PLACE: Planning Based Language for Agents and Computational Environments

Muhammad Adnan Hashmi[1], Muhammd Usman Akram[2], and Amal El Fallah-Seghrouchni[3(✉)]

[1] Department of Computer Science, University of Lahore, Lahore, Pakistan
Muhammad.Adnan@cs.uol.edu.pk
[2] Department of Computer Science, COMSATS Institute of I.T. Lahore, Lahore, Pakistan
musmanakram@ciitlahore.edu.pk
[3] Laboratoire d'Informatique de Paris 6, 75016 Paris, France
Amal.ElFallah@lip6.fr

Abstract. We are interested in the development of a language, called PLACE, that allows to program agents as well as their environments. Agents' actions are durative and different priorities can be associated with the goals of agents. The agents autonomously achieve their goals by using the planning mechanism built in the language. The planning mechanism ensures the achievement of higher priority goals before the lower priority goals, but allows to perform low priority actions in parallel to the high priority actions. Plans are repaired if unanticipated changes in the environment cause the plan to become unfeasible. The environment is modeled visually to help the user simulate the behavior of agents and see the execution of agents' plans.

Keywords: Agent oriented programming · Temporal planning
Plan repairing · Environment modeling

1 Introduction

Over the years software agents has proved to be an appropriate paradigm for the development of complex systems that are distributed in nature and require autonomy. In order to tackle the inherent complexity of such systems, three different abstractions have been defined i.e. the agents, the environments and the organizations. This separation of abstractions has lead towards the development of Agent Oriented Programming (AOP) Languages (see e.g. [1–3]), Environment Oriented Programming (EOP) Languages (see e.g. [4,5]), and Organization Oriented Programming (OOP) Languages (see e.g. [6,7]). This paper contributes to the AOP languages as well as to the EOP languages by proposing a language PLACE (Planning based Language for Agents and Computational Environments) that not only facilitates the user to program the agents but also allows him to program the environment as per his requirements. A platform is also

A. El Fallah-Seghrouchni et al. (Eds.): EMAS 2017, LNAI 10738, pp. 142–158, 2018.
https://doi.org/10.1007/978-3-319-91899-0_9

developed that allows PLACE agents and different environment entities to be distributed on different hosts and allows user to visually monitor the interaction of agents with the environment.

PLACE has a syntactic structure close to the Belief-Desire-Intention (BDI) based AOP languages. Most of the BDI based languages (see e.g. Jason [1], 3APL [3]) do not incorporate look-ahead planning. Sometimes the execution of actions without planning results in the inability to achieve the goals as the actions may not be reversible and the executed actions may have used the limited resources. Moreover conflicts could arise among simultaneously executing plans and redundant actions may also be ignored. So, last few years have seen a shift towards the look-ahead planning based approach for a BDI language (see e.g. [8–10]), but these languages do not take into account the duration of agents' 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. So, PLACE tries to fill this gap by using a look-ahead planning based approach, where the agents' actions are durative, priorities have been assigned to goals and plans are repaired if unprecedented changes in the environment cause the plan to become unfeasible. A planning based approach also allows to coordinate the plans of multiple agents which is otherwise difficult in a BDI model, as the BDI model in its definition is a single agent model and it is the responsibility of the programmer to explicitly state the preconditions in order to avoid conflicts among multiple agents.

Plan synthesis in PLACE is done by using the Hierarchical Task Network (HTN) planning [11] techniques. Specifically, we adapt an HTN planner JSHOP2 [12] for computing the plans of agents. HTN planning, as explored by [13], is a natural candidate for planning in the BDI style programming languages. A temporal converter procedure then converts a totally ordered plan generated by JSHOP2 into a parallel position constrained plan[1], where each action is assigned a time stamp and multiple actions can be executed in parallel if possible. The agents have the ability to execute the plans and monitor the execution. Moreover agents are continuously waiting for new tasks from the user, and if the new task requires immediate achievement, the agent preempts the execution of the task currently being executed and immediately plans for and achieves the new task. An important point is that those actions of the current plan that can be executed in parallel with the higher priority task are not postponed, they are executed in parallel and only those actions are postponed which can not be executed in parallel. For this purpose we use the Proactive-Reactive Plan Merging algorithm that we had originally proposed in [14]. The agent constantly monitors the execution of the plans and a plan mender component is added to the language that is used to repair the plan if some unexpected changes in the environment cause the failure of original plan.

[1] Position constrained plans specify the exact start time for each action, whereas order constrained plans just specify the precedence constraints between actions.

In our framework it is easier for the user to monitor the execution of the agents, because the agents' environment is modeled visually. Our environment modeling approach has some similarities with the model of artifacts proposed in [5]. One important difference of our approach to theirs is the fact that in their model an artifact is a physical or computational entity in the environment e.g. a printer, a sensor, a web-service, but in our framework an artifact is the conceptual or logical entity e.g. a train, a room, a city. Another notable difference is that while [5] provides Java APIs for programming the environments, we are presenting a new language with its own syntax and semantics.

Rest of the paper is organized as follows. Section 2 presents some related work. Section 3 discusses the environment modeling and syntactic aspects of PLACE. Planning related issues (planning, plan execution, plan repairing and merging) are presented in Sect. 4. A case study is presented in Sect. 5 which elaborates different planning and environment modeling concepts presented in the paper. Section 6 concludes the paper.

2 Related Work

2.1 Planning Based AOP Languages

Sardina et al. [8] proposed a conceptual framework and agent programming language CanPlan for incorporating HTN planning into a BDI like AOP language. This work is triggered by an earlier work of the authors [13] where they discussed about the similarities among BDI systems and HTN planning framework. CanPlan provides flexibility to the programmers about when to choose full look-ahead planning. The proposed language extends the high level formal agent language Can (Conceptual Agent Notation) [15] by adding more constructs to the language in order to incorporate HTN planning. The additional formal operational semantics for such constructs have been proposed. An important construct that has been added to the language Can is `Plan`. If P is a BDI method body, then `Plan(`P`)` in simple words mean, 'plan for P offline, searching for a complete hierarchical decomposition.' So the BDI agent using `Plan` has to perform a full look-ahead search before the execution commences.

Lesperance et al. [9] takes into account the uncertainty in the environment by proposing contingent planning model in an AOP language. They propose to compute plan, in advance, for different possibilities that can arise during execution of the plan. IndiGolog [16] is another language, in the context of situation calculus, that supports planning by including a deliberation module.

In [10], De Silva et al. proposes a *classical first principles* planning approach for BDI languages to find the plans that are not currently available in the plan library. The plans generated by their approach are called *hybrid plans*, and may contain abstract operators that can be mapped back to the goals, thus allowing the agent to execute the plan using its BDI plan library. Another framework incorporating classical planning in a BDI language is presented in [17]. It extends the X-BDI [18] model to use the propositional planning algorithms for performing

means-end reasoning. It is a rather theoretical work concerning the mapping of BDI internal mental states to a STRIPS like notation and back.

In another work [19], Chaouche et al. proposes a concrete software architecture in the domain of Ambient Intelligence, that incorporates contextual planning in order to synthesize plans for agents taking into consideration the current and future context. In [20] they take this work forward by endowing the agents with the ability to learn the future context from the previous experiences of actions.

2.2 Environment Modeling in AOP Languages

Environment modeling is a core ingredient for any agent oriented programming language. Researchers in multi-agent systems community have long been working on the lines of AEIO approach (Agent, Environment, Interaction, Organization) given in [21]. Any agent oriented programming language should be able to represent the agents and environment and most importantly the interaction between agents and the environment in which they reside.

An Action and Perception model sense-plan-act (SPA) for the exogenous environments has been proposed in [22] where the interaction of an agent with its environment is a three step process. In the first step the agent perceives its environment updating its beliefs, in the second step it plans the actions to be carried out and lastly it performs the actions causing the environment to be changed. This work is inspired by the author's previous work of simpA [23] which is agent-oriented approach for programming concurrent applications on top of Java. simpA is a theoretical framework which is later implemented in the form of CArtAgO [5]. CArtAgO has introduced a computational notion of artifacts called A&A (Agent and Artifact) to design and implement agent environments. Artifact based environment modeling has been successfully used for designing multi-agent systems for data mining domains [24]. CArtAgO is an annotation-based framework built on top of the basic Java environment. On the principles of artifacts [25] has proposed a unified interaction model with Agent Organization, and Environment.

Another approach is presented by [26] to model the Multi-Agent Based Social Simulations (MABS) in which a virtual environment is partitioned into areas called cells and is supported by an underlying autonomic software system consisting of specialized agents called controllers and coordinators. Controllers manage specific cells while coordinators monitor and guide controllers in the execution of their tasks. In such an approach the abstraction level for underlying complex environment is provided. This approach is somewhat similar to previously discussed artifact based approach, the difference is that the management of artifacts or cells is handled through dedicated agents. Another variation of designing intelligent environments is proposed by [27] in which the designer is interested in delegating a part of the agent's tasks to its body.

For context-aware agents [28] has proposed a model for the interaction between context-aware virtual agents and the environment. This work emphasizes the use of extensible agent perception module, allowing agents to perceive

their environment through multiple senses. Perception combination for an agent is further investigated in [29] which proposes a multi-sense perception system for virtual agents situated in large scale open environments for the DIVA [30] project.

There are many different models for environment representation which are in use. There is a strong need to establish the uniformity of the environment's representation and the agents. In order to bridge this gap, [31] have proposed an interface for the environment which is discrete in space (grid-world) and time (step-wise evolution). This is an extra layer introduced between environments and agents so that different platforms could be able to interact with same environment. This new language is called Interface Intermediate Language (IIL) providing a conventional representation for actions and percepts. Another such approach is presented by [32] for the modeling of Agent-Environment Interactions in adaptive MAS. In this work, the interaction levels are further broken down in multiple abstraction layers.

There are many other computational frameworks for environment modeling. AGRE [33] integrates the AGR (Agent-Group-Role) organizational model with a notion of environment. In AGR the agents are considered to be working in groups in a space which contains them. AGRE is based on MadKit platform [34]. Problem with the AGR is that it is only suitable for geometrical environments. Logic based frameworks are also introduced such as GOLEM [35] to represent environments for situated cognitive agents. But these frameworks do not account for the mobility of the agents as the environment is modeled through objects. Logic based environments are well suited for game intelligence.

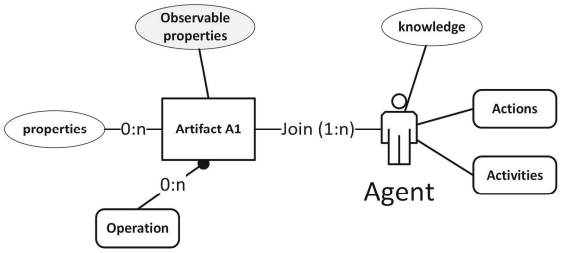


Fig. 1. PLACE Meta-Model

3 Syntax and Environment Modeling in PLACE

In PLACE we are concerned with two aspects of the environment programming: physical and logical. In the physical aspect we are concerned with the actual deployment of the agent i.e. the host and the network on which the agent is deployed etc. For this purpose, we have borrowed some ideas from AOP language CLAIM [2]. Like CLAIM, we have a central system in PLACE to which all the

Listing 1.1. PLACE Agent Definition

```
defineAgent agentName {
      knowledge = null; | { (knowledge;)+ }
      goals = null; |{ (goal;)+ }
      actions = null; | { (action)+ }
      activities = null; | { (activity)+ }
      environment = null; | { environmentName }
      agentIn = null; | { artifactName }
      artifacts = null;| { (artifactName;)+ }
}
```

Listing 1.2. PLACE Artifact Definition

```
defineArtifact artifactName {
      parent = null; | {artifactName}
      connectedTo = null; | { (artifactName[{notBreakable}];)+ }
      environment = null; | { environmentName }
      properties = null; | { (property[{notChangeable}][observableTo = {(
          artifactName;)+}];)+ }
      operations = null; | { (operation)+ }
}
```

hosts are connected. The deployment of agents is then inside those hosts. The agents can join environment at any host. Physical mobility is possible for agents from one host to another, but in the visual modeling this physical mobility is transparent to the user, because he is more concerned with the logical mobility. In the logical aspect of environment, we are concerned with the environment as perceived by the user i.e. the user is perceiving the agent inside a train in the Paris city. For this purpose, our work has some similarities with the Agent and Artifact (A & A) meta-model of environment [5]. Artifact is an entity which presents the functionalities and knowledge to the agents. Figure 1 shows the meta model of the PLACE environment modeling. An environment in PLACE can contain multiple artifacts and agents within it. One artifact can contain multiple artifacts and can host multiple agents but an agent can be situated only in one artifact.

There are three different building blocks of PLACE language i.e. Agents, Artifacts and Environments. Following sections describe these three aspects of PLACE in detail.

3.1 Agents in PLACE

An agent is defined as shown in Listing 1.1. Its components are described as follows:

- **knowledge** of the agent is what it believes about the world at a certain moment. It can be described with the help of first order propositions containing a name and list of arguments. Initially the knowledge can be empty or given by the programmer but it evolves over time as the agent executes its planned actions.

- **goals** represent current goals of the agent. The designer can give goals in the form of the tasks to be performed. Goals can be of high priority or low priority and it is the responsibility of planning and reasoning mechanism to ensure that high priority goals are achieved before low priority goals. A goal can be defined as:
  *goal = {proposition [**, high**|**, low**]}*
- **actions** in PLACE are the primitive tasks that an agent is capable of carrying out. Actions are the way through which an agent can interact with the environment and manipulate it. If the agent has desired knowledge at the time of execution of an action, then the action adds/removes some knowledge to/from agent's belief base, hence modifying agent's beliefs. Actions in PLACE are durative in nature. Some actions are pre-defined in the language e.g. an agent can move from one artifact to another by using the pre-defined action *move(?source, ?destination)*. An action can be defined as:

  action = actionSignature {
  *[**preconditions**=precondition]*
  *[**add_effects**= {proposition (,proposition)+ }]*
  *[**del_effects**= {proposition (,proposition)+ }]*
  *[**duration**=number;]*
  }

  Precondition is a collection of knowledge that an agent must have at the time of action's execution. A precondition can be a function that returns a *boolean*, a condition about agent's knowledge, a condition about being in a particular artifact, a condition about possession of an artifact, a condition about an artifact being inside another artifact, a condition about connection of an artifact to another artifact or a conjunction of any of these:

  precondition = function(args) | ***hasKnowledge****(knowledge)* | ***agentIn****(artifactName)*
  | ***hasArtifact****(artifactName)* | ***artifactIn****(artifactName, artifactName)*
  | ***connectedTo****(artifactName, artifactName)* | ***and****(precondition (,precondition)+)*

- **activities** are the short plans that a designer can provide to the agent. Unlike actions, activities do not add or delete any knowledge in agent's knowledge base.

  activity = activitySignature {
  *[**preconditions**= precondition]*
  ***do** { ActivityActionSequence }*
  }

  Activity's precondition is somewhat different from an action's precondition. Here the developer can use *and*, *or* and *not* operators and in any nested way he wants. The *do* element of an activity is the sequence of action or activity calls in the form of messages, that is in fact the short plan that designer is supposed to provide.
- **environment** represents the environment in which agent is situated. PLACE agent can only be the part of a single environment at any time but the agent's designer can change the information of the environment in one of its actions

to move agent from one environment to another if the agent is required to perform its tasks in multiple environments.
- **agentIn** represents the name of artifact in which the agent is currently situated.
- **artifacts** are the artifacts currently possessed. When an agent moves from one place to another, it moves with all the artifacts that he currently possesses.

3.2 Artifacts in PLACE

Listing 1.2 shows, how an artifact can be defined in PLACE. Artifacts in PLACE environment present operations, properties and observable properties. Its components are described as follows:

- **parent** describes the artifact in which the artifact is currently situated.
- **connectedTo** represents a list of artifacts to which the artifact is connected. In order to model the static parts of the environment, a connection can be labeled *notBreakable* for the cases where designer wants to forbid agents from breaking connections.
- **properties** of an artifact is the knowledge which is directly accessible to the agent which is currently situated in that artifact. An artifact's property can be labeled *notChangeable* to forbid any agent from modifying it. Observable properties are accessible to all those agents which are situated in those artifacts to which the property is observable, by using the label *observableTo*.
- **operations** are the way for an artifact to expose its functionalities. Any agent attached to the artifact can perform the operation unless the artifact allows certain operations to some specific agents with the help of a keyword *agents*. On the agent's part, a boolean function *allowed* is used to check whether an agent is allowed to perform a certain operation or not.

```
operation = operationSignature {
   [agents=agentName (,agentName)+]
   [concurrent=number | agentName (,agentName)+]
   [preconditions=precondition]
   [add_effects_artifact= {proposition (,proposition)+ }]
   [del_effects_artifact= {proposition (,proposition)+ }]
   [add_effects_agents= {proposition (,proposition)+ }]
   [del_effects_agents= {proposition (,proposition)+ }]
   [duration=number;]
}
```

Operations has one-to-one mapping with actions of agent if the agent is situated inside the artifact. Operations are the only way through which an artifact's properties can be changed. Agents can also move the artifacts by changing their parent information, if such operation is allowed to the agent by the artifact. Sometimes an operation can only be performed concurrently by two or more than two agents. This is achieved through keyword *concurrent*. It can be the number of agents, or the names of agents that should perform a certain operation concurrently. Concurrency issues can quickly arise

Listing 1.3. PLACE Environment Definition

```
defineEnvironment environmentName{
      properties = null; | { (property;)+}
      operations = null; | { (operation)+}
}
```

in such meta-model as multiple agents are linked to the same artifact and they can perform the operations concurrently. In order to resolve this issue, the operations of PLACE environment are thread locked and only one agent can perform operation at any given time. When the designer uses keyword *concurrent* inside an operation, the thread locked capability is turned off and the environment designer is supposed to provide preconditions accordingly.

When an operation is performed over an artifact, some facts are added/removed from the knowledge of those agents who are performing that operation. This is specified by keywords *add_ effects_ agents* and *del_ effects_ agents*. Moreover some properties of the artifact may also get changed, this is specified by keywords *add_ effects_ artifact* and *del_ effects_ artifact*.

3.3 Environment in PLACE

An environment can be defined as shown in Listing 1.3. Environment presents global knowledge to all the agents situated in the environment as the form of environment properties, through keyword *properties*. It is like a shared memory for the agents to facilitate them in communicating in decentralized manner. The global knowledge can only be accessed by the keyword *global*. Agents knowledge is thus a union of its own knowledge, global knowledge provided by the environment, artifact's knowledge and the knowledge of the artifacts which are observable from the artifact in which the agent is situated. One advancement in PLACE from the earlier version is that now the environment can also offer operations to the agent's designer. A designer now have the flexibility to design the global operations on the environment which would be available to all agents if the operation's agent property is left blank. For the cases in which designer wants to write global operations for some selected agents, he can do so by mentioning the names of the agents in individual environment operation.

3.4 Deployment and Visual Environment Modeling

The physical deployment is concerned with how the agents and artifacts are distributed among hosts on the network as shown in Fig. 2 (right), whereas the logical deployment deals with how the agents perceive the environment. It shows the whole system hosted on one virtual machine as shown in Fig. 2 (left). The physical mobility of agents in between hosts is hidden from the user. The logical deployment of agents and artifacts is represented by an Environment Graph, and can be viewed by user at any time during the execution of agent on the Graphical User Interface of PLACE. In an environment graph, the artifacts are

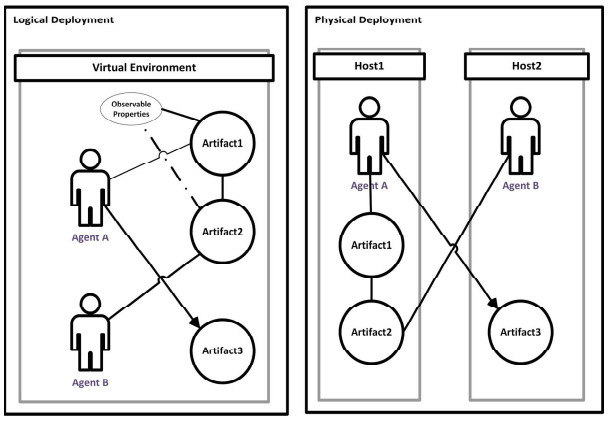


Fig. 2. Physical and Logical Deployment of PLACE Agents and Environment

represented with circle notations whereas agents are represented with a standard user like figure. The connected artifacts are shown with the help of a straight solid line e.g. Artifact1 and Artifact2 are connected. An agent can also possess artifacts which is shown with the help of a directed solid line e.g. Agent A is holding Artifact3. If an artifact is in the possession of an agent then it can only be connected with those artifacts which are in the possession of same very agent. An artifact's observable properties are shown with an oval connected to the artifact with a solid line. An artifact's observable properties can be observed by the agents from other artifacts to which these properties are observable to e.g. Agent B is inside Artifact2 and it can see the Artifact1's observable properties. These properties are linked with the artifacts with a dashed line.

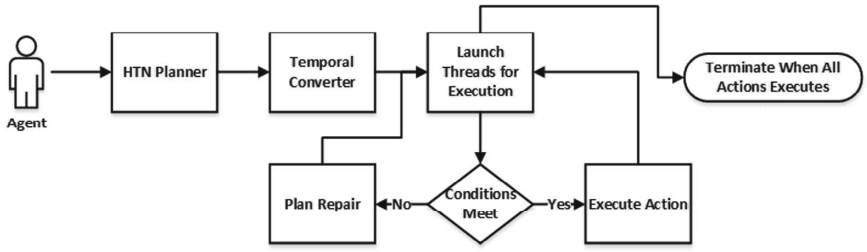


Fig. 3. PLACE Planning Work Flow

4 Planning for PLACE Agents

Planning is the core component of PLACE agents. Agents once launched in an environment are supposed to act intelligently without the intervention of designer. PLACE is equipped with built-in planning capability which is reusable

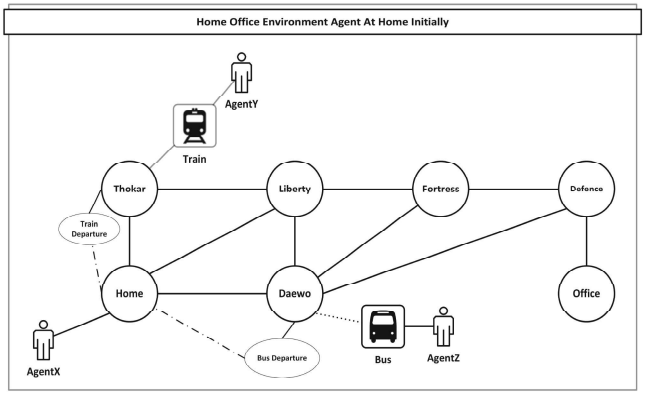
and extendable in its nature. Figure 3 shows a work flow of PLACE planning process. At first the agent invokes an HTN planner to create a total order plan. A total order plan is the plan which is computed to generate a sequence of actions in exact order in which they are to be carried out in order to achieve the goals. For the purpose of total order planning we have chosen JSHOP2 planner in our framework. The reason to choose JSHOP2 for PLACE agents is two fold. Firstly, it is an HTN planner and the domain and problem information from PLACE can easily be translated to the domain and problem information required by JSHOP2 planner due to the similarities among HTN and BDI agent architectures [13]. Secondly, JSHOP2 plans in the same order in which the actions would be executed later e.g. JSHOP2 knows at each step the current state of the agent. PLACE agents are reactive in nature making them responsive to any change in environment. At any stage if a change occurs in the environment or the agent is given a new goal, the planner can then incorporate that change easily.

The actions in PLACE are temporal in nature e.g. each action has associative time duration. In second phase of the planning, the total order plan is converted into a parallel position constrained plan where each action is assigned a time stamp and multiple actions can be executed in parallel if possible. Temporal converter takes a total order plan as an input and specifies the start time for each action in such a way that the actions which can be carried out in parallel are scheduled with overlapping time windows.

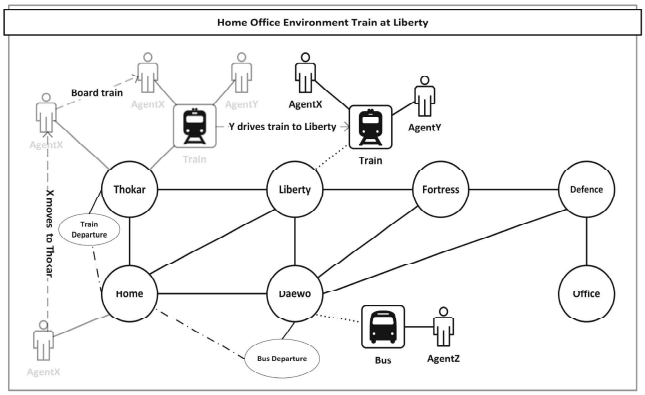
After the planning is completed the sequence of actions generated is now passed to the executioner which launches a separate thread for each action at its start time. If the action's condition is met then the action is executed e.g. add effects are added and delete effects are deleted from agent's knowledge base and control returns to executioner with success message. If the condition is not met due to some influence of another agent or designer then the control is shifted to plan repair module.

The main idea of plan repair algorithm is that if the state of world has been changed by some unanticipated event, and the preconditions of an action A no longer hold in the world, remove this action from the plan and try to compute a temporal plan from the current actual world state to a state where all the necessary effects of A hold. So, Plan Mender algorithm uses the well-known planner Sapa [36] to compute a temporal plan from the current actual world state to a state which has all the necessary effects of A. If such a plan is possible, it is returned as a replacement for the removed action. Otherwise, in the next iteration, the next action B of plan is also removed and algorithm tries to compute a plan for the achievement of the goals of the previous iteration, minus the preconditions of B, union the necessary effects of B. If such a plan is possible, it is returned as a replacement for the actions A and B. This gap is gradually widened, unless a plan is found or there is no more action to remove from the plan.

It is worth noting that the agent is receiving the goals *on the fly* and using the same procedure of planning for each goal. If the returned plan is for some reactive goal, it is merged at the beginning of agent's global plan. We had presented a



(a) Case Study Initial State



(b) Case Study Execution

Fig. 4. Case Study

plan merging algorithm in [14], so here we are just making use of that algorithm without going into its details. If the returned plan is for some low priority goal, it is appended at the end of agent's global plan.

When all the planned actions are executed then the process of a single agent planning and execution terminates, and the agent waits for new goals.

5 Case Study

In order to demonstrate the capabilities of environment modeling and planning in PLACE, lets consider the case study presented in Fig. 4(a). There are three agents AgentX, AgentY and AgentZ in the environment. Moreover, there are seven artifacts representing places, namely Home, Thokar, Liberty, Fortress, Defence, Daewoo and Office. In addition, there are two artifacts representing vehicles, namely Train and Bus. AgentX is situated at Home and its goal is

Listing 1.4. PLACE: Environment & Artifacts Representation

```
defineEnvironment HomeOffice {
  properties = null;
  operations = null;
}

defineArtifact Thokar{
  environment = {HomeOffice}
  connectedTo = {Liberty,Home}
  parent = null;
  properties = {trainDepartureTime(this, 15){obseravableTo={Home}};}
  operations={
    sellTicket(?a){
      preconditions = {agentIn(?a,this)}
      add_effects_agents = {hasTrainTicket()}
        }
  }
}
defineArtifact Train{
  environment = {HomeOffice}
  connectedTo = null;
  parent={Thokar}
  operations={
      driveTo(?s,?d,?a){
      agents = {AgentY}
      preconditions = {
        and(agentIn(?a,this),
        artifactIn(this,?s),
        trainDepartureTime(?s,?t),
        >=(java.currentTime(),?t))
      }
      add_effects_artifact = {artifactIn(this,?d)}}
    }
  }
}
```

to be in Office. The Train is situated in Thokar. AgentY is a driver which is present inside the Train and only he can drive train to stations Liberty, Fortress and Defence. Similarly Daewoo is a bus-stand which has a Bus situated inside it and AgentZ is its driver who can drive the bus to Defense.

A snippet of code for artifacts Train and Thokar is given in Listing 1.4 and a snippet of code for AgentX is given in Listing 1.5. While the AgentX is at Home it can see the knowledge of train departure time as the artifact Thokar has an observable knowledge *trainDepartureTime* to Home artifact.

At home the AgentX has three options to start its travel. It can move to the train station Thokar from where it can board the train after purchasing the ticket, or it can move to the bus-stand Daewoo and board the bus, or it can move to the train station Liberty and board the train from there if it has missed the train from Thokar. The agents AgentY and AgentZ are the driver agents which can move train and bus respectively. The *activities* of AgentX, that would help him in planning, are not shown for space reasons.

Suppose the AgentX generates a plan to move to Thokar and then to the Train, and AgentY moves the Train from station Thokar to Liberty then to station Fortress and Defence, from where the AgentX can move to its Office. The AgentX's execution of the plan upto Liberty is shown in Fig. 4(b) as shaded path.

Listing 1.5. PLACE: Agent Representation

```
defineAgent AgentX{
  agentIn = {Home}
  artifacts = null;
  environment = {HomeOffice}
  knowledge = null;
  goal = {{agentIn(this,Office)};}
  actions = {
  moveTo(?s,?d){
    preconditions ={and(agentIn(this,?s), connectedTo(?s,?d))}
    add_effects={agentIn(this,?d)}
    }
  }
}
```

Let us consider a scenario in which the station Fortress is under construction and the Train cannot continue its journey, then at this point the AgentX calls the Plan Mender to repair the plan. A new plan is given to AgentX to move from Liberty to Daewoo and then go to Defense by Bus.

6 Conclusion

We have presented an AOP language that endows agents with the capability to plan ahead and also facilitates the user to visualize the behavior of agents through environment modeling. The presented language is called PLACE (Planning based Language for Agents and Computational Environments). 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. Currently, PLACE is not handling the goals with deadlines. Moreover, the time duration for an action is static i.e. it is not dependent on other parameters e.g. distance. We are working on these improvements in the language. We are also investigating the use of heuristics to guide the search for better plans in lesser time.

References

1. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of agent-oriented programming. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming. MASA, vol. 15, pp. 3–37. Springer, Boston (2005). https://doi.org/10.1007/0-387-26350-0_1
2. El Fallah-Seghrouchni, A., Suna, A.: CLAIM: a computational language for autonomous, intelligent and mobile agents. In: Dastani, M.M., Dix, J., El Fallah-Seghrouchni, A. (eds.) ProMAS 2003. LNCS (LNAI), vol. 3067, pp. 90–110. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25936-7_5
3. Dastani, M., van Riemsdijk, M.B., Meyer, J.-J.C.: Programming multi-agent systems in 3APL. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-agent Programming. MASA, vol. 15, pp. 39–67. Springer, Boston (2005)

4. Weyns, D., Omicini, A., Odell, J.: Environment as a first class abstraction in multiagent systems. Auton. Agents Multi-agent Syst. **14**(1), 5–30 (2007)
5. Ricci, A., Piunti, M., Viroli, M.: Environment programming in multi-agent systems: an artifact-based perspective. Auton. Agents Multi-Agent Syst. **23**(2), 158–192 (2011)
6. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. Int. J. Agent-Oriented Softw. Eng. **1**(3), 370–395 (2007)
7. Kitio, R., Boissier, O., Hübner, J.F., Ricci, A.: Organisational artifacts and agents for open multi-agent organisations: "Giving the Power Back to the Agents". In: Sichman, J.S., Padget, J., Ossowski, S., Noriega, P. (eds.) COIN -2007. LNCS (LNAI), vol. 4870, pp. 171–186. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79003-7_13
8. Sardina, S., de Silva, L., Padgham, L.: Hierarchical planning in BDI agent programming languages: a formal approach. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1001–1008. ACM, New York (2006)
9. Lespérance, Y., De Giacomo, G., Ozgovde, A.N.: A model of contingent planning for agent programming languages. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, vol. 1, pp. 477–484. International Foundation for Autonomous Agents and Multiagent Systems (2008)
10. De Silva, L., Sardina, S., Padgham, L.: First principles planning in BDI systems. In: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, vol. 2, pp. 1105–1112. International Foundation for Autonomous Agents and Multiagent Systems (2009)
11. Erol, K., Hendler, J., Nau, D.S.: HTN planning: complexity and expressivity. In: Proceedings of the National Conference on Artificial Intelligence, pp. 1123–1123. Wiley (1995)
12. Nau, D., Au, T.C., Ilghami, O., Kuter, U., Murdock, J.W., Wu, D., Yaman, F.: SHOP2: an HTN planning system. J. Artif. Intell. Res. **20**(1), 379–404 (2003)
13. de Silva, L., Padgham, L.: A comparison of BDI based real-time reasoning and HTN based planning. In: Webb, G.I., Yu, X. (eds.) AI 2004. LNCS (LNAI), vol. 3339, pp. 1167–1173. Springer, Heidelberg (2004)
14. Hashmi, M.A., El Fallah Seghrouchni, A.: Coordination of temporal plans for the reactive and proactive goals. In: 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, pp. 213–220. IEEE (2010)
15. Winikoff, M., Padgham, L., Harland, J., Thangarajah, J.: Declarative and procedural goals in intelligent agent systems. In: International Conference on Principles of Knowledge Representation and Reasoning. Morgan Kaufman (2002)
16. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: a high-level programming language for embedded reasoning agents. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming, pp. 31–72. Springer, Boston, MA (2009). https://doi.org/10.1007/978-0-387-89299-3_2
17. Meneguzzi, F.R., Zorzo, A.F., da Costa Móra, M.: Propositional planning in BDI agents. In: Proceedings of the 2004 ACM Symposium on Applied Computing, pp. 58–63. ACM (2004)
18. Mora, M.C., Lopes, J.G., Viccariz, R.M., Coelho, H.: BDI models and systems: reducing the gap. In: Müller, J.P., Rao, A.S., Singh, M.P. (eds.) ATAL 1998. LNCS, vol. 1555, pp. 11–27. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-49057-4_2

19. Chaouche, A.-C., El Fallah Seghrouchni, A., Ilié, J.-M., Saïdouni, D.E.: A higher-order agent model with contextual planning management for ambient systems. In: Kowalczyk, R., Nguyen, N.T. (eds.) Transactions on Computational Collective Intelligence XVI. LNCS, vol. 8780, pp. 146–169. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44871-7_6
20. Chaouche, A.-C., El Fallah Seghrouchni, A., Ilié, J.-M., Saïdouni, D.E.: Improving the contextual selection of BDI plans by incorporating situated experiments. In: Chbeir, R., Manolopoulos, Y., Maglogiannis, I., Alhajj, R. (eds.) AIAI 2015. IAICT, vol. 458, pp. 266–281. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23868-5_19
21. Demazeau, Y.: From interactions to collective behaviour in agent-based systems. In: Proceedings of the 1st European Conference on Cognitive Science, Saint-Malo. Citeseer (1995)
22. Ricci, A., Santi, A., Piunti, M.: Action and perception in agent programming languages: from exogenous to endogenous environments. In: Collier, R., Dix, J., Novák, P. (eds.) ProMAS 2010. LNCS (LNAI), vol. 6599, pp. 119–138. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28939-2_7
23. Ricci, A., Viroli, M., Piancastelli, G.: simpA: an agent-oriented approach for programming concurrent applications on top of Java. Sci. Comput. Program. **76**(1), 37–62 (2011)
24. Limón, X., Guerra-Hernández, A., Cruz-Ramírez, N., Grimaldo, F.: An agents and artifacts approach to distributed data mining. In: Castro, F., Gelbukh, A., González, M. (eds.) MICAI 2013. LNCS (LNAI), vol. 8266, pp. 338–349. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45111-9_30
25. Zatelli, M.R., Hübner, J.F.: A unified interaction model with agent, organization, and environment. Anais do IX ENIA@ BRACIS, Curitiba, Brazil (2012)
26. Al-Zinati, M., Wenkstern, R.: A self-organizing virtual environment for agent-based simulations. In: Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems, pp. 1031–1039. International Foundation for Autonomous Agents and Multiagent Systems (2015)
27. Saunier, J.: Bridging the gap between agent and environment: the missing body. In: International Workshop on Environments for Multiagent Systems (E4MAS 2014). IFAAMAS. Springer, Paris (2014)
28. Steel, T., Kuiper, D., Zalila-Wenkstern, R.: Context-aware virtual agents in open environments. In: 2010 Sixth International Conference on Autonomic and Autonomous Systems (ICAS), pp. 90–96. IEEE (2010)
29. Kuiper, D.M., Wenkstern, R.Z.: Virtual agent perception combination in multi agent based systems. In: Proceedings of the 2013 International Conference on Autonomous Agents and Multiagent Systems, pp. 611–618. International Foundation for Autonomous Agents and Multiagent Systems (2013)
30. Vosinakis, S., Anastassakis, G., Panayiotopoulos, T.: Diva: distributed intelligent virtual agents. Behaviour **3**, 5 (1990)
31. Behrens, T., Hindriks, K.V., Bordini, R.H., Braubach, L., Dastani, M., Dix, J., Hübner, J.F., Pokahr, A.: An interface for agent-environment interaction. In: Collier, R., Dix, J., Novák, P. (eds.) ProMAS 2010. LNCS (LNAI), vol. 6599, pp. 139–158. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28939-2_8
32. Mili, R.Z., Steiner, R.: Modeling agent-environment interactions in adaptive MAS. In: Weyns, D., Brueckner, S.A., Demazeau, Y. (eds.) EEMMAS 2007. LNCS (LNAI), vol. 5049, pp. 135–147. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85029-8_10

33. Ferber, J., Michel, F., Baez, J.: AGRE: integrating environments with organizations. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2004. LNCS (LNAI), vol. 3374, pp. 48–56. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32259-7_2
34. Gutknecht, O., Ferber, J.: The MadKit agent platform architecture. In: Wagner, T., Rana, O.F. (eds.) AGENTS 2000. LNCS (LNAI), vol. 1887, pp. 48–55. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47772-1_5
35. Bromuri, S., Stathis, K.: Situating cognitive agents in GOLEM. In: Weyns, D., Brueckner, S.A., Demazeau, Y. (eds.) EEMMAS 2007. LNCS (LNAI), vol. 5049, pp. 115–134. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85029-8_9
36. Do, M.B., Kambhampati, S.: Sapa: a domain-independent heuristic metric temporal planner. In: Proceedings of ECP-01, pp. 109–120 (2001)