TEMPORAL PLANNING AND PLAN EXECUTION IN A DYNAMIC ENVIRONMENT

Jeroen Swanborn

CIS-TR-94-16
June 10, 1994

Computer and Information Science Department
Univeristy of Oregon

TEMPORAL PLANNING AND PLAN EXECUTION IN A DYNAMIC ENVIRONMENT

Jeroen Swanborn

ABSTRACT

Setting goals and determining their associated plans of action have long been considered fundamental aspects of intelligent behaviour. Much of the work in the artificial intelligence research area of planning has made simplifying assumptions and thereby neglected important real-world dimensions. It is the intent of this thesis to relax two of these assumptions and discuss their implications on the planning process. We explore the modeling of plans of action in a *temporal setting* and consider the execution of those plans in a *dynamic environment.*

Temporal consistency criteria for plans are set forth and their associated algorithms are discussed in terms of their time complexity. We present a polynomial-time algorithm for determining the temporal consistency of partial-order plans. For dynamic plan execution, we present an algorithm which leads to a minimal amount of replanning. Both algorithms are planner-independent and have been implemented to enhance a classical, partial-order planner.

ACKNOWLEDGEMENTS

The work for this thesis was a collaborative effort. For his input, guidance, and patience I express sincere appreciation to Professor Art Farley. I also thank Kathy Stevens for her contribution to the early research encompassed in this thesis and her diligent review of the manuscript. Above all, I thank my wife, Sara, for her love and understanding throughout the course of this endeavor.

TABLE OF CONTENTS

TABLE OF FIGURES

Whatever your hands find to do, do it with all your might, for in the grave, where you are going, there is neither working nor planning nor knowledge nor wisdom.

Ecclesiastes 9:10

CHAPTER I

INTRODUCTION

Setting goals and determining their associated plans of action have long been considered fundamental aspects of intelligent behaviour. As humans, we spend a great deal of time deciding and reasoning about actions, some with much deliberation and some without much forethought. The numerous desires that we wish fulfilled are sometimes in conflict with each other. They may require us to choose among them and to reason about how best to accomplish those that we choose. Wise decisions about our course of action towards realizing desired ends will depend upon currently held beliefs about present and future situations and upon any commitment or intentions that we may have decided upon earlier.

Often it will be necessary to obtain more information about the tasks to be performed, either prior to choosing a plan of action or during its execution. Furthermore, our knowledge of the world itself is frequently incomplete, making it necessary for us to have some means of forming reasonable assumptions about the possible occurrence of other events or the behaviour of other agents. As time passes and more information about the current state of the world becomes apparent to us we may need to adjust our plan to accommodate unexpected events due to incorrect assumptions.

All this has to be accomplished in a complex and dynamic world populated with many other agents. The agent planning or deciding upon possible courses of action can choose from an enormous repertoire of actions, and these in turn can influence the world in exceedingly complicated ways. Moreover, because of the presence of other agents and processes, the environment is subject to continuous change -- even as the planner deliberates on how best to achieve its goals.

Consequently, recent debate has called into question the extent to which human beings actually utilize plans of action. Even though humans lay out an approximate plan of action, it has been observed that their actions are more strongly dictated by the immediate surroundings or situation. These situated actions are much more adaptable to the complex environments in which humans operate. [19] It is not our intention to discuss the merits of either planning or situated action; it is clear, however, that in order to model intelligent behaviour we must incorporate both approaches.

In this thesis, we explore the modeling of plans of action and their execution in a dynamic environment. Much of the planning research up to this point has neglected important real-world dimensions and therefore has tended to simplify the planning process. Unfortunately, these simplifications have impaired the application of the results with respect to solving real-world problems. The work that is presented in this thesis attempts to deal with two of these simplifications and explores the implications of removing those limitations in a planning system.

The first addition to traditional planners is the consideration of temporal issues in the planning process. In traditional planners actions are instantaneous and goals can be satisfied in any particular order without regard to time. The introduction of time, in the form of goal deadlines and action durations, adds new conditions that must be satisfied by the plan and places additional constraints on the planning process. All actions in the plan will have durations, and goals will also specify when certain conditions must hold true.

Another limitation is the assumption that an agent is the sole cause of change in the world. We remove this limitation by placing the planning system in a dynamic environment where the individual actions of the agent are carried out. Once a course of action has been constructed, the execution of the particular steps in the plan interacts with an environment in which other agents also act to give possibly unexpected or unplanned results. These results render the original plan potentially incorrect and may require a different plan to achieve the desired goals.

Both of these extensions to traditional planners are considered in a domain- and planner-independent approach. The algorithms we describe are equally applicable to all partial-order planners. To accomplish this independence, our algorithms only preprocess the input to a planner and manipulate the output of the planner. Theoretically, no modifications to the planner need to be made to accommodate our notions of temporal and dynamic planning. This approach has the advantage of allowing us to utilize any advances in the field of planning.

With respect to planning in a dynamic environment, planner-independence implies that we take the complete plan returned by the planner and model the world as time progresses and individual actions of the plan are performed in the world. The model allows for outside agents to affect the state of the world and potentially disrupt the assumptions made by the planner. In addition, the agent carrying out the plan may be given additional goals while executing its current plan of action. The current plan of action may no longer be correct with respect to new conditions of the world or new goals, yet portions of the plan may still be usable. The thrust of our work in this area is the provision of an algorithm that determines the portions of the plan that are not affected by the unexpected changes. We assume the planner is a generative and give the remaining portion of the plan to the planner to complete. Refinement of the plan in this way allows for a minimal amount of replanning.

The interleaving of planning, executing, and specifying goals is not analogous to producing a plan with a conjunction of all the goals. In traditional planning, if two conflicting goals are specified, the planner will not be able to produce a complete plan. However, the interleaving process and the temporal nature of the plan makes it possible for potentially conflicting goals to be satisfied at different times.

To provide for planner-independence in the case of temporal planning, an external temporal database is maintained. Temporal constraints are posted to the database while preprocessing the goal specifications; additional constraints are posted once the planning process is completed. The latter constraints are partial order constraints on the actions and can easily be extracted from the complete plan. The temporal database is responsible for determining if the plan meets all of the temporal constraints. If so, the plan is considered time consistent and can be passed on for execution.

The algorithms described in this thesis have been implemented in Common Lisp and have been tested with the systematic non-linear planner UCPOP. [18] UCPOP was developed at the University of Washington and provided an excellent environment for applying temporal and dynamic planning. In the following chapters, we relate our work directly to UCPOP, yet it is not dependent on that particular planner. Figure 1.1 shows the architecture of our system and reveals the relationship of the modules to the core planner, UCPOP.

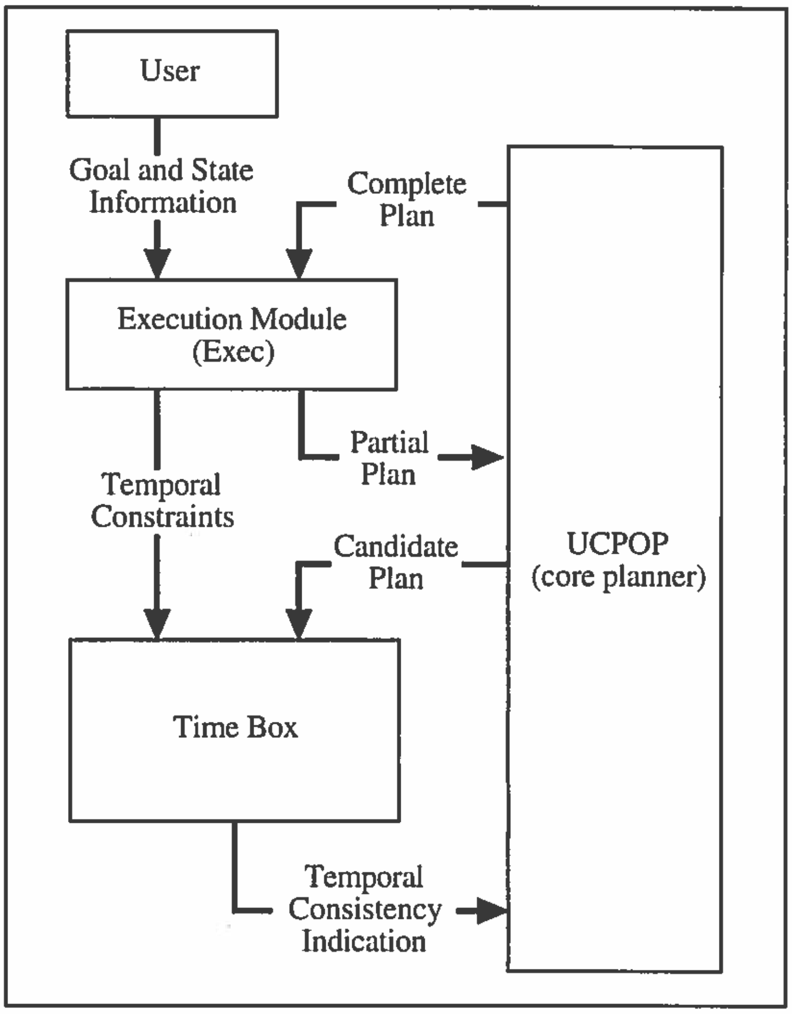

Figure 1.1 Software Architecture of Our Temporal Planner with Dynamic Execution.

The next chapter will briefly discuss some history and background of planning, leading to the development of non-linear planning. The chapter concludes with a general overview of the UCPOP planner and introduces some terminology that we will carry forward throughout the remainder of the thesis. Chapter III describes the kind of extensions that must be made to the UCPOP formalism to account for the inclusion of temporal issues. In particular, the domain and goal specification languages must be enhanced to accommodate action durations and goal time constraints, respectively. In that chapter, we also consider how to parse and interpret the input to a planning problem and how to preprocess the input into an acceptable form for UCPOP. A discussion of the representation of time and the representation of the temporal constraints concludes the chapter.

Temporal consistency criteria are the main focus of the next two chapters. We relate some theoretical results, in chapter IV, concerning the time complexity of determining temporal consistency. Based on these observations, an algorithm is presented in chapter V and shown to be an adequate determiner of the temporal consistency criteria. Chapter VI is concerned with the execution of a plan in a dynamic environment. We describe the mechanisms for changing the state of the world and introducing new goals during the execution of a plan. Algorithms for adjusting the current plan to take into account the new world state are described. Furthermore, the issue of how to choose an action in the plan to execute is also discussed. The last chapter concludes this thesis with a summary and discussion of future directions.

CHAPTER II

LITERATURE BACKGROUND

Planning has been an integral research area within artificial intelligence for several decades. In this chapter, we provide a short historical overview of research in planning, paying particular attention to partial-order and temporal planning. First, we provide a formal definition of the planning problem.

A planning problem has three inputs: a description of the initial state of the world, a description of the desired (goal) state of the world, and a description of the possible actions that the agent can perform (domain theory). All three inputs are specified by formal languages, which may vary in expressiveness. The planner's output is a sequence of actions which, when applied to the initial state, will achieve the goal state.

This is a very general statement of the problem and all planners have to make simplifying assumptions regarding the world and the actions they are attempting to model. As we mentioned in chapter I, it is our intent to relax two of these assumptions and discuss their implications on the planning algorithms. First, however, we briefly discuss the progression of planning research.

Short Introduction to Planning

STRIPS

Probably the single most influential work in planning was that of Fikes and Nilsson in the early 1970s. [8] Their STRIPS planner was implemented initially to control a robot at Stanford Research Institute (SRI). STRIPS is in the class of problem solvers that searches a space of world models or world states to find one in which a certain goal is achieved. A set of operators, which represent the possible actions of the agent (robot, in this case), each of which transforms a world state into another world state. The object of the STRIPS planner is to find some composition of operators that transforms a given initial world state into one that satisfies some stated goal condition. A world state is modeled by a finite set of logical propositions whose conjunction reflects the actual state of the world.

Each operator is defined by an operator description consisting of two main parts: a description of the effects of the operator and a description of conditions under which the operator is applicable. The applicability conditions are referred to as the preconditions of the operator. The effects of the operator are broken into two sets: those propositions that are added to the world state and those that are deleted from the world state, as a result of executing the operator when its preconditions are met.

An operator, then, consists of three lists of logical propositions. The *precondition list* contains those propositions that must be true in the world state before the operator can be successfully employed by the agent. The propositions that become true as a result of executing the operation are in the *add list* and the propositions that become false are contained in the *delete list*. An example of a STRIPS-like operator that allows the block labeled A to be moved from the table top onto the block labeled B is shown in figure 2.1.

```
(operator move A-from-table-to-B
    :precondition-list ((clear A) (clear B))
    :add-list ((on A B))
    :delete-list ((clear B)))
```

Figure 2.1 Example of a STRIPS Operator.

The STRIPS planner proceeds by placing all of the goal propositions on a stack. It performs a means-ends analysis process, adding an operator to the plan that will satisfy the goal on top of the stack. The preconditions of the goal which are not met in the current world state are pushed on the stack and planning continues. The search space can be represented as a tree where each edge is the application of an operator and each node is a state of the world. The solution to a planning problem is not the goal node, but the path from the node representing the initial world state to the goal node.

Inherent in STRIPS planning is the linearity assumption. The construction of a linear plan assumes that the sequential completion of goals and all the subgoals necessary to realize each goal will satisfy the conjunction of those goals. That is, goals are believed to be independent in that they do not adversely interact. This assumption is not wholly reasonable, since it is easy to envision a set of goals where the execution of the actions that lead to one goal may remove those propositions in the world state that satisfied an earlier goal.

It has been apparent since the earliest days of planning that strictly linear planning is inadequate. The "Sussman Anomaly" problem in the blocks world domain is an example which a linear planner like STRIPS, cannot solve. The notion of a partially ordered plan was introduced to allow interleaving of goals and subgoals, thereby permitting a larger set of problems to be solved.

SNLP

McAllester [15] describes a sound, complete, and systematic non-linear planner (SNLP) based on the STRIPS representation. Unlike STRIPS which explores a search space of states and operators, SNLP explores the search space of plans. Each node in the search space is a partially specified plan. Edges denote plan refinement operations. These refinements may take the form of the addition of an action to the plan, the insertion of a constraint on temporal ordering, or the addition of variable bindings. The solution to a SNLP planning problem is a node in the plan space that contains a consistent and complete plan, to be defined below. Searching in plan space is advantageous, because we can circumvent the linearity assumption and represent plans as partially ordered sequences of actions.

A partially ordered plan does not have a strict order imposed on the sequence of its operators. Partial order planning is a form of least commitment planning in the sense that when operators are chosen for inclusion in the plan no explicit commitment is made to when that operator must be executed in relation to all of the other operators. The longer such ordering commitment can be delayed, the more general the plan is and the more flexible the planning process. However, most operators require that certain propositions be established before their execution can take place. Those propositions are represented by a *prerequisite set*. Thus, each proposition in the prerequisite set of an operator must have a source, or several sources, that establishes it. The connection between the source operator and the consuming operator is referred to by McAllester as a *causal link*. The causal link of proposition *P* implies the source step has *P* in its add list and the consumer step has *P* in its prerequisite list. The causal link also implies an implicit ordering of the source and consumer operators. Notice, however, the source and consumer do not necessarily have to be consecutive steps in the plan.

Since a causal link indicates a dependency among steps in a plan, care must be taken to protect *P* from being disestablished by an intervening step. Since partial order plans do not specify the order of the steps in the plan, any step that contains *P* in its delete list is a potential *threat* to the causal link.[1] A *safety condition* is introduced to protect the causal link from the threatening step. The safety condition is an explicit temporal ordering of steps to ensure that the threat either precedes the source step of the causal link or the consumer step precedes the threat.

Considering these concepts, McAllester goes on to define a plan as sets of steps, causal links, and safety conditions. The set of steps contains two special steps: *start* and *finish*. The start step has empty prerequisite and delete lists, but all the propositions of the initial state of the problem are contained in the add list. The add and delete lists of the finish step contain no propositions, but its prerequisite list includes all of the goal propositions. A plan is said to be *complete* if all prerequisites of all steps have a causal link and all threats have an associated safety condition. The procedure to find a complete plan that constitutes a solution is broken into four separate steps.

The planning procedure takes as input a sparsely defined, partially ordered plan. The initial plan consists of only two steps, *start* and *finish*, with no causal links or safety conditions. The recursive procedure first determines if the plan is order inconsistent. If it is, then the procedure returns failure. At this point, the plan is a solution if it is complete. If it is not, however, the procedure attempts to resolve any threat to a causal link by adding a safety condition to the plan. The procedure is called recursively with the new plan. The final step is to add a causal link for a prerequisite that does not already have a causal link. The source of the causal link can either be an existing step that adds the proposition or a new step with an operator that adds the proposition. Again, the procedure is called recursively with the new plan.

A complete plan is not necessarily a solution; only if the complete plan is not order inconsistent is it a correct solution. McAllester defines *order inconsistency* in terms of a *topological sort* on the plan. A topological sort is merely a linearization or total ordering of the partial order plan. The partial order relations of the casual links and the safety conditions constrain the full ordering of the steps. Thus a topological sort is any sequence of the steps that does not violate the implied orderings of the causal links or the explicit orderings of the safety conditions. By definition, the *start* step must be the first step in the sequence and the *finish* step must be the last. If a plan has no topological sort, it is considered order inconsistent. McAllester points out that if and only if the partial ordering constraints from the causal links and the safety conditions of the plan define a cycle in the plan steps, then the nonlinear plan is order inconsistent. Therefore, the $O(n^3)$ transitive closure algorithm can be used to determine order inconsistency.

[1] McAllister actually defines a more restricted form of threat to ensure systematicity in his planner. In his definition a threat can neither delete nor add the proposition *P*.

UCPOP

Realizing some of the limitation of the STRIPS representation used by SNLP, Penberthy and Weld describe a partial order planner for a subset of Pednault's Action Description Language (ADL). [18] Their UCPOP planner is the basis for our discussion in subsequent chapters as well as the basis for our concrete exploration of temporal issues and dynamic plan execution. We will introduce the basic components of the UCPOP planner and its associated terminology here and conclude the chapter with a closer look at its algorithms after completing the survey of planning.

UCPOP modifies McAllester's representation of plans used in SNLP. The notion of add and delete sets have been merged into one set of *effects*. When an operator is executed, the propositions in its effect set are merged with the state. This implies, if the proposition is negated, the proposition is deleted from the state. Otherwise, the proposition is simply added to the state. UCPOP allows for conditional effects in the operator specification. That is, an effect will only be applied to the state (added or deleted) when its condition holds. Furthermore, UCPOP allows for both existential and universal quantification in the *preconditions* (McAllester's prerequisites) of operators and effects, as well as the post-conditions of the effects, and goals. An example of the extended domain specification language is given in figure 2.2. The operator moves a briefcase from a location, represented by the variable ?l, to another location, represented by the variable ?m. As a result, all of the objects that are in the briefcase also change location.

```
(operator move-briefcase
    :parameters     (?b ?l ?m)
    :precond        (and (briefcase ?b) (at ?b ?l) (≠ ?l ?m))
    :effect         (and (at ?b ?m) (not (at ?b ?l))
                    (forall ((object ?x))
                       (when (in ?x ?b)
                          (and (at ?x ?m) (not (at ?x ?l)))))))
```

Figure 2.2 UCPOP Operator with Conditional Effects and Universal Quantification.

Like SNLP, two steps are in the initial plan. The initial state is captured in the effects of the *start* step and the goal is encoded in the preconditions of the *goal* (or *finish*) step. All of the preconditions of the goal step are placed in a set of *open conditions*, those preconditions that do not have an associated causal link. UCPOP then attempts to complete the initial plan by adding new steps and/or causal links which satisfy the open conditions. The preconditions of the new steps are added to the set of open conditions. Since UCPOP is a partial-order planner, it needs to be aware of threats which may clobber established causal links. If a threat to a causal link occurs in the plan, it is resolved by any of three methods: demotion or promotion of the threat or separation.

Demotion and promotion are the addition of constraints to the plan, which insure that the threat is either ordered before the source of the causal link or ordered after the consumer of the causal link, respectively. One form of separation adds binding constraints to the plan so that the effects of the threat cannot unify with the negation of the proposition which is the subject of the causal link. This means that if the threat is eventually ordered between the two steps of the causal link it cannot clobber the precondition established by the causal link. A second form of separation chooses a precondition of the threat and creates an open condition with the negation of the chosen precondition. This form of separation is also called confrontation.

Since operators and goal specifications allow for variables in their propositions, UCPOP must be able to deal with variables efficiently. Bindings of variables to values and binding constraints are maintained in a *var-set*. Variables that are mutually constrained may either *codesignate* or *non-codesignate*. In the case of the former, the two variables must have the same value, namely the constant associated with the var-set. Non-codesignation means the two variables may not have the same value. If any of the variables of the codesignation set gets bound in the process of unification, the value is added to the var-set and implicitly the other variables in the codesignation set are also bound. Only one such value may exist for any particular var-set.

A UCPOP plan, then, is a five-tuple: steps, binding constraints (var-sets), ordering constraints, causal links, and open conditions (also referred to as sub-goals). The initial planning problem is given to UCPOP in terms of this five-tuple. The initial plan has two steps, the *start* step, S_0, and *goal* step, S_∞, as described above, one ordering constraint, $\{S_0 < S_\infty\}$, a set of open conditions matching the preconditions of the *goal* step, and the sets of causal links and binding constraints are empty. UCPOP refines this initial plan until it is complete or no more refinements are possible. A plan is considered complete when the set of open conditions is empty and the sets of binding and ordering constraints are not inconsistent.

Unlike SNLP, UCPOP does not enforce systematicity. A planner is *systematic* in its search technique if it does not explore the same plan more than once. McAllester and Penberthy provide rigorous proofs that their respective planners are both sound (any solution returned is a correct solution) and complete (if there is a solution, the planner will find it given enough time). Neither, however, addresses the issues of time beyond the notion of partially ordering the steps. Goals cannot be given time constraints and operators do not contain the concept of deadlines or durations.

Temporal Planning

The exploration of time in the context of planning systems has been a topic of particular interest in artificial intelligence planning research. However, temporal issues in planning have been generating more interest in the last several years. Much of the work, up until recently, has concentrated on the formal framework of representing time and reasoning about time. We will consider some of the formal framework and discuss several noteworthy implementations of time-based planners.

Temporal Logic

McDermott proposed a first-order temporal logic upon which programs involving time could be based. [16] By specifying this formal framework, McDermott attempts to capture some concepts of time not adequately addressed before. In particular, the formalism attempts to include the concepts of continuous change and the indeterminacy of the future. While addressing the latter, McDermott sheds some light on the frame problem.

A current situation in the world is encapsulated in a *state*. Every state has a time occurrence, known as its *date*. States are partially ordered on the relation ≤ (e.g. (≤ state1 state2) is used to denote that one state occurs before or is identical to a second state). A totally ordered set of these states, extended infinitely in time, is a *chronicle* denoting a complete possible history of the universe. At any particular point in time there may be more than one possible future, based on an actor's decisions or random events. Therefore, a chronicle may branch in the future. That is, one chronicle is one of many possible total orderings of the states in the universe.

States and chronicles are necessary building blocks for the encapsulation of *facts* and *events* in this logic. Since a fact changes its truth value over time, a fact is defined to be the set of states in which the "statement" is true. A fact that is never true constitutes the empty set and the fact that is always true is the set of all possible states. An event is when something is happening and is characterized by a set of time intervals. An interval is a totally ordered set of states, much like a chronicle but not extending indefinitely in time. The event is said to be taking place in each of the states associated with the interval. A fact is true during an interval if and only if the interval is a subset of the fact.

To capture the notion that previously established facts remain true unless explicitly contradicted, McDermott introduces the idea of *persistence*. That is, a fact persists if in all chronicles it remains true until its lifetime has expired or until the fact ceases to be true (i.e. the fact is negated by another rule). Persistence is important in overcoming the frame problem. Using the persistence notion, one can reason about facts that were previously established, even though information about the world is incomplete.

Ensuring that certain facts remain established is important if some latter action depends on that fact. Thus certain facts must be *protected* from becoming false. McDermott distinguishes between *unrestorable* and *restorable* protection intervals. That is, certain facts must remain true for an entire interval of time, while it suffices for other facts to be true only at certain points of the interval. Even though McDermott did not formalize this, the idea of restorable and unrestorable protections seems important in planning.

DEVISER

Time intervals were also explored by Vere in his planner called DEVISER. [21] This parallel, or non-linear, planner incorporates durations and time intervals for activities and goals. An activity is the general term describing either an action (change in the world determined by the plan actor), event (change triggered by the state of the world), or inference (assertions based on the truth of preconditions). All activities are represented as productions which resemble the representation of STRIPS rules with preconditions, add sets and delete sets. The unique aspect of DEVISER is the association of durations to actions, events, inferences, and goals.

In addition to specifying durations for goals, the time when a goal is to be satisfied may also be included in the goal description. A goal can be assigned a window of time when the conditions of the goal must be met. The window specifies both the earliest time and the latest time the conditions must be achieved in order for the goal to be satisfied. The time windows can also be used to represent the concepts of before, after, or at an absolute time. The duration specifies how long the conditions of the goal must continue to hold.

Activities are represented with a context, antecedents, and consequents. Each is a set of propositions which correspond closely to the precondition, delete, and add lists of the STRIPS representation. Actions in DEVISER are identical semantically to operators, except a duration for the action may be specified. An event specifies actions that occur spontaneously and is triggered by a particular state of the world. Events differ from actions, because the actor in the plan does not control their occurrence. Scheduled events are those events that occur at a constant time and do not depend on the state of the world.

The interesting aspects of DEVISER concern the implementation and implications of time windows. Since activities occur over a period of time, one question it must address is: when do the changes to the world state take place? For DEVISER, the changes occur at the termination of the activity. Both proposition additions and deletions occur when the activity has been in progress for the specified duration. For those activities where some of the changes are to take place sometime during the activity, Vere provides a solution. Suppose the several changes are to occur after 10 seconds have elapsed since the start of the activity. The activity can be broken into two separate activities. The two activities are marked as CONSECUTIVE and the first activity contains all of the preconditions and specifies those propositions that are added or deleted. The duration of the first activity is set to 10 seconds. The second activity contains the propositions whose effects are to take place at the end of the original activity.

Windows are implemented as a triple containing the earliest start time (EST), the ideal start time, and the latest start time (LST). Windows may be specified explicitly for goals, but are calculated dynamically for activities. When a new activity is introduced in the plan it initially has an infinite time window (0 nil infinity). As planning progresses, causal links are added and conflicts (threats) are detected. The ordering constraints and the activity durations imply that the time windows of activities need to be adjusted. When resolving conflicts, there are several choices with respect to compressing time windows. The LST of one activity could be moved up (earlier in time) or the EST of another could be delayed. Each choice represents a node in the planning search space. Note that windows are never expanded. A window must always be consistent with respect to the principles of time. That is, the inequality $EST \leq LST$ must always hold. Furthermore, when a causal link exists between two activities, $EST_1 + duration_1 \leq LST_2$. If either inequality does not hold, the plan fails and DEVISER must backtrack. Notice that the compression of a time window for one step may have implication for the windows of other steps in the plan. Those affected windows must be compressed before planning can continue.

Temporal Constraint Satisfaction Problem

Another approach to representing time is explored by R. Dechter, et al. [7] They map the temporal information of a planning problem into a *temporal constraint satisfaction problem* (TCSP). The variables in this constraint satisfaction problem are the time points of steps or events. The continuous variables are constrained by time intervals, which are represented by a set of unary and binary constraints. This work is significant because they describe an algorithm of time complexity $O(n^3)$, which solves a simplified TCSP.

A *simple temporal problem (STP)* is a specific instance of a TCSP, where each constraint on a time point specifies only one interval. This is quite adequate for the time representation of DEVISER. For example, the variables of the STP can represent the starting time of an activity and the constraints specify the interval from the EST to the LST. Constraints can either specify a time point's relation to absolute time (such as 9:30 am) or specify a relationship between two time points. If activity A is to start between 60 and 75 minutes from now (t_0) and activity B, with a 10 minute duration, is to finish at least 20, but no more than 40, minutes earlier, the resulting constraint consist of the following four inequalities: $60 \leq A - t_0 \leq 75$ and $30 \leq A - B \leq 50$. The STP is associated with a directed graph, called a distance graph, shown in figure 2.3. The vertices of the graph are the time points and the edges represent temporal relationships between time points. Each edge is labeled with a weight representing the temporal distance between the two vertices.

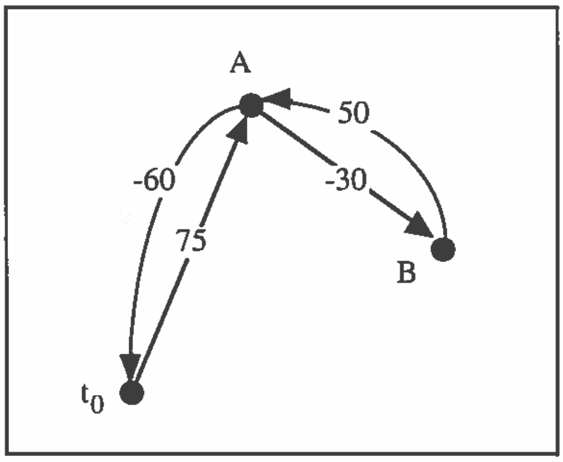

Figure 2.3 Distance Graph of a Simple Temporal Problem (STP).

The distance graph of an STP can be transformed into a complete *d*-graph, where edges are labeled by the shortest path length, by applying Floyd-Warshall's all-pairs-shortest-path algorithm. The algorithm detects inconsistencies in the distance graph and generates the *d*-graph in $O(n^3)$ time. Dechter shows that finding a solution to the STP once the *d*-graph is generated requires only $O(n^2)$ time. This work is utilized in our implementation of the temporal consistency criteria and we return to STPs in more detail in chapter V.

CHAPTER III

EXTENDING THE UCPOP FORMALISM

If we wish to reason about plans with respect to a particular time frame, it is important, nay essential, that the notion of durations for actions is introduced into the UCPOP formalism. Without such a notion, actions are considered to be performed instantaneously and consequently an infinite number of actions can take place in any arbitrary interval of time. Although this would make temporal planning no more difficult than non-temporal planning, it is certainly neither interesting nor realistic. Consider the simple example of driving to the airport to catch an early morning flight. Suddenly you realize you have forgotten your toothbrush. There is no threat of missing your flight if driving home, finding your toothbrush, and driving back to the airport can be done instantaneously. The question that is of interest to you is not only "Can I make it home, find my toothbrush, and drive to airport again?", but more importantly "Can I do all of those activities and still make it to the airport in time to catch the red-eye?"

Augmenting UCPOP with Time Lengths for Operators

To capture the amount of time an action takes to perform, the domain theory of UCPOP must be extended. In addition to specifying preconditions and effects, the length of time required to successfully execute an action must also be included in an operator specification. Figure 3.1 shows the `puton` operator from the blocks world domain, augmented with a duration. We have made the simplifying assumption that all actions that move a block from one location to another take the same amount of time, namely 15 time units. The use of variables in the operator specification means that each operator schemata is capable of encoding only one duration for all the operators it captures. Thus, if it is critical to distinguish the duration of the `(puton A table B)` operator from that of the `(puton A B table)` operator, they must appear in different operator schemata.

```
(operator puton
    :parameters      (?x ?from ?to)
    :precond         (and (on ?x ?from) (clear ?from) (clear ?to))
    :duration        15
    :effect          (and (on ?x ?to) (not (clear ?to))
                          (not (on ?x ?from)) (clear ?from)))
```

Figure 3.1 UCPOP Operator with Duration.

This change does not require a modification in the UCPOP planner. As a matter of fact, the domain specification can be preprocessed and the operation duration information can be stored in the temporal database. Since the planner does not take into account the duration of operators, the temporal information is only utilized during the temporal constraint verification process after regular planning has finished. A more detailed discussion of this process and how the duration of an operator comes into play is forthcoming in chapter IV.

Augmenting UCPOP with Temporal Constraints on Goals

The primary purpose for reasoning about plans in a temporal setting is to place stipulations or constraints on the time certain goals must be met. A plan succeeds only if the goals can be met within the constraints given. Returning to our earlier example, the plan to retrieve our toothbrush would be considered successful only if we both fetched the toothbrush and made it to the airport in time for our flight. To make that determination we must then place temporal limitations on the goals we wish the planner to achieve. To accomplish this the goal specification language must be enhanced to capture temporal constraints.

We have decided to incorporate Allen's temporal relationships and their conjunctions. [2] This implies that a goal may be temporally related to one or more other goals. These relationships, however, are not sufficient to fix a goal in absolute time, thus we have augmented these temporal relationships with an additional five constraints that may be applied to a goal. These additional constraints do not relate a goal to another goal, rather, they place the goal on a specific location on the time continuum.

Let us take a closer look at these temporal constraints, and how they are added to the goal specification language. By associating names or labels with particular goal specifications, subsequent goals may refer to them by name. Thus, in our example of retrieving the toothbrush we explicitly state that we must be at the airport in one hour and forty-five minutes and label that goal *airport*. To ensure that we have our toothbrush before we take flight we add a second goal which specifies that it must be accomplished before goal *airport*. Figure 3.1 shows the portion of the goal specification language that accomplishes this task.

```
(:name airport          :time (:at 105) :goal (at terminal))
(:name get-brush        :time (:before airport)
                        :goal (in toothbrush bag))
```

Figure 3.2 Goal List Specification with Goal Name Reference.

The complete revised goal language specification is shown in figure 3.3. The processing of the goals is performed in order, thus a reference to another goal is only valid if that goal has already been processed. The goals may otherwise be specified in any order. Furthermore, certain time constraint operators can be extended to include a time range. For example, `:after` can be constrained by the `:range` operator as follows:

```
(:name goal-two :time (:after goal-one :range (10 20)))
```

which implies that *goal-two* must be met somewhere between 10 and 20 time units after *goal-one*. (Note, since the start times of goals are not strictly enforced this really means that *goal-two* may start no later than 20 time units after *goal-one* is met.) The `:range` designator may similarly be used with `:before`.

Conjunction of temporal constraints is also allowed. This means that the time of a goal may be constrained by a relation to more than one goal or by a goal and a fixed point in time. The inclusion of conjunctions, however, makes it significantly easier to specify goals that are explicitly impossible; that is, those whose time constraints are inconsistent regardless of the planning process. The syntax does not expressly disallow specification of such inconsistent constraints, but a simple and efficient check by the temporal database will detect any initial inconsistencies. If temporal inconsistencies are found there is no need to introduce the planning problem to the planner.

```
<goal-spec> ::= ( [:name <goal-name>] :goal <goal>
                  [:duration <number>]
                  [time: <time>] ) [<goal-spec>]
<goal>      ::= ( :and <prop> <prop>*) | <prop>
<prop>      ::= ( <string> <variable>*)
<goal-name> ::= <string>
<time>      ::= ( :and <time-cns> <time-cns>*) | <time-cns>
<time-cns>  ::= ( :before       <goal-name> [(:range <range>)]    |
                  :after        <goal-name> [(:range <range>)]    |
                  :equal        <goal-name>                       |
                  :overlaps     <goal-name>                       |
                  :during       <goal-name>                       |
                  :finishes     <goal-name>                       |
                  :starts       <goal-name>                       |
                  :starts-at    <number>                          |
                  :finish-at    <number>                          |
                  :after-time   <number>                          |
                  :before-time  <number>                          |
                  :at           <number> )
<range>     ::= <number> <number>
<variable>  ::= ?<string> | <string>
<number>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <number>
<string>    ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n |
                o | p | q | r | s | t | u | v | w | x | y | z |<string>
```

Figure 3.3 Goal Specification Language Syntax.

The final extension introduced into the goal specification language is the notion of a minimum duration for a goal. Once a goal is established we may wish for it to remain established for a certain amount of time. Although the planner will not introduce any actions that explicitly negate a goal, it is possible that the portion of a plan required to attain a subsequent goal affects propositions of the earlier goal. Specifying a duration for a goal introduces a temporal constraint on any action that may interfere with the goal.

The time specified for a goal is actually the time at which its propositions must be true, without regard for the time that the goal was initially established. The planner may establish a goal at any time, as long as the goal remains established at the specified interval. For most instances this does not pose a problem. We may stipulate that our tax preparation be done by April 15th, but if we get done earlier no one will complain. However, if we plan to meet someone for an ice cream cone in the park at one o'clock, a very early arrival will result in ice cream soup. We may actually wish to restrict our purchasing of the ice cream to somewhere between 12:45 and 12:50, and use the remaining time to walk to the park. The temporal aspect of our planner does not properly consider those instances exemplified by the latter situation. We discuss reasons for this limitation in the next chapter of the thesis.

Formulating the Input for UCPOP

Operators or actions in a plan are represented by steps which have preconditions and effects. As we discussed in chapter II, the preconditions of a step are those propositions in the world that must be true before the step can be carried out. The action performed by a step is represented by the effect(s) that it has on the state of the world. An entire plan can be represented by a sequence of these steps, and a plan is complete (in the sense that it accomplishes its goals) if and only if the preconditions of all the steps are properly established.

The representation of goals in the planning domain is rather straightforward. Before temporal issues and dynamic execution were introduced into UCPOP, the planner considered the goal to be a special terminal step, the *goal* step, whose preconditions corresponded to the goal and whose effects were empty. Similarly, the initial state was represented by an initial step, the *start* step, with no preconditions and the propositions in the initial state as its effects. These two steps were treated specially, in that no steps could precede the initial step and no steps could follow the goal step (see chapter II). Introducing multiple goals with time constraints into the planning process means the representation of goals needs to a bit more flexible. However, the representation of the initial state suffices.

Unlike actions in a plan, which cannot be interrupted and require the full attention of the acting agent, goals behave more like events: they just happen. For the duration of a goal, the agent may be engaged in other activities as long as those activities do not interfere with the goal. For example, a nurse may be on-call for the local hospital (the goal is being able to hear the phone for an extended period), but he may engage in other activities as long as those activities do not preclude him from hearing the phone.

To accommodate this nuance, goals are represented by two steps: a *goal-start* step and a *goal-finish* step. Both goal steps have preconditions and effects corresponding directly to the goal that is to be achieved. Notice that the effects for the goal steps are essential because a consequence of allowing multiple goals is that plan steps may follow any goal. We associate a duration of zero with each goal step indicating it takes no effort on the part of the agent to realize the goal. Indeed, all the work is performed by those preceding steps that establish the preconditions. The duration of a goal is captured in the temporal database, which we discuss in the subsequent section. A causal link is created between each effect of the start-goal step and the corresponding precondition of the goal-finish step. These causal links ensure that no step can intervene between the start-goal and finish-goal steps, unless the step does not threaten the goal. The causal links also define an implicit ordering of these goal steps so that the *start-goal* step precedes the *finish-goal* step. The ordering is made explicit by adding the appropriate ordering constraint to UCPOP.

Initializing the Temporal Database

At this point we have not yet addressed the processing of the temporal constraints specified with each goal. The constraints have to be considered at two different levels. Since the only notion of time that UCPOP possesses is the partial ordering of steps, time constraints that specify absolute time cannot be given to UCPOP for consideration in the planning process. Other constraints that relate two separate goals with time ranges must first be cast into a relative ordering constraint. Thus, with respect to the planning process of UCPOP, some of the time constraint information is lost. Figure 3.4 shows the transformations from the time constraints given in the goal specification language to the ordering constraints for UCPOP. The second level of consideration is the transformation of the constraints from the goal language to a temporal database that will ultimately be used to perform a temporal verification of a plan returned by the UCPOP algorithm. Yet, before considering the issues of this transformation, we must briefly discuss how we propose to represent time and the constraints thereon.

Temporal constraints for goal X	Partial ordering constraints for UCPOP
`:after Y`	(f-Y < s-X)
`:before Y`	(f-X < s-Y)
`:equal Y`	(s-X < s-Y) (s-Y < s-X) (f-X < f-Y) (f-Y < f-X)
`:overlaps Y`	(s-X < s-Y) (s-Y < f-X) (f-X < f-Y)
`:during Y`	(s-Y < s-X) (f-X < f-Y)
`:starts Y`	(s-X < s-Y) (s-Y < s-X) (f-X < f-Y)
`:finishes Y`	(s-Y < s-X) (f-X < f-Y) (f-Y < f-X)
`:at`	none
`:starts-at`	none
`:finishes-at`	none
`:after-time`	none
`:before-time`	none
where f-Z represents the finish step	of goal Z
where s-Z represents the start step	of goal Z

Figure 3.4 Transformation from Temporal Constraints to Partial Order Constraints.

Since time is a relative notion, we shall choose some arbitrary point in history, which we call time zero or t_0, to which other events and activities can be temporally related. In particular, t_0 is fixed to correspond to the start of the plan. When the plan is executed, all actions are performed a certain number of time units after t_0. For the sake of simplicity, time units range over the set of integers and may be interpreted as seconds, minutes, hours, or days, whichever is appropriate for the particular plan.

With the introduction of the time point t_0, all time constraints on steps and goals can be considered relative constraints; constraints over the temporal distance between two steps. Namely, if T_i and T_j are two time points, a constraint on their temporal distance would be of the form $c \leq T_i - T_j \leq k$. To represent such constraints we specify a closed interval $I_{ij} = [c, k]$.[1] An interval associated with steps is an ordered pair of integers of which the first component represents the shortest temporal distance and the second component represents the longest temporal distance between the two steps. In the case where a step is constrained with respect to t_0, i.e. constrained with respect to absolute time, the components of the interval define the earliest and latest possible time, respectively, that the step may be executed. For example, the interval $I_{i0} = [c, k]$ associated with step S_i represents the relationship: $c \leq S_i - t_0 \leq k$, where S_i will start no earlier than c time units and no later than k time units after the initiation of the plan.

This representation of time and temporal constraints allows for easy translation from the time relationships given in the goal specification language to interval constraints. The temporal database, which we refer to as the *time box*, maintains the list of these interval constraints. The time box is later called upon to verify the temporal consistency of the plan. In the following chapter we will consider the representation of the time box and its associated functionality in more detail. At this point, it suffices to state that the temporal constraints in the goal specification are translated into time intervals constraining the temporal distance and stored in the time box. Figure 3.5 shows the transformations for all the possible constraints.

[1] Allen [2] first introduced the use of interval algebra with respect to temporal issues.

Temporal Constraints	$I_{sx,sy}$	$I_{sx,fy}$	$I_{fx,sy}$	$I_{fx,fy}$
`:after Y (:range` n_1 n_2`)`	none	$[n_1, n_2]$	none	none
`:before Y (:range` n_1 n_2`)`	none	none	$[-n_2, -n_1]$	none
`:overlaps Y`	$[-\infty, 0]$	none	$[-\infty, 0]$	$[-\infty, 0]$
`:during Y`	$[0, \infty]$	none	none	$[-\infty, 0]$
`:equal Y`	$[0, 0]$	none	none	$[0, 0]$
`:starts Y`	$[0, 0]$	none	none	$[-\infty, 0]$
`:finishes Y`	$[0, \infty]$	none	none	$[0, 0]$
`:at n` ($Y=t_0$)	$[0, n]$	none	$[n, \infty]$	none
`:starts-at n` ($Y=t_0$)	$[n, n]$	none	none	none
`:finishes-at n` ($Y=t_0$)	none	none	$[n, n]$	none
`:after-time n` ($Y=t_0$)	$[n, \infty]$	none	none	none
`:before-time n` ($Y=t_0$)	none	none	$[0, n]$	none

Figure 3.5 Transformation from Temporal Constraint to Time Intervals.

Summary

We have extended the UCPOP domain specification language by allowing operators to have a duration. These durations are necessary to ensure that an agent is not performing multiple operations simultaneously. Further, the goal specification language is extended to allow for multiple goals and temporal constraints on goals. Before the actual UCPOP planning process starts, a partial plan consisting of goal steps, causal links, open conditions, and ordering constraints is constructed from the goals and initial state given. This partial plan is given to UCPOP to complete.

CHAPTER IV

INTRODUCTION TO THE TEMPORAL CONSISTENCY CRITERIA

In this chapter and the next we explore the implication of adding temporal constraints to plans. First we focus on theoretical considerations, while the next chapter introduces some of the pragmatic considerations. We begin this chapter by giving a general and formal definition of the *temporal consistency criteria in planning*. These criteria are necessary to determine if a plan is temporally consistent. The criteria are formulated in terms of the temporal constraint satisfaction problem (TC-SAT). Determining if the criteria are met in a plan is a hard problem. To help us prove that, we introduce a known NP-complete problem from scheduling theory in definition 4.2 and we show in theorem 4.3 that the temporal constraint satisfaction problem is indeed NP-complete.

The TC-SAT problem, which we define in definition 4.1, is very general and incompasses more temporal constraints than those considered in chapter III. To facilitate our discussion of finding an efficient algorithm to determine the temporal consistency of a plan, we reformulate the TC-SAT problem into an event scheduling with constraints problem (ESC). The ESC problem is not easier to solve, rather it makes it easier to state simplifying assumptions about the temporal consistency problem. Some of these simplifying assumptions are incorporated into a reduced statement of the temporal consistency criteria. This statement, referred to as simple ESC (SESC), is not able to express all the temporal notions that are expressible by the ESC formulation, but it is sufficient to capture all of the possible temporal constraints introduced in the goal specification language of chapter III (see figure 3.3). This formulation of the problem, however, remains NP-complete. Thus, we migrate from a general statement of the temporal consistency problem in the beginning of this chapter to a more restricted form at the conclusion.

In chapter V, we further explore the nature of the temporal constraint problem. We will develop an algorithm to determine temporal consistency of a plan and attempt to isolate the portions of that algorithm which lead to exponential time complexity. In order to arrive at an efficient algorithm we make further simplifying assumptions about allowable temporal constraints in the plan by restricting our goal specification language. In particular, we eliminate required start times and constraints on durations for goals, leaving only deadlines on goals and durations for tasks. At the end of chapter V we arrive at a polynomial time temporal consistency algorithm for our restricted goal specification language, definition 5.1, and prove correctness in theorem 5.2.

Time Complexity of General Scheduling and Time Constraint Problems

The introduction of time into planning requires an additional step of verification. This step can easily be added as additional criteria before a plan can be considered a complete and correct plan. If these criteria are not met, then the planner must continue to search for an alternative plan. The temporal consistency criteria, which become part of the test for plan completeness, can be defined as a temporal constraint satisfaction problem (TC-SAT).

TC-SAT consists of a set of goals and a set of tasks. The constraints on a goal consist of the time before which the goal cannot start (release time), the time by which the goal must be established (deadline), and a duration for which the goal must hold (length). In addition, the temporal distance between a pair of goals can be constrained by a set of intervals. This implies that the temporal distance must satisfy one of the intervals in the set. An operator, or task, is an action that must be taken by the agent in order to establish one or more of the goals. Tasks also have durations, or lengths, and may be ordered with respect to one another or individual goals. At the point when the temporal consistency criteria are applied, the tasks will establish all of the goals. This is guaranteed by the planner. The essence of the problem is encapsulated in question: is it possible to arrange the tasks in such a way that one agent can execute them in accordance with the constraints and establish the goals within their given time frames? We formalize the TC-SAT problem in definition 4.1.

Definition 4.1 Temporal constraint satisfaction problem is defined by the following instance and question.

TC-SAT Instance: A set G of goals and $\forall g \in G$, a release time r(g), a deadline d(g), and a length l(g).
A set I of interval constraints on the goals.
A set T of tasks that establish the goals in G and $\forall t \in T$, a length l(t).
A set P of precedence constraints on some elements of T and G.

Question: Is there a sequence σ (i.e. a function $\sigma: T \cup G \rightarrow Z^+$ with $\sigma(x) > \sigma(x')$ implying $\sigma(x) \geq \sigma(x') + l(x')$) such that $\forall g \in G$, $\sigma(g) \geq r(g)$, $\sigma(g) + l(g) \leq d(g)$, $\forall I_{ij} \in I$, where $I_{ij} = \{[a_0, b_0], \ldots, [a_k, b_k]\}$ and $a_0 \leq \sigma(g_j) - \sigma(g_i) \leq b_0$ or … or $a_k \leq \sigma(g_j) - \sigma(g_i) \leq b_k$, and $\forall p \in P$, where $p = (x, y)$ and $x, y \in T \cup G$, $\sigma(x) + l(x) \leq \sigma(y)$?

The question of the formal definition above asks if there exists a total function, or sequence, that maps a task or a goal to its start time and meets several conditions. The function, represented by σ, must make sure that none of the tasks overlap, since in our planning domain there is only one agent that can enact a plan and the agent cannot perform more than one action at a given moment. The function must also map the goals to times that adhere to their release times and deadlines. The set of intervals gives rise to the condition on σ, so that the difference in time between two goals must fall within one of the intervals in the set associated with those two goals. The final condition which the function must satisfy enforces the partial order on the tasks.

We formulate the TC-SAT problem in this way, because it makes the similarities with the one-processor scheduling problem more explicit. The one-processor scheduling problem has been explored extensively in the job-shop scheduling literature; some the results of the research can be applied directly to our temporal consistency problem.[4] The one processor corresponds to the single agent in our planning domain. The scheduling problem asks if there is a one-processor sequence of tasks that satisfies the release time and deadline functions for each task (or action). The TC-SAT problem, in comparison, contains the additional constraints of relative temporal relationship between some of the goals (set of intervals I) and some ordering constraints imposed on the sets of tasks and goals (P). Both problems are concerned only with the existence of a sequence that satisfies the conditions. The optimization problems in scheduling (e.g. minimizing task tardiness, minimizing completion times, etc.) also shed light on finding the plan with "the closest fit." However, we will only consider the question of whether a valid sequence exists (i.e., if the time constraints are satisfiable or consistent).

Garey and Johnson proved that the scheduling problem with release times and deadlines (SRTD) is NP-complete. [9] We will use this result to show that TC-SAT is also NP-complete. It is important to realize that the general problem of temporal consistency is NP-complete, since this allow us to focus our energy on either finding a less general formulation of the problem where a polynomial time algorithm may be applicable or refining an exponential algorithm. In the subsequent chapter, we will consider a combination of both of these approaches.

We must first define SRTD more formally before we can proceed to prove TC-SAT is NP-complete. The problem states that there are a number of tasks that must be scheduled. Each task takes a certain amount of time, must start after its release time, and must be completed before its deadline. Given these constraints on tasks, is there a schedule of these tasks such that no tasks overlap and all of the constraints are met? The schedule is represented by a function σ, which maps a task to its start time. SRTD is formally defined in definition 4.2.

Definition 4.2 One processor sequencing with release times and deadlines problem (SRTD) is defined by the following instance and question.

Instance: Set T of tasks and $\forall t \in T$, a length $l(t)$, a release time $r(t)$, and a deadline $d(t)$.

Question: Is there a one-processor schedule for T that satisfies the release time constraints and meets all the deadlines, i.e. a total function $\sigma: T \rightarrow Z^+$, with $\sigma(t) > \sigma(t')$ implying $\sigma(t) \geq \sigma(t') + l(t')$, such that $\forall t \in T$, $\sigma(t) \geq r(t)$, $\sigma(t) + l(t) \leq d(t)$?

Before we proceed with our proof, there is an anomaly that arises in temporal planning, but not in task scheduling that needs to be considered. The release times associated with goals imply that a goal may not occur before the specified release time. When viewed in the scheduling domain this does not present a problem, since the time assigned to the start of a goal can be delayed to satisfy the criteria. However, goals are not executed in the same sense that tasks are. A robot cannot choose to delay the realization of a goal when all of the tasks establishing the goal have already been executed. In other words, as soon as all the conditions of a goal are met, the goal starts, it cannot be delayed. There are two alternatives to rectify this situation. First, release time can be imposed on the task that establishes the last outstanding condition of a goal. The second alternative is to forgo release times for goals. The implications are that a goal can be established any time before its deadline. The planner has already ensured, by way of precedence constraints, that a goal will remain established until all its other temporal criteria are met. Thus a goal γ must be established for the length $l(\gamma)$ in the interval $[r(\gamma), d(\gamma)+l(\gamma)]$, but may be established any time before then and remain established for any period of time after. We choose the less general alternative.

Garey and Johnson proved that SRTD is NP-complete by transformation from the 3-PARTITION problem. With this result it is easy to show TC-SAT is also NP-complete. We proceed by proving that TC-SAT is in NP, giving a polynomial time transformation from SRTD to TC-SAT, and showing that if and only if there is a valid sequence for TC-SAT there is a valid sequence of tasks for SRTD.

Theorem 4.3 TC-SAT is NP-complete.

Proof
TC-SAT is in NP: We can find a sequence non-deterministically and verify its validity by checking the five constraints (non-overlapping tasks, partial ordering on tasks and goals, release time for goals, deadline for goals, and relative temporal distances for goals). The time complexity of the verification process is $O(|T|^2 + |P| + |I| + |G|)$.

SRTD $\propto$ TC-SAT. Transformation from SRTD to TC-SAT.
Instance of SRTD: Set T of tasks and $\forall t \in T$, a length $l(t)$, a release time $r(t)$, and a deadline $d(t)$.
Instance of TC-SAT: A set G of goals and $\forall g \in G$, a release time $\rho(g)$, a deadline $\delta(g)$, length $\lambda(g)$. A set I of interval constraints. A set Θ of tasks that establish the goals in G and $\forall t \in \Theta$, a length $\lambda(t)$. A set P of precedence constraints on Θ and G.

Transformation: Let $G = \{t \mid t \in T\}$. $\forall t \in T$, let $\rho(t) = r(t)$, $\delta(t) = d(t)$, and $\lambda(t) = l(t)$. Let $\Theta = P = \emptyset$. Define $I = \{I_{ij} \mid I_{ij} = \{[l(t_i),\infty], [-\infty, -l(t_j)]\}, \forall t_i,t_j \in T \text{ where } i \neq j\}$. The set I captures the condition that tasks may not overlap. This transformation is quadratic in the size of the set T.

Clearly, if and only if there exists a valid sequence for the instance of SRTD then there a exists a valid sequence for TC-SAT.
($\Rightarrow$) Assume σ is a valid sequence for SRTD, then let $\alpha(t_i) = \sigma(t_i)$, $\forall t_i \in T$. α defines a valid sequence for TC-SAT. The constraints in P are trivially satisfied and no tasks overlap, since there are none. The interval constraints are satisfied because tasks in σ do not overlap. Thus for every pair of goals in G, either one goal is before or after the other. Furthermore, release times and deadlines are met since they are equal to those in SRTD and they were met in σ.
($\Leftarrow$) Assume α is a valid sequence for TC-SAT, then let $\sigma(t_i) = \alpha(t_i)$, $\forall t_i \in G$. σ defines a valid sequence for SRTD. The tasks in σ are guaranteed not to overlap, since the constraint I_{ij} implies t_i is before t_j or after t_j, but never during. The release times and deadlines are met in σ because they are equal to those in TC-SAT and met in α.

Since TC-SAT is in NP, SRTD is NP-complete, and SRTD is polynomially transformable into TC-SAT, TC-SAT is NP-complete.

QED

The above proof is not entirely surprising. Intuitively, SRTD is a "simpler" problem with fewer conditions to meet and SRTD is already NP-complete. The fact that TC-SAT is NP-complete suggests that we must relax or change some of the conditions of the criteria in order to formulate a problem that can be solved in polynomial time. An undesirable consequence of relaxing the conditions is that the notion of temporal consistency will become less general. The kinds of temporal planning problems that we can specify become more limited. Therefore, for every simplification of the problem or relaxing of the conditions we must be careful to consider the implications on the expressiveness of the temporal planning problem.

Reformulating the Temporal Consistency Criteria

Before proceeding to adjust the consistency criteria, we will state the temporal consistency criteria in a less cumbersome formulation. In doing so, a more efficient algorithm for satisfying the temporal constraint criteria may be easier to discover. Note that the entire problem can be formulated using only temporal interval constraints on events. The problem then reduces to finding a temporal assignment to each event that adheres to the interval constraints.

Definition 4.4 describes the event scheduling problem with temporal distance constraints (ESC). We can re-formulate the TC-SAT problem in terms of ESC with a straight forward transformation. The set of events in ESC consists of the all of the tasks and goals, along with a special event (e_0) that marks the beginning of the plan execution. The latter event will be used to measure absolute time with respect to the start of the plan. All of the temporal constraints on tasks and goals are cast into temporal distance constraints between the events. The details of the transformation are contained in the proof for theorem 4.5.

Definition 4.4: Event scheduling with temporal distance constraints (ESC) is defined by the following instance and question.

Instance: A set of events E and a set of constraints on the temporal distance between each pair of events in E. The set of constraints is designated by C, where each element of C is a set of intervals. The element $Ce_i,e_j=\{[a,b], \ldots, [c,d]\}$ means $a \leq \sigma(e_j) - \sigma(e_i) \leq b$ or ... or $c \leq \sigma(e_j) - \sigma(e_i) \leq d$.

Question: Is there a sequence σ (i.e. a function σ:T$\rightarrow$Z$^+$), so that for every pair of events e_i, $e_j \in E$, one of the intervals in Ce_i,e_j is satisfied?

Thus the TC-SAT problem is reduced to an instance of a set of events E and a set of interval constraints C. Notice that for every pair of events there is a set, possibly empty, of interval constraints. For a valid schedule, only one of the intervals in the set need be satisfied. However, it can easily be shown that this simpler formulation of event scheduling with constraints is also NP-complete, as stated in theorem 4.5. The proof is straight forward and therefore only states a formalization of the transformation from TC-SAT to ESC.

Theorem 4.5. The ESC problem is NP-complete.

TC-SAT $\propto$ ESC.
Transformation from TC-SAT to ESC: For every goal and task, define one event. In addition, define an event to represent time-zero (e_0), the starting time of execution. Thus, we have the set of events $E = \{e_0, g_1, g_2, \ldots, g_k, t_1, t_2, \ldots, t_m\}$, where $g_i \in G$, $t_i \in T$, $|G| = k$ and $|T| = m$. The temporal interval constraints are defined as follows:

$\forall I_{ij} \in I$ define $C_{gi,gj} = I_{ij}$.	original goal constraints
$\forall g_i \in G$ define $C_{e0,gi} = \{[r(g_i), d(g_i)-l(g_i)]\}$.	goal release time & deadline
$\forall (t_i, t_j) \in P$ define $C_{ti,tj} = \{[l(t_i), \infty]\}$.	t_i must occur before t_j
$\forall t_i, t_j \in T$ define $C_{ti,tj} = \{[l(t_i),\infty], [-\infty, -l(t_j)]\}$, where $i \neq j$	no overlapping of tasks

Theorem 4.5, of course, is no surprise as we have just restated the original TC-STA problem. However, the problem is now stated in a much simpler form. Now we can proceed by making simplifying assumptions regarding the ESC problem. In particular, let us consider the problem corresponding to the constraints discussed in chapter III.

The ESC problem can be further simplified to allow only <u>one</u> interval constraint per pair of tasks. In chapter III we considered the essential and elementary temporal relationships between two events, whether they be be actions, goals, or absolute time references. None of these elementary relationships require disjunctive interval constraints. Thus, the ESC problem can be simplified to allow only <u>one</u> interval constraint per pair of tasks. The notion of a single agent schedule, however, requires the disjunctive interval constraints. A single agent schedule stipulates that no task may overlap with another task. This means a task occurs either before <u>or</u> after that other task, implying a disjunctive interval constraint.

A single agent schedule is critical to our notion of the temporal consistency criteria. We can recover the restriction of non-overlapping tasks by re-introducing the length of each task as part of the problem instance. By limiting the temporal distance interval constraints to one per pair of events, the following relationship for $C_{ij} = \{[a, b]\}$ occurs: $a \leq \sigma(e_j) - \sigma(e_i) \leq b$. This relationship can be written as two inequalities: $\sigma(e_j) - \sigma(e_i) \leq b$ and $\sigma(e_i) - \sigma(e_j) \leq -a$. The two inequalities lend themselves to being organized in a square interval matrix I, where $I_{ij} = b$ and $I_{ji} = -a$. The new formulation of the one agent task scheduling with single interval constraints is considered in definition 4.6. The transformation from TC-SAT to SESC is identical to the transformation from TC-SAT to ESC, with one exception. The non-overlapping tasks conditions cannot be encapsulated with interval constraints, thus they must be checked explicitly as in TC-SAT.

> <u>*Definition 4.6:*</u> Simple event scheduling with single temporal distance constraints (SESC) is defined by the following instance and question.
>
> *Instance*: Set E of events and $\forall e \in E$, a length $l(e)$. And an $|E|*|E|$ interval matrix I, where $I_{ij} = x$ means $\sigma(e_j) - \sigma(e_i) \leq x$. Thus, each element of the matrix specifies a maximum temporal distance between two events.
>
> *Question*: Is there a sequence σ for E, such that $\forall e_i, e_j \in E$, $\sigma(e_j) - \sigma(e_i) \leq I_{ij}$ and $\sigma(t_i) > \sigma(t_j)$ implies $\sigma(t_i) \geq \sigma(t_j) + l(t_j)$?

Even though we have made some simplifications, the formulation of the problem in definition 4.6 is sufficiently powerful to represent the temporal constraints that can be specified by the goal specifcation language of figure 3.3. It is clear that the formulation of the problem in definition 4.6

remains NP-complete. We have simplified and made several restrictions on the original problem, but the temporal consistency problem remains NP-complete. This leaves us with the question: Are there further restrictions that can be considered such that an efficient algorithm can be applied? This question will be considered in the following chapter discussing the pragmatic aspect of determining temporal consistency. The formulation of the problem in definition 4.6 is the basis upon which we build the implementation of the temporal consistency criteria.

CHAPTER V

IMPLEMENTATION OF THE TEMPORAL CONSISTENCY CRITERIA

In this chapter, we introduce several restrictions on the general temporal constraint problem. Since the temporal consistency criteria problem in its most general form is proven to be NP-complete in chapter IV, we would like to explore the restrictions on the problem in order to to develop an efficient (polynomial time) algorithm to determine the satisfiability of temporal constraints. We do this in the context of our particular implementation. We describe the temporal database which embodies the representation of time and temporal constraints, as well as the temporal consistency criteria algorithm. Several temporal planning issues are raised and clarified with examples. In conclusion, we consider some implications of the restrictions that lead to an efficient algorithm.

To keep track of temporal constraints that are associated with a plan, we introduce a temporal database. This database, called the *time box*, is completely independent of the planner and is solely responsible for temporal information associated with a particular plan. Each plan, therefore, has its own time box. Let us first consider how time points and constraints thereon are represented in our implementation.

Representation of Time Points and Temporal Constraints

We associate time points with the beginning and ending points of actions and goals, as well as absolute points in time, such as 2 o'clock. Temporal constraints on actions and goals, then, become constraints on their associated time points. Time zero (t_0), as discussed in chapter III, is an example of a time point.

The time points that are of particular interest are the time points associated with the start of the plan, the goals, and the actions of the plan. Since all of the actions, goals, and initial conditions are represented by steps in the plan, we associate a time variable with the starting and finishing time point of each step. Each of these variables represents one time point and its domain is continuous. We can then specify constraints with respect to these variables.

As a convention, the variables that specify starting time points are denoted by $?st_i$, where i is the identification number of the step. Similarly, the finishing time points are represented by $?ft_i$. The variable for the starting point of the entire plan is analogous to time zero, as discussed in chapter III, and is given by $?st_0$.

Recall that temporal constraints are given in terms of restrictions on the temporal distance between time points. These time points are given in terms of time variables, and the constraints can be expressed as intervals constraining the values of the variables. The interval $I_{si,fj} = [a, b]$ can now be written more explicitly as: $a \leq ?st_i - ?ft_j \leq b$. Moreover, the constraint can be expressed as a pair of inequalities: $?st_i - ?ft_j \leq b$ and $?ft_j - ?st_i \leq -a$. Solving the temporal consistency criteria amounts to solving a system of linear inequalities.

As we saw in chapter IV, it is convenient to organize these constraints in a two dimensional array D, which is indexed by the time variables. Thus, in our example above, $D[?st_i][?ft_j] = b$ and $D[?ft_j][?st_i] = -a$. If there is no explicit constraint between two time points, the cell in the matrix is initialized with infinity, denoting the temporal distance is not bounded.

In the event that there is more than one interval constraint on the temporal distance of two time points, we store the intersection of the intervals. This is because both constraints can be satisfied only when the temporal distance falls within the window that both intervals share. The intersection of two intervals implies that we store the least upper bound and the greatest lower bound of the intervals in the matrix. We can see this with a simple example. If time points are constrained by the intervals [3, 10] and [5, 12], clearly the temporal distance must lie within [5, 10] to satisfy both constraints. Practically, if we have two intervals, $I_{si,fj} = [a_1, b_1]$ and $I_{si,fj} = [a_2, b_2]$, the matrix will be filled in as follows: $D[?st_i][?ft_j] = \min(b_1, b_2)$ and $D[?ft_j][?st_i] = \min(-a_1,-a_2)$.

Interval constraints over a temporal distance are not the only temporal constraints we need to deal with. The planner also specifies a partial order of the steps. The ordering constraints have the form $S_i < S_j$, implying step i must come before step j. With the addition of time and durations for steps, we need to ensure not only that step i comes before step j, but also that they do not overlap. Thus we have the expression $?ft_i < ?st_j$ or $?ft_i - ?st_j < 0$. We can cast this into $?ft_i - ?st_j \leq 0$ without worry, because even if step i ends at the same instant that step j begins, the necessary conditions are still met. The other relationship we can infer from the partial ordering is $?st_j - ?ft_i \leq \infty$, implying step i can come anytime before step j.

The purpose of the time box is to keep track of the temporal constraints currently associated with a plan. The time box has two access functions that facilitate this requirement. Constraints can either be added to or removed from the time box. The function for adding a constraint requires the two variables representing the time points to be constrained and the constant for which the temporal distance of the two time points must be less. The only condition under which we might want to remove constraints from the time box is when an entire step is removed from the plan. Thus the function which removes constraints requires only the step id, i, as an argument. The function searches the database for any references to step i (i.e. $?st_i$ or $?ft_i$) and removes all occurrences.

The other major task of the time box is to determine the temporal consistency of the plan given all the information in its database. The time consistency function requires only the list of steps from the plan as an argument and returns an indication if the plan is temporally consistent. The algorithm employed to make this determination is the subject of the next section.

Determining Temporal Consistency

The temporal consistency criteria are met if and only if there exists an assignment of values to the variables such that all of the temporal constraints are satisfied. Clearly, we would like to find an algorithm that can make that determination efficiently. The goal specification language described in chapter III induces temporal constraints such that determining temporal consistency is an NP-complete problem. This was shown in the latter part of chapter IV. In this section we endeavor to show where the complexity in the algorithm arises.

We begin our analysis with the formulation of the temporal consistency problem as stated by the definition 4.6. For a simple class of plans, Dechter introduced an efficient algorithm for solving the problem. [7] Illustrated examples will show that progressively more complex plans require revisions of or additions to that algorithm. Through this process, we isolate the complexity of the temporal consistency problem to a class of plans that contain *floating steps*. That is, the possible presence of floating steps in a plan causes the temporal consistency problem to NP-complete. The occurrence of floating steps can be eliminated by restricting the input language for goal specification. The goal specification language is reformulated and a polynomial time algorithm for temporal consistency criteria is presented. A proof of correctness is provided to show that our algorithm is sufficient to establish temporal consistency for this reduced set of temporal constraints. First, we develop a temporal consistency algorithm for simple classes of plans.

Simple Classes of Plans

As was mentioned in the previous section, with the representation of time points and constraints we have chosen, the problem can be seen as solving a system of linear inequalities. This special class of linear inequalities has a convenient graph representation, which we introduced in chapter II. We explore this representation in more detail below.

We can associate a directed edge-weighted graph with the temporal consistency criteria problem, where there is one vertex for each event or time point . An edge between t_i and t_j denotes there is a constraint between t_i and t_j and the label a_{ij} suggests t_j is no more than a_{ij} time units away from t_i. Thus, the constraint interval $I_{ij} = [a_{ij}, b_{ij}]$ ($a_{ij} \leq t_j - t_i \leq b_{ij}$) is represented in the distance graph in figure 5.1.

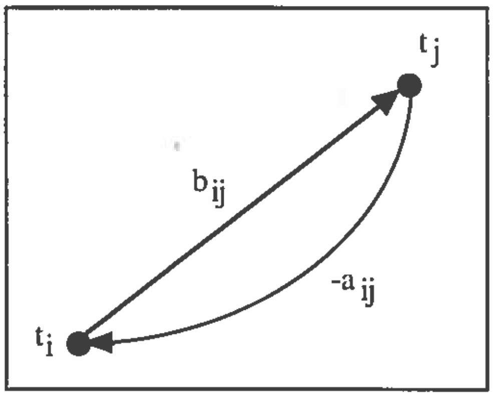

Figure 5.1 A Distance Graph Representation of a Temporal Constraint.

Each path from t_i to t_j going through intermediate vertices $i_0, ..., i_k$ induces the following constraint: $t_j - t_i \leq \Sigma a_{i_{j-1},i_j}$ for $1 \leq j \leq k$. As we have seen, if there is more than one path (i.e., more than one interval constrains the relationship, explicitly or implicitly) a temporal assignment to t_i and t_j will lie in the tightest interval, which implies the shortest path in the graph between t_i and t_j. For instance, consider the example with three events and these constraints, $30 \leq t_j - t_i \leq 50$, $10 \leq t_j - t_k \leq 20$, $10 \leq t_k - t_i \leq 20$.

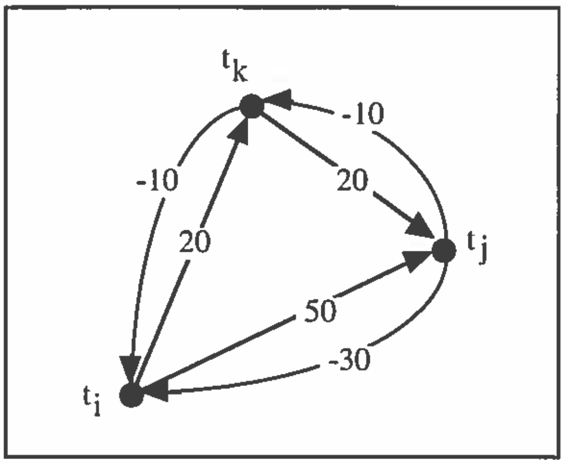

Figure 5.2 A Distance Graph with Multiple Temporal Relationships.

The intersection of the constraints will yield the minimum constraints that the time values of the events must satisfy. For the time points t_i and t_j the modified constraints are: $30 \le t_j - t_i \le 40$ (since t_i is a maximum of 20 time units from t_k and t_k is a maximum of 20 time units from t_j, it follows t_i must be a maximum 40 time units from t_j). In the distance graph of figure 5.2, finding the shortest path from vertex t_i to t_j and from t_j to t_i gives the same result.

This observation is critical in our quest to find an efficient algorithm. Temporal interval constraints on events are consistent if and only if the associated distance graph does not have any negative cycles. If there is a negative cycle in the graph (e.g., from t_i to t_i), then the weights on the intermediate edges in the cycle sum to less than zero. This would imply $t_i - t_i < 0$, which is impossible to satisfy. Conversely, if there is no negative cycle in the distance graph, then there is a shortest path between each pair of vertices. Since the shortest path defines the smallest interval constraint, a temporal assignment to the time points can be made that satisfies the original constraints.

The all-pairs shortest path algorithm can be employed to determine negative cycles and consequently temporal constraint consistency. The algorithm has $O(n^3)$ time complexity (where n is the number of vertices in the graph, or time points in our planning problem).
The matrix algorithm applied to the well-known “Sussman Anomaly” problem, results in accurately determining temporal consistency. The problem states that from the initial state, (on c a) (on c table) (on b table), we wish to meet the goal, (and (on a b) (on b c)). Figure 5.3 shows the initial state of the blocks and the desired state of the blocks. The resultant plan is a linear succession of three steps: (puton c table), (puton b c), and (puton a b).

Depending on the duration for each puton operation and the time constraint given to the goal, the algorithm will return an indication whether the plan meets the constraints. Say that we fix the length of each puton action at 10 time units and that all plans start at a time point with which we associate 0 (i.e. $t_0 = ?st_0 = 0$). It is easy to see that the plan is temporally consistent if we wish the goal to be established at 30 time units or later.

In the figures 5.3 through 5.7, the top portion of the figure represents the plan, with the solid vertical bar at the left indicating the start of the plan. Solid lines or arcs connecting the plan steps or goals denote a temporal relationship. The bottom portion of the figure depicts a blocks world example, where the initial state is on the far left and successive configurations of the blocks are the goal states at increasingly later points in time (i.e., $t_0 < t_1 < t_2 \ldots$). Blocks which are present in the initial configuration and are not present in goal configurations implies that the goal does not stipulate where those blocks must be.

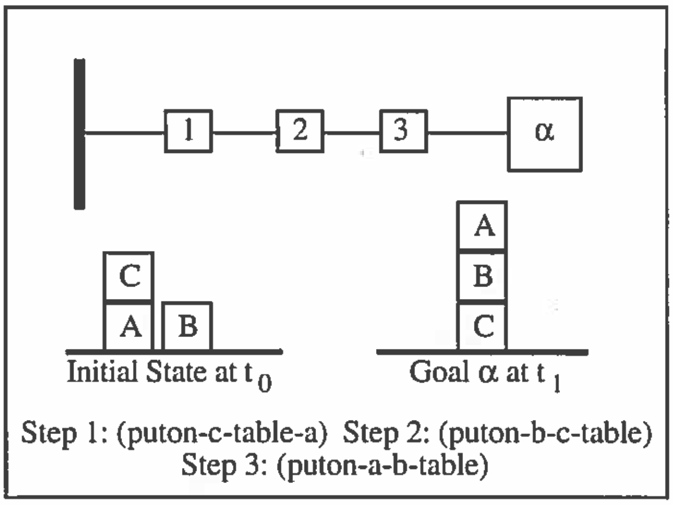
1
2
3
α
C
A
B
A
B
C
Initial State at t_0
Goal α at t_1
Step 1: (puton-c-table-a) Step 2: (puton-b-c-table)
Step 3: (puton-a-b-table)

Figure 5.3 Plan of the Sussman Anomaly.

The *matrix algorithm*, however, is not sufficient in determining temporal consistency of all plans. It makes the assumption that tasks in the planning domain are events which may occur simultaneously. This is not a sufficient condition for the one-agent planning domain. The plan in figure 5.3 does not contain steps that can potentially overlap and, therefore, does not expose the shortcoming of this algorithm. The following example, figure 5.4, demonstrates that a partially ordered sequence of steps that satisfies the temporal constraints, in the sense of the matrix algorithm, may not be linearizable (total order given to all the steps) in such a manner that none of the steps overlap. Note that steps 1 and 2 are not constrained, explicitly or implicitly, with respect to one another. If the goal is constrained to be established at 15 time units, all of the temporal intervals are met, but the plan cannot be successfully executed by one agent. This example suggest that the temporal consistency algorithm needs to be more discerning.

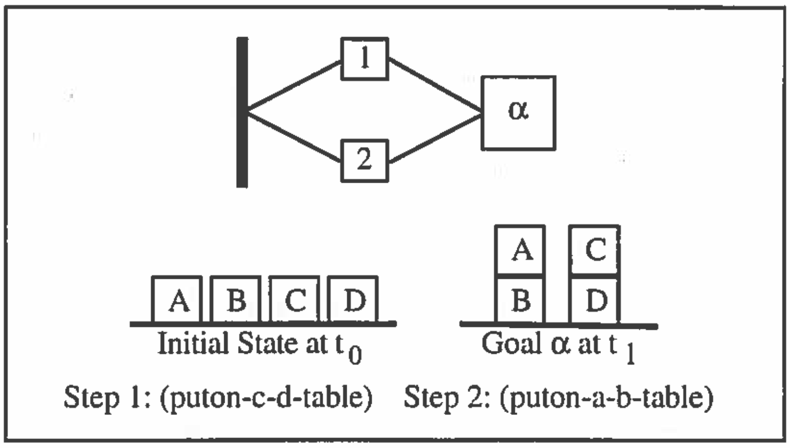

Figure 5.4 Plan with Parallel Steps.

This problem shows we need the summation of the duration of all necessarily preceding steps to be less than the latest time the goal may be established. If this condition holds then the above plan can be linearized. By adding up the durations we account for the amount of time necessary for a single agent to perform the steps. If that amount of time is less than the goal's deadline, the temporal constraints of the goal are satisfied. In general, we add an additional check to the matrix algorithm which guarantees that the sum of all the steps' durations necessarily occurring before a goal is less than the latest start time of that goal. This aspect of the algorithm requires us the differentiate between the "regular" steps and the goal steps. Thus, the time box is also required to keep a list of goal steps.

The more extensive example of figure 5.5 demonstrates the need to perform the summation for all goals. Here step 3 may be executed anytime before goal β; however, the summation of steps between the start of the plan and goal β makes sure step 3 is properly considered. It is interesting to note that the execution of step 3 and the realization of goal α are allowed to overlap. An agent is not engaged in any action when a goal is realized; all of the work is done by the steps that precede the goal. We captured this nuance of goals by representing a goal by two steps in the plan, both with a duration of zero. (See chapter III.) The two steps are mutually constrained by forcing the temporal distance between the two to be equal to the desired duration of the goal. Thus, any step can be executed during, or overlapping with, the duration of a goal, as long as its effects do not negate a proposition of the goal.

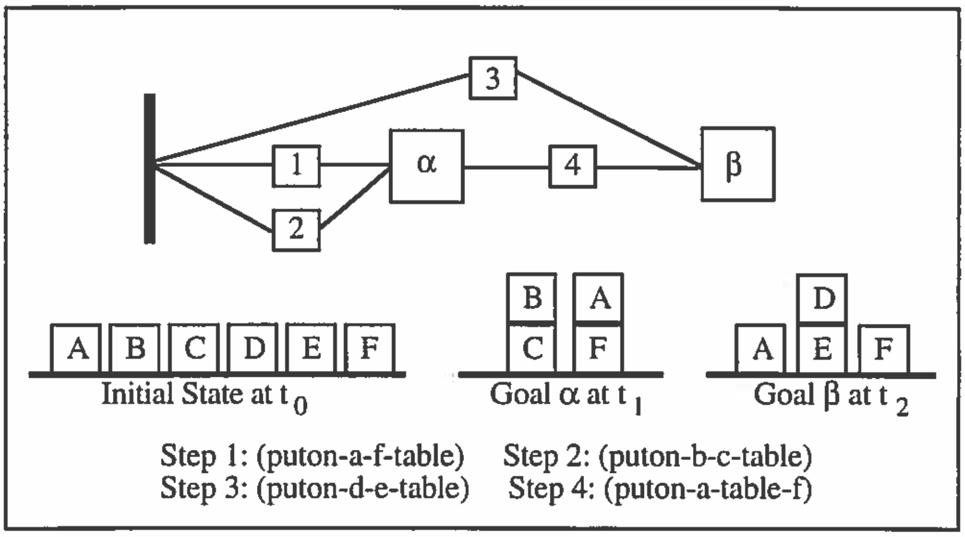
3
1
α
4
β
2
B
A
D
A
B
C
D
E
F
C
F
A
E
F
Initial State at t_0
Goal α at t_1
Goal β at t_2
Step 1: (puton-a-f-table)
Step 2: (puton-b-c-table)
Step 3: (puton-d-e-table)
Step 4: (puton-a-table-f)

Figure 5.5 Multiple Goals for Which Summation Is Necessary.

The more complete algorithm that solves the problem raised by figures 5.4 and 5.5 remains insufficient for more complex plans. Not only do we need to perform a summation between the start step of the plan and all the goals, but we also need to sum the durations of all the steps necessarily <u>between</u> each pair of goals. To see this more clearly, consider the example plan of figure 5.6. There are two set of potentially parallel steps. It may be possible that the summation between the goal β and the start step is less than the earliest start time of goal β. But steps 3 and 4 are restricted to occur after goal α, thus the sum of their durations must be less than the temporal distance between the earliest possible finish time of goal α and the latest possible start time of goal β.

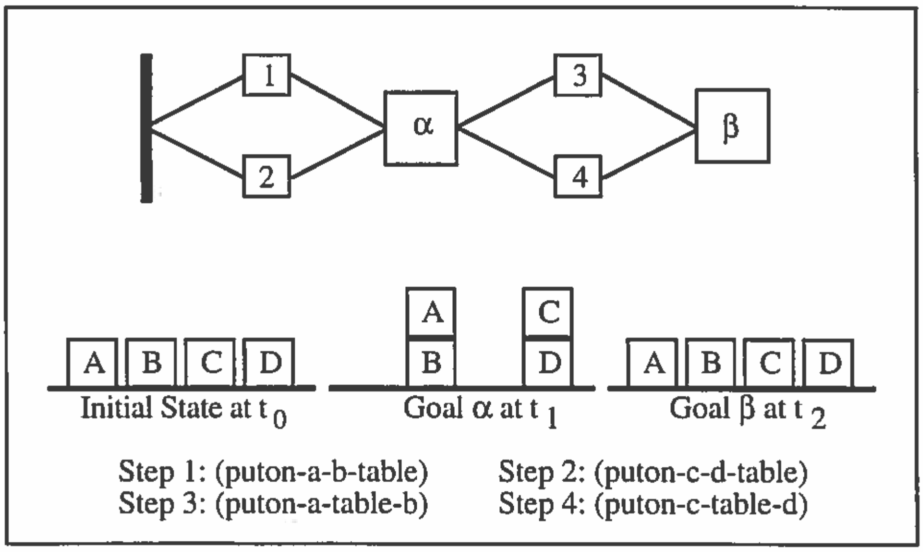
1
3
α
β
2
4
A
C
A
B
C
D
B
D
A
B
C
D
Initial State at t_0
Goal α at t_1
Goal β at t_2
Step 1: (puton-a-b-table)
Step 2: (puton-c-d-table)
Step 3: (puton-a-table-b)
Step 4: (puton-c-table-d)

Figure 5.6 Plan Requiring Summation of Steps Between Successive Goals.

The earliest finish time of goal α, of course, depends on the steps that precede goal α. Our algorithm will need to calculate the earliest possible finish time for all goals. In general, the start time of the first goal will affect the start time of subsequent goals. Therefore, we must calculate the earliest finish times of goals in ascending time order. For goal α, we sum the durations of steps 1 and 2. The difference between the summation of the durations of necessarily preceding steps and the latest possible start time of goal α is the amount of *slack* for goal α. The earliest finish time of the goal α can now be easily calculated: it is the maximum of the earliest finish time of the goal and the latest finish time minus the slack. The reason we use the maximum function here is because we still need to be cognizant of the original constraints on the goal. Thereby, we ensure the goal starts no earlier than allowed.

Now that we have established the earliest finish time of goal α we can determine the slack and the earliest finish time for goal β. For goal β, there are two slack calculations to make. First we consider the slack between goal β and goal α and the slack between goal β and the start step. The final slack value associated with goal β will be the minimum of the two calculated slack values.

In general: for each goal γ, starting with the earliest possible goal, calculate the slack values between goal γ and all earlier goals, including the start step. Slack between two goals, γ' and γ, is defined as the sum of durations of the step necessarily between γ' and γ subtracted from the temporal difference between the earliest finish time of γ' and the latest start time of γ. The minimum of the slack values between goal γ and all earlier goals will determine the earliest possible finish time of goal γ. This earliest possible finish time will be utilized by the slack calculation of subsequent goals. Note that anytime the slack of a goal is less than zero the temporal consistent criteria fails.

The assumption of the summation algorithm is that we can consolidate all of the steps between two goals to be executed consecutively. This assumption is grounded in our earlier discussion concerning the fact that actions or tasks do not have release times, or specific start times. These steps are only constrained by a partial order. Thus, it seems reasonable that the steps between any two goals may be executed as one block with no temporal gaps separating them. This block of step can then be executed as early as possible.

For the classes of plan that we have discussed so far, the algorithm for determining the temporal consistency criteria involves a two step process. First, we interpret the time variables and their constraints as a distance graph and calculate the transitive closure with the matrix algorithm. The time complexity for this algorithm is $O(n^3)$, where n is the number of steps in the plan. Second, we calculate the slack of all the goals, a task requiring $O(nm^2)$ time, where m is the number of goals and $m < n$.[1] Thus, the temporal consistency algorithm we have presented is a polynomial time algorithm. As we will see next, the algorithm is not adequate for all classes of plans.

Plans with Floating Steps

The assumption made with respect to the summation algorithm is adequate for most cases, unfortunately there are plans for which this assumption does not hold. These instances involve plans that have a floating step. A *floating step* is one whose temporal interval spans more than one goal. For an example see figure 5.7. The problem arises because we erroneously assume that steps before goal α combined with the steps between goal α and β can be executed as one block. The slack for goal α is five time units and slack between goal β and δ is also five time units. The sum of these slacks gives ample time to execute step 5. When a step only spans one goal, as in figure 5.5, the temporal gap is indeed the size of the sum of the slacks for goals α and β. However, the slacks of a pair of goals separated by a third goal do not sum in this manner.

If our algorithm for simpler plans makes a determination that a plan has a temporal inconsistency, then we can be assured the temporal consistency criteria fails. There are at least two possible adjustments that can be made to the current temporal consistency algorithm to ensure it is sufficient. The first adjustment endeavors to take floating steps into account, but this is computationally expensive. The second relaxes the initial conditions of temporal consistency so that the floating steps problem is eliminated. If the floating steps problem is eliminated then the polynomial algorithm of the previous chapter can be utilized to establish consistency.

[1] m must be less than n, because goals are represented as two steps in a plan. See chapter III.

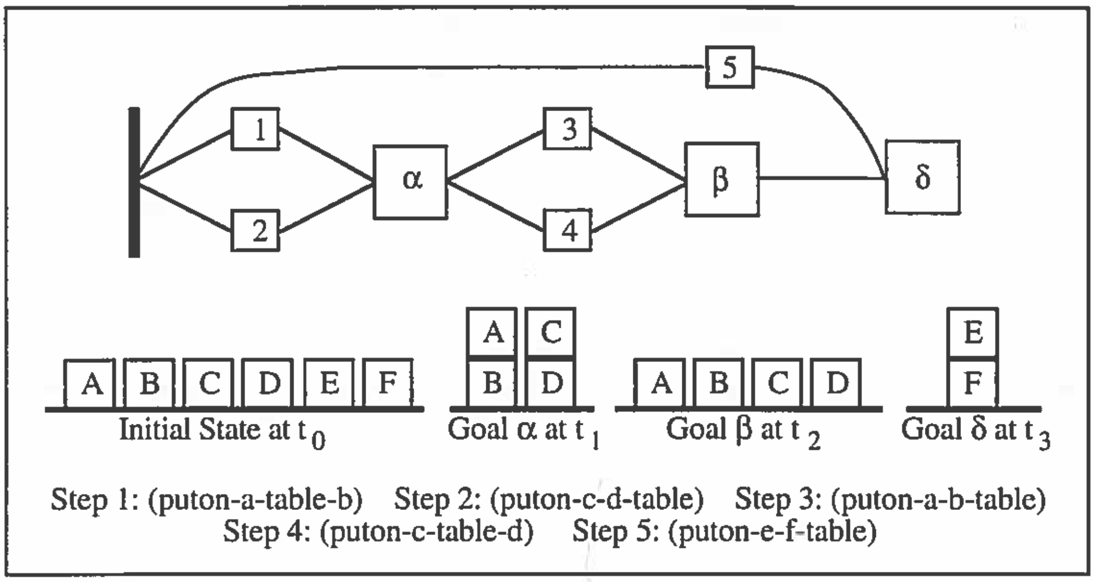
5
1
2
α
3
4
β
δ
A B C D E F
Initial State at t_0
A C
B D
Goal α at t_1
A B C D
Goal β at t_2
E
F
Goal δ at t_3
Step 1: (puton-a-table-b) Step 2: (puton-c-d-table) Step 3: (puton-a-b-table)
Step 4: (puton-c-table-d) Step 5: (puton-e-f-table)

Figure 5.7 Plan with a Floating Step Spanning Multiple Goals.

First we consider how our algorithm changes if we take the floating steps into account. One approach would be to consider complete linearizations of the plan through backtracking. Each linearization would be checked for temporal consistency. However, we can do better by restricting the search to placement of floating steps. This can be accomplished by adding a third component to the polynomial temporal consistency algorithm of the previous section. We would take floating steps into consideration only after the other two aspects of our algorithm have not produced an inconsistency.

Each floating step has a set of possible intervals which restricts it to span no more than one goal. For example, in figure 5.7 step 5 can be placed either between the *start* step and goal β or between goal α and goal δ. Both of these intervals are consistent with the partial ordering of the plan. We can perform a backtracking algorithm over the assignment of floating steps to an element of its interval set. As with so many problems in artificial intelligence, this approach brings us back searching an exponential search-space. In comparison to the approach above, however, the space is a function of the number of floating steps and goals which is likely to be significantly smaller than the number of steps.

We wish to keep the temporal consistency criteria from requiring an exponential-time algorithm. A second possible solution to the floating step problem is not to allow time constraints on the start times of goals and the durations of goals. Thus, we maintain only deadlines for goals and disregard release times, or start times, completely. This implies that goals are met as long as they are completed by their deadline, or latest start time. This is in contrast to our earlier condition which stipulated that the goal may start anytime but must also hold at some specific time for a specific duration. By relaxing the release or start time condition, all of the steps in the plan can be executed consecutively. The earliest possible start time of a goal is merely the sum of the durations of the steps which necessarily precede the goal. It is also necessary to relax the goal duration restriction. Without this stipulation, there will remain pockets of time which cannot be combined with others. It was these pockets of time that resulted in the need for an exponential back tracking algorithm to fill the pockets with floating steps. With this less restrictive condition on the start time and duration of a goal, the matrix algorithm and the summation of steps preceding a goal is sufficient to establish temporal consistency. Definition 5.1 gives a formal statement of the algorithm and theorem 5.2 establishes its correctness.

Definition 5.1 The following algorithm determines temporal consistency of a plan, where goals have no temporal constraints on durations or start times.

Input: Set of goals G, set of steps S, and an interval matrix D. The matrix D contains the constraints on the temporal distances between all of the time points. The starting and finishing times of steps and goals are represented as time points. If ?x and ?y are time points, then an element D[?x][?y] = z is interpreted as the constrraint ?x - ?y $\leq$ z. The duration of step s_i ($d(s_i)$), is denoted in the matrix, $D[?st_i][?ft_i] = -d(s_i)$ and $D[?ft_i][?st_i] = d(s_i)$. As noted in chapter III, each goal is represented by two steps with 0 duration, a *goal-start* step and a *goal-finish* step. Since the duration of the *goal-start* step is zero, the start time point and the finish time point of the *goal-start* step are identical and denoted by $?st_g$ (for some goal g). Likewise, the finish time point of the *goal-finish* step is denoted by $?ft_g$. A deadline δ for a goal g is denoted in the matrix as $D[?st_g][?st_0] = \delta$. Where $?st_0$ refers to the start time of the plan (t_0).

Algorithm: Perform the matrix (all-pairs shortest path or transitive closure) algorithm on D. If there is a negative cycle detected (i.e. a negative value along the diagonal of D), the plan is temporally inconsistent. If there is no negative cycle, continue. For each goal $g \in G$, find the list L of steps that necessarily precede g. L consists of all steps s_i for which $D[?ft_i][?st_g] < 0$. Sum the durations of all the steps in L. That is, $\Sigma d(s_i)\ \forall s_i \in L$. If the sum is less than the deadline for goal g ($D[?st_g][?st_0]$), the goal can be established and meet its deadline constraint. Only if all goals meet their deadline constraints is the plan considered temporally consistent.

Time complexity: The matrix calculation can be performed in $O(|S|^3)$ time. For each goal, finding the list L is $O(|S|)$ and the summation operation is $O(|S|)$, and therefore is bounded by $|S|^2$. Thus, the time complexity of the entire algorithm is $O(|S|^3)$.

Theorem 5.2 The algorithm of definition 5.1 correctly establishes temporal consistency.

Proof: To prove that the algorithm determines temporal consistency, we must show that only if all goals can be established by their designated deadline, is the plan consistent. Choose any arbitrary goal $g \in G$. If a step does not necessarily precede g, than we can assume that it will be performed after g has been established. This assumption is reasonable since all steps will necessarily precede at least one goal. (Steps are only introduced into the plan to establish goals.) The assumption that goals do not have constraints on durations or start times guarantees that there need be no time when the agent executing the plan is idle. That is, there is no need for temporal gaps between the execution of steps. Steps can be executed consecutively and the sum of the durations of all the steps necessarily preceding g will account for the time necessary to establish g. Therefore, if the time to establish g is less than the time between the start of the plan and the deadline for g, then g meets its temporal criteria. Since we made no assumptions about our choice of g, this criterion is sufficient for all $g \in G$.

QED

With the added restrictions that goals no longer have constraints on start times or durations, the goal specification language shown in figure 3.3 contains temporal relationships that cannot be expressed. Figure 5.8 shows the revised goal specification language. Since goals cannot be constrained to have specific start times, the only reference to absolute time that a goal specification can make is `:before-time`, which establishes a deadline for that goal. For similar reasons, the `:range` operator has been removed.

```
<goal-spec> ::= ( [:name <goal-name>] :goal <goal>
                  [time: <time>] ) [<goal-spec>]
<goal>      ::= ( :and <prop> <prop>*) | <prop>
<prop>      ::= ( <string> <variable>*)
<goal-name> ::= <string>
<time>      ::= ( :and <time-cns> <time-cns>*)   | <time-cns>
<time-cns>  ::= ( :before        <goal-name>     |
                  :after         <goal-name>     |
                  :equal         <goal-name>     |
                  :overlaps      <goal-name>     |
                  :during        <goal-name>     |
                  :finishes      <goal-name>     |
                  :starts        <goal-name>     |
                  :before-time   <number> )
<variable>  ::= ?<string> | <string>
<number>    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <number>
<string>    ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n |
                o | p | q | r | s | t | u | v | w | x | y | z |<string>
```

Figure 5.8 Simplified Goal Specification Language Syntax.

While the first specification gives an exponential time algorithm, it is more expressive than the latter language above. Under what conditions is the temporal representation of goals in figure 5.8 adequate? All domains where goals have no necessary length and are irreversible or static are good candidates for being modeled by our temporal planner with the restricted temporal representation. Goals in this domain cannot be threatened by later steps once they have been established. Such goals might be referred to as milestones. Project planning is a prime example where goals, or milestones, may be met anytime before the deadline and do not have a duration. It is clear that the representation is not adequate for all planning domains. However there is a sizable class of problems for which the limited temporal goal constraints works quite well.

CHAPTER VI

PLAN EXECUTION IN A DYNAMIC ENVIRONMENT

The second extension that we have made to traditional planners is to add a plan execution module. The plan execution module, referred to as Exec, is an external interface to the UCPOP planning engine. This interface not only handles the execution of a plan, but is also responsible for the input of the planning domain, the initial world state, and a list of initial goals as described in chapter III. In this chapter we will discuss the aspects of Exec which incorporate plan execution, the dynamic acquisition of new goals, and a dynamic, current world model. We start with in-depth analysis of plan execution followed by a discussion of the modeling of a dynamic world.

Executing Steps in a Plan

The process of executing a step in a plan can be broken into two parts. For one, we must determine which step is the best candidate for execution. Secondly, the effects of the chosen step must be applied to the *start state*, which will change as a result. Let us first consider the implication of applying the effects of a step.

Applying the Effects of a Step

The state of the world is modeled as a set of propositions that describe what is true in the world and, by the closed world assumption, what is not true in the world. The world at the time we execute a step is referred to as the *current state*. The portion of a step that affects the state of the world is called its effects. These effects embody propositions that reveal how the world has changed in the course of executing the step. If the effect is a negation of a proposition, then the proposition must be removed from the start state. Otherwise, the proposition is merely added to the state.

If the step contains conditional effects, we must first survey the state of the world before applying the effect. This is easily accomplished by checking if the condition of the effect can unify with any proposition in the current state. (If the condition asserts a negation we merely check to make sure the condition cannot unify with any proposition in the current state.) The effect is applied only if a unification can be found. Similarly, when a step contains for-all effects, all the bindings which satisfy the conditions are considered in the application of the effect.

Once the current state has been updated, the plan must be updated to reflect the execution of a step and the new state. The step that was executed can be removed from the plan and the current time can be incremented by the duration of that operation. If the step was a producer of causal links, these links can now be redirected to the *start* step. Also, the the causal links that were consumed by the recently executed steps are no longer needed and are deleted. These are the only changes we need to make to the plan. Unlike the case where changes to the current state were perpetrated by an external agent, the planner took into account the changes to the current state made by steps in the plan and, therefore, no replanning is necessary. The process of choosing a step and executing it continues until all of the user goals have been satisfied. This, of course, is only possible if the user stops supplying new goals.

Determining Which Step to Execute

Deciding which step to execute is critical to the successful completion of the plan. The complete plan returned from the planner has ordering constraints that must be considered. At the very least the step chosen to be executed must be a possible first step. That is, no other step must necessarily precede the chosen step. This condition is easily verified given the temporal constraints maintained by the time box. The question, however, is whether there is a step among the list of possible first steps, which would make the best candidate for execution. We shall now consider this question.

What might make one step a better candidate for execution than another? In a closed world, where there is only one agent changing the state of the world, the choice does not matter as long as it meets the minimum requirement stated above. In a world where conditions change unexpectedly and new goals can be added at any time, we wish to leave as much freedom or slack in the plan as possible. There is no sense in establishing preconditions that may later be negated by an unexpected turn of events. Conversely, the earlier that tasks are completed the more time is available for potential new goals to be satisfied. The two viewpoints lead to different strategies in choosing a step to execute. In our implementation we adopt the latter point of view.

When the plan returned from the planner is completely linearized there is only one possible first step that can be applied to the current world. This is not a very interesting scenario. On the other hand, a partially ordered plan provides an opportunity to make an informed decision. In either case, we need to be always on the lookout for goal steps, since they have special meaning. When a *goal-start* step is a candidate for execution, it must be chosen. This is due to the nature of goals: they cannot be delayed. They behave like events: once their preconditions are met, they happen on their own accord. The agent cannot choose to stop a goal from occurring. Likewise, care must be taken to note when a *goal-finish* step can be executed. Neither goal step affects the state of the world, but it is important to notify the user that a goal has been satisfied. Now, let us consider how to choose a step from among steps that are not goal steps.

First, with every goal we associate a list of steps that necessarily precede that goal. All the steps in this list must be executed before the goal can be established. It is possible that the same step ends up in several different lists. We then order the goals in ascending order with respect to their latest finish time. All of the necessary temporal information is readily available in the time box associated with the plan. Once we assemble a list of candidate steps, we can take the intersection of that list with the list of steps associated with the first goal. The idea is that we execute a step associated with the goal that must be done the earliest. We can do better, however. The intersection will result in a smaller list from which we must choose a step. We want to choose the step that is also associated with another goal. In this manner, we can continue to take successive intersections until we are left with one step. This step is the one we want to execute. If no single step is found we merely choose one from the previous non-empty list.

If at all possible, the agent ought not to be idle. With the above approach to selecting a step to execute, the agent is only idle when all steps in the plan necessarily follow a goal that has not yet reached its earliest finish time. In this sense, our approach is greedy in its attempt to finish all of its tasks as soon as possible. As mentioned earlier, this is certainly not the only approach. A comparison with other approaches would make an interesting subject of further study.

Dynamic World

In our model, the world is considered dynamic because there is possibly more than one source of change in the world. Traditional planners make the assumption that the only change in the world is the agent enacting the plan. We assume, however, that there is an outside agent which may affect the state of the world. The changes in the world may be due to physical changes in the realworld or due to inaccurate information. If the world unexpectedly changes, the plan may no longer be guaranteed to work properly. That is, the preconditions of certain steps may no longer be present in the current world and the plan will have to be adjusted.

The outside agent may also specify additional goals to be met. These new goals may also render the current plan incomplete. Both situations require replanning. However, the current plan may be of some value in that it embodies information about steps and their sequences that may not have to change. Thus, we must decide which portions of the plan can be salvaged. A minimal, partially complete plan can then be augmented with the new world state information and given to UCPOP to solve.

Interleaved with the execution process, the execution module (Exec) polls for world state changes and new goal specifications from an external agent, henceforth referred to as the user. Between the execution of each step of the plan, the user is prompted for additional state information and a list of new goals to be satisfied. If the world does not change and there are no new goals to consider, the execution process proceeds unhindered. On the other hand, if the user wishes additional goals to be satisfied, Exec must incorporate the new goals into the plan. New goals are specified in the same manner as the initial goals. The new goals can specify temporal constraints and can also make references to other goals by name. The new goals are converted into steps; propositions of the goals are added to the open conditions of the plan (see chapter III). If new goals are the only changes the user specified, then the new, augmented plan is passed to UCPOP to solve.

Change to the world state can either be the negation of an already existing proposition or an assertion of a new fact in the world. Changes in the current state may affect the plan. Steps may have depended on propositions no longer holding in the current state. These propositions now need to be re-established by additional steps. Moreover, it is possible that the new proposition satisfies a precondition of an existing step in the plan. This step may no longer be needed. It is clear that the plan needs to be refined or repaired to account for the new state of the world.

The current state can easily be updated by adding the new proposition or deleting the proposition if the proposition from the user was negated. For each new proposition added to the current state, if the proposition is the object of a causal link, that causal link can be removed from the plan. For each causal link that is removed, we check which step established the link. We then chech to see if that step has become superfluous. A step S is superfluous in the plan if S is not a source in any of the remaining causal links. If S is superfluous, it can be removed from the plan along with all causal links that establish its preconditions. This, in turn, may make other steps superfluous. This process guarantees that all the steps that remain in the plan are necessary. This is important if we wish to find a plan that meets the temporal constraints. The time that it takes to execute superfluous steps may cause a goal to miss its deadline.

If it is the case that a proposition is removed from the current state, all casual links whose source is the *start* step and whose object is the proposition being negated must be removed from the plan. That proposition becomes an open condition of the current plan. The implication is that the planner must re-establish those propositions or preconditions by means of adding steps. The process of adding and removing steps and causal links involves directly manipulating the data structures of the plan. In addition, if a step is removed from the plan, all of the temporal constraints associated with the step must also be deleted from the time box.

The last thing that remains to be done when propositions are added or deleted from the current state is to update the *start* step of the plan. Recall that the *start* step embodies the initial or current state of the world in its effects. Once this is accomplished, the updated plan is given to UCPOP to complete. All of the planning and its associated logic is left completely to the planning engine, UCPOP. Thus, the execution module is decoupled from the planning engine. Of course, if no new goals are added and no state changes occur then there is no need to replan.

Summary

The UCPOP planner has been extended with an executive module, Exec, that models a dynamic environment in which it executes a partial-order plan. It is capable of accepting input from a user, who may specify a change in the current state of the world or provide additional goals to be satisfied. The dynamic world model attempts to minimize replanning by saving as much of the planning information as feasible. Moreover, the execution model attempts to make wise decisions concerning the action which the agent should take. The execution model is mainly concerned with finishing tasks as early as possible.

CHAPTER VII

SUMMARY AND DIRECTIONS FOR FURTHER WORK

Summary of Extensions to Classical Planning

Many planners make simplifying assumptions about the world with which the planner interacts. Even though these simplifying assumptions make the planning process more tractable, they have impaired the application of these planners to many real-world problems. In this thesis, we have studied the implications of relaxing two of these assumptions. The classical partial-order planner has been extended to reason about time and to account for a dynamic execution environment.

The framework for implementing the extensions to classical planning does not depend on the particular planning engine. That is, the algorithms that we describe are sufficiently general to apply to any partial-order planner. For an example of an implementation of our algorithms, we have chosen UCPOP as our planning engine. UCPOP is a generative, partial-order planner, which takes a partially specified plan as input and returns a complete plan. We do not directly change the functionality of UCPOP, rather we provide a framework in which the original planner can be utilized.

In order to account for time in a plan, we must enhance the expressiveness of the input language for the planner. In particular, the domain specification language must allow for the inclusion of time durations for operators and the goal specification language must allow for temporal constraints on goals. To shield the planner from the changes in the input language, the input is parsed and preprocessed. The new temporal information gleaned from the input is stored in a temporal database, called the time box.

The introduction of time into a plan requires that additional criteria be met before a plan is considered to be complete. These temporal consistency criteria are applied to the plan returned by the planner. Each plan is augmented with a time box, which contains all of the temporal information and algorithms necessary to determine temporal consistency of the plan. If one of the temporal consistency criteria is not met, the planner must continue its search for a suitable plan.

The temporal consistency criteria assures that a complete plan has a linearization of the steps that satisfies all of the temporal constraints. In a general form, the problem of determining temporal consistency of a plan is NP-complete. By restricting the inclusion of several temporal constraints in our input specification language, the temporal consistency problem is computable in polynomial time. This restriction, however, reduces the expressiveness of our temporal planner.

The second extension to classical planners is the provision of a dynamic execution environment for the plan. The assumption for most classical planners is that the agent executing the plan is the only agent affecting changes in the world. We consider the addition of an external agent, which perpetuates unexpected changes in the world. In the process of execution, the external agent may also specify new goals to be satisfied by the plan.

When unexpected changes occur, the current plan of action may be rendered inadequate. Actions may depend on certain facts or conditions that are no longer true or actions may become superfluous. The unanticipated changes to the world need to be reflected in the plan. The execution module alters the plan to take the changes into account and the repaired plan is once again given to the planner to complete. The addition of new goals in the middle of execution implies that replanning has to occur. The new goals must be considered in the context of the previous goals, but the current plan need not be completely abandoned.

The thrust of our consideration of a dynamic environment is to minimize the extent of replanning. Only portions of the plan that are directly impacted by a change in the environment need to be altered. The work invested by the core planner on unaffected portions of the plan should not be discarded. Thus, changes to the environment that do not adversely impact the current plan, require no replanning.

Appendix A and B show portions of the implementation of the temporal planning extension and plan execution in a dynamic environment, respectively. Along with the LISP code, there are some annotated examples explicating nuances of the algorithms and issues discussed in the previous chapters.

Further Work

Our analysis of temporal planning and dynamic execution in this thesis is certainly not complete. There are many issues and possible extensions that we do not address. The purpose of this final section is to discuss some of those interesting issues. We do not propose to solve them, rather we highlight areas that require further study and research. First, we consider extensions of temporal planning, followed by a discussion of plan execution.

Extensions to Temporal Planning

Chapter V addressed several solutions to the floating step problem. The problem arises when a step is temporally ordered in such a way that it spans the temporal intervals of several goals. One solution requires an exponential backtracking algorithm to eliminate floating steps by forcing the step to span no more than one goal. Even though backtracking is a search through an exponential search-space, the domain of the search space (the set of floating steps) might be relatively small compared to the set of all steps. It would be interesting to perform experiments to determine the impact on computational time that the backtracking solution would have. This backtracking algorithm is not currently part of our implementation.

An important issue that has not been adequately addressed in this thesis is the ending of goals. It is not clear what we mean when we say a goal has ended. Does the ending of a goal connote that one or all of the propositions of the goal no longer hold or does it mean we are simply no longer interested in the propositions of the goal? These two alternative give very different strategies for planning. We have assumed throughout this thesis that goals end as soon as they have met their temporal criteria and we are no longer interested in them. In other words, goals end and are not prtotected from threats as soon as possible. This has significant implications on such constraints as `:during`. With our scheme, "goal α during goal β" does not capture the notion that as long as goal α holds goal β must hold; it merely states that for some finite period of time, goal α holds while goal β holds. Conditions of goal β may be negated in the world long before conditions of goal α.

An alternative approach is not to consider the goal as ending until one or all of its conditions are negated in the world. In our above example, this would imply that a step that negates a condition of goal α would have to come after a step negates a condition of goal β. This makes the planning process more complex. The planner may have to introduce steps in the plan for the single purpose of negating a condition of the goal. In most classical, partial-order planners this is not an intuitive notion.

Our notion of temporal planning can be further extended by allowing for variable length duration of operators. We cannot always determine before hand exactly how long a task will take to complete. Often the length of performing the same task will vary or depend on the circumstances in the world. Durations of tasks could be determined probabilistically at execution time. Another approach to adding variable length durations is to make the duration of an operator a function of the state of the world. This approach is much like the conditional effects of an operator. The state of the world at the time of executing an operator will determine how long it takes the agent to complete the operation.

Currently, our temporal planner does not plan for interaction with other agents. The execution model is able to detect unanticipated changes in the execution environment, but the planner does not take active steps to account for possible multiple agent interaction. A possible extension of our temporal planner is to allow for scheduled events in the planning domain. These events occur at predetermined times, without requiring any action on the part of the agent. Events could be utilized by the agent as sources for establishing preconditions of tasks or they may be threats to causal links. An example of how scheduled events can be utilized by a planner can be seen in Vere's planner. [21]

Extensions to Plan Execution in a Dynamic Environment

Critical to the correct execution of a plan is the choice of step to execute. A partial-order plan may have several candidates to choose from. Without the notion of a dynamic environment, the selection of a step from among the candidates is unimportant. Since the environment might change while executing the current plan, the choice of step may influence the replanning process. The researchers in the areas of job-shop scheduling and project planning have considered several scheduling approaches that may be applicable to choosing the best step from among the candidates. These approaches include critical path analysis and slack time calculations. Heuristics based on these approaches could be used to extend plan execution.

A shortcoming of the current implementation of the execution module is the synchronous input method of state changes and new goals. Every cycle of the algorithm executes a step and then asks the user for input. This means that no state changes can occur while the agent is executing a step. Likewise, the planner cannot plan for new goals while it is executing a step. An interesting area for further study is to determine the impact of allowing asynchronous input by the user. This issue raises such questions as: "When do the effects of a step become part of the current state?" and "How does a state change in the middle of performing a task affect the outcome of that task?"

The output of the core planner is one of possibly many complete plans that satisfy the goals. Based upon certain heuristics, the planner searches the plan space and returns the first complete plan it encounters. The alternate, complete plans in the search space are abandoned. Once we start executing the plan, new goals can be specified. The new goals are incorporated into the current plan and the planner is asked to complete the plan once again. It is possible, however, that the new goals cannot be satisfied in the context of the current plan. If this is the case, the execution module informs the user that the goal is not satisfiable and continues with the execution of the original plan. A better solution to this scenario is to also consider more plans than just the current plans. It may be that the new goals are satisfiable within the context of an alternate plan that was originally abandoned.

Conclusion

We have explored the implications of relaxing two significant assumptions made by classical planners: the assumptions that actions are instantaneous and that the agent executing the plan is the sole cause of change in the world. Our analysis has shown that relaxing those assumptions results in making the planning process considerably more complex, but also gives tremendous power for solving new classes of planning problems. It is clear, however, that much work still remains in our exploration of temporal planning and dynamic plan execution.

APPENDIX A

IMPLEMENTATION AND EXAMPLES OF TEMPORAL PLANNING

Temporal Planning Implementation Details

In this appendix we present portions of the implementation of our temporal planner and examples which explicate the use and operation of the planner. First, we show several of the critical functions in the implementation of our temporal planner. The LISP code shown below constitutes the main portion of the *time box*. The functions and algorithms are also described in chapters III and V. The program segments are followed by a number of examples involving temporal planning problems.

The first three functions shown below, `tb-add-constraint`, `tb-add-goal`, and `tb-remove-constraint`, are the access functions needed to manipulate the contents of the time box. They are invoked only from the executive module, Exec (see figure 1.1). The final two functions, `slack` and `tb-time-consistent?`, implement the temporal consistency criteria. The latter is part of the test to determine if a given plan is complete, and therefore, is called from the core planner UCPOP (see figure 1.1). The function `slack`, is an auxiliary function called by `tb-time-consistent?`. Note that the slack calculations performed in `slack` are saved in a special field in the time box and are used to determine the order of execution of the steps in the plan (see appendix B). Other functions needed to implement temporal planning are not shown here.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; A procedure to add a time-constraint to the time-box.  The implied temporal constraint is: sym1 - sym2 ≤ c      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tb-add-constraint (time-box sym1 sym2 c &key (copy t))
  (let* ((constraint (list sym1 sym2 c))
         (box        (if copy (copy-time-box time-box) time-box)))

    ;; add the constraint in the hash table
    (setf (gethash sym1 (time-box-table box)) (adjoin constraint (gethash sym1 (time-box-table box))))

    ;; reset the consistency array, add the variables list and return the new time box
    (setf (time-box-c-array box) nil)
    (setf (time-box-var-list box) (adjoin sym1 (time-box-var-list box)))
    box))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Add the pair (start-id . finish-id) to the goal list                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tb-add-goal (time-box start-id finish-id &key (copy t))
  (let ((new-box (if copy (copy-time-box time-box) time-box)))
    (setf (time-box-goals new-box) (cons (cons start-id finish-id) (time-box-goals new-box)))
    (setf (time-box-c-array new-box) nil)
    new-box))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Remove the time constraints associated with a step id                                                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tb-remove-constraints (time-box step &key (copy t))
  (let* ((new-box   (if copy (copy-time-box time-box) time-box))
         (table     (time-box-table new-box))
         (var-list  (time-box-var-list new-box))
         (goal-list (time-box-goals new-box))
         (id        (p-step-id step))
         (s-sym     (make-sym "?st" id))
         (f-sym     (make-sym "?ft" id))
         (t-fun     (lambda (s c) (or (equal s (first c)) (equal s (second c))))))

    (dolist (sym var-list)
      (setf (gethash sym table) (remove-1 s-sym (gethash sym table) :test #'t-fun))
      (setf (gethash sym table) (remove-1 f-sym (gethash sym table) :test #'t-fun)))

    ;; Remove references to id in the variable and the goal list
    (setf (time-box-var-list new-box)
       (remove id var-list :test #'(lambda (id var) (string-equal (string-left-trim "?stft" var)
                                                                   (write-to-string id)))))
    (setf (time-box-goals new-box) (remove id goal-list :test #'(lambda (x y) (= x (car y)))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Determine the amount of excess time there exists with respect to all the goals in the plan.  Excess time is  ;;
;; determined by subtracting the sum of the durations of all the steps that must occur before a goal from the   ;;
;; latest start time of that goal. return a list in this form:                                                 ;;
;;    ((goal slack (preceding steps ...)) (goal ...) ... ))                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun slack (d goal-list num-of-steps assoc-list &aux slack-list)
  (dolist (goal goal-list slack-list)
    (let ((list nil)         ; list of steps necessarily before a step
          (sum  0)           ; sumation of duration of steps in list
                             ; find the matrix index of the goal
          (i    (conv-sym1 (make-sym "?st" (car goal)) assoc-list)))

      ;; compile the list of steps necessarily preceding the goal i
      (do ((j 1 (1+ j))) ((= j num-of-steps))
        (if (and (not (= i j)) (not (eq (aref d j i) *infinity*))
                 (< (aref d j i) 0))
            (setq list (cons j list))))

      ;; calculate the sum of the durations of steps necessarily preceding the goal
      (dolist (index list)
        (setq sum (+ sum (aref d (+ index num-of-steps) index))))

      ; update the list to contain the step ids, not indeces
      (do ((l list (cdr l))) ((null l))
        (setf (car l) (conv-sym (car (rassoc (car l) assoc-list))
                                num-of-steps)))

      ;; put the slack into the slack list, and return if less than zero
      (setq slack-list (cons `(,goal ,(-* (aref d i 0) sum) ,list) slack-list))
      (if (<* (cadar slack-list) 0) (return-from slack nil)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Determine if the constraints in the time box are consistent with one another.                                          ;;
;; The process is divided into two stages:                                                                                 ;;
;;   1. Build a matrix of the constraints, and perform the transitive closure to find negative cycles.                     ;;
;;   2. Determine if any goals have negative slacks                                                                        ;;
;; If the plan is temporally consistent return true, otherwise nil                                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tb-time-consistent? (time-box step-list &optional (verbose nil))
  ;; If the consistency array has not been reset to zero
  ;; there is no need to check for consistency again
  (if (time-box-c-array time-box) (return-from tb-time-consistent? t))

  (let* ((num-of-steps (1+ (length step-list)))
         (n             (* 2 num-of-steps))
         (d             (make-array (list n n) :initial-element *infinity*))
         (var-list      (time-box-var-list time-box))
         (hash-table    (time-box-table time-box))
         (slack-list    '())
         (assoc-list    `((?st0 . 0) (?ft0 . ,n))))

    (setf (time-box-c-array time-box) nil)
    (setf (time-box-assoc-list time-box) nil)
    (setf (time-box-slack-list time-box) nil)

    (dotimes (i n)
      (setf (aref d i i) 0)      ;; all steps are equal to themselves
      (setf (aref d 0 i) 0))     ;; all steps must occur after S0.

    ;; build the association list between variable and array index
    (do ((lst step-list (cdr lst)) (i 1 (1+ i))) ((= i num-of-steps))
      (nconc assoc-list `((,(make-sym "?st" (car lst)) . ,i)
              (,(make-sym "?ft" (car lst)) . ,(+ num-of-steps i)))))

    ;; Fill matrix with the constraints. D[?x][?y] = z means ?x - ?y ≤ z
    (dolist (var var-list)
      (dolist (constraint (gethash var hash-table))
        (setf (aref d (conv-sym1 (first constraint) assoc-list) (conv-sym1 (second constraint) assoc-list))
              (third constraint))))

    ;; First interpret the time constraints as a directed graph and use the all-pairs shortest path algorithm to
    ;; determine if the STP is consistent. If any of the diagonal elements are negative, the STP is inconsistent.
    ;; If so return, return nil (false).
    (dotimes (k n)
      (dotimes (i n)
        (dotimes (j n)
          (setf (aref d i j)
            (min* (aref d i j) (+* (aref d i k) (aref d k j)))))
        (if (< (aref d i i) 0) (return-from tb-time-consistent? nil))))


    ;; If there are no negative cycles, then check to see if there is a linearization of the partial orders. This is
    ;; accomplished by calculating the slack in the plan for each goal. If any slack < 0 return nil (false)
    (setq slack-list  (slack d (time-box-goals time-box) num-of-steps assoc-list))
    (if (not slack-list) (return-from tb-time-consistent? nil))

    ;; Plan is temporally consistent.  Update the fields in the time-box
    ;; and return true
    (setf (time-box-c-array time-box) d)
    (setf (time-box-assoc-list time-box) assoc-list)
    (setf (time-box-slack-list time-box) (sort slack-list #'<* :key #'second))
    (if verbose (progn (print-d d num-of-steps assoc-list) (print (time-box-slack-list time-box)))))
  t)
```

Temporal Planning Examples

The following examples of temporal planning are actual transcripts from planning sessions using our temporal planner. There are several examples, the first three of which correspond to three example plans given in chapter V. The final two examples show the use of additional temporal constraints not considered in the previous examples. The examples have been divided into two separate problems. The first problem specifies a set of goals with minimal temporal constraints for a plan can be found by the planner. The second problem specifies the exact same goals with sufficiently restrictive temporal constraints so that no temporally consistent plan exists. The planner is initiated at the **LISP:** prompt and a transcript of the results follows immediately.

The transcript consists of the invocation of the planner with the problem instantiation, followed by the partially ordered plan that solves the problem. The plan is listed as a series of steps. With each step is a list of its preconditions, augmented by the id of the step that establishes each precondition. If the problem cannot be solved by the planner, the simple message "`goal is unattainable`" is returned. All of the examples are from the blocks world domain, which is specified by the `blocks-world-domain` function listed below.

Domain function:

```
(defun blocks-world-domain ()
  (reset-domain)

  ;; Define step for placing one block on another.
  (define (operator puton)
      :parameters (?X ?Y ?Z)
      :duration 10
      :precondition (:and (on ?X ?Z) (clear ?X) (clear ?Y) (:neq ?Y ?Z)
                    (:neq ?X ?Z) (:neq ?X ?Y) (:neq ?X Table))
      :effects
      ((:effect (:and (on ?X ?Y) (:not (on ?X ?Z))))
       (:effect (clear ?Z)          :when (:neq ?Z Table))
       (:effect (:not (clear ?Y)) :when (:neq ?Y Table)))))
```

Example 1: The first example demonstrates the Sussman anomaly. A diagram of the plan and its associated steps are shown in figure 5.3. The plan consists of three steps (not including the goal steps) which are completely ordered. This implies that the matrix algorithm (transitive closure) described in chapter V is sufficient to determine the temporal consistency of the plan. Since each step requires 10 time units, the goal can be established by time 50 (`ex-1a`), but cannot be established by time 25 (`ex-1b`).

```
(make-problem :name 'ex-1a
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block table)
             (clear B) (clear C) (clear table)
             (on C A) (on A table))
    :goal '((:name alpha :goal (:and (on a b) (on b c))
                         :time (:before-time 50))))
```

```
(make-problem :name 'ex-1b
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block table)
             (clear B) (clear C) (clear table)
             (on C A) (on A table))
    :goal '((:name alpha :goal (:and (on a b) (on b c))
                         :time (:before-time 25))))
```

```
LISP: (exec 'ex-1a)
The plan returned by UCPOP is:

Initial :((block a) (block b) (block c) (block table) (clear b) (clear c) (clear table)
          (on a table) (on b table) (on c a))
Step 5  : (puton-c-table-a)
           0  -> (on c a)
           0  -> (clear c)
           0  -> (clear table)
Step 3  : (puton-b-c-table)
           0  -> (on b table)
           0  -> (clear b)
           0  -> (clear c)
Step 4  : (puton-a-b-table)
           0  -> (on a table)
           5  -> (clear a)
           0  -> (clear b)
Step 1  : (goal alpha (begin))
           4  -> (on a b)
           3  -> (on b c)
Step 2  : (goal alpha (end))
           1  -> (on a b)
           1  -> (on b c)

LISP: (exec 'ex-1b)
Goal is unattainable.
```

Example 2: This example corresponds to the plan shown in figure 5.5. Since there are several potentially parallel steps, it is necessary to supplement the matrix calculation with the summation of the steps between the start of the plan and each goal in order to determine temporal consistency. The transitive closure algorithm ensures that each individual step meets the appropriate temporal constraints. Since several of the steps are not constrained with respect to one another, the transitive closure does not take their interaction into consideration. For example, the step to place block D on block E is not temporally constrained with respect to any other step and, therefore, it is necessary to ensure there is enough time to perform that step. This is accomplished by summing the durations of all the steps before goal `beta`. Both problems meet the temporal criteria enforced by the matrix algorithm. Problem `ex-2a` meets the summation criteria, while problem `ex-2b` does not.

```
(make-problem :name 'ex-2a
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block D) (block E) (block F) (block table)
             (clear A) (clear B) (clear C) (clear D) (clear E) (clear F) (clear table)
             (on A table) (on B table) (on C table) (on D table) (on E table) (on F table))
    :goal '((:name alpha :goal (:and (on b c) (on a f))
                         :time (:before-time 25))
           (:name beta :goal (:and (on a table) (on d e))
                       :time (:after alpha))))
```

```
(make-problem :name 'ex-2b
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block D) (block E) (block F) (block table)
             (clear A) (clear B) (clear C) (clear D) (clear E) (clear F) (clear table)
             (on A table) (on B table) (on C table) (on D table) (on E table) (on F table))
    :goal '((:name alpha :goal (:and (on b c) (on a f))
                         :time (:before-time 25))
            (:name beta :goal (:and (on a table) (on d e))
                        :time (:and (:after alpha) (:before-time 35)))))
```

```
LISP: (exec 'ex-2a)
The plan returned by UCPOP is:

Initial :((block a) (block b) (block c) (block d) (block e) (block f) (block table) (clear a) (clear b)
          (clear c) (clear d) (clear e) (clear f) (clear table) (on a table) (on b table) (on c table)
          (on d table) (on e table) (on f table))
Step 8  : (puton-b-c-table)
           0  -> (on b table)
           0  -> (clear b)
           0  -> (clear c)
Step 7  : (puton-a-f-table)
           0  -> (on a table)
           0  -> (clear a)
           0  -> (clear f)
Step 1  : (goal alpha (begin))
           8  -> (on b c)
           7  -> (on a f)
Step 2  : (goal alpha (end))
           1  -> (on b c)
           1  -> (on a f)
Step 6  : (puton-a-table-f)
           7  -> (on a f)
           0  -> (clear a)
           0  -> (clear table)
Step 5  : (puton-d-e-table)
           0  -> (on d table)
           0  -> (clear d)
           0  -> (clear e)
Step 3  : (goal beta (begin))
           6  -> (on a table)
           5  -> (on d e)
Step 4  : (goal beta (end))
           3  -> (on a table)
           3  -> (on d e)


LISP: (exec 'ex-2b)
Goal is unattainable.
```

Example 3: This example, taken from figure 5.7, incorporates a *floating step*. A floating step is a step whose interval of execution spans more than one goal. The step to place block E on block F may occur anytime before goal `delta`, since none of the other steps or goals threaten the effects it establishes. The algorithm of definition 5.1 is able to correctly determine temporal consistency, because we assume that the staring time of goals and the durations of goals have no temporal constraints. Problem `ex-3b` fails because there is no linearization of the plan that allows five steps to be performed in less than 45 time units, even though there is only one step that contributes directly to the goal `(on e f)`.

```
(make-problem :name 'ex-3a
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block D) (block E) (block F) (block table)
             (clear A) (clear B) (clear C) (clear D) (clear E) (clear F) (clear table)
             (on A table) (on B table) (on C table) (on D table) (on E table) (on F table))
   :goal '((:name alpha :goal (:and (on a b) (on c d))
                        :time (:before-time 25))
           (:name beta :goal (:and (on a table) (on c table)) :time (:after alpha))
           (:name delta :goal (on e f)
                        :time (:and (:after beta) (:before-time 50)))))

(make-problem :name 'ex-3b
    :domain #'blocks-world-domain
    :inits '((block A) (block B) (block C) (block D) (block E) (block F) (block table)
             (clear A) (clear B) (clear C) (clear D) (clear E) (clear F) (clear table)
             (on A table) (on B table) (on C table) (on D table) (on E table) (on F table))
    :goal '((:name alpha :goal (:and (on a b) (on c d))
                        :time (:before-time 25))
           (:name beta :goal (:and (on a table) (on c table)) :time (:after alpha))
           (:name delta :goal (on e f)
                        :time (:and (:after beta) (:before-time 45)))))

LISP: (exec 'ex-3a)
The plan returned by UCPOP is:

Initial :((block a) (block b) (block c) (block d) (block e) (block f) (block table)
          (clear a) (clear b) (clear c) (clear d) (clear e) (clear f) (clear table)
          (on a table) (on b table) (on c table) (on d table) (on e table) (on f table))
Step 11 : (puton-a-b-table)
           0  -> (on a table)
           0  -> (clear a)
           0  -> (clear b)
Step 9  : (puton-c-d-table)
           0  -> (on c table)
           0  -> (clear c)
           0  -> (clear d)
Step 1  : (goal alpha (begin))
           11 -> (on a b)
           9  -> (on c d)
Step 2  : (goal alpha (end))
           1  -> (on a b)
           1  -> (on c d)
Step 10 : (puton-a-table-b)
           11 -> (on a b)
           0  -> (clear a)
           0  -> (clear table)
Step 8  : (puton-c-table-d)
           9  -> (on c d)
           0  -> (clear c)
           0  -> (clear table)
Step 7  : (puton-e-f-table)
           0  -> (on e table)
           0  -> (clear e)
           0  -> (clear f)
Step 3  : (goal beta (begin))
           10 -> (on a table)
           0  -> (on b table)
           8  -> (on c table)
           0  -> (on d table)
Step 4  : (goal beta (end))
           3  -> (on a table)
           3  -> (on b table)
           3  -> (on c table)
           3  -> (on d table)
Step 5  : (goal delta (begin))
           7  -> (on e f)
Step 6  : (goal delta (end))
           5  -> (on e f)

LISP: (exec 'ex-3b)
Goal is unattainable.
```

APPENDIX B

IMPLEMENTATION AND EXAMPLES OF PLAN EXECUTION IN A DYNAMIC ENVIRONMENT

Dynamic Plan Execution Implementation Details

In this appendix we present portions of the implementation for dynamic plan execution and a simple example of its use. The LISP code shown below is part of the Exec module which receives user input and interacts with the core planner UCPOP (see figure 1.1). Dynamic execution of a plan has been integrated with the temporal aspects of our planner and Exec is partially responsible for parsing goal specifications and passing temporal constraints to the *time box*. The program sections are followed by an example exploring plan execution in a temporal environment.

The functions listed below comprise the heart of the execution module. The function `exec` builds a partially specified plan from the user's new goal and state information. This plan is then passed to UCPOP for completion. The functions `add-goals-to-plan` and `add-goal-time-constraints` are responsible for incorporating goals into the partially specified plan and adding a goal's temporal constraints to the time box, respectively. The next three functions carry out the execution of a step in a plan. A good candidate step is chosen by `find-first-step` and is executed by `apply-first-step`, for which `apply-effect` is an auxiliary function. Changes to the current plan, necessitated by unanticipated changes in the environment, are handled by the three functions `update-state`, `update-plan`, and `update-links`. Other functions needed to implement dynamic execution are not shown here.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function uses the UCPOP planning algorithm interactively to solve the given problem.  The steps in the      ;;
;; partially ordered plan (returned from UCPOP) are executed incrementally, with the possibility of new goals        ;;
;; being added.                                                                                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun exec (&optional (problem 'ex-1a))
  (let* ((user-state     nil)
         (user-goal      (problem-goal problem))
         (initial-state  (update-state (append user-state (problem-inits problem)) nil))
         (init-time      (get-internal-run-time))
         (good-plan      (tweak-plan
                          ;; initialize a new plan w/o the goal step
                          nil
                          :reason '(:init)
                          :steps nil
                          :links nil
                          :effects `(,(make-effect :id 0 :add initial-state))
                          :unsafe nil
                          :open nil
                          :ordering nil
                          :time-box (new-time-box)
                          :bindings (new-bindings)
                          :high-step 0))
         (plan           nil)
         (state          initial-state)
         (good-state     state)
         (time           0)
         (step           nil))

    ;; Initialize the domain
    (funcall (problem-domain problem))

    (setq plan (add-goals-to-plan good-plan user-goal))

    ;; Go into a loop until all the user goals have been satisfied
    (loop
      ;; Do the planning
      (multiple-value-bind (new-plan done? unused q-len av-branch)
          (UCPOP plan #'bestf-search #'rank-jps)
        (unless done?
          (progn  ;; Revert back to the plan without the unattainable goal
            (setq new-plan good-plan state good-state)
            (format msg-window "~%~%Goal is unattainable.")))

        (setq user-goal nil user-state nil)

        ;; Execute steps in the plan as long as there are no new user-goals or new user-state.
        ;; We need to replan if a new user goal or state arrives.
        (while (and (null user-goal) (null user-state))
          ;; Find, execute, and remove the first step
          (when (plan-steps new-plan)
            (multiple-value-bind (n-state n-new-plan step goal?) (apply-first-step state new-plan)
              (setq state n-state new-plan n-new-plan)
              (incf time (p-step-duration step))))

          ;; Get user input, both new state propositions and goals
          (multiple-value-setq (user-state user-goal) (get-goal)))

        ;; Check for a user quit indication
        (when (or (eq user-goal 'quit) (eq user-state 'quit)) (return-from exec 'good-bye))

        ;; We know the goals of this plan are attainable.  We know this state is consistent, so save them
        (setq good-plan new-plan)
        (setq good-state state)

        ;; Update the state and plan according to the new user input.
        (multiple-value-setq (state new-plan) (add-state-to-plan user-state state new-plan))

        ;; make the goals into steps, and add them to the plan
        (setq plan (add-goals-to-plan new-plan user-goal)))))
  ;; parting message
  'goodbye)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function adds the list of goals to the plan. It makes two steps for each goal, adds them to the plan, and  ;;
;; makes sure the time constraints are added to the temporal database (time box)                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun add-goals-to-plan (plan goals)
  (cond ((null goals)
         plan)
        (t      ;; recursive call (add-goals-to-plan new-plan (cdr goals))
         (add-goals-to-plan
          ;; Change the plan to reflect the new goals and changed steps
          (let* ((goal        (car goals))
                 (high-step   (1+ (plan-high-step plan)))
                 (goal-steps  (make-goal-steps goal high-step))
                 (g-begin-id  (p-step-id (car goal-steps)))
                 (g-end-id    (p-step-id (cadr goal-steps)))
                 (open-cond   (make-openc :condition (p-step-precond (car goal-steps)) :id g-begin-id))
                 (time-box    (add-goal-time-constraints plan goal-steps goal)))

            (tweak-plan
             plan
             ;; Incorporate the new effects of the goal, create an open conditions, make casual links from
             ;; the beginning goal step to the ending goal step, add partial ordering constraints from the
             ;; temporal constraints in goal.
             :reason `(:add-goal ,(p-step-action (car goal-steps)) ,open-cond)
             :effects (append (p-step-add (car goal-steps)) (plan-effects plan))
             :steps (list* (instantiate-step (car goal-steps) g-begin-id)
                           (instantiate-step (cadr goal-steps) g-end-id)
                           (plan-steps plan))
             :add-goal open-cond
             :links (append (make-links g-begin-id g-end-id (p-step-add (car goal-steps)))
                            (plan-links plan))
             :ordering (append (parse-time :goal-begin (car goal-steps) :goal-end (cadr goal-steps)
                                           :time (get-key goal ':time)  :steps (plan-steps plan))
                               (plan-ordering plan))
             :time-box time-box
             :high-step g-end-id))
          (cdr goals)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function parses a goal and gives the list of temporal constraints to the time box.                          ;;
;; The list of goals is also updated.                                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun add-goal-time-constraints (plan goal-steps goal)
  (let ((constraint-list (parse-time :goal-begin (car goal-steps)
                                     :goal-end (cadr goal-steps)
                                     :time (get-key goal ':time)
                                     :steps (plan-steps plan)))
        (new-time-box    (plan-time-box plan)))

    ;; iterate over the constraint list, adding one constraint to the time box at a time
    (dolist (constraint constraint-list new-time-box)
      (setq new-time-box (apply #'tb-add-constraint (cons new-time-box constraint))))

    ;; Update the goal list in the time-box
    (tb-add-goal new-time-box (p-step-id (car goal-steps)) (p-step-id (cadr goal-steps)))
    new-time-box))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function applies the first step (operator) to the state and returns the newly created state. We assume the ;;
;; preconditions of the step are satisfied by the state. We can do that because we only apply the FIRST step of     ;;
;; the plan, and the first step always has its preconditions satisfied by the state.                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun apply-first-step (state plan)
  (let* ((step     (find-first-step plan))
         (add-list (p-step-add step))
         (bind     (plan-bindings plan)))

    (unless (goal-step? step)
      (dolist (effects add-list)
        (let ((effect (effect-add effects))
              (precond (effect-precond effects)))

          ;; If there are forall effects, find all bindings that satisfy the pre-conditions. If those
          ;; preconditions are met in the state, apply the effects of the step. If there are no forall
          ;; conditions, simply test the preconditions
          (if (effect-forall effects)
              (multiple-value-bind (p new-bind)
                  (handle-precond `(:and ,precond) bind)
                (dolist (b (satisfy p state new-bind))
                  (when (precond-met? precond b state) (setq state (apply-effect effect state b)))))
            (when (precond-met? precond bind state) (setq state (apply-effect effect state bind))))))

      ;; Set the initial state for the plan, by augmenting the effects
      (setq plan
        (tweak-plan
         plan
         :effects (append (remove-step-effects 0 (plan-effects plan)) `(,(make-effect :id 0 :add state))))))

    ;; remove the time constraints associated with the step
    (tb-remove-constraints (plan-time-box plan) step)

    (values
     ;; return: new state, new plan, step, if step is a goal step
     state
     (remove-step (p-step-id step) (redirect-links step plan))
     step
     (goal-step? step))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Find a suitable step to execute.                                                                          ;;
;;  1. calculate all possible first steps in the plan.                                                       ;;
;;  2. If any is a goal step return it.                                                                      ;;
;;  3. find the step from the set of possible first steps that is in the list of steps preceding the goal with ;;
;;     the least slack.                                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun find-first-step (plan)
  (let* ((first-steps '())
         (steps       (step-list plan))
         (num-steps   (1+ (length steps)))
         (d           (time-box-c-array (plan-time-box plan)))
         (assoc-list  (time-box-assoc-list (plan-time-box plan)))
         (slack-list  (time-box-slack-list (plan-time-box plan))))

    ;; 1. Steps that necessarily don't have any other steps before them
    (dolist (i steps)
      (if (dolist (j steps t)
            (if (<* (aref d (conv-sym1 (make-sym "?st" j) assoc-list) (conv-sym1 (make-sym "?st" i) assoc-list)) 0)
                (return nil)))
          (setf first-steps (cons i first-steps))))

    ;; 2. Are any of those first steps a goal step?
    ;; 3. Return the step that occurs in the most successive slack lists
    ;; slack-list = ((goal slack (preceding_steps ...)) ...)
    (let ((tmp-list   first-steps)
          (final-list first-steps)
          (goal-list  (intersection first-steps slack-list :test #'(lambda (x y) (= x (caar y))))))
      (if goal-list
          (setq final-list goal-list)
        (while (and tmp-list slack-list)
          (setq tmp-list (intersection final-list (third (car slack-list))))
          (if tmp-list (setq final-list tmp-list))
          (setq slack-list (cdr slack-list))))

      (return-step (car final-list) (plan-steps plan)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function will apply the effects from a particular step to the current state.  The new state is returned.  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun apply-effect (effect state bind)
  (dolist (eff effect state)
      (setq state
            (cond ;; negated propositions are removed from the state
             ((eq (car eff) ':not)
              (remove-effect state (bind-variable (cadr eff) bind)))
             ;; non-negated propositions are added to the state
             (t
              (adjoin (bind-variable eff bind) state))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; This function integrates the user input with the current state.  If there are any conflicts, the user input is  ;;
;; assumed to have priority.  The plan is altered, as necessary, to reflect the new current state.  The function   ;;
;; returns the new state and plan.                                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun update-state (input state &optional (plan nil))
  ;; This is a recursive function.  For each new proposition in the state add (or remove)
  ;; the proposition and update the plan.
  (cond ((null input)
         (values state plan))
        ((null state)
         (update-state (cdr input) `(,(car input)) plan))
        ((eql ':not (caar input))
         ;; remove the negated proposition from the state
         (update-state (cdr input) (remove-1 (cadar input) state :test #'equal) (update-plan (car input) plan)))
        ((my-member (car input) state :test #'equal)
         ;; duplicate proposition, ignore it
         (update-state (cdr input) state plan))
        (t
         (update-state (cdr input) (cons (car input) state) (update-plan (car input) plan)))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Updates the plan given the propostion added to the current state. It checks all links to see if the proposition ;;
;; is created by a step. If it is, it deletes the link and the step producing it.                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun update-plan (prop plan &aux link)
  (cond ((null plan) nil)
        ((null prop) plan)

        ;; if prop is found in a link, delete link
        ((setf link (find prop (plan-links plan) :key #'link-condition))
         (update-plan
          prop
          (update-links
           link
           (remove-1 prop (plan-links plan) :test #'(lambda (prop link) (equal prop (link-condition link))))
           plan)))

        ;; if a direct negation of prop is found in a link to the initial state, delete it
        ((and (eq (link-id1 link) 0)
              (setf link (find prop (plan-links plan) :test #'negates :key #'link-condition)))
         (update-plan
          prop
          (update-links
           link
           (remove-1 prop (plan-links plan) :test #'(lambda (prop link) (negates prop (link-condition link))))
           plan)))

        ;; otherwise just return the plan
        (t
         plan)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Link is to be deleted from the plan, determine which links and steps are affected.  "links" are all the links   ;;
;; in the plan excluding "link".  Add open condition to compensate for the deleted link, unless otherwise          ;;
;; indicated by add-open-cond.                                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun update-links (link links plan &optional (add-open-cond t))
  (let* ((id1 (link-id1 link))
         ;; find if at least one other link whose producer is the same as the producer of "link"
         (l1 (find id1 links :test #'(lambda (id l) (eql id (link-id1 l))))))

    ;; if there is one such link that satisfies the above condition, don't remove the producer step,
    ;; but add open condition and replace link...
    (cond ((or l1 (eql id1 0))
           (tweak-plan
            plan
            :reason '(:jps-remove-link)
            :open (if add-open-cond
                      (cons (make-openc :condition (link-condition link) :id (link-id2 link))
                            (plan-open plan))
                    (plan-open plan))
            :links links))

          ;; Otherwise, remove the producer step and all the links to its preconditions
          (t
           (let ((list (mappend #'(lambda (x) (if (eql id1 (link-id2 x)) `(,x) nil)) links)))

             (dolist (x list)
               (setq plan (update-links l (delete-1 x links) plan nil)))

             (remove-step
              id1
              (tweak-plan plan
                 :reason '(:jps-add-open-cond)
                 :open (cons (make-openc :id (link-id2 link) :condition (link-condition link))
                             (plan-open plan)))))))))
```

Dynamic Execution Example

The example that follows is a simple demonstration of how our planner handles new goals and changes in the environment. The initial problem (`ex-4`) does not specify a goal. A goal `alpha` is given by the user at the prompt and a plan of 5 steps results. In the next iteration, after a step is executed, the user alters the state of the world and adds another goal. Notice that some of the steps from the previous plan are replaced by causal links to the current state (`Initial`) and additional steps are added to attain the new goal `beta`. The execution module continues to prompt the user for input until `quit` is entered at any prompt.

Example Problem Specification:

```
    (make-problem :name 'ex-4
                  :domain #'blocks-world-domain-jps
                  :inits '((block A) (block B) (block C) (block table) (clear B) (clear table)
                           (on C A) (on B C) (on A table) ))
```

```
LISP: (exec 'ex-4)

The plan returned by UCPOP is:

Initial :((block a) (block b) (block c) (block table) (clear b) (clear table) (on a table) (on b c) (on c a))
```

Here we enter a new goal to be satisified by the plan. Note the current state does not change.

```
Please enter a LIST of propositions to be added to the current state:
   <none>
Please enter a LIST of goals to be satisfied:
   ((:name alpha :goal (on a b)))

State after user input:
((block a) (block b) (block c) (block table) (clear b) (clear table) (on a table) (on b c) (on c a))

The plan returned by UCPOP is:

Initial : ((block a) (block b) (block c) (block table) (clear b) (clear table) (on a table) (on b c) (on c a))
Step 5  : (puton b table c)
          0  -> (on b c)
          0  -> (clear b)
          0  -> (clear table)
Step 4  : (puton c table a)
          0  -> (on c a)
          5  -> (clear c)
          0  -> (clear table)
Step 3  : (puton a b table)
          0  -> (on a table)
          4  -> (clear a)
          0  -> (clear b)
Step 1  : (goal alpha (begin))
          3  -> (on a b)
Step 2  : (goal alpha (end))
          1  -> (on a b)

After applying the step <5:puton b table c>, the current state is:
((block a) (block b) (block c) (block table) (clear b) (clear c) (clear table) (on a table) (on b table) (on c a))

The current time is: 10
```

A plan to satisfy goal alpha is found and the first step of that plan has been executed. Now we change the state of the world (but block b on block c) and add a new goal.

```
Please enter a LIST of propositions to be added to the current state:
   ((on b c) (:not (clear c)) (:not (on b table)))

Please enter a LIST of goals to be satisfied:
   ((:name beta :goal (on b c) :time (:after alpha)))

State after user input:
((block a) (block b) (block c) (block table) (clear b) (clear table) (on a table) (on b c) (on c a))

The plan returned by UCPOP is:

Initial : ((block a) (block b) (block c) (block table) (clear b) (clear table) (on a table) (on b c) (on c a))
Step 4  : (puton c table a)
          0  -> (on c a)
          0  -> (clear c)
          0  -> (clear table)
Step 3  : (puton a b table)
          0  -> (on a table)
          4  -> (clear a)
          0  -> (clear b)
Step 1  : (goal alpha (begin))
          3  -> (on a b)
Step 2  : (goal alpha (end))
          1  -> (on a b)
Step 6  : (goal beta (begin))
          0  -> (on b c)
Step 7  : (goal beta (end))
          6  -> (on b c)

After applying the step <4:puton c table a>, the current state is:
((block a) (block b) (block c) (block table) (clear a) (clear b) (clear table) (on a table) (on b c) (on c table))
```

Note the changes in the new plan from the old due to the changes in the state and new goal.

```
The current time is: 20
Please enter a LIST of propositions to be added to the current state:
   <none>
Please enter a LIST of goals to be satisfied:
   <none>

After applying the step <3:puton a b table>, the current state is:
((block a) (block b) (block c) (block table) (clear a) (clear table) (on a b) (on b c) (on c table))
```

No additional changes are made to goals or to the current state, so the plan runs to its completion.

```
The current time is: 30
Please enter a LIST of propositions to be added to the current state:
   <none>
Please enter a LIST of goals to be satisfied:
   <none>

**** The user goal <6:goal beta (begin)> is attained ****
**** The user goal <1:goal alpha (begin)> is attained ****
**** The user goal <2:goal alpha (end)> is attained ****
**** The user goal <7:goal beta (end)> is attained ****
Please enter a LIST of propositions to be added to the current state:
   quit

The final state consists of the following propositions:
((block a) (block b) (block c) (block table) (clear a) (clear table) (on a b) (on b c) (on c table))
```

BIBLIOGRAPHY

[1] Aho, A.V., J.E. Hopcroft, and J.D. Ullman. 1974. *The design and analysis of computer algorithms.* Reading, MA: Addison-Wesley.

[2] Allen, James. 1984. Towards a general theory about action and time. *Artificial Intelligence* 23: 123-154.

[3] Allen, James, and Johannes Koomen. 1983. Planning using a temporal model. In *Proceedings of IJCAI 1983*: by the IJCAI. Palo Alto, CA: Morgan Kaufmann, 741-747.

[4] Coffman, E.G., ed. 1976. *Computer and job-shop scheduling theory.* New York: John Wiley & Sons.

[5] Dantzig, G.B. 1962. *Linear programming and extensions.* Princeton, NJ: Princeton University Press.

[6] Dean, Thomas L., and Drew V. McDermott. 1987. Temporal database management. *Artificial Intelligence* 32: 1-55.

[7] Dechter, Rina Itay Meiri, and Judea Pearl. 1991. Temporal constraint networks. *Artificial Intelligence* 49: 61-95.

[8] Fikes, R., and N. Nilsson. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2: 189-205.

[9] Garey, M.R., and D.S. Johnson. 1979. *Computers and intractability.* New York: W.H. Freeman and Company.

[10] Garey, M.R., and D.S. Johnson. 1977. Two processor scheduling with start times and deadlines. *SIAM Journal of Computation* 6: 416-426.

[11] Georgeff, Michael P. 1987. Planning. *Annual Review of Computer Science* 2: 359-400.

[12] Goodman, Louis J., and Ralph N. Love, eds. 1980. *Project planning and management.* New York: Pergamon Press.

[13] Horz, Alexander. 1993. On the relation of classical and temporal planning. In *Foundations of automatic planning: Papers from the 1993 Spring Symposium,* by the AAAI. Menlo Park, CA: AAAI Press.

[14] Lawler, E.L. 1973. Optimal sequencing of a single machine subject to precedence constraints. *Management Science* 19: 544-546.

[15] McAllester, David, and D. Rosenblitt. 1991. Systematic nonlinear planning. In *Proceedings AAAI-91,* by the AAAI. Menlo Park, CA: AAAI Press, 634-639.

[16] McDermott, Drew. 1982. A temporal logic for reasoning about processes. *Cognitive Science* 6: 101-155.

[17] Penberthy, Scott. J., and Daniel S. Weld. 1993. Temporal planning with constraints. In *Foundations of automatic planning: Papers from the 1993 Spring Symposium,* by the AAAI. Menlo Park, CA: AAAI Press.

[18] Penberthy, Scott. J., and Daniel S. Weld. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92,* 103-114.

[19] Suchman, Lucy A. 1987. *Plans and situated action.* Cambridge: Cambridge University Press.

[20] Tate, Austin, James Hendler, and Mark Drummond. 1990. A review of AI planning techniques. In *Readings In Planning*, ed. James Allen, James Hendler, and Austin Tate, 26-49. San Mateo, CA: Morgan Kaufmann.

[21] Vere, Steven A. 1983. Planning in time: windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 5: 246-267.

[22] Weld, Daniel S. 1993. An introduction to partial-order planning. Seattle, WA: University of Washington. Photocopied.