Report LSA-95-10e, Centre for Learning Systems and Applications, Kaiserslautern, 1995.

Combining SNLP-like Planning and Dependency-Maintenance

Frank Weberskirch
University of Kaiserslautern, Dept. of Computer Science
P.O. Box 3049, D-67653 Kaiserslautern, Germany
E-mail: weberski@informatik.uni-kl.de

Abstract

Real world planning tasks like manufacturing process planning often don't allow to formalize all of the relevant knowledge. Especially, preferences between alternatives are hard to acquire but have high influence on the efficiency of the planning process and the quality of the solution. We describe the essential features of the CAPLAN planning architecture that supports cooperative problem solving to narrow the gap caused by absent preference and control knowledge. The architecture combines an SNLP-like base planner with mechanisms for explict representation and maintenance of dependencies between planning decisions. The flexible control interface of CAPLAN allows a combination of autonomous and interactive planning in which a user can participate in the problem solving process. Especially, the rejection of arbitrary decisions by a user or dependency-directed backtracking mechanisms are supported by CAPLAN.

1 Introduction

In real world planning tasks, the performance of a system often decreases because it's impossible to formalize all of the relevant knowledge. A generative planning system is left alone with the general domain specification and has to perform en exhaustive search as preferences between alternatives are hard to acquire. Nevertheless, this control knowledge plays a very important role and highly influences the efficiency of the solution process and the quality of the result. Unfortunately, autonomous problem solving, especially so-called precondition achievement planning, lacks efficient general purpose mechanisms and heuristics to control such a search process (Drummond, 1993).

(Wilkins, 1984) proposed a combination of automatic and interactive plan generation to be a suitable way for solving complex real world problems that cannot be solved autonomously by a planning system within a justifiable amount of time. Such a system that is able to work together with an user (usually a domain expert) will be able to solve even complex problems, because for the human expert often it's easy to decide how to solve some of the goals of a problem although he might not be able to make a general rule from his decision. He probably will solve the main difficulty of a planning problem and might reduce the remaining planning work to a (simple) routine matter. Both, system and user profit from each other: the user often can decide how to solve some critical points of a problem where many alternatives exist but no concrete preferences are defined for the system, the system easily can do uninteresting routine work for the user. Additionally, such a system should offer opportunities of intervention to the user by enabling him to reject decisions taken by the system that are wrong or not adequate from his point of view.

This report describes the CAPLAN (Computer Assisted Planning) architecture that was motivated by the scenario illustrated above. The CAPLAN architecture is based on two pillars:

- an SNLP-like (McAllester and Rosenblitt, 1991) base level planner and
- a mechanism for dependency maintenance to support user interactions, learning, and sophisticated backtracking.

SNLP (McAllester and Rosenblitt, 1991) has been chosen as the basic planning algorithm because of two main reasons:

- The main application domain of CAPLAN is manufacturing process planning (Paulokat and Wess, 1994) that is characterized by the fact that it contains many subgoal interactions. Using the terminology of (Barrett and Weld, 1994) it can be established that this domain is trivially serializable for the SNLP/POCL planner (Barrett and Weld, 1994) but laboriously serializable for a total-order planner. So, a plan-space planner seemed to be a better choice than a state-space planner. (Minton et al., 1991; Minton et al., 1994) confirm this hypothesis.
- SNLP is a sound, complete and systematic planning algorithm (McAllester and Rosenblitt, 1991). Especially, the fact of being complete was an important foundation as this property is a guarantee that the system can find a solution if one exists.

The resulting architecture CAPLAN also was expected to be an interactive system that can act as an planning assistant as described above. Here we need a mechanism for dependency maintenance as, especially, interactions of a user (the rejection of planning decisions) should be possible at any time during a solution process. The system is expected to have an opportunistic behaviour, i.e., if a user rejects a certain planning decision the system should be able to reject exactly all dependent decisions, independent parts of the partial solution should be preserved. For doing this job, the generic REDUX architecture (Petrie, 1991b; Petrie, 1992) was chosen and the SNLP-like planner was built on top of REDUX. Thus, CAPLAN is a combination of SNLP as a planning algorithm and REDUX for dependency maintenance and it can be used for any combination of autonomous and interactive planning.

The CAPLAN system is fully implemented using the object-oriented technology of Smalltalk (VisualWorks 2.0) and runs on serveral hardware platforms. Preferred development platforms are Sun Sparc workstation and IBM PC. The sources of the system can be obtained from the author of this report.

The following pages are organized as follows: Section 2 summarizes the important ideas of the SNLP-like (McAllester and Rosenblitt, 1991; Barrett and Weld, 1994) base level planner of CAPLAN, the plan representation, the extended domain and problem specification facilities. CAPLAN has a very flexible control interface to enable different search and backtrack strategies. Section 3 gives an overview of this control interface. Section 4 then presents the CAPLAN architecture itself which is a combination of the SNLP algorithm with extended mechanisms to represent domain and problem knowledge and the generic REDUX architecture (Petrie, 1991b; Petrie, 1992) that is used to maintain dependencies between planning decisions. The report ends with some conclusions about the presented architecture.

2 The SNLP-like Base Level Planner

The CAPLAN architecture is based on an SNLP-like (McAllester and Rosenblitt, 1991; Barrett and Weld, 1994) domain-independent generative planner. Given a domain and problem specification it searches in the space of partial plans for a solution. In this section we summarize the important aspects of this base level planner. We start with the representation of plans in CAPLAN and the general plan refinement methods of the algorithm. The extended domain and problem specification meachanisms are explained then. The last part recalls the basic planning algorithm and the necessary extensions for CAPLAN.

2.1 Plan Representation

CAPlan is a planning architecture that is based on a plan-space planner. It searches in the space of partial plans. Partial means two things, plan steps are only partially ordered with respect to each other, and variables of steps may only be partially instantiated.

A *partial plan* is a triple (S, O, B) where

- S is set of plan steps,
- O is a set of ordering constraints that define a partial order among the steps, and
- B are the variable binding constraints over variables occurring in plan steps.

Plan steps represent the possible actions in the planning world and are defined by the *operator schemata* of the domain specification (see section 2.3). Each plan step is associated with exactly one operator schema, we say that the step is an instance of the schema. The operator schema defines preconditions, effects and constraints of this step. Of course, there can be multiple different plan steps that are instantiated from the same operator schema.[1]

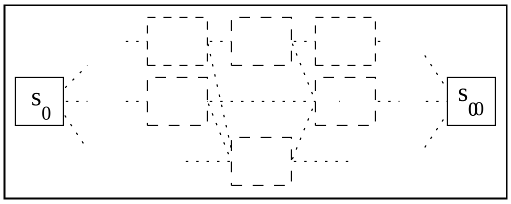

Figure 1: Partial Plan

S always contains two special steps, s_0 and s_∞ which represent the problem to solve. The effects of s_0 correspond to conditions true in the initial state of the problem. Similarly, the preconditions of s_∞ correspond to the goals of the problem. Following the least commitment approach (Weld, 1994) the set of ordering constraints O establishes a partial order among the plan steps in which parts of the plan remain in parallel as long as possible (see figure figure 1), therefore, we also speak about a *partially ordered plan*. The set of binding constraints B contains the conventional codesignation and noncodesignation constraints (Chapman, 1987). CAPLAN additionally allows to use type constraints (see section 2.3).

[1]For this reason, other publications (McAllester and Rosenblitt, 1991; Kambhampati et al., 1995b) use a symbol table that maps step names to domain operators.

The planning process starts with an *initial plan*

$$(S_0, O_0, B_0) = (\{s_0, s_\infty\}, \{s_0 \prec s_\infty\}, \emptyset)$$

that only consists of the two steps s_0 and s_∞ that are necessary to represent the problem and an initial ordering between them. In this initial plan the preconditions of the final plan step s_∞ are not explicitly established, we say these conditions are *open*. It is the job of the planner to find explicit establishments for each open goal. During the planning process, new plan steps with new open preconditions might be added. We call both *open conditions* or *open goals* as long as the planner didn't establish them and denote an open goal g as a tupel (p, s) where p is a preconditions of step s.[2]

Plan Consistency. The notion of a consistent plan is important for later sections. Basically, a plan (S, O, B) is said to be consistent if the following obvious conditions hold:

- No step is ordered before s_0 or after s_∞.
- There are no cycles with respect to the ordering of steps, i.e., there exist no steps $s, d \in S$ with $s \prec d$ and $d \prec v_1, \cdots, v_n \prec s$ for $v_i \in S$.
- The set of variable binding constraints B is consistent.

The first two conditions are also called the *ordering consistency* property of plans, the last one *binding consistency*.

Solutions and Complete Plans. A *solution* to a planning problem is a (totally ordered) sequence of plan steps (actions), which when executed from the initial state, results in a world state in which all goals are satisfied. So, a partial plan (S, O, B) is a kind of shorthand notation (cf. (Kambhampati et al., 1995a)) for a set of totally ordered action sequences that are consistent with constraints in O and B. (McAllester and Rosenblitt, 1991) calls a totally ordered action sequence that is derived from a partially ordered plan a *topological sort*. A topological sort only extends the orderings of a partially ordered plan. A partial plan is *complete* if every topological sort of this plan is a solution to the planning problem. Thus, finding a complete plan is the goal of a planning process.

2.2 Plan Refinement Methods in Plan-space Planning

The planning process starts with an initial plan and refines this plan repeatedly until a termination criterion detects a solution plan or no more plan refinements are possible, i.e., no solution can be found at all. (Kambhampati et al., 1995b) calls this planning process *refinement search* and illustrated the possible plan refinements within this framework.

Given a partial plan (S, O, B) a refinement method is a function that maps a partial plan P_i to another partial plan P_j by adding one ore more elements (steps, orderings, or bindings) to the plan. The following refinement methods have to be considered in SNLP-like planning:

- *step addition* (adding a new plan step)

[2] In the following text we will also call p an open goal if step s is clear from the context, e.g., given a problem (I, G) we might speak about the planning goals $g_1, \cdots, g_n \in G$ and mean exactly $(g_1, s_\infty), \cdots, (g_n, s_\infty) \in G$.

- *simple establishment* (adding an ordering)
- *promotion/demotion* (adding an ordering)
- *separation* (adding variable binding constraints)

Each of these refinement methods modifies the current plan by adding steps, ordering constraints or binding constraints. None of them will ever remove constraints or steps. Thus, the current partial plan is monotonously growing in the number of steps, orderings and binding constraints.

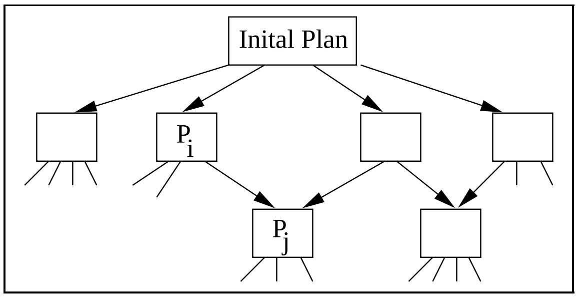

Figure 2: Plan refinement as graph search

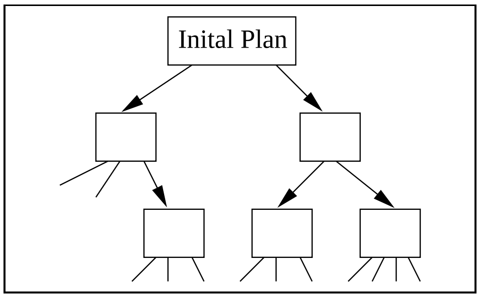

Figure 3: Search tree of SNLP

The search process in the space of partial plans is comparable with a graph search, where graph nodes are partial plans and a directed edge from P_i to P_j indicates that P_j is obtained by applying one of the refinement methods listed above to P_i (see figure 2). In the special case of SNLP this graph in fact is a tree (see figure 3) because of the systematicity of the algorithm as each plan state is reached at most once during a planning episode (McAllester and Rosenblitt, 1991). But from the beginning there was discussion about the use of systematicity (e.g., (Kambhampati, 1993)) and there are examples where it is an advantage to have this property as well as examples where it doesn't improve the planning process.

In fact, this tree can also represents the search process performed by SNLP. The planning process searches through the space of partial plans. The edges of the search tree represent the planning decisions (corresponding to a refinement method) that transform for example plan state P_i into P_j. During search, this tree (or in general the graph) is traversed in some way depending on the concrete search procedure and search control. In case of SNLP we have a bounded depth-first search where the algorithm backtracks if no further consistent refinement is possible or the cost bound is exceeded. Later we will come back to this way of understanding the search process of SNLP as it's helpful to explain different backtracking strategies.

2.3 Extended Domain Specification

CAPLAN is a domain-independent planner and needs to be told about the planning world in which it has to find plans. CAPLAN uses a STRIPS-like notation for domain specifications and it is a typical example for the so called precondition achievement planners (Drummond, 1993)): operators are primitive actions in the planning world that have preconditions and effects. One major difference to many typical planners described around SNLP is the extension of the a domain specification by type taxonomies and, as a result of this, type constraints in the definition of operator schemas.

A domain specifications of CAPLAN basically consists of three parts, a definition of

- *object types*,
- *predicate names* and
- *operator schemata.*

Example domains available for CAPLAN are versions of the well known blocks world domain, the transportation domain (Veloso, 1994), and artificial domains from (Barrett and Weld, 1994; Kambhampati, 1993; Veloso and Blythe, 1994). The actual application domain of CAPLAN is concerned with manufacturing process planning for rotary symmetrical workpieces (Paulokat and Wess, 1994) and will be referred to as the workpiece domain in the following sections. Although this application domain is restricted to rotary symmetrical workpieces it's a quite large domain that contains difficulties of various other domains. It will be characterized in the following section before the elements of a domain specification are explained in more detail.

2.3.1 Characteristics of CAPlans Main Application Domain

The domain we are concerned with in CAPLAN is manufacturing process planning for rotary-symmetrical workpieces to be machined on a lathe. A planning problem in this domain is given by a geometrical description of a workpiece and of a stock. Workpieces description are designed using an AutoCAD application as a tool for the construction process (see figure 4). This AutoCAD application communicates with the planning system to submit geometrical descriptions and to start the planning process for a certain workpiece.

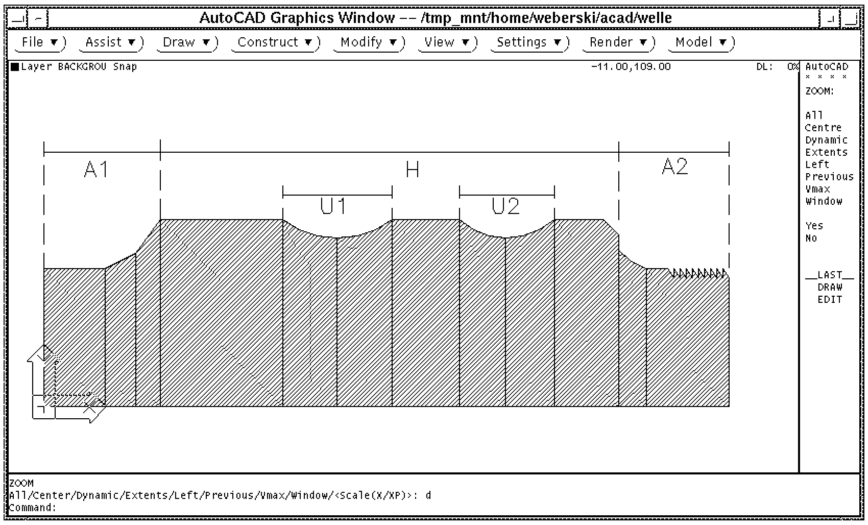

Figure 4: A rotary symmetrical workpiece.

The description of a workpiece is built up from geometrical primitives like cylinders, cones and toroids that describe monotone areas of the outline, possibly augmented by features (threads, undercuts, surface conditions, etc). For such a planning problem a sequence of processing operations is to be found that will machine the workpiece considering available resources (i.e. tools, machines) and technological constraints related to the use of these resources. The manufacturing process starts with clamping the stock on a lathe machine that rotates it at a very high speed. In most cases, the outline of the workpiece cannot be machined in one step but repeated cutting operations are necessary to cut the difference between the raw material and the workpiece in thin horizontal or vertical layers.

For detecting the interactions between parts of the workpiece, a domain dependent system called the *geometrical reasoner* is used. It establishes constraints on the order for manufacturing certain parts of the workpiece (see also section 2.4). As each part of the workpiece constitutes a goal in the problem description, these constraints are interpreted as ordering constraints that must be met by any plan for manufacturing that workpiece. Further, these constraints are stated by the geometrical reasoner before the planning process begins, as they depend only on the geometry of the workpiece and not on available tools or on the clamping material. For example, for the workpiece given in figure 4, the geometrical reasoner establishes that the horizontal processing area H must be manufactured before the undercuts $U1$ and $U2$. These constraints must be met by any partial-ordered plan for manufacturing that workpiece.

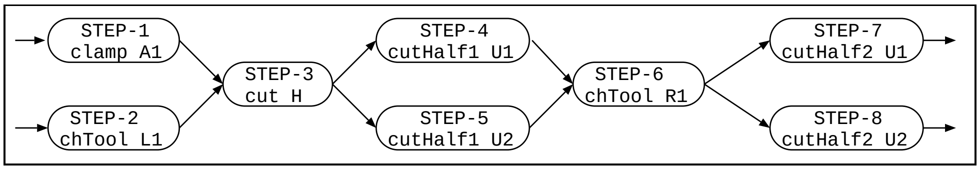

Figure 5: Outline of a manufacturing plan.

Figure 5 shows a plan fragment for the workpiece in figure 4. Boxes represent the plan steps and arcs the orderings among them. There are steps for specifying a fixturing method, inserting cutting tools and for the cutting operations itself. E.g., for processing the second half of the area *U2* in *STEP-8*, using a certain cutting tool (a "right cutting tool", R1) has been chosen (*STEP-6*). The plan in figure 5 also is consistent with the additional ordering requirements established by the geometrical reasoner for this problem, i.e., the processing area H is manufactured before the two undercuts are manufactured.

2.3.2 Types and Constraints

It's quite usual for a planning system to use variables and allow constraints to be defined on these variables. Many planners user typeless variables and only have simple codesignation/noncodesignation constraints (Chapman, 1987) that constrain one variable to have the same or not the same binding like another variable. In CAPLAN all we additionally allow to define types and type constraints.

Types. In CAPLAN all objects of a planning problem must have a certain type, e.g., in the blocks world domain there are the two types `Block` and `Table`. Object types can be organized in taxonomies by assigning at most one supertype to each type. Figure 6 shows a part of a type taxonomy of the workpiece domain as an example. In this domain we have more than 50 different types that are organized in several taxonomies.

As we will see later, types are used in the definitions of operators and problems: operator schemata can define type constraints over variables, problem definitions specify a type for each of its object.

```
ProcessingArea                    Tool
  ExteriorOutline                   RotatoryTool
    LeftAscendingOutline              LeftRotatoryTool
    RightAscendingOutline             RightRotatoryTool
    HorizontalOutline               FeatureTool
  ExteriorFeature                     Drill3
    Slope                             Drill4
    Finish                            Drill5
    Prickout                          ThreadCuttingTool
    Roundoff                          ThreadTappingTool
    Thread                            MillingTool
```

Figure 6: Part of a type taxonomy (workpiece domain).

Constraints. CAPLAN allows two kinds of constraints: simple codesignation or noncodesignation constraints (Chapman, 1987) that can be found in nearly every planner, and, as a major extension to standard SNLP, we have type constraints.

Codesignation and noncodesignation constraints are denoted as

- `Same(x,y)` and
- `NotSame(x,y)`

and force variables `x` and `y` to have the same or a different binding.

There are two possibilities for type constraints:

- `IsOfType(<TYPE>,x)` forces variable `x` to be of type <TYPE>,
- `IsNotOfType(<TYPE>,y)` forces variable `y` not to be of type <TYPE>.

Both types of constraints can be used within the specification of operator schemata (see section 2.3.4).

2.3.3 Predicates

Predicates express relations between objects and are obligatory to be able to describe situations (states) in the planning world as well as effects or preconditions of operators. The definition of a predicate consists of its name and arity. The blocks world domain, for example, contains a binary predicate `On` and an unary predicate `Clear`. The workpiece domain of CAPLAN is much more complex and has 38 predicates. Figure 7 shows some examples for predicates of the workpiece domain of CAPLAN.

subarea(U1, H)	processed(s1)
noSubareaThread(A1)	processed(H)
leftRotatoryTool(lm1)	processed(U1)
unprocessedSide(S1)	processedUndercutHalf1(U2, H)
toolHolderFree()	processedUndercutHalf2(U2, H)

Figure 7: Examples for predicates (workpiece domain)

2.3.4 Operators

Operator schemata are the most important part of a domain definition. While types and predicates basically define a language for describing problem situations, operators define the primitive elements a solution is composed of, the possible primitive actions in the planning world.

In CAPLAN domain operators are specified in an extended STRIPS operator formalism (Fikes and Nilsson, 1971) and consist of

- (typed) *preconditions*,
- *effects* (purposes and side effects) and
- *constraints* (codesignation, noncodesignation, and type constraints).

Figure 8 shows an example for an operator schema, the operator `InsertFeatureTool` of the workpiece domain. Operator schemata are instantiated to plan steps during the planning process. This instantiation will replace variables used in the definition of the schema by concrete plannning objects of the problem or planning variables that are bound by the constraint propagation algorithm. The current version of the workpiece domain consists of about 25 different operator schemata.

```
InsertFeatureTool(tool)
  Constraints:  IsOfType(FeatureTool,tool)
      Purpose:  toolInserted(tool)
 Side effects:  ¬toolHolderFree()
     Phantoms:  toolAvailable(tool)
     Subgoals:  toolHolderFree()
```

Figure 8: Example for an operator definition (workpiece domain).

Beside the type taxonomies and type constraints, the example operator from figure 8 shows the two basic extensions in the definition of an operator schemata. CAPLAN distinguishes between

- *purposes* and *side effects* of an operator and
- uses different types of operator preconditions.

Both extensions aren't new in planning but they were presented for hierarchical planners like SIPE (Wilkins, 1988) or NONLIN (Tate, 1977) but not for SNLP-like planners.

Purposes and Side Effects. Distinguishing *purposes* and *side effects* was first proposed for SIPE (Wilkins, 1984; Wilkins, 1988). Purposes are thought to be the main reasons for adding an operator instance to the plan and CAPLAN uses the heuristic to prefer operators with matching purposes to operators with matching side effects.[3] This heuristic is a default that can be overwritten by control components if necessary (see section 3.2.1).

[3]Restricting operator selection to the set of operators which have matching purposes will destroy the algorithms completeness property, so this distinction can only be used for a heuristic.

Typed Preconditions. Another aspect reflected in the definition of preconditions is the distinction between two types of preconditions,

- *normal preconditions* and
- applicability conditions, called *phantom preconditions*.

Phantom preconditions of CAPLAN are also known as *filter conditions* from hierarchical planning, e.g., they are called *only-use-when* conditions in NONLIN (Tate, 1977) or O-PLAN (Tate et al., 1994). Normal preconditions define conditions for the executability of an operator. The idea behind filter conditions is to give the planner some information about the applicability of an operator hoping that they will prevent applying operators in some situations. The primary usage of filter conditions in hierarchical planners like NONLIN or SIPE is to rule out operators, but they do not explicitly consider them for establishment. As discussed below, they are implicitly also necessary to find the correct binding for some variables of an operator schema. So, filter conditions are treated very differently from normal preconditions. (Collins and Pryor, 1992) pointed out that using filter conditions this way within SNLP will lead to loss of completeness.

(Kambhampati, 1995) stated another view on filter conditions on which phantom preconditions of CAPLAN are based. Here, filter conditions are treated like normal preconditions in that they become open goals if a step is added for which filter conditions are defined. The planner explicitly tries to establish filter conditions also, but there is a difference to normal preconditions that can be best explained by the possibilities available for the establishment of an open condition. In general, there are two ways to satisfy a precondition also known as *simple establishment* and *step addition*. Simple establishment searches the plan for existing plan steps that can be used to satisfy a precondition (such a step is called the *establisher* of the precondition). Step addition uses an operator schema of the domain to instantiate a new plan step and add it to the plan as the establisher of the precondition. CAPLAN puts the following restriction on phantom preconditions:

Restriction for phantom preconditions: A phantom precondition is characterized by the fact that only simple establishment will be allowed to satisfy it while for normal preconditions CAPLAN allows both types of establishment.

So, the planner tries to find an already existing establisher for open phantom preconditions. Sometimes however, the planner even will have to *wait* for this establisher to be added to the plan. Of course, the existence of phantom preconditions and the restriction for them affects the completeness of the planning algorithm (cf. (Collins and Pryor, 1992)), so allowing them makes it necessary to modify the control structure of the planning as waiting for an establisher is not what the original SNLP algorithm does (see section 2.5.3).

Filter Conditions in NONLIN. There was one observation about the use of filter conditions in NONLIN that motivated the use of them in CAPLAN. On the one hand, they define conditions for the applicability of an operator and will rule out an operator if it filter conditions cannot be proved to be true. On the other hand, they are necessary to find correct bindings for free variables of the operator if it is applied. An operator is applicable if concrete bindings for all free variables of the operator schema can be found, so the filter conditions play a very important role for correct instantiation of operators in NONLIN. Therefore, they cannot be omitted in NONLIN and should not only be seen as an efficiency hack to speed-up the planner.

If we only have simple STRIPS-like operators in such a case, we also cannot omit filter conditions. But here, we have to add them normal preconditions to an operator although we know that these preconditions must be true in the plan without doing extra work to achieve them (i.e., adding new steps).

Example: Imagine the operator `PutOn(x,y)` from the blocks world domain as defined in (Barrett and Weld, 1994). Its purpose is to achieve `On(x,y)` but it doesn't make any sense to apply this operator if there isn't a `z`, `x`≠`z`≠`y`, with `On(x,z)`. This `z` is the block from which `x` has to be taken and, of course, it doesn't make sense to apply this operator if such a `z` doesn't exist, i.e., `On(x,y)` is already true. Additionally, `z` must be known as it appears in side effects of this operator, `Clear(z)` and ¬`On(x,z)`.
NONLIN here would simply define a filter condition `On(x,z)` to ensure that this variable is bound to the right value whenever this operator schema is instantiated. With the STRIPS-like operators of SNLP we have to declare `On(x,z)` to be a normal precondition. But SNLP doesn't know that this precondition has to be satisfied by phantomization (simple establishment) and will also try to establish it using step addition, so SNLP cannot take advantage of the knowledge that this operator would not make sense if we cannot establish this precondition by simple establishment.

In CAPlan we may declare such preconditions to be phantom preconditions which will prevent adding steps to the plan to establish them.

2.4 Extended Problem Specification

Problem specifications for planners with STRIPS-like operators traditionally consist of an initial state and the planning goals. In CAPlan a problem specification is extended by additionally being able to define orderings between planning goals. These orderings are optional and are used to speed up planning as they give information about orders in which goals have to be achieved in the solution plan. The goal orderings are established prior to the beginning of the planning process by an external reasoner (e.g., a domain specific reasoner or a user) and are thought to help the planner in finding the solution by giving additional constraints that must be met by any solution. E.g., in the domain of process planning the external reasoner is a geometrical reasoner that establishes constraints on the order for manufacturing certain parts of a workpiece (Muñoz-Avila and Hüllen, 1995). The case-based control component CbC of CAPlan takes advantage of them by allowing a more powerful indexing mechanism for the case base based on this goal orderings (Muñoz-Avila and Hüllen, 1995).

A *planning problem* in CAPlan is a triple $(I, G, <_G)$, where

- I is the initial state consisting of a set of predicates that describe the initial situation,
- $G = \{g_1, \cdots, g_n\}$ is the set of goals and
- $<_G$ is an ordering relation on the set of goals.

If there are no goal ordering constraints ($<_G = \emptyset$), we have the traditional way of specifying a planning problem with initial state and goals and we will abbreviate $(I, G, \emptyset)$ with (I, G). Otherwise, $<_G \neq \emptyset$, this goal orderings put additional constraints with respect to the ordering of plan steps on potential solution plans, we say a solution plan must be *consistent modulo* $<_G$ to be a candidate.

Extended plan consistency. Given a partial plan (S, O, B) for a problem $(I, G, <_G)$ the goal orderings $<_G$ put some additional constraints on the plan with respect to the ordering of the establishers of the ordered goals, i.e., the plan steps that we chosen to achieve the goals.

The plan (S, O, B) is said to be *consistent modulo* $<_G$ if the following conditions hold:

- (S, O, B) is a consistent plan (see definition of plan consistency in section 2.1).
- For all $g_1 <_G g_2$ with $s_1 = establisher(g_1)$, $s_2 = establisher(g_2)$ the ordering O of the plan must contain $s_1 \prec s_2$.

Figure 9 illustrates the second condition. The problem specification contains a goal ordering $g_1 <_G g_2$ for two goals $g_1, g_2 \in G$, s_1 is the plan step that achieves g_1 (the establisher of g_1) and s_2 is establisher of g_2. Goals like g_1, g_2 that occur in goal ordering definitions will be referred to as *ordered goals*. This goal ordering forces that in any solution plan g_1 is achieved before g_2. In terms of plan elements it forces the ordering constraint $s_1 \prec s_2$ to be added to the plan.

Problem:

$(I, \{g_1, g_2\}, \{g_1 <_G g_2\})$

Ordering constraint on solution:

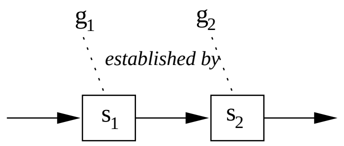

Figure 9: Ordering constraints on planning goals

Interpretation of Goal Orderings. There are relations between the extended problem specification and *hierarchical task network (HTN) planning* (Erol et al., 1994a; Erol et al., 1994b) where we have an initial task network that describes a certain problem by specifying a set of partially ordered tasks that have to be solved to solve the problem. This partially ordered set of task is comparable to the partially ordered set of goal in CAPLAN.

There is an interesting way of explaining the co-operation of a domain definition D with the orderings of an extended problem definition $(I, G, <_G)$:[4] the orderings in $<_G$ combined with the domain D can be seen to be equivalent to a problem-specific domain definition D' for this problem in which the special ordering requirements are taken into consideration (see figure 10). It can be imagined that the operator definitions of D' itself will force orderings equivalent to the orderings in $<_G$ of the extended problem specification. But as the goal

$$(I, G, <_G) + D \;\approx\; (I, G, \emptyset) + D_{<_G}$$

where:

- D is a problem independent domain specification and
- $D_{<_G}$ is a problem-specific domain

Figure 10: Co-operation of extended problem and domain specification

[4]Motivated by S. Kambhampati (private communications in September 1995).

orderings $<_G$ are part of the problem definition they will dynamically create a specific domain definition for the problem that takes its specific ordering requirements into account. So, if we want to use the knowledge $<_G$ about a certain problem, the advantage is obvious: instead of having to find an improved domain specification $D_{<_G}$ for each problem where orderings $<_G$ can be determined, in CAPLAN it's enough to have the general domain specification that doesn't make any assumption about such necessary orderings. CAPLAN will dynamically be able to take advantage of problem specific orderings without any change in the domain specification.

2.5 The Basic Planning Algorithm of CAPlan

CAPLAN is a domain independent planner that is based on the SNLP planning paradigm. Given a domain and a problem specification the planning process consists of repeatedly picking an open precondition and establishing it, or alternatively, resolving conflicts between steps in the plan. Both types of refinement operations add further constraints to the partial plan, they never delete any existing constraints (see section 2.2).

Within the following description of the basic planning algorithm of CAPLAN we mostly use standard vocabulary of SNLP (McAllester and Rosenblitt, 1991). Other notations are adopted from (Kambhampati et al., 1995a).

2.5.1 Basic Concepts

There are two key concepts that are important for SNLP planning. The first is to protect the establishment of a precondition using so-called *causal links*.[5] The second concept, *threats* to causal links, represent interactions between plan steps.

Causal Links: A causal link $s_i \xrightarrow{p} s_j$ records the fact that step s_i has been selected to establish the precondition p of step s_j. The existance of $s_i \xrightarrow{p} s_j$ forces s_i to be ordered before s_j, so if the planner add a causal link $s_i \xrightarrow{p} s_j$ it will always add an ordering $s_i \prec s_j$ to the current plan.

Threats: Causal links help to detect interactions between plan steps called *threats*. A threat t is a tripel $(s_k, s_i \xrightarrow{p} s_j, C)$, where s_k has an effect q (notation: $s_k \mapsto q$). C are the binding constraints that make q necessarily equal to p or $\neg p$. If q is equal to p we call t a *positive threat*, if q is possibly equal to $\neg p$ we call it a *negative threat*.[6]

An important point here is that the definition of threats above doesn't make any assumption about the consistency of the constraints of the threat or orderings of the current plan that are responsible for a threat to be harmful. Others, e.g., (Kambhampati et al., 1995a), only speak about a threat if its constraints are in fact consistent and the threatening step is definitely in parallel to the causal link, i.e., the threat is really harmful. This distinction is important for CAPLAN as we will see later.

[5]There are other names for the same idea, e.g. *range* (Tate, 1977), *protection interval* (Kambhampati and Hendler, 1992), but with (McAllester and Rosenblitt, 1991) the term *causal link* became well known.

[6]In this terms SNLP and NONLIN differ by the fact that SNLP, a descendant of NONLIN, resolves positive and negative threats while NONLIN only cares about negative threats.

Active and Potential Threats. Motivated by the fact that the validity of a threat depends on the elements of the current plan and based on the definition of threats given above, CAPLAN explicitly distinguishes between two kinds of threats: (let $t = (s_k, s_i \xrightarrow{p} s_j, C)$ be a threat and (S, O, B) the current plan)

Active threat: A threat is called *active* if s_k can come in between s_i and s_j (i.e., $s_i \prec s_k$ and $s_k \prec s_i$ are consistent with O) and all necessary binding constraints C are consistent with the set of current binding constraints B. This is the kind of threat considered in the standard SNLP.

Potential threat: Otherwise the threat is called *potential* and either $s_k \prec s_i \in O$ or $s_j \prec s_k \in O$ is valid or the necessary binding constraints C are inconsistent with the set of current constraints B. Potential threats are always annotated with a justification that encapsulates the reason for which the threat is potential (see section 4.4.2).

A potential threats records a dependency between a causal link and a step that currently is not harmful, either because of existing orderings or binding constraints. The reason that makes a threat to be potential is determined by CAPLAN and is stored as an annotation of this threat. The advantage is that CAPLAN never has to check the same threat twice as an evaluation of this annotation is enough to see whether the threat is still potential or not. Thus, the costs of the threat computation are reduced. Section 4.4.2 will show that such a reason that justifies a threat to be potential is stored in the dependency network of CAPLAN with the effect that evaluation is done automatically by the propagation algorithm for the dependency network.

In fact, the distinction of these two kinds of threats doesn't cause more computational overhead than threat computation of SNLP does. SNLP, that only considers the active threats, has to check exactly the conditions mentioned above for the two kinds of threats, i.e., checking the consistency of constraints C and orderings between s_k and the steps of the causal link. While SNLP will not care about a threat if it is only potential, CAPLAN also collects this kind of threats and the results of consistency tests to save work later on.

Besides that, later sections will show that the distinction is also helpful for CAPLAN as a planning assistant when it allows arbitrary planning decisions to be rejected. There are two main reasons why arbitrary (non-chronological) rejection can happen:

- *user interactions*, a user wants to modify a certain part of the plan by rejecting planning decisions, or
- *dependency-directed backtracking* that doesn't backtrack chronologically and so has to keep track of dependencies between planning decisions.

In both cases no recomputation of all threats possibly occurring in the plan after a rejection is necessary as the evaluation of the annotations of all known potential threats is enough to find all possible active threats after the rejection.

2.5.2 The Planning Algorithm

Figure 11 shows the basic planning algorithm of CAPLAN that was derived from the POCL algorithm (Barrett and Weld, 1994). There are only a few differences to POCL that are due to the distinction between active and potential theats (section 2.5.1), other extensions are summarized later.

Algorithm $CAPlanBase((S,O,B),G,L,T,T_{pot})$

1. **Termination:** If $G = \emptyset$ and $T = \emptyset$, then stop (with success).
2. if $T \neq \emptyset$ then
   (a) **Threat selection:** *t := select-threat(T)*
   (b) **Threat resolution:** *op := select-protection(t)* **(backtracking point)**
   $S' = S$
   $O' = O \cup orderings(op)$
   $B' = B \cup bindings(op)$
   (c) **Threat update:**
   $T_p := \{t | t \in T \text{ potential threat}\}$
   justify-threats(T_p)
   $T' = T - \{t\} - T_p$
   $T'_{pot} := T_{pot} \cup T_p$
3. else if $G \neq \emptyset$ then
   (a) **Goal selection:** (p, s_{need}) := *select-goal(G)*
   (b) **Goal establishment:** *op := select-op*(p, s_{need}) **(backtracking point)**
   $s_{add} := establisher(op)$ where $s_{add} \in S$ or new step that adds p before s_{need}
   $S' = S \cup \{s_{add}\}$
   $O' = O \cup \{s_{add} \prec s_{need}\}$
   $B' = B \cup \{\text{binding-constraints, so that } s_{add} \text{ adds } p\}$
   $G' = G - \{(p, s_{need})\} \cup preconditions(s_{add})$
   $L' = L \cup \{s_{add} \xrightarrow{p} s_{need}\}$
   (c) **Threat detection:**
   $T_p := \{t | t \in T \text{ potential threat in}(S', O', B')\}$
   justify-threats(T_p)
   $T' = T \cup \{t | t \text{ new active threat in}(S', O', B')\}$
   $T'_{pot} := T_{pot} \cup T_p$

   endif
4. **Recursive call:** $CAPlanBase((S',O',B'),G',L',T',T'_{pot})$

Figure 11: The basic planning algorithm

The parameters of the algorithm are:

- a partial plan (S, O, B),
- a set G of open goals (as defined in section 2.1) and
- the following SNLP-specific parameters:
  - a set L of causal links,
  - a set T of unresolved (active) threats and
  - a set T_{pot} of detected potential threats.

Given a planning problem $(I, G, <_G)$ the planning process starts with the initial plan and the initial planning goals. Planning proceeds by making decisions about the resolution of threats (2.(a)-2.(c) in the algorithm) and the establishment of open conditions (3.(a)-3.(c) in the algorithm). Each possible decision at the two backtracking points is represented with an operator that will add elements to the plan before threats are checked or updated again:

- *Threat resolution* adds ordering or binding constraints depending on the selected threat resolution method (promotion, demotion, separation).
- *Threat update* finds previously active threats that become potential because of the selected protection.
- *Goal establishment* also selects an operator that uses an existing step or adds a new step as the so-called *establisher* for the open condition. Simple establishment is preferred to step addition in CAPLAN here as this is important for this algorithm to find minimal plans.
- *Threat detection* searches for new threats and divides them into active and potential ones.

As mentioned before, potential threats are annotated with the reason for being potential (see also section 4.4.2). This has some advantages for the threat detection mechanism. Threat detection can be devided into two subtasks, each of which works incrementally even if an arbitrary decision is rejected:

1. Finding new threats after a step addition: Only the new step has to be considered as a source for new threats to existing causal links (potential or active). Of course, theats to the added new causal link may occur too.
2. Updating potential threats: Threat update and threat detection both have to check the currently active threats to see whether some have become potential because of the selected threat resolution or goal establishment method.

Finally, the recursive calls to the algorithm terminate if there are no open conditions or unresolved threats left, i.e., $G = T = \emptyset$. If a problem is unsolvable this algorithm will terminate because there is no backtracking point left to which the algorithm could go back (not shown in figure 11).

CAPlan-specific Extensions to the Algorithm

Section 2.3 and section 2.4 gave a detailed description of the domain and problem specification in CAPLAN and already stated that there are some extended representation mechanisms. Section 2.5.1 introduced the notion of potential threats.

All these extensions have consequences on the basic planning algorithm:

- The algorithm in figure 11 already takes the distinction between two kinds of threats into consideration: threat detection and threat update distinguishes potential and active threats and annotate potential threats with justifications.
- CAPLAN allows type constraints in operator specifications, so the algorithm needs a more powerful constraint tester than the standard SNLP algorithm as steps 2.(b) and 3.(b), the two backtracking points of the algorithm, implicitly have to check consistency of possibilities for threat resolution and goal establishment.

The extended problem specification mechanism of CAPLAN allows to define orderings on the set of goals of a problem that put additional constraints on a plan. This also forces some extensions of the algorithm:

- The algorithm has to check the extended plan consistency that also considers the additional goal orderings specified in a problem description. This plan consistency criterion is more restrictive than the simple consistency criterion.

- Whenever there are ordered goals and both are established by a plan step an ordering contraints between both steps must be added. This reduces the extended consistency test to the normal one with an extended set of orderings in the plan. It can be achieved by extending step 3.(b) of the algorithm that is responsible for goal establishement:

  3.(b): (...)
  if $isOrderedGoal((p, s_{need}))$ and $t = establisher((p, s_{need}))$ then
  for all $(p, s_{need}) <_G (p', s')$ with $t' = establisher((p', s'))$ do: $O' := O' \cup \{t \prec t'\}$
  for all $(p', s') <_G (p, s_{need})$ with $t' = establisher((p', s'))$ do: $O' := O' \cup \{t' \prec t\}$
  endif

Finally, section 2.3 introduced the notion of phantom preconditions or filter conditions in SNLP that force simple establishment for this type of conditions. The next section explains the modification of the algorithm that are necessary to retain the completeness property.

2.5.3 Modifications for Filter Conditions

Allowing normal preconditions and filter conditions and establishing both in CAPLAN (see section 2.3.4) causes the most complicated modification of the algorithm. (Collins and Pryor, 1992) already investigated the implementation of filter conditions in a complete and correct partial-order planner and concluded that any such implementation fails to archieve the functionality of filter conditions. These implementations use a modified version of the SNLP planner in which filter conditions are ignored until all outstanding subgoals of a partial plan are satisfied. So, filter conditions do not do much filtering in this implementation. (Collins and Pryor, 1992) stated that it is problematic to consider filter conditions earlier in the planning process as in an incomplete plan it cannot be stated with certainty that a particular filter condition cannot be established since it is possible that such a step will subsequently be added to the plan.

But: While it is impossible to rule out an incomplete plan based on blocked[7] filter conditions, there are still a number of ways to take advantage of them during the planning process. If a normal precondition is blocked the algorithm from figure 11 will backtrack, i.e., it will go to the last backtracking point, reject the selected alternative and try another one. This behaviour is problematic for phantom preconditions (cf. (Collins and Pryor, 1992)). Treating filter conditions like normal preconditions would then force backtracking on goal selection whereas the fact that SNLP does not backtrack over goal selection was mostly seen as an advantage.

In CAPLAN we make a compromise: phantom precondition are allowed in the definition of operators and used during the planning process. Basically, phantoms are established like normal conditions and they are preferred as the often do nothing else that binding free

[7] A goal is called *blocked* if there is no consistent way (operator) to establish it. Normally, a blocked goal will cause backtracking to be invoked.

variables of an operator (see section 2.3.4). The difference is that only existing establishers will be used. But if a phantom precondition is blocked the algorithm behaves different than in case a normal precondition is blocked. Blocked phantoms neither are completely ignored or only used to penalize partial plans with unestablished filters (Collins and Pryor, 1992) nor backtracking over goal selection is used to retain the completeness of the algorithm. Instead CAPLAN reacts as follows if there exists at least one blocked phantom:

1. If there is no unestablished normal precondition left, then the planner backtracks as no step addition is possible that could supply the blocked phantom with an establisher.
2. If there is a normal precondition that is not blocked, then ignore the blocked phantom and select one of the not blocked phantoms or normal goals.
3. Whenever a phantom is blocked it doesn't make much sense to resolve a threat as threat resolution will never change anything with the blocked phantom. The only exception are so-called *forced threats* (Peot and Smith, 1993), threats for which less than two resolution possibilities are left. These are selected and resolved even if a phatom is is blocked.
4. If there is an unresolvable active threat, then backtracking takes place.

The standard SNLP algorithm is complete, i.e., it will find a solution whenever one exists. Now we will argue that completeness isn't affected by the strategy that allows filter conditions in SNLP described above. For each of the four cases listed above we have to show why backtracking does jump over a solution:

- In case 1, we definitely loose a solution because there is no normal precondition that could lead to a new plan step necessary to serve as the establisher of the blocked phantom. This is the simplest case already stated in (Collins and Pryor, 1992) where a phantom is ignored until all outstanding subgoals are satisfied and then helps to rule out plans based on unestablished filter conditions.
- Case 2 follows the simple strategy of ignoring the blocked phantom. As there is another unestablished normal preconditions there is still the possibility that a new step is added. Avoiding backtracking here is necessary to be sure that the algorithm doesn't jump over a solution.
- Threat resolution will never add a plan step, so, it never will help to establish the blocked phantom directly. But combining the strategy above with **DUnf** from (Peot and Smith, 1993) seems to be promising. A forced threat can be resolved in at most one way, so resolving it will add one more constraint to the plan without creating a real backtracking point with a branching factor bigger than one. Especially no solution will be lost as the forced threat also must be resolved in the solution.
- Case 4 is comparable to the **DRes** strategy from (Peot and Smith, 1993) and also cannot jump over a solution as all threats must be resolved in a solution. So, by selecting an unresolvable threat the planner backtracks because of the threat not because of the blocked phantom.

These arguments show that the compromize allows filter conditions on the one hand, and on the other hand blocked filter conditions do not immediately cause backtracking and cause the algorithm to becaome incomplete. Section 3.2.1 will describe the control component **SNLP+** for CAPLAN that implements the compromise for taking advantage of filter conditions as explained above.

3 Controlling the Planning Process

The control structure of SNLP is very rigid: it always resolves threats before going on with open conditions. (Harvey et al., 1993) has shown that this strategy is not inevitably necessary for the algorithm to be complete. This motivated other control strategies (e.g., (Peot and Smith, 1993; Joslin and Pollack, 1994)).

CAPLAN has a flexible control interface to be able to implement and test certain control strategies. The next sections summarize the decision points of the planning algorithm, present the control interface that allows to define control strategies and describe some important control components available for CAPLAN, e.g., a control component for a combination of autonomous and interactive planning.

3.1 Decision Points of the Algorithm

The algorithm from figure 11 has four direct decision points in which an alternative has to be chosen from a set of possible candidates. These decision points are:

- threat selection (figure 11: 2.(a) *select-threat(T)*),
- threat resolution (figure 11: 2.(b): *select-protection(t)*),
- goal selection (figure 11: 3.(a): *select-goal(G)*) and
- goal establishment (figure 11: 3.(b): *select-op(g)*).

Goal and threat selection are not backtracking points. Given the fact that the algorithm is complete even if threats aren't resolved immediately (Harvey et al., 1993), there is a kind of *meta decision point* beside the decision points listed above: the algorithm has to decide whether to resolve threats or establish open conditions if both is possible.

3.2 The Control Interface of CAPlan

In CAPLAN the control aspect is taken into account using exchangable *control components* (figure 12) for making the concrete selection at the decision points (including the meta decision point) of the algorithm. Therefore, it is necessary to modify the algorithm from figure 11 in a way that steps 2 and 3 can be processed in any order depending on the meta decision of the current control component.

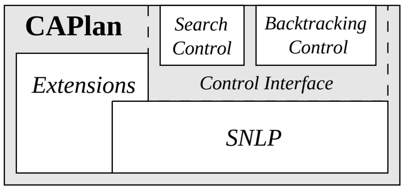

Figure 12: Exchangable Control Components

Another point for control decisions is backtracking. SNLP always was presented as an algorithm that performs simple chronological backtracking. CAPLAN is more flexibel by using exchangable *backtracking control components* to define the backtracking behaviour of the planning algorithm. Besides the chronological backtracking there is a simple dependency-directed strategy (backjumping) available for CAPLAN that retains the completeness property of the algorithm.

3.2.1 Control Components

A control component defines the behaviour of the planning algorithm at the decision points listed in section 3.1. Each control component has to define a set of five selection and decision functions which correspond to decision points of the algorithm:

- *process-threats(G, T):* This function decides what to do first, whether to resolve a threat from T or to establish a goal from G.
- *select-threat(T), select-goal(G):* These functions define which threat or goal should be processed next.
- *select-protection(t), select-op(g):* These two functions represent the selection decisions at the backtracking points of the algorithm. As said before, *select-op(g)* will prefer to select operators that perform simple establishment.

Figure 13 sketches parts of a modified version of the planning algorithm from figure 11. It shows how the five selection and decision functions of control components are integrated into the planning algorithm.

Algorithm *CAPlanBase*

1. **Termination-Check**
2. **Task-Selection:**
   if *process-threats(G, T)=true* then
   (a) (figure 11: 2.a) *t := select-threat(T)*
   (b) (figure 11: 2.b) *op := select-protection(t)* ...
   (c) (figure 11: 2.c) ...

   else

   (a) (figure 11: 3.a) (p, s_{need}) *:= select-goal(G)*
   (b) (figure 11: 3.b) *op := select-op*(p, s_{need}) ...
   (c) (figure 11: 3.c) ...
3. **Recursive Invocation**

Figure 13: Enbedding control components into the planning algorithm

Examples for Control Components: After the overview of the general mechanism for defining control components we now give two examples for the concrete implementation of main parts of two control components for CAPLAN:

- The simplest control component implements the standard SNLP strategy. The main characteristic of SNLP is to prefer threat resolution to goal establishment what is expressed in the definition of the *process-threats(G,T)* function (figure 14). Additionally, SNLP prefers simple establishment to step addition.

```
process-threats(G,T)
if T = ∅ then
    return false
else
    return true
```

```
select-op(g)
if conflictSet(g) = ∅ then
    backtrack
else if ∃op ∈ conflictSet(g) and isSimpleEst(op) then
    return op
else
    return some op ∈ conflictset(g)
```

Figure 14: Control component SNLP

- Figure 15 shows the main control functions for the strategy **SNLP+** to take advantage of filter conditions (see section 2.5.3) by preferring the establishment of filter conditions as this will bind free variables. But here the behaviour at the meta decision point (*process-threats(G,T)*) changes with the presence of blocked phantom preconditions. Phantom preconditions for which no consistent simple establishment exists in the current plan (blocked phantoms) result in deferring backtracking until no normal precondition exists. Additionally, **SNLP+** prefers forced threats in this situation as they don't increase the branching factor.

```
process-threats(G,T)
if ∃g ∈ subset-of-phantoms(G) and blocked(g)
then
    if ∄g' ∈ subset-of-normal-goals(G) then
        backtrack
    else if ∃t ∈ T and forced(t) then
        return true
    else
        return false
else
    ... default selection of SNLP ...
```

```
select-threat(T)
if ∃t ∈ T and forced(t) then
    return t
else
    ... default selection of SNLP ...
```

```
select-goal(G)
if ∃g ∈ subset-of-phantoms(G) and ¬blocked(t)
then
    return g
else
    ... default selection of SNLP ...
```

Figure 15: Standard control component SNLP+ of CAPlan

It's quite easy to define other control components, e.g., control components that delay threats in certain situations (Peot and Smith, 1993) or the well known LCFR (Least Cost Flaw Repair) presented in (Joslin and Pollack, 1994) that often is a very successful control component although it does a lot of computations before it selects a flaw. All we have to do is to define the behaviour at the decision points that are reflected in the strategy.

3.2.2 Controlling Backtracking

If goal establishment or threat resolution fail because there is no consistent alternative left the planning algorithm backtracks. Similar to the control components for selecting among different alternatives for threat resolution or goal establishement (section 3.2.1), CAPlan uses so-called *backtracking control components* to enable different backtracking strategies. A backtracking control component defines the exact behaviour whenever the planner has to backtrack. In general, a backtracking control component determines the decision point to which the planner has to go back. Additionally, it rejects one or more decisions that are found to be wrong by this component.

Chronological Backtracking. The simplest implemented backtracking strategy is the default strategy of SNLP, *chronological backtracking*: backtracking here means to go back to the last backtracking point, reject the current operator and select another consistent one or to go back again until a consistent operator is found. To do this, the algorithm needs to keep track of the order in which it processes open conditions or threats.

As mentioned in section 2.2, the search process with chronological backtracking can be represented as a tree (figure 16). Nodes are the goals that are selected to work on in a certain plan state, where a goal can be to resolve a threat or to establish an open condition.[8] Leafs in the tree represent a plan state were unresolvable inconsistencies occurred or a solution is found. Edges from a node n_i to successor nodes n_j represent a refinement operator (protection or establishment operators) for goal n_i. We also speak about a *decision* d_{ij} as the specific corresponding to it has been selected from a set of alternatives.

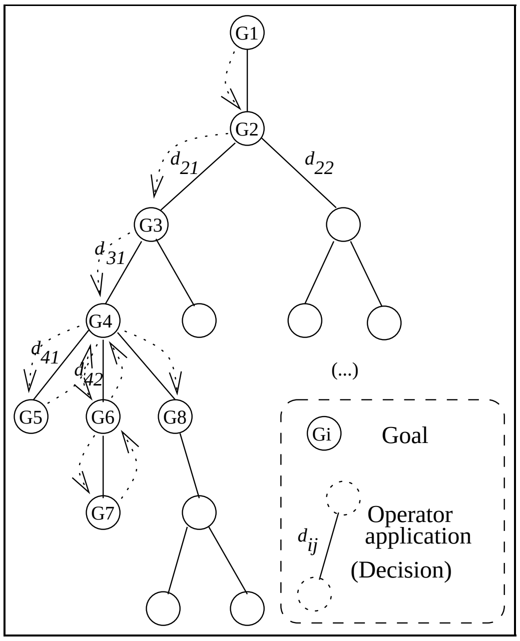

Figure 16: Chronological backtracking

Traversing the search tree in the order G1, G2, ..., G5, G4, G6, G7, ... is what chronological backtracking leads to. The algorithm simply goes back to the last backtracking point no matter what the real reason for this failure was.

Chronological backtracking is a simple backtracking strategy. A more sophisticated *dependency-directed backtracking* mechanism is implemented for CAPLAN (see section 4.5) that is motivated by the *backjumping* strategy (Ginsberg, 1993).

[8] Each node labeled G_i represents a certain plan state (S, O, B) and the goal G_i is an unresolved threat or an open condition in this plan that has been selected to be processed next.

3.3 Interactive Planning

As already said in the introduction, CAPLAN was motivated by a scenario where a user can participate in the planning process because a combination of autonomous and interactive plan generation seems to be a promising way for solving complex real world problems. This combination is one way to solve problems for which autonomous generative problem cannot efficiently control the search process.

There are two possibilities how a user can participate in the planning process:

1. A user *selects* among several possible alternatives at a decision point.
2. A user *rejects* parts of an existing plan because, for example, it is wrong or not optimal from his point of view.

CAPLAN supports both aspects. The first one can easily be done using the mechanisms of the control interface of CAPLAN. A special interactive control component **UC (User Control)** allows to control the planning process via a graphical user interface. E.g., the interface in figure 17 shows the current lists of open goals and unresolved threats and enables interactive selection for all possible decision points. Other elements of the graphical user

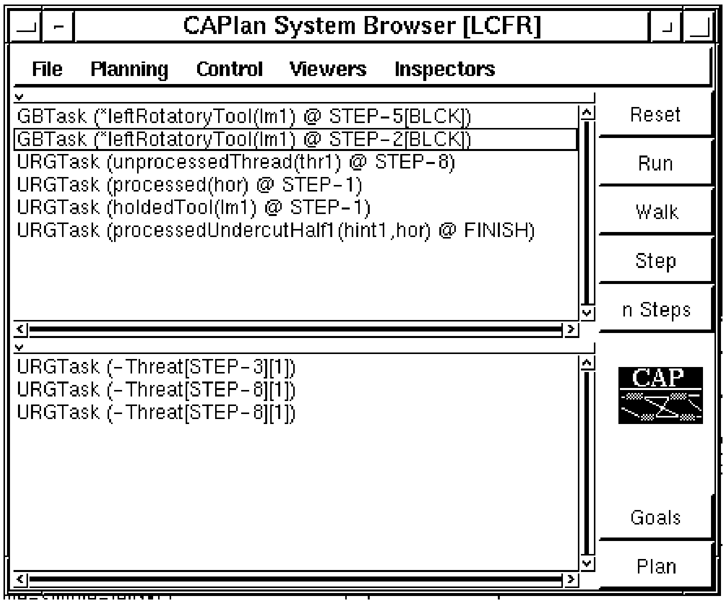

Figure 17: User interface for interactive planning with CAPLAN

interface show the current state of the plan, the so-called subgoal graph (see section 4.3.3) and of the underlying dependency network.

Decision Rejection. Interactive planning also means that a user might have the opportunity to correct plans that are wrong or not optimal from his point of view. The task of the system here is a little more complex: it should be able to identify parts of an existing plan that are affected by an arbitrary rejection. CAPLAN always records dependencies between planning decisions using the generic REDUX architecture (see section 4.2 or (Petrie, 1991b; Petrie, 1992)) and is able to propagate such rejections. The propagation mechanism effectively identifies all other decisions that are affected by the rejection. So, it makes a conservative update of the current situation by removing only those parts of a plan that definitely were based on the rejected part. Some details about the stored dependencies for decision rejection and the advantages to stack oriented archtectures are described in (Weberskirch and Paulokat, 1995). The mechanism used here is similar to what has been published for NONLIN in (Daniel, 1984), but in CAPLAN the integration of dependency maintenance and planning algorithm more straightforward as the complete planning algorithm is realized based on a system for dependency maintenance.

3.4 Control components for CAPlan

So far, the following control components for search control are available:

SNLP+: This control component carries out the control strategy of SNLP and is the default control component. The main difference in comparison with SNLP is that it modifies the control structure to be able to allow filter conditions. But completeness of the algorithm is retained.

UC (User Control): This control component makes CAPLAN act as a planning assistant and allows the user to make all decisions that come up. The degree of interaction with the user is variable, e.g., it can be restricted to the selection of goals or operators or can cover alle possible decision points. A special property of this control component is that it also allows the user to reject arbitrary planning decision and determines the minimal set of dependent decisions that have to be rejected as a consequence of the users rejection.

CbC (Case-based Control): Work on CAPLAN originally was located in the area of case-based plannning (Paulokat et al., 1992; Paulokat and Wess, 1993). Therefore, an additional objecture was to be able to integrate this work in the archtecture described here and enable further improvements. In CAPLAN/CBC (Muñoz-Avila et al., 1995) the case-based control component **CbC** performs this integration. This control component uses episodic knowledge in form of cases to guide the planner in selecting the right operator for an open goal.

For backtracking control there are only two alternatives:

ChronBT: This is the simplest backtracking control component that performs chronological backtracking.

BJ: Here the chronlogical backtracking is substituted by the more sophisticated *backjumping* strategy (Ginsberg, 1993). In section 4.5 the idea of this backtracking strategies is described in more detail.

Several other strategies have been realized for testing, e.g., the *threat delay* strategies from (Peot and Smith, 1993), the *least cost flaw repair* strategy (Joslin and Pollack, 1994) or *threat subsumption* strategy of (Yang, 1992). (Kettnaker, 1995) describes a mechanism to combine several of these strategies to a complex control component for CAPLAN, control components are plugged together from a toolbox of simple elemary strategies.

4 The Planning Assistant CAPlan

After the last sections have described the base level planner and the control interface, this sections gives some more details about the combination of this base level planner and the REDUX architecture (Petrie, 1991b; Petrie, 1992) that provides mechanisms for dependency maintenance. First, we give a general overview of the CAPlan architecture. Section 4.2 will explain important concepts and mechanisms of Redux. Finally, we will describe how Redux is used to build an SNLP-like planning application.

The all components of the CAPlan system that are described in this report are fully implemented using the object-oriented technology of Smalltalk (VisualWorks 2.0) and run on serveral hardware platforms. Preferred development platforms are Sun Sparc (SunOS 4.1.x, Solaris 2.x) and IBM PC (Windows 3.x, Windows 95). The sources of the system can be obtained from the author of this report.

4.1 System Architecture of CAPlan: An Overview

Figure 18 shows a schematic overview of the architecture of CAPlan. It is based on an extension of the generic Redux architecture (Petrie, 1991b; Petrie, 1992). The advantage of Redux is the explicit representation and maintenance of a various dependencies between concepts relevant to planning – goals, operators and elements of a plan (see section 4.2). CAPlan uses the concepts and mechanisms of Redux to represent knowledge about partially ordered plans, the SNLP-like planning process, and dependencies between planning decisions. Redux+ in figure 18 indicates that Redux had to be extended slighly to be adequate for the purposes of SNLP-like planning. Especially, the subgoal tree of Redux had to be extended to a subgoal graph for a correct representation of threats (see section 4.3.3).

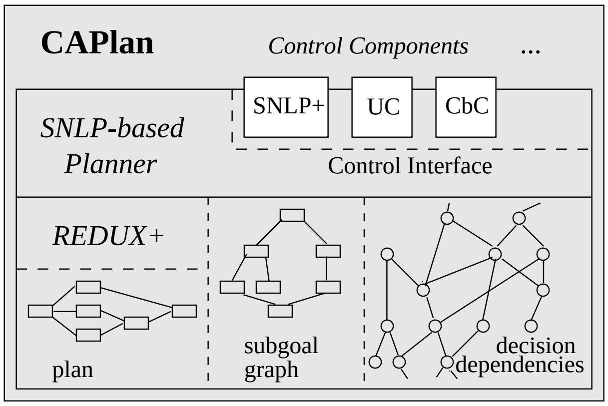

Figure 18: The CAPlan architecture

In this way, CAPlan is a combination of Redux and the SNLP planning paradigm. It inherits properties of both, the facilities for dependency maintenance that are very helpful for interactive planning, sophisticated backtracking strategies or EBL-based learning strategies on the one hand, and a promising partial-order planning paradigm (Barrett and Weld, 1994; Minton et al., 1991; Minton et al., 1994) on the other.

4.2 A Brief Redux Overview

Redux (Petrie, 1991a; Petrie, 1991b; Petrie, 1992) is a generic architecture to represent knowledge about plans and contingencies that occur during planning. It's intend is to make it easier to build applications embodying such knowledge where change is inevitable, e.g., planning or design applications. Redux combines primary elements like goals, constraints, and contingencies. This section will give a brief overview of basic Redux concepts and mechanisms, and the process of problem solving with a Redux application.

4.2.1 Key Concepts of Redux

Central concpets of Redux are *goal* and *operator*. These are also typical for the description of planning or design problems: designing a complex object or planning a complex action is the goal that has to be achieved. Operators represent ways to achieve such goals.[9]

Problem solving proceeds by applying operators to goals. The consequences of operator application are new subgoals and assignments (figure 19). As different operators might be applicable[10] to a goal, we speak about the so-called *conflict set* of operators for a goal. The selection of an operator from a conflict set represents a backtracking point for the search process and is called a *decision*. A decision can be *rejected* if Redux found a reason against the application of an operator (e.g., inconsistencies with other valid decisions). We also say the corresponding operator is rejected. These reasons for rejections are then explicitly represented as justifications for the rejection of the decision. If the conflict set of a goal is empty or if it contains only rejected operators, then the goal is called *blocked*. If an operator has been applied to a goal, this goal is said to be *reduced*.

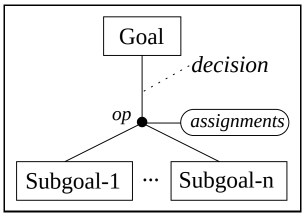

Figure 19: Operator application

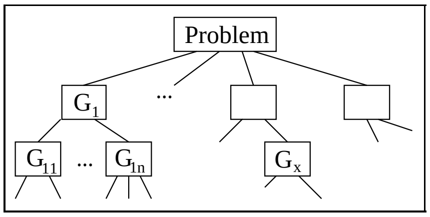

Figure 20: Subgoaling

Goals and subgoals build the *subgoal graph*, one of the basic dependencies recorded by Redux. The root node of the subgoal graph can be understood as the general goal to solve the problem and the immediate successors of the root node are the goals specified in the problem description. Successor nodes originate from operator application. Because of that, in REDUX each goal can have only one parent goal, so the subgoal graph is in fact a tree (figure 20). As we will see later, this is not completely adequate for an SNLP planning algorithm and so has been extended for CAPlan.

[9]Although there are many similarities, goals and operators of Redux should not and cannot be treated as equivalent to goals and operators in planning. The primary use of goals and operators in Redux is to model a search process, e.g., for example the search in the space of partial plans performed by an SNLP-like algorithm. The operators of Redux cannot be elements of partial plans but they have effects that modify plans.

[10]The term *applicable* in the context of Redux operators also should not be mixed up with this term in the context of plan operators or plan steps where applicability requires the preconditions to be valid. In most older planners, e.g., STRIPS (Fikes and Nilsson, 1971) or NONLIN (Tate, 1977), there is no distinction between operators and plan steps. In Redux, an operator is applicable to a goal if it in general represents a way to achieve the goal, so for different types of goals we can define sets of applicable operators.

Assignments represent the effects of an operator application and were originally thought to assign values to variables. In the planning architecture discussed here the operators of REDUX represent the different plan refinement operators for partial plans (see section 2.2) and the assignments of an operator are the modifications of a plan caused by this operator (new steps, orderings, or constraints). The current set of valid assignments determines the elements of the current plan. Checking plan consistency means to check the current set of valid assignments. An important point here is that REDUX will always be able to identify the decision that added a certain assignment. This feature is a preassumption for dependency-directed backtracking mechanisms which have to analyze sets of inconsistent assignments and identify culprits (decisions) for these inconsistencies. The validity of assignments always depends on the validity of the corresponding operator. This and other dependencies are explicitly represented.

Basically, REDUX represents several aspects of goals and operators: validity and local optimality of decisions and dependencies among them (see (Petrie, 1992) for details). Important dependencies for CAPLAN are (see also figure 19):

- subgoals depend on their parent goal,
- subgoals depend on the decision that added them,
- decisions depend on the goal they are applied to,
- assignments depend on the decision that added them.

Petrie suggested a TMS-based implementation for the maintenance of this dependencies. Each aspect, e.g., the validity or blockade of a goal or the necessity to reject a decision, is represented with a node in the TMS that can be justified. Thus, the dependencies are expressed by justifications of nodes in the TMS. The rejection of a decision basically means to add a justification to the node representing the reason for the rejection of the decision. A propagation algorithm is used to determine the state of goals, decisions, and assignments.

4.2.2 Backtracking and Replanning

In general, a problem solver based on REDUX is not assumed to be able to find a solution at once. It will make decisions that are locally good but may not be suitable in the context of the complete problem. This will lead to situations in which no selectable operator exists for an unreduced goal: in terms of REDUX this goal is *blocked*. REDUX, then, invokes a backtracking mechanism to solve this blockade, i.e., the concrete backtracking mechanism must identify one or more decisions and reject them, so that this goal is no longer blocked.

Backtracking isn't the only situation that leads to the rejection of decision. REDUX also defines the concepts of the *admissibility* and the local *optimality* of a decision (so-called *pareto optimality*). As these concepts are unimportant for CAPLAN they are omitted here and the reader may look at (Petrie, 1991b; Petrie, 1992) for a detailled discussion.

In general, REDUX allows arbitrary decisions to be rejected any time and will force replanning for goals that become unreduced because of the rejection of a decision. There is one special property of the rejection mechanism, the distinction between the necessity to reject a decision (*rejection*) and the real process of retracting the decision (*retraction*). Both are connected by the following relation:

1. The rejection of a decision always results in a retraction of the decision.

2. If an operator is rejected (so also retracted) and the reason for rejecting it becomes invalid, REDUX will NOT automatically change the decisions state, i.e., the fact that it is retracted.

Instead of an automatic change of the state of an retracted decision REDUX will only inform the problem solver about this event and give it a chance to react if necessary. This second point is one of the reasons why REDUX is more than a simple JTMS in which relations that are not bi-directional cannot be represented.

4.2.3 Problem Solving with Redux

Problem solving with REDUX means to reduce goals and to keep track of inconsistencies among the assignments of operators and of blocked goals.

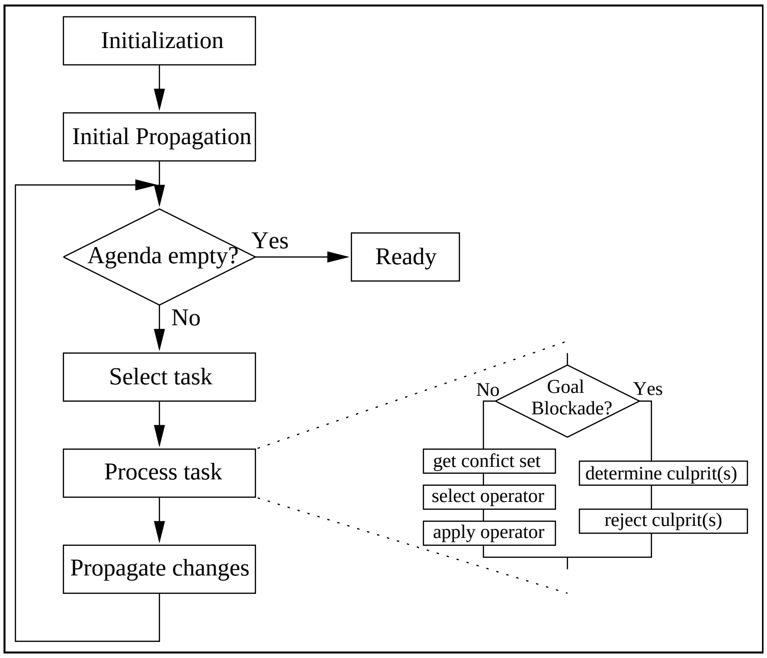

Figure 21: Problem solving with REDUX

REDUX is an agenda driven system: for each open goal, i.e., a goal for which no valid operator has been selected, there is a task on the agenda representing the necessity to find an operator for this goal. Blocked goals are also represented with tasks that indicates that something has to be done to handle this blockade. A selection mechanism picks the most important task from this agenda and processes it (see figure 21):

- In case of an *open goal* it selects and applies an operator that is both, applicable to the goal and consistent with the current plan. At this point REDUX needs application specific modules to check the consistency of an operator and to create justification for inconsistencies. Operator application then adds new assignments and/or subgoals.
- In case of a *blocked goal* REDUX determines and rejects one or more decisions to resolve the blockade. In general, this will result in some kind of backtracking, possibly dependency-directed backtracking if the blockade is analyzed and decisions that are responsible for it are rejected.

Finally, the changes of processing the task are propagated. As mentioned above, Petrie suggested a TMS-based implementation of REDUX, so propagation of changes is done by the labelling mechanism of a JTMS (Doyle, 1979).

4.3 Building CAPlan on Redux

The last section gave some important details about the REDUX architecture. We will now concentrate on how the planning assistant CAPLAN is built on the underlying REDUX architecture. First we describe how basic concepts of the planning algorithm are mapped to concepts of REDUX. Then some important modules of CAPLAN and aspects related to the extensions of REDUX and the creation of justifications are summarized. The last section presents an alternative to chronological backtracking, the so-called backjumping.

4.3.1 Mapping of SNLP-Concepts to Redux-Concepts

For building CAPLAN based on the REDUX architecture the partially ordered plan representation and the basic concepts of SNLP and CAPLAN have to be mapped to REDUX concepts. The mapping is straightforward since both have the notion of goals and operators. Backtracking points become goals for REDUX and the different alternatives are expressed by the different types of operators that can be applied to such goals.

There are two types of goals:

- goals to establish open conditions (*precondition goal*) and
- goals to resolve threats (*protection goal*).

Conflict sets for this two types of goals consist of operators that realize the plan refinement operators for partial-order planning (see section 2.2).

Each refinement method is represented as a class of operators and concrete instances of it can be chosen to be applied to such a goal. The concept of having assignments associated with operators is used to represent the partially ordered plans: all elements of plans (steps, orderings, constraints) and also causal links become assignments of operators.

The following are the four types of operators necessary to realize partial-order planning based on REDUX. Each of them corresponds to one plan refinement method:

- *phantom operators* for simple establishment, they add a causal link and probably come binding constraints as assignments,
- *domain operators* for step addition, they add a new plan step that corresponds to one operator of the domain specification, a causal link and all constraints specified in the domain definition,
- *ordering operators* for threat resolution by promotion/demotion, they add an ordering,
- *separation operators* for threat resolution by separation, they add one ore more constraints.

Phantom and domain operators are applicable to precondition goals whereas ordering and separation operators are applicable to protection goals. The application of these operators to a goal adds new assignments. The current plan is determined by the current set of valid assignments and applying operators to goals changes the current plan.

4.3.2 Modules of CAPlan

Figure 18 gave a schematic overview of CAPLAN – we will now put this in concrete form with the more detailled figure 22 that highlights the most important aspects of the realization of CAPLAN based on REDUX. The architecture is constituted by two main parts, the system kernel and the user interface.

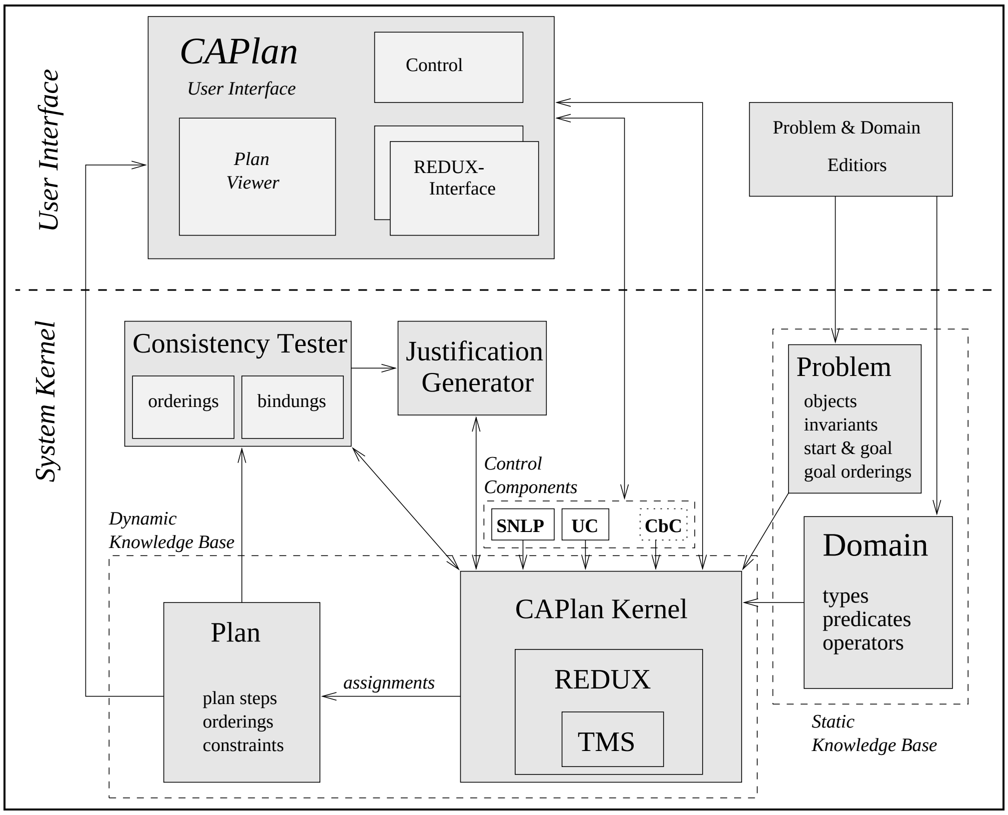

Figure 22: Details of the CAPlan Architecture as a REDUX Application

System Kernel. The *system kernel* consists of the static knowledge base which holds the current domain specifications and the dynamic knowledge base that holds information about the planning progress for the current planning problem.

The dynamic knowledge base, the control components, and the two REDUX-specific, modules consistency tester and justification generator, constitute in detail what is shown in figure 18. The CAPLAN kernel is the already mentioned extension of REDUX that we called REDUX+. The extensions are due to the integration of the control interface and the correct representation of threats. REDUX itself is based on a truth maintenance system for the maintenance of dependencies, in CAPLAN we have a JTMS (Doyle, 1979). Consistency tester and justification generator are two modules required by REDUX for the following purposes:

- The consistency tester checks the extended plan consistency property, as defined in section 2.4, for a given set of new assignments with respect to the current set of valid assignments.
- The justification generator uses the results of the consistency tester to generate various kinds of justifications for the REDUX kernel. The most important justifications are

the *rejection justification* for inconsistent decisions and the *satisfied justification* for potential threats (see section 4.4).

The Redux kernel calls these modules whenever a consistency check is necessary or a justification is required, e.g., when a conflict set for a goal is computed and an operator is to be selected (see section 4.2.3).

User Interface. The *user interface* consists of several components to define and manage domains and problems or to view the current subgoal graph or the plan at various levels of detail (see figure 23). Figure 17 on page 23 already has shown one of the central component

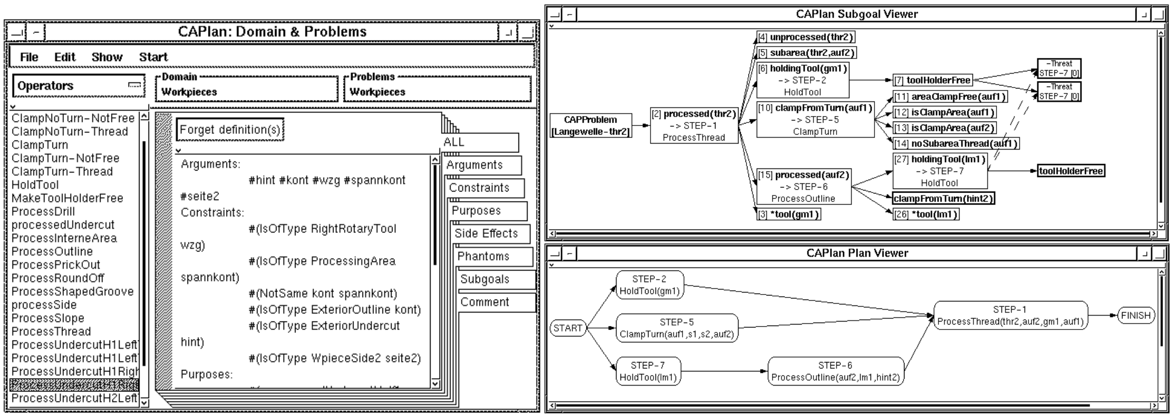

Figure 23: Components of the User Interface

of the user interfaces to control the planner, the interface for interactive planning. Here the user takes the role of the control component and selects at decision points.

Besides that there are special interfaces for viewing the internal dependency structures established by Redux during a planning process , e.g., figure 24 show a viewer for the part of the dependency network for a certain decision.

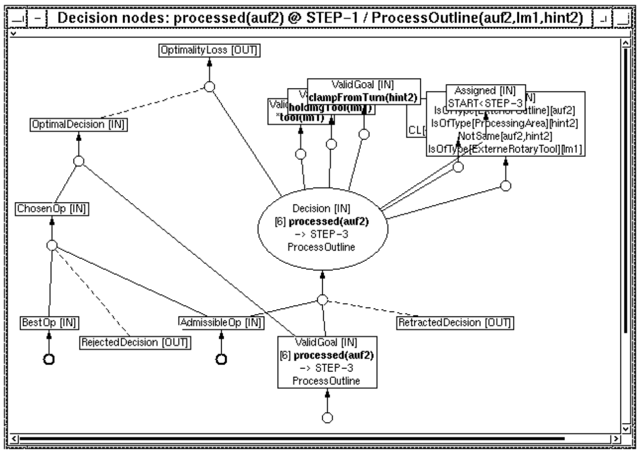

Figure 24: Viewer for Parts of the Dependency Networks

4.3.3 Redux+: Extensions to Redux

As said before, the CAPLAN kernel extends REDUX in the some ways:

- The CAPLAN kernel realizes the integration of REDUX, the control interface, extended representation mechanisms, and several modules used by REDUX (consistency tester, justification generator).
- The CAPLAN kernel also extends the represented dependencies for one type of goals used in CAPLAN, the protection goals.

The second aspect will be explained a little more precise here.

Originally, REDUX makes the assumption that each goal can have only one parent goal, so the subgoal graph is in fact a tree. However this is not adequate for building an SNLP-like planner on REDUX for the following reason: threats are a kind of dependency between two decisions, one that added a causal link as an assignment, another that added the threatening step. As threat resolution is a backtracking point of the algorithm threats must be represented with goals, protection goals in CAPLAN (see section 4.3.1). Protection goals are created as subgoals of the two goals that are responsible for the threat, first, the goal *Goal-1* with an operator that added the threatened causal link, and second, the goal *Goal-2* with an operator that added the threatening step (see figure 25).

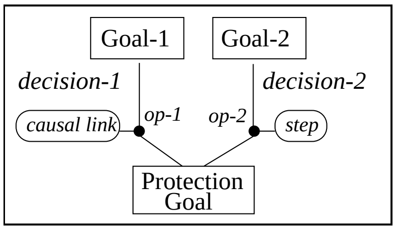

Figure 25: Protection goal

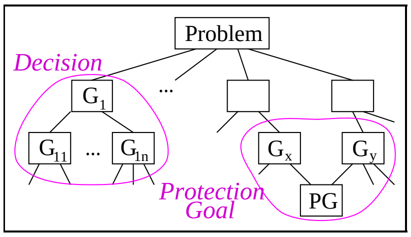

Figure 26: Subgoal graph

Figure 26 shows the consequence of this extension: we have a subgoal graph (not a tree). But this extension is necessary to be able to identify threats and threat resolutions that become invalid after the rejection of a decision (especially for interactive planning or dependency-directed backtracking). For all nodes in this subgoal graph we can demand that a node is valid only if all its predecessors are valid.

4.4 Justifications for Decisions and Goals

Adding justification to the dependency network is the way provided by REDUX to store information that might be interesting later. Section 2.5.1 and section 4.2 already mentioned that CAPLAN stores reasons for threats being potential and for inconsistencies of operators to prevent having to do the same test more than once.

There are three situations in which such justifications are created to store interim results and dependencies of the planning process:

- inconsistencies of operator,
- backtracking, and

- potential threats.

In each situation a justification has to be found for one aspect of the state of a decision or a goal. The first and second concern rejection justifications of decisions, the third one concerns the so-called satisfied justification of protection goals.

4.4.1 Rejection Justifications: Inconsistencies and Backtracking

There are two types of inconsistencies that can occur in a plan: ordering inconsistencies and binding inconsistencies (see section 2.1). Both are detected by the consistency tester modul of CAPLAN (see section 4.3.2) and in both cases the justification generator modul must give a kind of local explanation for the detected inconsistency.

Figure 27 gives an example for an ordering inconsistency. It shows three steps that are ordered with respect to each other. The new ordering $o = s_3 \prec s_1$ is obviously inconsistent because of the existence of the orderings o_1 and o_2. The consistency tester finds them by trying to prove the opposite of the new ordering (s_3 successor of s_1). This proof suceeds because of these two orderings. As long as both are valid, o will be inconsistent, so o_1 and o_2 are the reason for the inconsistency of o.

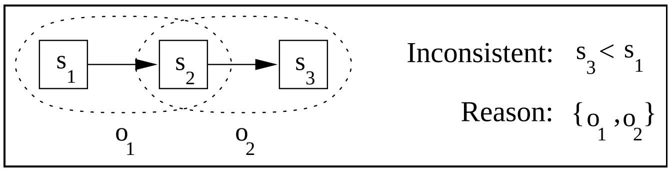

Figure 27: Ordering inconsistency

It's very similar for variable binding constraints. Imagine there are the two constraints $x \neq y, y \neq z$. Then the constraints $x = y$ is inconsistent because of the existence of these two constraints. In general, the constraint system of CAPLAN computes connected components in a graph consisting of variables as nodes and the valid constraints as edges. In the example, both constraints are in the same connected component of this graph. In case of a binding inconsistency, CAPLAN determines which variables are involved in the inconsistency (x, y, z in the example), gets the set of connected components in which these variables can be found and it always considers all the constraints in theses connected components to be responsible for this inconsistency.[11]

In both cases of inconsistencies, we get a set $A = \{a_1, \cdots, a_n\}$ of assignments (orderings or binding constraints) as reasons for an inconsistency. As REDUX always allows to determine the decision that was responsible for adding a certain assignment it's easy to identify the decisions $D = \{decision(a_1), \cdots, decision(a_n)\}$ that are responsible for the inconsistency. They describe the situation where the new constraint is inconsistent. This set of decision might be larger than necessary in some cases, but in general it's hard to reduce the set to the really necessary one. Besides that, for constructing correct justifications we doesn't need the minimal set but a set that is sufficient to describe the situation. CAPLAN exactly uses sets of decisions D as described to justify the rejection of inconsistent decisions.

[11] In the example it surely would be enough to store $x \neq y$ as the reason for this inconsistency, but in general it's quite difficult problem to reduce the set of possible candidates (the connected component of the variables x, y and the constraints $x \neq y, y \neq z$) in such a way.

Chronological Backtracking. Backtracking in REDUX is also realized the rejection of one or more previously selected decisions. Again a justification has to be computed that reflects the reason for the necessity to backtrack. The central idea here is to use the current context in which backtracking over a certain decision occurs as the justification for the rejection of that decision. The context is the the set of valid assignments or their associated decisions.

If we look back at figure 16 on page 22 we see that the current context, a certain plan state, can be described by a set of valid decision, e.g., at node G5 where backtracking is necessary we have the decisions $D_{valid} = \{\cdots, d_{21}, d_{31}, d_{41}\}$ (the edges on the path from the root node to the current node). For chronological backtracking, we also have to keep track of the order in which decisions are taken, so we for example know $d_{last} = d_{41}$. Chronological backtracking now simply means that we have to reject d_{last} and the justifying context for the rejection is the set $D_{valid} - \{d_{last}\}$ of decisions. This justification definitely prevents that d_{last} can be selected in this context again.

Dependency-directed backtracking differs from chronological backtracking only by the fact that some $d_{culprit} \in D_{valid}$ is chosen to be rejected but $d_{culprit}$ will not necessarily be the last decision before backtracking occured. The justification here again can be the full context $D_{valid} - \{d_{culprit}\}$ or a subset of that. The difficulty, however, is finding the culprit and a sufficient subset to justify the rejection.

4.4.2 Satisfied Justifications: Potential Threats

Section 2.5.1 introduced the distinction between active and potential threats in CAPLAN. Potential threats are a kind of interim result of the threat detection process that can be useful for later threat computations. A potential threat $(s_k, s_i \xrightarrow{p} s_j, C)$ is a threat that is already resolved in the current plan, either because the threatening step is already ordered with respect to the steps of the causal link ($s_k \prec s_i$ or $s_j \prec s_k$ is valid), or the necessary constraints C are inconsistent with the current constraints of the plan.

It's easy to see that the justification for this situations are exactly the same as described for inconsistencies in section 4.4.1. The only difference is that the justification now are used for another purpose than the rejection of a decision, here as an annotation of a protection goal saying that it is already satisfied. In REDUX this annotation again is represented as a justification of a node in the dependency network indicating the fact that a goal is satisfied or not. Because of the representation of this interim result in the dependency network, this threat will not have to be tested again as long as its justification saying it is a potential threat is still satisfied.

4.5 Backjumping

Chronological backtracking is a simple backtracking strategy. A more sophisticated backtracking mechanisms is implemented for CAPLAN that was motivated by *backjumping* (Ginsberg, 1993), a simple form of *dependency-directed backtracking.*

Backjumping is characterized by the fact that the failure in finding a consistent operator for a goal is analyzed to get some information about the *real reason* (an earlier decision) for this failure. Backjumping still means to go back chronologically but depending on the results of analyzing the failure it will probably go back more only than one step at once (figure 28). This results in the rejection of more than one decision. In figure 28 for example, the failure at G5 led to backtracking and the analyzing the failure may have determined D_{21} to be

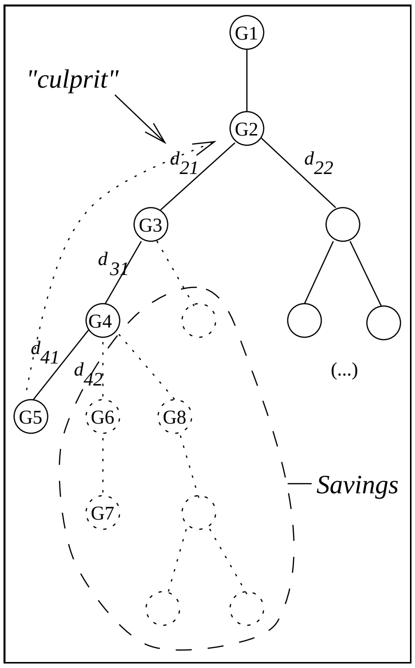

Figure 28: Backjumping

the nearest reason (culprit) for it. The savings with jumping back to G2 (and rejecting the decision d_{41}, d_{31}, d_{21} between G5 and G2) are obvious. The critical point is to guarantee that jumping back will not discard a part of the search space in which a solution can be found.

Analyzing a failure requires an explicit representation of the dependencies between decisions and their effects on the plan (orderings, constraints). This feature is provided by the REDUX architecture (see section 4.2). The conflict set of the failed goal has to be analyzed to identify the reasons. It can contain either operators that failed because of backtracking or operators that failed because of inconsistencies of the assignments of this operator with the current plan. The last point is interesting for backjumping. If G_i fails and the backtracking mechanism is invoked to find the decision point to go back to, backjumping does the following: for each d_{ij} that failed because of inconsistencies of the assignments this rejection is justified with a set D of decision (section 4.4.1). These are likely to be the real candidates D_{cand} for the failure. The algorithm now uses the available stack that stores the order in which decisions were taken and goes back until a decision appears on top of this stack that is in the computed set of candidates or until it reaches the so-called *parent decision* of G_i, i.e., the decision that was taken for an earlier goal G_m that added G_i as a new subgoal. Not to jump over the parent decision is the important condition here as rejecting the parent decision of a goal is always a possible candidate that will remove the failure at this goal by simply removing the goal. (Weberskirch, 1994) gives a proof that this strategy is enough conservative to guarantee that it never jumps over a solution.

Savings of backjumping could be improved if a smaller set of candidates could be determined (the smaller the better). This problem is connected with the problem of finding justification for inconsistencies. There we don't need minimal set of decisions that justify a rejection. For backjumping, a better analysis of the justifications for inconsistencies will improve backtracking behaviour. But it's a hard problem to determine the minimal set of reasons. Determining such minimal sets has not been investigated yet.

5 Conclusions

In this report we presented the main ideas of the CAPLAN architecture for domain independent generative action planning. The most important features of this architectures can be summarized as follows:

- CAPLAN is based on the idea to combine a complete base level planner with mechanisms for dependency maintenance to support a partially interactive planning process.
- The base level planner of CAPLAN is a descendant of the SNLP algorithm (McAllester and Rosenblitt, 1991). It uses extended mechanisms for domain specification (e.g., object types, types constraints in operator definitions, typed preconditions). Problem specifications are also extended by ordering constraints on the planning goals that allow to give additional information about the order in which the goals have to be achieved in a solution plan. Problem specifications where goals can be ordered with respect to each other are also known from HTN-planning (Erol et al., 1994a; Erol et al., 1994b).
- CAPLAN builds the SNLP-like planner on top of the REDUX architecture (Petrie, 1991b; Petrie, 1992) to integrate the planning algorithm and a mechanism for dependency maintenance. This is done by modelling the search process in the space of partial plans of SNLP by means of the REDUX concepts of goals, operators, and assignments.
- Being able to use and integrate various different control mechanisms to control the planning process motivated the control interface of CAPLAN. CAPLAN allows to define a certain control strategy by defining independent modules called *control components*. There are a number of control components available for CAPLAN, e.g., control components that realize the search strategies of SNLP, and extended version SNLP+ that also allows filter conditions, and several other strategies known from literature (Peot and Smith, 1993; Joslin and Pollack, 1994; Yang, 1992).
- The most important and complex control components of CAPLAN so far are the interactive control component and a case-based control component.
  - The interacitve control components **UC (User Control)** realizes a combination of autonomous and interactive planning. A user can participate in the planning process at various levels of detail here. He also can modify or correct a plan by rejecting planning decisions. Here the dependency network helps to propagate such rejections and reject all depending parts of the plan.
  - The control component **CbC (Case-based Control)** in CAPLAN/CBC (Muñoz-Avila et al., 1995; Muñoz-Avila and Hüllen, 1995) combines a case-based planning approach with the generative system CAPLAN.
- CAPLAN not only allows to influence the behaviour of the basic planning algorithm when searching foward but also the backtracking behaviour can be controlled with so-called *backtracking control components*. Chronological backtracking and backjumping are backtracking strategies for CAPLAN that are currently available.

The advantages of having two types of preconditions, normal preconditions and phantom preconditions (filter conditions), are not clear in all situations, especially, if the planner searches for a solution autonomously. There are situations in which we can observe that blocked phantoms decrease the planning performance. The performance also can depend on the definition of the domain. Whether phantoms are useful or not should be analyzed very carefully. On the other hand, declaring phantom preconditions has positive effects for interactive planning because it reduces the set of alternatives to be presented for a certain goal to the user. This often helps selecting the correct alternative.

Backjumping is only a first attempt for more sophisticated backtracking strategies in the context of SNLP planning. The implemented version only can analyze real inconsistencies of operators in a conflict set. Operators of a conflict set that are rejected because of blind backtracking do not contribute to finding the real reason for a failure. But here we might need a propagation mechanism similiar to the one described in (Kambhampati et al., 1995a) to explain the failure of such operators that are rejected because the backtracking.

Until now, autonomous problem solving, especially so-called precondition achievement planning, lacks efficient general purpose mechanisms and heuristics to control the search process. (Drummond, 1993) argues that the success of the big three planning systems, NONLIN (Tate, 1977), SIPE (Wilkins, 1984; Wilkins, 1990), and O-Plan (Currie and Tate, 1991), is maily attributed to the hierarchical action expansion, explicit languages for documenting a plan's causal structure, and to a very simple form of propositional resource allocation. Coding a domain in terms of plan fragments and expected causal structures in fact gives the planner more strategic knowledge that the STRIPS-like operator definitions of precondition achievement planners. In this context, (Young et al., 1994) presented an approach that allows complex actions to be processed by an extension of the SNLP planner. This approach will also be studied more detailled in CAPlan.

Contents

References

Barrett, A. and Weld, D. (1994). Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71–112.

Chapman, D. (1987). Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377.

Collins, G. and Pryor, L. (1992). Achieving the functionality of filter conditions in a partial order planner. In *Proceedings of AAAI-92*, pages 375–380.

Currie, K. and Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86.

Daniel, L. (1984). Planning and operations research. In *Artificial Intelligence: Tools, Techniques and Applications*, pages 423–452. Harper & Row.

Doyle, J. (1979). A truth maintenance system. *Artificial Intelligence*, 12(3):231–272.

Drummond, M. (1993). On precondition achievement and the computational economics of automatic planning. In *Proceedings of the 2nd European Workshop on Planning (EWSP-93)*.

Erol, K., Hendler, J., and Nau, D. (1994a). Htn planning: Complexity and expressivity. In *Proceedings of AAAI-94*, pages 1123–1128.

Erol, K., Hendler, J., and Nau, D. (1994b). Umcp: A sound and complete procedure for hierarchical task-network planning. In *Proceedings of the 2nd International Conference on AI Planning Systems (AIPS-94)*, pages 249–254.

Fikes, R. and Nilsson, N. (1971). Strips: A new approach to the application of theorem proving in problem solving. *Artificial Intelligence*, 2:189–208.

Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46.

Harvey, W., Ginsberg, M., and Smith, D. (1993). Deferring conflict resolution retains systematicity. In *Proceedings of AAAI-93*.

Joslin, D. and Pollack, M. (1994). Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proceedings of AAAI-94*, pages 1004–1009.

Kambhampati, S. (1993). On the utility of systematicity: Understanding tradeoffs between redundancy and commitment in partial-order planning. In *Proceedings of IJCAI-93*, pages 116–125.

Kambhampati, S. (1995). A comparative analysis of partial order planning and task reduction planning. *SIGART Bulletin*, 6(1).

Kambhampati, S. and Hendler, J. (1992). A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55:193–258.

Kambhampati, S., Katukam, S., and Qu, Y. (1995a). Failure driven dynamic search control for partial order planners: An explanation-based approach. Technical Report ASU-CSE-TR-95-010, Dept. of Computer Science and Engineering, Arizona State University.

Kambhampati, S., Knoblock, C., and Yang, Q. (1995b). Planning as refinement search: A unified framework for evaluating design tradeoffs in partial order planning. Technical Report ASU-CSE-TR-94-002, Dept. of Computer Science and Engineering, Arizona State University.

Kettnaker, V. (1995). Konzeption und Realisierung einer Toolbox statischer Kontrollmethoden zur Steuerung eines Causal Link Planers. Diplomarbeit, Universität Kaiserslautern.

McAllester, D. and Rosenblitt, D. (1991). Systematic nonlinear planning. In *Proceedings of AAAI-91*, pages 634–639.

Minton, S., Bresina, J., and Drummond, M. (1991). Commitment strategies in planning: a comparative analysis. In *Proceedings of IJCAI-91*, pages 259–265.

Minton, S., Bresina, J., and Drummond, M. (1994). Total-order and partial-order planning: A comparative analysis. *Journal of Artificial Intelligence Research*, 2:227–262.

Muñoz-Avila, H. and Hüllen, J. (1995). Retrieving relevant cases by using goal dependencies. In *Proceedings of the 1st International Conference on Case-Based Reasoning (ICCBR-95)*.

Muñoz-Avila, H., Paulokat, J., and Wess, S. (1995). Controlling non-linear hierarchical planning by case replay. In Keane, M., Halton, J., and Manago, M., editors, *Advances in Case-Based Reasoning. Selected Papers of the 2nd European Workshop (EWCBR-94)*, number 984 in Lecture Notes in Artificial Intelligence. Springer.

Paulokat, J., Praeger, R., and Wess, S. (1992). CAbPlan - fallbasierte Arbeitsplanung. In Messer, T. and Winkelhofer, A., editors, *Beiträge zum 6. Workshop 'Planen und Konfigurieren' (PuK-92)*. FORWISS.

Paulokat, J. and Wess, S. (1993). Fallauswahl und fallbasierte Steuerung bei der nichtlinearen hierarchischen Planung. In Horz, A., editor, *Beiträge zum 7. Workshop 'Planen und Konfigurieren' (PuK-93)*. GMD.

Paulokat, J. and Wess, S. (1994). Planning for machining workpieces with a partial-order nonlinear planner. In Gil, Y. and Veloso, M., editors, *AAAI-Working Notes 'Planning and Learning: On To Real Applications'*, New Orleans.

Peot, M. and Smith, D. (1993). Threat-removal strategies for partial-order planning. In *Proceedings of AAAI-93*, pages 492–499.

Petrie, C. (1991a). Context maintenance. In *Proceedings of AAAI-91*, pages 288–295.

Petrie, C. (1991b). *Planning and Replanning with Reason Maintenance*. PhD thesis, University of Texas at Austin, CS Dept.

Petrie, C. (1992). Constrained decision revision. In *Proceedings of AAAI-92*, pages 393–400.

Tate, A. (1977). Generating project networks. In *Proceedings of IJCAI-77*, pages 888–893.

Tate, A., Drabble, B., and Dalton, J. (1994). The use of condition types to restrict search in an ai planner. In *Proceedings of AAAI-94*, pages 1129–1134.

Veloso, M. (1994). *Planning and learning by analogical reasoning*. Number 886 in Lecture Notes in Artificial Intelligence. Springer Verlag.

Veloso, M. and Blythe, J. (1994). Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the 2nd International Conference on AI Planning Systems (AIPS-94)*, pages 13–19.

Weberskirch, F. (1994). Realisierung eines nichtlinearen Planungssystems zur Unterstützung der Arbeitsplanerstellung bei der computerintegrierten Fertigung (CIM). Diplomarbeit, Universität Kaiserslautern.

Weberskirch, F. and Paulokat, J. (1995). CAPlan - ein SNLP-basierter Planungsassistent. In Biundo, S. and Tank, W., editors, *Beiträge zum 9. Workshop 'Planen und Konfigurieren' (PuK-95)*. DFKI.

Weld, D. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.

Wilkins, D. (1984). Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301.

Wilkins, D. (1988). *Practical Planning - Extending the classical AI Planning Paradigm*. Morgan Kaufmann.

Wilkins, D. (1990). Can ai planners solve practical problems? *Computational Intelligence*, 6:232–246.

Yang, Q. (1992). A theory of conflict resolution in planning. *Artificial Intelligence*, 58:361–392.

Young, R., Pollack, M., and Moore, J. (1994). Decomposition and causality in partial-order planning. In *Proceedings of the 2nd International Conference on AI Planning Systems (AIPS-94)*, pages 188–193.