Abstract

Over the last decade the popularity of agent-based systems have increased rapidly because agents bring intelligence, reasoning and autonomy to software systems. Recent advances in middleware and run-time systems have helped in designing agent-based systems. However, little work has been reported in defining a software architecture, modeling and analysis tools that can be used by software engineers. In this paper, we present a framework for modeling, analysis and construction of agent-based systems. The framework is an extension of the Unified Modeling Language (UML) to include a new classifier called Agent which is modeled by three abstract types called Belief, Goal and Plan, respectively. In addition, we introduce Goal Diagram to depict the relationships between use cases and goals, Inter-Agent Sequence Diagram and Intra-Agent Sequence Diagram to model interactions between agents and within an agent. We illustrate the framework through an agent-based intelligent elevator system.

1. Introduction

Over the last decade the popularity of agent-based systems have increased rapidly because agents bring intelligence, reasoning and autonomy to software systems. Agents are being used in an increasingly wide variety of applications from simple e-mail filter programs to complex mission control and safety critical systems including air traffic control and nuclear power plant operations. Recent advances in middleware and run-time systems have helped in designing such agent-based software systems. However there appears to be very little work in defining software architecture, modeling and analysis tools that can be used by software engineers. This should be contrasted with object-oriented paradigm that is supported by modeling languages such as UML and a variety of CASE tools that aid during the analysis, design, implementation and validation phases of object-oriented software systems: all of which contributed to the universal acceptance of object-oriented paradigm. Only recently there have been a few proposals for Agent-Oriented Software Engineering and extensions to UML (e.g., AUML). However AUML addresses only the interactions among agents and do not facilitate the representation of reasoning and proactive nature of agent-oriented systems. In this paper, we propose a framework and extensions to UML to address this need. Our approach is rooted in the BDI formalism [22, 23, 24], but stresses practical software design methods instead of reasoning theories. In particular, we propose to extend UML [2] with a new classifier, called Agent, which is an abstract class and the superclass for all agent types. Three primitive abstract types called Belief, Goal and Plan, are introduced to model the reactive and proactive behaviors of agents. An agent is defined by a set of beliefs, goals and plans. In other words, an agent consists of, among other data types, a collection of beliefs, goals and plans. Beliefs are the agent’s observations and/or sensing of the environment and are updated by sensors or other agents. Changes in an agent’s beliefs affect the goals of the agent which need to be re-evaluated. Changes to the goals result in pre-empting some plans and initiating new plans. Execution of plans affects the environment which in turn changes the beliefs, and so on. Application-specific beliefs, goals and plans are defined by subclassing to the abstract Belief, Goal and Plan to provide application de-

1To be implemented as abstract classes or interfaces.
posed an architecture-centric design method based on ex-
supports automatic code generation. Yim et al. [32] pro-
Methodology (MaSE) [6] [29] is similar to Gaia but also
abilities are static. The Multiagent Systems Engineering
society modeling. It assumes that agent relationships and
translates the model into relational database schemes. The
[28] is based on the Entity-Relationship (ER) model [3] and
actions. The Agent-Object-Relationship (AOR) approach
modeling unconditional, inclusive-or and exclusive-or inter-
constraints on messages. Also introduced are notations for
interaction Protocol (AIP) for agent communication and
approach proposed by Odell et al. [20] introduced the Agent
mentation of the BDI formalism. The agent UML (AUML)
activity indicates that agents not only react, but also exhibit
proactive. Reactivity implies that agents must take timely
agents must be flexible in the sense that they should be both reactive and
liefs. Autonomy implies that agents should operate without
agents. In [13] the authors define agents with three con-
cepts: situatedness, autonomy and flexibility. The situated-
tional Reasoning System (PRS) [9] [11] is a pioneer imple-
ementation of the BDI formalism. The agent UML (AUML)
approach proposed by Odell et al. [20] introduced the Agent
Interaction Protocol (AIP) for agent communication and
constraints on messages. Also introduced are notations for
modeling unconditional, inclusive-or and exclusive-or inter-
actions. The Agent-Object-Relationship (AOR) approach
[28] is based on the Entity-Relationship (ER) model [3] and
translates the model into relational database schemes. The
Gaia Methodology [30] supports agent structure and agent
society modeling. It assumes that agent relationships and
abilities are static. The Multiagent Systems Engineering
Methodology (MaSE) [6] [29] is similar to Gaia but also
supports automatic code generation. Yim et al. [32] pro-
posed an architecture-centric design method based on ex-
tensions of UML and translates agent-oriented modeling to
OO modeling using design patterns [8]. Bergenti and Poogi
[1] treat agents as communicating entities. Similar to Yim et
al, this approach uses UML to model MAS and requires no
extensions to UML. Finally, Rumbaugh et al’s Object Mod-
eling Technique (OMT) was adapted by Kinny et al. [14]
to model agents in an Agent Model of inheritance hierar-
chy. In addition, a Belief Model, a Goal Model and a Plan
Model were proposed to specify the beliefs, goals and plans
for the agents.

Our approach is similar to existing approaches that ex-
tend UML. It is based on the BDI structure and introduces
the abstract classes Belief, Goal, Plan and others and the
abstract classifier Agent. This allows implementation of
application specific behavior. For example, a belief instance
can be a knowledge base which provides intelligent reason-
ing capabilities. While Kinny et al’s approach translates
the Belief, Goal, Plan and Agent Models to formal models
like BDI, our approach aims at providing a modeling ap-
proach for MAS that can accommodate various application
specific design and implementation alternatives. With re-
spect to modeling of agent interactions our approach uses
O’Dell et al’s notations described above.

A survey of the afore mentioned methodologies and other methodologies can be found in [26] and [10].

3. Background

3.1. Agents Versus Objects

An object is an entity that encapsulates state and behav-
ior while a class is a template from which objects can be
created. Every object is an instance of a class. Object-
oriented programming supports inheritance, which permits
new classes to be derived from existing classes simply by
modifying or extending the features of the superclass.

There is no accepted definition of what “agent-based”
or “agent-oriented” programming means. However there is
a generally accepted list of characteristics associated with
agents. In [13] the authors define agents with three con-
cepts: situatedness, autonomy and flexibility. The situated-
ness implies that agents receive input from an environment
and perform actions that may change the environment. To
distinguish from real-time control software, an agent may
decide to ignore an input event based on its goals and be-
liefs. Autonomy implies that agents should operate without
intervention of human beings or other agents. Agents must
be flexible in the sense that they should be both reactive and
proactive. Reactivity implies that agents must take timely
actions in response to changes in the environment. Proac-
tivity indicates that agents not only react, but also exhibit
goal-oriented behavior.
OO programming encourages encapsulation of object state and behavior but it is sometimes desirable to share beliefs among agents via a knowledge base or a blackboard. Agents have reactive as well as proactive goals and behaviors. The behavior of an agent may be different at different times and may be non-deterministic. Agents may inherit plans (which are similar to methods), beliefs (which are similar to instance variables) and goals (for which there is no direct counterpart in OO). The similarities and differences between the two paradigms will become clear from our framework.

3.2. BDI Formalism

The BDI architecture associates with agents, beliefs (typically about the environment and other agents), desires or goals to achieve, and intentions or plans to act upon to achieve its desires. In formal terms, one can utilize logic to describe these components and reason about MAS. In practical terms, beliefs can be viewed as the state of the world. Beliefs may be represented as simple variables and data structures or, complex systems such as knowledge-bases. Desires (or goals) may be associated with a value so that desires can be prioritized. In formal terms, desires can be evaluated using “path formulas” whereby all possible paths associated with a desire can be evaluated. In practical terms, evaluation functions can be used to dynamically update goal values. Intentions reflect the actions that must be exercised to achieve the goal values. Thus intentions indicate the actions along a path formula in the decision tree used to compute goal values. In our view, the BDI formalism may be used to model intelligent, autonomous, situated agents as shown in Figure 1. In general beliefs may be shared and modified by other agents. This can be achieved either by direct communication using KQML [7] or FIPA [4] messages, shared knowledge-bases or blackboards (e.g., Linda or its extensions such as LIME [19], [21]). Plans can be proactive or reactive – proactive plans reflect the desires or goals of an agent. These goals may impact how an agent reacts to external events (including the possibility of ignoring external stimuli). Reactive plans reflect how an agent can be situated in an environment.

3.3. The Unified Modeling Language (UML)

The Unified Modeling Language (UML) [2] is widely used in the object-oriented (OO) paradigm [16] for software requirements analysis and software design. The OO paradigm views an application as consisting of concurrent, communicating objects, each of which has states and state dependent behaviors. UML provides nine types of diagrams for modeling different aspects of a system. A Use Case Diagram specifies how a user interacts with the system. A Class Diagram defines the object types and their relationships. A Sequence Diagram describes object interaction along a timeline. A Statechart Diagram models the state dependent behaviors of objects and an Activity Diagram depicts both sequential and concurrent activities.

While UML is an excellent language for object-oriented (OO) systems, it lacks the capability to readily model and specify multi-agent systems. This is due to the fundamental differences between OO systems and agent-based systems (Section 3.1). Objects are passive components whereas agents are autonomous. Objects are reactive whereas agents are also situation-aware and proactive.

4. Extending UML for Multi-Agent Systems

The main motivation for developing the framework is to provide a MAS modeling language to application engineers. We introduce an abstract classifier called Agent and several abstract types including Belief, Goal and Plan to support the BDI structure. In the OO paradigm, an abstract class defines a type that must be subclassed by a concrete class providing concrete implementation either directly or indirectly (through other classes). For example, the abstract Agent class defines a template for all concrete agent types. Abstract Agent has Beliefs, Goals, Plans and concurrent threads. Therefore, each concrete agent type will inherit all these features and provide concrete implementation of these features.

4.1. Extensions to UML

The following provides (albeit an incomplete) list of abstract classifiers and abstract classes in our framework:

**Belief**: Belief is an abstract class which has a name for identifying Belief instances, a set of user-defined, application dependent annotations, and a list of goals that may be affected by changes to the belief. Examples of application dependent annotations are sampling frequency and probability of change of sensed values. Belief has a number of methods including but not limited to (in UML notation): eval(e:Expr):Boolean, which is an abstract method, to be implemented by concrete beliefs. It returns true if e is evaluated to true according to the state of the belief; otherwise, it returns false.

```
setV alue(key: String , value: String)
```

```
getV alue(key: String):String
```

which sets and gets component values of a belief.

```
addGoal(g: Goal)
```

and

```
removeGoal(g: Goal)
```

which adds g to and removes g from the affected goals.

```
gefGoals():Goals
```

returns the collection of affected goals.

**Goal**: Goal is an abstract class having a name for identifying the goal, a utility value and its set and get functions, and a plan to accomplish the goal. The utility value of a
goal indicates how valuable is the goal to the overall goal of the system. In addition, Goal has two abstract functions: beliefChanged(b:Belief) and eval():real. The former is automatically invoked when a belief is changed and the goal is affected. It allows the goal to respond to belief changes. The latter computes the utility value and returns a real value between 0 and 1 with 0 indicating unachievable goals. Its implementation is application dependent and can be a conventional decision tree, Computational Tree Logic (CTL) derivations as described in [22], or any other evaluation mechanism appropriate for the type of agent.

Plan: Plan is an abstract class defining a thread. It has an identifying name and an abstract execute() method which can be invoked by a Goal object to start a thread. A subclass of Plan must implement the execute method according to the concrete plan. The implementation may invoke KQML or FIPA performatives to communicate with other agents as well as conventional and knowledge based computations. A plan can be dynamically generated according to the reasoning steps. Plan also has a stop() method which can be invoked to terminate the plan.

Beliefs, Goals and Plans: These are collections of Belief, Goal and Plan objects and provide standard operations for querying, inserting, updating and deleting an element.

Agent: Agent is an abstract class for all concrete agent types. It has Beliefs, Goals, Plans and methods to select the optimal goal and also methods inherited from KQMLInterface. The goalChanged(g:Goal) method is automatically invoked when a goal instance is changed.

KQMLPerfomative: This is in fact a Command Pattern [8] introduced to accommodate all the KQML speech act performatives. Its subclasses are named after the performatives, one subclass for each performative. Each subclass implements the functionality of the performative.

KQMLInterface: This interface defines the methods corresponding to KQML performatives and is inherited by the abstract Agent, which will delegate the implementation of the performative to the appropriate KQMLPerfomative subclass. This way our framework can easily accommodate various ACLs and their extensions.

Blackboard: This is a concrete class to permit the use of shared blackboards. Agents can define polymorphic methods for reading, reading and removing, writing, appending to the blackboard (similar to Linda or LIME [19, 21]).

FIPA-ACL: These are standardized set of messages that contains a set of one or more message elements. Precisely which elements are needed for effective agent communication will vary according to the situation; with only the performative being mandatory. Java Agent DEvelopment framework (JADE) is one of the software frameworks implemented in JAVA. It can be used in the implementation of multi-agent system with performatives, which are very simple to handle. The setPerfomative(m:ACLmessage) and getPerfomative() functions are used to set and get FIPA-ACL messages.

These (except FIPA-ACL) are summarized in Figure 2.

4.2. Notations and Diagrams

In this section, we illustrate our modeling approach through the modeling of an intelligent elevator system (IES). In addition to the common features of an elevator system, IES must also optimize the service and minimize the total movements of all cars to reduce energy consumption and wear. Optimizing the service may be accomplished by minimizing the turn around time (or response time) to requests. In addition, distributed decision making among the elevator agents are assumed. In the example, we show only communication among the agent through a Linda like
Figure 2. A metamodel for the proposed modeling constructs
blackboard.

It is beyond the scope of this paper to describe a complete methodology, that is, the steps and guidelines, for MAS modeling and design. We simply adapt the Unified Process described in [18] and perform the following steps in each increment:

Step 1) Identify use cases and goals from requirements
Step 2) Refine use case diagrams and goals diagrams
Step 3) Refine system domain model and agent domain models
Step 4) Specify inter-agent and intra-agent sequence diagrams
Step 5) Refine design class diagram
Step 6) Refine other diagrams

We will follow steps 1) – 4) to illustrate the modeling concepts, notations and diagrams. We skip steps 5) and 6).

**Step 1) Identify use cases and goals from requirements.** From the elevator requirements and experience we identified the following use cases: Go to Floor, Request Elements. From the elevator requirements and experience we identified the following use cases: Go to Floor, Request Elevator, Open Door, Close Door. The gent goals are: Minimize Turn Around Time and Minimize Movement.

**Step 2) Refine use case diagrams and goals diagrams.** In this step, new use cases and goals are added and existing ones are revised. Therefore, in Step 2, we refine the use case diagrams and goal diagrams as well as the combined use case with goal diagram (UCGD). Our experience indicates that it is useful to relate the system use cases with the goals of the agents as shown in Figure 3. The UCGD shows which use case relates to which goal and vice versa. This information guides the construction of the inter-agent and intra-agent sequence diagrams, as we will show in Step 4).

In figure 3, the elevator system box shows the use cases and the actors. The actors are the users of the elevator. The diagram also contains a box representing the Elevator Car Agent indicated by the smiley face icon. The goals of the agent are shown as ovals with a curly paper icon. An association is drawn between the use case “Request Elev.” and the two goals signifying that when someone requests an elevator the two goals will be affected. The Minimize Turn Around Time goal attempts to reduce the response time by servicing the oldest requests as early as possible while the Minimize Movement goal tries to reduce the movement distances by servicing the closest requests. Note that these goals are sometimes conflicting and the goal’s utility value will determine which goal to be pursued, as we will illustrate in the intra-agent sequence diagram in step 4).

The use case diagrams are the same as conventional OO development and hence we will not repeat here. Goal diagrams are similar to use case diagrams. Part of Figure 3 shows a goal diagram. In addition to the goals, a goal diagram can also specify relationships among the goals using UML modeling constructs like inheritance, aggregation and association.

**Step 3. Refine system domain model and agent domain models.** In this step the system domain model [18] [12] — an ontologic or conceptual model [15] for the application domain objects, their attributes and relationships — is constructed or refined for the current increment. The system domain model for the elevator example would consist of objects representing various parts of an elevator and is familiar to most readers. Therefore we will not present it in the paper. We introduce the Agent Domain Model (ADM) to capture the application dependent beliefs, goals and plans of an agent and their properties and relationships. In our approach, an ADM is constructed for each type of application specific agent. Figure 4 shows an ADM for the elevator example.

The diagram indicates that the Elevator Car Agent has two beliefs: ElevCarState and Requests, which is a collection of Request objects. These beliefs are subclasses of the abstract Belief class. ElevCarState stores the floorNo, direction and load of the elevator car while Requests stores the outstanding requests and requests that the agent has scheduled to serve. Changes to the Requests belief will affect the two goals as shown in the diagram. Similarly, the two goals are subclasses of the abstract Goal class and hence must implement the beliefChanged(b:Belief) method. The diagram also indicates that the goals have plans and each plan delegates its task to a command object [8] that implements a thread.

**Step 4. Specify inter-agent and intra-agent sequence diagrams.** For each use case, at least one inter-agent sequence diagram is constructed to document how the agents and other objects work together to accomplish the business process underlying the use case. This is the same as in OO modeling [18] [2] except that agents may communicate with each other and with objects. The Agent Interaction Protocol (AIP) and notations for agent interaction proposed by O’Dell et al (see section 2) is used in our approach to model agent interactions. Again, to save space, we will only present an Intra-Agent Sequence Diagram (Intra-ASD), which depicts how the application dependent beliefs, goals and plans interact with each other to accomplish a use case and the goals of an agent. Figure 5 shows an intra-ASD for the elevator example.

As in Figure 5 the components, i.e., beliefs, goals and plans are encapsulated in a package, represented by the big folder icon. Note that within the folder icon, there is an instance of ElevatorCarAgent. This instance serves as both the container of the agent’s beliefs, goals and plans and the controller/coordinator for the agent. In Figure 5, the ElevatorCarAgent instance represents a role controller which coordinates the interactions among the beliefs, goals and plans of the agent. In our approach, there is often an agent role controller similar to the one in Figure 5.

Figure 5 also shows a Blackboard on the left. It is as-
Figure 3. Use case goal diagram for the elevator example

Figure 4. An Agent Domain Model for the elevator example
sumed that the ElevatorCarAgent instances (for a multi-car elevator system) communicate asynchronously through the Linda-like blackboard. It is assumed that when a floor button is pressed, the request is posted to the blackboard. The agents asynchronously read (represented by the dashed arrow line labeled “x=read()”) the blackboard to retrieve the requests and update the local belief “r:Requests”. It is assumed that the reading is triggered by some event like a timer going off or the elevator car approaching a floor. The beliefChanged(self) methods of the two affected goals are simultaneously invoked, where “self” is an OO “jargon” and refers to calling object, i.e., the collection of requests. Recall that beliefChanged(b:Belief) is an abstract method of Belief. Any subclass must implement this method. Any application engineer to implement application specific response to changes in the environment. As shown in Figure 5, the goals first request the Belief instance ElevCarState which stores the car’s current floorNo, direction and load. The goals then invoke their respective eval() methods to compute the utility values. The goals then fire the goalChanged(self) event to inform the agent that its goals have changed. The agent then gets the utility values from the goals and pursues with the higher utility goal. The selected goal then generates the plan and executes the plan which delegates its task to a thread (see Figure 4).

The abstract function eval() of Goal must be implemented by subclasses of Goal. For the elevator example, the following Java pseudo code illustrates how the two goals evaluate differently. It is assumed that an ArrayList r of size equals to the number of floors is used to store the requests. If there is no request from a floor then the status of the request is “null”.

class MinimizeTurnAroundTime extends Goal {
    // ...
    public double eval() {
        if (elevCar is going up)
            get requests above current floor
        if (elevCar is going down)
            get requests below current floor
        if (elevCar is idle)
            get all requests
        process the above requests satisfying oldest and same direction requests first
        double u=longest waiting time/threshold;
        if (u>1) return 1 else return u;
    }
}

class MinimizeMovement extends Goal {
    // ...
    public double eval() {
if (elevCar is going up)
satisfy requests above
  current floor in that order
if (elevCar is going down)
satisfy requests below
  current floor in that order
if (elevCar is idle)
satisfy requests in either
  side that require min movement
  per request
double u=longest waiting time/threshold;
if (u>1) return 0 else return 1-u;
}

5. Conclusions and Future Work

In this paper, we have proposed a framework and the necessary extensions to UML to address the need of modeling for MAS. We have introduced a new classifier, Agent and a set of abstract hhh Our approach draws from BDI formalism and introduces classifiers in UML to model the reactive and proactive behaviors of agents using beliefs, goals and plans. The agents in the MAS were considered to be peers in the decision making process. The communication between them is demonstrated by using a black board mechanism. Our approach allows the flexibility to use KQML or any other ACL(agent communication language).

Various diagrams are introduced, based on UML notations. The modeling process starts out with building a Use Case with Goals Diagram (UCGD) in which the associations between system use cases and goals is depicted. UCGD also includes the semantics of a UML Use Case Diagram. The next diagram is the Goal Diagram (GD) that illustrates the goals of a particular agent and depicts the relations among goals, sub-goals and subsystems of the MAS. We introduce the Agent Domain Model (ADM) which extends the UML class diagram to model belief, goals and plans. It incorporates interfaces and abstract classes to defer the binding of proactive and intelligent behaviors to a subclass. The sequence diagrams from UML are used to portray the interaction among system entities and agents for each use case. This is termed as Inter-Agent Sequence Diagram. To model the decision making process and interactions within a agent, we proposed Intra-Agent Sequence Diagrams.

We propose to explore modeling goal-directed (proactive and reactive) behavior by extending the state diagrams in UML. Collaboration diagrams and activity diagrams will also be investigated for suitability in modeling agent interactions and behaviors. We intend to demonstrate the usefulness of the above mentioned diagrams by using it to complete the modeling of Intelligent Elevator System (explained in 4.2. It would be an interesting exercise to implement the IES using the methodology discussed in this paper.

We plan to demonstrate the effectiveness and scalability of the proposed methodology by conducting realistic, domain specific case studies by systematically modeling and designing intelligent agents for projects such as WISE [27]. WISE is a National Science Foundation (NSF) funded project being investigated at the University of Texas at Arlington (UTA). WISE aims to model the Engineering Campus at UTA as a simulated environment to allow physically distributed agents to play an interactive game, cooperatively or competitively.

References


