Temporal Planning Under Uncertainty in a BDI frame work

Dissertation Submitted in the partial fulfillment of the

requirements for the award of degree of

Master of Technology

in

University of Hyderabad

By

Rajesh Kumar Tonukunoori

Department of Computers and Information Sciences
University of Hyderabad
Hyderabad

April 2010

Certificate

This is to certify that the project work entitled "**Temporal Planning Under Uncertainty in a BDI frame work**" being submitted to University of Hyderabad by **Rajesh Kumar Tonukunoori** (Reg.No.08MCMI01), in partial fulfillment for the award of the degree of Master of Technology, Department of Computers and Information Sciences, is a bonafide work carried out by him under my supervision.

Dr.Vineet P Nair
Project Supervisor,
Department of CIS,
University of Hyderabad.

Prof Arun Agarwal,
Head of Department,
Department of CIS,
University of Hyderabad

Prof T.Amaranath,
Dean,
School of MCIS,
University of Hyderabad.

Temporal Planning Under Uncertainty in a BDI framework

Rajesh Kumar Tonukunoori

Dissertation Submitted in the partial fulfillment of the requirements

for the award of degree of Master of Technology

Department of Computers and Information Sciences

April 2010

Abstract

Planning is a key area in Aritficial Intelligence (AI) which is typically defined (in the classical sense) by three components: a) A Formal description of the start state b) A formal description of the intended goals and c) A Formal description of the actions that may be performed. In this thesis we deal with Temporal Planning under Uncertainty which extends Temporal Planning by defining uncertainty in terms of duration. We show how this kind of planning can be incorporated in BDI like Agent architectures wherein currently planning is done in the classical sense. To this end we used a Phocus-HC algorithm to successfully plan under uncertainty.

Dedicated to,

My Parents

Acknowledgements

I would like to express my sincere gratitude to **Dr Vineet P Nair**, my project supervisor, for valuable suggestions and keen personal interest throughout the progress of my course of research. He encouraged, supported, corrected and guided me during the project. The project has been a learning and growing experience for me. Working under his my guidance I learned many things.

I am thankful to my lab mates for their advice and discussions with me. I would also like to thank all my teachers.

I am extremely greatful to our Head of the Department, **Prof. Arun Agarwal**, for providing excellent computing facilities and a nice atmosphere for doing my project. I convey my heartfelt thanks to AI Lab staff for their help in completing the project work successfully.

I would like to take this opportunity to thank my friends especially to Mr. Rajendar who have been morale boosters for me, encouraged me to take up this course and supported me throughout this course with their love and affection. Last but not the least my parents and sisters to whom I owe this work. I wouldnt have reached this stage without their support and encouragement.

T.Rajesh Kumar

Contents

List of Figures

List of Tables

Chapter 1

Introduction

`Planning` is the task of coming up with a sequence of actions that will achieve a goal. Planning is a very common thing in our daily life. For example, to open a door we need to go near the door and then open the door, i.e. we are following two actions simultaneously. Planning is a key ability for intelligent systems, increasing their autonomy and flexibility through the construction of sequences of actions to achieve their goals. It has been an area of research in artificial intelligence for over three decades. Planning techniques have been applied in a variety of tasks including robotics, process planning, web-based information gathering, autonomous agents and spacecraft mission control.

1.1 Search *vs* Planning

Problem Solving agents uses standard search algorithms like depth-first, breadth-first, A* etc to solve large real world problems. The most obvious difficulty is that problem solving agents can be overwhelmed by irrelevant actions. For example, consider the task of buying a copy of text book *Artificial Intelligence:A Modern Approach* from an online book seller. Suppose there is one buying action for each 10 digit ISBN (International Standard for Book Number) number, for a total of 10 billion actions. The search algorithm would have to examine the outcome states of all 10 billion actions to find one that satisfies the goal, which is to own a copy of ISBN 0137903952. A sensible planning agent, on the other hand, should be able to work back from an explicit goal descritpion such as Have(ISBN0137903952) and generate the action BUY(ISBN0137903952) directly. To do this, the agent simply needs the

general knowlede that Buy(x) result in Have(x). Given this knowledge and the goal, the planner can decide in a single unification step that Buy(ISBN0137903952) is the right action.

1.2 STRIPS

STRIPS stand for Stanford Reasearch Institute for Problem Solver. Some properties for representing planning problems using STRIPS are listed below;

- Only positive literals in states: $Poor \wedge Unknown$
- Closed World Assumption: Unlimited literals are false.
- Effect $P \wedge \neg Q$ means add P and delete Q.
- Only ground literals in goals: $Rich \wedge Famous$
- Goals are conjunctions: $Rich \wedge Famous$
- Effects are conjunctions.
- No support for equality.
- No support for types.

In this thesis we make use of the STRIPS language though it is well known that the language ADL can also be used for representing temporal planning problems.

1.3 Planning Problem

Planning problem in the classical sense is typically defined by three components. Here the classical sense means that the planning problem is fully observable, deterministic, finite, static and discrete. We outline below the three components that are needed for defining a classical planning problem.

1. A Formal description of the states.
2. A Formal description of the intended goals.
3. A Formal description of the actions that may be performed.

A better description of states, actions and goals should make it possible for planning algorithms [10] to take advantage of the logical structure of the problem. The key is to use a STRIPS (Stanford Research Institute Problem Solver) like language. Now we see the representaions using STRIPS.

1.3.1 Representation of states

Planners decompose the world into logical conditions and represent a state as a conjunction of positive literals. For example At(plane1,Melbourne) $\wedge$ At(plane2, Sydney) which is read as plane1 is at Melbourne and plane2 at Sydney.

1.3.2 Representation of goals

A goal is partially specified state and is represented as conjunction of positive ground literals. For example At(plane2,Tahiti).

1.3.3 Representation of actions

An action is specified in terms of the precondition which should hold before its execution and the effects that ensue when it is executed. For example

$$Action(Fly(p, from, to)) : -$$
$$PRECOND : At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$$
$$EFFECT : -At(p, from) \wedge At(p, to)$$

An action schema consists for three parts:

1. The action name and parameter list. For example go(from, to)

2. The *precondition* is a conjunction of function free positive literals stating what must be true in a state before the action can be executed. All variables in the precondition must also appear in actions parameter list.

3. The *effect* is a conjunction of function free literals describing how the state changes when the action is executed.

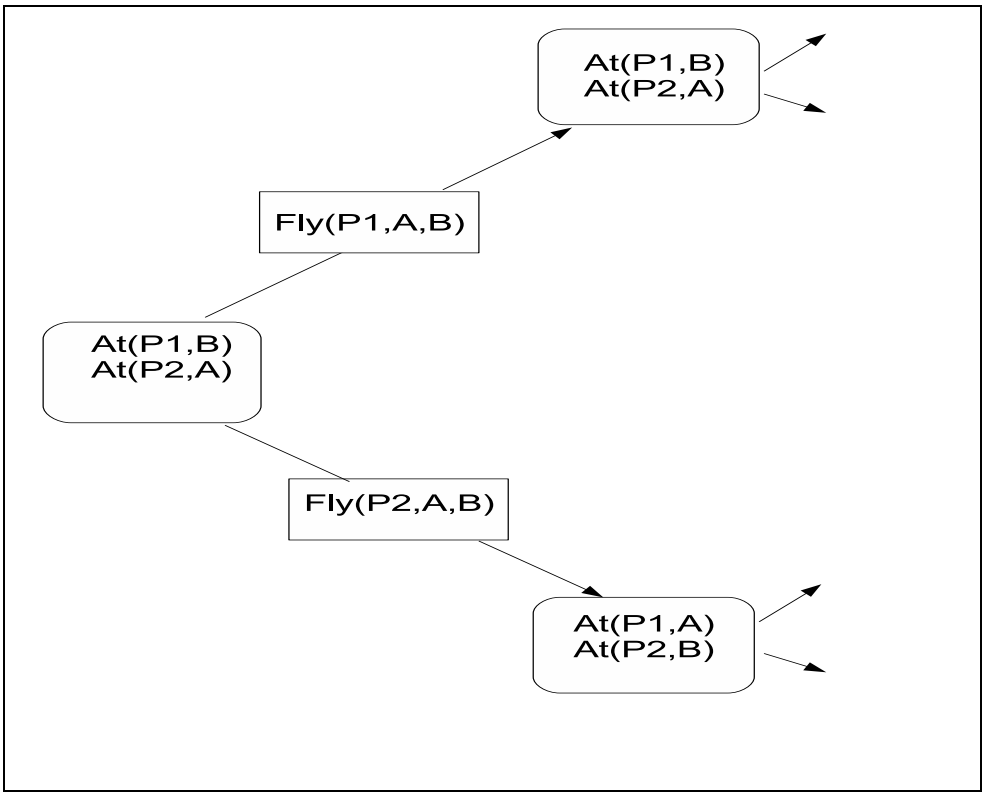

Figure 1.1: Progression planning

1.4 Planning Algorithms

We have different types of planning algorithms like *progression planning, regression planning, partial order planning* and *graph plan.* Progression and regression planning comes under total order planning.

1.4.1 Progression planning (Forward state-space search)

Planning with forward state-space search is similar to search techniques like Breadth first or Depth first search. It is also known as progression planning, because it moves in the forward direction. We start with the problem's initial state, considering sequences of actions untill we find a sequence that reaches a goal state. Figure 1.1 shows how progression works.

1.4.2 Regression planning (Backward state-space search)

Backward state-space search is also called as regression planning. It starts with the goal and ends at initial state. It starts at goal and searches which action leads to

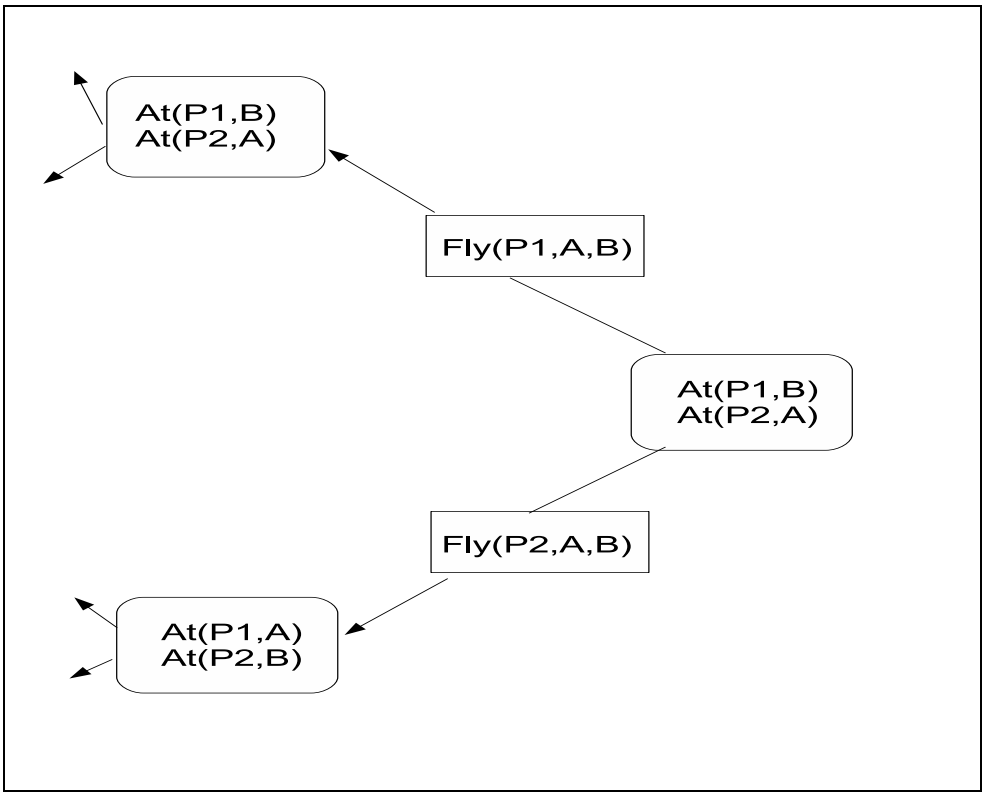

Figure 1.2: Regression planning

this goal condition and thereafter applies the same action to reach the new state. In this way it proceeds backward until it reaches to the initial state. Figure 1.2 shows how regression works. The main advantage of regression is that it considers only relevant actions. Regression is better than progression because it generates fewer states than progression.

1.4.3 Partial Order Planning

In total order planning, we are sure about which action to be executed next among the available actions. In the case of partial order panning [14] one cannot give guarantee of which action to execute next i.e., any planning algorithm that can place two actions into a plan without specifying which comes first is called a *Partial-order planner*. Progression and regression plannings are particular forms of total order planning. Total order planning is also called linear planning and partial order planning is also called non-linear planning. Consider the simple problem of putting

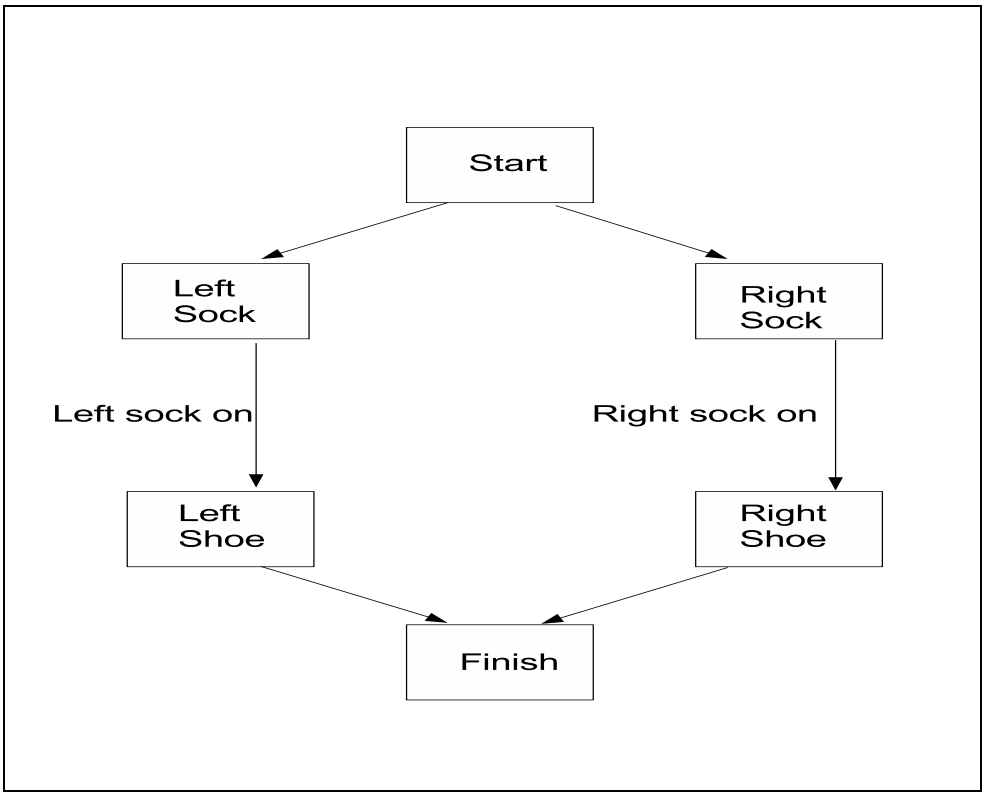

Figure 1.3: Partial order planning

on a pair of shoes, which can be seen as a planning problem as follows:

$$Goal(RghtShoeOn \wedge LeftShoeOn)$$

$Init()$

$Action(RightShoe, PRECOND : RightSockOn, EFFECT : RightShoeOn)$

$Action(RightSock, EFFECT : RightSockOn)$

$Action(LeftShoe, PRECOND : LeftSockOn, EFFECT : LeftShoeOn)$

$Action(LeftSock, EFFECT : LeftSockOn)$

Figure 1.3 shows partial order planning for the above planning problem. The partial order solution corresponds to six possible total order plans and each of these is called a linearization of the partial order plan. The six possible linearization are

1. *Start, RightSock, LeftSock, RightShoe, LeftShoe, Finish*
2. *Start, RightSock, LeftSock, LeftShoe, RightShoe, Finish*
3. *Start, LeftSock, RightSock, RightShoe, LeftShoe, Finish*

4. *Start, LeftSock, RightSock, LeftShoe, RightShoe, Finish*

5. *Start, LeftSock, LeftShoe, RightSock, RightShoe, Finish*

6. *Start, RightSock, RightShoe, LeftSock, LeftShoe, Finish*

From the above example, we can observe that we can wear either Right sock or Left Sock first. There are no restrictions that are telling to wear a specific sock first. We call this type of planning as partial order planning. We can observe this type of partial ordering in many situations in the above mentioned shoes example.

1.4.4 GraphPlan

GraphPlan [1] alternates between two phases (1) graph expansion (2) solution extraction. The graph expansion phase extends a planning graph (which is explained in the next Chapter) until it has achieved a necessary condition for plan existence. The solution extraction phase then performs a backward-chaining search for an actual solution: if no solution is found, the cycle repeats.

The planning graph contains alternating levels of proposition nodes (corresponding to ground literals) and action nodes. The zeroth level consists solely of the propositions that are true in the initial state of the planning problem. Nodes in an action level correspond to action instances, there is one such node for each action instance whose preconditions are present (and are mutually consistent) at the previous proposition level. Directed edges connect proposition nodes to subsequent action nodes whose preconditions reference those propositions. Similarly, directed edges connect action nodes to subsequent propositions made true by the actions effects. Persistent actions function like frame axioms: each proposition at a level is linked to its persistent action at the next level and the action connects to the same proposition at the next level. GraphPlan defines a binary mutual exclusion relation between nodes in the same level. For example, two action instances are mutex if one action deletes a precondition or effect of another or the actions have preconditions that are mutually exclusive at the previous level. Two propositions are mutex if all ways of achieving the propositions (that is actions at the previous level) are pair wise mutex.

1.4.5 Temporal Panning

Temporal planning [9] is an extension of classical planning, where in we have durations and their execution can overlap in time. More precisely the duration d(a) > 0 for each action a. It has a set of legal actions. Temporal plans thus becomes sequences A0, A1, A2 etc of legal sets of actions in which all actions in each set A_i start their execution at time t_i. The end or completion time of an action a in A_i is $t_i + d(a)$ where d(a) is the duration of a. The initial state is s_0, s_{i+1} is function of s_i at t_i and the set of actions A in the plan that complete exactly at time t+1. In temporal planning, the user may be interested in improving temporal quality of plan.

Many temporal planners exist in classical planning like SAPA [7], TLPlan, TGP [16]. SAPA is a domain-independent heuristic forward chaining planner that can handle durative actions, metric resource constraints and deadline goals. In this, heuristics are derived from the relaxed temporal planning graph structure which is a generalization of planning graphs to temporal domains. Another temporal planner proposed in [15] is a domain independent and works in highly dynamic environments with time constraints. It uses `any time planning` algorithm [18]. The two main principles of any time computation are (1) provide a valid (approximate) solution at any time (2) monotonically improve the solution quality with respect to the deliberation time. This planner starts computing a plan for the problem, in case no unexpected situation is detected, the planner continues improving the current plan or searching for better alternative. Otherwise, the current plan is rejected, and the algorithm starts computing a new plan for the new situation.

A theoretical basis is given in [5] for performing time limited planning within Belief Desire and Intention (BDI) agents. In our proposed method we are considering only durations and overlap of actions. For example a group of students want to go from Gachibowli to Bangalore. The two common options are (1) rent a car and drive from Gachibowli to Bangalore directly or (2) use public transport to reach the destination. When there is no direct service to reach destination using public transport, it is compulsory to break the journey and use another service which takes to destination. Each vehicle takes some time to reach a particular place. User want to reach the destination by imposing time constraint, i.e, the total time to reach the destination is not more than threshold time (such as to be in Bangalore within three

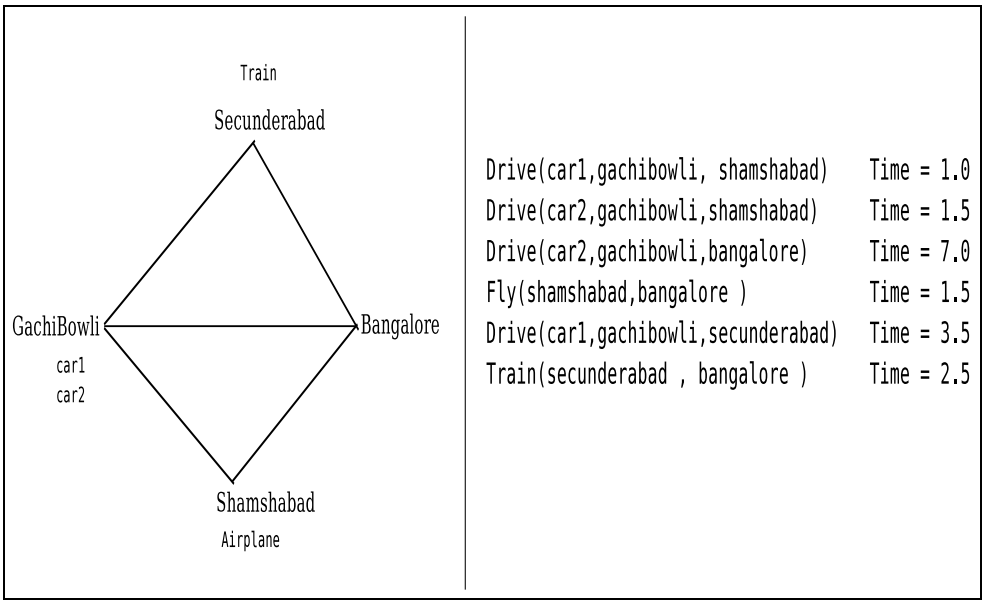

Figure 1.4: Travel problem

hours). The problem description and actions with durations for above scenario is given in Figure 1.4.

The simple example above shows the importance of a users objective and constraint on a given planning problem. In order to find plans that are good with respect to time, we need to track time of (sub)goals. The planner has to generate a plan that is optimal and at the same time should satisfy the constraints imposed by he user.

1.5 Uncertainty

Uncertainty means unsure about something i.e., we are not sure about the environment. For example, if we consider the travel problem as given in the previous section, one of the sources of uncertainty could be that of traffic. We are not sure of the traffic conditions on the road and that means because of traffic we get change in action duration which in turn will effect our generated plan. For doing temporal planning under uncertainty there are 3 methods available in the Literature;

- Replanning
- Conformant planning
- Contingency planning.

Here we are not solving temporal planning under uncertainty but we are just planning under uncertainty. For temporal planning under uncertainty we need one seed plan, so we first go for generating seed plan using relaxed planning graph and then only we consider uncertainty.

The thesis is organized as follows. In chapter 2 we give an overview of the BDI architecture. Chapter 3 gives details on how to incorporate temporal planning in a BDI system. Chapter 4 talks about the implementation aspects of temporal planning. In Chapter 5 we show how to combine temporal planning and uncertainty. Finally we conclude with Chapter 6 wherein some ideas related to future work is outlined.

Chapter 2

Incorporating Temporal planning in BDI

In this chapter we give a brief account of a particular agent architecture called BDI and then we show how to incorporate temporal planning in such an environment.

2.1 BDI Programming Languages

The BDI (Belief-Desire-Intention) architecture, based on Bratmans theory of practical reasoning and Dennets theory of intentional system, is a popular and well studied model for intelligent agents situated in complex and dynamic environments. Beliefs represent an agents knowledge of its environment. Desires represents an agents goals or its desired outcomes and an intention is a chosen desire that the agent commits to. The purpose of the BDI architecture is to make agents capable of behaving in a more human-like manner. Several BDI architectures as well as programming and specification languages such as PRS [13], 3APL [4], AgentSpeak(L) [12], JACK [3] and CAN [17] etc. can be found in the Literature.

BDI agent-oriented programming languages are built around an explicit representation of beliefs, desires, and intentions. A BDI architecture explains how these components are represented, updated, and processed to determine the agents actions. An agent consists of belief base, a set of plans, goals and intention. The belief base encodes the agents knowledge about world. The plan library contains predefined plans to achieve the goals. A BDI system responds to an event, commits to handle a pending event, and thereafter selects a plan from the plan library and

places the body in the intention for execution. The execution of the plan body may produce new goals to be achieved. If at some point the plan fails, then an alternative plan is found and its plan body is placed in intention base. Figure 2.1 roughly outlines a typical BDI architecture.

2.2 AgentSpeak(L)

Agent Speak(L) [12] is a programming language designed to capture the major features of a BDI system. It is based on restricted first-order logic language with events and actions. The beliefs, desires and intentions of the agent are not explicitly represented as modal formulas. It consists of set of belief base (or facts in the logic programming sense) and set of plans. Plans are context-sensitive, event-invoked, that allows to decompose the goal into sub goals as well as the execution of actions. The adoption of programs to satisfy desire can be viewed as intentions. There are two types of goals, `achievement goals` and `test goals`. Achievement goals are prefixed with the ! operator which states that the agent want to achieve a state of the world. Test goal is prefixed with the ? operator. Specifies that agent wants to test whether the associated predicate is true belief or not. Plan have a specific form as e: $\varphi \leftarrow$ P where e is a triggering event, φ is a context of the plan which should believed to hold and P is plan body which is placed in intention base to execute when event is called. Belief base, Events (Goals), Plan Library and Intentions are modelled as in Figure 2.1 and the explanation is as follows;

When a goal (event) comes in, selection function S_E selects a goal. The selected goal is unified with the plans in the Plan Library. The plans obtained after unification are called Relevant Plans. These relevant plans are then unified with the belief in the Belief Base to check which plans satisfies the context. The plans which satisfies the context are called Applicable Plans. Selection function S_o selects the applicable plan and pushes the plans either on the existing intention (if it is a sub goal) or creates a new intention in the set of intentions (if the goal is external). Intentions are partially instantiated plans i.e. plans where some of the variables have been instantiated. S_I is a selection function which selects an plan from the top of the Intentions stack for execution. If the plan have sub goals then it is placed in the Events queue and process the sub goal. The applicable plan of the sub goal

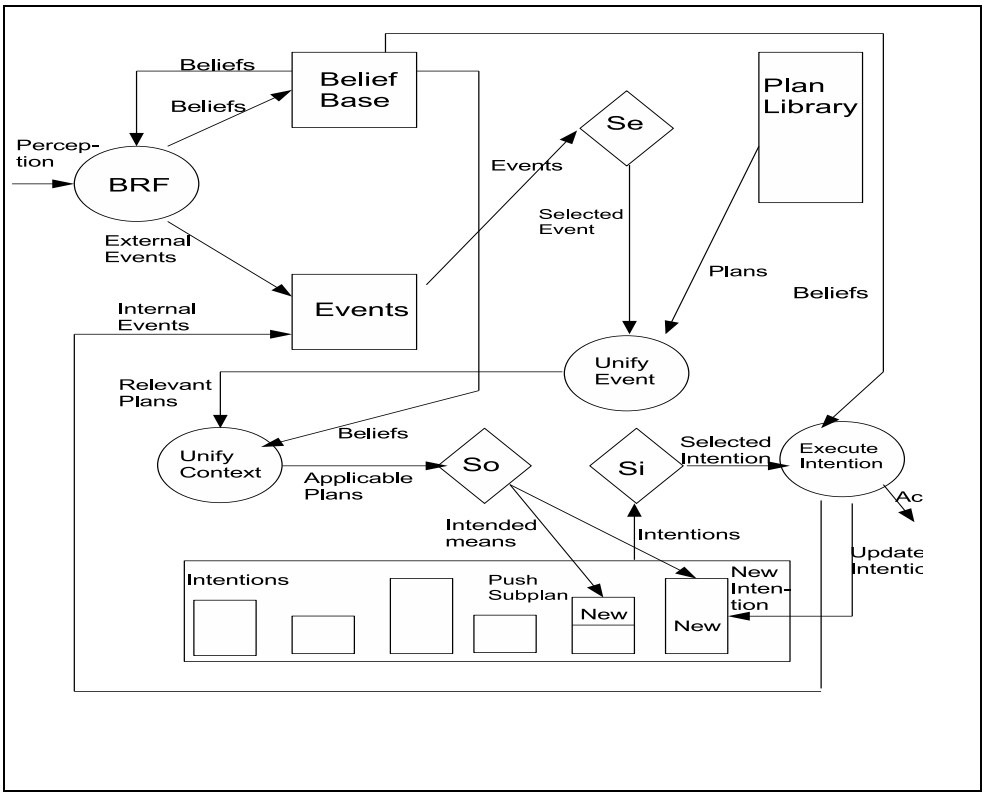

Figure 2.1: AgentSpeak(L)-Architecture

is placed in the intention. If goal is not a sub goal then it performs action. If the plans are not found in the plan library then goal fails.

When Agent speak (L) is compared to other agent programming languages like 3APL [4] we can see that it does not deal with plan failure. Agent speak (XL) [2], an extension of the Agent speak (L) Architecture, contains precise mechanisms to handle plan failures. In Agent Speak(XL), Agent speak(L) interpreter is extended to have an efficient intention selection function (S_I) using DTC(Design-T0-Criteria Scheduler) technique. We will use Agent Speak(XL) in our temporal planning under uncertainty.

One of the drawbacks of both Agent speak (L) and Agent Speak(XL) is that the programmer has to encode all the possible plans in a given domain. So the number of plans are more in the plan library. In the case of agent speak (PL) [11] graph plan is used to extract context of a plan. Here first the problem is converted to the STRIPS like domain and then they apply Graph Plan for propagation of prepositions and finally they extract context of the plan. The numbers of plans are reduced in the plan library when compared to Agent Speak (L).

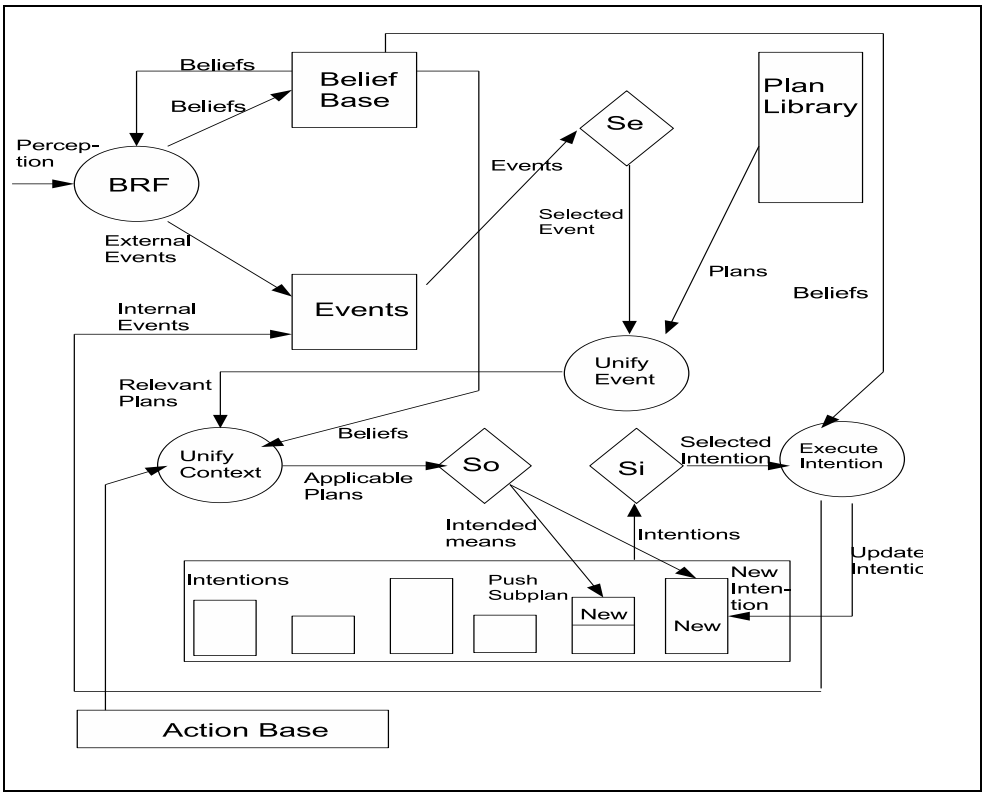

Figure 2.2: AgentSpeak(XL)-Architecture

2.2.1 Proposed model

In our proposed model we follow the notations of the Agent speak (XL) language. In this Agent speak(XL), we introduced action base to be passed along with belief base and plan libry. Figure 2.2 shows the extended Agent speak(XL) architecture. Figure 2.3, 2.4, 2.6 represents action base, belief base and plan library respectively of the travel problem mentioned earlier. Figure 2.3 contains the actions with the duration time to complete. For example `Drive(car 1, gachibowli, secunderabad) Time = 3.5` means that it takes 3.5 hours to reach secunderabad from gachibowli

1.	Drive(car1, Gachibowli,Secunderabad) Time = 210
2.	Drive(car1, Gachibowli,Shamshabad) Time = 60
3.	Drive(car2, Gachibowli,Shamshabad) Time = 90
4.	Drive(car2, Gachibowli,Bangalore) Time = 420
5.	Fly(Shamshabad,Bangalore) Time = 90
6.	Train(Secunderabad,Bangalore) Time = 150

Figure 2.3: Action Base

1.	at(Gachibowli)
2.	at(car1,Gachibowli)
3.	at(car2,Gachibowli)

Figure 2.4: Belief Base

at (Bangalore) Time = 210

Figure 2.5: Goal

1	(Drive(X,Y,Z):at(Y),at(X,Y)⟵+at(Z),at(Y),-at(X,Y),+at(X,Z))
2	(Train(X,Y) :at(X)⟵+at(Y), - at(X))
3	(Fly(X,Y) :at(X)⟵+at(Y), - at(X))

Figure 2.6: Plan Library

by road. Figure 2.4 contains belief of the world which holds at that point of time. For instance `at(car1, gachibowli)` means that car1 is at gachibowli. If some actions are performed then the belief base will be changed accordingly. Figure 2.6 specifies what Happens when an action is performed and which beliefs should hold to perform this action. For example

$$Train(X,Y) : at(X) \longleftarrow +at(Y), -at(X)$$

is one of the plan in the plan library. By substituting variables with ground terms we get

$$Train(secunderabad, bangalore) : at(secunderabad) \longleftarrow$$
$$at(bangalore), -at(secunderabad).$$

To execute this plan it has to satisfy the context, i.e., at(secunderabad) which in turn should be in the belief base. Only then the plan body is executed.

2.3 Incorporating Temporal Planning into BDI

In this section we show how to incorporate Temporal Planning in a BDI Architecture. In our model, as mentioned in the previous section, we include an action base to the existing BDI architecture. Figure 2.7 shows modified version of BDI architecture. In traditional BDI architecture programmer has to encode all possible plans for a given domain. When a plan is not found in the plan library for a given goal then it fails even though there are set of actions which can achieve the goal. In our model we are trying to build a plan when plans are not present in the plan library using **Relaxed**

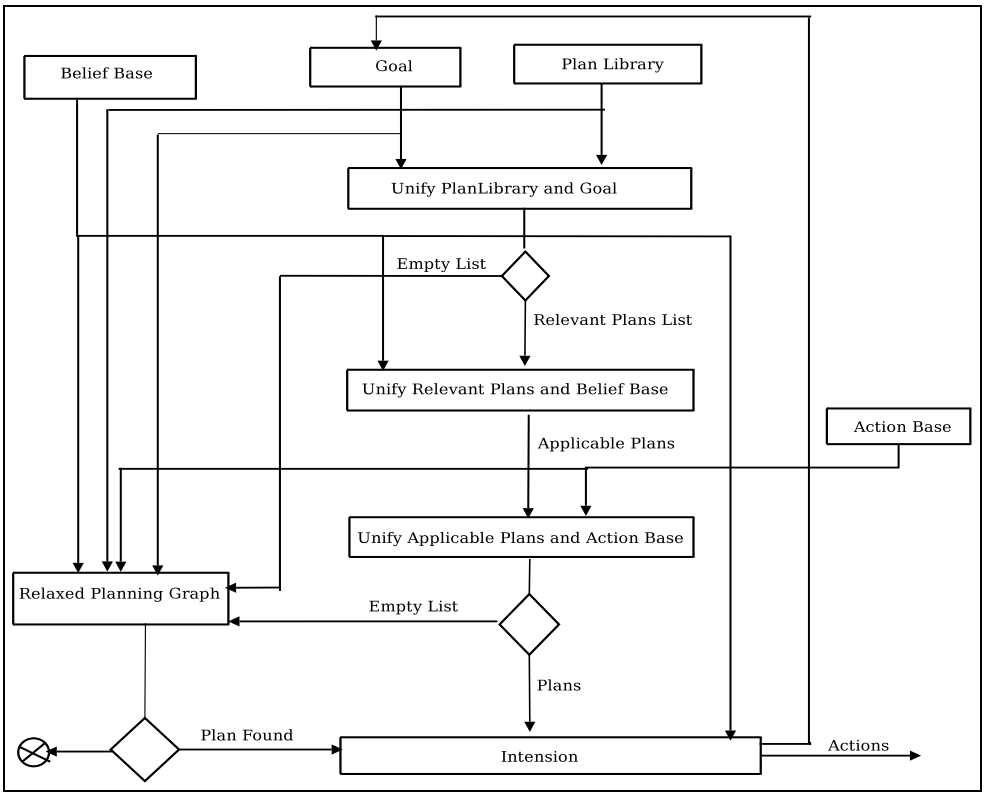

Figure 2.7: Modified BDI architecture

Planning Graph approach. In our approach belief base is similar to the traditional BDI belief base wherein set of beliefs are involved. Action base have actions with time duration because each action takes some time to complete. The proposed algorithm for Temporal Planning in BDI architecture is given in Algorithm1. Our model works as follows : when a goal is given, firstly it matches with the plans in the plan library by matching predicate name and number of parameters of goal irrespective of the parameters (ground terms or variables) it have. Theses plans are unified with the goal for Relevant Plans. If the relevant plans are empty list then we try to find a plan by using Relaxed Planning Graph(RPG). Relaxed Planning graph returns a plan with an optimal time which may be less than the user specified time or not. The plan time is checked with the user time (goal time) and if its time is less than the goal time then we put that plan in intention for execution. Otherwise return null (plan not found). Once the relevant plans are found using RPG check whether these plans (relevant plans) satisfies the context (preconditions) or not. Relevant plans may have variables in the precondition. Substitute these variables with the ground terms by all possible values of the beliefs. For Example, consider

that the belief base contains the following beliefs

$$at(gachibowli), at(car1, gachibowli), at(car2, gachibowli), at(car3, gachibowli).$$

Suppose the plan is

$$at(Bangalore) : at(X), at(Z, X) \longleftarrow Drive(Z, X, P), Fly(P, Bangalore).$$

The variable X takes {X/gachibowli} and Z takes {Z/car1,car2,car3}. When we substitute ground terms it generates extra plans as follows;

- at (Bangalore): at (gachibowli), at (car1, gachibowli) ⟵ Drive (car1, gachibowli,P), Fly(P,Bangalore)
- at (Bangalore): at (gachibowli), at (car2, gachibowli) ⟵ Drive (car2, gachibowli,P), Fly(P,Bangalore)
- at (Bangalore): at (gachibowli), at (car3, gachibowli) ⟵ Drive (car3, gachibowli,P), Fly(P,Bangalore)

The plans given above satisfies the context and these plans are called Applicable Plans. In the applicable plans there may be variables in the plan body. Substitute these variables with the possible ground terms by unifying action and applicable plans. Suppose the action base looks like (Drive(car1, gachibowli, secunderabad) 210), (Drive(car2, gachibowli, Bangalore) 600), (Drive(car2, gachibowli, shamshabad) 90), (Fly(shamshabad, Bangalore) 90), (Drive(car1, gachibowli, shamshabad) 150). Considering the above set of actions and plans as given before we can see that plans have variable P in their plan body. Hence possible values are P/shamshabad. Substitute P values in the applicable plans. Therefore we get the following plans after substitution.

- at (Bangalore) : at(gachibowli), at(car1,gachibowli)⟵ Drive (car1, gachibowli, shamshabad), Fly(shamshabad, Bangalore)
- at (Bangalore) : at(gachibowli), at(car2,gachibowli)⟵ Drive (car2, gachibowli, shamshabad), Fly(shamshabad, Bangalore)
- at (Bangalore) : at(gachibowli), at(car3,gachibowli)⟵ Drive (car3, gachibowli, shamshabad), Fly(shamshabad, Bangalore)

Algorithm 1: Temporal Planning

Require: B = Belief Base (Current State of Environment or World)
PL = Plan Library (Predefined Plans)
G = Goal State including time
AD = Actions with Durations
Ensure: Plan found or no solution
$relevantPlans \leftarrow$ **unify**(PL, G)
if *relevantPlans is empty list* **then**
$plan =$ **RelaxedPlanningGraph**(PD, B, AD, G)
if $planTime <= goalTime$ **then**
intention$(plan)$
else
plan not found
end if
else
$applicablePlans \leftarrow$ **unify**$(relevantPlans, B)$
$extendedApplicablePlans \leftarrow$ **substituteGroundTerms**$(applicablePlans, AD)$
$plans \leftarrow$ **calculate**$(extendedApplicablePlans)$
Sort the list based on the time of a plan and remove plans which have time greater than the goal time
if *list is empty* **then**
$plan =$ **Relaxed Planning Graph**(PD, B, AD, G)
if $planTime <= goalTime$ **then**
intention$(plan)$
else
plan not found
end if
else
intention$(list)$
end if
end if

The plans obtained after the substitution of variables in the applicable plans are called Extended Applicable Plans. Extended applicable plans includes unnecessary plans by substitution of variables. So, remove unnecessary plans by checking unnecessary actions in the plan. After removing unnecessary plans calculate the time for all the plans. This can be done by selecting an item in the plan body which may be either sub goal or action. If the item is a sub goal then we need to find plans and process the plan body until item is an action. If item is an action then search in action base for time. Calculate the time for all the plans and sort them. The plans which has less than or equal to the goal time, that plans are put in the intention for execution. For the above example the following plans are obtained after removing unnecessary plans.

- Plan1 is at(Bangalore):at (gachibowli), at (car2,gachibowli)⟵ Drive (car2, gachibowli, shamshabad), Fly(shamshabad, Bangalore)
- Plan2 is at(Bangalore):at (gachibowli), at (car1,gachibowli)⟵ Drive (car1, gachibowli, shamshabad), Fly(shamshabad, Bangalore)

The time taken for plan1 and plan2 are 3 hours and 4 hours respectively. If the user goal time is > 4 then plan1 and plan2 is selected. If goal time is < 4 then only one plan i.e., plan1 is selected. If user specified time is < 3 then no plan is selected in that case it will find a plan using relaxed planning graph.

2.4 Extending BDI with Relaxed Planning Graph

In this section, we use RPG to build a plan when plans are not found in the plan library of a BDI system. We do this without converting BDI into STRIPS notation as is the common practice in Agent speak (PL) (most recent BDI architectute).

Graph Plan [1] builds a layered structure called the planning graph and then searches this structure backwards for a plan. It finds and records mutual exclusion relationships by propagating them through the planning graph. Graph Plan technique is applicable only for STRIPS-like domains. Relaxed Planning Graph(RPG) is similar to graph Plan approach, but it ignores delete effects of actions. Most of the planners like SAPA [7] [6]and Metric-FF uses relaxed planning graph approach to find the heuristics. Agent speak(PL) [11] uses Graph Plan approach to generate

context of plan in BDI architecture by converting from BDI to STRIPS notation. In our approach we are using Relaxed Planning Graph approach to build a plan when plans are not found in the plan library, without converting BDI into STRIPS notation.

In our approach Relaxed Planning Graph requires belief base, action base, plan library and goal to build a plan. Belief base is required to find the initial state (target). Action base is used to find which set of actions can lead to goal. Plan library is required to find the next (sub) goal. Algorithm 2 takes all the input and set the initial state (target) list by comparing the goal name and number of parameters with the belief in the belief base. We will generate all optimal paths from goal to the target in the target List and returns the list of plans. For example goal = at(Bangalore) then target List contains at(gachibowli). If the belief base in Figure 2.3 contains at(ranchi) then target List has at(Bangalore), at(ranchi). Algorithm 3 and Algorithm 4, uses functions like `select Actions, Unify Possible Actions, get Action, getNewGoal, get Time, minimum Cost` and `minCostPath`. **Select**

Algorithm 2: Relaxed Planning Graph

Require: Belief Base, Action Base, Plan Library and goal

$targetList = getTarget(goal)$

for all i in $targetList$ **do**

$target = targetList[i]$

$plan = emptylist$

$plan = rpg(goal)$

$finalPlanList.append(plan)$

end for

return finalPlanList

Actions function takes one parameter that is goal and searches in the action base for actions that have parameter name equal to the goal parameter. For example goal =at(Bangalore). When this is a parameter for the function then it searches the action base as shown in Figure 2.2 and returns the following actions as output :

1. Drive(car2, gachibowli, Bangalore)
2. Fly(shamshabad, Bangalore)

Algorithm 3: rpg

Require: *goal*

Ensure: *plan* or *empty list*

```
actionList = selectActions(goal)
for all i in actionList do
    planList.append(unifyPossibleActions(actionList[i]))
end for
for all i in planList do
    action = getAction(planList[i], goal)
    if action ≠ empty list then
        applicableActionList.append(action)
    end if
end for
for all i in applicableActionList do
    temp = emptylist
    t = minCostPath(getNewGoal(applicableActionList[i], goal), temp1)
    if t == 0 then
        t = True
    else if t == False then
        continue
    else
        minValue.append(float(t) + gettime(applicableActionList[i]))
        temp1.append(applicableActionList[i])
        temp.append(temp1)
    end if
end for
if len(minValue) > 1 then
    min = float(minimumCost(minValue, index))
    finalPath.append(temp[index[0]])
else if len(minValue) == 1 then
    min = float(minValue[0])
    finalPath.append(temp[0])
else
    finalPath.append(emptylist)
end if
```

3. Train(secunderabad, Bangalore)

unify Possible Actions This function takes action as parameter. This function selects a plan from the plan library by matching action name and number of parameters. After matching it unifies this plan and action, and returns the plan. For example the action

$$Fly(shamshabad, Bangalore)$$

returns the following plan

$$Fly(shamshabad, bangalore) : at(shamshabad) \longleftarrow$$
$$+at(Bangalore), -at(shamshabad)$$

get Action This function takes a goal and a plan as parameters and checks whether this plan have goal in the plan body as a positive literal. If goal is present in plan body as a positive literal then we return the plan otherwise null (empty list).

get New Goal This a function to generate new goal which takes a goal and a plan as inputs. Each plan has precondition and plan body. The given goal is matched with the positive literals in the plan body. If match not found then there is no new goal, otherwise search for a negated literal that has a goal name and same number of parameters. If negated literal found then take that literal as a new goal. For example let goal = at(Bangalore) and plan =

$$Fly(shamshabad, Bangalore) : at(shamshabad) \longleftarrow$$
$$+at(Bangalore), -at(shamshabad) then new goal is at(shamshabad)$$

get Time takes a plan as parameter and then plan name is checked in the action base. If a match is found then the corresponding time for that action is returned and in the case of 'no match' null is returned. For example if

$$(Fly(shamshabad, bangalore) : at(shamshabad) \longleftarrow$$
$$+at(Bangalore), -at(shamshabad)$$

is considered as the parameter then the plan name

$$Fly(shamshabad, Bangalore)$$

is searched in the action base of Figure 2.2 and gets a return value of 1.5 hours.

Algorithm 4: minCostPath

Require: *goal* and *path*
Ensure: *value* or $False$
  if $target == goal$ **then**
    return 0
  else if $g ==$ *empty list* **then**
    return []
  else
    $actionList =$ `selectActions`$(goal)$
    for all i in $actionList$ **do**
      $planList.append($`unifyPossibleActions`$(actionList[i]))$
    end for
    for all i in $planList$ **do**
      $action =$ `getAction`$(planList[i], goal)$
      if $action \neq$ *empty list* **then**
        $applicableActionList.append(action)$
      end if
    end for
    for all i in $applicableActionList$ **do**
      $temp = emptylist$
      $t =$ **minCostPath**(`getNewGoal`$(applicableActionList[i], goal), temp1)$
      if $t == 0$ **then**
        $t = True$
      else if $(t ==$ *empty list*$)or(t == False)$ **then**
        $returnFalse$
      else
        $minValue.append(float(t) +$ `gettime`$(applicableActionList[i]))$
        $temp1.append(applicableActionList[i])$
        $temp.append(temp1)$
      end if
    end for
    if $len(minValue) > 1$ **then**
      $min = float($`minimumCost`$(minValue, index))$
      $path.append(temp[index[0]])$
    else if $len(minValue) == 1$ **then**
      $min = float(minValue[0])$
      $path.append(temp[0])$
    else
      $returnFalse$
    end if
  end if

minimum Cost This function returns minimum cost value of the (sub) plan and index value in the list by taking list of plan cost and variable to store index value.

min Cost Path is a recursive function that returns the (sub) plan path cost if it finds one otherwise return False. Algorithm 4 is a min Cost Path function. It takes goal and variable to store plan path. Termination condition of this function is whether goal is equal to the initial state(target). Then return 0 because it reaches the initial state or goal is empty list which means there is no path to reach the initial state then return False. Otherswise there may be other states from which initial state can be reachable. We will apply this function recursively until the target is reached or dead state (no states which achieve this state).

Till now we explained about the functions used in the relaxed planning graph. We will show how a plan can be build using RPG. Firstly it sets the initial state (target) because RPG does not know where to stop. `Get Target` function is used to find the initial state. Algorithm 3 is used to generate plan which internally uses Algorithm 4. When a goal is given to Algorithm 3, it will select all the actions from which goal is achievable. Unify all the actions with the plans in plan library. Select plans which have goal in the plan body as a positive literal. These plans are called action Plans (actions unifies with the plans), using the action (plan) in action Plans, goal can be achievable. We are not selecting an action (plan) from action Plan depends on the cost but we find the cost for all the actions in action Plans to reach the initial state. To do this, select the action Plan and find the path from current state to the initial state. Precondition of the action plan having same predicate name (goal name) and same number of parameters of goal become the new goal. We apply Algorithm 4 giving new goal and variable to store the complete plan path from new goal to the initial state. `min Cost Plan` function returns the cost of the plan and path if it finds otherwise it returns False (plan path not found). The minCostPath returns the complete path and time to reach initial state from action. We will add this complete path to the action and store in variable (possiblePath). Select one plan from possible Path which is having the lowest cost among them. In this way Algorithm 3 returns the optimal path if it finds one otherwise returns nothing (plan cannot be obtained) to the Relaxed Planning Graph (i.e., Algorithm 2). Algorithm 2 finds the optimal path for all the initial states(targets) and returns

paths to the intention otherwise empty list. If the initial states are more than one then user has to take decision which path to consider for execution.

Chapter 3

Implementation of temporal planning

We implemented **Temporal Planning in BDI** using **Python** programming language. From a logic point of view, belief base and action base are written in first order logic form. We converted first order logic statements into a list. As mentioned earlier plans in the plan library have a specific structure like e: $\varphi \leftarrow$ P. We convert belief base, action base and plan library into lists. Table 5.1 shows the conversion of beliefs into list. To perform this action we wrote parse Belief function which removes the parentheses and convert it to list. Table 5.2 shows the conversions of actions into list. For this we wrote parse Action function which removes the parentheses and convert actions into to list. Table 5.3 shows how the plans are converted into list. To perform this action we implemented parse Plan function which removes the parentheses and converts it to list. In the case of Plan conversion a different structure is adopted. We will place plan name in one list, plan precondition in one list and plan body in one list.

Beliefs	**List Representaion**
(at(gachibowli))	[‘at’,‘gachibowli’]
(at(car1,gachibowli))	[‘at’,‘car1’,‘gachibowli’]
(at(car2,gachibowli))	[‘at’,‘car2’,‘gachibowli’]

Table 3.1: Conversion of Belief Base to List

Actions	List Representation
(Drive(car1,gachibowli, secunderabad)210)	[‘Drive’,‘car1’,‘gachibowli’, ‘secunderabad’,‘210’]
(Drive(car2,gachibowli, bangalore)420)	[‘Drive’,‘car2’,‘gachibowli’, ‘bangalore’,‘420’]
(Drive(car2,gachibowli, shamshabad)90)	[‘Drive’,‘car2’,‘gachibowli’, ‘shamshabad’,‘90’]
(Train(secunderabad,bangalore)150)	[‘Train’,‘secunderabad’,‘bangalore’,‘150’]
(Fly(shamshabad,Bangalore)90)	[‘Fly’,‘shamshabad’,‘bangalore’,‘90’]
(Drive(car1,gachibowli, shamshabad)60)	[‘Drive’,‘car1’,‘gachibowli’, ‘shamshabad’,‘60’]

Table 3.2: Conversion of Action Base to List

1. Plan Name
2. Plan Precondition
3. Plan Body

For example

[[‘Train’,‘X’,‘Y’],[[‘at’,‘X’]],[[‘+’,‘at’,‘Y’],[‘-’,‘at’,‘X’]]]

Plan Name : [‘Train’,‘X’,‘Y’]
Plan Precondition : [[‘at’,‘X’]]
Plan Body : [[‘+’,‘at’,‘Y’],[‘-’,‘at’,‘X’]]
Plan precondition may consists of one or more list of beliefs but in the example given above plan precondition have only one item. It should be noted that we use prolog notion i.e., small case letters for constants and capital letters for variables.

3.1 An Illustrative Example (Travel Problem)

As an example consider a simple travel agent. The agents task is to find a route to reach a place within user specified time constraints. When a request arrives, the

Plans	List Representation
(Drive(X,Y,Z)): at(Y), at(X,Y) +at(Z),-at(Y),-at(X,Y),+at(X,Z))	[[‘Drive’,‘X’,‘Y’,‘Z’],[[‘at’,‘Y’],[‘at’,‘X’,‘Y’]]] [[‘+’, ‘at’,‘Z’],[‘-’,‘at’,‘Y’], [‘-’,‘at’,‘X’,‘Y’],[‘+’,‘at’,‘X’,‘Z’]]]
(Train(X,Y)):at(X) +at(Y), -at(X))	[[‘Train’,‘X’,‘Y’],[[‘at’,‘X’]], [[‘+’,‘at’,‘Y’],[‘-’,‘at’,‘X’]]]
(Fly(X,Y)) : at(X) +at(Y), -at(X))	[[‘Fly’,‘X’,‘Y’],[[‘at’,‘X’]], [[‘+’,‘at’,‘Y’],[‘-’,‘at’,‘X’]]]

Table 3.3: Conversion of Plan Library to List

agent will try to find the route. If the route is not possible for user specified time then it will suggest a possible route of minimum time. For the Travel Planning Problem Table 5.1, Table 5.2 and Table 5.3 are belief base, action base, and plan library respectively. If the user gives the goal as (at(Bangalore)3) that means user wants to reach Bangalore within 3 hours. We break user goal into goal and goal time. For the above goal we break it as at(Bangalore) and goal time as 3. When this goal is given to our system it firstly converts the first order logic beliefs and actions into list representation. There is no relevant plans because the goal does not have matching plans in the plan library. It will build a plan using relaxed planning graph. Algorithm 2 is relaxed planning graph and we will show how this algorithm works.

- It will set target (initial state) that is to find a plan from goal to initial state.
- This target is passed to Algorithm 3 to find the plan. We are not directly calling this algorithm because there may be more than one target. If there are multiple targets then we will call this algorithm for each target to get a plan.
- When Algorithm 3 is called

  1. Select all the actions which have goal name in its parameters. For our example it gives following actions.

     (a) (Drive(car2, gachibowli, Bangalore))

     (b) (Train(secunderabad, bangalore))

     (c) (Fly(shamshabad, Bangalore))

2. We are selecting actions blindly without thinking whether it will reach the goal or not. For example if (Drive(car2, bangalore, gachibowli)) action is present in the action base then it will also be picked because we are checking action name and number of parameters and if we perform this action it will not reach the goal (at(Bangalore)). For this reason actions are unified with the corresponding plan in the plan library. After unification of above actions following plans are obtained.
   (a) Drive(car2,gachibowli, bangalore): at(gachibowli), at(car2, gachibowli) $\leftarrow$ + at(bangalore), − at(gachibowli),− at(car2,gachibowli),+ at(car2,bangalore)
   (b) Train(secunderabad, Bangalore): at(secunderabad) $\leftarrow$ + at(bangalore), − at(secunderabad)
   (c) Fly(shamshabad, bangalore): at(shamshabad) $\leftarrow$ + at(bangalore), − at(shamshabad)
3. After unification, we will select the relevant actions which reach the goal by checking whether the goal is present in the plan body as positive literal or not. If plan have goal in the plan body then we pick that action as a relevant action. All the above actions (plans) satisfies goal element as a positive literal, so we pick all the actions(plans) as relevant actions.
4. For the above actions we are not selecting one action which ever has less time but we process all the actions(plans) because there may be possibility of getting optimal plan by selecting an action which have more time initially.
   (a) Select first action(plan) and process for plan to reach the target (initial state) that is
   Drive(car2,gachibowli, bangalore): at(gachibowli), at(car2, gachibowli) $\leftarrow$ + at(bangalore),− at(gachibowli),− at(car2,gachibowli),+ at(car2,bangalore)
      i. Apply Algorithm 4 This algorithm takes two parameters (1) new goal (2) variable to store path. New goal for above action (plan) is gachibowli. Algorithm 4 takes gachibowli and a variable as parameters.

ii. MinCost Agorithm returns 0

(b) The time for this plan is calculated as sum of minCost value and action time. Therefore total time of the plan is 0+getTime(Drive(car2, gachibowli, bangalore))=0+7=7. Store this path and time in a variable (possible plans).

(c) Select second action (plan) that is Train(secunderabad, Bangalore): at(secunderabad) ← + at(bangalore), − at(secunderabad)

i. Apply Algorithm 4 taking secunderabad as a new goal and variable as parameters. minCost Algorithm is recursive function, this algorithm terminates if goal is equal to target or null. Otherwise it selects the actions to reach the new goal and process all the actions until it reach the target or null.

– There is one action which can reach new goal (secunderabad) that is Drive(car1, gachibowli, secundrabad).

– Apply miniCost algorithm taking new goal as gachibowli and variable which returns 0

– To reach secunderabad from gachibowli using this action takes 0+getTime (Drive (car1, gachibowli, secunderabad))= 0+3.5 = 3.5 minCost function returns 3.5 and variable containing this action (plan).

(d) Time taken to reach bangalore from gachibowli is sum of the minCost value and action time = 3.5 + getTime(Train(secunderabad, bangalore)) = 3.5 + 2.5 = 6.0 This action (plan) is added to path return by minCost function. This plan time and plan path is added to variable (possible plans).

(e) Select third action(plan) that is Fly(shamshabad, bangalore): at(shamshabad) ← + at(bangalore), − at(shamshabad)

– New goal is not equal to target or null. So, select all the actions which reach this new goal. There are two actions possible to reach the new goal. (1) Drive(car2, gachibowli, shamshabad)(2) Drive(car1, gachibowli, shamshabad)

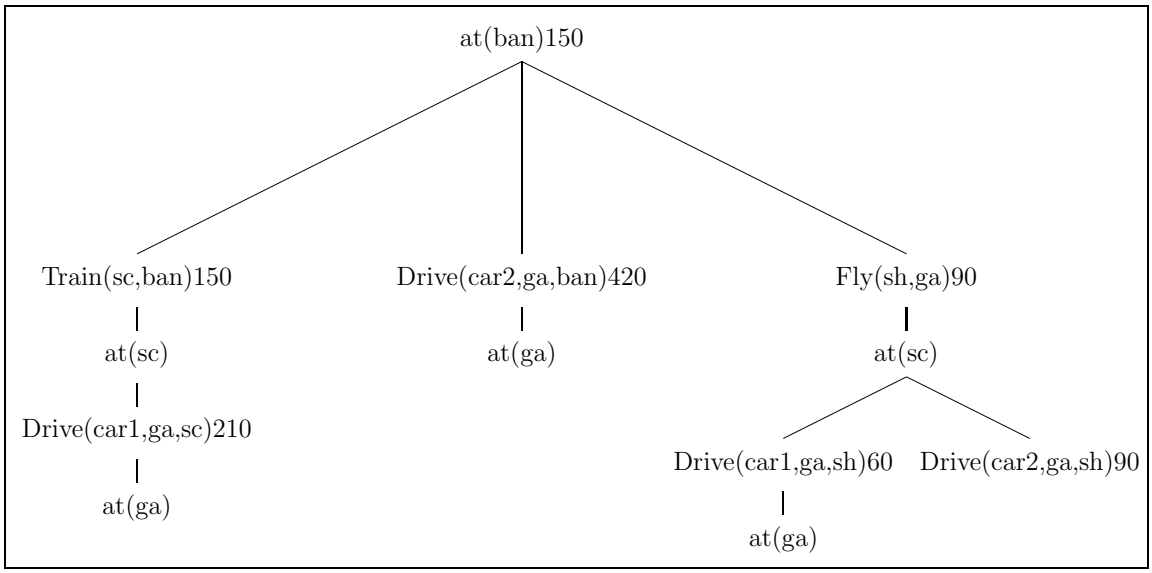

Figure 3.1: Complete Solution of the problem

– Process each action until it reach the target or null. After processing all the actions return the plan which ever has less time to reach the target if it finds. minCost function returns 1.0 and Drive(car1, gachibowli, shamshabad) assigned to variable.

(f) Total time taken for this plan to reach Bangalore from gachibowli is 1.0+get Time(Fly(shamshabad, Bangalore) = 1.0+1.5=2.5. The plan time and path is added to the variable (possible plans).

5. Select the plan which have minimum plan time from variable(possible plans). It returns Drive(car1, gachibowli, shamshabad), Fly(shamshabad, Bangalore).

- Algorithm 3 returns path to the relaxed planning graph function. The path obtained is only for single target(initial state). If there are more number of initial states then we will find the optimal path for all initial states from goal. The user has to take decision which path to take from the set of plan paths.

In this way we build an optimal plan path using relaxed planning graph. Figure 5.1 shows complete solution for the problem. This plan is placed in the intention for execution. The results related to using RPG on the Travel problem as mentioned earlier is shown in the screenshots given below.

Here figure-3.2 shows screen shot for belief base, plan library, action base and goal of our travel problem.

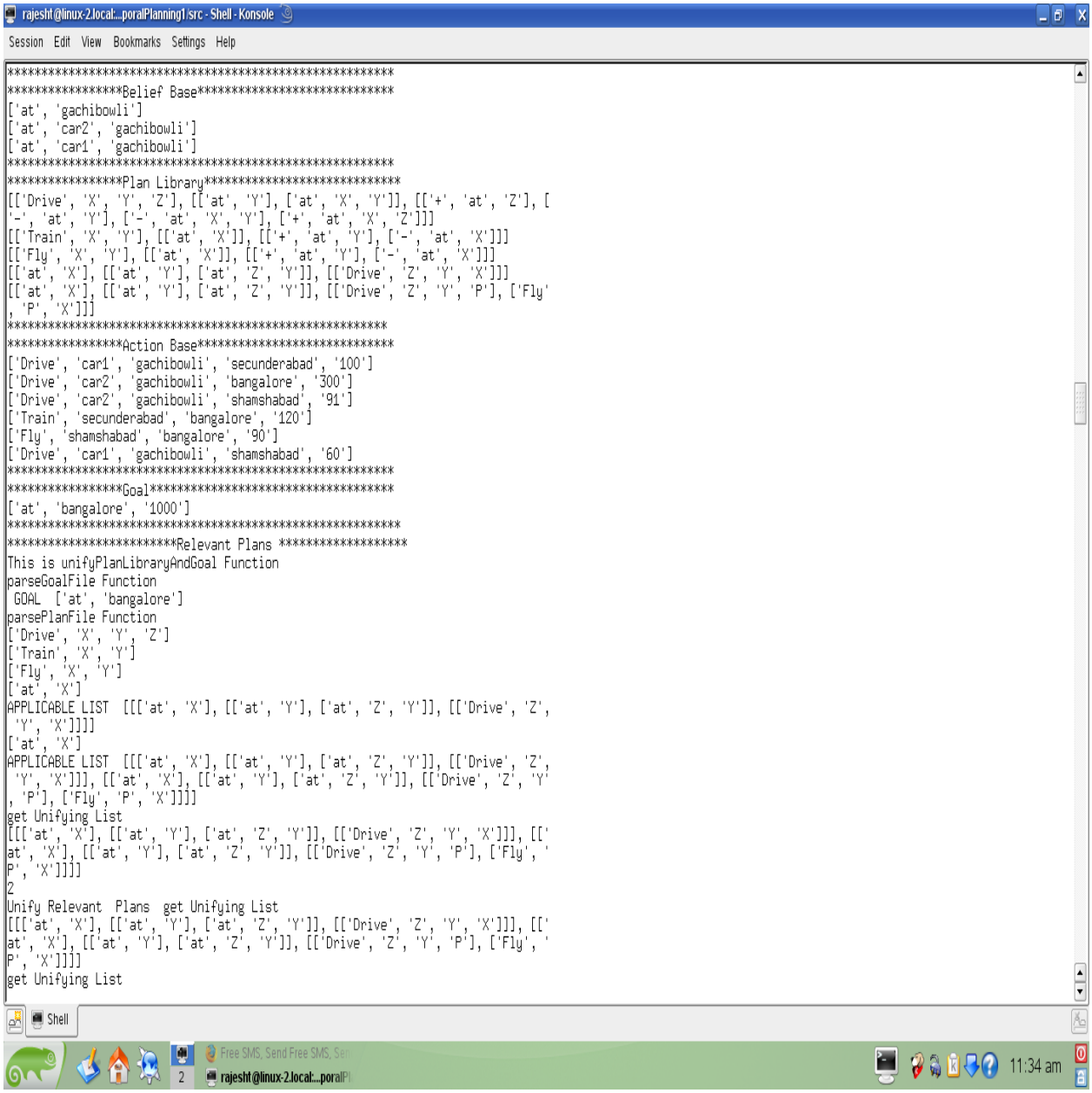

Figure 3.2: Screen 1

Here figure-3.3 shows screen shot for change in belief base after execution of our algorithm.

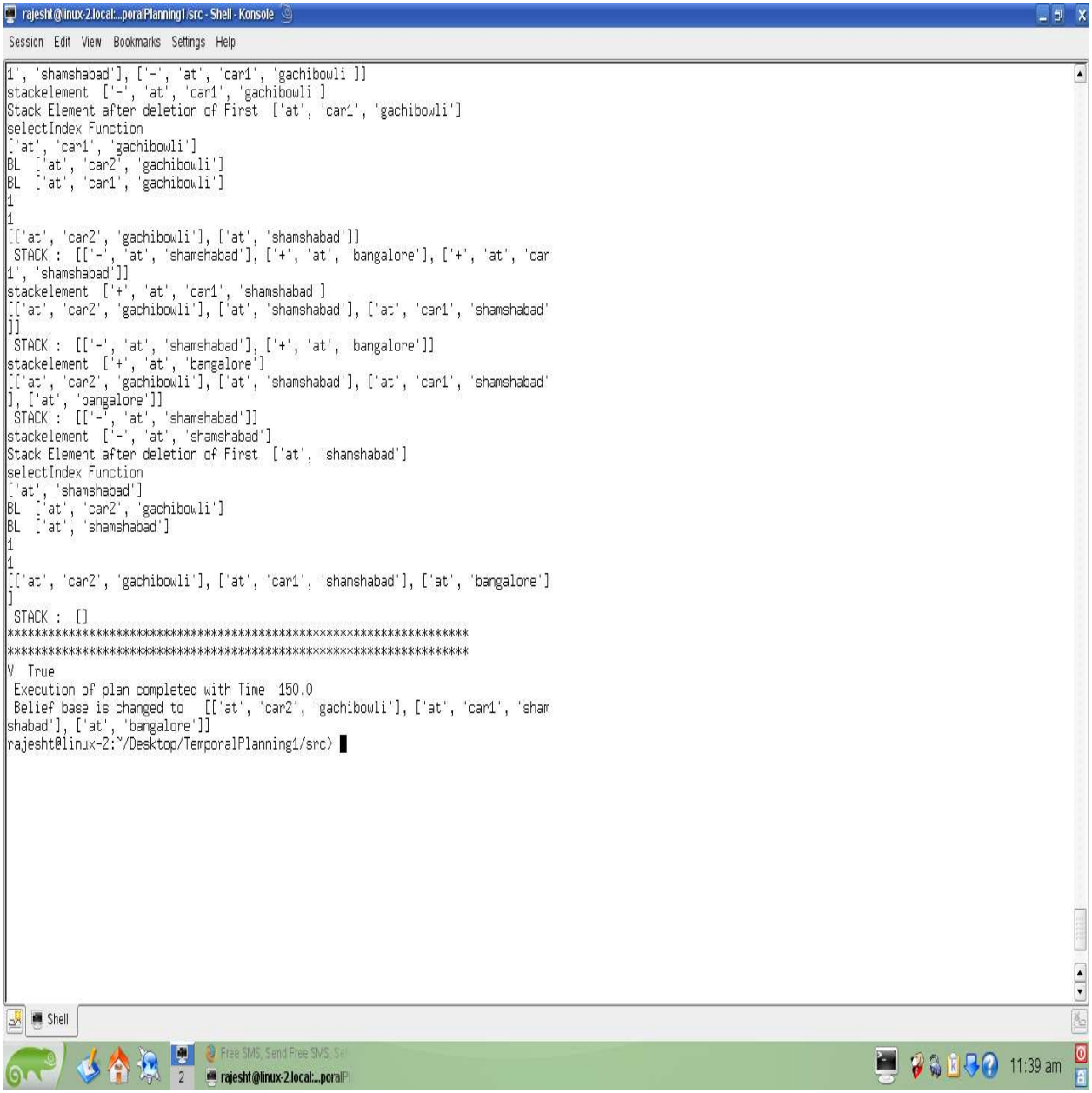

Figure 3.3: Screen 2

Chapter 4

Temporal Planning under Uncertainty

In the previous chapter we showed how temporal planning can be incorporated in a BDI framework. The main contribution was related to the implementation of a *Temporal Seed Plan*. In this chapter we will show how to accommodate an element of `Uncertainty` into the temoral structure developed in the previous chapter. Here we are not solving uncertainty but we are just doing *temporal planning under uncertainty*.

The main drawback of the work depicted in the previous chapter is that there was no consideration of uncertainty. Planning under uncertainty is crucial as far as AI planning is considered because it allow us to think in complex and dynamic environments. For Temporal planning under uncertainty, we use the method of temporal contingent plans i.e, plans with branches that are based on the duration of an action at execution time, using a hill climbing approach. We do this work as an extension of the previous work "Temporal Planning within a BDI architecture" which is used to find a plan that is valid when all actions complete quickly.

In the approach outlined in the previous chapter, we have taken only a single

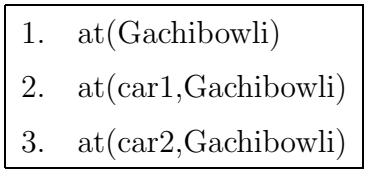

Figure 4.1: Belief Base

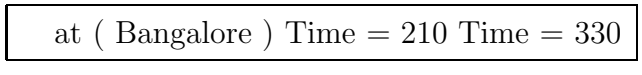

Figure 4.2: Goal

1.	Drive(car1, Gachibowli,Secunderabad)	Time = 210 Time = 270 Cost = 90
2.	Drive(car1, Gachibowli,Shamshabad)	Time = 60 Time = 120 Cost = 70
3.	Drive(car2, Gachibowli,Shamshabad)	Time = 90 Time = 150 Cost = 50
4.	Drive(car2, Gachibowli,Bangalore)	Time = 420 Time = 480 Cost = 240
5.	Fly(Shamshabad,Bangalore)	Time = 90 Time = 150 Cost = 1000
6.	Train(Secunderabad,Bangalore)	Time = 150 Time = 210 Cost = 200

Figure 4.3: Action Base2

time duration assuming that the given action will be completed within that duration. But this may not be possible always. For suppose if there is heavy traffic on road or else train or flight is not arriving at the proper time then the plan generated by the previous method fails. So as part of resolving this situation we take some other duration along with the previous duration assuming that the corresponding action will complete within a duration which is in between the given two durations. For our convenience we define this uncertainty by assigning each action a closed interval duration [min-d,max-d] where min-d and max-d are minimum and maximum reasonable durations required by the action respectively. Here min-d is previous duration and we take it's value as the same as given previously (see 2.3) and we take max-d such that it is greater than min-d.

Here we are going to generate a new plan by taking cost (expense required to complete action) into consideration. Here also the goal is to reach Bangalore but the intention is to decrease total cost of plan (total cost is sum of costs of actions involved in the plan).

Goal timings does not indicate total time durations for completing the plan. These timings are used for increasing action durations (We will see it later).

4.1 Temporal Contingency Plans

To create a temporally contengency plan we need a seed plan. Hence we take our previously generated plan as seed plan i.e, drive car1 from Gachibowli to Shamshabad and take a flight to go from Shamshabad to Bangalore (This takes less time duration). We represent this by using a distance graph (fig-4.4) with possitive and negative edjes between two successive actions.

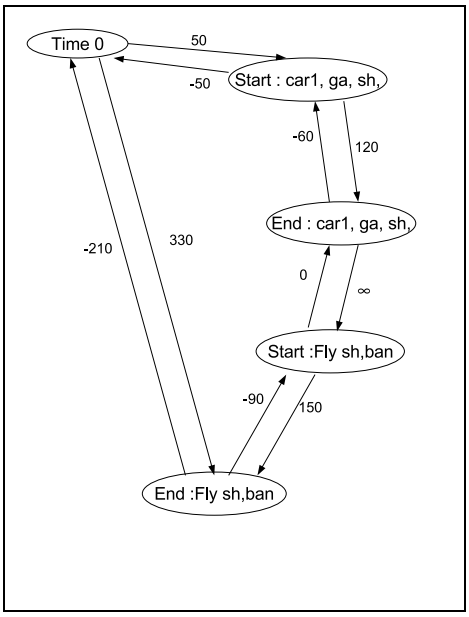

Figure 4.4: Distance graph 1

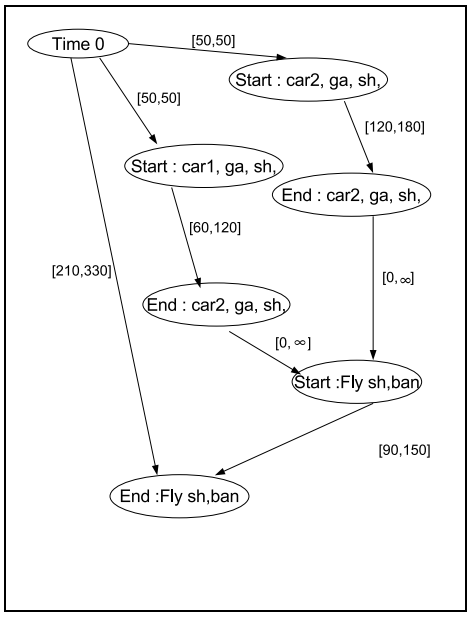

Figure 4.5: Distance graph 2

Here the Phocus-Hc [8] algorithm will guide us to generate a temporally contengent plan. Phocus-Hc means *planning with an algorithmic focus*.The PHOCUS-HC planner starts by making an optimistic assumption that all actions complete as quickly as possible and generates a seed plan with high utility that may become invalid when the assumption proves wrong. It then analyzes the plan and generates more costly contingency branches to be executed only when actions in the seed plan run long enough that an unsafe situation occurs. In addition to generating contingency branches, this hill climbing approach has the advantage of being able to replace the entire seed plan when adding a contingency branch is not possible, or when starting with a new seed plan yields a better expected utility.

The first step in construction of distance graph is to add node s_0 representing time0, and two nodes of each action i, one for its start time s_i and one for its end time e_i. Edges are then added in pairs to represent temporal relations. For each action i, a pair of edges is added between s_i and e_i.

The edges $s_i \rightarrow e_i$ is weighted with max-d of i and the edges $e_i \rightarrow s_i$ is weighted with -1*min-d of i. When an action does not have a constrained start time, the edge $s_0 \rightarrow s_i$ is weighted with ∞ and the edge $s_i \rightarrow s_0$ is weighted with 0.In this way we expressed our plan as simple temporal network. We use this distance graph in our algorithm (Algorithm-5).

The Generate-plan method will be used to find the next immediately available

Algorithm 5: PHOCUS-HC algorithm

Require: Belief base B,Action base A,Goal G

P_0 ←Generate-Seed-Plan(B,A,G,DG)

$P_{current} \leftarrow P_0$

loop

$DG \leftarrow$ Construct-Distance-Graph($P_{current}$,B,A)

$P_{next} \leftarrow$ Generate-Plan(Plan P,Distance graph DG,B,A,G)

if P_{next} is null **then**

return failure

end if

$P_{current} \leftarrow P_{next}$

end loop

Algorithm 6: The Generate-plan method

Require: Plan P,Distance graph DG,B,A,G

for all i=down to 1 in P **do**

$maxAllowedDuration \leftarrow Shortest - Path - Distance(s_i, e_i, DG)$

if maxAllowedDuration $\leq$ max-d of i **then**

$DG, A \leftarrow DG, A$ updated to constrain i to always require max-d of i

$DG, A \leftarrow DG, A$ updated to constrain i to always start at latest possible time that allows max-d of i

else

$newMinDuration \leftarrow maxAllowedDuration + 1$.

$A \leftarrow A$ modified so that action i requires newMinDuration i.e, new min-d.

$P_{new} \leftarrow$generate plan with A_{mod}.

if P and P_{new} have the same steps through step i **then**

return a contengency plan created out of P and P_{new}.

else

return P_{new}.

end if

end if

end for

plan with cost slightly more than that of seed plan. If there are no plans available it returns the same plan. Since we have generated distance graph, now we will see the Generate-plan method.

The Shortest-Path-Distance method used is *Bellman-Ford single source shortest path* algorithm because it is best suitable for finding shortest path distance when negative edges are involved in the graph. In Fig-4.4 we see that duration of the Drive-car1 action is expressed by the interval [60,120] and that of the fly action is expressed by the interval [90,150]. But after applying shortest path algorithm, it is found that the absolute bounds on the duration of Drive-car1 action is expressed by the interval [60,190] and also the duration of the fly action is expressed by the interval [90,220]. We apply the difference between this values to corresponding action intervals of Generate-plan method so that it generates a new plan with in the goal time. This newly generated plan is our desired plan i.e, new plan completes within the goal time and takes less cost compared to the current plan. If for example there is no plans available with less cost compared to current plan, current plan remains itself i.e, our algorithm cann't generate a new plan.

As far as the implementation of our travel problem is concerned we got the new plan *Drive car2 from Gachibowli to Shamshabad* and *take a flight from Shamshabad to Bangalore*. This plan takes less cost compared to our previous plan and also the belief base changed to `car2 at Shamshabad` where initially it was `car2 at Gachibowli`.

Here figure-4.6 shows screen shot for change in belief base.

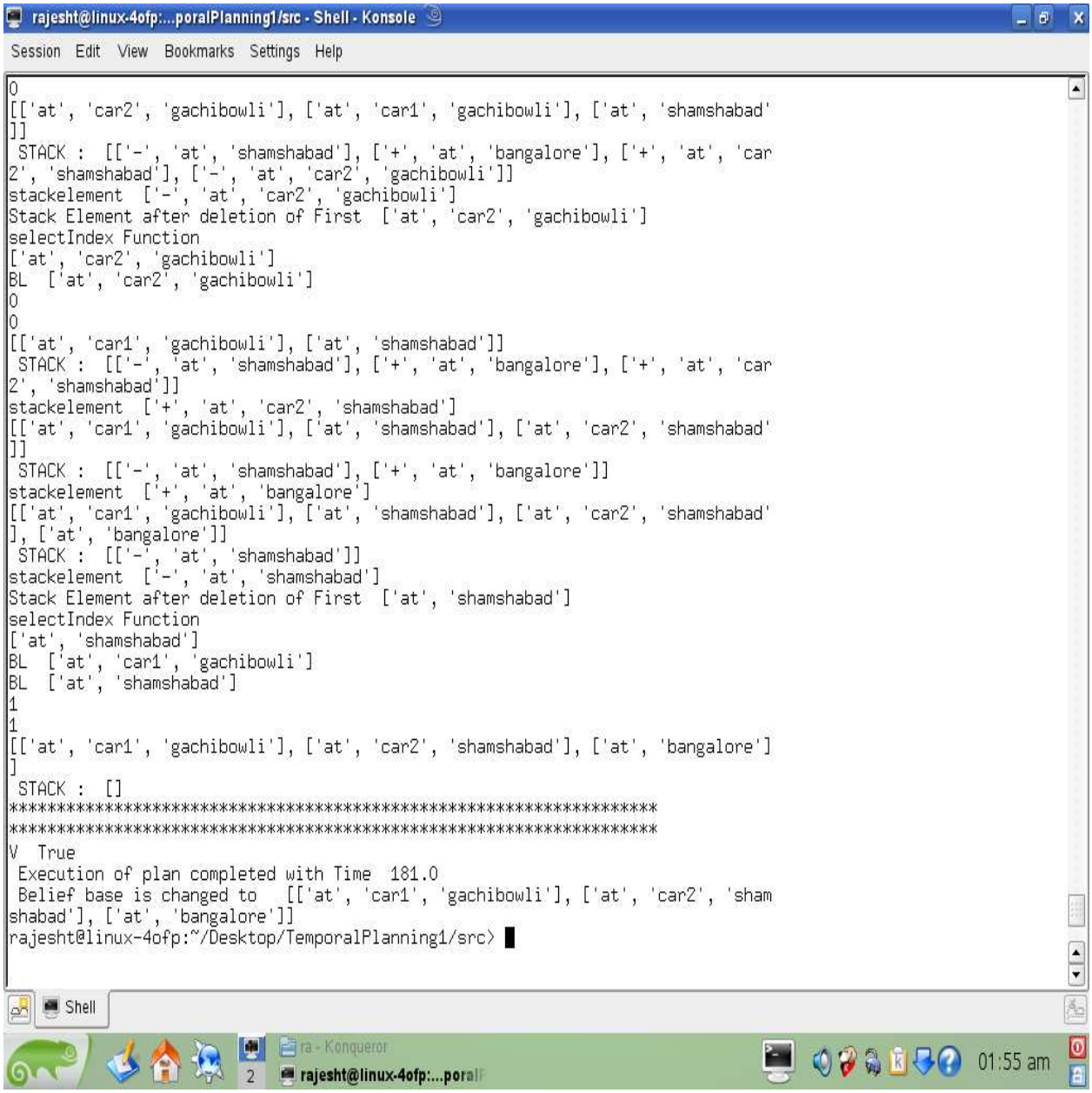

Figure 4.6: Screen 3

Chapter 5

Conclusion and Future Work

In this thesis we gave an in-depth study of Temporal planning and showed how such planning techniques can be applied in Agent architectures. We incorporated the notion of *uncertainty* into temporal planning and demonstrated how a plan could be generated taking into consideration both *temporality* and *uncertainty*. We defined temporal uncertainty by assigning actions interval durations rather than single point durations. To this end we implemented the PHOCUS-HC algorithm which to our knowledge has not been implemented before. The PHOCUS-HC planner starts by making an optimistic assumption that all actions complete as quickly as possible and generates a seed plan with high utility that may become invalid when the assumption proves wrong. It then analyzes the plan and generates more costly contingency branches to be executed only when actions in the seed plan run long enough that an unsafe situation occurs. In addition to generating contingency branches, this hill climbing approach has the advantage of being able to replace the entire seed plan when adding a contingency branch is not possible, or when starting with a new seed plan yields a better expected utility.

In the current version of PHOCUS-HC, a uniform distribution is assumed over all uncertain action durations. In the future we plan to further develop the implementation to allow user specified distributions. Also, the current implementaion always searches untill a plan with 100% safety is found. We plan to improve PHOCUS-HC so that user can choose the the level of safety that is required. We should also like to extend our work to be able to handle actions with uncerain effects and uncertain consumption of non temporal resources.

Bibliography

[1] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *IJCAI*, pages 1636–1642, 1995.

[2] Rafael H. Bordini, Ana L. C. Bazzan, Rafael de Oliveira Jannone, Daniel M. Basso, Rosa Maria Vicari, and Victor R. Lesser. Agentspeak(xl): efficient intention selection in bdi agents via decision-theoretic task scheduling. In *AAMAS*, pages 1294–1302, 2002.

[3] Paolo Busetta, Ralph Ronnquist, Andrew Hodgson, and Andrew Lucas. Jack intelligent agents - components for intelligent agents in java, 1999.

[4] Mehdi Dastani, Frank S. de Boer, Frank Dignum, and John-Jules Ch. Meyer. Programming agent deliberation: an approach illustrated using the 3apl language. In *AAMAS*, pages 97–104, 2003.

[5] Lavindra de Silva, Anthony Dekker, and James Harland. Planning with time limits in bdi agent programming languages. In *CATS*, pages 131–139, 2007.

[6] Minh Binh Do and Subbarao Kambhampati. Planning graph-based heuristics for cost-sensitive temporal planning. In *AIPS*, pages 3–12, 2002.

[7] Minh Binh Do and Subbarao Kambhampati. Sapa: A multi-objective metric temporal planner. *J. Artif. Intell. Res. (JAIR)*, 20:155–194, 2003.

[8] Janae N. Foss and Nilufer Onder. A hill-climbing approach for planning with temporal uncertainty. Technical report, In FLAIRS 2006 Conference, 2006.

[9] Hector Geffner. Perspectives on artificial intelligence planning. In *AAAI/IAAI*, pages 1013–1023, 2002.

[10] S M LaValle. *Planning Algorithms*. Cambridge University Press, 2006.

[11] Felipe Rech Meneguzzi and Michael Luck. Leveraging new plans in agentspeak(pl). In *DALT*, pages 111–127, 2008.

[12] Anand S. Rao. Agentspeak(l): Bdi agents speak out in a logical computable language. In *MAAMAW*, pages 42–55, 1996.

[13] Anand S. Rao and Michael P. Georgeff. An abstract architecture for rational agents. In *KR*, pages 439–449, 1992.

[14] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 2006.

[15] Oscar Sapena and Eva Onaindia. Planning in highly dynamic environments: an anytime approach for planning under time constraints. *Appl. Intell.*, 29(1):90–109, 2008.

[16] David E. Smith and Daniel S. Weld. Temporal planning with mutual exclusion reasoning. In *IJCAI*, pages 326–337, 1999.

[17] Michael Winikoff, Lin Padgham, James Harland, and John Thangarajah. Declarative & procedural goals in intelligent agent systems. In *KR*, pages 470–481, 2002.

[18] Shlomo Zilberstein. Operational rationality through compilation of anytime algorithms. *AI Magazine*, 16(2):79–80, 1995.