Title: A Framework For Designing, Modeling and Analyzing Agent Based Software Systems

While there are numerous ambitious proposals aimed at developing software that is self-tuning, self-organizing and anticipates user needs, in this proposal we approach a very modest goal: to investigate practical software engineering methodologies for the design of intelligent and proactive agents. Over the past decade agent paradigm has been gaining popularity because agents bring intelligence, reasoning and autonomy to software systems. Agents are being used in an increasingly wide variety of applications from simple email 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 a 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 extensions address only the interactions among agents and do not facilitate the representation of reasoning and proactive nature of agents-oriented systems. In this research, we propose a framework and extensions to UML to address this need. Our approach is rooted in the BDI formalism, but stresses practical software design methods instead of reasoning theories. In the proposed project we will focus on the development of a prototype environment for specifying agent-oriented software systems. Our analysis tools will be domain specific – concentrating on real-time and embedded network of sensors during this project. We hope to demonstrate the usefulness of our agent-oriented software engineering by exploring “proactive” and intelligent extensions to well documented and understood problems from object-oriented software development arena, as well as new applications that cannot be easily described using object-oriented methodology. Our approach, when fully developed, can be customized to specific application domains by modifying domain specific aspects of our framework (specification level annotations, patterns, templates and reasoning mechanisms).

1. Introduction and Motivation

The advances in computing and communication and the availability of every inexpensive wireless devices has excited the imagination of Computer Science and Engineering researchers, leading to such ambitious projects as MIT's Oxygen which aims to make computing as available as Oxygen; and CMU's Aura that aims to design systems that address demanding human needs, including proactively anticipating and self-tuning available resources to meet user needs. Our goal is much more modest and focused to address one aspect of such future computing needs -- to provide practical software engineering methodologies for the design of intelligent proactive agents.

Over the past decade agent paradigm has been gaining popularity because agents bring intelligence, reasoning and autonomy to software systems. Agents are being used in an increasingly wide variety of applications from simple email 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 (see, [Wooldridge 00], [Tveit 01]) and extensions to UML (e.g., AUML [Bauer 00], [Odell 00]). AUML extensions, however address only the interactions among agents and do not facilitate the representation of reasoning and proactive nature of agents-oriented systems. In this research, we propose a framework and extensions to UML to address this need. Our approach is rooted in the BDI formalism, but stresses practical software design methods instead of reasoning theories. In the proposed project we will focus on the development of a prototype environment for specifying agent-oriented software systems. Our analysis tools will be domain specific – concentrating on real-time and embedded network of sensors during this project. We hope to demonstrate the usefulness of our agent-oriented software engineering by exploring "proactive" and intelligent extensions to well documented and understood problems from object-oriented software development arena, as well as new applications that cannot be easily described using object-oriented methodology. Our approach, when fully developed, can be customized to specific application domains by modifying domain specific aspects of our framework (specification level annotations, patterns, templates and reasoning mechanisms). Before we present our research, a brief overview of agents, objects, BDI and UML are presented

1.1. Agents versus Objects

Object-Oriented programming received a large following over the past two decades. There are several reasons for this popularity. Object-Oriented programming can be viewed as the next evolution in programming -- from a single sequence of code, to modularization using functions (or procedures), to abstract data types and encapsulation to inheritance. Object-Oriented programming also benefited from the availability of programming languages, compilers, modeling and analysis tools, and software engineering environments. *We contend that agent-oriented programming is the next expected evolution of object-oriented programming*. Agent-oriented programming draws heavily from object-oriented paradigms, but also introduces a number of new concepts that are alien to object-oriented programming. In order to appreciate the similarities and difference, we first describe object-oriented paradigm informally[1].

An object is an entity that encapsulates private state information or data and a set of associated operations or procedures that manipulate the data. An object's state can only be examined or modified by making an explicit request or by invoking an accessible operation. A class is very similar to an abstract data type; it is a template from which objects may be created. Every object is an instance of some class. A group of objects that have the same set of operations and the same state representations are considered to be of the same class. It should be remembered that a class maintains neither a state nor executes an operation – it is only an abstract entity used to define object instances in its mold. Object-oriented programming is distinguished (from object-based) by supporting the concept of inheritance, which permits new classes to be developed from existing classes simply by modifying or extending the operations and state information. A class may inherit the operations and behavior of a base class (or superclass) and it may have its operations and behavior inherited by a derived class or subclass. More than one class may be derived from a single base class. A superclass provides functionality that is common to all its subclasses while a subclass

---

[1] For more detailed and formal definitions the reader is referred to any of numerous publications (for example [Booch 94], [Meyer 90], [Rambaugh 91]).

provides additional functionality to specialize its behavior. Such associations between a superclass and its subclasses are often referred to as IS-A or IS-A-KIND-OF relationships [Chin 91].

Because of the simultaneous origin of "agents" in different disciplines, 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 [Jennings 98a] the authors define agents with three concepts: situatedness, autonomy and flexibility. The situatedness implies that agents receive input from an environment and perform actions that may change the environment. This however does not clearly distinguish agents from real-time control software. We would like add the ability of an agent to not react to sensory inputs depending on other properties of an agent (goals and beliefs as described later). Autonomy implies that the software system should operate without the direct intervention of human being 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. Proactivity indicates that agents not only react, but also exhibit goal-oriented behavior.

Object-oriented aficionados feel that the OO paradigm is expansive enough to include these characteristics of software agents. At the other end of the spectrum, devotees of agent-oriented paradigm claim that agents are radically different from objects. We, however believe that that agent-oriented paradigm subsumes object-oriented paradigm. Agents are autonomous and thus are inherently created with an autoprocess (or thread). Some object-based systems permit active objects, which are endowed with an active thread. The beliefs of an agent may be viewed as an object state. Other agents may directly manipulate an agent's beliefs. Using the concept of "friends", object-based systems may be designed to permit such direct manipulations. However, in a general agent-oriented system, it is possible to share beliefs among agents (via a knowledge-base, blackboard), which is contrary to the object encapsulation philosophy. Agents have desires (or goals), including proactive goals, requiring autonomous actions without any explicit method invocations. *The behavior of an agent (or the outcome of executing a method) may be different at different times (and may be non-deterministic)—since the proactive goals of an agent may lead to changes in its reactive behavior.* The different behaviors may result from the current beliefs and goals of an agent. An agent may not respond to method invocations by other agents (or objects). Such behaviors are not inherent in object-oriented systems. Agent-oriented systems do permit inheritance, but such inheritance must be distinguished from inheritance in object-oriented systems [Xu 01]. Agents may inherit plans (or actions, which are similar to methods), beliefs (which are similar to instance variables) or goals (for which there is no direct counter part in OO). The similarities and differences between the two paradigms will become clear from our framework.

1.2. BDI Formalism[2].

Given the above overview of agents, we will briefly summarize a related formalism that has been widely accepted in the AI community. 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 for describing these three components and reason about multi-agent systems. In practical terms, beliefs can be viewed as the state of the world (at least the state as viewed by agents, even if this information is inaccurate or outdated). 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 (including paths that become infeasible due to current beliefs). 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 Figure1.

In general beliefs may be shared and modified by other agents. This can be achieved either by direct communication (using KQML [Finn 97] messages), using shared knowledge-bases or blackboards (for example using Linda or extensions to Linda such as LIME [Murphy 01], [Picco 99]). 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.

[2] For a more detailed and formal treatment the reader is referred to any of the numerous publications on BDI (for example, [Rao 91, 92, 95]).

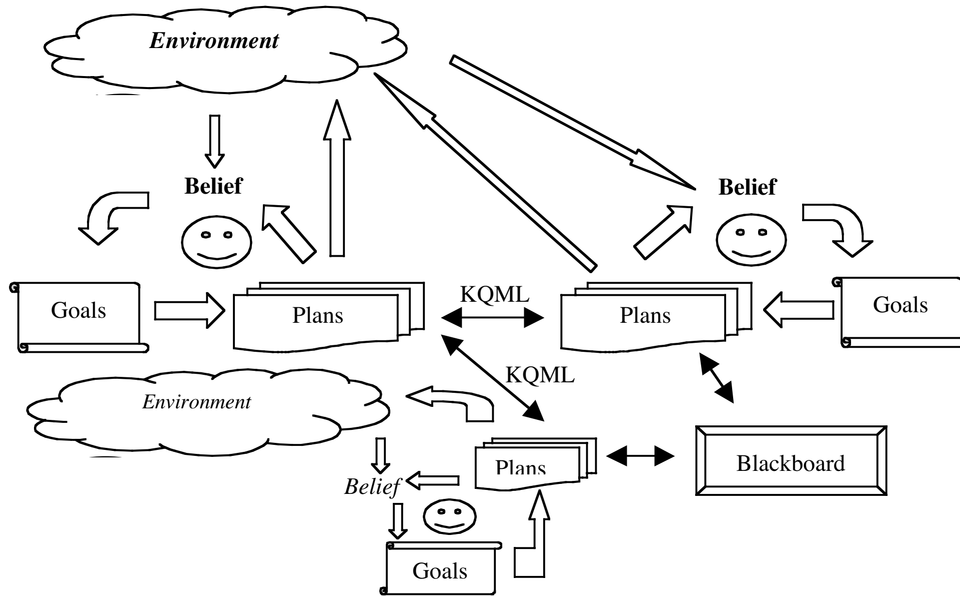

Figure 1. A conceptual model for a multi-agent system

1.3. Unified Modeling Language

The Unified Modeling Language (UML) [Booch 99] is widely used in the object-oriented (OO) paradigm 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. For example, pressing the gas paddle may cause a car to move forward, backward or remain still, depending on the state of the transmission (drive, reverse, or park). UML provides 8 different types of diagrams for modeling different aspects of a system.

1) Use Case Diagram models the "relates-to" relationship between an external entity (called an actor, which is usually a system user) and a process (called a use case), which depicts how a user would interact with the system to cause the latter to carry out a task for the former.
2) Class Diagram depicts the types of objects (called classes) that are of interest to the application at hand, properties (called attributes) that are common to a type of object, actions (called methods) an object can perform, and relationships to other types of objects.
3) Sequence Diagram is used to model the interactions among a set of objects to collectively realize a use case. In particular, a sequence diagram depicts the interactions as time-ordered sequences of invocations of methods of objects (called message passing). A semantically equivalent alternative is the Collaboration Diagram, which depicts the message flows between the objects. These diagrams are used in the design phase to help assign responsibilities (or functionalities) to objects and derive the shell for the methods [Larm01a].
4) Statechart Diagram is a hierarchy of concurrent, nested state machines originally proposed by David Harrel [Harr87a] for modeling real-time reactive systems. It is a significant improvement over the conventional flat state machine formalism [Meal55a] both in terms of much reduced complexity and enhanced expressiveness. It can be used to model state dependent behaviors of complex objects, which contain other objects as its components.
5) Activity Diagram is useful for modeling the information processing activities of a system or subsystem. It can model various types of activities including sequential, concurrent, synchronous, asynchronous, conditional and mutually exclusive activities. The “swim lane” mechanism of an activity diagram can be used to show which activities belong to which organizational unit.
6) Package Diagram is useful for depicting the software architecture or overall structure of the system in term of interrelated packages, each of which contains a set of classes and/or sub-packages. A package can be used to group classes of a subsystem like the graphical user interface subsystem (GUI), or classes that serve a special purpose like the utilities package.
7) Component Diagram is useful for depicting the executable software components and their relationships.
8) Deployment Diagram is useful for showing the allocation of components to hardware resources in the target environment.

1.4. Agent-Oriented Interaction Diagrams.

The Agent UML (AUML) approach proposed by [Odell 2000a] suggests extensions to UML to represent the interactions among agents. The proposal has been submitted to the UML standardization committee for consideration to be included in the forthcoming UML 2.0 The most important extensions include the extension to the sequence diagram to be able to model agent interactions, called Agent Interaction Protocols (AIP). An AIP describes a communication pattern as an allowed sequence of messages between agents and the constraints on the content of those messages. The major extensions to the sequence diagrams are notations for expressing concurrent threads of interactions. Instead of a straight arrow line from one agent to another, [Odell 2000] propose the notations in Figure 2 to express: a) (unconditional) concurrent interaction, b) ``inclusive or'' concurrent interaction, and c) ``exclusive or'' interaction.

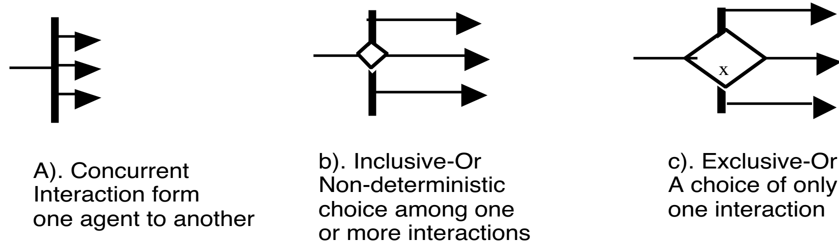

Figure 2. Concurrent Interactions Among Threads

Another extension to the sequence diagram utilizes the UML stereotype mechanism to enable agents to change their roles during a sequence of interactions. For example, a contractor agent in a B2B application may change its roles from being a contractor to a competitor analyzer. *We feel that these extensions are insufficient to model fully the agent-oriented paradigm. It is necessary to explore extensions to each of UML diagrams or introduce new diagrams to represent new concepts. That is a major goal of this research.*

2. A Framework For Agent-Oriented Software Engineering

In this section we will outline our approach to the specification, modeling, analysis and implementation of Agent-Oriented Software systems. First we will show the basic structure of an Agent Specification language. Then we will describe our extension to UML.

2.1. Agent Specification Language.

Utilizing the BDI framework, we propose the following language structure[3] to describe agent-oriented systems. We use an informal BNF like notation here.

```
Agent ::
        <agent-name> {
                [<Belief_Specification>] [<Goal_Specification>] [<Plan_Speceification>]
                protected <active-process>                /* see note 8 below
                }
        <Belief_Specification> ::
                { public        [<Beliefs_Structure>]*
                  private [<Beliefs_Structure>]*}
        <Goal_Specification> ::
                { public        [<Goals_Structure>]*
                 private        [<Goals_Structure>]*}
        <Plan_Specification> ::
                { public        [<Plans_Structure>]*
```

[3] There have been others who have either used the term agent-oriented programming (AOP) [Shaham 93], or defined languages for specifying agent behavior [Wagner 2000]. Our intention in the project is not define a completely new programming language, but to define a structure that can be used to extend existing languages such as Java. Our goal is to develop appropriate preprocessing tools so that the agent specifications can be converted into Java programs.

```
                    private             [<Plans_Structure>]*}

<Beliefs_Strucure> :: <Belief_Body> <Beliefs_Structure> | NULL
<Belief_Body>:: <Belief_Name> <Belief_Spec>                /* notes 1 and 2 below
                    [<$annotation_name>=<string_value>]*<Goals_Effected>          /* notes 3 and 4
<Belief_Spec>:: <Variable_Decl> | <Belief_Eval_Function>    /* note 2
<Goals_Structure> :: <Goal_Body> <Goals_Structure> | NULL
<Goal_Body> :: <Goal_Name> <Goal_Spec> <Goal_Value> <Plans_To_Execute> /* notes 5 and 6
<Goal_Spec> :: <Goal_Decl> <Goal_Eval_Function>

<Plans_Structure> :: <Plan_Body> <Plans_Structure> | NULL
<Plan_Body> :: <Plan_Name> <Invoke_Trigger> <Context_Preconditions><Sequence_Of_Statements>

<Sequence_Of_Statements> :: <Statement> <Sequence_Of_Statements> | NULL
<Statement> :: <Simple_Statement> | <Compund_Satement>                    /* note 7
```

Notes:

1. Belief_Name is for identification and indexing purposes. Same with Goal_Name and Plan_Name.
2. Belief_Spec is specific to a belief and defines the necessary data structure to store the values of observations. These structures together can be viewed as a description of the state of the world. This structure can be a simple variable, data structure, a database or a knowledge-base. The specification can provide a means of obtaining the value for the belief using the evaluation function (<Belief_Eval_Function>).
3. Annotations can be specified with beliefs for the purpose of analysis of the specification. The actual annotation name and value is domain specific. For example, for a real-time system, an annotation can be "Sampling_Frequency" and the specified value describes how often a sensor value should be sampled. Another example of annotation can be "Probability_of_Change" and the value indicates the likelihood of a change of a belief value. Such annotations can be used for feasibility or correctness analyses of the specification. The example annotations can be used for schedulability analyses in real-time systems. The annotations can also be used by agent run-time systems or middleware to schedule agent plans.
4. Goals_Effected links the goals that must be updated when the value of a belief changes.
5. Goal_Spec is specific to the goal and includes an evaluation function that can be used to update the value of the goal. Goal_Value is typically a real number indicating how valuable this goal is to the overall goals of an agent (or system). If a goal is not achievable, the value will be either zero or negative. It is necessary to select values to permit some selection and prioritization among goals. Goal Specification can be similar to a decision tree; based on the current state of the observed values, the decision tree is traversed to obtain a goal value. The path through the decision tree can also be viewed as the sequence of plans needed to achieve goal value. New goal valuations may lead to abortion of previously scheduled plans.
6. Plans_To_Extecute links to one or a sequence of plans that must be executed to achieve the goal (and its associated value). Both reactive and proactive goals can be defined using our syntax: reactive goals often receive higher goal values than proactive such that reactive goals will be executed in a timely manner The above syntax implies that goals are triggered when belief values change. The changes can be either due to external triggers, messages from other agents or changes to a knowledge-base.
7. Compound Statements for describing a plan can include any programming language statement. It is straightforward to include KQML [Finn 97] messages in our language structure, since a <Compound Statement> of a plan can be a KQML statement. KQML represents communication among agents. The communication is typically for one of the following purposes (we can map these to KQML performatives).

   Assert a Belief value. This could be either because of a previous request, a routine sensing of the state of the world, or other reasons. This message will have the same effect as if the agent is sampling the state of the world and setting its belief values; which in turn may require re-computation of goal values.

   Ask for a Belief value. Again an agent who does not have the responsibility of sampling for a particular state value can request another agent for the value. The reply will contain the value or the reply can take the form of "Assert a Belief Value" message.

   Ask for a service. An agent may execute a plan in response to this or ignore the request.
8. Autoprocess. An agent can be designed with one or more threads to meet the scheduling requirements If a single thread is used the structure of autoprocess may look like (using ADA like choice)

```
autoprocess() {
      do {
                  Select      (If_Reactive_Message_Queue_Not_Empty)
                                          Perform_Reactive_Plan
      or
                  Select (If_Belief_Sampling_Is_Due)
                                          Sample_External_State();
                                          Update_Goal_Values()
                                          Select_A_Plan_To_Achieve_Optimal_Goal_Value();
        or
                  Select
                              Proactive_Goal_Plan();
      }
```

If multiple threads can be utilized, we can associate separate threads to respond to belief changes, external stimuli, external service requests and proactive goals. Internal mechanisms are needed to assure proper coordination and synchronization of concurrent threads (e.g., using mutual exclusion).

2.2. Extensions to UML

As described in section 1.3, UML is adequate for modeling object-oriented (OO) systems. But UML lacks the capability to readily model and specify multi-agent systems. Unlike [Odell 2001a]'s Agent UML, we feel that every component of the UML must be extended. At the highest level, the OO diagrams can be contrasted from Agent diagrams as shown in Figure 3 – we will elaborate on each component of the agent diagram in coming sections.

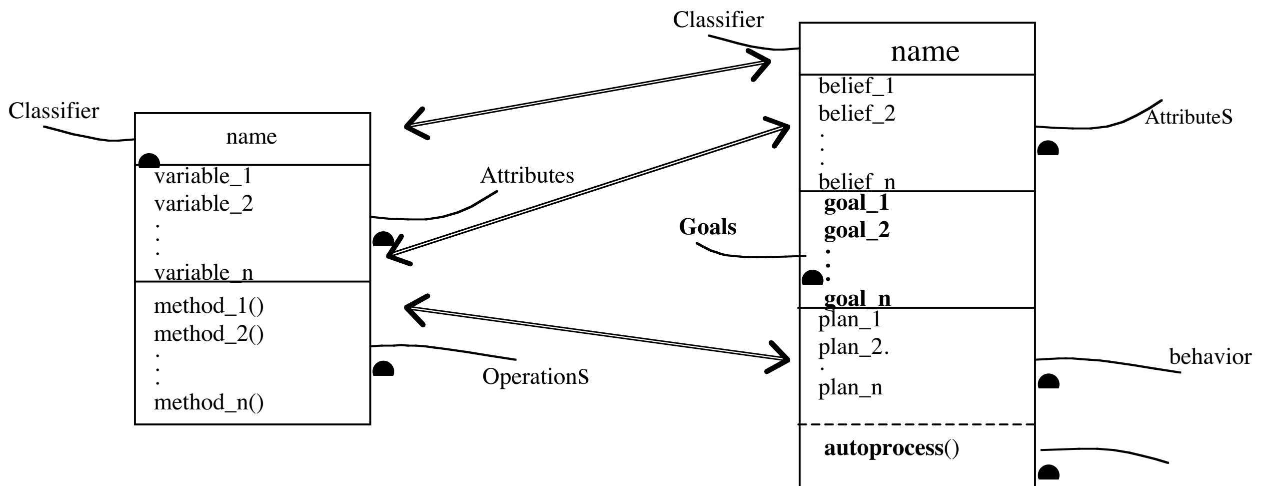

Figure 3. An Agent Diagram

Our approach is to extend UML through the introduction of a set of primitive types, which will be realized by interfaces and/or abstract classes. In the OO paradigm, an interface defines a set of operations without providing concrete implementation. 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 AbstractAgent class defines a template for all concrete agent types. AbstractAgent has Beliefs, Goals, Plans and concurrent threads. Therefore, each concrete agent type will inherit all these properties and provide concrete implementations according to its own behavior. The following provides (albeit an incomplete) list of such primitive types we propose to investigate in this project. More types will be added during the course of the project:

Belief. Belief is an abstract class, which has a name for identifying Belief instances, a set of (optional) user-defined, application dependent annotations, and a list of goals that will be affected by changes to the belief. Examples of application dependent annotations are frequency of sampling of sensors in real-time systems, probability of change of the sensed value. These may be accessed using (user-defined) keys like "sampling frequency" and "change

probability", respectively. Belief also has a number of methods including but not limited to the following (in UML notation):

eval (e:Expr):Boolean which is an abstract method, to be implemented by subclasses (i.e., concrete beliefs) of Belief. It returns true if e is evaluated to true according to the values of the beliefs; otherwise, it returns false.

setAnnotation (key:String, value:String) and getAnnotation (key:String) :String which sets and gets annotations, if any, respectively.

addGoal(g:Goal), removeGoal(g:Goal) which adds g to and removes g from the affected goals respectively. In addition, getGoals():Goals returns the affected goals when the belief changes.

Beliefs. Beliefs is a collection of Belief objects which can be accessed using Belief names. This class provides ordinary collection operations for querying, inserting, updating and deleting an element, that is, a Belief instance.

Goal. Goal is an abstract class having a name for identifying the goal, a priority or significance value indicating how valuable the goal is to the overall goals of the multi-agent system, an abstract function for evaluating the goal, priority set and get functions, and a plan to execute (see below). A non-positive priority value indicates that the goal is unachievable. In addition, Goal has a beliefChanged (b:Beliefs) function which is invoked when a belief is changed and the goal is affected. This function allows the goal to response to any belief changes. Subclasses of Goal must implement the beliefChanged and the evaluation functions. The implementation of the latter can be a conventional decision tree, Computational Tree Logic (CTL) derivations as described in [raoa91a], or any other evaluation mechanism appropriate for the type of agent.

Goals This is similar to Beliefs and is a collection of Goal objects.

Plan. Plan is an abstract class. It has an identifying name and an abstract execute() method which can be invoked by a Goal object to achieve the goal. A subclass of Plan must implement the execute method according to the concrete plan. The implementation my involve KQML communications with other agents as well as conventional and knowledge-based computations. Plan also has a stop() method which can be invoked to terminate the plan.

Plans. This is similar to Beliefs and Goals, and represents a collection of Plan objects.

KQMLMessage. This concrete class is introduced to accommodate all of the KQML speech act performatives. KQMLMessage has a performative name and a number of KQML parameters like content, language, sender and receiver, etc. The performative name and the parameters can be set using polymorphic constructors or the setPerformative(String performatice) and the set (String parameter, String value) methods.

SharedBlackBoard Functions. 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 [Murphy 01], [Picco 99])

AbstractAgent. AbstractAgent is an abstract class that provides a template for modeling and specifying all concrete agent types. AbstractAgent has Briefs, Goals, Plans and at least the following methods: performReactivePlan(), performProactivePlan() and performOptimalGoalPlan(). These methods may spawn concurrent threads to perform the tasks as indicated by the methods. The performReactivePlan() thread will have higher priority than the performProactivePlan() thread so that reactive goals will be accommodated in a timely manner. Multithreading for an agent can be achieved by the Command Pattern [gamm95a] as shown in Figure 5 (Beltway), where each of the performXXX() methods delegates its work to a command object which implements a thread with a certain priority. AbstractAgent also has methods to accommodate all the KQML performatives and access to shared blackboards. That is, the execution of a plan changes the beliefs, which trigger changes to the goals, resulting in continuation of the current thread or the execution of another plan(s). Note that through KQML communications and the use of blackboards, the behavior of an agent may affect the beliefs, goals and plan executions of other agents.

2.3. An Example.

In order to illustrate the use of our framework for the design of agent-oriented software systems, we describe a portion of a complex real-time system using our specification language (for a complete description of the example as well as other examples see [Aborizka 02]). Here we describe the use of a multi-agent based system for

controlling traffic flow along several major highways (or beltways). The agents are responsible for detecting traffic flows (using sensors to detect the number and rate of flow of cars) and controlling traffic signals as well as posting warning messages as well as suggested alternate routes on Variable Message Panels (VMP). Figure 4 shows a block diagram of the system. Here each agent is responsible for a section of a highway and they are identified as Beltway1a, Beltway 1b, etc., P1, P2, .. are VMPs

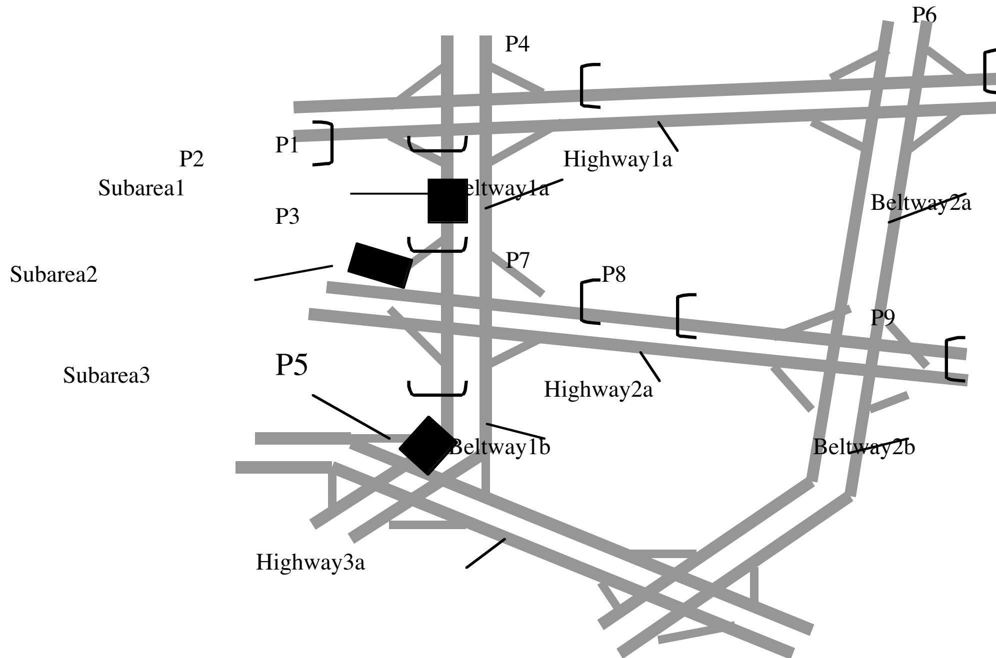

Figure 4. Traffic Control Example

Figure 5 shows the model for Beltway_1a agent in detail using our UML extensions. Here we outline the agent using our Agent Specification Language.

Beliefs
Public:
Belief_1 = (Belief_1, Sensor_1(Working), sampling_frequency =hour , probability_of_chage =0.001, Goal_2)
Belief_2 = (Belief_2, Circulation_Regime(Beltway_1b, "Congested"), sampling_frequency =hour, probability_of_chage=.0.5, Goal_1)
Belief_3 = (Belief_3, Saturation_Level(Beltway_1b, "Critical"), sampling_frequency =hour, probability_of_chage=.0.5, Goal_1)
Belief_4 = (Belief_4, Rush_Hour(YES, Lunch), sampling_frequency =hour, probability_of_chage=.0.2, Goal_3)
Private:
Belief_5 = (Belief_5, Traffic_Flow(Beltway_1a, 120), sampling_frequency =1 minute probability_of_chage=.0.5, Goal_3)
Belief_6 = (Belief_6, Sensor(Beltway_1, Sensor_1), NULL)
Belief_7 = (Belief_7, VMP_1(" Slow Traffic Ahead"), sampling_frequency =15 minutes, probability_of_chage=.0.5, NULL)
Belief_8 = (Belief_8, Traffic_Light_1(Green_Period = 3 min, Red_Period = 1 min, Yellow_Period = .05 min), Goal_3)

Goals
Public:
Goal_1 = (Goal_1, Smooth_Traffic (Beltway_1), 10, Identify_Problem, Diagnose, Configure_Singal_Plan)
Goal_2 = (Goal_2, Fix(Sensor_1), 20, (Inform_Maintenance, Write_To_VMP))
Private:
Goal_3 = (Goal_3, Traffic_Flow (Beltway_1a) < 80 cars/min, 15, Emergency, Rush_Hour, Congestion, Message_Consistency)

```
Plans
    Public:
        Plan_1 = (Inform_Maintenance {
                Invoke on Fix(Sensor_1)
                With Context        Sensor_1(Not Working)
                  Do
                        (tell
                                :sender             Beltway_1a
                                :language           prolog
                                :content            Sensor_1(NOT Working)
                                :receiver           Maintenance Person
                                :reply-with                 xyz )
                )

    Private
        Plan_2  = (Configure_Signal_Plan{
                Invoke on Traffic_Flow (Beltway_1a) < 80 cars/min
                With context …
                Do
                        (tell
                                :sender             Beltway_1a
                                :language           prolog
                                :content            Congestion at Subarea 3
                                :receiver           VMP_1
                                :reply-with                 abc)
                )

        Private
        Plan_3 = ( Rush_Hour{
                Invoke on Traffic_Flow (Beltway_1a) < 80 cars/min
                With context        Rush_Hour(YES, Lunch)
                Do
                        SET Traffic_Light_1(Green_Period = 2 min, Red_Period = 2 min, Yellow_Period = .05 min
                )

Autoprocess() {
        Do {
                Select (If_Reactive_Message_Queue_Not_Empty)
                Perform_Reactive_Plan
        Or
                Select(If_Belief_Sampling_Is_Due)
                Sample_External_State();
                Update_Goal_Values();
                Select_A_Plan_To_Achieve_Optimal_Goal_Value();
        Or
                Select
                        Proacative_Goal_Plan();
```

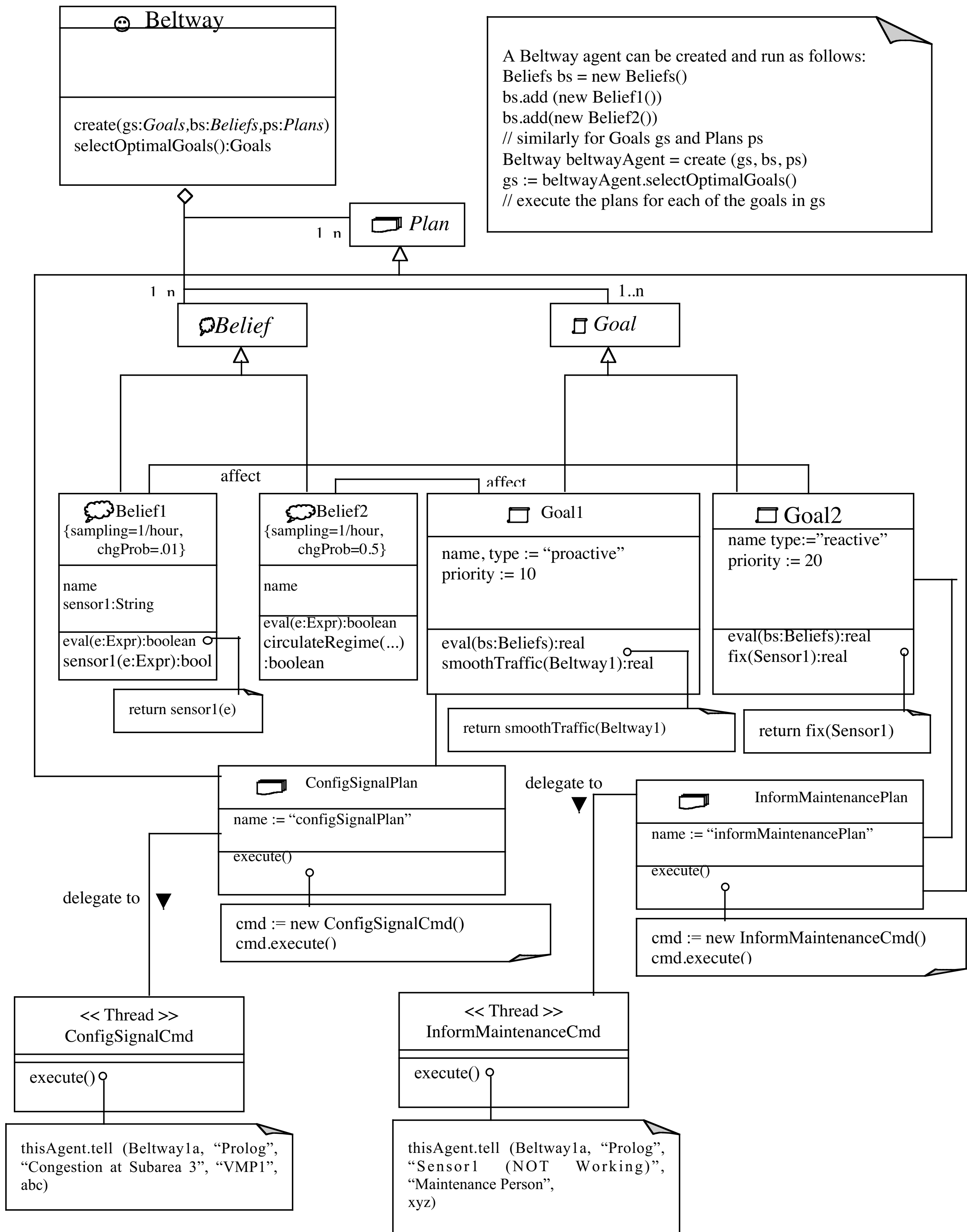

Figure 5. Extended UML For Example in Figure 4

3. Proposed Research

Agent-oriented systems are a natural extension of object-oriented systems. This implies that agent-oriented systems may contain agents as well as objects. Objects are the software models of real world objects which exist in the application domain. Thus, there is an urgent need to develop concepts and constructs, methods and principles for identifying, modeling and relating agents and objects. This need is similar to the need for unifying object-oriented analysis and design methodologies a decade ago (such as the Object Modeling Technique (OMT) [rumb91a], Booch's design method [booc86a] and Jacobson's USE Engineering Methodology [jaco92a]) for identifying, modeling and relating objects). In this project we propose to carry out the following research activities.

a). **Design and Implement** a framework for modeling, specifying, designing and analyzing multi-agent systems.
b). **Extend UML** to provide concepts and constructs for modeling multi-agent systems, particularly, concepts and constructs to support the BDI formalism from a practical software design view point instead of reasoning about agents.
c). **Develop a Systematic Methodology** for applying the framework and UML extensions to modeling, design, analysis, prototyping and construction of multi-agent systems.
d). **Investigate, Design and Implement** a software engineering environment for experimenting, validating and improving the methodology.
e). **Demonstrate** with case studies the usefulness of our approaches, and identify weaknesses and potential remedies.

3.1. Agent Specification Language

As outlined in section 2, we will develop our agent specification language primarily as an extension to an existing language such as Java. All Agent Specification Keywords will be preprocessed and replaced with Java programs (using predefined abstract classes, evaluation functions, KQML messages). The application specific annotations will be utilized for both analyses (e.g., schedulability) and for generating run-time schedules for agent threads. For the duration of this project we will restrict our attention to real-time intelligent systems. We will investigate the applicability of real-time analyses tools for schedulability. We will also investigate how various reasoning mechanisms can be incorporated within our analysis framework.

We feel that location of an agent can be represented as a belief, and mobility can be specified as a behavior resulting from goal-oriented nature of threads. In other words, an agent moves from location to location (or uses an itinerary) to achieve certain goals. The actual movement of an agent can be specified as a plan. During the proposed research we will explore these ideas. In addition to KQML message for interaction among agents, we will build interfaces to shared structures such as LIME [Murphy 01], [Picco 99].

3.2. Extensions to UML

As described earlier, UML is widely used to model object-oriented (OO) systems. But UML lacks the capability to model multi-agent systems. Qe propose to extend UML diagrams to permit the specification of a BDI agent. We base our approach on BDI because BDI has been widely used to design and implement agents. We propose to investigate the following extensions to UML.

1. Introducing Agent Use Case Diagrams as an addition to UML's existing System Use Case Diagrams. This is motivated by the observation that systems and agents are mutually containing; meaning that a system may contain agents whereas an agent may contain other systems. Thus, System Use Case Diagrams, which have been used to depict how external actors would interact with the system in the OO paradigm, are not adequate for multi-agent systems. This extension will blur the boundary between Actors and Systems: an agent is a system as well as an actor to another agent or system, a system can be an actor of an agent. Thus, the extension allows an agent to be modeled as an autonomous actor, which can initiate interactions at her will as well as systems which can react to stimuli and act according to her beliefs. Such an extension can also permit role change behavior of agents.
2. Extending the class diagram through the introduction of a set of agent specific, BDI-based classifiers for modeling the Belief, Desire and Intention of an agent. By incorporating interfaces, abstract classes and design patterns [gamm95a] from the OO paradigm, the extension can model any agent theory including dynamic creation of emergent behavior [petr2000a]. In the OO paradigm, an interface defines a set of operations without providing concrete implementations. An abstract class defines a type which must be subclassed by a concrete class that will provide concrete implementations for the operations of the abstract class. These features can be

used to defer the binding of proactive and intelligent behaviors to a subclass. The subclasses in turn can delegate the work to application specific implementations through application specific adapters. Please see section 3.4 software architecture for more details.)

3. In addition to Bauer and Odell's Agent Interaction Protocol (AIP) [baue2000a], we will extend the interaction diagrams (sequence diagram and collaboration diagram) to allow agents to act as external actors to other agents and modeling of interactions among the internal components (sensors, knowledge-based systems, decision trees, etc.) of an agent. We treat these components as building blocks implemented in application-dependent ``inner languages''. We restrict agents to communicate only with a common Agent Communication Language (ACL) [gene1994a] such as KQML [Finn 97] or FIPA [xxxx9xa]. Our study will use KQML but the research result will be equally applicable to other ACLs. We will also investigate agent interactions via blackboards.
4. Extending the state diagram to model goal-directed behavior in addition to reactive behavior currently provided by state diagram of UML. Proactive behavior also requires extensions to allow an agent to initiate interaction(s) with other agents. Again we restrict such communication to take place only through an ACL.

Figure 6 depicts possible extensions. We propose to investigate the feasibility and effectiveness of these and other extensions and their impact on modeling of multi-agent systems.

3.3. Agent Oriented Software Engineering Methodology

As discussed at the beginning of this section, there is an urgent need to investigate and develop methods and principles for identifying and modeling agents, relating an agent with other agents or objects, modeling the co-existence, interaction and collaboration among agents and between agents and objects. The existing Unified Process [jaco99a] is an architecture centric, use case driven, iterative, incremental process. It is effective for interactive systems where user-system interaction to accomplish a business task is the driving factor during the design process. Agent systems are reactive, proactive, concurrent, communicating and intelligent systems. It is not clear if the existing process still work for agent systems. We propose to:

1. Investigate the suitability of the existing UML based Unified Process for the design and analysis of agent based systems. In particular, we shall evaluate the existing process in terms of its effectiveness, adequacy, strengths and weaknesses with respect to agent systems modeling.
2. Invent an improved process and a new methodology for applying the framework and UML extensions to design, analysis and prototyping of multi-agent systems.
3. Conduct realistic, domain specific case studies, to systematically model, design and analyze intelligent agents that deal with real time embedded network of sensors and actuators. We shall demonstrate the effectiveness and scalability of the proposed methodology. The case studies will be described in greater detail in a later subsection.

3.4. Software Architecture

We propose to design and prototype a software engineering environment for validating our ideas. Figure 7 outlines the software architecture of such an environment. A multi-agent systems designer uses the Agent Unified Modeling Language (AUML) to conduct Agent Use Case Modeling, multi-agent systems Domain Modeling, multi-agent Interaction Modeling, multi-agent State Dependent Behavior Modeling and multi-agent systems Activity Modeling. We view multi-agent systems or agent-oriented systems as a natural extension of object-oriented systems; and hence, agent-oriented systems are a superset of object-oriented systems. This implies that agent-oriented systems may contain agents as well as objects.

Since modeling of agents is modeling at a high level, we need to investigate higher level modeling concepts and constructs and modeling methodologies. The methodology must not only address modeling of agents but also modeling of the co-existence and collaboration of agents and objects as well as various complex relationships among agents themselves and between agents and objects. Without such a methodology, AUML notions cannot aim to produce quality design for multi-agent systems. During this project we propose to develop a systematic, effective and efficient modeling approach, formulated in terms of phases and steps, and associated guidelines for modeling, designing, analyzing and constructing multi-agent systems.

An Agent Use Case Diagram Extends UML Use Case Diagram to allow modeling of agents as systems that can interact with other systems. Note a multi-agent system contains one or more agents.

Extensions to sequence diagram to show the interactions between agents and interactions within an agent.

Extending UML state diagram to model agent proactive behaviors. AgentB queries AgentA and waits in S1 until AgentA answers. AgentB then doAct() which cause device D to doX(). AgentB also sends a performative to AgentC.

Figure 6. Extension To UML

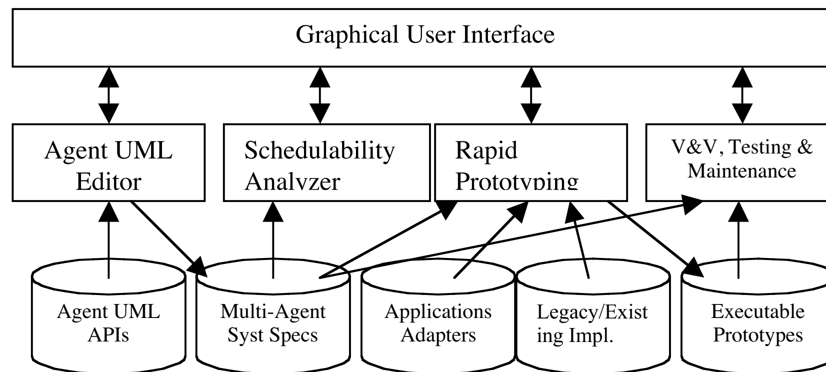

Figure 7. The Software Engineering Environment

3.5. Prototyping and Demonstration

We will apply the extended UML modeling concepts and constructs to multi-agent intelligent systems dealing with real time embedded network of sensors and actuators. We plan to cooperate with the UTA Computer Science and Engineering WISE project currently funded by NSF (NSF grants EIA-0086260 and EIA-0115885). The WISE project will provide wireless access to a network. One of the case study will be the modeling, design, analysis and prototyping intelligent agents that can direct robots to move in a building according to the moves of a chess game played by human players and intelligent agents.

A more aggressive project currently undertaking at UTA is the NSF ITR MavHome project. The MavHome Smart Home project is a multi-disciplinary research project at the University of Texas at Arlington focused on the creation of an intelligent home environment. The approach is to view the smart home as an intelligent agent that perceives its environment through the use of sensors, and can act upon the environment through the use of actuators. The home has certain overall goals, such as minimizing the cost of maintaining the home and maximizing the comfort of its inhabitants. In order to meet these goals, the house must be able to reason about and adapt to provided information. We believe that the software engineering framework and modeling capabilities we propose will effectively in describing the software for such a complex system.

In addition to the UTA project, we will also explore the use of our methodology for the SensorNet project underway at UNT. In this project an agent-based monitoring infrastructure is being developed to facilitate the data acquisition and event-correlation for a large number of widely distributed environmental sensors of different types. The sensors provide measurements such as ambient temperature, air-pollutants, UV-radiation, water/air quality, or pollen counts. The SensorNet project aims to investigate how to dynamically change the sensing frequencies based on detected events and how data mining techniques can be used to identify ecological events.

Controlled experiments, in terms of student team projects, will be designed to assess the impact of our software engineering approach to the modeling and design of intelligent multi-agent systems.

4. Broader Impact.

We feel that our approach (if successful and completely developed) can impact how software agents are programmed and deployed. At present designers utilize ad hoc techniques for designing and testing agent-oriented software systems. Our research will also have a direct impact on Software Engineering education both at the University of North Texas and the University of Texas at Arlington, both in terms of courses, design projects and MS/PhD research. In addition to dissemination of our research via publications, we will make available our tools and software environment upon request. We hope to seek out support from Rationale to incorporate our UML extensions, and approach OMB to evaluate our proposed extensions for inclusion in a future UML definition.

References

[Aborizka 02] M. Aborizka. An Architectural Framework for the Specification, Analysis and Design of Intelligent Real-Time Monitoring Agent Based Software Systems, *PhD Dissertation* (in Preparation), Dept of ECE, University of Alabama in Huntsville, Under K.M. Kavi's supervision.

[Bauer 00] B. Bauer, J. P. Muller and J. Odell. "Agent UML: A formalism for specifying multi-agent software systems", *First International Workshop on Agent Oriented Software Engineering,* Limerick, Ireland, June 2000, pp 91-104.

[Booch 94] G. Booch. *Object-Oriented Analysis and Design*, 2nd edition, Addison-Wesley, Reading, MA.

[Booch 99] G. Booch, J. Rambaugh and I. Jacobson. *The Unified Modeling Language User Guide*, Addison-Wesley: Reading, MA 1999

[Chin 91] R.S. Chin and S.T. Chanson. "Distributed object-based programming systems", *ACM Computing Surveys*, March 1991, pp 91-124.

[Finn 97] T. Finn, Y. Labrou and J.Mayfield. "KQML as an agent communication language", in *Software Agents*, edited by J. Bradshaw, MIT Press, Cambridge, 1977

[Gomaa 00] H. Gamaa. *Designing concurrent, distributed, and real-time applications with UML*, Addison-Wesley, New York, July 2000.

[Jennings 98a] N. R. Jennings, K. Sycara and M. Wooldridge. "A roadmap of agent research and development", in *Autonomous Agents and Multi-Agent Systems*, Kluwer Academic Publishers.

[Jennings 98b] N.R. Jennings and M. Wooldridge, editors. *Agent Technology: Foundations, Applications and Markets*, Springer-Verlag, 1998

[Kavi 02] K.M. Kavi, M. Aborizka and D.J. Chen. "A framework for the design, modelling and analysis of agent-oriented software systems", submitted to the *IEEE Transactions on Software Engineering*.

[Kendall 00] E. Kendell. "Agent Software Engineering with Role Modeling", *First International Workshop on Agent Oriented Software Engineering,* Limerick, Ireland, June 2000, pp 163-170.

[Kinny 96] D. Kinny and M. Georgeff. A methodology and modeling technique for systems of BDI agents", *Proceedings of the 7th European Workshop on Modeling Autonomous Agents in Multi-Agent World* (Springer-Verlag Lecture Notes 1038), Berlin, 1996, pp 56-71.

[Lind 00] J. Lind. "Issues in Agent-Oriented Software Engineering', *First International Workshop on Agent Oriented Software Engineering*, Limerick, Ireland, June 2000, pp 45-58.

[Meyer 87] B. Meyer. "Reusability: the case for object-oriented design", *IEEE Software*, March 1987, pp 50-64.

[Meyer 90] B. Meyer. *Object-oriented software construction*, 2nd edition, Prentice-Hall, Englewood Cliffs, N.J.

[Miles 00] S. Miles, M. Joy and M. Luck "Designing Agent-Oriented systems by analyzing agent interactions", *First International Workshop on Agent Oriented Software Engineering*, Limerick, Ireland, June 2000, pp 171-184.

[Murphy 01] A. Murphy, G. Picco and G.-C. Roman. "LIME" A middleware for physical and logical mobility", Proceeding of the 21st International Conference on Distributed Computing Systems (ICDCS), April, 2001, pp 524-533.

[Odell 00] J. Odell, H. Van Dyke Parunak and B. Bauer. "Representing Agent Interaction Protocols in UML", *First International Workshop on Agent Oriented Software Engineering*, Limerick, Ireland, June 2000, pp 121-140.

[Picco 99] G. Picco, A. Muphy and G.-C.Roman. "LIME: Linda meets mobility", *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[Rambaugh 91] J. Rambaugh, et. al. *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[Rao 91] A. Rao and M. Georgeff. Modeling rational agents within a BDI architecture. *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, Cambridge, MA, 1991, pp 473-484.

[Rao 92] A. Rao, M. Georgeff. An Abstract Architecture for Rational Agents. *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, Boston, MA, 1992, pp 439-449.

[Rao 95] A. S. Rao and M.P. Georgeff. "BDI agents; From theory to practice", *Proceedings of the first international conference on multi-agent systems (ICMAS-95),* San Francisco, pp 312-319.

[Shoham 93] Y. Shoham. "Agent-Oriented programming", *Artificial Intelligence,* (Vol 60), pp 51-92.

[Tveit 01] A. Tveit. "A survey of agent-oriented software engineering", First CS Graduate Students Conference, http://www.csgsc.org

[Wagner 00a] G. Wagner. "Agent-Oriented-Relationship Modeling", Proceedings of 2nd Internaitonal Symposium –From Agent Theory to Agent Implementations, in connection swith EMCRS 2000, April

[Wooldridge 97] M. Wooldridge. "Agent-based software engineering". *IEE Proceedings of Software Engineering*, Feb.1997, pp 26-37.

[Wooldridge 00] M. Wooldridge and P. Ciancarini. "Agent-oriented software engineering; The state of the art", *First International Workshop on Agent Oriented Software Engineering,* Limerick, Ireland, June 2000, pp 1-28

[Xu 01] H. Xu and S.M. Shatz. "A framework for modeling agent-oriented software", *Proceedings of the IEEE 21st International Conference on Distributed Computing Systems (ICDCS),* Phoenix, Arizona, April 2001, pp. 57-64.