• Feed the ticket to the gate machine, the machine returns it back
• Take the train
• Feed the ticket to the exit machine which opens the exit gate

Now we want to ride a subway in New York. In New York a token is bought from a teller, then the token is put into a turnstile. We then enter the turnstile, ride the train, and exit by pushing through the exit turnstile. There are a great number of differences between the BART-way and the NYC-way. Still, the overall structure is very similar. We should be able to modify the original plan to know how to ride the NYC-subway.

Trying to apply the first step in the BART-plan in NYC, we see that we do not have any ticket machines. Through abstraction the planner finds another way to purchase a ticket, namely to buy it from the teller, so the planner moves down the isa-hierarchy, changing the first step in the plan to buy ticket from the teller instead. In this manner the planner is substituting the rest of the steps in the plan into new modified steps.

In here, as in the other methods of modifying old plans, the planner depends heavily on domain-knowledge about the plan and the current context. Problems in this approach arise, e.g., in how to choose the right abstraction (there might be a number of abstractions associated with each step), knowing when to look for an alternative version of a step, when to stop abstracting and so on. Alterman gives some techniques on some of the problems in [Alterman 86].

5.6 Dependency Information

Truth Maintenance System (TMS) is a system for maintaining explicit records of the assumptions made and the justifications for beliefs derived from these assumptions. By keeping track of these assumptions, TMS is able to determine on which assumptions a conclusion is based. If any assumptions cease to be believed, the conclusion is withdrawn.

TMS maintains dependencies between derived information and the assumptions on which the derivations are based. This dependency information can be used by the planner for replanning. For instance, a planner might have identified for each operator a set of propositions whose truth before this step assures that nominal execution of the remainder of the plan, achieves the goal (cf. kernels in STRIPS). Given such dependency information, the system can respond to a failure by invoking dependency-directed replanning effort, in which new plan fragments are constructed only for the goals affected by the failure. For example [Swarthout 88], if two towers are to be constructed, and a failure occurs during the construction of the first tower, then only the remainder of the plan for the first tower is changed. The existing plan for the second tower is left intact.
Such dependency information is important for the structure and intent of the plan. See also [Tate 77, 84].

Dependency information can also be used in comparing the environmental circumstances when the plan was developed and the circumstances when the plan is to be executed. In order to identify all different environmental features that were used to choose the plan, the planner would need to use TMS-like justifications for choosing one plan step over another, one ordering of steps over another, and so on.

Similar of knowing dependency information, we also need to know what facts cause new alternatives to become available to the planner. For example, we need to know whether a new block is put on the table or there is a change in the operator repertoire, maybe a new operator is added due to a reconstruction of the robot hand that moves the blocks.
6. Making PLANKEE Able To Replan

In the previous chapter we have seen that for replanning, it is important to keep information gained from the planning process. This knowledge is necessary for doing an effective modification of the plan. We shall in this chapter look at what information we need for doing an effective replanning and develop a replanner that uses this information for doing effective replanning.

A modification is done to PLANKEE, so that it is able to replan its own failed plans. We will also discuss the implementation of this.

6.1 A Dynamic World Description

In chapter 4 we showed how the initial and goal state were described in KEE by KEE-worlds. We have one world that represents the starting state and one world that represent the goal state. From these worlds, PLANKEE works to develop the plan. Similarly we have a world describing the new initial state. This world is called NEW_START in PLANKEE. This new world has exactly the same features as the other two worlds.

This new world PLANKEE uses along with the old plan to do replanning. The original starting world and the goal world lies, as we shall see, implicitly in the old plan.

The NEW_START world can be set in PLANKEE individually from the other worlds. For instance, we can follow the plan and see what new worlds are generated, then in some step of the plan we can do a change in the new starting world, which thus can simulate a disturbance while doing the execution. If we then define a new starting state, PLANKEE takes this as NEW_START and performs replanning from this new world and the old plan. In this way we can simulate an error occurring during execution of the old plan.

6.2 Different Kinds of Operators and Their Intentions

To make effective replanning we need to know a lot more about the plan than just the final plan. In this section we shall see what information we need and how it is implemented in PLANKEE.

When the replanner is regarding the old plan and are trying to modify it to new situations it should delete some operators and add some operators. In the original plan some operators are added to satisfy certain subgoals in the goal state, while others are added to satisfy precondition for other operators. We shall differ these two types of operators. The first kind we call core-operators and the second kind we call pre-operators.

Thus,

core-operators are added to satisfy sub-goals in the goal state,

and

pre-operators are added to satisfy pre-conditions.
We shall look at the same example as in chapter 4. Refer to figure 6.1.

![Initial state diagram](image)

![Goal state diagram](image)

Figure 6.1 The example from chapter 4.

In this example the plan developed was:

(puton A T1), (puton B T1), (puton D T1), (puton C D), (puton B C), (puton A B).

Here, the operators (puton D T1), (puton C D), (puton B C) and (puton A B) all respond to subgoals in the goal state. Thus, these all are core-operators. The other two operators,

(puton A T1) and (puton B T1)

are just added to make block D clear, so it can be moved. These two operators are therefore pre-operators.

This dividing of the operators are useful when deleting operators. If a core-operator are deleted then the pre-operators asserted to satisfy the core-operator's precondition can also be deleted.

To delete an operator we need to know why the operator was asserted. Therefore we attach an intention to each operator. The core-operators have intentions that match sub-goals in the goal state. For instance the operator (puton B C) from the previous example are asserted to satisfy the sub-goal that block B is on top of block C. We say that its intention is (on B C). The pre-operators' intention all correspond to pre-conditions of the type (clear <x>). For instance, the operator (puton B T1) has the intention (clear D).

When pre-operators are generated by PLANKEE, their preconditions maybe also have to be satisfied. In the example the operator (puton A T1) were asserted to accomplish a clear top of block B. Its intention is therefore (clear B). Hence, pre-operators can be linked to pre-plans in front of a core-operator. The pre-plan depends of the intention of the operator that generated it. If the core-operator is deleted the whole pre-plan can also be deleted.

We have now defined some new concepts that we can use while reasoning about the plan. These concepts are represented in PLANKEE by the slots CORE-OPERATOR?, PRE-OPERATOR and INTENTION. These slots are attached to the unit representing the current operator. CORE-OPERATOR? has the value YES if the current operator is a core-operator. PRE.OPERATOR has the value of the pre-operator to the current operator. INTENTION has the value of the intention of the operator.

To summarize we will show the unit to the operator (puton D T1):

<table>
<thead>
<tr>
<th>Unit</th>
<th>Slot</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTON-D-T1</td>
<td>CORE-OPERATOR?</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>PRE.OPERATOR</td>
<td>(puton B T1 #:PUTON-B-T1)</td>
</tr>
<tr>
<td></td>
<td>INTENTION</td>
<td>(on D T1)</td>
</tr>
</tbody>
</table>

42
6.3 Extending the Preconditions

In chapter 4 we defined preconditions for the planning algorithm. Here we shall see that PLANKEE needs to extend this feature to cope with more general situations. We shall also see that some preconditions are very important to achieve the goal state. Therefore these preconditions need to be protected.

Preconditions, as described in chapter 4 were of two kinds. The first kind was of the (clear &lt;x&gt;)-type and the second one was of the (on &lt;x&gt; &lt;z&gt;)-type. We shall here extend the second type of preconditions.

Where an operator is inserted in the plan depends upon which new state the previous operators have transformed the initial state into. For example when building a tower of blocks it is very important to do the putons in the right order. In the previous example we notice that when applying the operator (puton B C) it is very important that block C is on block D and that block D is on the table. We thus extend the precondition definition to hold for all facts that must be asserted before the current operator.

This extension is done because of the situation where a subgoal that holds in the initial state does not hold in the new initial state. A sub-goal is either true in the initial state or is asserted by the plan. If it was true in the initial state it will not be added by the plan. If then it is not true in the new initial state it will never be asserted.

PLANKEE recognizes these preconditions never asserted and calls them protected precondition. These protected precondition are attached to the operator that must check it before itself is applied. If the protected precondition does not hold in the new initial state and are not asserted by any other operator before the current one, the replanner must insert a new operator to satisfy this protected precondition. Refer to the example shown in figure 6.2.

![Figure 6.2 A problem with a protected precondition.](image)

The plan generated by PLANKEE is:

(puton A T1), (puton B C), (puton A B)

The unit for the operator (puton B C) will, together with the slots explained in the previous section look like:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Slot</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUTON-B-C</td>
<td>PRE_CONDITION</td>
<td>((on D T1) (on C D))</td>
</tr>
<tr>
<td></td>
<td>PROTECTED_PRE_CONDITION</td>
<td>(on D T1) (on C D)</td>
</tr>
<tr>
<td></td>
<td>CORE_OPERATOR?</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>PRE_OPERATOR</td>
<td>(puton A T1 #:PUTON-A-T1)</td>
</tr>
<tr>
<td></td>
<td>INTENTION</td>
<td>(on B C)</td>
</tr>
</tbody>
</table>

The features presented in this and the previous section are all maintained by PLANKEE. We shall now look how these features are used in the replanning part of PLANKEE.
6.4 The Replanning Strategy

PLANKEE's replanning algorithm works iteratively in several steps to transform the old plan into a new, correct plan.

The first step in the replanning algorithm looks for any subgoals that are not asserted by any operator and doesn't either hold in the new initial state. If such a subgoal is found, then PLANKEE generates a new operator to satisfy this. The new operator is inserted in the old plan in accordance to the planning algorithm described earlier.

Step 2 in the replanning algorithm tries to delete any superfluous operator. PLANKEE starts from the beginning of the plan and checks all operators and while

- The intention of the operator is satisfied
and
- the protected precondition is satisfied
or
- there isn't any protected precondition attached to the operator

delete the operator and any attached pre-operators.

Step 3 satisfies any preconditions, the old definition, not satisfied. This is done by asserting new operators just in front of the core-operator with the precondition not satisfied. This step is similar to step 4 in the planning algorithm described in chapter 4.

In step 4 we eliminate any redundant operators not previously deleted. This is done by simulating the plan and checking for any redundancy. It is simply done by checking whether the intention of the operator are true just before the operator is invoked.

Step 5 in the replanning algorithm is similar to step 5 in the planning algorithm. Here we make a final check that all subgoals are satisfied. If we find any that aren't, we insert a new operator into the plan. Step 4 and 5 are in the same slot in PLANKEE. This is done because of them using common variables.

To summarize, the overall algorithm for replanning is:

1) Find any subgoals not holding in the new initial state neither asserted by the plan. For each of these insert an operator in the old plan.

2) Start from the beginning of the plan and for all operators: while

   the intention of the operator is satisfied

   and

   › the protected precondition is satisfied

   or

   › there isn't any protected precondition attached to the operator

   delete the operator and any attached pre-operators.

3) Find preconditions not satisfied. Satisfy them by adding pre-operators.

4) Delete all redundant operators.

5) If a subgoal is never satisfied, insert a new operator to satisfy it.

Let us see how this algorithm works by regarding two examples.
6.5 Two Replanning Examples

Consider the example shown in figure 6.3

![Diagram showing initial, goal, and new initial states](image)

Figure 6.3 The three world models in PLANKEE.

The plan that PLANKEE will develop in this example is:

(puton A T1), (puton B C), (puton A B).

The representation of the plan in PLANKEE is shown in table 6.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>(puton A T1)</th>
<th>(puton B C)</th>
<th>(puton A B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE-OPERATOR?</td>
<td>nil</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>PRE.OPERATOR</td>
<td>nil (puton A T1 #:PUTON-A-T1)</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td>INTENTION</td>
<td>(clear B)</td>
<td>(on B C)</td>
<td>(on A B)</td>
</tr>
<tr>
<td>PRE_CONDITION PROTECTED.</td>
<td>nil</td>
<td>((on D T1) (on C D))</td>
<td>((on D T1) (on C D) (on B C))</td>
</tr>
<tr>
<td>PRE_CONDITION</td>
<td>nil</td>
<td>(on D T1) (on C D)</td>
<td>(on D T1) (on C D)</td>
</tr>
</tbody>
</table>

Table 6.1 The plan with associated slot values.

Step 1 in the replanning algorithm tells us to find sub-goals that aren't satisfied neither by the new initial state nor by the plan operators. In this example we find that the fact (on C D) is not satisfied anywhere. Thus we generate the operator (puton C D). This operator are inserted just in front of the operator (puton B C) because it destroys the precondition (clear C). Hence, the plan after step 1 is:

(puton A T1), (puton C D), (puton B C), (puton A B).

In step 2 we eliminate any operators with their intentions satisfied and no violating protected precondition. We see that (puton A T1)'s intention, (clear B), is already satisfied, thus we can delete that operator. We also notice that (puton B C)'s intention, on B C), is also satisfied, but it has a protected precondition not satisfied why we cannot delete that operator. The plan after step 2 is:

(puton C D), (puton B C), (puton A B).

The next step, step 3, tells us to find preconditions not satisfied and satisfy them. In the example we find that (puton C D) has a precondition, (clear C), not satisfied. We there-
fore generate the operator (puton B T1) and adds it just in front of (puton C D). The other operators have their preconditions satisfied. The plan is now:

(puton B T1), (puton C D), (puton B C), (puton A B).

Step 4 tells us to delete all redundant operators. Since there aren't any in this example, we will continue to step 5. Here we shall look for subgoals not satisfied anywhere. We cannot find any and this ends the replanning in this example. The final plan is thus:

(puton B T1), (puton C D), (puton B C), (puton A B)

which is correct and optimal. We shall look at another example which shows other effects in the replanning algorithm. Refer to figure 6.4

![Figure 6.4](image)

The plan developed by PLANKEE is:

(puton B T1), (puton C D), (puton B C), (puton A B)

Step 1 in the replanning algorithm will not change the plan, but step 2 will. Any operator with their intentions already satisfied will be deleted. The operator (puton C D)'s intention is (on C D), but this is already satisfied in the new initial state, thus we can delete it. PLANKEE also deletes that operator's preoperator which is (puton B T1). Any more operators can't now be deleted. Hence the plan after step 2 is:

(puton B C), (puton A B)

In step 3 we shall look for preconditions not yet satisfied. We see that to invoke (puton B C) block B and C must be clear, but block C is not clear because block A is on top of it, thus we generate the operator (puton A T1) just in front of (puton B C). After step 3 we will have the plan:

(puton A T1), (puton B C), (puton A B).

Step 4 and 5 will not change the plan, thus this is the final plan which is correct and optimal.

We have now seen some examples of how the replanning algorithm works in practice. It makes strong use of the information associated with each operator. By keeping information, PLANKEE easily can modify the plan. The replanning algorithm works mainly in the beginning of the plan. This can be viewed as it is working to get the plan "back on the right track".
6.6 Execution of the New Plan

To simulate the new plan, PLANKEE supports the same function for running the new plan as it did for running the original one. It is the same method slot as described in chapter 4.6. The simulation is done in the NEW.INITIAL.STATE window. The same features as described in 4.6 is associated with the blocks in the NEW_START world.

The EXECUTE slot uses the NEW_START world to do the moving in. This world is therefore changed in accordance to the plan. After simulating a new plan, the NEW_START world can be used as a new state and the user only needs to change the GOAL world to continue working with PLANKEE.
Appendices
Appendix A

GPS - The General Problem Solver

See [Newell 60] for a thorough description of GPS. For a discussion of search methods in planning problem see [Kibler 81] or [Korf 87]. Both [Nilsson 80] and [Winston 84a] have a shorter description of GPS.

The General Problem Solver was the first actual automatic problem solver. GPS introduced means-ends analysis (chap 3.2). GPS compared the current state with the goal state, and any differences were eliminated by operators in a step-by-step fashion. The system held a table with differences, and what operator to apply when a certain difference arose. All operators had preconditions which had to be satisfied when an operator were to be applied. If the preconditions weren't satisfied, GPS tried to satisfy them before applying the operator to the current situation. The old operators and preconditions were stacked. GPS rejected any movement to an adjacent state that is more distant than the goal state. The main algorithm looked something like this.

\[
\text{S} \leftarrow \text{the Initial State} \\
\text{G} \leftarrow \text{the Goal State} \\
\text{while S doesn't match G loop} \\
\quad \text{D} \leftarrow \text{difference between S and G } \quad \text{; here we have one backtracking point} \\
\quad \text{O} \leftarrow \text{operator for reducing D } \quad \text{; here we have another backtracking point} \\
\quad \text{P} \leftarrow \text{preconditions for O} \\
\quad \text{call GPS with P} \\
\quad \text{S} \leftarrow \text{result of applying O to S} \\
\text{endwhile}
\]

Some versions of GPS dropped concern for prerequisite conditions temporarily. The problem was first solved on the top level with main steps only. In this way GPS avoided working on prerequisites for a step if it is clear that the step is bad. In filling the gaps on the next level, again no attention is paid to prerequisites. We see here the first approach to hierarchical planning.
Appendix B

STRIPS - STanford Research Institute Problem Solver

STRIPS is described in [Fikes 71a]. STRIPS was only a part of a robot called Shakey. Part-time reports of the robot are given in [Nilsson 69] (the overall structure), [Green 69] (the theorem proving mechanism), [Fikes 71b] (the plan executor) and in [Coles 69] (the natural language interface). Further development of STRIPS is described in [Fikes 72] and [Sacerdoti 74]. Most of the later planners build on STRIPS ideas. Therefore many of the descriptions refer and compare to STRIPS. See for instance [Siklozy 73], [Sacerdoti 75], [Tate 75] and [Kibler 81]. For a semantic analysis on STRIPS, see [Lifschitz 86]. In [Nilsson 80] we have a shorter description of STRIPS.

STRIPS was developed for a robot that moved boxes around in a house. It adopted the means-ends analysis strategy from GPS. STRIPS maintained a stack of goals and focused its problem-solving effort on the top goal of the stack. Initially, the goal stack contained just the main goal. Whenever the top goal in the main stack matched the current state, it was eliminated from the stack, and the match substitution was applied to the expressions beneath it in the stack. Otherwise, if the top goal in the stack was a compound goal, STRIPS added each of the component sub-goals above the compound goal in the stack. When all of the compound goals were solved, it reconsidered the compound goal again, re-listing the component sub-goals on top of the stack if the compound goal didn't match the current state. This reconsideration of the compound goal was the, rather primitive, safety feature that STRIPS used to deal with when interacting goal problem occurred. If solving one component goal undid an already solved component, the undone goal was reconsidered and solved again.

The order in which STRIPS stacked its goal in, was important. In problems with interacting goals, STRIPS sometimes produced non-optimal plans because it ordered its goal wrong. This problem was one of the biggest problem in the development of planners. Sacerdoti solved it in [Sacerdoti 75].

STRIPS used preconditions to determine when an operator was applicable. Preconditions were stated as wffs. To determine the effects, STRIPS used two lists, the delete list and the add list. On the delete list were specified those clauses that might no longer be true in the new state. On the add list were those clauses that might not have been true in the original model but are true in the new model. STRIPS had for instance an operator to push blocks like:

```
operator push(k,m,n)
precondition ATR(m) ; the robot in position m
^ AT(k,m) ; object k in position m
delete list ATR(m)
^ AT(k,m)
add list ATR(n) ; the robot is in position n
^ AT(k,n)
```

The parameters of an operator schema were instantiated by constants at the time of operator application. Thus, when the add and delete lists were used to create new states, all parameters occurring in them would have been replaced by constants.

Each new state produced by STRIPS was defined by two clause lists. The first list, Deletions, named all those clauses from the initial state that were no longer present in the state
being defined. The second list, *Additions*, named *all* those clauses in the state being defined that are not also in the initial state. When an operator was applied to a world model, the Deletions list of the new state was a copy of the Deletions list of the old state plus any clauses from the initial state that were deleted by the operator. The Additions list of the new state consisted of the clauses from the old state’s Additions list plus the clauses from the operator’s add list. This is the way STRIPS solved the frame problem. It simply ignored it by using idealized world description.

The main algorithm of STRIPS looked very much like that of GPS. The differences were in the way to choose sub-goals and operators.
Appendix C

NOAH - Nets Of Action Hierarchies

NOAH is described in [Sacerdoti 75]. NOAH was actually a further development of the same project as STRIPS and ABSTRIPS. [Tate 75] discusses similar ideas. NOAH's ideas are still today the state of the art. Therefore, discussions on NOAH ideas are common. See for instance [Waldinger 77], [Dawson 77], [Nilsson 80] (here NOAH is called DCOMP) or [Rosenschein 81]. Further developments of NOAH ideas are discussed in [Tate 77], [Corkill 79], [Wilkins 82], [McDermott 83], [Allen 83], [Tate 84], [Berlin 85], [Chapman 85], [Vere 85], [Drummond 85], [Miller 85], [Ginsberg 86] and [Chapman 87]. Theories of the hierarchical approach to planning are discussed in [Hobbs 85], [Manna 86] and [Wilkins 86]. Other approaches to planning are discussed in [Waldinger 77], [Hayes-Roth 79], [Stefik 81], [Rich 81], [Wall 82] and [Faletti 82]. For a shorter description of hierarchical planning and NOAH, see [Cohen 82]. In [Sacerdoti 79] the problem solving tactics are summarized.

NOAH works in a hierarchical, non-linear way to produce plans. It doesn't assume any ordering between the partial goals until it is necessary. By using critics to the "parallel" plan, it orders the goals. When a level of abstraction is completed, it descends further and expands the subgoals to new subgoals or operators.

The plan is represented in a procedural net. The procedural net is a network of nodes, each of them corresponding to a (sub-)goal or an operator. The nodes contain both procedural and declarative information. The procedural information includes functions that expand goals into subgoals and simulates the actions of the operators. The declarative information represents the effects of an operator. For instance, if the operator puton(A,B) is executed, the declarative information reports that block B no longer has a clear top.

NOAH represented its procedures in SOUP functions. SOUP is an abbreviation for Semantics Of User's Problems. They represent the knowledge of how to expand goals. Achieving the on(A,B)-goal looked something like:

```procedural net

(PUTON
  (QLAMDA (ON ?X ?Y)
    (PAND (PGOAL (Clear X))
      (CLEARTOP X)
      APPLY
      (CLEAR))

    (PGOAL (Clear Y)
      (CLEARTOP Y)
      APPLY
      (CLEAR))

    (PGOAL (Put X on top of Y)
      (ON X Y)
      APPLY
      NIL)

    (PDENY (CLEARTOP Y)))))
```

When a PGOAL is activated, a node is generated at the next lower level in the net. The first argument specifies the subgoal's name, e.g. (Clear X). The three PGOALS
(subgoals) in PUTON are (Clear X), (Clear Y) and (Put X on top of Y). The first two are conjunctive goals as specified by PAND. They are not ordered, thus they may be attained in parallel. The effects of applying the goal are described by the second argument to PGOAL, here (CLEARTOP X), (CLEARTOP Y) and (ON X Y). The SOUP code also suggests which other SOUP-code that should be applied to expand the sub-goals it has just created, here for instance (CLEAR). Finally the postconditions are specified, here NOAH should deny that Y has a clear top, thus deny (CLEARTOP Y).

Every time NOAH has created a new level of the plan, critics are applied to the plan to find defiances in it. A critic simply checks for the kinds of conflicts it is designed to correct. The critics are:

- Resolve-Conflicts: Examines conjunctive goals that are to be achieved in parallel. If violation may occur, the critic tries to order the actions so that neither violates the preconditions of the others.

- Eliminate-Redundant-Precondition: If the same operator gets specified twice when it needs to be done only once, this critic deletes one of the specifications.

- Use-Existing-Objects: Uninstantiated objects are bound to real objects whenever a clear choice is possible.

The planning algorithm of NOAH operates repeatedly on the current lowest level of the procedural net. Once all nodes in the current level have been expanded, critics check for interacting goals before another, lower level of abstraction is tried. The algorithm is something like:

```plaintext
while not a complete plan is ready loop
    simulate the current plan and expand it into a
    more detailed plan
    criticize the new plan and order nodes if necessary
endwhile
```

To summarize NOAH we quote Sacerdoti:

"NOAH makes no rash assumptions, but preserves all the freedom of ordering that is implicit in the statement of a conjunctive goal. It assumes the conjuncts are independent, but the nonlinear representation frees it from worrying about additivity. It applies its critics constructively, to linearize the plan only when necessary. By waiting until it knows the nature of the conjuncts' interactions, NOAH is sure to place actions in the correct order, and thus needs never undo the effects of a false assumption." [Sacerdoti 75]"
References


Bonissone 87 Bonissone, Piero P., Gans, Steven S. and Decker, Keith S., RUM: A Layered Architecture for Reasoning with Uncertainty, *Proceedings of the Tenth Int. Joint Conf. on Artificial Intelligence (IJCAI-10)*, 1987, pp 891-8


Broverman 87 Broverman, Carol A. and Croft, W. Bruce, Reasoning about Exceptions During Plan Execution Monitoring, *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, 1987, pp 190-5


Chien 75 Chien, R. T. and Weissman, Planning and Execution in Incompletely Specified Environments, *Proceedings of the Fourth Int. Joint Conf. on Artificial Intelligence (IJCAI-4)*, 1975, pp 169-74


Corkill 79 Corkill, Daniel D., Hierarchical Planning in a Distributed Environment, *Proceedings of the Sixth Int. Joint Conf. on Artificial Intelligence (IJCAI-6)*, 1979, pp 168-75


Davis 77 Davis, P. R. and Chien, R. T., Using and Re-using Partial Plans, *Proceedings of the Fifth Int. Joint Conf. on Artificial Intelligence (IJCAI-5)*, 1977, p 494


Dean 87 Dean, Thomas and Boddy, Mark, Incremental Causal Reasoning, *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, 1987, pp 196-201

de Kleer 84 de Kleer, Johan, Choices Without Backtracking, *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, 1984, pp 79-85

de Kleer 86 de Kleer, Johan and Williams, Brian C., Reasoning About Multiple Faults, *Proceedings of the National Conference on Artificial Intelligence (AAAI-86)*, 1986, pp 132-9


Doyle 85  Doyle, Jon, Reassembled Assumptions an Pareto Optimality, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 87-90

Dressler 88  Dressler, Oskar, Extending the Basic ATMS, *Proceedings of the Eighth European Conf. on Artificial Intelligence (ECAI-88)*, 1988, pp 335-40

Drummond 85  Drummond, Mark E., Refining and Extending the Procedural Net, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 1010-2


Freitag 88  Freitag, Hartmut and Reinfrank, Michael, A Non-Monotonic Deduction System Based on (A)TMS, *Proceedings of the Eighth European Conf. on Artificial Intelligence (ECAI-88)*, 1988, pp 601-6


88

Ginsberg, Matthew L., Counterfactuals, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 80-6


Hammond, Kristian J., Explaining and Repairing Plans that Fail, *Proceedings of the Tenth Int. Joint Conf. on Artificial Intelligence (IJCAI-10)*, 1987, pp 109-14


Hayes-Roth, Barbara, Hayes-Roth, Frederick, Rosenschein, Stan and Cammarata, Stephanie, Modeling Planning as an Incremental, Opportunistic Process, *Proceedings of the Sixth Int. Joint Conf. on Artificial Intelligence (IJCAI-6)*, 1979, pp 375-83


Hendler, James A., Marker-Passing and Microfeatures, *Proceedings of the Tenth Int. Joint Conf. on Artificial Intelligence (IJCAI-10)*, 1987, pp 151-4

89


Hewitt 75  Hewitt, Carl, How to Use What You Know, *Proceedings of the Fourth Int. Joint Conf. on Artificial Intelligence (IJCAI-4)*, 1975, pp 189-98

Hobbs 85  Hobbs, Jerry R., Granularity, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 189-208

Kibler 81  Kibler, Dennis and Morris, Paul, Don’t be stupid, *Proceedings of the Seventh Int. Joint Conf. on Artificial Intelligence (IJCAI-7)*, 1981, pp 345-7


Lehnert 87  Lehnert, Wendy G., Case-Based Problem Solving with a Large Knowledge Base of Learned Cases, *Proceedings of the National Conference on Artificial Intelligence (AAAI-87)*, 1987, pp 301-6


McCarthey 87  

McDermott 83  

McDermott 87  

Miller 85  
Miller, David, Firby, R. James and Dean, Thomas, Deadlines, Travel Time and Robot Problem Solving, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 1052-4

Minton 85  
Minton, Steven, Selectively Generalizing Plans for Problem-Solving, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 596-9

Newell 60  

Newell 72  

Nilsson 69  

Nilsson 80  

Porter 84  

Rich 81  

Rosenschein 81  

Rosenschein 82  
Rosenschein, Jeffrey S., Synchronization of Multi-Agent Plans, *Proceedings of the Sixth Int. Joint Conf. on Artificial Intelligence (IJCAI-6)*, 1979, pp 829-35

Sacerdoti 73  

Sacerdoti 75  

Sacerdoti 79  Sacerdoti, Earl D., Problem Solving Tactics, *Proceedings of the Sixth Int. Joint Conf. on Artificial Intelligence (IJCAI-6)*, 1979, pp 1077-85


Shoham 87a  Shoham, Yoav, Reasoning about Change: Time and Causation from the standpoint of Artificial Intelligence, PhD thesis at Yale University, 1987


Shrobe 79  Shrobe, Howard Elliot, Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures, *Proceedings of the Sixth Int. Joint Conf. on Artificial Intelligence (IJCAI-6)*, 1979, pp 829-35


Stuart 85  Stuart, Christopher, An Implementation of a Multi-Agent Plan Synchronizer, *Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)*, 1985, pp 1031-3


Tate 75  Tate, Austin, Interacting Goals and Their Use, *Proceedings of the Fourth Int. Joint Conf. on Artificial Intelligence (IJCAI-4)*, 1975, pp 215-18

Tate 77  Tate, Austin, Generating Project Networks, *Proceedings of the Fifth Int. Joint Conf. on Artificial Intelligence (IJCAI-5)*, 1977, pp 888-93

Tate 84  Tate, Austin, Goal Structure - Capturing the Intent of Plans, *Proceedings of the Sixth European Conf. on Artificial Intelligence (ECAI-84)*, 1984, pp 273-6


92
<table>
<thead>
<tr>
<th>Author</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vere, Steven A.</td>
<td>Splicing Plans to Achieve Misordered goals, <em>Proceedings of the Ninth Int. Joint Conf. on Artificial Intelligence (IJCAI-9)</em>, 1985, pp 1016-21</td>
</tr>
<tr>
<td>Wilkins, David E.</td>
<td>Hierarchical Planning: Definition and Implementation, <em>Proceedings of the Seventh European Conf. on Artificial Intelligence (ECAI-86)</em>, 1986, vol 1, pp 466-78</td>
</tr>
<tr>
<td>Winston, Patrick Henry and Horn, Berthold Klaus Paul</td>
<td><em>LISP</em>, Addison-Wesley Publishing Company, 1984</td>
</tr>
</tbody>
</table>
Appendix D.1-F

The complete code is available separately from:
the SICS Prototype Archives
reference: SICS/PA89/89002A

Swedish Institute of Computer Science
Box 1263
S-164 28 Kista
Sweden