Temporal Planning in Dynamic Environments for P-CLAIM Agents

Muhammad Adnan Hashmi
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
75016 Paris, France
Email: Adnan.Hashmi@lip6.fr

Amal El Fallah Seghrouchni
Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
75016 Paris, France
Email: Amal.Elfallah@lip6.fr

***Abstract*—Time and uncertainty of the environment are very important aspects in the development of real world applications. Another important issue for the real world agents is, the balance between deliberation and reactivity. But most of the agent oriented programming languages ignore some or all of these important aspects. In this paper we try to fill this gap by presenting an extension to the architecture of CLAIM agent oriented programming language to endow the agents with the planning capability. We remove the assumption that agents' actions are instantaneous. We are interested in the temporal planning of *on the fly* goals. A coherrent framework is proposed in which agents are able to generate, monitor and repair their temporal plans. Our proposed framework creates a balance between reactivity and deliberation. This work could be considered as a first step towards a complete temporal planning solution for an AOP language.**

I. Introduction

Most of the agent oriented programming languages in the current literature use a PRS like approach to achieve the goals of agent. Some examples of these programming languages are Jason[1], 3APL[2], 2APL[3] and JACK[4]. But these languages lack the ability to incorporate planning. Sometimes the execution of the actions without planning results in the unability to achieve the goals. There has been some work to incorporate planning within such programming languages [5], [6], [7] but these systems do not take into account the duration of agent actions, neither do they consider the uncertainty of the environment. These systems assume that the agents' actions are instantaneous and that the effects produced on the environment are only those which are produced by the agent's actions. But these assumptions are unrealistic for the development of real world applications. There are some systems like ZENO[8], TGP[9], SAPA[10] which give the ability to plan with durative tasks and even there are systems which give this ability in the dynamic environments like IxTeT[11]. But these systems are separate planning solutions. They are not programming languages, so they lack the flexibility and control that a programming language offers to its programmers. Moreover, these systems are built on a proactive approach but in the real world applications it is necessary to create a balance between proactivity and reactivity because it is a dynamic world and the goals of agents are not necessarily given to him at the start, new goals arrive and some old goals are dropped during the life cycle of the agent and some goals require immediate achievement. In this work, we try to fill these gaps by incorporating a temporal planner, an executor, an execution monitor and a plan repairing component to a CLAIM agent[12]. We call this extension of the language as P-CLAIM. The main problems dealt with in this work are 1) Modifications and extensions to the CLAIM agent's architecture to include a temporal planning component. 2) Execution monitoring and plan repairing. 3) Creating a balance between deliberation and reactivity.

In our proposed framework, we have made use of Hierarchical Task Network (HTN) planning technique. The main algorithm used to generate plan for a goal is JSHOP2[13], which is very efficient HTN planning system and plans for tasks in the same order that they will later be executed. The main motivation behind using the HTN planning technique is the similarities among the BDI model of agency and the HTN planning technique[14]. Due to these similarities HTN planning is more suitable and natural candidate for incorporating planning in a BDI like system.

The remainder of this paper is organized as follows. Section 2 puts some light on the current architecture of CLAIM language and JSHOP2 planner. In section 3, some important representations are presented which are helpful in understanding the agent architecture in P-CLAIM. Our proposed architecture of P-CLAIM agent with planning, execution and plan repairing components is presented in section 4. In section 5, we give an example to describe the working of system. Section 6 discussed some of the related work. Section 7 concludes the paper and some future directions are discussed.

II. Background

In this section, we briefly discuss the architecture of CLAIM language and JSHOP2 algorithm to generate a plan. A multi-agent system in CLAIM is a set of distributed hierarchies of agents deployed on computers connected via a network. All the computers have a global clock. With respect to the hierarchical representation, an agent is a node in a hierarchy. It is an autonomous, intelligent and mobile entity. It has a parent and contains (optional) sub-agents, running processes and cognitive elements (e.g. knowledge, goals, capabilities). An agent can dynamically create another agent, and the newly

created agent becomes the sub-agent of the creator agent. In addition, an agent has three mobility primitives, *in* (enter another agent), *out* (leave another agent) and *move* (move from one hierarchy to another).

In CLAIM language, an agent can be defined as follows:

defineAgent *agentName* {
parent=null | *agentName* ***;***
knowledge=null | { *(knowledge;)**}
goals=null | { *(goal;)**}
messages=null | { *(message;)**}
capabilities=null | { *(capability;)**}
processes=null | { *(process* | *)**}
agents=null | { *(agentName;)**}
}

For a more detailed description of CLAIM language, we refer to [12].

JSHOP2 is an HTN planning algorithm, and it deals with the procedural goals. Domain description required by JSHOP2 consists of methods and operators. A method indicates how to decompose a compound task into partially ordered subtasks. A method has three parts. The task for which the method is to be used, the condition which must be true in the current state to apply the method, and subtasks that need to be accomplished in order to accomplish that task. An operator is similar to the operators in classical planning and it tells how to perform a primitive task. It has a condition, a list of add effects and a list of delete effects. Planning proceeds by using the methods to decompose tasks recursively into smaller and smaller subtasks, until the planner reaches primitive tasks that can be performed directly using the planning operators.

The rationale behind choosing JSHOP2 for our work is threefold. Firstly, it is an HTN planner and the domain information from CLAIM can be easily transformed into the domain information needed by the planner due to the similarities among BDI like systems and HTN planning systems as discussed in [14]. Secondly, JSHOP2 plans for the actions in the same order that they will later be executed. So it knows the current state at every planning step. This property of the planner can be exploited for interleaving planning with execution and at every step planner can plan using the current state of the world. Thirdly, it can call external user defined functions to check the precondition of a method or an operator and this property is important for a planning component for CLAIM agents because in CLAIM language there could be calls to user defined functions to check the precondition of capabilities.

III. Some important representations

In this section some important representations are presented which are helpful in understanding the architecture of an agent in P-CLAIM.

A. Domain Representation in P-CLAIM

We have modified the domain representation in CLAIM [12], in order to facilitate the translation to the representation needed by a planner. Agent's *capabilities* have now been divided into *actions* and *activities*. *Actions* are the primitive actions that an agent can perform. Some of the *actions* are programmer defined while the others are already defined in the language like mobility primitives *in*, *out*, *move*. Programmer can also override the already defined *actions* to define his requirements more accurately. An action consists of a condition, a triggering message, the effects and a duration. $TriggerMessage(Act)$ returns the triggering message of an action Act. Each effect of an action has an offset associated with it. This offset is the time taken by the action to produce the effect after the start of the action and it could be zero if this effect is achieved as soon as the action is started or it could be greater than zero. $Offset(Eff)$ denotes the offset associated with an effect Eff. *Activities* are the short plans (recipes) in the plan library of the agent to achieve different composite goals.

B. Goal Representation in P-CLAIM

Goals in P-CLAIM are procedural goals. It means the goals of an agent are the tasks that agent wants to achieve. Some goals are initially given to the agent, when the multi-agent system is launched and some goals are given to the agent during the life of the agent using message passing by other agents or by user interaction. Goals have priorities associated with them. The priority of a goal could be Preemptive High, High or Normal. A goal having Preemptive High priority means that this goal should be immediately achieved by the agent, we also call this goal a reactive goal. High priority means that goal should be achieved before all the Normal priority goals currently present. Normal priority goals are the lowest priority goals. Goals with Preemptive High priority are stored in Global Reactive Goals (GRG) list and all other goals of agent are stored in a priority queue called Global Proactive Goals (GPG) list.

C. Messages Format

A message received by an agent in P-CLAIM has five parts. First part is the *identity*. Each message is assigned a unique number as identity. Second part is the *sender*, which represents the sender of the message. Thirdly, a message has a *priority* associated with it. This field has a value among Preemptive High, High and Normal. Fourthly, a message has a *proposition* which is the actual contents of the message. This *proposition* could be a new goal to achieve or it could be an information given to the agent which was demanded by the agent in an earlier message. Finally, a message has a *ResponseTo* field which is either blank or it contains a number pointing to the *identity* of an earlier message to which this message is responding.

D. Translation of Domain Description

The information needed by JSHOP2 algorithm to generate the plan includes the initial state information, goals information and domain description (methods and operators). In our formalism, this information is automatically extracted from the agent. Initial state information is taken from the *knowledge* of

the agent and from the hierarchical representation of MAS. Goal for the Planner is a one to one mapping from agent's goal to Planner's goal. In our framework, only one goal is passed to the JSHOP2 algorithm at a time. Agent's actions are mapped to the operators in JSHOP2. P-CLAIM agent's *activities* are converted into JSHOP2 methods. For each *activity* of the agent, an equivalent method is generated with the same name as that of *activity*. *Activity's* condition is mapped to the method's precondition. In JSHOP2, methods have subtasks. Subtasks may be primitive tasks or other composite tasks. Equivalently in P-CLAIM, the body of an *activity* consists of different processes. So we need to convert these processes into JSHOP2 methods and operators. To read in detail about this conversion, we refer to our earlier article[15].

E. Policy File

Each agent maintains a policy file in which it stores the importance of all other agents in the MAS for him. Importance of an agent depends on its position in the hierarchy relative to the position of the agent who is maintaining the policy. Importance also depends on the services provided by the agent during the life cycle of the agent. After receiving the message, the agent analyzes the policy file. Importance of the agent could be Normal or High.

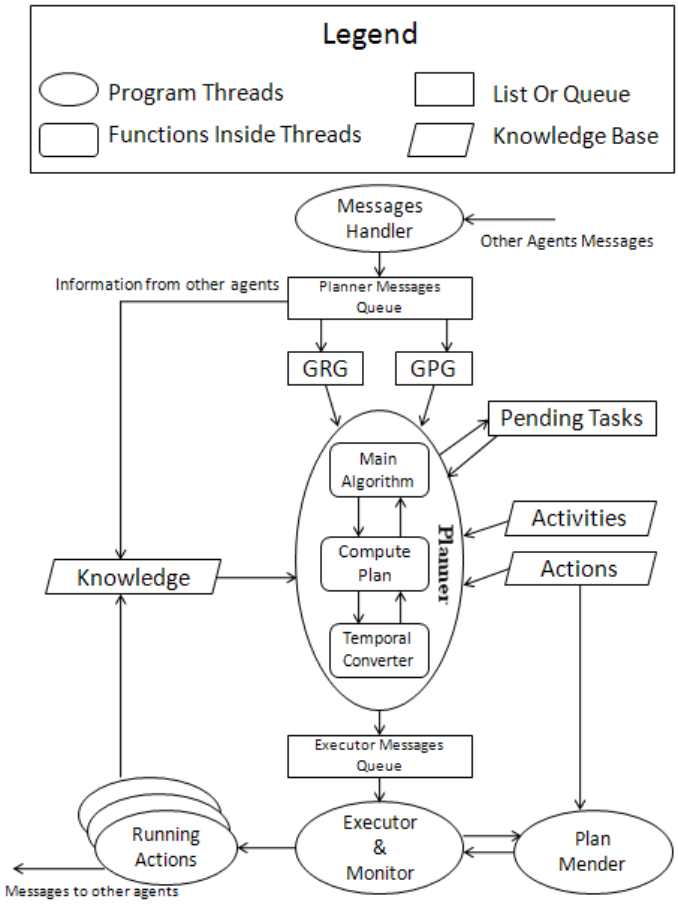

Fig. 1. A Running P-CLAIM Agent

IV. Agent Architecture

There are concurrently four threads running inside the agent all the time. In the following subsections, we explain these threads in detail. Figure 1 is showing the architecture of an agent.

A. Messages Handler

This thread is always waiting for the messages from other agents or from the user. It puts the messages into Planner Messages Queue(PMQ). These messages are either a request to achieve some goal or they are responses to some earlier sent message. After putting in the PMQ, these messages are fetched and analyzed. If the message contains some information demanded in an earlier message then this information is added to the *knowledge* of the agent along with an *acknowledgement* having the *identity* of the message in which this information was demanded. Agent's treatment of a message, which is a request to achieve some goal, depends on the priority associated with message and the importance of sender.

The Messages Handler fetches the goal attached with a message and assigns a priority to the goal based on the priority associated with message and the importance of sender. A goal fetched from a message of priority Preemptive High or High which is assigned by an agent having Normal importance in the policy file is assigned a High priority. It means that agent does not preempt his own goals for the goals assigned by an agent of Normal importance. A goal fetched from a message, sent by an agent having High importance in the policy file is assigned the same priority as of the message. After assigning a priority to the goal, the goal is added to one of the two global goals lists. A goal of priority Preemptive High is added to GRG list and a goal of priority High or Normal is added to GPG list.

There is another messages queue maintained inside the agent, called Executor Messages Queue(EMQ). Messages which are sent by the Planner for the execution of actions are put in the EMQ. These are the triggering messages for the actions in the plan generated by the Planner. Number of messages in EMQ are denoted by $length(EMQ)$ and $EMQ[i]$ denotes the i^{th} message in EMQ. Each triggering message in EMQ has a time stamp associated with it. $TimeStamp(Msg)$ denotes the time stamp associated with a triggering message Msg.

B. Planner

Planner starts when the multi-agent system is launched. Once started, the Planner procedure runs throughout the life cycle of the agent. When there are no goals in either of the goals lists then it sits idle and waits for new goals to arrive and as soon as a new goal arrives, it starts planning. Before starting the Planner, the agent goes through an initialization phase, in which it sets the values of certain global variables.

Three states of the world are maintained in the system, $SP(Act)$ which denotes the state of the world anticipated by planner just before the execution of the action Act, secondly SW is the current actual state of the world and $FinalSP$

Algorithm 1 $Main\_Algorithm$

```
1: loop
2:   repeat
3:     Call Treat_Reactive_Goal
4:   until GRG = φ
5:   if GPG ≠ φ then
6:     Fetch first goal g ∈ GPG
7:     PPlan ← Compute_Plan(FinalSP, g, D)
8:     if PPlan ≠ Fail then
9:       for i = 1 To length(PPlan) do
10:        TimeStamp(ExeMessages[i]) ← TimeStamp(PPlan[i])
11:        ExeMessages[i] ← TriggerMessage(PPlan[i])
12:      end for
13:      Send ExeMessages to EMQ
14:    end if
15:  end if
16: end loop
```

Algorithm 2 $Treat\_Reactive\_Goal$

```
1: Fetch first goal g ∈ GRG
2: Suspension_Signal ← ON
3: Wait until {Execution_Signal = OFF}
4: Start_Time ← Current system time
5: RPlan ← Compute_Plan(SW, g, D)
6: if RPlan ≠ Fail then
7:   for i = 1 To length(RPlan) do
8:     TimeStamp(ExeMessages[i]) ← TimeStamp(RPlan[i])
9:     ExeMessages[i] ← TriggerMessage(RPlan[i])
10:  end for
11:  End_Time ← Current system time
12:  Duration ← End_Time - Start_Time
13:  for i = 1 To length(EMQ) do
14:    TimeStamp(EMQ[i]) ← TimeStamp(EMQ[i]) + Duration
15:  end for
16:  Send ExeMessages to EMQ
17: end if
18: Suspension_Signal ← OFF
```

Algorithm 3 $Compute\_Plan(S, G, D)$

```
1: P ← The Empty Plan
2: I ← S
3: LG ← G
4: LG0 ← {g ∈ LG : no goal is constrained to precede g}
5: loop
6:   if LG = φ then
7:     Plan ← Call Temporal_Converter(I, P, D)
8:     Return Plan
9:   end if
10:  Non deterministically choose any g ∈ LG0
11:  if g = Some Primitive Action then
12:    if g = Information Gathering Task then
13:      Generate and send message with identity x, for information retrieval to other agent
14:      Put all tasks depending on g in Pending Tasks list and assign them an identity x
15:      Remove g from LG
16:    else
17:      A ← {(a, Θ) : a is a ground instance of an operator in D, Θ is a substitution that unifies {head(a), g}, and S satisfies a's preconditions}
18:      if A = φ then
19:        Return Fail
20:      else
21:        Non deterministically choose a pair (a, Θ) ∈ A
22:        S ← S + Add(a) − Del(a)
23:        Append a to P
24:        Modify LG by removing g and applying Θ
25:      end if
26:    end if
27:    LG0 ← {g ∈ LG : no other goal is constrained to precede g}
28:  else
29:    M ← {(m, Θ) : m is an instance of a method in D, Θ unifies {head(m), g}, pre(m) is True in S, and m and Θ are as general as possible}
30:    if M = φ then
31:      Return Fail
32:    end if
33:    Non deterministically choose pair (m, Θ) ∈ M
34:    Modify LG by removing g, adding sub(m), constraining each goal in sub(m) to precede the goals that g preceded, and applying Θ
35:    if sub(m) ≠ φ then
36:      LG0 ← {g ∈ sub(m) : no goal in LG precedes g}
37:    else
38:      LG0 ← {g ∈ LG : no goal in LG precedes g}
39:    end if
40:  end if
41:  if New acknowledgement in knowledge then
42:    id ← identity of the message whose acknowledgement has arrived
43:    Fetch all goals associated with message id from Pending Tasks list and put in the LG
44:  end if
45:  if G is a proactive goal then
46:    repeat
47:      Call Treat_Reactive_Goal
48:    until GRG = φ
49:  end if
50: end loop
```

denotes the state of the world to which the Planner has planned till now. More precisely, it is the state of the world anticipated by planner after the very last action that the Planner has planned for. In the initialization phase $FinalSP$ is set equal to the SW. $Suspension\_Signal$ is set to OFF and $Execution\_Signal$ is set to ON.

The $Main\_Algorithm$ (Algorithm 1) runs in an infinite loop and ensures that reactive goals are immediately planned for and achieved. First it looks at the GRG list and if it is not empty, (Lines 2-4) the control moves to the procedure $Treat\_Reactive\_Goal$ (Algorithm 2). Some of the notations used inside the $Treat\_Reactive\_Goal$ procedure are as follows. $length(RPlan)$ denotes the number of actions in the plan $RPlan$. $ExeMessages$ is an array of triggering messages for the actions in the plan. $TimeStamp(Act)$ denotes the time stamp assigned to an action Act for its execution. $Treat\_Reactive\_Goal$ fetches the first reactive goal g and sets the $Suspension\_Signal$ to ON to ask the Executor to suspend the execution and waits for the $Execution\_Signal$ to go OFF which indicates that the Executor has suspended the execution (Lines 1-3) then it calls the $Compute\_Plan$ procedure to plan for the reactive goal (Line 5). The current state of the world SW, the reactive goal just fetched g and domain description D are passed to $Compute\_Plan$ procedure. This procedure call returns a temporal plan $RPlan$ for the reactive goal. Because every action in P-CLAIM is executed using a triggering message, so an array $ExeMessages$ is generated containing the triggering messages for all the actions in the temporal plan $RPlan$ with a $TimeStamp$ associated with every message (Lines 7-10) and this array of

Algorithm 4 $Temporal\_Converter(I, P, D)$

1: **for** $j = 1$ TO $no\_of\_literals(I)$ **do**
2: $Production\_Time(Literal(I[j])) \leftarrow 0$
3: **end for**
4: **for** $i = 1$ TO $length(P)$ **do**
5: $TimeStamp(P[i]) \leftarrow Max\ \{Production\_Time(Pre(P[i][j]))$ $:j = 1$ To $no\_of\_pre(P[i])\}$
6: $Prereq(P[i]) \leftarrow$ Actions which achieve the preconditions of $P[i]$
7: $SP(P[i]) \leftarrow$World state anticipated before the execution of $P[i]$
8: **for** j = 1 TO $no\_of\_effects(P[i])$ **do**
9: $Production\_Time(Literal(P[i][j])) \leftarrow TimeStamp(P[i])$ + $Offset(Literal(P[i][j]))$
10: **end for**
11: **end for**

messages is sent to EMQ (Line 17) from where the Executor executes the actions triggered by these messages. But before sending $ExeMessages$ to EMQ, the $TimeStamp$ of all the messages currently in the EMQ is updated, because due to the suspension of execution, those triggering messages can not be executed at their intended time. So every message's $TimeStamp$ is increased by the duration of the suspension (Lines 13-15). $Suspension\_Signal$ is then set to OFF (Line 18) to allow the Executor to resume execution and control is passed back to $Main\_Algorithm$ (Algorithm 1) which looks for another goal in GRG. The $Main\_Algorithm$ turns its attention to the proactive goals only when it finds that there is no reactive goal (Line 5). Algorithm fetches the first goal from GPG (Line 6). High priority goals are always fetched before Normal priority goals. Then $Compute\_Plan$ procedure is called with the parameters $FinalSP$, g and D. A plan $PPlan$ is returned (Line 7) which is then sent to EMQ in the form of triggering messages (Lines 9-13). Now we elaborate the working of $Compute\_Plan$ procedure (Algorithm 3) (Many lines of the algorithm are taken from [13]).

$Compute\_Plan$ procedure is an extension of JSHOP2[13] algorithm. It takes three parameters S, G and D as input, where S is initial state, G is a list of goals and D is the agent's domain description. $Compute\_Plan$ procedure has an internal goals list called Local Goals (LG) list . Algorithm chooses a goal $g \in LG$ which has no predecessors (Line 4). At this point there could be two cases. The first case is if g is a primitive task, then procedure finds an operator a that matches g and whose preconditions are satisfied in S. It applies the action a to state S and adds it to his plan P (Lines 17,21-23). If no such operator a exists then procedure returns failure (Lines 18-19). In P-CLAIM a message to other agent is also treated as primitive action. So, g could be a message to other agent for information retrieval. If this is the case, then a message for the retrieval of information is generated with identity x and is sent to other agent. And all the tasks which depend on this information are put in the Pending Tasks list (Lines 11-15). All these tasks are assigned same identity x as of the message before sending them to Pending Tasks list.

The second case is where g is a compound goal, so a method needs to be applied for the decomposition of g into its subtasks. In this case the planner nondeterministically chooses a method instance m matching g, that decomposes g into subgoals (Line 29) and applies this method (Lines 33-34). If no such method m exists then procedure returns failure (Lines 30-32).

At the end of each planning step, the $Compute\_Plan$ procedure looks for any newly arrived $acknowledgement$ for an earlier sent message. If a new $acknowledgement$ for a message with identity id has been arrived then the procedure removes all the tasks depending on id, from Pending Tasks list and puts them in the Local Goals list to process those goals (Lines 41-44).

While planning for a proactive goal, the $Compute\_Plan$ procedure checks GRG for any new goals after each planning step and whenever it finds a goal in GRG, it suspends planning for the proactive goal and calls the procedure $Treat\_Reactive\_Goal$, which we have explained earlier (Lines 45-49). When GRG becomes empty, procedure resumes planning for the proactive goal from the same state at which it had suspended the planning. While planning for a reactive goal, the $Compute\_Plan$ procedure does not look at GRG, because a new reactive goal is treated only when all the previous reactive goals have been treated.

When $Compute\_Plan$ finds a plan for one goal, it converts the total order plan into a temporal plan by calling the procedure $Temporal\_Converter$ (Algorithm 4). The procedure takes three parameters I, P and D, where I is the initial state, P is the total order plan which is to be converted and D is the domain description file which is needed to extract the information about the durations of all the actions and offsets of all the effects. Some notations used in the procedure are as follows. $no\_of\_literals(I)$ denotes the number of literals in the initial state and $Literal(I[j])$ points to the j^{th} literal in initial state. $Production\_Time(Lit)$ represents the time of achievement of a literal Lit. $length(P)$ returns the number of actions in the plan P. $no\_of\_pre(Act)$ and $no\_of\_effects(Act)$ denote the number of preconditions and number of effects of an action Act respectively while in the same vein $Pre(P[i][j])$ and $Literal(P[i][j])$ denote the j^{th} precondition and j^{th} effect of i^{th} action in plan P respectively. We have used a simple and efficient technique to convert a plan into temporal plan. The procedure starts by setting the $Production\_Time$ of all the literals in the initial state to 0 (Lines 1-3). Then procedure loops through all the actions starting from the first action, going towards the last one and sets the $TimeStamp$ of the action to the maximum of the $Production\_Time$ of all its preconditions, because an action can be executed at least when all of its preconditions have been achieved (Lines 4-5). After setting the $TimeStamp$ of an action, the procedure sets the $Production\_Time$ of all the effects of the action. The production time of an effect is the $TimeStamp$ of the action plus the time at which the effect is produced by the action, the $Offset$ of the effect (Lines 8-10).

C. Executor

The Executor is running in parallel with the Planner. It waits for triggering messages to come in the EMQ, fetches the messages and executes the actions associated with the

Algorithm 5 $Executor$

```
1: loop
2:   if Suspension_Signal = ON then
3:     Execution_Signal ← OFF
4:     Wait until {Suspension_Signal = OFF}
5:     Execution_Signal ← ON
6:   end if
7:   if EMQ ≠ φ then
8:     NextActions ← Fetch all next messages C from EMQ having
       the earliest TimeStamp from current system time
9:     NextTime ← TimeStamp(NextActions)
10:    Wait for system time to reach NextTime
11:    for i = 1 TO length(NextActions) do
12:      if All the actions in Prereq(NextActions[i]) has not sent
         acknowledgement for termination then
13:        Wait for all the acknowledgements
14:        Duration ← Time spent waiting for acknowledgements
15:        for i = 1 To length(EMQ) do
16:          TimeStamp(EMQ[i]) ← TimeStamp(EMQ[i]) +
             Duration
17:        end for
18:      end if
19:      if SP(NextActions[i]) = SW  then
20:        Execute NextActions[i] in a separate thread
21:      else
22:        MPlan ← Plan_Mender(SW, SP(NextActions[i]))
23:        Execute MPlan
24:        for i = 1 To length(EMQ) do
25:          TimeStamp(EMQ[i]) ← TimeStamp(EMQ[i]) +
             TimeSpan(MPlan)
26:        end for
27:        Execute NextActions[i] in a separate thread
28:      end if
29:    end for
30:  end if
31: end loop
```

messages at their planned time stamps. Every running action sends an acknowledgement just before its termination to the Executor. Algorithm 5 is a simplified version of the Executor. The Executor fetches all the next messages from EMQ that have the closest $TimeStamp$ to the current system time. Then the Executor waits for the system time to reach the $TimeStamp$ of these messages (Lines 10-11). When system time approaches that time, the Executor checks whether the prerequisite actions of the actions associated with these messages have been terminated or not. If they have not been terminated then it waits for their termination. And increases the $TimeStamp$ of all the messages in EMQ by the duration of waiting for their termination (Lines 14-20). Then it checks for any discrepency among the current world state and the one anticipated by the Planner for the execution of these actions. If there is no discrepency then these actions are executed in a separate thread (Lines 21-22) and the Executor fetches the next messages from EMQ. But if there is discrepency among the two world states, then the Executor calls the Plan Mender to generate a plan from the current world state to the intended world state and executes the plan thus returned to remove the discrepency (Lines 24-25). After executing this plan the Executor is ready to execute the actions which it had suspended due to discrepency (Line 29). But before executing these actions, it augments their $TimeStamp$ by the duration of the discrepency removal. Moreover, after executing each action, the Executor checks the $Suspension\_Signal$. When $Suspension\_Signal$ is set to ON, it turns $Execution\_Signl$ to OFF, suspends the execution, and waits for $Suspension\_Signal$ to go OFF. The Executor resumes the execution once the $Suspension\_Signal$ is turned to OFF. But now the triggering messages for the plan of reactive goal are at the front of EMQ, so the Executor first executes the plan for the reactive goal for which it had suspended the execution and then it resumes the execution of plan on which it was working before the suspension (Line 29).

Algorithm 6 $Plan\_Mender(I, G)$

```
1: Generate a plan P using SATPLAN from I to G ignoring the duration
   of actions
2: TP ← Call Temporal_Converter(I, P, D)
3: Return TP
```

D. Plan Mender

This procedure is responsible for repairing the plan. It takes as input the current actual world state I and the anticipated world state G. The Plan Mender generates a temporal plan for the agent to reach the anticipated world state starting from the current world state and returns this plan to the Executor. The Plan mender uses the classical STRIPS style planning technique to compute its plan because now the goals for the planner are a state to be reached (declarative goal). So, the Plan Mender just uses operators from the domain description file to compute the plan. In this case, the activities are not helpful in generating the plan which were used by the Planner component. The basic algorithm used by the Plan Mender is shown in Algorithm 6. Plan mender computes a plan without taking into account the durations of the actions using the SATPLAN planner[16] and then uses the procedure $Temporal\_Converter$ to convert the plan to a temporal plan.

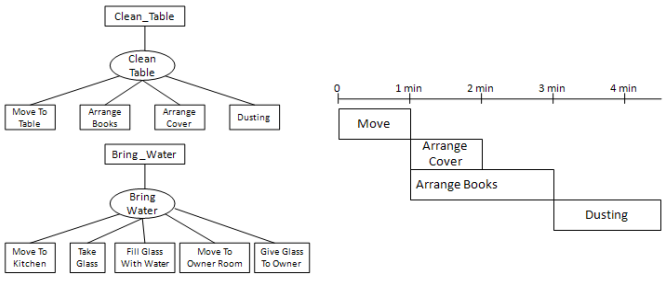

Fig. 2. (a).ROCO Activities (b).Plan for $Clean\_Table$

V. Example

In this example scenario we have one mobile agent ROCO, which is a home servant. When the MAS is launched then ROCO has the goal $Clean\_Table$. ROCO has *activity* associated with this goal $Clean\_Table$. All the *activities* of ROCO are shown in tree form in Figure 2(a) (A rectangle is showing a goal and an associated oval is an Activity associated with the goal, rectangles at the bottom without an

associated oval are Actions.). $Main\_Algorithm$ fetches goal $Clean\_Table$ from GPG list. And calls $Compute\_Plan$ to plan for this goal which generates a plan consisting of the following actions $Move(Room1, Table)$, $ArrangeBooks$, $ArrangeCover$, $Dusting$. A short description of the actions in this plan is shown in Figure 3.

Move(Room1, Table){
Duration = 1 minute
Pre = At(ROCO, Room1)
Effects = At(ROCO, Table)
}
	ArrangeCover{
	Duration = 1 minute
	Pre = At(ROCO, Table)
	Effects = Arranged(Cover)
	}

ArrangeBooks{
Duration = 2 minutes
Pre = At(ROCO, Table)
Effects = Arranged(Books)
}
	Dusting{
	Duration = 1.5 minutes
	Pre = At(ROCO, Table), Arranged(Books), Arranged(Cover)
	Effect = Clean(Table)
	}

Fig. 3. Description of actions in the temporal plan for $Clean\_Table$

The plan is converted to the temporal plan using the procedure $Temporal\_Converter$ and the plan returned is shown in figure 2(b). In this example, all the effects of all the actions have an offset equal to the duration of the action. Here we explain the conversion of totally ordered plan to temporal plan. Procedure starts by assigning the $Production\_Time$ of all the literals in the initial state to 0. There is only one literal $At(ROCO, Room1)$ in the initial state so $Production\_Time(At(ROCO, Room1))$ is set to 0. Now the procedure takes first action $Move(Room1, Table)$ and sets its $TimeStamp$ to 0, which is the maximum $Production\_Time$ from all of its preconditions. Next, the procedure sets the $Production\_Time$ of the effects of $Move(Room1, Table)$. This action has only one effect $At(ROCO, Table)$. $Production\_Time(At(ROCO, Table))$ is assigned the value $TimeStamp(Move(Room1))$ plus $Offset(At(ROCO, Table))$. Putting the values, we get $Production\_Time(At(ROCO, Table))$ equals 1 minute, because $Offset(At(ROCO, Table))$ is equal to the duration of $Move(Room1, Table)$. Now procedure moves to second goal which is $Arrange\_Books$ and sets its $TimeStamp$ to be the maximum of $Production\_Time$ of all of its preconditions. It has only one precondition $At(ROCO, Table)$ whose $Production\_Time$ has already been calculated to 1 minute. So $TimeStamp(Arrange\_Books)$ is assigned 1 munite. In this way the procedure continues and finds the plan shown in figure 2(b). Planner sends the messages for each action of the plan along with their $TimeStamp$ to the EMQ for execution and Executor starts executing the plan. When the Executor has executed $Move(Room1, Table)$, $ArrangeBooks$ and $ArrangeCover$, it checks that $Suspension\_Signal$ is set to ON, because the Planner has just fetched a reactive goal $Bring\_Water$ from the GRG. The Executor suspends the execution, sets the $Execution\_Signal$ to OFF and waits for the $Suspension\_Signal$ to go to OFF again. It receives the following plan in the EMQ, $Move(Table, Kitchen)$, $TakeGlass$, $FillGlassWithWater$, $Move(Kitchen, OwnerRoom)$, $Give(Glass, Owner)$. Now the Executor executes this plan. After the execution of this plan ROCO is in $OwnerRoom$. Now the Executor resumes its suspended plan but before resuming the suspended plan, it increases the $TimeStamp$ of all the actions in the suspended plan by the $TimeSpan$ of the plan for goal $Bring\_Water$, then it checks whether the preconditions of the suspended plan hold in the current state. The preconditions of its suspended plan are $At(ROCO, Table) \wedge Arranged(Books) \wedge Arranged(Cover)$ and the current state is $Arranged(Books) \wedge Arranged(Cover) \wedge At(ROCO, OwnerRoom)$. The Executor calls Plan Mender to generate a plan from current state of the world to the intended state of the world. Plan Mender returns a plan consisting of only one action $Move(OwnerRoom, Table)$. Executor executes this plan, so ROCO moves to Table. Now again the Executor checks for any discrepency among the current state and the anticipated state but now both states are same so the Executor executes the suspended plan i.e. it executes the $Dusting$ action.

VI. Related Work

In this section, we briefly review some work from the existing literature which is related to our work. Some of the research related to ours is CYPRESS[17], RETSINA[18], DECAF[19] and the systems proposed in [5] and [6].

In our opinion, the system closest to our research is CYPRESS system, which also integrates a planning system SIPE-2[20] with an execution system PRS[21]. It has the ability to react to the unanticipated changes in the environment by replanning and also deals with probabilistic planning. Our approach has the added advantage of handling temporal knowledge. Another aspect differentiating P-CLAIM to CYPRESS is the mobility of the agents. In P-CLAIM, the agents are mobile so the context of an agent changes while moving from one machine to another. The planner component must be able to deal with the changing context because the planning is interleaved with execution. An advantage of CYPRESS system over our proposed system is in the way CYPRESS performs replanning. We suspend the execution while computing a plan to remove any discrepencies. While CYPRESS system uses *asynchronous replanning* in which the system continues to execute the unaffected portion of the plan while a planning module computes a new plan.

Our system has many similarities with RETSINA. Like our system, RETSINA also interleaves planning with execution and supports planning for dynamic and real environments. But one main difference of RETSINA system with our system is that RESTINA system plans by only reduction of the top level task and it does not plan among the top level tasks, but our system uses a HTN planner which also plans among the top level tasks. So the plan generated is more optimal in our system than in RETSINA system. Another main difference is that RETSINA system does not use the existing information from the BDI system whereas our system proposes a method to use the existing agent's and world's information.

Another framework DECAF[19] which can be seen as an inspiration of RETSINA, relates to our system. But, in DECAF,

the planner only estimates preconditions, select task templates and instantiates them. It lacks the ability to anticipate future actions.

Like our system, [5] also provides a way to translate the information from a JACK[4] agent to the information needed by JSHOP[22] planner. Main differences of this approach with our approach are that in [5] it is the responsibility of the programmer to specify the points where the planner should be called while our system plans for each goal. Our system has the ability to deal with the unanticipated changes in the environment, while [5] has no such ability.

Another framework incorporating planning in a BDI language is presented in [6]. It incorporates classical planning into BDI framework. More precisely it extends the X-BDI[23] model to use the propositional planning algorithms for performaing means-end reasoning. Our hypothesis is that our proposed system has the advantage of being more efficient as the HTN planning technique can find plans more rapidly with the help of additional domain knowledge provided by the programmer. Another important aspect is the loss of the domain knowledge provided by the programmer in [6]. The advantage of using the HTN planning is that the plans can be synthesized according to the intentions of the programmer without loosing the domain knowledge.

VII. Conclusion And Future Work

In this paper, we have presented an extension to the CLAIM language to endow the agents with the capability to plan ahead. This modified and extended language is called P-CLAIM. Agents are able to create temporal plans. Execution monitoring and plan repairing components are added. A balance between deliberation and reactivity has been established and the agents are able to turn their attention while planning to the newly arrived reactive goals. This work can be considered as a first step towards a comprehensive temporal planning solution for an Agent Oriented Programming language.

After creating the temporal plan for an agent but before its execution, the plan of an agent should be coordinated with the plans of those agents with which the plan could be in conflict or whose plans could be helpful for this agent. Our next task is to propose a coordination mechanism to coordinate the temporal plans of different agents. Coordinating the plan of agent with every other agent in the MAS is very costly, so another important task to do is to intelligently calculate the set of those agents whose plan could be in conflict or whose plans could be helpful for the agent and then the plan should be coordinated with only those agents.

References

[1] R. Bordini, J. Hubner, and R. Vieira, "Jason and the Golden Fleece of agent-oriented programming," *Multiagent systems artificial societies and simulated organizations*, vol. 15, p. 3, 2005.

[2] M. Dastani, M. van Riemsdijk, and J. Meyer, "Programming multi-agent systems in 3APL," *Multiagent systems artificial societies and simulated organizations*, vol. 15, p. 39, 2005.

[3] M. Dastani and J. Meyer, "A practical agent programming language," *Lecture Notes in Computer Science*, vol. 4908, p. 107, 2008.

[4] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas, "Jack intelligent agents-components for intelligent agents in java," *AgentLink News Letter*, vol. 2, pp. 2–5, 1999.

[5] L. de Silva and L. Padgham, "Planning on demand in BDI systems," *Proc. of ICAPS-05 (Poster)*, 2005.

[6] F. Meneguzzi, A. Zorzo, and M. da Costa Mora, "Propositional planning in BDI agents," in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM New York, NY, USA, 2004, pp. 58–63.

[7] de Silva et al., "First Principles Planning in BDI Systems," in *Proceedings of the 8th international joint conference on Autonomous agents and multiagent systems*, S. Decker, Sichman and C. (.eds), Eds., 2009, pp. 1105–1112.

[8] J. Penberthy and D. Weld, "Temporal planning with continuous change," in *Proceedings of the national conference on Artificial Intelligence*. John Wiley & Sons Ltd., 1995, pp. 1010–1010.

[9] D. Smith and D. Weld, "Temporal planning with mutual exclusion reasoning," in *International joint conference on artificial intelligence*, vol. 16. Lawrence Erlbaum Associates Ltd., 1999, pp. 326–337.

[10] M. Do and S. Kambhampati, "Sapa: A domain-independent heuristic metric temporal planner," in *Proceedings of ECP-01*, 2001, pp. 109–120.

[11] M. Ghallab and H. Laruelle, "Representation and control in IxTeT, a temporal planner," in *Proc. 2nd Int. Conf. on AI Planning Systems*, 1994, pp. 61–67.

[12] A. El Fallah-Seghrouchni and A. Suna, "An unified framework for programming autonomous, intelligent and mobile agents," *Lecture notes in computer science, Springer*, pp. 353–362, 2003.

[13] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, vol. 20, no. 1, pp. 379–404, 2003.

[14] S. Sardina and L. Padgham, "Hierarchical planning in BDI agent programming languages: A formal approach," in *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*. ACM New York, NY, USA, 2006, pp. 1001–1008.

[15] H. M. Adnan, "A Planning Component for CLAIM Agents," in *To appear in the Proceedings of International Workshop On Multi-Agent Systems Technology and Semantics*. IEEE Romania, 2009.

[16] H. Kautz, B. Selman, and J. Hoffmann, "Satplan: Planning as satisfiability," in *5th International Planning Competition*. Citeseer, 2006.

[17] D. Myers, L. Wesley, and A. Center, "CYPRESS: Reacting and Planning under Uncertainty," in *DARPA Proceedings: Rome Laboratory Planning Initiative*. Morgan Kaufmann, 1994, p. 111.

[18] M. Paolucci, O. Shehory, K. Sycara, D. Kalp, and A. Pannu, "A planning component for RETSINA agents," *Lecture notes in computer science, Springer*, pp. 147–161, 2000.

[19] J. Graham and K. Decker, "Towards a distributed, environment-centered agent framework," *Lecture notes in computer science, Springer*, pp. 290–304, 2000.

[20] D. Wilkins, "Can AI planners solve practical problems?" *Computational Intelligence*, vol. 6, no. 4, pp. 232–246, 1990.

[21] M. Georgeff and A. Lansky, "Procedural knowledge," *Proceedings of the IEEE, Special Issue on Knowledge Representation*, vol. 74, no. 10, pp. 1383–1398, 1986.

[22] D. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, "SHOP: Simple hierarchical ordered planner," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence table of contents*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1999, pp. 968–975.

[23] M. Mora, J. Lopes, R. Viccari, and H. Coelho, "BDI models and systems: Reducing the gap," in *Proc. of ATAL-98, LNCS*, vol. 1555. Springer.